3-Party Secure Computation for RAMs: Optimal and Concretely Efficient

Atsunori Ichikawa[1], Ilan Komargodski[2], Koki Hamada[1], Ryo Kikuchi[1], and Dai Ikarashi[1]

[1] NTT Social Informatics Laboratories, Tokyo, Japan
atsunori.ichikawa@ntt.com, koki.hamada@ntt.com, ryo.kikuchi@ntt.com, dai.ikarashi@ntt.com
[2] The Hebrew University of Jerusalem, and NTT Research, Israel
ilank@cs.huji.ac.il

Abstract. A distributed oblivious RAM (DORAM) is a method for accessing a secret-shared memory while hiding the accessed locations. DORAMs are the key tool for secure multiparty computation (MPC) for RAM programs that avoids expensive RAM-to-circuit transformations.

We present new and improved 3-party DORAM protocols. For a logical memory of size N and for each logical operation, our DORAM requires $O(\log N)$ local CPU computation steps. This is known to be asymptotically optimal. Our DORAM satisfies passive security in the honest majority setting. Our technique results with concretely-efficient protocols and does not use expensive cryptography (such as re-randomizable or homomorphic encryption). Specifically, our DORAM is 25X faster than the known most efficient DORAM in the same setting.

Lastly, we extend our technique to handle malicious attackers at the expense of using slightly larger blocks (i.e., $\omega(\log^2 N)$ vs. $\Omega(\log N)$). To the best of our knowledge, this is the first concretely-efficient maliciously secure DORAM.

Technically, our construction relies on a novel concretely-efficient 3-party oblivious permutation protocol. We combine it with efficient non-oblivious hashing techniques (i.e., Cuckoo hashing) to get a distributed oblivious hash table. From this, we build a full-fledged DORAM using a distributed variant of the hierarchical approach of Goldreich and Ostrovsky (J. ACM '96). These ideas, and especially the permutation protocol, are of independent interest.

1 Introduction

Secure multiparty computation (MPC) is a method that enables mutually distrustful parties to jointly compute an arbitrary function over their private inputs. Since breakthrough feasibility results in the 80s, the quest for practically efficient MPC protocols is a central research area in cryptography. Efficiency is measures in terms of local computation and/or communication, as a function of the size of the representation of the function that needs to be computed.

There are several common ways to represent computation, e.g., the circuit model or Random Access Memory (RAM) model. Any function can be computed in either of the models and a representation in one model can be translated to the other. However, such translations have a cost: a RAM program of size N can be turned into a circuit of size $O(N^3 \log N)$ [52]. Therefore, due to efficiency reasons, it would be highly desirable to be able to perform secure computation for RAM programs, *directly*.

This is challenging because MPC protocols need to guarantee, in particular, that the running time, memory accesses and communication patterns of the participants, do not depend on their private inputs. The circuit model guarantees these properties for free as circuits can be computed in a gate-by-gate fashion, independently of the inputs. In general, RAM programs do not have these features and therefore some extra work is needed.

There is a generic way to turn any RAM program into another that computes the same functionality but whose memory accesses do not reveal anything about the program's secret inputs. This is called an *Oblivious RAM* (ORAM), originally proposed by Goldreich and Ostrovsky [26, 48, 27]. The traditional setting for ORAMs is the client-server one. That is, a large memory is stored on an untrusted server and a client can make accesses to it using a small trusted memory. Ostrovsky and Shoup [49] observed that by simulating the client of an ORAM using traditional circuit-based MPC protocol, one can generically get an MPC for RAM programs. However, designing an efficient ORAM with a client that is "compatible" with circuit-based MPC is not at all obvious and has been (so far) sub-optimal in terms of the efficiency of the resulting protocol (see, e.g., [57]).

Due to the inherent inefficiency of circuit-based MPC for certain computations, there has been significant efforts in the last decade in building efficient MPCs for RAM programs, for example [44, 21, 57, 57, 12, 8, 23]. By now, due to its relation to oblivious simulation, the common terminology for this problem is **Distributed Oblivious RAM** (DORAM)—informally, this is a protocol that allows parties to collectively maintain and perform reads/writes on a memory (a formal definition appears in Section 2.2).

The complexity measure of DORAMs of interest to us is their *computational overhead*. That is, the maximal amount of CPU instructions[3] performed by each of the parties when serving a single logical request.[4] Some prior works measure *bandwidth* overhead which accounts only for the maximal amount of bits communicated between the parties. Computational overhead is harder to optimize since an upper bound on the computational overhead implies an upper bound on the communication overhead. The other direction is not necessarily true; indeed, some prior works (e.g., [23]) optimize communication overhead at the expense of

[3] We model parties as RAM machines that can perform word-level addition and standard Boolean operations at unit cost.

[4] As commonly done in this line of works, we sometimes settle for overhead in an amortized sense, that is, we measure the average overhead over a sequence of requests. Often, such schemes can be made worst-case ("de-amortized") using known techniques [49, 5].

increased computational overhead. Thus, we choose the more stringent measure. This is particularly important if we aim for a concretely efficient and practically useful DORAMs.

In this work, we focus on the honest majority setting and more specifically on the 3-party setting where at most one server is corrupted. We note that there are several schemes in the 2-party setting (e.g., [44, 57, 20]), but due to the nature of the dishonest majority setting, existing techniques result with asymptotically and concretely less efficient schemes than in the 3-party honest majority setting.

A variant of a scheme due to Lu and Ostrovsky [44], suggested by Faber et al. [21],[5] gives a (3-party) DORAM with $O(\log N)$ computational overhead with block size $\Omega(\log N)$. While the asymptotical overhead of this construction is optimal, the concrete efficiency is quite poor. The reason is that their compiler requires the parties to securely and jointly compute a linear number of encryptions once in a while (which requires a circuit-based secure computation protocol of AES computation).

Later works attempt to present concretely efficient DORAMs. Wang et al. [57] and Faber et al. [21] proposed DORAMs with $O(\log N)$ computational overhead, but their block size is $\Omega(\log^2 N)$ which is less standard. Bunn et al. [8] constructed a DORAM with small concrete constants but poor asymptotic overhead ($\Omega(\sqrt{N})$). Most recently, Falk et al. [23] reduced the large constant factor of [44, 21] and achieved a scheme with $O(\log^2 N)$ computational overhead (and logarithmic communication overhead).

Moreover, all of the above scheme only guarantee security against a passive attacker, i.e., any single server cannot learn any non-trivial information about the others' inputs, as long as it follows the prescribed protocol. There are generic methods to boost security to the more standard setting of active security, where security holds even if a rouge server arbitrarily deviates from the protocol. However, these techniques do not preserve efficiency.

The current state of the art for 3-party DORAMs is summarized in Table 1. This brings us the the main problems that we consider in this work:

Is there a 3-party DORAM which is asymptotically optimal in terms of computational overhead and concretely efficient? Additionally, is there an efficient actively secure DORAM?

1.1 Our Contributions

An optimal 3-party DORAM. We present an asymptotically optimal and concretely efficient 3-party DORAM in the honest majority setting. Specifically, our DORAM has computational overhead of $O(\log N)$ and the hidden constant is rather small. The block size of our construction is $\Omega(\log N)$. Our

[5] The protocol of Lu and Ostrovsky [44] is in the multi-party setting where there are two non-communicating servers and a single trusted lightweight client (see Section 1.3). Faber et al. [21] observed that the client in [44]'s scheme can be efficiently simulated by an MPC.

Ref.	Communication	Computation	Security	Block size	Hidden const.
[49]	$O(\log^3 N)$	$O(\log^3 N)$	Passive	$\Omega(\log N)$	Large
[44, 21]	$O(\log N)$	$O(\log N)$	Passive	$\Omega(\log N)$	Large
[57, 21]	$O(\log N)$	$O(\log N)$	Passive	$\Omega(\log^2 N)$	Large
[8]	$O(\sqrt{N})$	$O(\sqrt{N})$	Passive	$\Omega(\log N)$	Small
[23]	$O(\log N)$	$O(\log^2 N)$	Passive	$\Omega(\log N)$	Small
Our	$O(\log N)$	$O(\log N)$	Passive	$\Omega(\log N)$	Small
Our	$O(\log N)$	$O(\log N)$	Active	$\omega(\log^2 N)$	Small

Table 1. Summary of known 3-party DORAMs in the honest majority setting together with our own schemes. The first column points to the paper that obtained the DORAM. The second column states the communication overhead of the proposed construction. The third column states the computational overhead of the proposed construction. The fourth column states whether the security guarantee is for passive or active attackers. The fifth column mentions the block size used in the construction. Lastly, the sixth column states whether the hidden constants are considered large or small.

DORAM requires (amortized) only $4 \log N$ oblivious pseudorandom function (OPRF) calls per access. This is about 2 times greater than that of the DORAM of Falk et al. [23],[6] but it is significantly more efficient than the known optimal DORAM of Lu and Ostrovsky [44, 21] that requires at least $100 \log N$ calls per access (see Appendix A for more details). This protocol is secure against passive (honest-but-curious) attackers.

A distributed oblivious permutation: Our main technical novelty is a new (concretely efficient and asymptotically optimal) 3-party oblivious permutation protocol. Our protocol can apply any permutation to the data with communication of $4nb + 2n\lceil\log n\rceil$ bits and $12nb + 2n\lceil\log n\rceil$ local CPU computation steps where n is the number of data elements to be shuffled and b is the bit-length of each data element. We also construct a procedure to invert that permutation. This procedure requires $8nb+2n\lceil\log n\rceil$ bits of communication and $19nb + 2n\lceil\log n\rceil$ steps of local computation.

A distributed oblivious hash table: Our DORAM construction is modular and, at a high level, is reminiscent of the hierarchical ORAM technique of Goldreich and Ostrovsky [27]. Recall that [27]'s hierarchical method basically reduces the problem of maintaining a memory to the problem of building a static hash table (supporting only lookups after an initial build). To this end, we implement a concretely efficient *distributed* oblivious hashing scheme. This is the first concretely efficient and asymptotically optimal distributed oblivious hash table construction. To store n data blocks of size b bits into a distributed hash table, each party needs to perform at most $O(n \cdot (b + \lceil\log n\rceil))$ local CPU computation steps, and our lookup protocol requires $O(b) + O(\sigma \cdot b)$ local CPU computation

[6] Here, we emphasize again that the DORAM of Falk et al. [23] requires $\Omega(\log^2 N)$ *computational* cost in addition to the communication cost. We only have $O(\log N)$ computational cost.

steps, where the first term for a lookup in a main table and the second for a linear scan of a σ-size stash. The storage size of the hash table is $O(nb)$. We obtain our distributed oblivious hash table by first randomly permuting the data to be hashed (using the above-mentioned permutation protocol) and then simply invoking an off-the-shelf (non-oblivious) distributed hashing technique.

Active security. We extend our passively secure schemes from above to be *actively* secure, without hurting efficiency, except that we rely on somewhat larger blocks. Specifically, we get a 3-party DORAM with $O(\log N)$ computational overhead and block size $\omega(\log^2 N)$. As far as we know, this is the first result achieving active security for DORAM with practical efficiency guarantees. We do not know how to achieve similar concrete efficiency guarantees with logarithmic size blocks and we leave it as an exciting open problem.

1.2 Technical Overview

Before showing the fundamental idea of our schemes, we first revisit the optimal DORAM of Lu and Ostrivsky [44, 21]. Their DORAM consists of a hierarchy of *permuted arrays*, i.e., oblivious hash tables, that are managed by multi-party protocols while hiding access patterns. The fundamental idea of their oblivious hashing (which comes from the 2-server setting [44]) is as follows. One of the two servers is *the permuter*, and set other is *the storage*. The storage sends all data that should be permuted to the permuter while *rerandomizing*, and the permuter constructs a hash table consisting of the data. The permuter sends the hash table to the storage while rerandomizing. Now, the storage can explore the table with a (randomized) query.

Though the storage can observe the access patterns on the table directly in the above scheme, the access patterns achieve one-shot obliviousness against the storage since it does not know the *permutation* for building the table. In other words, the table seems to be shuffled from the storage's point of view, and hence the single-short exploration of elements is randomized. On the other hand, since the permuter does not observe any access on the table, it never knows the access patterns even if it knows how to construct the table.

Due to the ingenuity of the server role splitting, they achieved optimal DORAM with optimal oblivious hashing. However, as Falk et al. [23] pointed out, while this DORAM is *asymptotically* optimal it is *not practically* efficient because of the large frequency of rerandomizations required. Their rerandomization can be implemented by *oblivious pseudorandom function* (OPRF) in the context of secure multiparty computation, but as mentioned in [23], the DORAM of Lu and Ostrovsky requires at least $100 \log N$ OPRFs per access. Falk et al. improved this by a factor of 50, at the expense of increased local computation overhead—$O(\log^2 N)$ local hash function evaluations per access.

An oblivious distributed permutation protocol. Our starting point is the idea of role splitting, but with a novel modification that maintains optimality and greatly improves practical efficiency. Our fundamental idea is as follows. Set the role of one of the *three* servers as the permuter; this server knows a permutation for hashing. The other two servers will be the storage that holds data in a secret-shared form. The servers obliviously compute hash values of all secret-shared data (which can be implemented by ORPFs) and reveal them to the permuter. The permuter calculates a permutation that sorts the data to make a hash table. The servers run a role-asymmetric oblivious permutation to apply the above permutation to the data obliviously. As the output of this protocol, the two storages obtain a hash table in secret-shared form. Now, the storages can explore the table with a secret-shared query.

By the description above, only one round of OPRF evaluations is required to build a hash table. Our permutation protocol is for 3-party computation, and if one party knows a permutation, it can apply the permutation to a secret-shared array in linear time while keeping the permutation secret from the other two parties (see Section 4.1 for more details).

From an oblivious distributed permutation to a DORAM. We obtain a DORAM using only $4 \log N$ OPRF evaluations and optimal computational complexity per access. This is obtained via the following very useful observations: (1) given an oblivious permutation protocol, there is an extremely efficient way to get an oblivious distributed hash table, and (2) given the latter, we can adapt the hierarchical ORAM framework (or its optimizations) to the distributed setting. We elaborate on both bullets next. First, we observe that the shuffle-sort paradigm can be applied to hashing: if data is randomly shuffled, we can invoke an insecure oblivious hashing algorithm. This allows us to completely get rid of complicated (distributed) oblivious hashing approaches by first shuffling the input and then invoking a simple hashing procedure. Concretely, we use (plain) cuckoo hashing with a stash [36] to achieve constant lookup time (ignoring scanning a logarithmic-size stash, which we will do once per logical access).

Once we have obtained our distributed hash table, we plug it into the hierarchical ORAM setting, while extending it to the distributed setting. I.e., we implement every level in the hierarchy with an oblivious distributed hash table, as above, and where each level can hold twice more elements than its previous level. The stashes from all levels are merged into one common stash and scanning it is done once per lookup.

Maliciously secure DORAM. In our permutation protocol roles of parties are asymmetric, and in particular, the permuter has complete control over the chosen permutation. Thus, it is non-trivial to extend our ideas to the malicious setting.

To this end, we augment our permutation evaluation protocol to check that *all elements in a hash table are actually in their correct cell.* The key insight is that the correctness of a hash table can be evaluated by hash values of elements calculated by OPRF whose correctness (in turn) can be guaranteed using a known actively secure MPC framework. That is, comparing the virtual address (obtained obliviously by OPRF) and the real address (known by all parties) of each element in a hash table, the correctness of the table can be achieved regardless of whether the permuter is the adversary or not. We observe that any attack of the permuter can be translated into some form of an *additive attack* [24, 25]. Thus, to achieve malicious security, we incorporate the permutation evaluation into an efficient evaluation process of security-with-abort MPC (e.g., [13, 32, 35]) and only need to deal with "additive attacks". This makes our hash table validation have almost no effect on overall efficiency.

Organization

We describe our passively secure oblivious permutation and oblivious hashing in Section 4. We combine them to get our passively secure DORAM in Section 5. The actively secure extension is described in Section 6.

1.3 Related Works

Standard ORAM. The first ORAM, i.e., in the client-server model, was introduced by Goldreich and Ostrovsky [26, 48, 27]. Their best scheme had poly-logarithmic overhead in the memory size. The also proved a lower bound for a restricted class of schemes saying that logarithmic overhead is necessary [7]. Larsen and Nielsen [40] (see also [34, 37]) showed that logarithmic overhead is inherent for all schemes as long as they support "online" accesses.

There have been significant efforts to improve the overhead of ORAMs, both asymptotically and concretely [28, 55, 38, 56, 57, 11, 51, 3, 5]. A few years ago, Asharov et al. [3] (building on Patel et al. [51]) got an asymptotically optimal ORAM with logarithmic overhead. The constants underlying their scheme are enormous (due to the use of certain expander graphs; see also [19]) which makes it far from being practically useful. The best ORAM for practical purposes is still based on Path ORAM [56], a technique that suffers from $\log^2$ overhead in the memory size.

Multi-server ORAM. Another model that received significant attention is the *multi-server* ORAM setting. Here, there are multiple servers and a client. The client can communicate with each server but the servers cannot communicate amongst themselves. As in the client-server setting of ORAM, the client has a small trusted memory. The adversarial model is that only some fraction of the servers are corrupted, making it possible for more efficient solutions than in the standard single-server ORAM setting. This model was first introduced by Lu and Ostrovsky [44] who proposed a two-server ORAM with logarithmic communication overhead. By now, we have schemes with the same asymptotic overhead in the single-server setting [3]. Assuming larger block size, Kushilevitz and Mour [39] achieved a multi-server ORAM with sub-logarithmic communication from the client but poly-logarithmic work by the servers. Hoang et al. [30] proposed a multi-server ORAM with constant client-server communication overhead using secret sharing and MPC techniques. Lower bounds for this setting were proven by Larsen et al. [41].

Actively secure ORAM. Since Ren et al. [53] introduced an actively secure ORAM in the standard model, various actively secure ORAMs have been proposed. Devadas et al. [18] proposed a generic construction of ORAMs that guarantees the correctness of metadata in the standard model. Also, Blass et al. [6] and Maffei et al. [45] proposed standard ORAMs (with multiple clients) that force the dishonest server or clients to behave correctly. Hoang et al. [29] construct a maliciously secure multi-server ORAM that detects servers that deviate from the protocol.

However, in the distributed model, no concrete construction for actively secure DORAM has been proposed.

2 Preliminaries

For an integer $n \in \mathbb{N}$, we denote by $[n]$ the set $\{0, \dots, n-1\}$. By $\|$ we denote the operation of string concatenation.

Throughout this work, the security parameter is denoted λ, and it is given as input to algorithms in unary (i.e., as 1^λ). A function $\mathsf{negl}\colon \mathbb{N} \to \mathbb{R}^+$ is *negligible* if for every constant $c > 0$ there exists an integer N_c such that $\mathsf{negl}(\lambda) < \lambda^{-c}$ for all $\lambda > N_c$. Two sequences of random variables $X = \{X_\lambda\}_{\lambda\in\mathbb{N}}$ and $Y = \{Y_\lambda\}_{\lambda\in\mathbb{N}}$ are *computationally indistinguishable* if for any probabilistic polynomial-time algorithm $\mathcal{A}$, there exists a negligible function $\mathsf{negl}(\cdot)$ such that $\left|\Pr[\mathcal{A}(1^\lambda, X_\lambda) = 1] - \Pr[\mathcal{A}(1^\lambda, Y_\lambda) = 1]\right| \le \mathsf{negl}(\lambda)$ for all $\lambda \in \mathbb{N}$. We say that $X \equiv Y$ for such two sequences if they define *identical* random variables for every $\lambda \in \mathbb{N}$.

Definition 2.1 (Pseudorandom functions (PRFs)). *Let F be an efficiently computable function family indexed by keys $\mathsf{sk} \in \{0,1\}^\lambda$, where each F_{sk} takes as input a value $x \in \{0,1\}^{n(\lambda)}$ and outputs a value $y \in \{0,1\}^{m(\lambda)}$. A function family F is a secure* pseudorandom function *(PRF) if for every (non-uniform) probabilistic polynomial-time algorithm $\mathcal{A}$, there is a negligible function $\mathsf{negl}(\cdot)$ such that*

$$\left| \Pr_{\mathsf{sk} \leftarrow\$ \{0,1\}^\lambda}\left[\mathcal{A}^{\mathsf{F}_{\mathsf{sk}}(\cdot)}(1^\lambda) = 1\right] - \Pr_{u \leftarrow\$ U_\lambda}\left[\mathcal{A}^{u(\cdot)}(1^\lambda) = 1\right] \right| \le \mathsf{negl}(\lambda),$$

for all $\lambda \in \mathbb{N}$, where U_λ is the set of all functions mapping $\{0,1\}^{n(\lambda)}$ to $\{0,1\}^{m(\lambda)}$.

It is known that one-way functions are existentially equivalent to PRFs for any polynomial $n(\cdot)$ and $m(\cdot)$ [31, 46]. Our construction will employ PRFs in several places and we present each part modularly with its own PRF, but note that the whole DORAM construction can be implemented with a single PRF from which we can implicitly derive all other PRFs.

2.1 Secret Sharing Schemes

A (threshold) secret sharing scheme (SSS) is a cryptographic primitive that allows to "split" a secret between a collection of parties such that a set of parties of some predefined cardinality can reconstruct the secret. A "piece" that is held by a party is called a *share*. When a secret is split into m shares and reconstructed with at least $t+1$ shares, we refer to such a scenario as (t, m)-SSS.

To share and reconstruct a secret, we introduce three functionalities as follows. We use the notations $[\![\cdot]\!]_i$ to represent the share of party $i \in [m]$.

- $\mathcal{F}_{\text{SHARE}}$ receives a secret s and distributes the shares among the parties; share $[\![s]\!]_i$ is sent to party i.
- $\mathcal{F}_{\text{REVEAL}}$ receives shares $[\![s]\!]_i$ from at least $t+1$ parties, recovers the secret s, and sends it to all parties.
- $\mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}$ behaves the same as $\mathcal{F}_{\text{REVEAL}}$ except that it sends the recovered secret s only to parties $\in \mathcal{P}$.

Also, we use the notations $\langle\cdot\rangle_{i\in\{0,1\}}$ to represent the shares in a $(1,2)$-*additive* SSS, i.e., $a = \langle a\rangle_0 + \langle a\rangle_1$ for any secret a. Under Shamir's SSS [54] or the replicated SSS [33], any two parties $P_i, P_{i+1 \bmod 3}$ can convert their shares $[\![s]\!]_i, [\![s]\!]_{i+1 \bmod 3}$ to $\langle s\rangle_0, \langle s\rangle_1$ by performing local computation.

We extend the notation to sharing arrays. For an array $\mathbf{A}$ of length X, we denote its x-th element by $\mathbf{A}[x]$. When secret sharing such an array, we denote its sharing by $[\![\mathbf{A}]\!] := ([\![a_0]\!], \dots, [\![a_{X-1}]\!])$ and $\langle\mathbf{A}\rangle = (\langle a_0\rangle, \dots, \langle a_{X-1}\rangle)$, where $a_x = \mathbf{A}[x]$.

For concreteness and clarity of this work, we assume that all shares $[\![s]\!]$ are of the $(1,3)$-replicated SSS on the extension field $\mathbb{Z}_{2^\ell}$ as [2, 15], i.e., for any $s = \sum_{i=0}^{\ell-1} s_i 2^i$; $s_i \in \mathbb{Z}_2$, its share is of the form $[\![s]\!] = \sum_{i=0}^{\ell-1} [\![s_i]\!] 2^i$. In this setup, all shares have the following properties.

Linear homomorphism. The replicated SSS [33], by definition, support share-to-share addition by performing only local operations on each parties shares. That is, for any $a, b, c \in \mathbb{Z}_{2^\ell}$, without any interaction, the parties can compute

$$[\![a]\!] + [\![b]\!] = [\![a+b]\!] \text{ and } c \times [\![a]\!] = [\![ca]\!].$$

We extend the above notation and operations to arrays of secrets. For any arrays $[\![\mathbf{A}]\!] = ([\![a_0]\!], \dots, [\![a_{X-1}]\!])$ and $[\![\mathbf{B}]\!] = ([\![b_0]\!], \dots, [\![b_{X-1}]\!])$ where $a_i, b_i \in \mathbb{Z}_{2^\ell}$, we denote entry-wise addition as $[\![\mathbf{A}]\!] + [\![\mathbf{B}]\!] := ([\![a_0]\!] + [\![b_0]\!], \dots, [\![a_{X-1}]\!] + [\![b_{X-1}]\!])$ and entry-wise multiplication by a scalar as $c \times [\![\mathbf{A}]\!] := (c \times [\![a_0]\!], \dots, c \times [\![a_{X-1}]\!])$.

Bit-decomposition. Since all shares are of the form $[\![s]\!] = \sum_{i=0}^{\ell-1} [\![s_i]\!] 2^i$ where each $[\![s_i]\!]$ is a share of the replicated SSS on $\mathbb{Z}_2$, parties can perform the bit-decomposition operation $[\![s]\!] \to ([\![s_{\ell-1}]\!], \dots, [\![s_0]\!])$ by their local conversion.

2.2 Distributed Oblivious RAM

A RAM consists of a memory of N cells and each cell is of size w bits and it allows for "clients" to perform read and write operations of the form $(\mathsf{op}, \mathsf{addr}, \mathsf{d})$, where $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{d} \in \{0,1\}^w \cup \{\perp\}$. If $\mathsf{op} = \mathsf{read}$, then $\mathsf{d} = \perp$ and the returned value is the content of the block located in logical address addr in the memory. If $\mathsf{op} = \mathsf{write}$, then the memory data in logical address addr is updated to d. We can think of this as an ideal (reactive) functionality $\mathcal{F}_{\mathrm{RAM}}$ that supports the following operation:

$\underline{\mathcal{F}_{\mathrm{RAM}}}$:

- $v \leftarrow \textsc{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{d})$: The input is an operation $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, a key $\mathsf{addr} \in [N]$, and a value $\mathsf{d} \in \{0,1\}^w \cup \{\perp\}$. An internal size N array $\mathbf{X}$, initialized to all 0s, is maintained. The procedure does:
  1. If $\mathsf{op} = \mathsf{read}$, then set $\mathsf{d}^* = \mathbf{X}[\mathsf{addr}]$.
  2. If $\mathsf{op} = \mathsf{write}$, then set $\mathbf{X}[\mathsf{addr}] = \mathsf{d}$ and $\mathsf{d}^* = \mathsf{d}$.
  3. Output d^*.

Distributed oblivious simulation. Our goal is to simulate a RAM correctly while guaranteeing the standard security notion of secure multi-party computation. Towards this goal, we have m servers that can communicate between themselves over a fully connected network in synchronous rounds of communication. Each server can further perform arbitrary local computation between rounds. We model each server machine as a RAM. The view of each machine includes the contents of its own memory and the contents of the incoming messages, where the latter include addresses of memory cells to access. Such a secure system is called *Distributed Oblivious RAM* (DORAM). The security guarantee stipulates that the view of a colluding subset of dishonest servers cannot learn anything about the computation being performed, except what is absolutely necessary (e.g., the length of the computation). We shall consider a passive (semi-honest) or active (malicious) adversary who controls up to $t < m$ servers.

We shall define distributed oblivious simulation with respect to an arbitrary (possibly reactive or stateful) functionality. The definition for the RAM functionality will be implied as a special case. For concreteness, it is convenient (though not necessary) to imagine that the input and RAM state are secret-shared between the servers, that is, each party holds one out of m shares of the RAM, and operations from a client are also written to each server in a secret shared fashion. We follow the real-ideal paradigm by defining two "worlds" and requiring that they are indistinguishable (following, e.g., Canetti [9]).

Definition 2.2 (View). *The view of party i consists of its auxiliary input and randomness followed by the honest input and all the messages sent and received by the party during the computation. Since we model parties as RAMs, the incoming and outgoing messages contain physical memory locations.*

In what follows, we suppress mentioning the auxiliary information to simplify notation and presentation. All of the definitions and results readily extend to the setting where auxiliary input is present.

Non-reactive functionalities. Let $\mathcal{F}$ be a non-reactive functionality. Let Π be a distributed protocol implementing $\mathcal{F}$, $C \subseteq [m]$ be the set of $\leq t$ corrupted servers, and Sim be a PPT simulator. Denote $\bar{C} = [m] \setminus C$. We introduce the following experiments to define active (malicious) security and remark the necessary changes for passive (semi-honest) security.

- $\mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$: Run the protocol with security parameter λ, where honest parties (ones not in C) run the protocol Π honestly with their private input $x_i^* = x_i$, whereas corrupt parties (ones in C) get the corresponding x_i's but can deviate from the prescribed protocol arbitrarily, according to $\mathcal{A}$'s strategy. Let V_i be the view of server $i \in C$ throughout the execution and let y_i be the output of some honest party $i \in \bar{C}$. Output $(\{V_i\}_{i\in C}, \{y_i\}_{i\in\bar{C}})$.
  In the passive (semi-honest) setting, the experiment is the same except that corrupt parties use $x_i^* = x_i$ and they follow the specification of the protocol (i.e., $\mathcal{A}$ is passive).
- $\mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$: First, the adversary sees the inputs of corrupted parties $\{x_i\}_{i\in C}$ and outputs $\{x_i^*\}_{i\in C}$ that may depend on them. Denote $x_i^* = x_i$ for each $i \in \bar{C}$. Then, we invoke the functionality $y_1, \ldots, y_m \leftarrow \mathcal{F}(x_1^*, \ldots, x_m^*)$. Finally, the simulator is executed and the following pair is outputted $(\mathsf{Sim}^{\mathcal{A}}(\lambda, C, \{x_i^*\}_{i\in C}), \{y_i\}_{i\in\bar{C}})$.
  In the passive (semi-honest) setting, the experiment is the same except that the adversary is passive and uses $x_i^* = x_i$.

A distributed protocol obliviously simulates a functionality $\mathcal{F}$ against active (resp. passive) adversaries if the corrupted servers in the real world have views that are indistinguishable from their views in the ideal world.

Definition 2.3 (Distributed oblivious simulation of non-reactive functionalities). *An m-server protocol Π (t, m)-obliviously simulates $\mathcal{F}$ if for any attacker there exists a PPT simulator* Sim *such that, for every subset of t passive/active corrupt parties C, any non-uniform PPT adversary $\mathcal{A}$, and all inputs $x_0, \ldots, x_{m-1}$, the distributions* $\mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *and* $\mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *are computationally indistinguishable.*

Reactive functionalities. A reactive functionality is one that can be repeatedly invoked and it may keep an internal secret state between invocations (a RAM is, in particular, a reactive functionality). The adversary $\mathcal{A}$ chooses the next operation $(\mathsf{op}, \{x_i\}_{i\in[m]})$ adaptively in each stage. In the real execution, the corrupt parties may deviate arbitrarily from the prescribed protocol and the goal is to ensure that they do not learn anything beyond what is absolutely necessary. That is, we execute the protocol in the presence of the malicious adversary. In the ideal execution, the adversary obtains inputs $\{x_i\}_{i\in C}$ and may choose new inputs $\{x_i^*\}_{i\in C}$. The new inputs are fed (together with the inputs of the honest parties) into the functionality $\mathcal{F}$ which outputs an output $\{y_i\}_{i\in\bar{C}}$. At this point, using the output of malicious parties, a simulator must simulate the view of the malicious parties, including their internal state and the obtained messages (and access pattern) from other servers. The adversary can then choose the next command, as well as the next input, in an adaptive manner, based on everything it has seen so far.

Definition 2.4 (Distributed oblivious simulation of a reactive functionality). *We say that a stateful protocol Π is a (t, m)-distributed oblivious implementation of the reactive functionality $\mathcal{F}$ if there exists a stateful PPT simulator* Sim*, such that for any non-uniform PPT (stateful) adversary $\mathcal{A}$, the view of the adversary $\mathcal{A}$ in the following two experiments* $\mathsf{Real}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *and* $\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$ *is computationally indistinguishable:*

```
$\mathsf{Real}_{\Pi,C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$:
Let $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}(1^\lambda)$.
Loop while $\mathsf{op} \neq \bot$:
    Let $x'_i = (\mathsf{op}, x_i)$ for each $i \in [m]$.
    $\{V_i\}_{i\in C}, \{y_i\}_{i\in \bar{C}}$
        $\leftarrow \mathsf{Real}^{\mathsf{nr}}_{\Pi,C,\mathcal{A}}(\lambda, \{x'_i\}_{i\in[m]})$.
    $(\mathsf{op}, \{x_i\}_{i\in[m]})$
        $\leftarrow \mathcal{A}(1^\lambda, \{V_i\}_{i\in C}, \{y_i\}_{i\in \bar{C}})$.
```

```
$\mathsf{Ideal}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x_i\}_{i\in[m]})$:
Let $(\mathsf{op}, \{x_i\}_{i\in[m]}) \leftarrow \mathcal{A}(1^\lambda)$.
Loop while $\mathsf{op} \neq \bot$:
    Let $x'_i = (\mathsf{op}, x_i)$ for each $i \in [m]$.
    $\{V_i\}_{i\in C}, \{y_i\}_{i\in \bar{C}}$
        $\leftarrow \mathsf{Ideal}^{\mathsf{nr}}_{\mathcal{F},\mathsf{Sim},C,\mathcal{A}}(\lambda, \{x'_i\}_{i\in[m]})$.
    $(\mathsf{op}, \{x_i\}_{i\in[m]})$
        $\leftarrow \mathcal{A}(1^\lambda, \{V_i\}_{i\in C}, \{y_i\}_{i\in \bar{C}})$.
```

3 Secure Computation Building Blocks

Our schemes rely on various existing building blocks from the secure computation literature. To encapsulate these building blocks, we assume the existence of an Arithmetic Black Box (ABB) functionality, $\mathcal{F}_{\text{ABB}}$, which is a (reactive) multiparty functionality. $\mathcal{F}_{\text{ABB}}$ should consist of functions $\textsc{Mult}, \textsc{Rnd}, \textsc{Reshare}, \textsc{Eq}, \textsc{IfElse}, \textsc{Bitext}, \textsc{Trunc}$, and $\textsc{Prf}$, each listed below.

To simplify the notation, we denote *"calling a function p of $\mathcal{F}_{\text{ABB}}$"* as *"calling $\mathcal{F}_p$"*, e.g., *"parties call $\mathcal{F}_{\textsc{Mult}}$"* represents that the parties invoke $\mathcal{F}_{\text{ABB}}$ to call its function $\textsc{Mult}$ with their inputs.

Assuming that all secrets are in $\mathbb{Z}_{2^\ell}$, all implementations we introduce below consumes $O(\ell)$-bit communication and $O(\ell)$ local CPU computation steps.

Multiplication. Let $\mathcal{F}_{\textsc{Mult}}$ be a secure multiplication functionality that receives $[\![a]\!]$ and $[\![b]\!]$ and returns $[\![ab]\!]$. For the $(1,3)$-replicated SSS on the extension field $\mathbb{Z}_{2^\ell}$, we can use the implementation of Araki et al. [2] or Chida et al. [15].

For ease of notation, we occasionally denote $\mathcal{F}_{\textsc{Mult}}$ as $[\![a]\!] \times [\![b]\!]$.

Generating random shares. Let $\mathcal{F}_{\textsc{Rnd}}$ be a functionality that requires no inputs but returns a share $[\![r]\!]$ of a secret random value r. In the $(1,3)$-replicated SSS, since the form of shares is $[\![a]\!]_{i \bmod 3} = (a_i, a_{i+1})$; $a = a_0 + a_1 + a_2$, $\mathcal{F}_{\textsc{Rnd}}$ is simply implemented in information-theoretical security as that: Each party P_i locally generate a random r_i, send it to P_{i-1} and set $[\![r]\!]_i$ as (r_i, r_{i+1}).

It is also known that, trading off the information-theoretical security, a pseudorandom r can be shared without any communication excepting a precomputation. This is called Pseudorandom Secret Sharing (PRSS) [16].

Resharing. Let $\mathcal{F}_{\textsc{Reshare}}$ be a functionality that receives $\langle a \rangle_0$ and $\langle a \rangle_1$, and returns $[\![a]\!]$. It can be simply implemented as that the parties P_{i_0} and P_{i_1}, who have $\langle a \rangle_0$ and $\langle a \rangle_1$ respectively, secret-shares their shares as $[\![\langle a \rangle_i]\!]$ for all parties and then they observe $[\![a]\!] = [\![\langle a \rangle_0 + \langle a \rangle_1]\!]$. Under the use of PRSS, the slightly efficient implementation is known [14].

Equality test. Let $\mathcal{F}_{\mathrm{EQ}}$ be a functionality that receives $[\![a]\!]$ and $[\![b]\!]$ then return $[\![c]\!]$ where $c \in \{0,1\}$ is equal to $(a =_? b)$. Though there are numerous implementations of these functionalities, for concreteness, we expect to use the one of Catrina and de Hoogh [10].

Selection. Let $\mathcal{F}_{\textsc{IfElse}}$ be a functionality that receives $[\![c]\!], [\![t]\!]$ and $[\![f]\!]$ such that $c \in \{0,1\}$, and returns $[\![t]\!]$ if $c = 1$, or $[\![f]\!]$ otherwise. Observe that $\mathcal{F}_{\textsc{IfElse}}$ is equal to $f + c(t - f)$.

Bit operations. Let $\mathcal{F}_{\textsc{Bitext}}, \mathcal{F}_{\textsc{Trunc}}$, and $\mathcal{F}_{\textsc{R\_Shift}}$ each be a functionality that receives shares $[\![a]\!]$, whose bit-representation is $a = a_\ell \dots a_1$, and an integer $i; 1 \leq i \leq \ell$, then returns the following output:

- $\mathcal{F}_{\textsc{Bitext}}([\![a]\!], i) \rightarrow [\![a_i]\!]$ s.t. $a_i (\in \{0,1\})$ is the i-th least significant bit of a.
- $\mathcal{F}_{\textsc{Trunc}}([\![a]\!], i) \rightarrow [\![a']\!]$ s.t. $a' = a_\ell \dots a_{i+1} \parallel 0^i$.
- $\mathcal{F}_{\textsc{R\_Shift}}([\![a]\!], i) \rightarrow [\![a']\!]$ s.t. $a' = 0^i \parallel a_\ell \dots a_{i+1}$.

Based on the local bit-decomposition described in Section 2.1, $\mathcal{F}_{\textsc{Bitext}}$ can be implemented straightforwardly as that: For $[\![a]\!] = \sum_{i=0}^{\ell-1} [\![a_i]\!] 2^i$, parties extract the target bit $[\![a_i]\!]$, generate $[\![\vec{0}]\!] = \sum_{i=0}^{\ell-1} [\![0]\!] 2^i$, and compute $[\![\vec{0}]\!] + ([\![a_i]\!] 2^0)$. $\mathcal{F}_{\textsc{Trunc}}$ and $\mathcal{F}_{\textsc{R\_Shift}}$ can be realized in the similar way.

Oblivious PRF (OPRF). Let $\mathcal{F}_{\textsc{Prf}}$ be a functionality that receives shares $[\![\mathsf{sk}]\!]$ and $[\![x]\!]$ from all parties and sends them $[\![y]\!]$ where y is given by a PRF $\mathsf{F}_{\mathsf{sk}}(x)$. A combination of known oblivious block ciphers [1, 15, 17, 42] and $\mathcal{F}_{\textsc{R\_Shift}}$ implements $\mathcal{F}_{\textsc{Prf}}$.

4 Efficient Passively Secure Distributed Oblivious Hashing

A (static) hash table is a data structure supporting three operations Build, Lookup, and Extract, that realizes the following reactive functionality. The Build procedure creates an in-memory data structure from an input array **I** containing real and dummy elements where each element is a (key, value) pair. Dummy elements have their key be $\perp$. It is assumed that all real elements in **I** have distinct keys. The Lookup procedure allows a requestor to look up the value of a key. A $\perp$ symbol is returned if the key is not found or if $\perp$ is the requested key. We say a (key, value) pair is visited if the key was searched for and found before. Finally, Extract is the destructor and it returns a list containing unvisited elements padded with dummies to the same length as the input array $\mathbf{I}$.

The description of this functionality, denoted $\mathcal{F}_{\mathsf{HT}}$, is described next:

$\underline{\mathcal{F}_{\mathsf{HT}}}$:

- Build($\mathbf{I}$): The input is an array $\mathbf{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted $a_i = (k_i, v_i)$. It is assumed that keys and values fit into $O(1)$ memory words and that all real keys are distinct. The procedure does:
  1. Initialize the state $\mathsf{H} = (\mathbf{I}, \mathbf{P})$ where $\mathbf{P} = \emptyset$.
- Lookup(k): The input is a key k (that might be $\perp$, i.e., dummy). The procedure does:
  1. If $k \in \mathbf{P}$ (i.e., k is a recurring lookup), then halt and return $\perp$.
  2. If $k = \perp$ or $k \notin \mathbf{I}$, set $v^* = \perp$.
  3. Otherwise, set $v^* = v$, where v is the value corresponding to $k (\in \mathbf{I})$.
  4. Update $\mathbf{P} = \mathbf{P} \cup \{k\}$.
  5. Output v^*.
- Extract(): The procedure does:
  1. Define $\mathbf{I}' = \{a'_1, \ldots, a'_n\}$ such that: For $i \in [n]$, set $a'_i = a_i$ if $a_i = (k, v)$ and $k \notin \mathbf{P}$. Otherwise, set $a'_i = \perp$.
  2. Output $\mathbf{I}'$.

In this section, we propose a simple $(1, 3)$-distributed oblivious implementation of $\mathcal{F}_{\mathsf{HT}}$ that is inspired by Lu et al. [44]. Since Lu's DORAM requires too many (at least $100 \log N$) OPRF calls for distributed blocks, the practical computation cost becomes expensive even if the asymptotic overhead is down to $O(\log N)$. To reduce the practical computation complexity without increasing the asymptotic overhead, we construct a concretely efficient distributed oblivious hashing from a simple new permutation protocol for 3-party computation.

In a very high level, our distributed oblivious hashing scheme works as follows (assuming a permutation protocol):

1. Starting with $(1, 3)$-shares of input blocks, the parties securely compute (pseudorandom) addresses for the blocks to be placed.
2. The parties divide their roles into one *permuter* and two *storages*, and then only the permuter reveals the addresses of the blocks.
3. The permuter computes a permutation for the blocks, which is a *sorting permutation* depending on the addresses.
4. The parties obliviously apply the permutation (that is secret for the storages) and the storages receive the $(1, 2)$-shares of the permuted blocks.
5. Now, the permuted blocks can take the form of some hash table (if the permutation is valid), and only the storages can access the table.

4.1 Distributed Oblivious Permutation

As a building block for our distributed oblivious hash table, we first construct an efficient 3-party secure permutation protocol. The precise functionality that we implement is described below. This section is devoted to the implementation of these functionalities via a 3-party protocol where at most one may be corrupted.

$\underline{\mathcal{F}_{\text{PERM}}}$:

1. Receive a permutation π from the *permuter* P, and receive $(1,3)$-shares of an array $[\![\mathbf{I}]\!]$ from all parties.
2. Obtain $\mathbf{I}$, compute $\pi \cdot \mathbf{I}$, and choose a random string $\mathbf{R}$ of the same size as $\mathbf{I}$.
3. Send $\langle \mathbf{I}' \rangle_0 = \mathbf{R}$ to the first *storage* S_0 and $\langle \mathbf{I}' \rangle_1 = \pi \cdot \mathbf{I} - \mathbf{R}$ to the second S_1.

$\underline{\mathcal{F}_{\text{UNPERM}}}$:

1. Receive a permutation π from the *permuter* P and 2-out-of-2 shares of an array $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$ from the *storages* S_0, S_1.
2. Reconstruct $\mathbf{I}$ and compute $\pi^{-1} \cdot \mathbf{I}$.
3. Return $[\![\mathbf{I}']\!] \leftarrow \mathcal{F}_{\text{SHARE}}(\pi^{-1} \cdot \mathbf{I})$ for all parties.

Lemma 4.1 (Realization of $\mathcal{F}_{\text{Perm}}$). *There is a (1,3)-distributed oblivious implementation, described as Algorithm 1, of $\mathcal{F}_{\text{PERM}}$ in the presence of a passive adversary that controls one party. The protocol consumes $4nb + 2n\lceil \log n \rceil$ bits of communication and $12nb + 2n\lceil \log n \rceil$ local CPU computation steps where n is the number of blocks in the input array and b is the bit-length of each block. It also consumes 2 communication rounds.*

Lemma 4.2 (Realization of $\mathcal{F}_{\text{Unperm}}$). *There is a distributed 3-party protocol, described as Algorithm 2, that securely realizes $\mathcal{F}_{\text{UNPERM}}$ in the presence of a passive adversary that controls one party and in the $\mathcal{F}_{\text{RESHARE}}$-hybrid model. By composition and using the implementation of $\mathcal{F}_{\text{RESHARE}}$ described in Section 3, the protocol consumes $8nb + 2n\lceil \log n \rceil$ bits of communication cost and $19nb + 2n\lceil \log n \rceil$ local CPU computation steps where n is the number of blocks in the input array and b is the bit-length of each block. It also consumes 3 communication rounds.*

Algorithm 1 $(\cdot, \langle \mathbf{I}' \rangle_0, \langle \mathbf{I}' \rangle_1) \leftarrow \mathit{\Pi}_{\text{PERM}}((\pi, [\![\mathbf{I}]\!]_0), [\![\mathbf{I}]\!]_1, [\![\mathbf{I}]\!]_2)$

Notation: P is the "permuter" and S_0, S_1 are two "storages."
Require: P has a permutation π and each party has shares of an array $[\![\mathbf{I}]\!]$.
Ensure: $\mathbf{I}' = \pi \cdot \mathbf{I}$.

```
1: $P$ and $S_0$ convert their $(1,3)$-shares $[\![\mathbf{I}]\!]_0, [\![\mathbf{I}]\!]_1$ to $(1,2)$-shares $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$, respectively.
2: $P$ chooses random strings $\mathbf{U}, \mathbf{V}$ of the same size as $\langle \mathbf{I} \rangle_0$, and random permutations
   $\pi_0, \pi_1$ s.t. $\pi_1 \circ \pi_0 = \pi$.
3: $P$ sends $\mathbf{U}, \mathbf{V}, \pi_0$ to $S_0$ and $\widetilde{\mathbf{I}}_0 := \pi \cdot \langle \mathbf{I} \rangle_0 - \pi_1 \cdot \mathbf{U} - \mathbf{V}$ to $S_1$.
4: $S_0$ sends $\widetilde{\mathbf{I}}_1 := \pi_0 \cdot \langle \mathbf{I} \rangle_1 + \mathbf{U}$ to $S_1$.
5: $S_0$ outputs $\langle \mathbf{I}' \rangle_0 := \mathbf{V}$, and $S_1$ outputs $\langle \mathbf{I}' \rangle_1 := \widetilde{\mathbf{I}}_0 + \pi_1 \cdot \widetilde{\mathbf{I}}_1$.
```

Algorithm 2 $[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2 \leftarrow \Pi_{\text{Unperm}}(\pi, \langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1)$

```
Notation: $P$ is the "permuter" and $S_0, S_1$ are two "storages."
Require: $P$ has a permutation $\pi$ and $S_0, S_1$ have a shares $\langle \mathbf{I} \rangle_0, \langle \mathbf{I} \rangle_1$, respectively.
Ensure: $\mathbf{I}' = \pi^{-1} \cdot \mathbf{I}$.
1: $S_1$ chooses a random strings $\mathbf{U}, \mathbf{V}$ of the same size as $\langle \mathbf{I} \rangle_0$.
2: $P$ chooses random permutations $\pi_0, \pi_1$ s.t. $\pi_1 \circ \pi_0 = \pi$.
3: $S_1$ sends $\mathbf{U}$ to $P$ and $\mathbf{V}$ to $S_0$. $P$ sends $\pi_0$ to $S_0$ and $\pi_1$ to $S_1$.
4: $S_0$ sends $\widetilde{\mathbf{I}}_0 := \langle \mathbf{I} \rangle_0 + \mathbf{V}$ to $P$. $S_1$ sends $\widetilde{\mathbf{I}}_1 := \pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}$ to $S_0$.
5: $P$ computes $\langle \mathbf{I}' \rangle_0 := \pi^{-1} \cdot \widetilde{\mathbf{I}}_0 + \pi_0^{-1} \cdot \mathbf{U}$, and $S_0$ computes $\langle \mathbf{I}' \rangle_1 := \pi_0^{-1} \cdot \widetilde{\mathbf{I}}_1$.
6: Parties call $\mathcal{F}_{\text{Reshare}}$ to convert $\langle \mathbf{I}' \rangle_0, \langle \mathbf{I}' \rangle_1$ to $[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2$.
```

Proof of Lemma 4.1. The output of S_0 is $\mathbf{V}$ and the output of S_1 is

$$\begin{aligned} \widetilde{\mathbf{I}}_0 + \pi_1 \cdot \widetilde{\mathbf{I}}_1 &= \pi \cdot \langle \mathbf{I} \rangle_0 - \pi_1 \cdot \mathbf{U} - \mathbf{V} + \pi_1 \circ \pi_0 \cdot \langle \mathbf{I} \rangle_1 + \pi_1 \cdot \mathbf{U} \\ &= \pi \cdot \mathbf{I} - \mathbf{V}. \end{aligned}$$

Together, they form a $(1,2)$-share of $\pi \cdot \mathbf{I}$, as needed for correctness. The claimed efficiency follows by direct inspection. The strings $\mathbf{U}$ and $\mathbf{V}$ are each nb bits long, similarly to $\widetilde{\mathbf{I}}_0$ and $\widetilde{\mathbf{I}}_0$. The bit-length of π_0 and π_1 is $n\lceil \log n \rceil$, each.

For security, observe that the permuter P never receives any message and does not have any output, and therefore it is trivial to simulate its view. Similarly, the first storage server S_0 only gets one message $\mathbf{U}, \mathbf{V}, \pi_0$ (from P), all of which are uniformly random and independent of the inputs of all parties. The output of S_0 contains $\mathbf{V}$ and so overall it is immediate to simulate its view. The only case remaining is when S_1 is corrupted. Its view in the protocol consists of $\pi_1, \widetilde{\mathbf{I}}_0, \widetilde{\mathbf{I}}_1$ which can be simulated by 3 uniformly random strings of appropriate length. Indeed, $\widetilde{\mathbf{I}}_0$ is masked by $\mathbf{V}$ and then $\widetilde{\mathbf{I}}_1$ is masked by $\mathbf{U}$, all of which are not known to S_1. □

Proof of Lemma 4.2. Since we are in the $\mathcal{F}_{\text{Reshare}}$-hybrid model, for correctness we need to show that $\langle \mathbf{I}' \rangle_0 + \langle \mathbf{I}' \rangle_1 = \pi^{-1} \cdot \mathbf{I}$. Indeed,

$$\begin{aligned} \langle \mathbf{I}' \rangle_0 + \langle \mathbf{I}' \rangle_1 &= \pi^{-1} \cdot (\langle \mathbf{I} \rangle_0 + \mathbf{V}) + \pi_0^{-1} \cdot \mathbf{U} + \pi_0^{-1} \cdot (\pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}) \\ &= \pi^{-1} \cdot \mathbf{I}. \end{aligned}$$

The claimed efficiency follows by direct inspection. The strings $\mathbf{U}$ and $\mathbf{V}$ are each nb bits long, similarly to $\widetilde{\mathbf{I}}_0$ and $\widetilde{\mathbf{I}}_0$. The bit-length of π_0 and π_1 is $n\lceil \log n \rceil$, each.

For security, if S_1 is corrupted, we can easily simulate its view as it does not receive any message except π_1 which is uniformly random and independent of the other inputs. If P is corrupted, then its again immediate to simulate its view since it only receives $\mathbf{U}$ and $\widetilde{\mathbf{I}}_0 := \langle \mathbf{I} \rangle_0 + \mathbf{V}$ throughout the execution, both of which are uniformly distributed (in P's view). Lastly, assume that S_0 is corrupted. Its view consists of $\mathbf{V}, \pi_0$ and $\widetilde{\mathbf{I}}_1 := \pi_1^{-1} \cdot (\langle \mathbf{I} \rangle_1 - \mathbf{V}) - \mathbf{U}$. Since it does not know $\mathbf{U}$, the term $\widetilde{\mathbf{I}}_1$ looks completely uniform and therefore the whole view can be simulated by 3 uniformly random strings of the appropriate length. □

4.2 Distributed Oblivious Hashing

Equipped with the above permutation protocols, we proceed with our distributed oblivious hashing scheme. The idea is simple to describe at a high level: given a set of elements we first obliviously permute them and then we index the permuted set using a non-oblivious efficient hashing scheme. For the latter, we use *Cuckoo hashing* [50], a hashing paradigm that resolves collisions in a table by using two hash functions and two tables, cleverly assigning each element to one of the two tables, and enabling lookup using only two queries. The standard version of Cuckoo hashing suffers from inverse polynomial probability of build failure which does not suffice for our application (since we aim for negligible error). To this end, we use a variant of Cuckoo hashing where items that cannot be stored in one of the two tables are stored in a (typically small) "stash". According to Noble [47], for any number of elements $n = \omega(\log \lambda)$, there exists Cuckoo hashing that has the table of size $\tau = (1 + \epsilon)n$ and the stash of size $\sigma = \Theta(\log \lambda)$ with negligible failure probability. Henceforth, we specify $\tau = 2n$ and $\sigma = \lceil \log \lambda \rceil$ for concrete efficiency analysis.

To ease presentation of our implementations, we use the following notations to represent shares of real/dummy blocks.

- Let $[\![\mathsf{d}]\!] = ([\![k]\!], [\![v]\!])$ be a share of a data block that contains shares of a key k and value v.
- Let $[\![\mathsf{d}^{\mathsf{dummy}}]\!] = ([\![\bot]\!], [\![\bot]\!])$ be a share of a dummy block. We assume that $\bot$ is a number greater than any real k.
- Let $[\![\mathsf{D}]\!] = ([\![\mathsf{d}_0]\!], \ldots, [\![\mathsf{d}_{n-1}]\!])$ be a share of a dataset of size n.

Algorithm 3 constructs 2-out-of-2 shares of a Cuckoo hash table, $(\langle \mathsf{T} \rangle, \langle \mathsf{S} \rangle)$, from 2-out-of-3 shares of a dataset $[\![\mathsf{D}]\!]$. In this protocol, the party assigned the role of *permuter* obtains all addresses the data should be placed and then computes a permutation π that moves the data to (one of) the corresponding addresses. Now, parties can efficiently convert D into the table (T, S) via $\mathcal{F}_{\text{PERM}}$.

Algorithm 4 fetches a queried item from the Cuckoo hash table. The main part of this protocol is that the parties assigned the role of *storage* obtain two addresses of the queried item, access T of the location indicated by them, and select the one out of the two items of T. When the parties receive a dummy query, random locations of T are fetched.

Algorithm 5 is a deconstruction procedure that applies the inverted permutation π^{-1} to the hash table. This π^{-1} sorts all real blocks in the hash table into the order in which they were input to Π_{BUILD}.

Lemma 4.3. *Assume that the input array consists of* $n = \omega(\log \lambda)$ *blocks and each block is* b *bits. There exists a distributed* 3*-party protocol, described as Algorithms 3, 4 and 5, that securely realizes* $\mathcal{F}_{\mathsf{HT}}$ *in the* $(\mathcal{F}_{\text{SHARE}}, \mathcal{F}^{\mathcal{P}}_{\text{REVEAL}}, \mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{PERM}}, \mathcal{F}_{\text{UNPERM}})$*-hybrid model and in the presence of a passive adversary that controls one party.*

- *The* BUILD *procedure consumes* $O(n(b' + \lceil \log n \rceil))$ *local computation steps.*
- *The* LOOKUP *procedure consumes* $O(b' + b' \log \lambda)$ *local computation steps.*

Algorithm 3 $\pi, (\langle \mathsf{T} \rangle_i, \langle \mathsf{S} \rangle_i)_{(i \in \{0,1\})}, [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P} \leftarrow \Pi_{\text{BUILD}}([\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2)$

```
Notation: Let $P$ be the permuter and $S_0, S_1$ be the storages. Let $\tau := |\mathsf{T}|$ and $\sigma := |\mathsf{S}|$
    be the size of expected hash table $(\mathsf{T}, \mathsf{S})$ storing $n$ items, and let $n' = \tau + \sigma$.
Require: Parties have $(1,3)$-shares of a dataset $[\![\mathsf{D}]\!]_{i \in \{0,1,2\}} = ([\![\mathsf{d}_0]\!]_i, \ldots, [\![\mathsf{d}_{n-1}]\!]_i)$.
Ensure: $S_0$ and $S_1$ obtain $(1,2)$-shares of Cuckoo hash table, $(\langle \mathsf{T} \rangle_0, \langle \mathsf{S} \rangle_0)$ and
    $(\langle \mathsf{T} \rangle_1, \langle \mathsf{S} \rangle_1)$, respectively. $P$ obtains a permutation $\pi$, and all parties hold $(1,3)$-
    shares of PRF keys, $[\![s_0]\!]$ and $[\![s_1]\!]$. Parties also hold a query counter $\mathsf{ctr}$, and $S_0, S_1$
    hold a set $\mathbf{P}$, as their states.
(Computing pseudorandom addresses for the input data.)
 1: Parties call $\mathcal{F}_{\text{RND}}$ to generate random PRF keys $[\![s_0]\!]$ and $[\![s_1]\!]$.
 2: for all $i \in [n]$ do
 3:     Parties call $\mathcal{F}_{\text{EQ}}$ to obtain $[\![\mathsf{isDummy}_i]\!] := [\![k_i =_? \perp]\!]$.
 4:     Parties call $\mathcal{F}_{\text{IFELSE}}$ to simulate:
            If $\mathsf{isDummy}_i = 1$, then $[\![\widetilde{k_i}]\!] := [\![\perp + i]\!]$; otherwise $[\![\widetilde{k_i}]\!] := [\![k_i]\!]$.
 5:     Parties call $\mathcal{F}_{\text{PRF}}$ to obtain pseudorandom addresses $[\![\mathsf{addr}_{i,0}]\!]$ and $[\![\mathsf{addr}_{i,1}]\!]$ from
        $([\![s_0]\!], [\![\widetilde{k_i}]\!])$ and $([\![s_1]\!], [\![\widetilde{k_i}]\!])$, respectively.
(Building a Cuckoo hash table via permutation.)
 6: Parties call $\mathcal{F}_{\text{SHARE}}$ to generate an array $[\![\mathsf{E}]\!]$ consisting of $n' - n$ dummy blocks.
 7: Let $[\![\widetilde{\mathsf{D}}]\!] := [\![\mathsf{D}]\!] \parallel [\![\mathsf{E}]\!]$ be a concatenated dataset.
 8: Parties call $\mathcal{F}^{\{P\}}_{\text{REVEAL}}$ to reveal to $P$ all $\mathsf{addr}_{i,0}, \mathsf{addr}_{i,1}$ for $i \in [n]$.
 9: $P$ computes a permutation $\pi \colon [n'] \to [n']$ that indicates the bin-placements of
    Cuckoo hashing. That is, $\pi$ says where to place $[\![\widetilde{\mathsf{D}}]\!]$'s first $n$ elements (i.e., the
    elements of $[\![\mathsf{D}]\!]$), as indicated by either $\mathsf{addr}_{i,0}$ or $\mathsf{addr}_{i,1}$.
10: Parties call $\mathcal{F}_{\text{PERM}}$ with $\pi$ and $[\![\widetilde{\mathsf{D}}]\!]$ to make $S_0$ and $S_1$ obtain $\langle \widetilde{\mathsf{D}'} \rangle_0$ and $\langle \widetilde{\mathsf{D}'} \rangle_1$
    respectively, where $\widetilde{\mathsf{D}'} = \pi \cdot \widetilde{\mathsf{D}}$.
11: Each $S_i$ for $i \in \{0,1\}$ organizes the array $\langle \widetilde{\mathsf{D}'} \rangle_i$ into a hash table $\langle \mathsf{T} \rangle_i$ and stash
    $\langle \mathsf{S} \rangle_i$, by separating $\langle \widetilde{\mathsf{D}'} \rangle_i$ into the first $\tau$ and the last $\sigma$ elements.
12: Parties set their state $\mathsf{ctr} = 0$, and $S_0, S_1$ allocate an empty set $\mathbf{P} = \emptyset$.
13: Return $\pi, (\langle \mathsf{T} \rangle_0, \langle \mathsf{S} \rangle_0), (\langle \mathsf{T} \rangle_1, \langle \mathsf{S} \rangle_1), [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P}$
```

– *The* EXTRACT *procedure consumes $O(nb')$ local computation steps.*

Furthermore, all those procedures consumes $O(1)$ communication rounds.

Proof. The proof of security and correctness is given in Appendix B.1. We focus on the efficiency analysis below.

Algorithm 3 consists of $O(n)$ calls of $\mathcal{F}_{\text{EQ}}, \mathcal{F}_{\text{IFELSE}}, \mathcal{F}_{\text{PRF}}, \mathcal{F}_{\text{SHARE}}$, and $\mathcal{F}_{\text{REVEAL}}$, and an invocation of $\mathcal{F}_{\text{PERM}}$. In Algorithm 4, the parties need to call $\mathcal{F}_{\text{MULT}}, \mathcal{F}_{\text{RESHARE}}, \mathcal{F}_{\text{EQ}}, \mathcal{F}_{\text{IFELSE}}, \mathcal{F}_{\text{REVEAL}}$, and $\mathcal{F}_{\text{PRF}}$ $O(1)$ times for the table lookup, and also call $\mathcal{F}_{\text{MULT}}, \mathcal{F}_{\text{RESHARE}}, \mathcal{F}_{\text{EQ}}$, and $\mathcal{F}_{\text{IFELSE}}$ $O(\sigma)$ times for exploring the stash. Lastly, Algorithm 5 requires $\mathcal{F}_{\text{UNPERM}}$ at once. In addition, all iterative operations in Algorithm 3, 4, and 5 can be performed in parallel. □

Algorithm 4 $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!] \leftarrow \Pi_{\text{Lookup}}([\![k]\!], (\langle \mathsf{T} \rangle_i, \langle \mathsf{S} \rangle_i)_{(i \in \{0,1\})}, [\![s_0]\!], [\![s_1]\!], \mathsf{ctr}, \mathbf{P})$

```
Notation: Let $P$ be the permuter and let $S_0, S_1$ be the storages. Let $\sigma = |\mathsf{S}|$.
Require: $[\![k]\!]$ is a $(1,3)$-share of an input key to be searched for. $S_0, S_1$ have the
    distributed hash table $(\langle \mathsf{T} \rangle_i, \langle \mathsf{S} \rangle_i)$ and a set $\mathbf{P}$. Parties have $(1,3)$-shares of PRF
    keys $[\![s_0]\!], [\![s_1]\!]$ and a query counter $\mathsf{ctr}$.
Ensure: $\mathsf{d} = (k, v), \mathsf{found} = 1$ if $(\mathsf{T}, \mathsf{S})$ contains $(k, v)$. Otherwise, $\mathsf{d} = (0, 0), \mathsf{found} = 0$.
(Computing pseudorandom addresses to be fetched.)
 1: Parties call $\mathcal{F}_{\text{Eq}}$ to obtain $[\![\mathsf{isDummy}]\!] := [\![k =_? \perp]\!]$.
 2: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate:
        If $\mathsf{isDummy} = 1$, then set $[\![\widetilde{k}]\!] := [\![\perp + \mathsf{ctr}]\!]$; otherwise $[\![\widetilde{k}]\!] := [\![k]\!]$.
 3: Parties call $\mathcal{F}_{\text{Prf}}$ to obtain $[\![\mathsf{addr}_0]\!], [\![\mathsf{addr}_1]\!]$ from $([\![s_0]\!], [\![\widetilde{k}]\!]), ([\![s_1]\!], [\![\widetilde{k}]\!])$, respectively.
 4: Parties call $\mathcal{F}_{\text{Reveal}}^{\{S_0, S_1\}}$ to recover $\mathsf{addr}_0, \mathsf{addr}_1$ to both $S_0$ and $S_1$. If $(\mathsf{addr}_0, \mathsf{addr}_1) \in \mathbf{P}$,
    $S_0$ and $S_1$ halt. Otherwise, they update $\mathbf{P} \leftarrow \mathbf{P} \cup \{(\mathsf{addr}_0, \mathsf{addr}_1)\}$.
(Searching for the table T.)
 5: for all $i = 0, 1$ do
 6:     Parties call $\mathcal{F}_{\text{Reshare}}$ to convert $\langle \mathsf{T}[\mathsf{addr}_i] \rangle = (\langle k_i \rangle, \langle v_i \rangle)$ to $([\![k_i]\!], [\![v_i]\!])$.
 7:     Parties call $\mathcal{F}_{\text{Eq}}$ to obtain $[\![\mathsf{isQueried}_i]\!] = [\![k_i =_? k]\!]$.
 8:     Parties call $\mathcal{F}_{\text{IfElse}}$ and $\mathcal{F}_{\text{Mult}}$ to simulate:
            If $\mathsf{isDummy} = 0$ and $\mathsf{isQueried}_i = 1$,
                then $\mathsf{found} = 1$, $[\![\mathsf{d}]\!] = ([\![k_i]\!], [\![v_i]\!])$, and $\langle \mathsf{T}[\mathsf{addr}_i] \rangle = \langle \mathsf{d}^{\mathsf{dummy}} \rangle$.
(Searching for the stash S.)
 9: for all $u \in [\sigma]$ do
10:     Parties call $\mathcal{F}_{\text{Reshare}}$ to convert $\langle \mathsf{S}[u] \rangle = (\langle k_u \rangle, \langle v_u \rangle)$ to $([\![k_u]\!], [\![v_u]\!])$.
11:     Parties call $\mathcal{F}_{\text{Eq}}$ to obtain $[\![\mathsf{isQueried}_u]\!] = [\![k_u =_? k]\!]$.
12:     Parties call $\mathcal{F}_{\text{IfElse}}$ and $\mathcal{F}_{\text{Mult}}$ to simulate:
            If $\mathsf{isDummy} = 0$ and $\mathsf{isQueried}_u = 1$,
                then $\mathsf{found} = 1$, $[\![\mathsf{d}]\!] = ([\![k_u]\!], [\![v_u]\!])$, and $\langle \mathsf{S}[u] \rangle = \langle \mathsf{d}^{\mathsf{dummy}} \rangle$.
13: Parties increment $\mathsf{ctr}$.
14: Return $[\![\mathsf{d}]\!], [\![\mathsf{found}]\!]$
```

5 Optimal DORAM Against Passive Adversary

In this section, we give our optimal 3-party DORAM that is secure against a passive adversary who colludes with one of the three servers.

Let $N \in \mathsf{poly}(\lambda)$ be the memory size. Our DORAM is on the known hierarchical paradigm, i.e. the data structure is built via a hierarchy of $L := \lceil \log N - \log\log N \rceil$ distributed hash tables and one top-level array. All levels $i = 1, \ldots, L$ in the hierarchy are implemented using our distributed oblivious hash table $\mathcal{F}_{\mathsf{HT}}$ from Section 4.2. The size of the stash in our distributed oblivious hash table is set to $\sigma = \lceil \log N \rceil$. The top-level array can store up to $c = 2\sigma$ data blocks. The capacity of level $i \in [L]$ is $c2^{i-1}$. Each data block may be associated with metadata that is used to keep track of the location of an element. Specifically, an "augmented data block" $[\![\widetilde{\mathsf{d}_i}]\!] := ([\![\mathsf{d}_i]\!], [\![\mathsf{lv}_i]\!])$ consists of the main

Algorithm 5 $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2 \leftarrow \Pi_{\text{Extract}}(\pi, (\langle \mathsf{T} \rangle_i, \langle \mathsf{S} \rangle_i)_{(i \in \{0,1\})})$

```
Notation: Let $P$ be the permuter and let $S_0, S_1$ be the storages.
Require: $S_0$ and $S_1$ have the distributed hash table $(\langle \mathsf{T} \rangle_0, \langle \mathsf{S} \rangle_0)$ and $(\langle \mathsf{T} \rangle_1, \langle \mathsf{S} \rangle_1)$,
    respectively. $P$ have the permutation $\pi$.
Ensure: A dataset $\mathsf{D}$ contains all real elements in $\mathsf{T}$ and $\mathsf{S}$.
1: Each $S_i$ for $i \in \{0,1\}$ reorganizes $\langle \mathsf{T} \rangle_i$ and $\langle \mathsf{S} \rangle_i$ into an array $\langle \widetilde{\mathsf{D}} \rangle_i$ as $\widetilde{\mathsf{D}} = \mathsf{T} \| \mathsf{S}$.
2: Parties call $\mathcal{F}_{\text{Unperm}}$ with $(\pi, \langle \widetilde{\mathsf{D}} \rangle_0, \langle \widetilde{\mathsf{D}} \rangle_1)$ to obtain $[\![\widetilde{\mathsf{D}'}]\!]_{j \in \{0,1,2\}}$ where $\widetilde{\mathsf{D}'} = \pi^{-1} \cdot \widetilde{\mathsf{D}}$.
3: Let $[\![\mathsf{D}]\!]_j$ be the first $n$ elements of $[\![\widetilde{\mathsf{D}'}]\!]_i$.
4: Return $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2$
```

data block $[\![\mathsf{d}_i]\!] = ([\![k_i]\!], [\![v_i]\!])$ and additional information $[\![\mathsf{lv}_i]\!]$; $\mathsf{lv}_i \in \{0,1\}^{L-1}$ that indicates the levels to which $[\![\widetilde{\mathsf{d}}_i]\!]$ is associated.[7]

Recall that each instance of HT_i consists of two parts: the main table and a stash. Looking forward, the stashes from all levels are combined to the top-level array, as commonly done (e.g., in [44]). Thus, when we perform Lookup or Extract to some HT_i, we ignore stash-related operations (which will be done in a centralized fashion at the beginning of the operation).

Theorem 5.1. *There is a 3-party protocol, described as Algorithm 6 and 7, that securely realizes $\mathcal{F}_{\text{RAM}}$ in the $(\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model and in the presence of a passive adversary that controls one party. The construction costs $O(\log N)$ amortized computational overhead and $O(\log N)$ communication rounds with $\Omega(\log N)$ block size.*

Proof. The proof of security is given in Appendix B.2. In addition, correctness is clear from the algorithms since we are in the $\mathcal{F}_{\mathsf{HT}}$-hybrid model,. Indeed, a lookup is performed through the whole hierarchy and when an element is found, it is re-inserted into the hierarchy. Hence, we focus on efficiency analysis next.

Algorthm 6 requires $O(c)$ calls of $\mathcal{F}_{\text{Eq}}, \mathcal{F}_{\text{IfElse}}$, and $\mathcal{F}_{\text{Mult}}$, $O(L)$ calls of H.Lookup (without stash), and an invocation of $\Pi_{\text{Reshuffle}}$, for $c = L = O(\log N)$. For any block size $b = \Omega(\log N)$ bits, this procedure requires $O(b \log N) + \mathcal{C}$ computational steps where $\mathcal{C}$ is the amortized cost of $\Pi_{\text{Reshuffle}}$. Algorithm 7 costs H_i.Extract for all $i = 1, \ldots, p$, and H_p.Build at once, per $c2^{p-1}$ access. Hence, its amortized cost can be estimated as $\mathcal{C}_{\text{Reshuffle}} = \sum_{p=1}^{L} \frac{O(c2^p b)}{c2^{p-1}}$. $\square$

Concrete Efficiency. As we mentioned in Section 1.1, the overhead claimed above depends on a small hidden constant. Specifically, our Π_{Access} consists of only $4 \log N$ (amortized) calls of $\mathcal{F}_{\text{Prf}}$ per access, that is 25 times smaller than the known optimal DORAM [44]. For more detailed analysis, see Appendix A.

[7] The metadata associated with each data blocks is used to avoid the stash-resampling attack of [22], same as was done in [3, 4, 22].

Algorithm 6 $[\![\mathsf{d}]\!] \leftarrow \mathit{\Pi}_{\text{Access}}([\![\mathsf{op}]\!], [\![k]\!], [\![v']\!])$

```
Require: The input contains shares of an operation $[\![\mathsf{op}]\!]$; $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, key $[\![k]\!]$,
    and value $[\![v']\!]$.
Ensure: $\mathsf{d} = (k, v)$ if the ORAM holds $(k, v)$, or $\mathsf{d} = (k, v')$ if $\mathsf{op} = \mathsf{write}$. Otherwise,
    $\mathsf{d} = (\perp, \perp)$.
(Searching for the top-level array.)
 1: Allocate an empty data block $[\![\widetilde{\mathsf{d}}]\!] = ([\![\mathsf{d}]\!], [\![\mathsf{lv}]\!])$ and initialize $[\![\mathsf{found}]\!] = [\![0]\!]$.
 2: for all $u \in [c]$ do
 3:     Parties retrieve $[\![\widetilde{\mathsf{d}}_u]\!] = (([\![k_u]\!], [\![v_u]\!]), [\![\mathsf{lv}_u]\!])$ from the top-level array.
 4:     Parties call $\mathcal{F}_{\text{Eq}}$ to obtain $[\![\mathsf{found}_u]\!] = [\![k_u =_? k]\!]$.
 5:     Parties call $\mathcal{F}_{\text{IfElse}}$ and $\mathcal{F}_{\text{Mult}}$ to simulate:
          If $\mathsf{found} = 0$ and $\mathsf{found}_u = 1$, then $\mathsf{found} = 1$ and $[\![\widetilde{\mathsf{d}}]\!] = [\![\widetilde{\mathsf{d}}_u]\!]$.
(Searching for the cuckoo hash table in the hierarchy.)
 6: for $i = 1$ to $L$ do
 7:     Parties call $\mathcal{F}_{\text{BitExt}}$ to obtain $[\![\mathsf{lv}_i]\!]$ where $\mathsf{lv}_i$ is the $i$-th least significant bit of $\mathsf{lv}$.
 8:     Parties call $\mathcal{F}_{\text{IfElse}}$ and $\mathcal{F}_{\text{Mult}}$ to simulate:
          If $\mathsf{found} = 1$ and $\mathsf{lv}_i = 0$ then $[\![\widetilde{k}]\!] := [\![\perp]\!]$, otherwise $[\![\widetilde{k}]\!] := [\![k]\!]$.
 9:     Parties call $\mathsf{HT}_i$.Lookup($[\![\widetilde{k}]\!]$) with its state $(\langle \mathsf{T}_i \rangle_0, \langle \mathsf{T}_i \rangle_1, [\![s_{i,0}]\!], [\![s_{i,1}]\!], \mathsf{ctr}, \mathbf{P})$
        to obtain $[\![\widetilde{\mathsf{d}}_i]\!], [\![\mathsf{found}_i]\!]$.
10:     Set $[\![\widetilde{\mathsf{d}}]\!] = [\![\widetilde{\mathsf{d}}]\!] + [\![\widetilde{\mathsf{d}}_i]\!]$ and $[\![\mathsf{found}]\!] = [\![\mathsf{found}]\!] + [\![\mathsf{found}_i]\!]$.
(Rewriting (if needed) and re-storing the retrieved data.)
11: Parties call $\mathcal{F}_{\text{IfElse}}$ to simulate: If $\mathsf{found} = 0$ then set $[\![\mathsf{d}]\!] = [\![\mathsf{d}^{\mathsf{dummy}}]\!]$.
12: Parties also call $\mathcal{F}_{\text{IfElse}}$ to simulate: If $\mathsf{op} = \mathsf{write}$ then set $[\![\mathsf{d}]\!] = ([\![k]\!], [\![v']\!])$.
13: Parties set $[\![\widetilde{\mathsf{d}}]\!] = ([\![\mathsf{d}]\!], [\![0]\!])$ and concatenate it into the end of the top-level array.
14: If the size of the top-level array is $c$, parties run $\mathit{\Pi}_{\text{Reshuffle}}()$ to refresh the hierarchy.
15: Return $[\![\mathsf{d}]\!]$
```

6 Actively Secure Extension

We extend our oblivious hashing and DORAM to be secure against an adversary that can deviate from the prescribed protocols. Though it is *almost* feasible by a generic framework of actively secure MPC, we require a new permutation (Section 6.1) and *verifying permutations* protocol (Section 6.2) in addition.

By known frameworks of actively secure MPC with abort [13, 32, 35], we assume that our protocols are in the flow of the following three phases:

- **Randomization phase.** For any share $[\![a]\!]$, parties compute $[\![ra]\!]$ with random secret r and store $([\![a]\!], [\![ra]\!])$ as the share of *a with a MAC.*
- **Computation phase.** Parties compute a target function F on input $([\![a]\!], [\![ra]\!])$ while recording checksums. For example, let $F = f_0 \circ f_1$ and assume $\mathit{\Pi}_{f_0}, \mathit{\Pi}_{f_1}$ that are protocols used to compute $([\![f_{\{0,1\}}(a)]\!], [\![rf_{\{0,1\}}(a)]\!])$ from $([\![a]\!], [\![ra]\!])$ and are futher secure up to additive attacks [25, 24]. Now, the parties allocate a set of checksums $\mathcal{S} = \emptyset$ and perform $\mathit{\Pi}_{f_2}$ and $\mathit{\Pi}_{f_1}$ in sequence to obtain $[\![F(a)]\!]$ while storing all inputs and outputs of the protocols, i.e., $([\![a]\!], [\![ra]\!])$, $([\![f_2(a)]\!], [\![rf_2(a)]\!])$ and $([\![F(a)]\!], [\![rF(a)]\!])$, into $\mathcal{S}$.

Algorithm 7 $\Pi_{\text{Reshuffle}}()$

```
Require: $p$ is the level s.t. all $\mathsf{HT}_{i<p}$ in the hierarchy is filled and $\mathsf{HT}_p$ is not.
Ensure: All $\mathsf{HT}_{i<p}$ in the hierarchy become empty, and $\mathsf{HT}_p$ is filled.
(Extracting all the data that needs to be reshuffled.)
1: Parties retrieve all $c$ blocks from the top-level array and let them be an array $[\![\mathbf{A}]\!]$.
2: For all $i = 1, \ldots, p-1$, parties call $\mathsf{HT}_i.\textsc{Extract}()$ with its state $(\pi_i, \langle \mathsf{T}_i \rangle_0, \langle \mathsf{T}_i \rangle_1)$ to
   extract all real elements as $[\![\mathsf{D}_i]\!]$ and concatenate them to $[\![\mathbf{A}]\!]$ as $[\![\mathbf{A}]\!] = [\![\mathbf{A}]\!] \parallel [\![\mathsf{D}_i]\!]$.
3: For all elements $[\![\widetilde{\mathsf{d}}_j]\!] = ([\![\mathsf{d}_j]\!], [\![\mathsf{lv}_j]\!])$ of $[\![\mathbf{A}]\!]$, parties call $\mathcal{F}_{\text{Trunc}}$ to set the $p$-least
   significant bits of $\mathsf{lv}_j$ to 0.
(Building a new cuckoo hash table.)
4: Parties call $\mathsf{HT}_p.\textsc{Build}([\![\mathbf{A}]\!])$ to construct a distributed hash table
   $(\pi_p, (\langle \mathsf{T}_p \rangle_i, \langle \mathsf{S}_p \rangle_i)_{(i \in \{0,1\})}, [\![s_{p,0}]\!], [\![s_{p,1}]\!], \mathsf{ctr}, \mathbf{P})$.
5: Parties call $\mathcal{F}_{\text{Reshare}}$ to convert $\langle \mathsf{S}_p \rangle_i$ to $[\![\mathsf{S}_p]\!]$ and store it to the top-level array.
6: For all $[\![\widetilde{\mathsf{d}}_u]\!] = ([\![\mathsf{d}_u]\!], [\![\mathsf{lv}_u]\!])$ in the top-level array, call $\mathcal{F}_{\text{Eq}}$ and $\mathcal{F}_{\text{IfElse}}$ to simulate:
       If $k_i \neq \perp$, set $[\![\mathsf{lv}_u]\!] = [\![\mathsf{lv}_u]\!] + [\![1]\!] 2^{p-1}$ .
```

– **Proof phase.** To detect cheating, parties evaluate the shares and their MACs recorded as checksums. Following the above example, the parties generate new random shares $[\![\rho_0]\!], [\![\rho_1]\!]$ and $[\![\rho_2]\!]$, and compute inner products

$$[\![\gamma]\!] = ([\![\rho_0]\!] \times [\![a]\!] + [\![\rho_1]\!] \times [\![f_2(a)]\!] + [\![\rho_2]\!] \times [\![F(a)]\!]) \text{ and}$$
$$[\![r\gamma]\!] = ([\![\rho_0]\!] \times [\![ra]\!] + [\![\rho_1]\!] \times [\![rf_2(a)]\!] + [\![\rho_2]\!] \times [\![rF(a)]\!]).$$

The parties recover $[\![\eta]\!] = [\![r]\!] \times [\![\gamma]\!] - [\![r\gamma]\!]$, and if $\eta \neq 0$ then they abort.

For our DORAM, we should be more concerned about the form of shares than in the passive security model. Since part of building blocks in Section 3, e.g., $\mathcal{F}_{\text{PRF}}$, requires bit-wise operations, we should assign MACs to bits $a_{\ell-1}, \ldots, a_0 \in \mathbb{Z}_2$, instead of the whole $a \in \mathbb{Z}_{2^\ell}$. According to Kikuchi et al. [35], we can provide a MAC for a bit $u \in \{0,1\}$ as $[\![ru]\!] := ([\![r_{\kappa-1}u]\!], \ldots, [\![r_0 u]\!])$ where each r_j is in $\mathbb{Z}_2$. To detect cheating with overwhelming probability $1 - \lambda^{\omega(1)}$, this κ should be $\omega(\log \lambda)$. Hence, by encoding $([\![a]\!], [\![ra]\!])$ as $\big(([\![a_{\ell-1}]\!], \ldots, [\![a_0]\!]), ([\![ra_{\ell-1}]\!], \ldots, [\![ra_{\ell-1}]\!])\big)$, the $\mathcal{F}_{\text{ABB}}$ functionality described in Section 3 is also available in active security model. Note that the communication and computation cost of each function increases to $O(\ell')$ where $\ell' = \omega(\ell \log \lambda)$ is the bit-length of a share with MAC, and the cost of the Proof phase is at most twice that of an original function [13].

To simplify the notation, we denote $([\![a]\!], [\![ra]\!])$ as $[\![a]\!]^{\mathsf{m}}$ in the following.

6.1 Secure Oblivious Permutation Up To Additive Attacks

We start at constructing a secure permutation protocol up to additive attacks. In contrast to Section 4.1, we assume that a permutation π provided by one party has been already separated into π_0 and π_1 s.t. $\pi = \pi_1 \circ \pi_0$ and shared between parties. We discuss verification for the permutation π in Section 6.2, and here we focus on cheating on an array that should be permuted.

Our permutation protocol described in Algorithm 1 relies on (semi-)honest parties and is difficult to be converted to be secure up to additive attacks. Instead, we construct a permutation protocol using a reshare-based shuffling as in Ikarashi et al. [32]. Let $\mathcal{F}_{\text{Shuffle}}$ be a functionality for secure shuffling up to an additive attack s.t. it receives shares of an array $[\![\mathbf{I}]\!]$ and a permutation π from honest parties and an array Δ of the same size as $\mathbf{I}$ from an adversary, then it returns $[\![\pi \cdot \mathbf{I} + \Delta]\!]$. Now, the following protocol, originally proposed by Laur et al. [43], is the implementation of $\mathcal{F}_{\text{Shuffle}}$ that costs n calls of $\mathcal{F}_{\text{Reshare}}$.

$\underline{[\![\mathbf{I}']\!]_0, [\![\mathbf{I}']\!]_1, [\![\mathbf{I}']\!]_2 \leftarrow \Pi_{\text{Shuffle}}((\pi, [\![\mathbf{I}]\!]_0), (\pi, [\![\mathbf{I}]\!]_1), [\![\mathbf{I}]\!]_2)}$:

1. P_0 and P_1 convert their $[\![\mathbf{I}]\!]_{i\in\{0,1\}}$ to $\langle \mathbf{I} \rangle_i$ and compute $\langle \mathbf{I}' \rangle_i = \pi \cdot \langle \mathbf{I} \rangle_i$ each.
2. Parties call $\mathcal{F}_{\text{Reshare}}$ to obtain $[\![\mathbf{I}']\!]_{\{0,1,2\}}$ from $\langle \mathbf{I}' \rangle_{\{0,1\}}$.
3. Output $[\![\mathbf{I}']\!]_{\{0,1,2\}}$.

Algorithm 8 $[\![\mathbf{I}']\!]^{\mathsf{m}}_0, [\![\mathbf{I}']\!]^{\mathsf{m}}_1, [\![\mathbf{I}']\!]^{\mathsf{m}}_2 \leftarrow \Pi^{\text{active}}_{\text{Perm}}((\pi_0, \pi_1), [\![\mathbf{I}]\!]^{\mathsf{m}}_0, [\![\mathbf{I}]\!]^{\mathsf{m}}_1, [\![\mathbf{I}]\!]^{\mathsf{m}}_2)$

```
Notation: Let $P$ be the permuter and let $S_0, S_1$ be the storages. Let $\mathcal{S}$ be a set of
    checksums.
Require: $P$ has both $\pi_0, \pi_1$ and $[\![\mathbf{I}]\!]^{\mathsf{m}}_0$. Each $S_{i=0,1}$ has $\pi_i$ and $[\![\mathbf{I}]\!]^{\mathsf{m}}_{i+1}$.
Ensure: $\mathbf{I}' = \pi \cdot \mathbf{I}$ and $r\mathbf{I}' = \pi \cdot (r\mathbf{I})$ where $\pi = \pi_1 \circ \pi_0$.
1: Parties record their input shares into $\mathcal{S}$ as $\mathcal{S} = \mathcal{S} \cup \{[\![\mathbf{I}]\!]^{\mathsf{m}}\}$, and let $[\![\mathbf{I}'_0]\!]^{\mathsf{m}} = [\![\mathbf{I}]\!]^{\mathsf{m}}$.
2: for $i = 0, 1$ do
3:     Parties call $\mathcal{F}_{\text{Shuffle}}$ with inputs $\pi_i$ and $[\![\mathbf{I}'_i]\!]^{\mathsf{m}}$ to receive $[\![\mathbf{I}'_{i+1}]\!]^{\mathsf{m}} = [\![\pi_i \cdot \mathbf{I}'_i]\!]^{\mathsf{m}}$.
4:     Parties record their shares into $\mathcal{S}$ as $\mathcal{S} \leftarrow \mathcal{S} \cup \{[\![\mathbf{I}'_{i+1}]\!]^{\mathsf{m}}\}$.
5: Parties store $[\![\mathbf{I}'_2]\!]^{\mathsf{m}}_0, [\![\mathbf{I}'_2]\!]^{\mathsf{m}}_1, [\![\mathbf{I}'_2]\!]^{\mathsf{m}}_2$ as $[\![\mathbf{I}']\!]^{\mathsf{m}}_0, [\![\mathbf{I}']\!]^{\mathsf{m}}_1, [\![\mathbf{I}']\!]^{\mathsf{m}}_2$, respectively.
6: Return $[\![\mathbf{I}']\!]^{\mathsf{m}}_{\{0,1,2\}}$
```

Now, we achieve an actively secure permutation protocol for our distributed oblivious hashing in the $\mathcal{F}_{\text{Shuffle}}$-hybrid model. The detailed protocol is described in Algorithm 8. We can also obtain an actively secure *unpermutation* protocol, $\Pi^{\text{active}}_{\text{Unperm}}$, as an inverse of Algorithm 8.

6.2 Actively Secure Distributed Hashing

Even though an actively secure oblivious hashing can be obtained *almost* completely straightforwardly by replacing all functionalities in Algorithm 3, 4, and 5 by actively secure ones, it is still *not secure against a corrupted permuter* that can input an invalid permutation. We thus focus on solving this problem. The permutation π is *valid* if all keys are always found in their place, *whatever* π *actually is.* This means that the pseudorandom addresses can work as *witnesses* for the correctness of the hash table, i.e., we can verify π by checking whether

$$(\textit{one of two pseudorandom addresses computed from a non-dummy key}) \\ - (\textit{the actual address where the element resides})$$

is equal to 0 for all real keys. This verification can be done in the Build procedure as follows: For all real blocks $[\![\mathsf{d}_i]\!]$ located in $\mathsf{T}[\mathsf{addr}^{\mathsf{r}}_i]$ and assigned to the pseudorandom addresses $(\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket, \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket)$, parties check the following equation with uniformly random ρ_i.

$$0 =_? \sum_i \llbracket \rho_i \rrbracket \times (\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket - \mathsf{addr}^{\mathrm{r}}_i) \times (\llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket - \mathsf{addr}^{\mathrm{r}}_i). \tag{6.1}$$

In addition, for efficiency, we combine the above verification with the MAC verification. Remember that, in the Proof phase for MACs, fresh random shares $\llbracket \rho_i \rrbracket$ are given for each checksum $(\llbracket a \rrbracket, \llbracket ra \rrbracket)$. Noticing the similarity in the use of the random ρ_i, for any r, we can transform Equation (6.1) as below:

$$\llbracket r \rrbracket \times \sum_i \llbracket \rho_i \rrbracket \times (\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket \times \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket + (\mathsf{addr}^{\mathrm{r}}_i)^2) =_? \\ \sum_i \llbracket \rho_i \rrbracket \times (\llbracket r \times \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket + \llbracket r \times \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket) \times \mathsf{addr}^{\mathrm{r}}_i. \tag{6.2}$$

Now, to make parties evaluate the above equation in the Proof phase, we propose the following new protocol that checks the consistency between the virtual address and the actual address of each non-dummy block.

$\underline{\mathit{\Pi}_{\text{VerPerm}}(\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket^{\mathsf{m}}, \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket^{\mathsf{m}}, \mathsf{addr}^{\mathrm{r}}_i)}$:

1. At first, parties call the actively secure $\mathcal{F}_{\text{Mult}}$ to obtain
$$\llbracket \mathsf{addr}'_i \rrbracket^{\mathsf{m}} = (\llbracket \mathsf{addr}'_i \rrbracket, \llbracket r \times \mathsf{addr}'_i \rrbracket) = \llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket^{\mathsf{m}} \times \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket^{\mathsf{m}}.$$
$\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket^{\mathsf{m}}, \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket^{\mathsf{m}}$ and $\llbracket \mathsf{addr}'_i \rrbracket^{\mathsf{m}}$ are recorded in $\mathcal{S}$ as checksums.
2. Parties locally compute below and record $\llbracket \mathsf{ver}^{(1)}_i \rrbracket^{\mathsf{m}}$ and $\llbracket \mathsf{ver}^{(2)}_i \rrbracket^{\mathsf{m}}$ to $\mathcal{S}$.
$\llbracket \mathsf{ver}^{(1)}_i \rrbracket^{\mathsf{m}} := (\llbracket \mathsf{addr}'_i \rrbracket + (\mathsf{addr}^{\mathrm{r}}_i)^2, (\llbracket r \times \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket + \llbracket r \times \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket) \times \mathsf{addr}^{\mathrm{r}}_i)$, and
$\llbracket \mathsf{ver}^{(2)}_i \rrbracket^{\mathsf{m}} := ((\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket + \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket) \times \mathsf{addr}^{\mathrm{r}}_i, \llbracket r \times \mathsf{addr}'_i \rrbracket + \llbracket r \rrbracket \times (\mathsf{addr}^{\mathrm{r}}_i)^2)$.

Since whether the permuter has provided a valid permutation is equivalent to whether $\mathsf{addr}^{(1)}_i = \mathsf{addr}^{(2)}_i$, the verification for *checksums* $\llbracket \mathsf{ver}^{(1)}_i \rrbracket^{\mathsf{m}}$ and $\llbracket \mathsf{ver}^{(2)}_i \rrbracket^{\mathsf{m}}$ includes Equation (6.2) and its transformation. Furthermore, since the input and output of $\mathcal{F}_{\text{Mult}}$, $\llbracket \mathsf{addr}^{\mathrm{p}}_{i,0} \rrbracket^{\mathsf{m}}, \llbracket \mathsf{addr}^{\mathrm{p}}_{i,1} \rrbracket^{\mathsf{m}}$ and $\llbracket \mathsf{addr}'_i \rrbracket^{\mathsf{m}}$, are recorded as checksums, we can attribute the attack providing an invalid permutation to an additive attack that modifies $\mathsf{addr}^{\mathrm{r}}_i$ to $\mathsf{addr}^{\mathrm{r}}_i + \delta$ for some δ.

Now, we describe the detailed algorithms of our actively secure hashing scheme in Algorithm 9, 10, and 11. To simplify notation, for each block $\llbracket \mathsf{d} \rrbracket = (\llbracket k \rrbracket, \llbracket v \rrbracket)$, let $\llbracket \mathsf{d} \rrbracket^{\mathsf{m}} = \big((\llbracket k \rrbracket, \llbracket rk \rrbracket), (\llbracket v \rrbracket, \llbracket rv \rrbracket)\big)$. This is necessary and sufficient form of MACs for our protocols including functions applied only to k. For the original block size b, the block size of $\llbracket \mathsf{d} \rrbracket^{\mathsf{m}}$ becomes $b' = \omega(b \log \lambda)$ because of MACs (see the beginning of Section 6). The parameters for the Cuckoo hash table, τ, σ, and n', are the same as those in passive security.

Lemma 6.1. *There exists a 3-party distributed hash table, described as Algorithms 9, 10 and 11, that securely realizes $\mathcal{F}_{\mathsf{HT}}$ in the $(\mathcal{F}_{\text{Share}}, \mathcal{F}^{\mathcal{P}}_{\text{Reveal}}, \mathcal{F}_{\text{ABB}})$-hybrid model in the presence of an active adversary that controls one party.*

Assume that the input array consists of n blocks and each block is b' bits.

- *The* Build *procedure consumes $O(n(b' + \lceil \log n \rceil))$ local computation steps.*
- *The* Lookup *procedure consumes $O(b' + b' \log \lambda)$ local computation steps.*

Algorithm 9

$\pi_0, \pi_1, (\langle \mathsf{T} \rangle_i^{\mathsf{m}}, \langle \mathsf{S} \rangle_i^{\mathsf{m}})_{(i \in \{0,1\})}, [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P} \leftarrow \mathit{\Pi}_{\text{BUILD}}^{\text{active}}([\![\mathsf{D}]\!]_1^{\mathsf{m}}, [\![\mathsf{D}]\!]_2^{\mathsf{m}}, [\![\mathsf{D}]\!]_3^{\mathsf{m}})$

```
Require: Parties have shares of a dataset $[\![\mathsf{D}]\!]^{\mathsf{m}}_{i\in\{0,1,2\}} = ([\![\mathsf{d}_0]\!]^{\mathsf{m}}_i, \ldots, [\![\mathsf{d}_{n-1}]\!]^{\mathsf{m}}_i)$.
Ensure: Each $S_i$ obtains a permuation $\pi_i$ and a $(1,2)$-share of cuckoo hash table, $(\langle \mathsf{T} \rangle_i, \langle \mathsf{S} \rangle_i)$. $P$ obtains permutations $\pi_0, \pi_1$, and all parties hold $(1,3)$-shares of PRF keys, $[\![s_0]\!]$ and $[\![s_1]\!]$. Parties also hold a query counter $\mathsf{ctr}$, and $S_0, S_1$ hold a set $\mathbf{P}$, as their states.
(Computing pseudorandom addresses and constructing a table.)
 1: Parties do the same as Line 1 to 7 of Algorithm 3 with actively secure $\mathcal{F}_{\text{ABB}}$.
    Let $[\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}} = [\![\mathsf{D} \parallel \mathsf{E}]\!]^{\mathsf{m}}$ be the result of the Line 7 and $[\![\widetilde{\mathsf{A}}]\!]^{\mathsf{m}} = [\![\mathsf{A} \parallel \mathsf{E}]\!]^{\mathsf{m}}$ be the array of addresses where $[\![\mathsf{A}]\!]^{\mathsf{m}} = \big(([\![\mathsf{addr}_{0,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}_{0,1}]\!]^{\mathsf{m}}), \ldots, ([\![\mathsf{addr}_{n-1,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}_{n-1,1}]\!]^{\mathsf{m}})\big)$ is non-dummy addresses corresponding to the input $[\![\mathsf{D}]\!]^{\mathsf{m}}$.
 2: Before revealing $\mathsf{A}$ for $P$, parties perform the Proof phase to verify the checksums.
    Then, $P$ receives all non-dummy addresses to compute a permutation $\pi$.
 3: $P$ computes $\pi_0$ and $\pi_1$ s.t. $\pi_1 \circ \pi_0 = \pi$, and sends them to $S_0$ and $S_1$ respectively.
    If $\pi_{i\in\{0,1\}}$ is not a permutation, $S_i$ aborts.
 4: Parties run $\mathit{\Pi}_{\text{PERM}}^{\text{active}}$ twice to obtain both $[\![\widetilde{\mathsf{D}}']\!]^{\mathsf{m}} = [\![\pi \cdot \widetilde{\mathsf{D}}]\!]^{\mathsf{m}}$ and $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}} = [\![\widetilde{\pi \cdot \mathsf{A}}]\!]^{\mathsf{m}}$.
(Verifying $\pi$.)
 5: for all $i \in [\tau]$ do
 6:     Let $[\![\widetilde{\mathsf{d}}'_i]\!]^{\mathsf{m}} = ([\![\widetilde{k}'_i]\!]^{\mathsf{m}}, [\![\widetilde{v}'_i]\!]^{\mathsf{m}})$ be the $i$-th element of $[\![\widetilde{\mathsf{D}}']\!]^{\mathsf{m}}$.
 7:     Let $([\![\widetilde{\mathsf{addr}}'_{i,0}]\!]^{\mathsf{m}}, [\![\widetilde{\mathsf{addr}}'_{i,1}]\!]^{\mathsf{m}})$ be the $i$-th element of $[\![\widetilde{\mathsf{A}}']\!]^{\mathsf{m}}$ and let $\mathsf{addr}^{\mathsf{r}}_i := i$.
 8:     Parties call $\mathcal{F}_{\text{EQ}}$ and $\mathcal{F}_{\text{IFELSE}}$ to simulate:
            If $\widetilde{k}'_i = \perp$ then set $([\![\mathsf{addr}^{\mathsf{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathsf{p}}_{i,1}]\!]^{\mathsf{m}}) = ([\![\mathsf{addr}^{\mathsf{r}}_i]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathsf{r}}_i]\!]^{\mathsf{m}})$,
            otherwise set $([\![\mathsf{addr}^{\mathsf{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathsf{p}}_{i,1}]\!]^{\mathsf{m}}) = ([\![\widetilde{\mathsf{addr}}'_{i,0}]\!]^{\mathsf{m}}, [\![\widetilde{\mathsf{addr}}'_{i,1}]\!]^{\mathsf{m}})$.
 9:     Parties perform $\mathit{\Pi}_{\text{VERPERM}}([\![\mathsf{addr}^{\mathsf{p}}_{i,0}]\!]^{\mathsf{m}}, [\![\mathsf{addr}^{\mathsf{p}}_{i,1}]\!]^{\mathsf{m}}, \mathsf{addr}^{\mathsf{r}}_i)$.
10: Parties perform the Proof phase to verify the checksums.
(Organizing a cuckoo hash table and stash.)
11: Each $S_{i\in\{0,1\}}$ converts $[\![\widetilde{\mathsf{D}}']\!]^{\mathsf{m}}$ to $\langle \widetilde{\mathsf{D}}' \rangle^{\mathsf{m}}_i$ and organize it into a hash table $\langle \mathsf{T} \rangle^{\mathsf{m}}_i$ and
    stash $\langle \mathsf{S} \rangle^{\mathsf{m}}_i$, by separating $\langle \widetilde{\mathsf{D}}' \rangle^{\mathsf{m}}_i$ into the first $\tau$ and the last $\sigma$ elements.
12: Parties set their state $\mathsf{ctr} = 0$, and $S_0, S_1$ allocate an empty set $\mathbf{P} = \emptyset$.
13: Return $(\pi_0, \pi_1), (\langle \mathsf{T} \rangle^{\mathsf{m}}_i, \langle \mathsf{S} \rangle^{\mathsf{m}}_i)_{(i\in\{0,1\})}, [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}, \mathsf{ctr}, \mathbf{P}$
```

– *The* EXTRACT *procedure consumes $O(nb')$ local computation steps.*

Furthermore, all those procedures consumes $O(1)$ communication rounds.

Proof. The proof of security is given in Appendix B.3. To proceed with the efficiency analysis, we start at the following claim.

Claim. $\mathit{\Pi}_{\text{PERM}}^{\text{active}}$ consumes $O(nb')$ local computation steps and $O(1)$ communication rounds. In addition, $\mathit{\Pi}_{\text{VERPERM}}$ consumes $O(b')$ local computation steps and $O(1)$ communication rounds.

Each claim is clear from the algorithms of $\mathit{\Pi}_{\text{PERM}}^{\text{active}}$ and $\mathit{\Pi}_{\text{VERPERM}}$.

The difference between the passively and actively secure BUILD procedure (Algorithm 3 and 9) is that the latter uses $\mathit{\Pi}_{\text{PERM}}^{\text{active}}$ instead of $\mathcal{F}_{\text{PERM}}$ and additionally runs $\mathcal{F}_{\text{EQ}}, \mathcal{F}_{\text{IFELSE}}, \mathit{\Pi}_{\text{VERPERM}}$ (each) τ times and the Proof phases twice. Since τ and n' are each $O(n)$, this difference increases the computation cost at

```
Algorithm 10
$[\![\mathsf{d}]\!]^{\mathsf{m}}, [\![\mathsf{found}]\!]^{\mathsf{m}} \leftarrow \Pi^{\text{active}}_{\text{LOOKUP}}([\![k]\!]^{\mathsf{m}}, (\langle \mathsf{T} \rangle^{\mathsf{m}}_0, \langle \mathsf{S} \rangle^{\mathsf{m}}_0), (\langle \mathsf{T} \rangle^{\mathsf{m}}_1, \langle \mathsf{S} \rangle^{\mathsf{m}}_1), [\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}})$
Require: 2-out-of-3 shares of a queried key $[\![k]\!]^{\mathsf{m}}$, 2-out-of-2 shares of the hash table
    $\langle \mathsf{T} \rangle^{\mathsf{m}}_i$ and its stash $\langle \mathsf{S} \rangle^{\mathsf{m}}_i$, and 2-out-of-3 shares of PRF keys $[\![s_0]\!]^{\mathsf{m}}, [\![s_1]\!]^{\mathsf{m}}$.
Ensure: $\mathsf{d} = (k, v)$ and $\mathsf{found} = 1$ if the table $\mathsf{T}$ contains an item $(k, v)$, otherwise
    $\mathsf{d} = (0, 0)$ and $\mathsf{found} = 0$.
(Computing pseudorandom addresses to be fetched.)
1: To obtain pseudorandom addresses $[\![\mathsf{addr}_0]\!]^{\mathsf{m}}$ and $[\![\mathsf{addr}_1]\!]^{\mathsf{m}}$, parties do the same as
    Line 1 to 3 of Algorithm 4 with actively secure $\mathcal{F}_{\text{ABB}}$.
(Searching for the cuckoo hash table and stash.)
2: Before revealing $\mathsf{addr}_0$ and $\mathsf{addr}_1$ to the storages, parties perform the Proof phase
    to verify the checksums.
3: To obtain a queried data $[\![\mathsf{d}]\!]^{\mathsf{m}}$ and flag $[\![\mathsf{found}]\!]^{\mathsf{m}}$, parties do the same as Line 5 to 13
    of Algorithm 4 with actively secure $\mathcal{F}_{\text{ABB}}$.
4: Return $[\![\mathsf{d}]\!]^{\mathsf{m}}, [\![\mathsf{found}]\!]^{\mathsf{m}}$
```

```
Algorithm 11 $[\![\mathsf{D}]\!]^{\mathsf{m}}_0, [\![\mathsf{D}]\!]^{\mathsf{m}}_1, [\![\mathsf{D}]\!]^{\mathsf{m}}_2 \leftarrow \Pi^{\text{active}}_{\text{EXTRACT}}((\pi_0, \pi_1), (\langle \mathsf{T} \rangle^{\mathsf{m}}_0, \langle \mathsf{S} \rangle^{\mathsf{m}}_0), (\langle \mathsf{T} \rangle^{\mathsf{m}}_1, \langle \mathsf{S} \rangle^{\mathsf{m}}_1))$
Require: 2-out-of-2 shares of a hash table $\mathsf{T}$ and stash $\mathsf{S}$, and their constructing
    permutation $(\pi_0, \pi_1)$.
Ensure: A dataset $\mathsf{D}$ contains all real elements in $\mathsf{T}$ and $\mathsf{S}$.
1: Each $S_{i \in \{0,1\}}$ reorganizes $\langle \mathsf{T} \rangle^{\mathsf{m}}_i$ and $\langle \mathsf{S} \rangle^{\mathsf{m}}_i$ into an array $\langle \widetilde{\mathsf{D}} \rangle^{\mathsf{m}}_i$ as $\widetilde{\mathsf{D}} = \mathsf{T} \parallel \mathsf{S}$.
2: Parties call actively secure $\mathcal{F}_{\text{RESHARE}}$ to convert $\langle \widetilde{\mathsf{D}} \rangle^{\mathsf{m}}_0, \langle \widetilde{\mathsf{D}} \rangle^{\mathsf{m}}_1$ to $[\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_0, [\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_1, [\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_2$.
3: Parties perform $\Pi^{\text{active}}_{\text{UNPERM}}$ with $(\pi_0^{-1}, \pi_1^{-1}, [\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_0)$ of $P$, $(\pi_0^{-1}, [\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_1)$ of $S_0$ and
    $(\pi_1^{-1}, [\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}_2)$ of $S_1$, to obtain $[\![\pi^{-1} \cdot \widetilde{\mathsf{D}}]\!]_{i \in \{0,1,2\}}$.
4: Let $[\![\mathsf{D}]\!]_i$ be the first $n$ elements of $[\![\widetilde{\mathsf{D}'}]\!]_i$.
5: Return $[\![\mathsf{D}]\!]_0, [\![\mathsf{D}]\!]_1, [\![\mathsf{D}]\!]_2$
```

most $O(nb')$ from Algorithm 3. On the other hand, since Algorithm 11 requires performing $\Pi^{\text{active}}_{\text{PERM}}$ at once, it costs $O(nb')$ steps. Moreover, since the difference between the passively and actively secure LOOKUP procedure (Algorithm 4 and 10) is only that the latter additionally requires to verify $[\![\mathsf{addr}_{\{0,1\}}]\!]^{\mathsf{m}}$, the computational cost increases by at most $O(b')$ steps from Algorithm 4. □

6.3 Actively Secure Distributed ORAM

Given our actively secure distributed oblivious hashing, we achieve an actively secure DORAM.

Theorem 6.2. *There exists a 3-party protocol that securely realizes $\mathcal{F}_{\text{RAM}}$ in the presence of an active adversary that controls one party. In addition, assuming $N = \mathsf{poly}(\lambda)$, this implementation consumes $O(\log N)$ amortized overhead and $O(\log N)$ communication rounds with $\omega(\log^2 N)$ block size.*

Proof. By straightforward composition, we can replace all functionalities related to $\mathcal{F}_{\text{ABB}}$ and $\mathcal{F}_{\mathsf{HT}}$ in Algorithms 6 and 7 with their concrete implementations and thereby get a concrete actively secure DORAM. We can also achieve the security proof by considering the same simulator as of Appendix B.2 except that it uses an actively secure distributed oblivious hashing simulator. Moreover, since the efficiency of each $\mathcal{F}_{\mathrm{ABB}}$ and $\mathcal{F}_{\mathsf{HT}}$ is asymptotically the same as in passive security except for the increased block size, the actively secure DORAM achieves $O(\log N)$ overhead using a slightly larger block size $b' = \omega(\log^2 N)$. $\square$

7 Conclusion

We proposed an optimal DORAM of $O(\log N)$ overhead with small hidden constant, that relies on only $4 \log N$ OPRF calls. The key building block is a novel 3-party permutation protocol, in which parties split their roles into one permuter that knows the whole permutation and two storages that hold a permuted table. Since these roles do not overlap, the permuter never observes access patterns to the permuted table even though it knows the structure of the table.

In addition, we extended the above (passively secure) DORAM to an actively secure one. Since our passively secure construction depends on the permutation protocol in which one party has full control of a permutation, we additionally construct a novel protocol to verify the permutation provided by the (possibly dishonest) party. Then, we achieved an actively secure DORAM of $O(\log N)$ overhead with slightly larger $\omega(\log^2 N)$-bit block size.

References

1. Albrecht, M.R., Rechberger, C., Schneider, T., Tiessen, T., Zohner, M.: Ciphers for mpc and fhe. In: EUROCRYPT. pp. 430–454 (2015)
2. Araki, T., Furukawa, J., Lindell, Y., Nof, A., Ohara, K.: High-throughput semi-honest secure three-party computation with an honest majority. In: CCS. pp. 805–817 (2016)
3. Asharov, G., Komargodski, I., Lin, W., Nayak, K., Peserico, E., Shi, E.: Optorama: Optimal oblivious RAM. J. ACM **70**(1), 4:1–4:70 (2023)
4. Asharov, G., Komargodski, I., Lin, W., Peserico, E., Shi, E.: Optimal oblivious parallel RAM. In: ACM-SIAM Symposium on Discrete Algorithms, SODA. pp. 2459–2521 (2022)
5. Asharov, G., Komargodski, I., Lin, W., Shi, E.: Oblivious RAM with worst-case logarithmic overhead. In: CRYPTO. pp. 610–640 (2021)
6. Blass, E.O., Mayberry, T., Noubir, G.: Multi-client oblivious ram secure against malicious servers. In: ACNS. pp. 686–707 (2017)
7. Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: ITCS. pp. 357–368 (2016)
8. Bunn, P., Katz, J., Kushilevitz, E., Ostrovsky, R.: Efficient 3-party distributed oram. Cryptology ePrint Archive (2018)
9. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptology **13**(1), 143–202 (2000)
10. Catrina, O., de Hoogh, S.: Improved primitives for secure multiparty integer computation. In: Garay, J.A., De Prisco, R. (eds.) SCN. pp. 182–199 (2010)

11. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient oram and opram. In: ASIACRYPT. pp. 660–690 (2017)
12. Chan, T.H., Shi, E.: Circuit OPRAM: unifying statistically and computationally secure orams and oprams. In: TCC. pp. 72–107 (2017)
13. Chida, K., Genkin, D., Hamada, K., Ikarashi, D., Kikuchi, R., Lindell, Y., Nof, A.: Fast large-scale honest-majority mpc for malicious adversaries. In: CRYPTO. pp. 34–64 (2018)
14. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Kiribuchi, N., Pinkas, B.: An efficient secure three-party sorting protocol with an honest majority. Cryptology ePrint Archive (2019)
15. Chida, K., Hamada, K., Ikarashi, D., Kikuchi, R., Pinkas, B.: High-throughput secure aes computation. In: WAHC. p. 13–24 (2018)
16. Cramer, R., Damgård, I., Ishai, Y.: Share conversion, pseudorandom secret-sharing and applications to secure computation. In: TCC. pp. 342–362 (2005)
17. Damgård, I., Keller, M.: Secure multiparty aes. In: FC. pp. 367–374 (2010)
18. Devadas, S., van Dijk, M., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion oram: A constant bandwidth blowup oblivious ram. In: TCC. pp. 145–174 (2016)
19. Dittmer, S., Ostrovsky, R.: Oblivious tight compaction in o(n) time with smaller constant. In: SCN. pp. 253–274 (2020)
20. Doerner, J., Shelat, A.: Scaling ORAM for secure computation. In: CCS. pp. 523–535 (2017)
21. Faber, S., Jarecki, S., Kentros, S., Wei, B.: Three-party ORAM for secure computation. In: ASIACRYPT. pp. 360–385 (2015)
22. Falk, B.H., Noble, D., Ostrovsky, R.: Alibi: A flaw in cuckoo-hashing based hierarchical ORAM schemes and a solution. In: Advances in Cryptology - EUROCRYPT. pp. 338–369 (2021)
23. Falk, B.H., Noble, D., Ostrovsky, R.: 3-party distributed ORAM from oblivious set membership. In: SCN. pp. 437–461 (2022)
24. Genkin, D., Ishai, Y., Polychroniadou, A.: Efficient multi-party computation: From passive to active security via secure simd circuits. In: CRYPTO. pp. 721–741 (2015)
25. Genkin, D., Ishai, Y., Prabhakaran, M.M., Sahai, A., Tromer, E.: Circuits resilient to additive attacks with applications to secure computation. In: STOC. pp. 495–504 (2014)
26. Goldreich, O.: Towards a theory of software protection and simulation by oblivious rams. In: STOC. pp. 182–194 (1987)
27. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3), 431–473 (1996)
28. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious ram simulation. In: ICALP. pp. 576–587 (2011)
29. Hoang, T., Guajardo, J., Yavuz, A.A.: MACAO: A maliciously-secure and client-efficient active ORAM framework. In: NDSS (2020)
30. Hoang, T., Ozkaptan, C.D., Yavuz, A.A., Guajardo, J., Nguyen, T.: S3oram: A computation-efficient and constant client bandwidth blowup oram with shamir secret sharing. In: CCS. pp. 491–505 (2017)
31. Håstad, J., Impagliazzo, R., Levin, L.A., Luby, M.: A pseudorandom generator from any one-way function. J. Computing **28**(4), 1364–1396 (1999)
32. Ikarashi, D., Kikuchi, R., Hamada, K., Chida, K.: Actively private and correct mpc scheme in $t < n/2$ from passively secure schemes with small overhead. Cryptology ePrint Archive (2014)

33. Ito, M., Saito, A., Nishizeki, T.: Secret sharing scheme realizing general access structure. GLOBECOM pp. 99–102 (1987)
34. Jacob, R., Larsen, K.G., Nielsen, J.B.: Lower bounds for oblivious data structures. In: SODA. pp. 2439–2447 (2019)
35. Kikuchi, R., Attrapadung, N., Hamada, K., Ikarashi, D., Ishida, A., Matsuda, T., Sakai, Y., Schuldt, J.C.N.: Field extension in secret-shared form and its applications to efficient secure computation. In: ACISP. pp. 343–361 (2019)
36. Kirsch, A., Mitzenmacher, M., Wieder, U.: More robust hashing: Cuckoo hashing with a stash. J. Computing **39**(4), 1543–1561 (2009)
37. Komargodski, I., Lin, W.: A logarithmic lower bound for oblivious RAM (for all parameters). In: CRYPTO. pp. 579–609 (2021)
38. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in)security of hash-based oblivious ram and a new balancing scheme. In: SODA. pp. 143–156 (2012)
39. Kushilevitz, E., Mour, T.: Sub-logarithmic distributed oblivious ram with small block size. In: PKC. pp. 3–33 (2019)
40. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: CRYPTO. pp. 523–542 (2018)
41. Larsen, K.G., Simkin, M., Yeo, K.: Lower bounds for multi-server oblivious rams. In: TCC. pp. 486–503 (2020)
42. Laur, S., Talviste, R., Willemson, J.: From oblivious aes to efficient and secure database join in the multiparty setting. In: ACNS. pp. 84–101 (2013)
43. Laur, S., Willemson, J., Zhang, B.: Round-efficient oblivious database manipulation. In: ISC. pp. 262–277 (2011)
44. Lu, S., Ostrovsky, R.: Distributed oblivious ram for secure two-party computation. In: TCC. pp. 377–396 (2013)
45. Maffei, M., Malavolta, G., Reinert, M., Schröder, D.: Maliciously secure multi-client oram. Cryptology ePrint Archive (2017)
46. Naor, M.: Bit commitment using pseudo-randomness (extended abstract). In: CRYPTO. p. 128–136 (1989)
47. Noble, D.: An intimate analysis of cuckoo hashing with a stash. Cryptology ePrint Archive (2021)
48. Ostrovsky, R.: Efficient computation on oblivious rams. In: STOC. pp. 514–523 (1990)
49. Ostrovsky, R., Shoup, V.: Private information storage (extended abstract). In: STOC. pp. 294–303 (1997)
50. Pagh, R., Rodler, F.F.: Cuckoo hashing. J. Algorithms **51**(2), 122–144 (2004)
51. Patel, S., Persiano, G., Raykova, M., Yeo, K.: PanORAMa: Oblivious RAM with logarithmic overhead. In: FOCS. pp. 871–882 (2018)
52. Pippenger, N., Fischer, M.J.: Relations among complexity measures. J. ACM **26**(2), 361–381 (1979)
53. Ren, L., Fletcher, C.W., Yu, X., van Dijk, M., Devadas, S.: Integrity verification for path oblivious-ram. In: HPEC. pp. 1–6 (2013)
54. Shamir, A.: How to share a secret. Commun. ACM **22**(11), 612–613 (1979)
55. Shi, E., Chan, T.H., Stefanov, E., Li, M.: Oblivious RAM with o((logn)3) worst-case cost. In: ASIACRYPT. pp. 197–214 (2011)
56. Stefanov, E., van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: An extremely simple oblivious ram protocol. In: CCS. pp. 299–310 (2013)
57. Wang, X., Chan, T.H., Shi, E.: Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In: CCS. pp. 850–861 (2015)

A Concrete Efficiency of Our Passively Secure DORAM and Comparison with [44] and [23]

Functionality	$\mathcal{F}_{\text{Share}}$	$\mathcal{F}_{\text{Reveal}}$	$\mathcal{F}_{\text{Mult}}$	$\mathcal{F}_{\text{Rnd}}$	$\mathcal{F}_{\text{Reshare}}$	$\mathcal{F}_{\text{Eq}}$	$\mathcal{F}_{\text{IfElse}}$	$\mathcal{F}_{\text{Bitext}}$	$\mathcal{F}_{\text{Trunc}}$	$\mathcal{F}_{\text{Prf}}$
# of calls	$3L+1$	$4L$	$3L+c$	2	$2L+1$	$4L+c+1$	$5L+c+3$	L	L	$4L$

Functionality	$\mathcal{F}_{\text{Perm}}$	$\mathcal{F}_{\text{Unperm}}$
# of calls	$p-1$, with input of size $c2^i + \frac{c}{2}$ for each $i = 1, \ldots, p-1$ per $c2^{p-1}$ accesses.	1, with input of size $c2^p + \frac{c}{2}$ per $c2^{p-1}$ accesses.

Table 2. The amortized number of calls of each functionality in Algorithm 6, where $L = \lceil \log N - \log\log N \rceil$ and $c = 2\lceil \log N \rceil$.

We evaluate the concrete computational complexity of our passively secure DORAM. We treat each functionality as a black box here and consider the (amortized) number of calls per access, for ease of estimation[8]. Table 2 describes the concrete number of calls of the building blocks required in Algorithm 6. Each entry of the table can be straightforwardly calculated using the facts below:

- In the main part of Algorithm 6, parties call $\mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{Eq}}$, and $\mathcal{F}_{\text{IfElse}}$ c times (line 1 to 5), call $\mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{IfElse}}$, and $\mathcal{F}_{\text{Bitext}}$ L times (line 6 to 8), call $\mathcal{F}_{\text{IfElse}}$ 2 times (line 11 to 12), and perform H_i.Lookup for each $i = 1, \ldots, L$.
  - Regardless of i, H_i.Lookup described in Algorithm 4 requires 2 calls of $\mathcal{F}_{\text{Reveal}}, \mathcal{F}_{\text{Mult}}, \mathcal{F}_{\text{Reshare}}$, and $\mathcal{F}_{\text{Prf}}$, and 3 calls of $\mathcal{F}_{\text{Eq}}$ and $\mathcal{F}_{\text{IfElse}}$ (line 1 to 8)[9].
- In the subroutine $\Pi_{\text{Reshuffle}}$ described in Algorithm 7, for each $c2^{p-1}$ accesses, parties perform H_i.Extract for all $i = 1, \ldots, p-1$ and H_p.Build, call $\mathcal{F}_{\text{Trunc}}$ $c2^{p-1}$ times, and call $\mathcal{F}_{\text{Reshare}}, \mathcal{F}_{\text{Eq}}$, and $\mathcal{F}_{\text{IfElse}}$ $\frac{c}{2}$ times.
  - H_i.Extract described in Algorithm 5, where the capacity of H_i is $c2^{i-1}$, requires $\mathcal{F}_{\text{Unperm}}$ at once with input size $|\mathsf{T} \parallel \mathsf{S}| = c2^i + \frac{c}{2}$.
  - H_p.Build described in Algorithm 3, where the capacity of H_p is $c2^{p-1}$, requires 2 calls of $\mathcal{F}_{\text{Rnd}}$, $c2^{p-1}$ calls of $\mathcal{F}_{\text{Eq}}$ and $\mathcal{F}_{\text{IfElse}}$, $c2^p$ calls of $\mathcal{F}_{\text{Reveal}}$ and $\mathcal{F}_{\text{Prf}}$, $c2^{p-1} + \frac{c}{2}$ calls of $\mathcal{F}_{\text{Share}}$, and a call of $\mathcal{F}_{\text{Perm}}$ with input size $c2^p + \frac{c}{2}$.

Comparing practical efficiency with [44] and [23]. Following a similar setting as Falk et al. [23], we can estimate the efficiency of our DORAM more concretely. From our Table 2 and Lemmas 4.1, 4.2, and Table 2 of [23], we conclude

[8] See Table 2 in [23], for example, for more specific costs of each functionality.

[9] Note that we can omit the stash-related operations (line 9 to 12) of H_i.Lookup in Algorithm 6.

that Algorithm 6 requires $(164L + 9c + 40)B$ bits of communication for block size $B = \Omega(\log N)$ bits. Considering $L \sim \log N$ and $c \sim 2\log N$, the cost of our DORAM becomes $(182\log N + 40)B$ bits per access.

According to [23], the DORAM of Lu and Ostrovsky [44] requires at least $100\log N$ OPRF calls and hence costs at least $2100\log N \cdot B$ bits of communication per access[10] — this is significantly higher than ours. On the other hand, the DORAM of Falk et al. [23] requires $(153.5\log N + 6)B$ bits per access. I.e., even though the number of our calls to the most expensive building block is double, it does not affect the practical bandwidth much. Moreover, considering that the DORAM of Falk et al. requires additional computational cost for $\log^2 N$ non-oblivious hash functions per access, our optimal DORAM appears to be faster in the practical size of N.

B Security Proofs

B.1 Proof of Security and Correctness of Our Passively Secure Distributed Oblivious Hashing (Lemma 4.3)

For correctness of our scheme, by [47], we observe that the permuter P achieves a permutation π for hashing with overwhelming probability. Since we are in the $\mathcal{F}_{\text{PERM}}$-hybrid model, all input blocks are correctly placed in their location as $\pi \cdot \mathsf{D} = \mathsf{T}\|\mathsf{S}$ in Algorithm 3. Similarly, since we are in the $\mathcal{F}_{\text{UNPERM}}$-hybrid model, the blocks are correctly retrieved as $\mathsf{D} = \pi^{-1} \cdot (\mathsf{T}\|\mathsf{S})$ in Algorithm 5. Moreover, when the above permutation π exists, in Algorithm 4, the storages always fetch $\mathsf{T}[\mathsf{addr}_0], \mathsf{T}[\mathsf{addr}_1]$, and S, one block of which is the queried data.

For obliviousness, consider a simulator Sim that simulates the view of a passive adversary that corrupts one of three party in $\Pi_{\text{BUILD}}, \Pi_{\text{LOOKUP}}$ and Π_{EXTRACT}.

- **Simulating $\Pi_{\textbf{Build}}$.** Receiving an instruction to simulate Π_{BUILD} with the size of input n and a collusion target $C \in \{P, S_0, S_1\}$, Sim runs the real protocol Π_{BUILD} on input n dummy blocks. If $C = P$ then Sim outputs π and all $\mathsf{addr}_{i,j}$ provided from $\mathcal{F}_{\text{PRP}}([\![s_j]\!], [\![\perp +i]\!])$ as *access patterns*. Otherwise, Sim outputs nothing. Sim holds the result of Π_{BUILD} simulation, $(\pi, (\langle\mathsf{T}\rangle_{\{0,1\}}, \langle\mathsf{S}\rangle_{\{0,1\}}), [\![s_{\{0,1\}}]\!], \mathsf{ctr}, \mathbf{P})$, as its state.
- **Simulating $\Pi_{\textbf{Lookup}}$.** When the adversary requests to run Π_{LOOKUP} with a key k, Sim simulates the real protocol Π_{LOOKUP} with a key $\perp$ and its state $(\pi, (\langle\mathsf{T}\rangle_{\{0,1\}}, \langle\mathsf{S}\rangle_{\{0,1\}}), [\![s_{\{0,1\}}]\!], \mathsf{ctr}, \mathbf{P})$. If C is either S_0 or S_1, Sim outputs $\mathsf{addr}_{\{0,1\}}$ provided from $\mathcal{F}_{\text{PRP}}([\![s_{\{0,1\}}]\!], [\![\perp +\mathsf{ctr}]\!])$ as *access patterns*. Otherwise, Sim outputs nothing.
- **Simulating $\Pi_{\textbf{Extract}}$.** When the adversary requests to run Π_{EXTRACT}, Sim halts and outputs π as an *access pattern* if $C = P$, otherwise it outputs nothing.

[10] In practice, the cost of other functionalities, besides OPRF, is added to this.

Since we are in $\mathcal{F}_{\text{ABB}}$-hybrid model and Algorithm 3, 4 and 5 are all deterministic except for the addresses recovered to a party, we should consider only the addresses as the access patterns. Now, consider the following hybrid argument.
$\underline{\mathsf{Hyb}_0}$: The real execution of Π_{Build}, Π_{Lookup} and Π_{Extract} in the ($\mathcal{F}_{\text{Share}}, \mathcal{F}^{\mathcal{P}}_{\text{Reveal}}$, $\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\text{Perm}}, \mathcal{F}_{\text{Unperm}}$)-hybrid model.
$\underline{\mathsf{Hyb}_1}$: The same as Hyb_0 except that, in the execution of Π_{Build} (Algorithm 3), the $\mathcal{F}_{\text{IfElse}}$ functionality of line 4 always set $\widetilde{k}_i$ to $\perp +i$.
$\underline{\mathsf{Hyb}_2}$: The same as Hyb_1, but parties always search for a dummy in Π_{Lookup}, i.e., the $\mathcal{F}_{\text{IfElse}}$ functionality of Algorithm 4, line 2, always set $\widetilde{k}$ to $\perp + \mathsf{ctr}$.
$\underline{\mathsf{Hyb}_3}$: The ideal simulation of Sim.

Let $\mathsf{View}_C(\mathsf{Hyb}_u)$ be the view of the adversary that colludes with $C \in \{P, S_0, S_1\}$ in Hyb_u. By the security of the PRF, $\mathsf{View}_P(\mathsf{Hyb}_0)$ and $\mathsf{View}_P(\mathsf{Hyb}_1)$ are computationally indistinguishable as long as $n < \mathsf{poly}(\lambda)$. $\mathsf{View}_{S_i}(\mathsf{Hyb}_0)$ and $\mathsf{View}_{S_i}(\mathsf{Hyb}_1)$ are obviously identical for any $i \in \{0, 1\}$.

It is also obvious that $\mathsf{View}_P(\mathsf{Hyb}_1) \equiv \mathsf{View}_P(\mathsf{Hyb}_2)$. In addition, $\mathsf{View}_{S_i}(\mathsf{Hyb}_1)$ and $\mathsf{View}_{S_i}(\mathsf{Hyb}_2)$ are computationally indistinguishable for any $i \in \{0, 1\}$ by security of the PRF. Furthermore, $\mathsf{View}_C(\mathsf{Hyb}_3)$ is the same as $\mathsf{View}_C(\mathsf{Hyb}_2)$ for any $C \in \{P, S_0, S_1\}$, directly.

B.2 Proof of Security of Our Passively Secure DORAM (Theorem 5.1)

Consider the following simulator Sim that simulates the view of a passive adversary that corrupts one of three party in Π_{Access}.

- Let $\mathsf{Sim}_{\mathsf{HT}}(n)$ be a simulator that simulates the view of a passive adversary in distributed oblivious hashing with input length n. As setup, Sim initializes $\mathsf{Sim}_{\mathsf{HT}}(c), \ldots, \mathsf{Sim}_{\mathsf{HT}}(c2^{L-1})$ where $c = 2\lceil \log N \rceil$ and $L = \lceil \log N - \log\log N \rceil$. In addition, Sim allocates an empty array of size c and initialize a query counter $\mathsf{ctr} = 0$
- Receiving a request to simulate Π_{Access}, Sim simulates the ctr-th execution of Π_{Access} with dummy inputs and an dummy data structure. All invocations of HT_i.Lookup is replaced with the lookup simulation of $\mathsf{Sim}_{\mathsf{HT}}(c2^{i-1})$, and also all HT_i.Extract and HT_i.Build in the subroutine $\Pi_{\text{Reshuffle}}$ are replaced with the extract and build simulation of that, respectively. Sim finally increments ctr and returns all outputs of $\mathsf{Sim}_{\mathsf{HT}}$.

Now, consider the following hybrid argument.

$\underline{\mathsf{Hyb}_0}$: The real execution of Π_{Access}.

$\underline{\mathsf{Hyb}_1}$: The same as Hyb_0 except performing $\Pi_{\text{Access}}([\![\mathsf{read}]\!], [\![\perp]\!], [\![\perp]\!])$ for any input.

$\underline{\mathsf{Hyb}_2}$: The ideal simulation of Sim.

Since we are in the ($\mathcal{F}_{\text{ABB}}, \mathcal{F}_{\mathsf{HT}}$)-hybrid model and the parties do not reveal any secret in the algorithm Π_{Access} and its subroutine $\Pi_{\text{Reshuffle}}$, the adversary's view of Hyb_1 is identical to that of Hyb_0 with overwhelming probability. Furthermore, in the $(\mathcal{F}_{\mathrm{ABB}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model, Hyb_1 and Hyb_2 are identical. The view of the adversary in the real $\Pi_{\textsc{Access}}$ and Sim are therefore indistinguishable.

B.3 Proof of Security of Our actively Secure Distributed Oblivious Hashing (Lemma 6.1)

Security and correctness without tampering follow the proof of Lemma 4.3. Thus, it suffices here to show that an active adversary succeed in tampering with shares with negligible probability.

In the execution of $\Pi^{\text{active}}_{\textsc{Build}}$, since there are $O(n')$ checksums in $\mathcal{S}$ including $2n$ addresses $[\![\mathsf{addr}_{i,j}]\!]^{\mathsf{m}}$, $2(n'-n)$ dummies $[\![\perp]\!]^{\mathsf{m}}$, and $O(n)$ others generated during the computation of $[\![\mathsf{addr}_{i,j}]\!]^{\mathsf{m}}$, in the first Proof phase (Algorithm 9, Line 3), the adversary's success probability in the first Proof phase is proportional to $n'b\lambda^{-\omega(1)}$ where b is the original size of a block. In addition, in the second Proof phase (Line 14), there are $O(\tau)$ checksums in $\mathcal{S}$ since there are τ calls of $\mathcal{F}_{\mathrm{EQ}}$ and $\mathcal{F}_{\textsc{IfElse}}$, and τ executions of $\Pi_{\textsc{VerPerm}}$. Hence, assuming $n', b < \mathsf{poly}(\lambda)$, the adversary's success probability is $\mathsf{negl}(\lambda)$.

In each execution of $\Pi^{\text{active}}_{\textsc{Lookup}}$, since there are $O(1)$ checksums in $\mathcal{S}$, adversary's tampering is detected with overwhelming probability $1 - \lambda^{\omega(1)}$.

Though parties do not verify the checksums in $\Pi^{\text{active}}_{\textsc{Extract}}$, all blocks in $[\![\widetilde{\mathsf{D}}]\!]^{\mathsf{m}}$ is stored to $\mathcal{S}$ during the unpermutation protocol. Hence, when the extracted data is used to build a new hash table, or parties terminate the protocol (that involves the final Proof phase), all data accessed in $\Pi^{\text{active}}_{\textsc{Extract}}$ is verified not to be tampered. Since the number of the checksums recorded in $\Pi^{\text{active}}_{\textsc{Extract}}$ is $O(n')$, the adversary's success probability is also negligible.