3-Party Distributed ORAM from Oblivious Set Membership

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

[1] University of Pennsylvania, fbrett@cis.upenn.edu
[2] University of Pennsylvania, dgnoble@cis.upenn.edu
[3] UCLA, rafail@cs.ucla.edu

Abstract. Distributed Oblivious RAM (DORAM) protocols allow a group of participants to obliviously access a secret-shared array at a secret-shared index, and DORAM is the key tool for secure multiparty computation (MPC) in the RAM model.
In this work, we present a novel 3-party semi-honest DORAM protocol with $O((\kappa + D)\log N)$ communication per access, where N is the size of the memory, κ is a security parameter and D is the block size. Our protocol performs polylogarithmic computation and does not require homomorphic encryption. Under natural parameter choices, this is the most communication-efficient DORAM with these properties.
To build this DORAM protocol, we first present an extremely efficient oblivious data structure for answering set membership queries. From this we build an oblivious hash table with asymptotically optimal memory usage and access cost and with negligible failure probability. We believe these are of independent interest.

1 Introduction

Oblivious RAM (ORAM) [Ost90,GO96] provides a method for a trusted processor to execute a program that reads from and writes to an untrusted memory array such that the *access pattern* is independent of the (private) inputs to the program itself. Although traditional encryption algorithms can protect the *content* of the data, protecting data *access patterns* is critical for security.

The original application of ORAM was for *software protection*, where a tamper-resistant CPU had to maintain program security while making use of an untrusted external memory. With the rise of "secure enclaves" like Intel's SGX [AGJS13,HLP$^+$13,MAB$^+$13,CD16] and AMD's SEV [KPW16] that make use of untrusted system memory, this problem becomes more acute. Indeed, several types of "cache-attacks" (e.g. [TOS10,GBK11]) have shown that access pattern leakage can be used to extract secret key material, and that even trusted enclaves like SGX are vulnerable [BMD$^+$17,MIE17].

A similar application of ORAM arises in the setting of cloud storage, where a client wants to outsource its data storage needs to an (untrusted) cloud provider. Encryption can hide the *contents* of the data, but not the access pattern. This setting is similar to the setting of trusted CPU, but the data sizes are larger,

and the bandwidth is reduced. On the other hand, in the cloud-storage setting, it may be reasonable to assume the cloud provider is willing to perform some amount of computation in order to respond to a user's request, and the efficiency requirements may be relaxed somewhat compared to the CPU setting.

Most ORAM protocols aim to minimize the amount of communication between the client and the server, and the efficiency of an ORAM protocol is measured by the (multiplicative) communication increase incurred by executing the ORAM protocol. In other words, the *overhead* of an ORAM protocol is the communication cost of accessing t blocks (of size D) under the ORAM protocol divided by tD (as the number of database accesses, t, tends towards infinity). Sometimes, the asymptotic overhead depends on the relationship between D and other parameters. In this case it is often simpler to explicitly state the amortized communication cost, which we often refer to just as the communication cost, which is the cost of t accesses divided by t.

Early ORAM protocols were designed to allow a single, trusted processor to make a series of reads and writes to a single untrusted memory store. The application we target, however, is *secure multiparty computation* (MPC) [Yao82,Yao86,GMW87,BOGW88,CCD88].

The goal of MPC is to allow a group of data owners to securely compute a function of their joint data without revealing any information beyond the output of the computation. This notion of security requires that MPC protocols be *data oblivious*, in particular the running time, memory accesses and communication patterns of the participants cannot depend on other private data. To achieve data obliviousness, most MPC protocols work in the circuit-model. Circuits are inherently data-oblivious, and the MPC protocol securely computes the target circuit gate by gate. Although any computation can be expressed as a circuit, this representation may not be compact. Thus for efficiency reasons, it would be highly desirable to securely compute *RAM programs*. (A time-bounded RAM program of size $O(N)$ can be converted to a circuit of size $O\left(N^3 \log N\right)$ [CR73,PF79], but in most situations this efficiency loss is unacceptable.) Combining ORAM with a traditional, circuit-based MPC protocol provides a method for securely computing RAM programs, and can drastically increase the efficiency of certain types of secure computations.

This type of MPC-compatible ORAM protocol is called *Distributed ORAM* (DORAM). One of the challenges of building a DORAM protocol is that the there is no longer a trusted client who is allowed to learn the indices being queried. Note that this is different from multi-server ORAM protocols (e.g. [OS97,GKK$^+$12,LO13,GKW18,KM19]) which aim to increase efficiency by using multiple, non-colluding servers, but still require a trusted client. The efficiency of multi-server ORAM protocols can be increased further by allowing the servers to communicate with each other as well as the client [HYG20].

Any k-server DORAM protocol trivially yields a k-server active ORAM with $\mathcal{O}(\log N)$ client-server communication, since you can always add a fully-trusted,

lightweight, client whose queries consist of secret-sharing an index without compromising security.[4]

Moving in the other direction, any ORAM protocol that requires a trusted client can be converted into a DORAM protocol by emulating the client using an MPC protocol, and allowing the MPC participants to play the role of the untrusted server(s). Since MPC requires a significant computational overhead, simulating the ORAM client under MPC is only practical if the ORAM client uses minimal computational and storage resources. To this end protocols like "circuit ORAM" [WCS15] have been designed to minimize the *computational* complexity of the (trusted) ORAM client with the aim of making it simple enough to allow efficient execution by an MPC protocol.

Since DORAM protocols already require multiple servers, it is natural to create DORAM protocols by using MPC to emulate the trusted client in a multi-server or active ORAM protocol. This is a promising approach since several efficient multi-server ORAM protocols exist [OS97,GKK+12,LO13,GKW18,KM19] and some have even been implemented [GKK+12,WHC+14,Ds17,ZWR+16]. Unfortunately, most of these *multi-server* ORAM protocols are not suited for secure multiparty computation, because their clients often perform complex computations that cannot be efficiently executed inside an MPC.

The problem of building efficient DORAM protocols has been widely studied [FJKW15,ZWR+16,Ds17,JW18,BKKO20], but the most efficient existing protocols are significantly less efficient than single-server and multi-server ORAM protocols that require a trusted client.

In the single-server setting, the most efficient ORAM protocols[5] have $\mathcal{O}(D \log N)$ communication [AKL+20], and in the multi-server setting, $\mathcal{O}(D \log N)$ communication is achievable using simpler techniques [LO13].

In the DORAM setting, where there is no trusted client, the best practical protocols have at (at least) $\mathcal{O}\left(\log^3(N) + D\log(N)\right)$ communication ([WCS15] [FJKW15] [JW18] [BKKO20]). See Section 2 for a more detailed discussion of existing DORAM protocols.

Our main construction is a novel 3-party DORAM protocol that has $\mathcal{O}((\kappa + D)\log(N))$ communication and is extremely efficient in theory and in practice.

Theorem 1 ($(3,1)$-DORAM (informal)). *There exists a 3-server DORAM protocol with amortized communication complexity $\mathcal{O}((\kappa + D)\log(N))$ bits. The protocol provides security in the semi-honest model against one corruption.*

As noted above, in general, a k-server DORAM protocol does not immediately yield a k-server ORAM protocol. Our construction, however, can also be

[4] Note that a k-server DORAM protocol does not immediately yield a $(k-1)$-server active ORAM by allowing one of the DORAM servers to play role of the trusted client, because in active ORAM the client must use sublinear storage, and there is no such restriction for DORAM servers.

[5] If the server is allowed to be *active*, i.e., if the server can perform computation in order to craft its responses, then the communication can be reduced to $\mathcal{O}(D)$ [AKST14,DvDF+16,FNR+15,RFK+14].

used in the multi-server setting, since one of the servers only uses polylogarithmic memory (except while building the hash tables).

Our construction builds on the "hierarchical ORAM" solution of [GO96]. The hierarchical ORAM was further refined in the single-server setting [KLO12], and the multi-server setting [LO13,KM19].

The main source of inefficiency in the hierarchical ORAM solution, comes from having to (obliviously) shuffle different levels of the hierarchy, then (obliviously) index into these levels.

In the 2-server setting, [LO13] eliminated the use of oblivious shuffles, but their solution requires the client to encrypt *each record* as it's being passed from one server to the other. In the setting of multiparty computation, where the client is simulated by an MPC protocol, the cost of simulating the client dominates the rest of the protocol.

Nevertheless, the *asymptotic* complexity of the DORAM protocol which simulates the [LO13] client under MPC is actually quite good. Assuming executing a symmetric cipher on D bits with security parameter κ under MPC requires $\mathcal{O}(\kappa + D)$ communication, the DORAM protocol derived from [LO13] has $\mathcal{O}\left((\kappa + D)\log(N)\right)$ communication cost, *which is asymptotically better than all the subsequent DORAM protocols* [WCS15,FJKW15,ZWR+16,Ds17,JW18,BKKO20].

Concretely, our DORAM protocol reduces the number of calls to the underlying block cipher by a factor of 50 (see Supplementary Material A).

Our construction makes use of a novel set-membership data structure that has negligible failure probability, and only requires accessing $\mathcal{O}(\log N)$ *bits* per access.

Theorem 2 (Set-Membership Data structure). *The data structure outlined in Section 6 can store $n = \omega(\log(N))$ elements, from a universe of size N, with linear storage overhead ($\mathcal{O}(n\log(N))$-bits), negligible false-positive rate (in N), zero false-negative rate, negligible probability of build failure (in N) and logarithmic lookup cost ($\mathcal{O}(\log(N))$ bits).*

Note that these properties are *not* simultaneously satisfied by existing data structures like Cuckoo Hash Tables and Bloom Filters. Note that cuckoo hashing has a non-negligible probability of build failure, cuckoo-hashing with a stash has $\mathcal{O}\left(\log^2(N)\right)$ lookup cost, and Bloom Filters cannot simultaneously achieve logarithmic lookup cost and negligible false-positive rates. See Section 6 for a more detailed discussion.

2 Prior work

Oblivious RAM [Ost90,GO96] has been extensively studied (see e.g. [SCSL11], [KLO12],[WCS15],[SVDS+13],[AKL+20]) with a variety of different applications and efficiency metrics.

Early ORAM protocols had an overhead of $O\left(\log^3 N\right)$ [GO96], while later works improved this to $O(\log N)$ [SVDS+13,WCS15,AKL+20] which is known

to be optimal [GO96,LN18]. These early work considered a model with a single, trusted client, and a single, server that could *only store and retrieve data*.

Several other models have been developed in an attempt to improve performance. These include adding multiple (non-colluding) servers, or allowing the server(s) to perform computations in order to respond to user queries.

Multi-server ORAM: The communication complexity of ORAM protocols can be improved by allowing multiple (non-colluding) servers. Multi-server ORAM protocols include [OS97,GKK$^+$12,LO13,GKW18,CKN$^+$18,KM19]

[LO13] used two (non-colluding) servers, each holding alternate levels of the hierarchy. To shuffle a level, the server would pass the (encrypted) level to the client, the client would decrypt, then re-encrypt the level, and pass it to the other server. The other server would perform the shuffle (in the clear), pass the level back to the client, who would decrypt it, re-encrypt it, and pass the shuffled layer back to the original server. This process avoids the costly oblivious shuffles (since all shuffles are done in the clear), but requires the client to perform a linear number of (symmetric-key) encryptions. In the traditional ORAM setting, this is acceptable, since the client can perform symmetric-key encryption (e.g. AES) extremely efficiently. Unfortunately, when using the ORAM protocol within a secure computation, the client is *simulated* by the MPC protocol, and thus all the client operations (including the encryptions) need to be performed under MPC. Although encryptions can be performed under MPC, this cost dominates the cost of the overall protocol, and makes the entire scheme inefficient in practice.

[KM19] provides two fundamentally different multiparty ORAM solutions, a PIR solution and a hierarchical solution with different requirements and performance. The PIR solution was pioneered in [OS97], and the [KM19] solution builds on it, essentially replacing information-theoretic PIR with FSS-based [GI14,BGI15] PIR. The [KM19] PIR solution requires four servers, and works as follows. The servers are divided into two groups of two. For each group, the entire memory array is secret-shared among the two servers in each group. Thus the servers hold two copies of the memory array, and each copy is 2-out-of-2 secret shared. To read (or write) an entry into the array, the client uses a 2-server PIR protocol to read the corresponding location in *both* secret-shares. Finally, the client reconstructs the memory location by summing the shares contained in the two PIR responses. Using 2-server PIR based on Function Secret Sharing (FSS) [GI14,BGI15], the communication complexity for each query is $O(\kappa \log N + D)$. Since there is no hierarchical structure, there is no complicated reshuffling procedure. The main drawback of the PIR solution is that the servers need to do $O(N)$ *computation* to respond to the PIR query. Thus while the *communication* overhead is small, the *computational* overhead is larger (on the server side) than in most other protocols. The hierarchical solution of [KM19] builds on that of [KLO12,LO13], but requires 3+ servers. Recall that in the balanced hierarchical solution, the memory array is stored in levels of increasing size, and each level contains multiple hash tables. In previous works, the client would query *all* the hash tables at each level. Using efficient 2-server FSS-based PIR [GI14,BGI15], the client can execute a PIR query for each level to extract only the entries from

the correct hash table. Like the [LO13] work, the [KM19] solution avoids expensive oblivious shuffles having different levels of the hierarchy stored by different servers. In addition to [KM19] FSS-based PIR was also used in FLORAM [Ds17] and 3-party ORAM [BKKO20].

If the servers are allowed to *communicate with each other*, then they can further reduce the *client-server* communication [HYG20].

These multi-server ORAM protocols are *active*, i.e., although the servers do not communicate with each other, they don't just read and write data, they perform complex calculations in order to respond to user queries. Even in the single-server setting, giving the server power perform computation, can reduce client-server bandwidth.

Active ORAM: If the ORAM server(s) are allowed to *compute* on data, rather than just store and retrieve data, the $\mathcal{O}(\log N)$ lower-bound of [GO96,LN18] can be avoided, and several *active ORAM* protocols achieve *constant* client-server [AKST14,DvDF+16,FNR+15,RFK+14].

Despite their differences, all the multi-server and active ORAM protocols assume the client is trusted, an assumption we hope to avoid.

Distributed ORAM (DORAM): One important application of ORAM is in secure multiparty computation (MPC), where ORAM can be used to avoid the linear cost of oblivious memory accesses that is inherent in the circuit model.

Although multi-server ORAM and DORAM both involve multiple servers, the models are fairly different. Multi-server ORAM assumes a single, trusted client (who can perform operations locally), whereas in DORAM, there is no "client." Converting a multi-server ORAM protocol (or an active-server ORAM protocol) into a DORAM protocol would require simulating the client's actions using MPC, which can be extremely inefficient.

One approach to building a DORAM scheme is to take any ORAM protocol and simulate the client under MPC. The efficiency of this type of generic transformation is highly dependent on the complexity of the original ORAM client. Circuit ORAM [WCS15] is an ORAM protocol particularly amenable to secure multiparty computation, since its client is "circuit friendly." Instantiating Circuit ORAM with a generic MPC protocol (e.g. garbled circuits or BGW [BOGW88]) yields a DORAM scheme with $\mathcal{O}\left(\log^3 N + D \log N\right)$ communication. The $(3,1)$-DORAM protocol of [JW18] also builds on Circuit ORAM, and achieves communication complexity $O\left(\kappa \log^3(N) + D \log(N)\right)$, where D is the record size. Although the asymptotic *communication* complexity of the [JW18] protocol is κ times larger than that of a generic 3PC implementation of Circuit ORAM, the *round* complexity is improved from $\mathcal{O}\left(\log^2 N \log\log N\right)$ to $\mathcal{O}(\log N)$.

Since DORAM protocols already involve multiple servers, it is possible to convert a *multi-server* ORAM protocol to the DORAM setting by using MPC to simulate the ORAM client just as in the single-server setting.

In fact, instantiating [LO13] client using a generic MPC protocol yields a DORAM protocol with communication cost of $\mathcal{O}((\kappa + D)\log N)$, which is *better* than that achieved by subsequent DORAM protocols [FJKW15], [JW18],[WHC+14],[BKKO20].

This was observed in [FJKW15], but seems to have been largely ignored in later works (e.g. [JW18,BKKO20]).

Although instantiating the [LO13] client using generic MPC yields an *asymptotically* efficient DORAM protocol, it is *not* practically efficient. The main obstacle to efficiency is that the ORAM client in [LO13] performs a linear amount of symmetric-key encryptions when rebuilding a level. In the trusted-client model, these encryptions are extremely efficient (AES can be evaluated using hardware acceleration) but translating this protocol to the DORAM setting would require doing all of the encryptions *under MPC* which would drastically reduce the efficiency. In Appendix A, we calculate the concrete number of Shared-Input, Shared-Output PRP (SISO-PRP) calls, and show that our construction achieves the same asymptotics, but reduces the concrete number of SISO-PRF calls by a factor of 50.

An alternative approach to building DORAM was given in [BKKO20], where they build a $(3, 1)$-DORAM based on Function Secret-Sharing (FSS) [GI14,BGI15]. Although FSS-based protocols have bad asymptotics ([BKKO20] has $\mathcal{O}\left(\sqrt{N}\right)$ communication, and $\mathcal{O}(N)$ server-side computation), they are extremely efficient in practice (note that $\sqrt{N} < \log^3(N)$ for $N < 6 \cdot 10^8$). The FSS-based DORAM of [BKKO20] is also the only known DORAM protocol with *constant* round complexity.

Asymptotically, the best communication efficiency is achieved by instantiating the 2-server hierarchical ORAM of [LO13] using a generic MPC, but in practice schemes with suboptimal asymptotics, e.g. the BGW-instantiated Circuit ORAM with a cost of $\mathcal{O}\left(\log^3 N + D \log N\right)$ are superior. See Table 1.

Our work achieves amortized communication cost $\mathcal{O}\left((\kappa + D) \log N\right)$, but unlike the only existing DORAM protocol with this asymptotic cost, our protocol is actually *efficient in practice*.

Hierarchical ORAM: Our work builds on the "hierarchical ORAM," which was explored in the single-server setting [GO96,KLO12] and the multiserver setting [LO13,KM19].

In the hierarchical model, the server stores a hierarchy of hash tables. The top level is of logarithmic size, and each subsequent level is twice as large as the level above it. Every read query performs a linear scan over the top level, then makes one query into the hash table at each level. At certain specified intervals, the table at level i is emptied and the contents "shuffled" into the level below it.

The periodic reshuffles require *oblivious* hash table rebuilds, which is an obstacle to the overall efficiency of the protocol.

Implementations: Several works have implemented ORAM protocols. [GKK+12] implements the [SCSL11] ORAM using garbled circuits, SCORAM [WHC+14] uses the ObliVM [WLN+15] framework, [ZWR+16] implements the square-root ORAM solution in the Obliv-C [ZE15] framework, FLORAM [Ds17] uses the Obliv-C framework to implement function-secret-sharing-based ORAM.

Obstacles to efficient DORAM: The main obstacle to converting existing ORAM (or multiparty ORAM) solutions to the DORAM setting is the overhead of executing the ORAM client under MPC. For example, in the 2-server ORAM

GC Circuit ORAM [WCS15]	$\mathcal{O}\left(\kappa \log^3 N + \kappa D \log N\right)$
2PC Sqrt-ORAM [ZWR+16]	$\mathcal{O}\left(\kappa D \sqrt{N \log^3 N}\right)$
2PC FLORAM [Ds17]	$\mathcal{O}\left(\sqrt{\kappa D N \log N}\right)$
2PC ORAM [HV20]	$\mathcal{O}\left(\sqrt{\kappa D N \log N}\right)$
BGW Circuit ORAM [WCS15]	$\mathcal{O}\left(\log^3 N + D \log N\right)$
BGW 2-server hierarchical [LO13]	$\mathcal{O}\left((\kappa + D) \log N\right)$
3PC ORAM [FJKW15]	$\mathcal{O}\left(\kappa\sigma \log^3 N + \sigma D \log N\right)$
3PC ORAM [JW18]	$\mathcal{O}\left(\kappa \log^3 N + D \log N\right)$
3PC ORAM [BKKO20]	$\mathcal{O}\left(D\sqrt{N}\right)$
Our protocol	$\mathcal{O}\left((\kappa + D) \log N\right)$

Table 1: Communication 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. Although instantiating the [LO13] 2-party ORAM using generic MPC has the same asymptotic complexity as our protocol, concretely, we reduce the number of SISO-PRP calls by a factor of more than 50 (Appendix A).

protocol of [LO13], the reshuffling stage requires the client to encrypt (and decrypt) every entry being reshuffled. In [KM19]'s PIR solution, the FSS scheme requires the client to perform $O(\log N)$ encryptions for each query, and the hierarchical scheme of [KM19] requires a linear number of encryption operations for each reshuffle (just like [LO13]).

When the client is being simulated by an MPC protocol, all these encryptions must be performed under the MPC, and even using the most "MPC-friendly" block-ciphers (e.g. [ARS+15,DEG+18]) the overhead of the encryptions dominates the overhead of the rest of the ORAM protocol.

The main contribution of our paper is to provide an efficient Distributed ORAM protocol, which can be dropped in to existing MPC frameworks to simulate RAM in secure multiparty computations.

3 Preliminaries

Let P_1, P_2 and P_3 be the three parties in the protocol. For a positive integer B, we use single square brackets, $[B]$, to represent the set $\{1, \ldots, B\}$.

We assume the parties have access to an Arithmetic Black Box (ABB) functionality $\mathcal{F}_{ABB}$ (Figure 1). This is a reactive functionality that provides input, retention and output of secret-shared data. Additionally, it provides basic arithmetic operations, as well as some more advanced operations which we explain further below.

We borrow standard secret-sharing notation (e.g. from [GRR+16]) to represent variables stored in the ABB. Each variable in the ABB has a public identifier. We use $[\![x]\!]$ to denote the identifier for a value x that is stored in the

ABB. In general, methods in the ABB functionality require all parties to call them. However, the Input method is called by a single party, and the Reshare method is called only by the qualified set.

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

Input(x, pId, varName): Receive x from party pId and store it as $[\![varName]\!]$.
InputTo2Sharing(x, pId, varName, sharing): Receive x from party pId and share it as $[\![varName]\!]$ accessible by qualified set $sharing$, where $|sharing| = 2$.
ReshareTo2Sharing($[\![x]\!]$, sharing, varName): Share x as $[\![varName]\!]$, accessible by qualified set $sharing$, where $|sharing| = 2$.
ReshareFrom2Sharing($[\![x]\!]_{sharing}$, sharing, varName): Receive shares of x from the qualified set $sharing$, where $|sharing| = 2$ and store it as $[\![varName]\!]$.
AND($[\![x]\!]$, $[\![y]\!]$, outName): Compute $z = x \wedge y$ and store z in $[\![outName]\!]$.
OR($[\![x]\!]$, $[\![y]\!]$, outName): Compute $z = x \vee y$ and store z in $[\![outName]\!]$.
Equal($[\![x]\!]$, $[\![y]\!]$, outName): If $x \stackrel{?}{=} y$ set z to $true$, otherwise to $false$. Store z in $[\![outName]\!]$.
IfThenElse($[\![b]\!]$, $[\![x]\!]$, $[\![y]\!]$, outName): If $b \stackrel{?}{=} true$, set z to x, otherwise to y. Store z in $[\![outName]\!]$.
PRP_keygen(N, keyName): Generate a new random key, k, for a PRP $f_k : [N] \to [N]$. Store k in $[\![keyName]\!]$.
PRP_eval($[\![x]\!]$, $[\![k]\!]$, outName): Compute $z = f_k(x)$ and store z in $[\![outName]\!]$.
Shuffle($[\![X]\!]$, outName): Given an array $[\![X]\!] = [\![X_1]\!], \ldots, [\![X_n]\!]$, generate a random permutation of the contents of $[\![X]\!]$ and store the result in $[\![outName]\!]$. If multiple arrays are given as inputs/outputs, they are all shuffled using the same permutation.
Output($[\![z]\!]$, pIds, localVarName): Send z to every party in set $pIds$, who store it in the local variable $localVarName$.

Fig. 1: Arithmetic Black Box functionality

For notational convenience, we use normal assignment notation to show a new variable being stored in the ABB and drop its variable name from the function declaration. For instance, $\mathcal{F}_{ABB}.\text{AND}([\![x]\!], [\![y]\!], z)$ will alternatively be written as $[\![z]\!] = \mathcal{F}_{ABB}.\text{AND}([\![x]\!], [\![y]\!])$. Assignment notation is similarly used to show a new local variable being assigned based on a call to Output. We also use the notation $[\![z]\!] = [\![x]\!]$ to show that the value of $[\![x]\!]$, stored in the ABB, was copied to a new location, $[\![z]\!]$. Lastly, we occasionally place constants in the secret-sharing notation (e.g. $[\![true]\!]$). In this case, the constant is implicitly first input as a secret variable from any party.

The ABB abstracts away the details of the secret-sharing implementation and underlying MPC framework. Our efficiency results are based on instantiating

$\mathcal{F}_{ABB}$ with the generic MPC protocol of Araki et al [AFL+16]. This is designed for the semi-honest $(3, 1)$ setting and uses a $(3, 2)$-secret sharing.[6]

In addition to basic arithmetic and IO functionalities, our ABB also has the ability to share and reshare to and from a $(2, 2)$ sharing. Although we define methods for converting between $(3, 2)$ and $(2, 2)$ sharing, our protocol never *computes* on $(2, 2)$ shares, and thus functionality does not define any calculations that occur on $(2, 2)$-shared data. The resharing protocols are described in Appendix C.

One way to implement the functionality $\mathcal{F}_{ABB}$.Shuffle is to use the 3-party shuffle of [LWZ11]. This shuffle has concrete communication cost of $24nD$ bits to shuffle n bits, each of size D (See Appendix D).

Functionality	**Communication (bits, total)**	**Source**
$\mathcal{F}_{ABB}$.Input	4 (per input bit)	[AFL+16] §2.1
$\mathcal{F}_{ABB}$.InputTo2Sharing	2 (per input bit)	Appendix C
$\mathcal{F}_{ABB}$.ReshareTo2Sharing	0	Appendix C
$\mathcal{F}_{ABB}$.ReshareFrom2Sharing	8 (per input bit)	Appendix C
$\mathcal{F}_{ABB}$.XOR	0	[AFL+16] §2.1
$\mathcal{F}_{ABB}$.AND	3	[AFL+16] §2.1
$\mathcal{F}_{ABB}$.OR	3	Using 1 AND and NOTs
$\mathcal{F}_{ABB}$.Equal (n bits)	3(n-1)	Standard, e.g. [Ang]
$\mathcal{F}_{ABB}$.IfThenElse (n bits)	3n	Mux $z = b \wedge (x \oplus y) \oplus y$
$\mathcal{F}_{ABB}$.PRP_keygen	0	[AFL+16] §3.4
$\mathcal{F}_{ABB}$.PRP_eval (D bits)	$21(\kappa + D)$	[ARS+15], Ours Appendix B
$\mathcal{F}_{ABB}$.Shuffle (n each D bits)	$24nD$	Appendix D
$\mathcal{F}_{ABB}$.Output	3	[AFL+16], send x_i to P_{i+1}

Table 2: Communication costs of $\mathcal{F}_{ABB}$
Costs ignore a setup phase in which each pair of parties pick a PRG key for generating identical randomness. This set-up phase occurs only once for $\mathcal{F}_{ABB}$ and requires 3κ bits of communication.

We will be looking at an ORAM of size N, *i.e.*, it represents a RAM of size N. We assume N is a power of 2, so that indices are representable by $\log(N)$ bits where all logs are base-2. We will seek to achieve statistical failure that is negligible in N. Each data block will be of size D, where $D = \Omega(\log(N))$.

Our protocol achieves κ-bit security, by which we mean the protocol achieves symmetric security roughly equivalent to AES-κ. We assume a PRP exists with κ-bit security that can be represented as a circuit with $\mathcal{O}(\kappa)$ AND gates. We

[6] Our protocol could also be executed using garbled circuits. This would increase the communication cost by a factor of κ, since 2 ciphertexts would need to be sent per AND gate [ZRE15]. However the round complexity would be reduced to linear in the "openings" depth of the protocol, rather than the AND depth of the circuit.

suggest instantiating this with LowMCv3 which requires about 7κ AND gates to achieve κ-bit security (see section 4 and Appendix B). Throughout this work, as is standard, we assume $\kappa = \omega(\log(N))$.

4 SISO-PRPs

A pseudo-random function (PRF) is a keyed deterministic function such that the output appears random to any polynomially bounded adversary.

A *Shared-Input, Shared-Output PRF* (SISO-PRF) is a multiparty protocol to securely evaluate a PRF when the input, outputs and keys are secret-shared between the participants. Note that this is slightly different from the notion of an *Oblivious Pseudo-Random Function* (OPRF) [FIPR05]. Most OPRF protocols have focused on a 2-party evaluation of a PRF, where one party holds a key, k, and the other holds an input, x, and the output, $F_k(x)$ is delivered to one party.

In our applications, however, it is critical that the inputs to the PRF are secret-shared, thus most existing OPRF protocols are not applicable. In principle, it is possible to evaluate any PRF with secret-shared keys, inputs and outputs, using generic MPC protocols, but this is often fairly inefficient.

In this work, we will focus on *Shared-Input Shared-Output PRPs* (SISO-PRPs) where the pseudorandom function is actually a permutation.

Concretely, we imagine implementing our SISO-PRP using the "MPC-friendly" LowMC block cipher, which is highly optimized for evaluation as a SISO-PRP [ARS+15]. In addition to being MPC-friendly, LowMC has two additional features that make it useful in our setting. (1) LowMC has configurable block sizes, allowing us to reduce the communication and computational costs when the index space is small, and (2) when the maximum number of queries to the PRP is bounded (as is the case in our construction), LowMC can be instantiated with more aggressive parameters, increasing efficiency.

In Table 3, we compare the efficiency of LowMC, vs AES for 128-bit security. We present various parameter choices for LowMC using the LowMCv3 security estimator.[7] "Data" represents the log of the number of PRP evaluations the adversary will ever learn.

5 Construction Overview

Our main construction is a $(3, 1)$-DORAM protocol with amortized $\mathcal{O}\left((\kappa + D)\log N\right)$ communication cost.

Our construction builds on the "hierarchical solution" which is essentially a tool for converting an oblivious hash table that only provides obliviousness on *distinct* queries into an oblivious data structure that provides obliviousness on *repeated* queries. We describe the hierarchical solution in more detail in Section 9.

Informally, an oblivious hash table is a data structure that provides obliviousness on distinct queries, thus the hierarchical solution reduces the problem

[7] https://github.com/LowMC/lowmc/blob/master/determine_rounds.py

Cipher	Blocksize	Data	rounds	AND gates
AES	128	128	40	5120
LowMC	128	128	19	1824
LowMC	128	128	252	861
LowMC	10	10	32	288
LowMC	10	10	94	282

Table 3: Block cipher costs for 128-bit Security (AES-128 from [ARS+15][Table 2], LowMC from LowMCv3 security estimator)

of designing a DORAM protocol to building an oblivious hash table that can be *built* and *queried* efficiently in a distributed manner.

Starting with [PR10], Cuckoo Hash Tables have been widely used in ORAM protocols [GMOT12,KLO12,LO13,PPRY18,KM19,AKL+20]. The key property of Cuckoo Hash Tables is that in a series of *distinct* queries, the physical access pattern is independent of the underlying queries.

However, it is tricky to correctly incorporate Cuckoo Hash Tables into a hierarchical ORAM protocol. Cuckoo Hash Tables have a non-trivial probability of build failure (and a failure would leak information). Thus Cuckoo Hash Tables are instantiated with a "stash" of size $\omega(1)$ to hold elements that cannot be stored in the main Cuckoo Hash Table [KMW09]. In the hierarchical solution, adding a separate stash at every level of the hierarchy increases the asymptotic query complexity, thus most hierarchical ORAM protocols sought to combine the stashes across different levels of the hierarchy. This breaks the abstraction of each level of the hierarchy being an Oblivious Hash Table and was often done in a way that led to flaws in the ORAM protocol [FNO21]. In short, while Cuckoo Hash Tables with combined stashes can be used to implement hierarchical ORAM protocols efficiently, this makes the protocol and its analysis undesirably complicated.

Rather than instantiating the hierarchical solution with Cuckoo Hash Tables, we design a novel Oblivious Hash Table that requires $\Theta(\kappa + D)$ bits of communication per access (amortized). Our starting point is the observation [MZ14] that once it is known whether an element is stored in a table, Distributed, Oblivious Hash Tables can be constructed using any non-oblivious, but secret-shared, hash table structure by searching for distinct pre-inserted dummy elements when an element is not in the set.

This essentially reduces the problem to that of designing an efficient data structure for *set membership*. We do this by building a Cuckoo Hash Table with a stash, but instantiating the stash with a Bloom Filter. Surprisingly, this simple combination increases the asymptotic efficiency of the data structure beyond what can be achieved by Cuckoo Hash Tables or Bloom Filters alone.

The main technical challenge is then to construct the Oblivious Set Membership structure and Oblivious Hash Tables in the distributed setting efficiently, and *without leaking any sensitive data.*

We solve this in the $(3, 1)$-security setting by using a shared-input PRP, where one party learns the outputs in the clear during builds and the other two parties to learn the PRP outputs during accesses. We show that a single party can construct the Bloom Filter and Cuckoo Table objects in the clear based on the PRP evaluations alone, but without any shares of the actual indexes or data. The Bloom Filter and Cuckoo Table data structures can then be secret-shared between the remaining two parties who can then evaluate lookups without revealing any PRP outputs to the third party.

We present our Oblivious Set Membership protocol in Section 7 followed by our full Oblivious Hash Table protocol in Section 8. Finally, we use our novel Oblivious Hash Table protocol together with the hierarchical solution to construct a $(3, 1)$-DORAM protocol (Section 9).

Our Supplementary Material contains some useful standard material namely the definitions of hashing (Section E), oblivious hashing (Section F), Bloom Filters (Section G) and Cuckoo Hash Tables (Section H).

6 Set membership

Let there be some set of n elements from a universe of size N, each represented by $\log(N) \geq \log(n)$ bits. In this section we outline a novel data structure that supports *set membership queries* that simultaneously achieves the following properties:

1. Linear storage overhead ($\mathcal{O}(n \log(N))$)
2. Negligible false-positive rate in N
3. Zero false-negative rate
4. Negligible probability of build failure in N
5. Logarithmic lookup cost ($\mathcal{O}(\log(N))$)

Bloom filters and Cuckoo hash tables (supplementary material sections G and H) are widely used data structures that provide efficient storage and retrieval, but they do not satisfy all of the above design criteria simultaneously.

Example 1 (Cuckoo Hashing). Standard Cuckoo Hash Tables have linear storage overhead, zero false-positive rate, and logarithmic[8] lookup cost. Unfortunately, Cuckoo Hash Tables (without a stash) have a non-negligible probability of build failure.

Example 2 (Cuckoo Hashing with a stash). Modifying a standard Cuckoo Hash Table to include a "stash" of size $s = \Theta(\log N)$, for any $n = \omega(\log(N))$ makes the failure probability negligible in N [Nob21]. Unfortunately, every lookup query scans the entire stash, which requires reading s locations, which means lookups require accessing $\Theta(\log^2(N))$ bits of memory.

[8] Note that lookups require looking in a constant number of locations, but each location stores an identifier which must be at least $\log(n)$-bits, so the total lookup cost requires transmitting (at least) a logarithmic number of bits. Even Cuckoo filters [FAKM14] requires storing keys that are at least $\log(n)$-bits.

Example 3 (Bloom filters). The false-positive rate for a Bloom filter of size m storing n elements (using k hash functions) is about $\left(1 - e^{-\frac{kn}{m}}\right)^k$. A standard analysis (e.g. [MU17][Chapter 5]) shows that the false-positive rate is minimized when $k = \log(2) \cdot (m/n)$, which makes the false-positive probability approximately $(\log 2)^{-m/n}$. Thus to make the false-positive probability negligible in N, we need $m = \omega(n \log N)$, which means that the storage overhead is super-linear.

Although Cuckoo Hashing, and Bloom filters alone cannot achieve our five goals (linear storage overhead, negligible false-positive rate, zero false-negative rate, negligible probability of build failure and logarithmic lookup cost), *combining* the Cuckoo Hashing with Bloom filters allows us to simultaneously achieve all these goals. This is achieved simply by creating a Cuckoo Hash table with a stash, but storing the stash in a Bloom filter.

Build Given a set $X_1, \ldots, X_n$

1. Create a Cuckoo Hash table with a stash as follows:
   (a) Pick 2 hash functions h_1, h_2 which map $[N] \to [m]$ for $m = \epsilon n$ for some constant $\epsilon > 1$.
   (b) Create 2 empty tables, T_1 and T_2, each of size m.
   (c) Try to store each X_i in either $T_1[h_1(X_i)]$ or $T_2[h_2(X_i)]$. Find a maximal allocation (e.g., through a matching algorithm). Let S be the set of elements that were not able to be stored in either T_1 or T_2. If $|S| > \log(N)$, the build fails.
2. Store the stash in a Bloom Filter as follows:
   (a) Create an array, B of length $n \log(N)$ of all zeros.
   (b) Pick $k = \log(N)$ hash functions, $g_1, \ldots, g_k$, which map $[N] \to [n \log(N)]$.
   (c) For each element $x \in S$, and for $1 \leq i \leq k$ set $B[g_i(x)] = 1$.

Query Given an index x

1. Check if x is stored in the Cuckoo hash table by checking locations $T_1[h_1(x)] = x$ or $T_2[h_2(x)] = x$. If so, return true.
2. Check if x is stored in the Bloom filter, by checking whether $B[g_i(x)] = 1$ for all $1 \leq i \leq k$. If so, return true. Otherwise return false.

Fig. 2: Set Membership

Theorem 3. *When $n = \omega(\log(N))$, the Set Membership protocol of Figure 2 provides a data structure with linear storage overhead, negligible false-positive rate (in N), zero false-negative rate, negligible probability of failure (in N) and logarithmic lookup cost (in bits).*

The proof of Theorem 3 is straightforward and can be found in Appendix I.

7 3-Party Oblivious Set Membership Protocol

We now show how we can securely build and access the set-membership data structure presented in Section 6. This will be fundamental to our efficient Oblivious Hash Table construction.

The core idea is that a single party, say P_1, can locally construct the Cuckoo Hash table and Bloom Filter objects. Since the indices must remain secret shared, the Cuckoo Hash table and Bloom Filter are constructed not from the indices X_i, but on PRP evaluations of the indices $q_i = \mathrm{PRP}_k(X_i)$. This PRP is evaluated in a secure computation, and the output revealed to P_1, who constructs the Cuckoo Hash Table and Bloom Filter and secret-share theses between P_2 and P_3. The hash functions for the Cuckoo Hash Table and Bloom Filter can be public, since the data structures are secret-shared.

If an index x is queried, the parties securely evaluate $q = \mathrm{PRP}_k(x)$ and reveal this to P_2 and P_3. The locations to be accessed in the secret-shared Cuckoo Hash table and the secret-shared Bloom Filter depend only on q and public hash functions. P_2 and P_3 can therefore access the required locations of the secret-shared Cuckoo Hash Table and Bloom Filter and securely calculate the result of the set membership query.

Our protocol works in the $\mathcal{F}_{ABB}$-hybrid model, where $\mathcal{F}_{ABB}$ is defined in Figure 1. The Oblivious Set functionality is defined below. Note that it reveals any repetitions in the array of inputs, or in the array of queries (but does not reveal publicly relationships between queries and inputs). This is necessary since certain parties will learn the PRP evaluations of the inputs, or the queries (but no parties will learn both). This will allow these parties to learn of any duplicates.

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

Build($[\![X]\!]$, N):
Build an Oblivious Set data structure consisting of the elements in array $[\![X]\!] = [\![X_1]\!], \ldots, [\![X_n]\!]$ of variables stored in the ABB, where $X_i \in [N]$ for all $i \in [n]$ (no outputs).
Set $first_i = \min(j : X_j = X_i)$ Reveal $first$ to all parties.
Query($[\![x]\!]$, res):
If $x \in X$ set $z = true$ else, set $z = false$. Store z in $[\![res]\!]$ in the ABB (no outputs).
The functionality maintains a counter of the number of times Query is called, $nQueries$. Set $queries_{nQueries} = x$. Set $firstFound = \min\{j : queries_j = x\}$. Reveal $firstFound$ to all parties.

Fig. 3: Oblivious Set functionality

Protocol Π_{OSet}

Build($[\![X]\!]$, N)

1. Set $[\![k]\!] = \mathcal{F}_{ABB}$.PRP_keygen($N$)
2. P_1 generates and shares with P_2 and P_3:
   (a) Public Cuckoo Hash functions $h_1, h_2 : [N] \to [m]$
   (b) Public Bloom Filter hash functions $g_1, \ldots, g_{\log(N)} : [N] \to [n \log(N)]$.
3. For $i \in [n]$
   (a) Securely evaluate the PRP on X_i: $[\![Q_i]\!] = \mathcal{F}_{ABB}$.PRP_eval($[\![X_i]\!], [\![k]\!]$).
   (b) Reveal to P_1: $qLocal_i = \mathcal{F}_{ABB}$.Output($[\![Q_i]\!], P_1$).
4. For $i \in [n]$, P_1 sets $first_i = j$ where Q_j, $j \leq i$ is the first occurrence of Q_i in Q. P_1 outputs $first$.
5. P_1 constructs a Cuckoo Hash table with a stash for the outputs $qLocal_i$ *i.e.*, P_1 stores $qLocal_i$ in $T_1[h_1(qLocal_i)]$ or $T_2[h_2(qLocal_i)]$ for as many encodings as possible, and stores the remaining random encodings in stash S. If $|S| > \log(N)$ the build fails and P_1 sends **abort** to all parties, who then abort. In all empty locations in the table, P_1 stores $\perp$ (where $\perp \notin [N]$). Let C be the appended tables of T_1 and T_2. P_1 secret-shares C between P_2 and P_3, i.e. for $i \in [2m]$ call $\mathcal{F}_{ABB}$.InputToSharing($C_i, P_1, cuckoo_i, \{P_2, P_3\}$)
6. P_1 constructs a Bloom Filter B of length $n \log(N)$, using inputs S and hash functions $g_1, \ldots, g_{\log(N)}$. P_1 then secret-shares B to P_2 and P_3 as follows: for $i \in [n \log(N)]$ call $\mathcal{F}_{ABB}$.InputToSharing($B_i, P_1, bloom_i, \{P_2, P_3\}$)
7. P_2 and P_3 create empty local dictionary $queries$ for storing PRP query results they learn. Set $nQueries = 0$.

Query($[\![x]\!]$, res)

1. Securely evaluate random the PRP on the query and reveal the output to P_2 and P_3. $[\![qShared]\!] = \mathcal{F}_{ABB}.PRP\_eval([\![x]\!], [\![k]\!])$.
   $q = \mathcal{F}_{ABB}.Output([\![qShared]\!], P_2)$
   $q = \mathcal{F}_{ABB}.Output([\![qShared]\!], P_3)$
2. P_2 and P_3 see if q is already stored in $queries$. If so, set $firstFound = nQueries$ and set $queries[q] = nQueries$. Else set $firstFound = queries[q]$. Reveal $firstFound$ to all players. Increment $nQueries$.
3. Securely query q in the Cuckoo Table as follows. For $i \in \{1, 2\}$,
   (a) P_2 and P_3 locally calculate $cLocal_i = h_i(q) + (i-1) * m$.
   (b) $[\![c_i]\!] = \mathcal{F}_{ABB}$.InputFromSharing($[\![cuckoo_{cLocal_i}]\!], \{P_2, P_3\}$)
   (c) $[\![eq_i]\!] = \mathcal{F}_{ABB}$.Equal($[\![c_i]\!], [\![qShared]\!]$)
   Call $[\![inCuckoo]\!] = \mathcal{F}_{ABB}$.OR($[\![eq_1]\!], [\![eq_2]\!]$).
4. Securely query q in the Bloom Filter i.e., call $[\![inBloom]\!] = [\![true]\!]$. For $i \in [\log(N)]$:
   (a) P_2 and P_3 locally calculate $bLocal_i = g_i(q)$ for $1 \leq i \leq k$.
   (b) $[\![b_i]\!] = \mathcal{F}_{ABB}$.InputFromSharing($[\![bloom_{bLocal_i}]\!], \{P_2, P_3\}$)
   (c) $[\![inBloom]\!] = \mathcal{F}_{ABB}$.AND($[\![inBloom]\!], [\![b_i]\!]$).
5. Securely determine whether it is in either the Cuckoo Table or the Bloom Filter: $[\![res]\!] = \mathcal{F}_{ABB}$.OR($[\![inCuckoo]\!], [\![inBloom]\!]$)

Fig. 4: 3 Party Secure Set Membership

Theorem 4. *Protocol Π_{OSet} (Figure 4) securely implements $\mathcal{F}_{OSet}$ (Figure 3) in the $\mathcal{F}_{ABB}$-hybrid model, in the $(3,1)$ semi-honest setting*

Proof. **Build:** Observe that Π_{OSet}.Build has a probability of failure, whereas $\mathcal{F}_{OSet}$.Build does not. However the failure probability is negligible (Theorem 3), so cannot be used to distinguish the real and ideal executions.

Let $S_{BUILD,1}$ be the simulator for P_1 for Π_{OSet}.Build. It is provided with P_1's input and output, and only needs to produce the messages P_1 received from $\mathcal{F}_{ABB}$, namely $qLocal_1, \ldots, qLocal_n$. It proceeds as follows:

1. $S_{BUILD,1}$ is given $first$.
2. Let $unused = \{1, \ldots, N\}$
3. For $i \in \{1, \ldots, n\}$
   (a) If $first_i = i$ select an element r of $unused$ uniformly at random, remove r from $unused$, and set $qLocal_i = r$.
   (b) Otherwise let $j = first_i$. Set $qLocal_i = qLocal_j$.

Firstly, observe that if $qLocal_i = qLocal_j$ in the program execution, then $first_i = first_j$, so $qLocal_i = qLocal_j$ in the simulator's transcript. The unique values in $qLocal_1, \ldots, qLocal_n$ are results of a *truly* random permutation. Therefore, distinguishing the values of $qLocal$ from the real and simulated executions amounts to distinguishing the pseudo-random permutation from a truly random permutation. Since $view_1^{\pi}(x,y,z)$ is indistinguishable from $S_1(1^n, x, f_1(x,y,z))$, and $f(x,y,z)$ and $output^{\pi}(x,y,z,n)$ are deterministic functions of $S_1(1^n, x, f_1(x,y,z))$ and $view_1^{\pi}(x,y,z)$ respectively, the combined distributions $(S_1(1^n, x, f_1(x,y,z)), f(x,y,z))$ and $(view_1^{\pi}(x,y,z,n), output^{\pi}(x,y,z,n))$ are computationally indistingishable. P_2 and P_3 receive no messages during a build..

Query: There is a negligible probability of a false positive and zero probability of a false negative (Theorem 3). Therefore, the event of a false result does not allow the true and simulated executions to become computationally indistinguishable. The sumulator for P_2 during a query is almost identical to that of P_1 during a build, except that rather than receiving an entire list of pseudo-random permutations to simulate, it receives one at a time. The simulator has to generate a single value that is consistent with a PRP evaluation. It is given a value $firstFound$ for the current query call. It keeps track of all previous values of $firstFound$, as well as all views generated for previous calls to the query. If $firstFound < nQueries$, $S_{QUERY,2}$ sets the message q to the same one that was generated in the the $firstFound^{th}$ query. Otherwise it generates a new, unused message from $[N]$ and sets this to be the message q for the current round. Since P_2 and P_3 have symmetric roles in the protocol, their simulators for the query are identical. P_1 receives no messages during a query.

The communication costs of Π_{OSet} are stated below. The proofs just consist of adding up the costs of each step, and are included in Appendix J.

Theorem 5. *Protocol $\Pi_{OSet.Build}$ (Figure 4) requires $\mathcal{O}(\kappa n)$ communication. In particular it requires n calls to $\mathcal{F}_{ABB}$PRP_eval.*

Theorem 6. *Protocol $\Pi_{OSet.Query}$ (Figure 4) requires $\mathcal{O}(\kappa)$ communication. In particular it requires 1 call to $\mathcal{F}_{ABB}$PRP_eval.*

7.1 3 Party Oblivious Set Membership for small n

Our hierarchical ORAM protocol will need Oblivious Hash Tables, and Oblivious Sets, where n is not $\omega(\log(N))$. In this case, the data structure presented above will have non-negligible failure probablity.

To solve this, when n is small we use a modified set membership protocol $\Pi_{OSetSmall}$, which uses larger Bloom filters and no Cuckoo Hash Tables. We have some security parameter κ, where $\kappa = \omega(\log(N))$. If $n < \kappa$, the Cuckoo Hash Table is not used, and P_1 places all n PRP evaluations in the Bloom Filter, and makes the Bloom filter of size $B = n\kappa$. As before the number of hash functions is $\log(N)$. This makes the probability of a false positive

$$\left(1 - e^{\frac{\log(N)n}{n\kappa}}\right)^{\log(N)} = \left(1 - e^{\frac{\log(N)}{\kappa}}\right)^{\log(N)}$$

which is negligible in N. The proof of security is identical to that of the Π_{OSet}, since the only messages revealed are the PRP evaluations, so $\Pi_{OSetSmall}$ securely implements $\mathcal{F}_{OSet}$ for $n < \omega(\log(N))$.

The communication complexity of a build remains $\mathcal{O}(\kappa n)$ with n secure PRP evaluations and the communication complexity of a query remains $\mathcal{O}(\kappa)$ with 1 secure PRP evaluation.

Therefore, in terms of security and communication cost, other protocols can call $\Pi_{OSetSmall}$ in place of Π_{OSet} when n is small and the behavior will be the same. One small difference, however, is that $\Pi_{OSetSmall}$ needs superlinear storage ($\Theta(\kappa n)$ rather than $\Theta(n \log(N))$). Nevertheless, in the ORAM data structure, only the smaller levels will be instantiated with this data structure, so it will not increase the asymptotic memory usage.

8 (3, 1)-secure Oblivious Hash Table

We will now present how an Oblivious Set can be used to construct an efficient Oblivious Hash Table. The essential realization is that once it is known whether an item is in the Hash Table, the protocol can choose whether to search for the item itself or to search for a pre-inserted dummy element. This means that the protocol need not hide where in the data structure data is stored, nor need it hide the location that is accessed. All that needs to be hidden is whether an item is a dummy element or not, and if not, to avoid revealing any information about which element it is. As such, Oblivious Sets turn out to handle the hardest part of the problem, and any regular hash table may be used to store the data.

Like the Oblivious Set, the Oblivious Hash Table will reveal which indices in the input are duplicates of each other, and will also reveal which queries are repetitions of each other. (In our final ORAM protocol, it will be ensured that there are no duplicates, so this will leak no information.)

The Oblivious Hash Table contains a fixed number of pre-inserted dummy items, and since each distinct non-member query needs to access a distinct dummy, the Oblivious Hash Table will only support a limited number of queries. (The table can be rebuilt with new dummies if need be, but this will not be needed for the Oblivious RAM application.) These pre-inserted dummy items

are searched for also using a PRP. This increases the size of the inputs space of the PRP to $\log(2N)$.

The Oblivious Hash Table is parameterized by the set of keys and values, $(X_1, Y_1), \ldots, (X_n, Y_n)$, the size of the domain of the indices, N, and the maximum number of distinct queries allowed, T. (By definition $T \leq N$.)

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

Build(($[\![X_1]\!]$,$[\![Y_1]\!]$), ..., ($[\![X_n]\!]$, $[\![Y_n]\!]$), N, T):
Let $first_i = \min\{j : X_j = X_i\}$. For all i such that $first_i = i$, store Y_i in a dictionary *dict* under index X_i. (No outputs.)
Reveal $first$ to all parties.
Set $t = 0$ (unique query counter) and $nQueries = 0$ (query counter)
Query($[\![X]\!]$, res):
If $t \geq T$ do nothing. Otherwise, if $X \in X_1, \ldots, X_n$, set $Y = dict[\![X_i]\!]$, otherwise set $Y = \perp$. Store Y in $[\![res]\!]$ (no outputs).
Set $queries_{nQueries} = x$. Set $firstFound = \min\{j : queries_j = x\}$. If $firstFound = t$ increment t. Increment $nQueries$. Reveal $firstFound$ to all parties.
Extract(res):
Returns $n + T - t$ elements, containing all pairs (X_i, Y_i) where $first_i = i$ and X_i was never queried. The remaining items are set to $(\perp, \perp)$. This array is stored in the ABB in $[\![res]\!]$.

Fig. 5: Oblivious Hash Table functionality

Theorem 7. *$\Pi_{OHTable}$ (Figure 6) securely implements $\mathcal{F}_{OHTable}$ in the $\mathcal{F}_{ABB}$-$\mathcal{F}_{OSet}$-hybrid model in the $(3, 1)$ semi-honest security setting.*

Proof. **Build:** $\mathcal{F}_{OSet}$ generates $first$ exactly according to the requirement for $\mathcal{F}_{OHTable}$. Furthermore this output is a deterministic function of the inputs. It follows that we only need to show that simulators exist for each party whose generated messages are computationally indistinguishable from the real messages.

P_1 receives no messages. P_2 receives $\hat{Q}_1, \ldots, \hat{Q}_{n+T}$. S_2 generates $n + T$ random distinct $\log(4N)$-bit messages in place of these. Since these are the result of PRP evaluations on distinct inputs, any entity that could distinguish these from random distinct messages would be able to distinguish the PRP from a random permutation. Hence, by the security of the PRP, the output of S_2 is indistinguishable from the view of P_2. P_3 role is symmetric to P_2.

Query: We need to show both that the correct value $firstFound$ is returned and that $[\![res]\!]$ is set to the correct value. Based on the definition of $firstFound$ in $\mathcal{F}_{OSet}$, and the fact that every query to $\Pi_{OHTable}$ results in exactly 1 query to $\mathcal{F}_{OSet}$, the value $firstFound$ that $\mathcal{F}_{OSet}$ reveals will satisfy exactly $\mathcal{F}_{OHTable}$.

Protocol $\Pi_{OHTable}$

Build(($[\![X_1]\!]$, $[\![Y_1]\!]$), . . . , ($[\![X_n]\!]$, $[\![Y_n]\!]$), N, T)

1. Call $\mathcal{F}_{OSet}$.Build($[\![X_1]\!], \ldots, [\![X_n]\!], N$). This reveals the value $first$ to all parties.
2. P_1 locally chooses k, a key for the PRP $f_k : [4N] \to [4N]$. Securely input k: $[\![k]\!] = \mathcal{F}_{ABB}$.Input(k, P_1).
3. For $i = 1, \ldots, n$
   (a) If $first_i \neq i$, X_i is a duplicate, set $[\![X_i]\!] = [\![N + i]\!]$ and $[\![Y_i]\!] = [\![\perp]\!]$.
   (b) Set $[\![Q_i]\!] = \mathcal{F}_{ABB}$.PRP_eval($[\![X_i]\!]$, $[\![k]\!]$).
4. P_1 creates and uploads dummies indexed from $2N+1$ to $2N+T$, to be queried when an item is not in the Oblivious Hash Table. For $i = 1, \ldots, T$
   (a) P_1 locally evaluates $Q_{n+i} = f_k(2N + i)$
   (b) $[\![Q_{n+i}]\!] = \mathcal{F}_{ABB}$.Input$(Q_{n+i}, P_1)$
   (c) Set $[\![X_{n+i}]\!] = [\![2N + i]\!]$ and $[\![Y_{n+i}]\!] = [\![\perp]\!]$
5. Shuffle the tuples. Set $[\![\hat{Q}]\!], [\![\hat{X}]\!], [\![\hat{Y}]\!] = \mathcal{F}_{ABB}$.Shuffle($[\![Q]\!], [\![X]\!], [\![Y]\!]$)
6. Reveal $\hat{Q}_1, \ldots, \hat{Q}_{n+T}$ to P_2 and P_3. This will allow P_2 and P_3 to find an item's index in the shuffled array, based on its PRP evaluation.
7. Initialize $t = 0$ (counter for unique accesses to data structure).

Query($[\![x]\!]$, res)

1. Check whether the index is stored in the table: $sharein = \mathcal{F}_{OSet}$.Query($[\![x]\!]$).
2. The previous function will also reveal to all parties the value $firstQuery$, which shows the first time x was queried to the Oblivious Hash Table. Let $nQueries$ be the number of times the Query function has been called before. If $firstQuery < nQueries$, this item has been queried before. If so, set $[\![res]\!]$ to the same query result that was provided the previous time $[\![x]\!]$ was queried and return. Otherwise increment t.
3. If the index is not in the table, an index of a pre-inserted dummy is used instead. $[\![x_{dummy}]\!] = \mathcal{F}_{ABB}$.Input$(2N + j, P_1)$.
   $[\![x_{used}]\!] = \mathcal{F}_{ABB}$.IfThenElse($[\![in]\!], [\![x]\!], [\![x_{dummy}]\!]$)
4. The PRP evaluation identifying the real/dummy element is calculated and revealed to P_2 and P_3:
   $[\![q]\!] = \mathcal{F}_{ABB}$.PRP_eval($[\![x_{used}]\!], [\![k]\!]$)
   $q = \mathcal{F}_{ABB}$.Output($[\![q]\!], \{P_2, P_3\}$)
5. P_2 and P_3 find the PRP tag in the permuted PRP array, which allows them to find the value corresponding to that tag. I.e. P_2 and P_3 find j such that $q = \hat{Q}_j$ and reveal j to P_1. The parties set $[\![res]\!] = [\![\hat{Y}_i]\!]$.

Extract(res)

1. It is publicly known which t of the $n + T$ $(\hat{X}, \hat{Y})$ pairs were visited. For $j = 1, \ldots, n + T - t$, let i be the j^{th} unvisited index and set:
   (a) $[\![isDummy]\!] = \mathcal{F}_{ABB}$.Equal($[\![Y_i]\!], [\![\perp]\!]$)
   (b) $[\![X_i]\!] = \mathcal{F}_{ABB}$.IfThenElse($[\![isDummy]\!], [\![\perp]\!], [\![X_i]\!]$)
   (c) $[\![res_j]\!] = ([\![X_i]\!], [\![Y_i]\!])$

Fig. 6: Oblivious Hash Table

If x has been queried before, the protocol stores in $[\![res]\!]$ the same value as it stored last time, so if x is correct on all new queries it will also be correct on all repeated queries. If x has not been queried before, then there are two cases. Either $[\![x]\!] \in [\![X]\!]$, in which case x is queried, or $[\![x]\!] \notin [\![X]\!]$, in which case $2N + j$ is queried. If $[\![x]\!] \in [\![X]\!]$, then $q = f_k(x)$. Let ρ be the permutation of the shuffle, and let i represent the indices prior to the shuffle and $j = \rho(i)$ represent the indices following the shuffle. Since $x = X_i$ for some $i \in [n]$, and $Q_i = f_k(X_i)$, then $q = \hat{Q}_j$ for some j such that $\hat{Y}_j = Y_i$, so $[\![res]\!]$ is set to the correct value. If $[\![x]\!] \notin [\![X]\!]$ then $q = f_k(2N + t)$. Since $1 \leq t \leq T$, $q = Q_{n+t}$. As such, there is some j such that $\hat{Q}_j = q$ and $[\![\hat{Y}_j]\!] = [\![Y_{n+t}]\!] = [\![\perp]\!]$. Therefore $[\![res]\!]$ is set to $[\![\perp]\!]$ as required.

This shows that the query protocol is correct, but it remains to show that it is secure. If the index was queried before, no messages are sent to any player and the protocol is trivially secure. If the index was not queried before, then P_1 learns the index of the accessed item in the shuffled array. $S_{QUERY,1}$ will generate a random index in $[n + T]$ that has not been accessed before. By the definition of $\mathcal{F}_{ABB}$.Shuffle, the accessed items after the shuffle will be truly random distinct values. As such the distribution of the view of P_1 is identical to the distribution generated by $S_{QUERY,1}$. P_2 receives, in each query, the message $q = \hat{Q}_j$. The simulator $S_{QUERY,2}$, simulates this by selecting an element of $\hat{Q}_j$ uniformly at random from among the elements that have not previously been selected. Again, from the security of the shuffle, each accessed element in the real protocol will also be selected uniformly at random from among the unaccessed elements. Therefore the real and simulated views are identical. Since P_3 is symmetric to P_2 its proof of security is the same.

Extract: $[\![res]\!]$ will contain $n + T - t$ elements as required. If $x = X_i$ for $i \in [n]$, and x was queried, then, from above, the index of x will be visited, so X_i and its data Y_i will not be stored in $[\![res]\!]$. If $x = X_i$ for $i \in [n]$, but x was not queried, then $Q_i = f_k(X_i)$ was never revealed as a result of query, so index $\rho(i)$ was never visited in the permuted array. Since Y_i is a real data element $Y_i \neq \perp$, so $[\![X_i]\!], [\![Y_i]\!]$ will be stored in $[\![res]\!]$. All remaining elements stored in $[\![res]\!]$ will be for some $X_i \notin X$, *i.e.*, $i > N$. In these cases $Y_i = \perp$, $([\![\perp]\!], [\![\perp]\!])$ will be stored in $[\![res]\!]$ as required. The security of Extract is trivial since no parties receive messages.

The communication costs of $\Pi_{OHTable}$ are stated below. The proofs just consist of adding the costs of each step, and are included in Appendix K.

Theorem 8. *$\Pi_{OHTable}$.Build (Figure 6) requires $\mathcal{O}(\kappa n + Dn + DT)$ communication. In particular, it requires $2n$ calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Theorem 9. *$\Pi_{OHTable}$.Query (Figure 6) requires $\mathcal{O}(\kappa)$ communication and, in particular, at most 2 calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Theorem 10. *$\Pi_{OHTable}$.Extract (Figure 6) requires $\mathcal{O}((n + T - t)D)$ communication and no calls to $\mathcal{F}_{ABB}$.PRP_eval.*

9 Hierarchical ORAM

The Oblivious Hash Table functionality of Figure 5 has 3 limitations. Firstly, it only allows items to be written once during the build, and then allows items to be read one at a time. Secondly, the build leaks which items in the build are duplicates of each other. Lastly, the query leaks which queried items were repetitions of each other.

We will now present the Oblivious RAM functionality which does away with these limitations. To simplify the protocol, our version of the ORAM functionality executes both a read and a write in a single function call (reads the old value and writes the new). This can easily be converted into calls to the individual functions: ignoring the returned value provides the write (but don't read) function; re-writing the read value provides the read (but don't write) function. The readAndWrite function is executed on a single index at a time. It leaks no information about whether the index was used in previous queries.

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

Init($[\![Y]\!]$, N, D) Initialize the ORAM to initially have stored $([\![i]\!], [\![Y_i]\!])$ for $1 \leq i \leq N$, where $|[\![Y_i]\!]|$ is D bits (no output).
ReadAndWrite($[\![x]\!]$, $[\![y]\!]$, res): (Where $x \in \{1, \ldots, N\}$ and y is D bits.) Set $[\![res]\!]$ to the value that was most recently written to $[\![x]\!]$ and set the new value of index $[\![x]\!]$ to $[\![y]\!]$ (no output).

Fig. 7: Oblivious RAM functionality

Our ORAM protocol builds on the "hierarchical ORAM" solution [Ost92], [Ost90], [GO96], [LO13] and uses the Oblivious Hash Table of Functionality 5 as a building block. Specifically, hierarchical ORAM schemes avoid leakage of repeated queries by building a hierarchy of Oblivious Hash Tables of geometrically increasing sizes, and ensuring that each element is only queried once at each level of the hierarchy.

Any time an item is accessed, the accessed item is placed in a "cache" (L_0). Each time a new query is performed, the entire cache is scanned. If the item is found, the old value is deleted from its previous location in the cache and the new value is appended to the cache.

In this scheme, the size of the cache grows with each access, so searching the cache becomes increasingly expensive. To solve this, the cache has a maximum size, which we set to $\log(N)$. When the cache is full, a new (small) Oblivious Hash Table is built from its contents. This allows these items to be accessed again efficiently. In general, once levels $L_0, \ldots, L_i$ are filled, their (unaccessed) elements are extracted and built into a new larger Oblivious Hash table at level, L_{i+1}, and the levels $L_0, \ldots, L_i$ are emptied. In this way, a hierarchy of $\approx \log_2(N)$ Oblivious

Hash Tables is constructed of geometrically increasing sizes. The hierarchical ORAM scheme is presented in detail in Figure 8.

Before reasoning about correctness, security and communication complexity, we will need to show some basic properties of the protocol.

Definition 1. *We say that an index* $[\![x]\!]$ *where* $x \in [N]$ resides *at a certain level,* L_j*, if one of the following is true.*

- $j = 0$ *and there exists exactly one index in* L_0 *that is set to* $[\![x]\!]$*.*
- $j > 0$ *and there is an Oblivious Hash Table at* L_j *that was built using* $[\![x]\!]$*, and* $[\![x]\!]$ *has not yet been queried to this table.*

Observe that when levels $L_0, \ldots, L_i$ are merged during a rebuild, the new Oblivious Hash Table is built using exactly the real indices that resided at levels $L_0, \ldots, L_i$ (and some dummy indices).

Lemma 1. *If index* $[\![x]\!]$*, where* $x \in [N]$*, was queried to an Oblivious Hash Table that exists at level* L_i*, for* $i \geq 1$*, at the start and end of each function call there exists exactly one level,* L_j*, where* $j < i$*, at which* $[\![x]\!]$ *resides.*

Proof. By induction on the query step and the rebuild step.

At the time $[\![x]\!]$ was queried to L_i, it must not have been found in $L_0, \ldots, L_{i-1}$, (otherwise $[\![N+t]\!]$ would have been queried to L_i instead). Then after the index was queried to L_i, it was written to L_0, so resided in exactly one level (L_0).

In each subsequent query (ignoring the rebuild step), if the index is queried again, it will, by induction, be found in some level L_j for $j < i$. If it is found in L_0, that location will be set to $[\![\perp]\!]$ and if it is found in some other level it will now have been queried in that level. Either way, it will not reside at any level at this moment. It will again be written to L_0, at which point it will again reside in exactly one level (L_0). If another index is queried, the index will not be queried at whichever level it is stored, so the invariant is preserved.

If a rebuild occurs, by the inductive assumption, there is a level, L_j, $j < i$, at which $[\![x]\!]$ resides. Let w be such that the rebuild is to merge levels $L_0, \ldots, L_w$. If $w < j$, L_j is unaffected, so $[\![x]\!]$ will continue to reside at L_j. Since $[\![x]\!]$ did not reside in $L_0, \ldots, L_w$ it will not be built into the new Oblivious Hash Table, so will not reside there. If $j \leq w < i$, since $[\![x]\!]$ resides at level L_j, it will be extracted, and be placed in $[\![A]\!]$. Therefore, the new Oblivious Hash Table produced after a rebuild will contain index $[\![x]\!]$, and it will not yet have been queried at that level before. Since L_i was not merged in the rebuild, it must be that $w \leq i - 2$. Therefore, the new Oblivious Hash Table will be placed in level $w + 1 < i$, so $[\![x]\!]$ now resides in only L_{w+1} satisfying the invariant. Lastly, if $w \geq i$, then the Oblivious Hash Table at level L_i is deleted, and so the condition of the lemma no longer holds.

Corollary 1. *Each index* $[\![x]\!]$ *for* $x \in [N]$ *resides at exactly one level at the start and end of each function call.*

Protocol Π_{ORAM}

Init($[\![Y]\!]$, N, D):

1. Let $top = \log(N) - \log(\log(N)) + 1$ be the index of the top level. Build $\mathcal{F}_{OHTable}$.Build($([\![1]\!], [\![Y_1]\!]), \ldots, ([\![N]\!], [\![Y_N]\!]), 2N, N$). Store it at level L_{top}.
2. Set L_0 to be an empty array of size $\log(N)$.
3. Initialize $t = 1$ (access counter).

ReadAndWrite($[\![x]\!]$, $[\![y]\!]$, res):

1. If $[\![x]\!]$ is found, the protocol searches for a nonce henceforth. Set $[\![x_{used}]\!] = [\![x]\!]$ to hold the updatable value that will be searched for.
2. Securely check whether $[\![x_{used}]\!]$ is in L_0. If so, securely (using $\mathcal{F}_{ABB}$.IfThenElse), set $[\![res]\!]$ to the corresponding data element, set the item at which it was found to $([\![\perp]\!], [\![\perp]\!])$ and update $[\![x_{used}]\!]$ to the nonce $[\![N+t]\!]$.
3. Securely check whether $[\![x_{used}]\!]$ exists in each of the remaining levels, from smallest to largest. Let $\mathcal{F}_{OHTable,i}$ represent the instance of the Oblivious Hash Table functionality for level L_i. I.e., for each non-empty L_i:
   (a) $[\![y_i]\!] = \mathcal{F}_{OHTable,i}$.Query($[\![x_{used}]\!]$)
   (b) $[\![in_i]\!] = \mathcal{F}_{ABB}$.Equal($[\![y_i]\!], [\![\perp]\!]$)
   (c) $[\![x_{used}]\!] = \mathcal{F}_{ABB}$.IfThenElse($[\![in_i]\!], [\![N+t]\!], [\![x_{used}]\!]$)
   (d) $[\![res]\!] = \mathcal{F}_{ABB}$.IfThenElse($[\![in_i]\!], [\![y_i]\!], [\![res]\!]$)
4. Place $([\![x]\!], [\![y]\!])$ in the first empty location in L_0.
5. If L_0 is full, call Rebuild.

Rebuild():

1. Merge all (unaccessed) key-value pairs from levels L_0 to L_i, where i is the largest number such that all levels L_1 to L_i have an Oblivious Hash Table, then delete these tables. I.e. set $[\![A]\!] = L_0$, then for $j \in [i]$:
   (a) Append $[\![extracted_i]\!] = \mathcal{F}_{OHTable,i}$.Extract() to $[\![A]\!]$.
2. The dummy values all have the index $\perp$ however. Before a new Oblivious Hash Table is build, these values need to be deleted or given new names. Let $([\![X_i]\!], [\![Y_i]\!]) = [\![A_i]\!]$. For $i = 1, \ldots, |[\![A]\!]|$,
   (a) $[\![isDummy_i]\!] = \mathcal{F}_{ABB}$.equal($[\![X_i]\!], [\![\perp]\!]$)
3. If $i < top$, give the dummy indices unique names and then build a new Oblivious Hash Table. For $i = 1, \ldots |[\![A]\!]|$:
   (a) $[\![X_i]\!] = \mathcal{F}_{ABB}$.IfThenElse($[\![isDummy_i]\!], [\![N+i]\!], [\![X_i]\!]$)
   Build $\mathcal{F}_{OHTable}$.Build($[\![A]\!], 2N, |[\![A]\!]|$) and store it at level L_{i+1}.
4. If $i = top$, remove dummies and then build a new Oblivious Hash Table.
   (a) $[\![isDu\hat{m}my]\!][\![\hat{X}]\!], [\![\hat{Y}]\!] = \mathcal{F}_{ABB}$.Shuffle($[\![isDummy]\!], [\![X]\!], [\![Y]\!]$)
   (b) For $j = 1, \ldots, |[\![A]\!]|$
      i. $isDu\hat{m}my_j = \mathcal{F}_{ABB}$.Output($[\![isDu\hat{m}my_j]\!], \{P_1, P_2, P_3\}$).
      ii. If $isDu\hat{m}my_j = false$, append $([\![\hat{X}_j]\!], [\![\hat{Y}_j]\!])$ to $[\![\bar{A}]\!]$.
   (c) Build $\mathcal{F}_{OHTable}$.Build($[\![\bar{A}]\!], 2N, N$) and store it at level L_{top}.
   (d) Reset $t = 1$.

Fig. 8: Hierarchical ORAM scheme

Proof. Apply Lemma 1 with $i = top$. Initially, all indices reside only in L_{top}. Any index that has been queried to L_{top}, no longer resides in L_{top} and, by Lemma 1, resides in exactly one level L_j, for $j < top$. Any index that has not been queried to L_{top} still resides in L_{top}, and cannot reside in any smaller level since it has never been written to L_0.

When L_{top} is rebuilt all indices in $[N]$ are therefore included in $\bar{A}$, and an Oblivious Hash Table is constructed from these indices. Since all other levels no longer have Oblivious Hash Tables, all indices now reside only in L_{top}.

Corollary 2. *Each query to each Oblivious Hash Table is distinct.*

Proof. If a real index was queried once to an Oblivious Hash Table at level L_i (even if the index resided in a larger level) the index will then reside in a smaller level until the table is deleted. If queried again to the ORAM, it will then be found at the smaller level first, and $[\![N+t]\!]$ will be queried to the Oblivious Hash Table instead. Since t increments with each access, the values of $N + t$ will be distinct for the lifetime of any Oblivious Hash Table.

Lemma 2. *For an Oblivious Hash Table at level L_i, for $1 \leq i < top$, the following properties hold:*

1. *It will be built using $\log(N)2^{i-1}$ items.*
2. *It will be accessed $\log(N)2^{i-1}$ times before being extracted and deleted.*
3. *It will have $\log(N)2^{i-1}$ items extracted from it.*
4. *A (new) table is extracted and deleted from L_i every $\log(N)2^i$ accesses,*

Proof. By strong induction. First observe that property 3 follows directly from property 1 and 2 since Oblivious Hash Tables are always instantiated with $T = |A| = \log(N)2^{i-1}$ items, so after $\log(N)2^{i-1}$ distinct accesses, $|A| + T - \log(N)2^{i-1} = \log(N)2^{i-1}$ elements will be extracted. Hence we do not need to separately prove property 3.

For $i = 1$, L_1 is built using the elements of the cache, L_0, so there are $\log(N)$ such elements, as per property 1. The table at L_1 will be extracted and deleted, when the cache becomes full again, that is after a further $\log(N)$ accesses, satisfying property 2. In general, every $\log(N)$ accesses, L_0 becomes full, and on alternating periods, a table will either exist at L_1 (in which case it is extracted) or it will not exist (in which case it is built), so extractions at level L_1 will occur ever $2\log(N)$ accesses.

For $1 < i < top$, when a table is built at L_i, it is built because L_0 is full, and is built by extracting all tables from previous levels. By property 3, the total number of items will be $\log(N) + \sum_{j=1}^{i-1} \log(N)2^{j-1} = \log(N)2^{i-1}$. When the table at L_i was build, the table at L_{i-1} was extracted and deleted. The table at L_i will be extracted and deleted again when that of L_{i-1} is, which by property 4 occurs after another $\log(N)2^{i-1}$ accesses. In general, every $\log(N)2^{i-1}$ accesses, L_{i-1} is extracted, which either causes L_i to have a table built if there is none, or extracted and deleted if there is one. Therefore, extractions and deletions will occur every $\log(N)2^i$ accessses.

Theorem 11. *Protocol Π_{ORAM} (Figure 8) implements functionality $\mathcal{F}_{ORAM}$ (Figure 7) in the $\mathcal{F}_{ABB}$-$\mathcal{F}_{OHTable}$-hybrid model.*

Proof. First we show that the correct value is returned by the read. Since an index $[\![x]\!]$ resides in a single level of the table, (Corollary 1) it will be found at that level. Initially, the data associated with $[\![i]\!]$ will be $[\![Y_i]\!]$ of the original array. Henceforth, whatever value was last written to the index will be associated with it (first by the pair being stored in L_0, and after because the pair are moved together during rebuilds until $[\![i]\!]$ is written to again).

Next, we show that the views are simulatable. Note that neither the theorem nor the protocol assume a $(3, 1)$ semi-honest access structure. The messages created by the protocol are the values $first$ and $firstFound$ revealed by the $\mathcal{F}_{OHTable}$ during builds and queries respectively. Before a new Oblivious Hash Table is built, each index resides at exactly one level, so $[\![A]\!]$ contains only a single copy of each real index. Each dummy input in $[\![A]\!]$ is also made distinct (to some value in $\{N + 1, \ldots, N + |[\![A]\!]|\}$). Therefore $first = 1, \ldots, |A|$, so can be easily simulated. Similarly, the queries to a given Oblivious Hash Table are distinct (Corollary 2) so the j^{th} value of $firstFound$ returned by a given Oblivious Hash Table will be j, so is easily simulated.

Therefore the views of any parties is identical in the real and simulated executions.

Theorem 12. *The amortized communication cost of Π_{ORAM} when $\mathcal{F}_{OHTable}$ is instantiated with $\Pi_{OHTable}$ is $\mathcal{O}(\log(N)(\kappa + D))$ per ReadAndWrite. In particular, it requires amortized less than $2\log(N)$ calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Proof. We amortize the rebuild costs over N accesses. We ignore the cost of Init, as this is a once-off cost.

Checking L_0 requires at most $\log(N)$ secure $\log(N)$-bit comparisons, and $\mathcal{O}(\log(N))$ calls to $\mathcal{F}_{ABB}$.IfThenElse on $\mathcal{O}(D)$-bit inputs, for a total cost of $\mathcal{O}(D\log(N))$.

There are $\log(N) - \log(\log(N)) + 1 \leq \log(N)$ remaining levels. In general though, based on the observation that tables exist at a level only half the time (Lemma 2), asymptotically only half of these levels will be queried. Each level that has a table will require 2 calls to $\mathcal{F}_{ABB}$.PRP_eval, and $\mathcal{O}(\kappa)$ communication for the OHTable query, and a further $\mathcal{O}(D)$ communication for the IfThenElse statements. Therefore, asymptotically, querying the other levels will require $\log(N)$ calls to $\mathcal{F}_{ABB}$.PRP_eval. and $\mathcal{O}(\log(N)(\kappa + D))$ communication.

Level i is rebuilt every $\log(N)2^i$ accesses, using a set of size $n_i = \log(N)2^{i-1}$ and parameter $T_i = \log(N)2^{i-1}$. Therefore the cost of building the level is $\mathcal{O}((\kappa + D)n_i)$ and, in particular, $2n_i$ calls to $\mathcal{F}_{ABB}$.PRP_eval. Per access this is $\mathcal{O}(\kappa+D)$ communication and 1 call to $\mathcal{F}_{ABB}$.PRP_eval. Similarly a table at level i is extracted every $\log(N)2^i$ levels, with cost $\mathcal{O}(\kappa + D)$ per access.

Identifying and renaming/deleting dummies incurs an additional $\mathcal{O}(D)$ communication cost per access per level. As such the total communication cost is $\mathcal{O}((\kappa + D)\log(N))$ per access, and in particular 2 calls to $\mathcal{F}_{ABB}$.PRP_eval.

Combining Theorems 11 and 12 completes the proof of Theorem 1.

10 Acknowledgements

This research was sponsored in part by ONR grant (N00014-15-1-2750) "SynCrypt: Automated Synthesis of Cryptographic Constructions". Supported in part by DARPA under Cooperative Agreement HR0011-20-2-0025, NSF grant CNS-2001096, US-Israel BSF grant 2015782, Google Faculty Award, JP Morgan Faculty Award, IBM Faculty Research Award, Xerox Faculty Research Award, OKAWA Foundation Research Award, B. John Garrick Foundation Award, Teradata Research Award, Lockheed-Martin Research Award and Sunday Group. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of DARPA, the Department of Defense, 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

AFL+16. Toshinori Araki, Jun Furakawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. High-throughput semi-honest secure three-party computation with an honest majority. In *CCS*, pages 805–817, 2016.

AGJS13. Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative technology for CPU based attestation and sealing. In *HASP*, 2013.

AKL+20. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In *EUROCRYPT*, pages 403–432. Springer, 2020.

AKST14. Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In *PKC*, pages 131–148. Springer, 2014.

Ang. Dana Angluin. Circuits to test equality. `https://zoo.cs.yale.edu/classes/cs201/topics/topic-compare-equality.pdf`.

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.

Aum10. Martin Aumüller. *An alternative analysis of cuckoo hashing with a stash and realistic hash functions*. PhD thesis, Diplomarbeit, Technische Universität Ilmenau, 2010.

BGI15. Elette Boyle, Niv Gilboa, and Yuval Ishai. Function secret sharing. In *EUROCRYPT*, pages 337–367. Springer, 2015.

BGK+08. Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Information Processing Letters*, 108(4):210–213, 2008.

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

BMD+17. Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software grand exposure: SGX cache attacks are practical. In *WOOT*, 2017.

BOGW88. Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation. In *STOC*, pages 1–10, New York, NY, USA, 1988. ACM.

CCD88. David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty Unconditionally Secure Protocols. In *STOC*, pages 11–19, 1988.

CD16. Victor Costan and Srinivas Devadas. Intel SGX explained. IACR ePrint 2017/086, 2016.

CGLS17. T-H Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In *ASIACRYPT*, pages 660–690. Springer, 2017.

CKN+18. T-H Hubert Chan, Jonathan Katz, Kartik Nayak, Antigoni Polychroniadou, and Elaine Shi. More is less: Perfectly secure oblivious algorithms in the multi-server setting. In *ASIACRYPT*, pages 158–188. Springer, 2018.

CR73. Stephen A Cook and Robert A Reckhow. Time bounded random access machines. *Journal of Computer and System Sciences*, 7(4):354–375, 1973.

CRJ10. Ken Christensen, Allen Roginsky, and Miguel Jimeno. A new analysis of the false positive rate of a bloom filter. *Information Processing Letters*, 110(21):944–949, 2010.

DEG+18. Christoph Dobraunig, Maria Eichlseder, Lorenzo Grassi, Virginie Lallemand, Gregor Leander, Eik List, Florian Mendel, and Christian Rechberger. Rasta: a cipher with low AND depth and few ANDs per bit. In *CRYPTO*, pages 662–692. Springer, 2018.

Ds17. Jack Doerner and abhi shelat. Scaling ORAM for secure computation. In *CCS*, pages 523–535. ACM, 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*, pages 145–174. Springer, 2016.

FAKM14. Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *CoNEXT*, pages 75–88, 2014.

FIPR05. Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *TCC*, pages 303–324. Springer, 2005.

FJKW15. Sky Faber, Stanislaw Jarecki, Sotirios Kentros, and Boyang Wei. Three-party ORAM for secure computation. In *ASIACRYPT*, pages 360–385. Springer, 2015.

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

FNR+15. Christopher W Fletcher, Muhammad Naveed, Ling Ren, Elaine Shi, and Emil Stefanov. Bucket ORAM: Single online roundtrip, constant bandwidth oblivious RAM. 2015.

GBK11. David Gullasch, Endre Bangerter, and Stephan Krenn. Cache games–bringing access-based cache attacks on AES to practice. In *S&P*, pages 490–505. IEEE, 2011.

GI14. Niv Gilboa and Yuval Ishai. Distributed point functions and their applications. In *EUROCRYPT*, pages 640–658. Springer, 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*, pages 513–524, 2012.

GKW18. S Dov Gordon, Jonathan Katz, and Xiao Wang. Simple and efficient two-server ORAM. In *ASIACRYPT*, pages 141–157. Springer, 2018.

GM11. Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *ICALP*, pages 576–587. Springer, 2011.

GMOT12. Michael T Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *SODA*, pages 157–167. SIAM, 2012.

GMW87. Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game. In *STOC*, pages 218–229, 1987.

GO96. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *JACM*, 43(3):431–473, 1996.

GRR+16. Lorenzo Grassi, Christian Rechberger, Dragos Rotaru, Peter Scholl, and Nigel P Smart. MPC-friendly symmetric key primitives. In *CCS*, pages 430–443, 2016.

HLP+13. Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *HASP*, 2013.

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

HYG20. Thang Hoang, Attila A Yavuz, and Jorge Guajardo. A multi-server ORAM framework with constant client bandwidth blowup. *ACM Transactions on Privacy and Security (TOPS)*, 23(1):1–35, 2020.

JW18. Stanislaw Jarecki and Boyang Wei. 3PC ORAM with low latency, low bandwidth, and fast batch retrieval. In *ACNS*, pages 360–378, 2018.

KLO12. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in) security of hash-based oblivious RAM and a new balancing scheme. In *SODA*, pages 143–156. SIAM, 2012.

KM19. Eyal Kushilevitz and Tamer Mour. Sub-logarithmic distributed oblivious RAM with small block size. In *PKC*, pages 3–33. Springer, 2019.

KMW09. Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2009.

KPW16. David Kaplan, Jeremy Powell, and Tom Woller. AMD memory encryption. Technical report, AMD, April 2016.

LN18. Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In *CRYPTO*, pages 523–542. Springer, 2018.

LO13. Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In *TCC*, pages 377–396. Springer, 2013.

LWZ11. Sven Laur, Jan Willemson, and Bingsheng Zhang. Round-efficient oblivious database manipulation. In *ISC*, pages 262–277. Springer, 2011.

MAB+13. Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R Savagaonkar. Innovative instructions and software model for isolated execution. In *HASP*, 2013.

MIE17. Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. Cachezoom: How SGX amplifies the power of cache attacks. In *CHES*, pages 69–90. Springer, 2017.

Mit09. Michael Mitzenmacher. Some open questions related to cuckoo hashing. In *ESA*, pages 1–10, 2009.

MU17. Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomization and probabilistic techniques in algorithms and data analysis*. Cambridge university press, 2017.

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

Nob21. Daniel Noble. An intimate analysis of cuckoo hashing with a stash. IACR ePrint 2021/447, 2021.

OS97. Rafail Ostrovsky and Victor Shoup. Private information storage. In *STOC*, volume 97, pages 294–303, 1997.

Ost90. Rafail Ostrovsky. Efficient computation on oblivious RAMs. In *STOC*, pages 514–523, 1990.

Ost92. Rafail Ostrovsky. *Software Protection and Simulation On Oblivious RAMs*. PhD thesis, 1992.

PF79. Nicholas Pippenger and Michael J Fischer. Relations among complexity measures. *JACM*, 26(2):361–381, 1979.

PPRY18. Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In *FOCS*, pages 871–882. IEEE, 2018.

PR01. Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. In *ESA*, pages 121–133. Springer, 2001.

PR10. Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *CRYPTO*, pages 502–519. Springer, 2010.

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*, pages 197–214. Springer, 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*, pages 299–310, 2013.

TOS10. Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient cache attacks on AES, and countermeasures. *Journal of Cryptology*, 23(1):37–71, 2010.

WCS15. Xiao Wang, Hubert Chan, and Elaine Shi. Circuit ORAM: On tightness of the Goldreich-Ostrovsky lower bound. In *CCS*, pages 850–861, 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*, pages 191–202. ACM, 2014.

WLN+15. X. S. Wang, C. Liu, K. Nayak, Y. Huang, and E. Shi. ObliVM: A programming framework for secure computation. In *S & P*, 2015.

Yao82. Andrew Yao. Protocols for secure computations (extended abstract). In *FOCS*, pages 160–164, 1982.

Yao86. Andrew Yao. How to generate and exchange secrets. In *FOCS*, pages 162–167, 1986.

ZE15. Samee Zahur and David Evans. Obliv-C: A language for extensible data-oblivious computation. IACR ePrint 2015/1153, 2015.

ZRE15. Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *EUROCRYPT*, pages 220–250. Springer, 2015.

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*, pages 218–234. IEEE, 2016.

Supplementary Material

A Comparison with LO13

As discussed in Section 2, the 2-Server ORAM protocol of [LO13] can also be used to implement a DORAM protocol, by simulating the client inside of an MPC. The main obstacle to efficiency is that the client performs symmetric-key encryptions and decryptions, which are expensive when executing inside an MPC.

Asymptotically, using MPC to simulate the [LO13] protocol is actually efficient. It is possible for block cipher operations on plaintexts of size D to be computed using $\mathcal{O}(\kappa + D)$ AND gates. Using this and generic MPC techniques to simulate the client of [LO13] leads to a DORAM protocol with ammortized $\mathcal{O}((\kappa + D)\log(N))$ communication per access. However, this protocol would be very far from practical. Specifically we show below that the number of SISO-PRPs we need is fewer than [LO13] by a factor of about 50. These results are tabulated in Figure 9, and are explained verbally below.

We examine Figures 3 and 4 of [LO13] and calculate the number of SISO-PRP calls. Encryptions, decryptions and PRF evaluations will all be counted as a single PRP call. Note that encryptions and decryptions in [LO13] are actually applied to both the index and the data payload, so will require PRPs on larger block sizes than those used by our protocol and will therefore require *more communication* than the SISO-PRPs in our protocol.

It is important to note that [LO13] uses the cache-the-stash technique. This doubles the frequency of rebuilds: in our protocol a table with n elements is rebuilt every $2n$ accesses, but in the protocol of [LO13] rebuilds occur every n accesses. The observation that a level only contains hash tables half of the time still holds. (Although this observation was not made in [LO13], it would be used by any implementation.)

Queries in [LO13] begin by querying the top level. Unlike our protocol, [LO13] caches the stash, which means that the top level will always have at least $\log(N)$ elements. When it is full it will have $2\log(N)$ elements, so on average it has $1.5\log(N)$ elemens. Each of these is decrypted in step 2, leading to amortized $1.5\log(N)$ SISO-PRP evaluations.

[LO13] then accesses the smaller levels, which use standard hash tables, with buckets of capacity $3\log(N)/\log(\log(N))$. There are approximately $7\log(\log(N))$ such levels, and on average half of these will have tables at a given point in time. This means that the average number of items that will need to be accessed in the small levels is $10.5\log(N)$. Each item will need to be both decrypted (step 3b) and re-encrypted (step 3d) leading to $21\log(N)$ SISO-PRPs per access. Additionally, a PRF will need to be executed at each level (step 3a), leading to a futher $7\log(\log(N))$ SISO-PRPs. To simplify the analysis, we will ignore $\log(\log(N))$ terms since these will be small relative to $\log(N)$ for large N.

The remainder of levels accessed are Cuckoo tables. These will only require accessing two locations per level. Ignoring $\log(\log(N))$ terms, there will be $\log(N)$

such levels, and therefore, on average, $\frac{\log(N)}{2}$ such tables. Each item, again, will need to be decrypted (step 4b) and again re-encrypted (step 4d), each leading to amortized costs of $\log(N)$ SISO-PRPs per access. Additionally there will be a PRF call for each Oblivious Hash Table (step 4a) each costing 1 SISO-PRP. Since there are on average $\frac{\log(N)}{2}$ levels this costs amortized $0.5\log(N)$ SISO-PRPs.

Lastly, the top level is accessed again. This time items are both decrypted and re-encrypted. Since on average there are $1.5\log(N)$ items in this level, this needs, on average, $3\log(N)$ SISO-PRPs.

In the build, the role that a server plays during the building of a table depends on which server will hold the built table, since the servers hold alternating levels of the hierarchy. To build a level that will be held by S_b, the items of S_a are first sent over to S_b. To do this these items are decrypted and re-encrypted by the client. Next these are combined with items held by S_b, permuted by S_b and then sent back to S_a, each first being decrypted and re-encrypted by the client. The client also sends S_a the PRF evaluations that S_a will need to build the table. S_a builds the table and the table is sent to S_b, but once again every item is decrypted and re-encrypted.

Additionally, since [LO13] use the cache-the-stash technique, rebuilds will occur twice as frequently as in our construction. Specifically, Table i will hold $c2^i$ elements (including dummies), but will be rebuilt every $c2^{i-1}$ accesses.

First the elements held by S_a need to be re-encrypted and send to S_b (Rebuild step 2). It will be easier to count this cost based on the table that is being extracted, rather than the one being built. We assume that when a table is extracted to be placed into a lower level, that half of the time it will be placed in a level that will be held by the other party. (In reality it will be more than half, since half of the time it is placed in the level below it, which is held by the other party.) In this case, the costs described in step 2 of the reshuffle will be incurred.

For the small levels, there will be $3n\log(N)/\log(\log(N))$ elements per level. Each will need to be encrypted and decrypted (2 SISO-PRPs). This will occur every $2n$ accesses. (Since the rebuilds occur every n accesses, and for at least half of these, step 2 will need to occur.) There are $7\log(\log(N))$ such levels. Therefore the cost of step 2 on small levels will be $3n\log(N)/\log(\log(N)) * 2/(2n) * 7\log(\log(N)) = 21\log(N)$ SISO-PRPs.

Cuckoo levels are similar, except that there will be only $2\epsilon n$ elements per level and will be $\log(N)$ levels (ignoring $\log(\log(N))$ terms). Therefore the cost for cuckoo levels will be $2\epsilon n * 2/(2n) * \log(N) = 2\epsilon\log(N)$ SISO-PRPs per level.

The reshuffle in step 3 happens locally so incurs no communication cost.

In step 4, each element in the table is again decrypted. Again we count these based on the table from which the elements came. This will happen to all elements (both S_a's and S_b's), so will always happen when a table is extracted. Since rebuilds occur every n accesses, the amortized number of SISO-PRPs per access for decrypting items from small tables will be $3n\log(N)/\log(\log(N))/n * 7\log(\log(N)) = 21\log(N)$ SISO-PRPs. The items are then re-encrypted, but empty items need not be re-encrypted. The small tables only have $\log(\log(7))$

items than need be re-encrypted, which will be ignored in this analysis. For larger Cuckoo tables the situation is similar, except there are $2\epsilon n$ items per level and about $\log(N)$ levels. This leads to $2\epsilon \log(N)$ items. Additionally, the non-empty items will need to be encrypted. There will be n of these, leading to an additional amortized $\log(N)$ SISO-PRPs per access for the cuckoo levels. Step 4 also simulates n PRF evaluations. For each level this occurs every n access, adding an amortized cost of $\log(N)$ SISO-PRPs.

Step 5 is again performed locally by S_A so incurs no communication cost.

When a table of capacity n is being built in step 6, the n non-empty items given from S_a need to be decrypted. Decrypting the non-empty items costs amortized $\log(N)$ SISO-PRPs per access (summed over all levels). Following this, every item in the resulting table (both empty and not) must be encrypted and sent to S_b. The total amortized cost for this over all small tables is $21\log(N)$ SISO-PRPs. The total amortized cost for this over all Cuckoo tables is $2\epsilon\log(N)$ SISO-PRPs.

Step	**SISO-PRPs per access (amortized)**
Query: 2	$1.5\log(N)$
Query: 3b	$10.5\log(N)$
Query: 3d	$10.5\log(N)$
Query: 4a	$\log(N)$
Query: 4b	$\log(N)$
Query: 4d	$0.5\log(N)$
Query: 6	$3\log(N)$
Query Total:	$28\log(N)$
Reshuffle: 2 (buckets)	$21\log(N)$
Reshuffle: 2 (cuckoo)	$2\epsilon\log(N)$
Reshuffle: 4 (encrypt non-empty)	$\log(N)$
Reshuffle: 4 (decrypt buckets)	$21\log(N)$
Reshuffle: 4 (decrypt cuckoo)	$2\epsilon\log(N)$
Reshuffle: 4 (PRFs)	$\log(N)$
Reshuffle: 6 (decrypt non-empty)	$\log(N)$
Reshuffle: 6 (encrypt buckets)	$21\log(N)$
Reshuffle: 6 (encrypt cuckoo)	$2\epsilon\log(N)$
Reshuffle Total:	$(66+6\epsilon)\log(N)$
Total:	$(94+6\epsilon)\log(N)$

Fig. 9: Communication Cost of [LO13]: Number of SISO-PRPs

Figure 9 shows the total costs. Since $\epsilon > 1$, the total cost is at least an amortized $100\log(N)$ PRPs per access. Our protocol requried amortized only $2\log(N)$ SISO-PRPs per access. Therefore [LO13] requires about 50 times more SISO-PRPs than our protocol.

B Asymptotic Cost of LowMC

One appealing property of LowMC as a block cipher is that the security level can easily be configured. While [ARS+15] does not explicitly state the asymptotic relationship between the security level and the number of AND gates of a LowMC circuit, their implememtation contains a script[9] to calculate the number of AND gates needed for any concrete security setting. From these we extrapolate that LowMC can encrypt data of length D bits with κ-bit security using $\mathcal{O}(\kappa + D)$ AND gates.

For a given security level, κ, we set the block-size equal to the security level. We also assume a data level equal to the security level. (The data level is the log of the number of elements viewable by the adversary. It is therefore upper-bounded by κ and the number of AND gates needed will be highest when it is κ.) We find the minimum number of AND gates needed at this level. These results are presented in Figure 10. It appears from these results that the number of AND gates needed is $\mathcal{O}(\kappa)$, with a constant of about 7.

If we wish to encrypt fewer than κ bits, we simply pad the input and the encryption costs $\mathcal{O}(\kappa)$ AND gates. If we wish to encrypt D bits, where $D > \kappa$, since our block size is also κ this will require $\mathcal{O}(D/\kappa)$ PRP calls, so if a PRP call requires $\mathcal{O}(\kappa)$ AND gates, encrypting all D bits will require $\mathcal{O}(D)$ AND gates. In general therefore, the cost of encrypting D bits with κ-bit security using LowMC is $\mathcal{O}(\kappa + D)$.

κ	Block Size	Data	AND gates	AND gates / κ
20	20	20	282	14.1
40	40	40	375	9.4
60	60	60	465	7.8
80	80	80	582	7.3
100	100	100	699	7.0
120	120	120	816	6.8
140	140	140	933	6.7
160	160	160	1050	6.6

Fig. 10: LowMCv3 Complexity

C Conversions to and from $(2, 2)$ sharing

Our Arithmetic Black Box presented in Figure 1 has the ability to share to and from a $(2, 2)$ sharing. Our $(2, 2)$ sharing is a simple XOR-sharing, i.e. if the sharing is between P_1 and P_2, P_i has x_i where x_1 and x_2 are uniformly random booleans subject to the constraint $x_1 \oplus x_2 = x$, where x is the secret.

[9] https://github.com/LowMC/lowmc/blob/master/determine_rounds.py

$\mathcal{F}_{ABB}$.InputTo2Sharing(x, pId, varName, sharing) can be implemented by player pId selecting a random $x_1 \xleftarrow{\$} \{0,1\}$, setting $x_2 = x_1 \oplus x$ and sending x_i to $sharing_i$. This is a ubiquitous protocol for generating a $(2,2)$ sharing of a secret. It requires 2 bits of communication (only 1 if pId $\in$ sharing).

For $\mathcal{F}_{ABB}$.ReshareTo2Sharing($[\![x]\!]$, sharing, varName) we will need a protocol to convert from the $(3,2)$ sharing of [AFL+16] to a $(2,2)$ sharing. The protocol of [AFL+16] represents a sharing of secret v as follows. x_1, x_2, x_3 are chosen uniformly at random subject to $x_1 \oplus x_2 \oplus x_3 = 0$. Then $a_i = x_{i-1} \oplus v$, where subscripts wrap-around in $[1, \ldots, 3]$. P_i's share is (x_i, a_i). Then P_i, P_{i+1} can form a $(2,2)$ secret-sharing of v by P_i keeping x_i and P_{i+1} keeping a_{i+1}. This is a correct $(2,2)$-XOR sharing since $x_i \oplus a_{i+1} = v$. Since no communication occurs, there are no messages to simulate, and so security is automatic.

$\mathcal{F}_{ABB}$.ReshareFrom2Sharing($[\![x]\!]_{sharing}$, sharing, varName) can be implemented simply by each of the two parties in the sharing calling $\mathcal{F}_{ABB}$.Input(x, pId, varName) on their share to store their share in the ABB. The real result can then be computed within the ABB as the XOR of these two stored values. Since all data is kept within the ABB, this protocol is secure.

D Multiparty secure shuffles

Our protocol requires a secure shuffle protocol, denoted $\mathcal{F}_{ABB}$.Shuffle.

Although our protocol can be instantiated with any 3-party shuffle, we imagine using the the multiparty shuffle of [LWZ11].

The key idea is that if a vector is secret shared among n participants, with an t-out-of-n secret sharing scheme, then for *every* subset C. of $n - t$ participants, the participants reshare the vector to the members of C, then the members of C permute their shares *using* a shuffle that is public to all members of C.

If there are only t corrupt participants, there will be some subset C, that is completely honest, and the permutation chosen by this subset will remain hidden from the adversary.

The key benefit of this approach is that all the shuffles are done in the clear, and the only communication is sharing seeds for the (pseudorandom) permutations, and repeatedly re-sharing the vector.

In the $(3,1)$ security setting this only requires 3 local shuffles and 3 resharings, so it is very efficient. The formal algorithm is presented as Algorithm 1.

Note that we sometimes pass to the shuffle procedure multiple arrays of the same length. In this case the same permutations are used for each array.

Asymptotic Complexity: Given computationally bounded adversaries it is possible for the randomness for the permutation to be generated by a PRG. Therefore, an unlimited number of local pair-wise shuffles could be performed with only a single PRG seed being shared between each pair of parties. The amortized cost of this PRG seed distribution therefore tends to 0. The communication cost is therefore dominated by the cost of resharing, which requires

$$\Theta(nD)$$

Algorithm 1 Secure $(3, 1)$ shuffle

procedure SHUFFLE($[\![X]\!]$)
  $n \leftarrow |[\![X]\!]|$
  $[\![\pi_1]\!]_{A,B} \xleftarrow{\$} \Pi(n)$ ▷ $\xleftarrow{\$} \Pi(n)$ chooses a pseudo-random permutation on $\{1, \ldots, n\}$.
  $[\![\pi_2]\!]_{B,C} \xleftarrow{\$} \Pi(n)$
  $[\![\pi_3]\!]_{A,C} \xleftarrow{\$} \Pi(n)$
  $[\![X_1]\!]_{BC} \leftarrow \text{Shuffle}_{AB}([\![X]\!]_{AB}, [\![\pi_1]\!]_{A,B})$
  $[\![X_2]\!]_{AC} \leftarrow \text{Shuffle}_{BC}([\![X_1]\!]_{BC}, [\![\pi_2]\!]_{B,C})$
  $[\![X_3]\!] \leftarrow \text{Shuffle}_{AC}([\![X_2]\!]_{AC}, [\![\pi_3]\!]_{A,C})$
  return $[\![X_3]\!]$
end procedure

bits of communication where n is is the number of elements in the list, and each element is D bits.

It requires 4 rounds of communication. This can be reduced to 3 rounds if the received or target sharings are pair-wise.

Locally, it requires each player to evaluate 2 permutations on sets of size n. Again, if the adversary is computationally bounded, this can be a pseudo-random permutation.

Concrete Complexity

For each subset C, the members of C need to agree on the seed for a permutation. This can be done by having one member generate a seed, and send it to the other members of C.

Since a single shared seed can be extended to provide new permutations for *all* subsequent calls to $\mathcal{F}_{ABB}$.Shuffle, and thus this cost is amortized across all calls to $\mathcal{F}_{ABB}$.Shuffle. As noted in Table 2 in Section 3, these one time setup costs are not included in the $\mathcal{F}_{ABB}$ model.

During the execution of the protocol, for each committee, C, of 2 players, the players execute $\mathcal{F}_{ABB}$.ReshareTo2Sharing to reshare to the members of C, the members of C shuffle the shares (which requires no communication), and finally the members of C call $\mathcal{F}_{ABB}$.ReshareFrom2Sharing to reshare back to the full set of participants.

Thus the total communication cost of the protocol is 3 calls to $\mathcal{F}_{ABB}$.ReshareTo2Sharing, and 3 calls to $\mathcal{F}_{ABB}$.ReshareFrom2Sharing.

Since $\mathcal{F}_{ABB}$.ResharingTo2Sharing has no communication, and $\mathcal{F}_{ABB}$.ReshareFrom2Sharing requires 8 bits of communication per bit of secret data, the total communication cost to shuffle n elements, each of size D is $24nD$.

Security: Since each player is excluded from one of these permutations, the resulting permutation remains secret.

The data remains secret-shared, so no data is leaked to any participants during this protocol.

E Hashing

Definition 2 (Hashing). *An (n, m, s)-hashing scheme, H, consists of three procedures* $(\mathsf{Gen}, \mathsf{Build}, \mathsf{Lookup})$.

1. ***Key generation:*** $\mathsf{Gen}(\cdot)$ *is a PPT algorithm that takes an index size, $N \in \mathbb{Z}$, and outputs a key, κ.*
$$\kappa \xleftarrow{\$} \mathsf{Gen}(N)$$
2. ***Build:*** Build *is a RAM program takes a key κ, and a list $X = \{(\tau, x) \mid \tau \in [N]\}$, where $|X| = n$, and each τ is distinct. The* Build *program returns a table T of size m and a stash S of size at most s. Every position in T contains a single element $(\tau, x) \in X$ or is empty.*
$$(T, S) \leftarrow \mathsf{Build}(\kappa, X).$$
3. ***Query:*** *The query procedure takes a bucket, $\tau \in [n]$, and a key κ, and returns a set of indices $I \subset [m]$.*
$$I \leftarrow \mathsf{Lookup}(\tau, \kappa).$$

The procedures $(\mathsf{Gen}, \mathsf{Build}, \mathsf{Lookup})$ *satisfy the following properties:*
Correctness: *If $\kappa \leftarrow \mathsf{Gen}(N)$, and $(T, S) \leftarrow \mathsf{Build}(\kappa, x)$.*

- ***Build failure:*** *The build procedure has a negligible probability of failing.*
$$\Pr\left[(T, S) = (\perp, \perp)\right] = O\left(N^{-\omega(1)}\right)$$
- ***Lookup:*** *If $T \neq \perp$, then for any $(\tau, x) \in X$, if $I \leftarrow \mathsf{Lookup}(\tau, \kappa)$, then either $T[i] = (\tau, x)$ for some $i \in I$, or $(\tau, x) \in S$.*

The complexity of the hashing scheme is measured as follows. ***Complexity:*** *The complexity of* Build *is measured by the amount of data exchanged by the CPU and RAM in an* Build *invocation. The complexity of* Lookup *is the maximum size of the set I returned by* Build.

F Oblivious hashing

Hash tables are widely used in ORAM protocols, but care needs to be taken in their application. An adversary who monitors the build procedure as well as subsequent queries to the hash table can distinguish real queries from "dummy" queries. To address this, many ORAM schemes rely on *oblivious* hashing. A hashing scheme is said to be oblivious if Build is oblivious, *i.e.*, the access pattern of the RAM program Build is independent of its inputs (κ, X). To address this, [GO96] gave an oblivious balls-and-bins hashing algorithm that relies on repeated applications of an oblivious sorting algorithm.

Other ORAM protocols relied on cuckoo hashing, and an oblivious Cuckoo hashing was proposed [GM11]. The original of obliviousness had flaws that were later fixed [CGLS17].

Unfortunately, all known oblivious hashing algorithms are fairly complex, requiring several applications of oblivious sorting.

To get around this, in the multi-server setting, it is common to have one server build the hash table and encrypts it, while a different server responds to queries about the (encrypted) table. Thus no single party is able to see both the transcript of the build procedure and subsequent queries [LO13,KM19].

Essentially any hashing scheme can be made oblivious in this way, and this eliminates the need for (expensive) oblivious sorting.

G Bloom Filter

A Bloom Filter is a standard data structure for performing set membership queries. We present it in Algorithm 2.

Algorithm 2 Bloom Filter

procedure BUILD(X, n, m, k, κ)
  Input: X a set of size n
  Input: m size of Bloom filter array
  Input: k number of hash functions
  Input: κ security parameter
  for i in $1, \ldots, k$ **do**
    $K_i \leftarrow \text{PRF.keygen}(\kappa)$
  end for
  $M = 0^m$ ▷ M is a bit-array
  for i in $1, \ldots, k$ **do**
    for j in $1, \ldots, n$ **do**
      $q_{i,j} = \text{PRF.Eval}(K_j, X_i) \bmod m$
      $M[q_{i,j}] = 1$
    end for
  end for
  return M, $K_1, \ldots, K_k$.
end procedure

procedure LOOKUP(x, M, $\{K_i\}_{i=1}^k$)
  Input: x an element to look up
  Input: M Bloom filter array of size m
  Input: $\{K_i\}$ hash function keys
  $b = 1$
  for i in $1, \ldots, k$ **do**
    $q_i = \text{PRF.Eval}(K_i, x) \bmod m$
    $b = \text{AND}(b, M[q_i])$
  end for
  return b
end procedure

Since the build algorithm explicitly sets $M[\text{PRF.Eval}(K_i, x_j)] = 1$ for all i and for all $x_j \in X$, if $x \in X$ then $\text{Lookup}(x, M, K, m)$ will return 1. There is however a possibility of false-positives, where $x \notin X$ but $\text{Lookup}(x, M, K, m)$ still returns 1.

Although it is difficult to calculate the exact false-positive rate for a Bloom filter [BGK+08,CRJ10], asymptotically, the false-positive rate for a Bloom filter with storing s elements is more straightforward [MU17][Chapter 5]. Storing a set of size s, in a Bloom filter of size m using k hash functions, the probability of false positives is

$$\Pr[\text{false positive}] \approx \left(\left(1 - e^{-\frac{ks}{m}}\right)^k\right).$$

If the Bloom Filter data structure is secret-shared between two parties, then Lookups will need to perform the AND operation inside a secure computation. Using pre-distributed randomness this can be achieved with $\mathcal{O}(k)$ communication.

H Cuckoo Hash Table

Cuckoo Hashing is a standard data Hash Table implementation. There are many variants–for concreteness and completeness we present a specific one below which would be used by our protocol.

If the "payload" is simply a single bit representing that the item is present in the table, a Cuckoo Hash Table also serves as a set membership protocol.

Cuckoo Hashing [PR01] is a dictionary implementation. Each element is stored in one of d locations in memory according to the index's evaluation on d distinct hash functions. Lookups therefore only require d memory accesses. With $d = \mathcal{O}(1)$ and a table of size $\mathcal{O}(n)$ the basic algorithm has a small, but non-negligible probability of build failure as some elements may not be able to be placed.

To reduce this failure probability, the cuckoo hash can include a "stash" of size s, which contains any elements that were not able to be placed in the table. For a constant s, the probability of failure becomes $\mathcal{O}(n^{-(s+1)})$ [KMW09,Mit09] In fact, with d hash functions, and a stash of constant size s, cuckoo hashing build fails with probability $\mathcal{O}\left(n^{(1-d)(s+1)}\right)$ [KMW09][Lemma 3.1]. For $s = \mathcal{O}(log(n))$ the failure probability is $\mathcal{O}(n^{-\frac{s}{2}})$.

Cuckoo hashing has been used in hierarchical ORAMs [KLO12,LO13,AKL+20,KM19]. However, building a Cuckoo Table *obliviously* in the context of an ORAM is challenging, and many works have observed flaws or unintended overheads in previous designs [GM11,CGLS17,FNO21].

For concreteness, Algorithm 3 outlines a basic cuckoo hashing scheme with $d = 2$. For consistency with the analysis of [Aum10] we use the version where the two hash functions map to different tables.

To allow compatibility with our protocols which perform execution on secret-shared data, we distinguish in the Lookup procedure between the variable which

determines the data's location in the table, z, and that used to verify which item in the table is correct, x. In our oblivious algorithms $z = \text{SISO-PRF.Eval}_{ABC}([k], [x])$ is known to the parties performing the Lookup, but x is secret-shared. In the case where the table is secret shared, the Lookup therefore requires 2 secure comparisons and also needs to execute the if statements in Lookup obliviously (*i.e.*, evaluate both branches). Similarly, to execute the stash Lookup when S and x are secret-shared, each comparison must be executed securely and each if statement evaluated obliviously.

I Proof of Theorem 3

Proof (Proof of Theorem 3).

Storage overhead: The total storage of the data structure is $|T| + |B|$. T has $\mathcal{O}(n)$ locations, each of size $\log(N)$ and B has $n\log(N)$ bits so the total space is $\Theta(n\log(N))$.

False-positive rate: Since Cuckoo Hash Tables have no false positives, the only way a false-positive can occur is during the Bloom Filter lookup. Given a Bloom Filter with k hash functions, and a table B of size $|B|$, storing s elements a standard analysis (e.g. [MU17][Chapter 5]) shows that the probability that a false positive occurs approaches

$$\left(1 - e^{-\frac{ks}{|B|}}\right)^k.$$

Here $k = \log(N)$, $s \leq \log(N)$ and $|B| = n\log(N)$. Therefore, the probability of a false positive is at most

$$\left(1 - e^{-\frac{\log^2(N)}{n\log(N)}}\right)^{\log(N)} = \left(1 - e^{-\frac{\log(N)}{n}}\right)^{\log(N)}$$

We wish that the failure probability be negligible in N, *i.e.*, $\Pr(\text{false positive}) \leq N^{-c_0}$ for any constant $c_0 > 0$ for sufficiently large N. This is equivalent to saying $\Pr(\text{false positive}) \leq (2^{-c_0})^{\log(N)}$, and setting $c_1 = 2^{-c_0}$ this is equivalent to saying that for all constants $0 < c_1 < 1$, $\Pr(\text{false positive}) \leq c_1^{log(N)}$.

$$\left(1 - e^{-\frac{\log(N)}{n}}\right)^{\log(N)} \leq c_1^{\log(N)} \Leftrightarrow \frac{\log(N)}{n} \leq -\ln(1 - c_1)$$

$-\ln(1 - c_1)$ will be a constant greater than 0. Since $n = \omega(\log(N))$ this is satisfied for any constant $0 < c_1 < 1$, for sufficiently large N.

False negatives: Any item in the set will be stored in either the Cuckoo Hash table or the Bloom Filter. Since neither the Cuckoo hash table nor the Bloom Filter have false negatives, every item in the set will be found.

Build failure: The build will only fail if $|S| > \log(N)$. For $n = \omega(\log(N))$ and a stash of size $\log(N)$ Cuckoo Hashing with a stash of size $\log(N)$ will succeed except with probability negligible in N [Nob21].

Algorithm 3 CuckooHT: Cuckoo Hash Table

procedure KEYGEN
  $K_1 \leftarrow \text{PRF.Keygen}(\kappa)$
  $K_2 \leftarrow \text{PRF.Keygen}(\kappa)$
  return K
end procedure

procedure INSERT(T_1, T_2, S, x, y, b, k_1, k_2, j)
  Input: Two tables T_1, T_2
  Input: A stash, S
  Input: A key-value pair (x, y)
  Input: A bit, b, indicating which table to try first
  Input: Keys for two hash functions k_1, k_2
  Input: A counter, j, indicating how many attempts have been made
  if $j > \log(N)$ **then**
    $S.\text{append}((x, y))$
    return
  end if
  $q \leftarrow \text{PRF.Eval}(k_b, x)$.
  $(\bar{x}, \bar{y}) \leftarrow (T_b)_q$
  $(T_b)_q \leftarrow (x, y)$
  if $\bar{x} \neq \perp$ **then**
    $\text{Insert}(T_1, T_2, \bar{x}, \bar{y}, (3 - b), k_1, k_2, j + 1)$
  end if
end procedure

procedure BUILD(X, Y, K, n, m)
  Input: A set of keys, X, with $|X| = n$
  Input: A set of values, Y, with $|Y| = n$
  Input: A set of PRF keys, K, with $|K| = 2$
  Input: A table size, m
  S is initialized to an empty list
  $T_1 \leftarrow (\perp, \perp)^{\frac{m}{2}}$
  $T_2 \leftarrow (\perp, \perp)^{\frac{m}{2}}$
  for i in $1, \ldots, n$ **do**
    $\text{Insert}(\text{T}_1, \text{T}_2, \text{X}_\text{i}, \text{Y}_\text{i}, 1, \text{K}_1, \text{K}_2, 0)$
  end for
  return $T_1 || T_2$, S
end procedure

procedure LOOKUPMAIN(z, x, T, k_1, k_2) ▷ Lookup in the table, but skip the stash
  Input: A key, z, that was used to build the table
  Input: A key, x, that is stored with each data element
  Input: A pair of tables T
  Input: A pair of keys k_1, k_2
  $q_1 \leftarrow \text{PRF.Eval}(k_1, z)$
  $q_2 \leftarrow \text{PRF.Eval}(k_2, z)$
  if $(T_1)_{q_1}.\text{left} = x$ **then**
    return $(T_1)_{q_1}.\text{right}$
  else if $(T_2)_{q_2}.\text{left} = x$ **then**
    return $(T_2)_{q_2}.\text{right}$
  else
    return $\perp$
  end if
end procedure

procedure LOOKUPSTASH(x, S) ▷ Lookup just in the stash
  Input: A key x
  Input: The stash S
  $y \leftarrow \perp$
  for i in $1, \ldots, |S|$ **do**
    if $S_i.\text{left} = x$ **then**
      $y = S_i.right$
    end if
  end for
  return y
end procedure

Lookup cost: The amount of memory accessed for the lookup is $2\log(N)$ bits for searching the Cuckoo Tables, and $k = \log(N)$ bits for searching the Bloom filter, so the total about of memory accessed is $\mathcal{O}(\log(N))$.

J Deferred proofs of Π_{OSet} communication costs

Theorem 13. *Protocol $\Pi_{OSet.Build}$ (Figure 4) requires $\mathcal{O}(\kappa n)$ communication. In particular it requires n calls to $\mathcal{F}_{ABB}$PRP_eval.*

Proof. Generating the secret key requires no communication. Sharing the hash functions requires sending $\mathcal{O}(\log(N)) = o(n)$ hash functions, each which can be represented with κ bits, so $\mathcal{O}(\kappa n)$ bits total. Step 3 requires n calls to $\mathcal{F}_{ABB}$PRP_eval. each of which costs $\mathcal{O}(\kappa + \log N) = \mathcal{O}(\kappa)$ bits of communication, so again $\mathcal{O}(\kappa n)$ bits total. Outputting these results to P_1 requires outputting n messages, each of size $\mathcal{O}(\kappa)$, and the communication cost to output a bit is $\mathcal{O}(1)$, so the total cost is again $\mathcal{O}(\kappa n)$.

Constructing the Cuckoo Hash table and Bloom Filter is achieved locally by P_1 so needs no communication. Inputting the Cuckoo Table requires inputting $2m = \mathcal{O}(n)$ messages, each of size $\mathcal{O}(\kappa)$, each bit of which requires $\mathcal{O}(1)$ communication, so $\mathcal{O}(\kappa n)$ communication total. The bloom filter is of length $n \log N = o(n\kappa)$ bits, so inputting it requires $o(n\kappa)$ communication.

Each step requires $\mathcal{O}(n\kappa)$ communication, and since there is a constant number of steps, the total communication for a Build is $\mathcal{O}(\kappa n)$ communication.

Theorem 14. *Protocol $\Pi_{OSet.Query}$ (Figure 4) requires $\mathcal{O}(\kappa)$ communication. In particular it requires 1 call to $\mathcal{F}_{ABB}$PRP_eval.*

Proof. The protocol begins with one (and the only) call to $\mathcal{F}_{ABB}$PRP_eval. This costs $\mathcal{O}(\kappa + \log(N)) = \mathcal{O}(\kappa)$ bits of communication. Reveal the output to P_2 and P_3 requires revealing 2 messages of size $\log(N) = o(\kappa)$, which requires $o(\kappa)$ communication total. P_2 and P_3's local examination of the *queries* object needs no communication. Securely checking the Cuckoo Hash table requres 2 calls to F_{ABB}.InputFromSharing on inputs of size $\log(N) = o(\kappa)$, 2 calls to F_{ABB}.Equal also on inputs of size $\log(N) = o(\kappa)$ and a single secure OR of cost $\mathcal{O}(1)$. Therefore the total cost of querying the Cuckoo Hash table is $\mathcal{O}(\log(N)) = o(\kappa)$. Querying the Bloom Filter requires $\log(N)$ calls to F_{ABB}.InputFromSharing on inputs of size 1, and $\log(N)$ calls to $\mathcal{F}_{ABB}$.AND. This sums to $\mathcal{O}(\log(N)) = o(\kappa)$ communication. The final OR requires $\mathcal{O}(1)$ communication. Since there are a constant number of steps, each costing $\mathcal{O}(\kappa)$ bits, the total communication cost of a query is $\mathcal{O}(\kappa)$ bits.

K Deferred proofs of $\Pi_{OHTable}$ communication costs

Theorem 15. *$\Pi_{OHTable}$.Build (Figure 6) requires $\mathcal{O}(\kappa n + Dn + DT)$ communication. In particular, it requires $2n$ calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Proof. $\mathcal{F}_{OSet}$.Build can be implemented $\mathcal{O}(\kappa n)$ communication and n calls to $\mathcal{F}_{ABB}$.PRP_eval (Theorem 13). Inputting the PRP key requires $\mathcal{O}(\kappa)$ communication. Resetting the values of duplicates (step 3a) requires inputting at most n values of size $\log(4N)$ and n of size D, for a total cost of $\mathcal{O}(nD)$. There are n more secure calls to $\mathcal{F}_{ABB}$.PRP_eval, needing a total of $\mathcal{O}(\kappa n)$ communication. Inputting the dummies in step 4 requires T iterations, each needing communication $\mathcal{O}(\log(4N) + D) = \mathcal{O}(D)$, or $\mathcal{O}(TD)$ total. The shuffle shuffles arrays of length $n + T$, where each is of size $\mathcal{O}(D)$, leading to $\mathcal{O}(Dn + DT))$ communication. Revealing $\hat{Q}$ to P_2 and P_3 requires $\mathcal{O}((n + T)\log(4N)) = \mathcal{O}((n + T)D)$ communication. Therefore the entire protocol requires $\mathcal{O}(\kappa n + Dn + DT)$ communication.

Theorem 16. *$\Pi_{OHTable}$.Query (Figure 6) requires $\mathcal{O}(\kappa)$ communication and, in particular, at most 2 calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Proof. Calling $\mathcal{F}_{OSet}$.Query requires $\mathcal{O}(\kappa)$ communication and 1 call to $\mathcal{F}_{ABB}$.PRP_eval. If x was queried before, the protocol is done. Otherwise an additional $\mathcal{O}(\log(4N)) = \mathcal{O}(\kappa)$ communication is required to input the dummy index and $\mathcal{O}(\log(4N)) = \mathcal{O}(\kappa)$ is needed to set $[\![x_{used}]\!]$ to the appropriate value. Securely evaluating the PRP on $[\![x_{used}]\!]$ requires an additional call to $\mathcal{F}_{ABB}$.PRP_eval and $\mathcal{O}(\kappa)$ communication. Revealing the result requires $\mathcal{O}(\log(4N)) = \mathcal{O}(\kappa)$ communication. Setting $[\![res]\!]$ is free (since the data is already in the ABB). The total cost is therefore $\mathcal{O}(\kappa)$.

Theorem 17. *$\Pi_{OHTable}$.Extract (Figure 6) requires $\mathcal{O}((n + T - t)D)$ communication and no calls to $\mathcal{F}_{ABB}$.PRP_eval.*

Proof. The protocol has $n + T - t$ iterations of a loop. Each iteration consists of a secure equality on D bits, costing $\mathcal{O}(D)$ communication, an IfThenElse on $\log(4N)$ bits, costing $\mathcal{O}(\log(N))$ communication and a secure renaming, which is free. Therefore the total cost is $\mathcal{O}((n + T - t)D)$.