MacORAMa: Optimal Oblivious RAM with Integrity*

Surya Mathialagan
MIT
smathi@mit.edu

Neekon Vafa
MIT
nvafa@mit.edu

February 25, 2023

Abstract

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky (J. ACM '96), is a primitive that allows a client to perform RAM computations on an external database without revealing any information through the access pattern. For a database of size N, well-known lower bounds show that a multiplicative overhead of $\Omega(\log N)$ in the number of RAM queries is necessary assuming $O(1)$ client storage. A long sequence of works culminated in the asymptotically optimal construction of Asharov, Komargodski, Lin, and Shi (CRYPTO '21) with $O(\log N)$ worst-case overhead and $O(1)$ client storage. However, this optimal ORAM is known to be secure only in the *honest-but-curious* setting, where an adversary is allowed to observe the access patterns but not modify the contents of the database. In the *malicious* setting, where an adversary is additionally allowed to tamper with the database, this construction and many others in fact become insecure.

In this work, we construct the first maliciously secure ORAM with worst-case $O(\log N)$ overhead and $O(1)$ client storage assuming one-way functions, which are also necessary. By the $\Omega(\log N)$ lower bound, our construction is asymptotically optimal. To attain this overhead, we develop techniques to intricately interleave online and offline memory checking for malicious security. Furthermore, we complement our positive result by showing the impossibility of a *generic* overhead-preserving compiler from honest-but-curious to malicious security, barring a breakthrough in memory checking.

*The first author was supported in part by the Siebel Scholars program. The second author is supported in part by NSF fellowship DGE-2141064. This research was supported in part by DARPA under Agreement No. HR00112020023, an NSF grant CNS-2154149, a grant from the MIT-IBM Watson AI, a grant from Analog Devices, a Microsoft Trustworthy AI grant, and a Thornton Family Faculty Research Innovation Fellowship from MIT. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the United States Government or DARPA.

Contents

1 Introduction **4**
- 1.1 Our Contributions . . . 6
- 1.2 Related Work . . . 8
- 1.3 Organization . . . 9

2 Technical Overview **9**
- 2.1 The Hierarchical ORAM Paradigm . . . 9
- 2.2 Insufficiency of Standard Techniques . . . 10
- 2.3 Our Techniques . . . 13
- 2.4 Word Size . . . 15

3 Preliminaries **16**
- 3.1 Random Access Memory . . . 16
- 3.2 Cryptographic Primitives . . . 17
- 3.3 Maliciously Secure Oblivious Implementations . . . 19

4 Memory Checking **24**
- 4.1 Definitions . . . 24
- 4.2 Online Memory Checking and Time-Stamping . . . 26
- 4.3 Offline Memory Checking . . . 27

5 Separated Memory Checkers **30**

6 Write-Deterministic Implementations **36**

7 Overview of Maliciously Secure Building Blocks **40**

8 Maliciously Secure Oblivious Hash Table **41**

9 Maliciously Secure Optimal ORAM Construction **51**
- 9.1 A Warm-Up Construction . . . 52
- 9.2 Final Construction . . . 55

A Additional Preliminaries **62**
- A.1 Hybrid Model . . . 62

B Deferred proofs from Section 4 **66**

C Maliciously Secure Building Blocks **68**
- C.1 Maliciously Secure Oblivious RAM with $O\left(\log^4 N\right)$ Overhead . . . 68
- C.2 Oblivious Sorting . . . 69

C.3 Oblivious Two-Key Dictionary 69
C.4 Oblivious Random Permutation 70
C.5 Oblivious Bin Placement 74
C.6 Oblivious Balls-into-Bins Sampling 75
C.7 Tight Compaction, Intersperse and Perfect Random Permutation 76
C.8 Oblivious Cuckoo Hashing 77
C.9 Deduplication 80

D Final ORAM Construction **83**
D.1 Oblivious Leveled Dictionary 83
D.2 Algorithm Description 84
D.3 Reducing Client Space 88

1 Introduction

Suppose a user would like to outsource their database to a server. While the user could encrypt the data to hide the contents within the database, it is possible that even the access pattern corresponding to the queries to the database could result in information leakage about the underlying data [IKK12, CGPR15]. To solve this issue, Goldreich and Ostrovsky [GO96] propose the notion of *Oblivious RAM* (ORAM), a primitive that supports the same database queries but transforms the access pattern to remove any information leakage. As a general technique to ensure privacy of RAM computations, ORAM has many applications including cloud computing, multi-party protocols, secure processor design, and private contact discovery, the latter as implemented by the private messaging service Signal [SCSL11, FDD12, LO13, WHC+14, BNP+15, FRK+15, GHJR15, LWN+15, ZWR+16, DFD+21, Con22].

To hide the access pattern, the ORAM client has to pay a communication cost. This cost is generally quantified in terms of *overhead*: the number of (physical) queries in the oblivious simulation per underlying (logical) database query. Note that ORAM is only interesting when we require both the overhead and the local storage of the client to be small. Otherwise, the client could simply conduct a linear scan of the whole database for every query or download the whole database and make queries locally.

Larsen and Nielsen [LN18] show that for a database of size N, $\Omega(\log N)$ overhead is necessary for $O(1)$ client space. After a long line of research [GO96, OS97, DMN11, GM11, SCSL11, KLO12, CGLS17, CCS17, CNS18, PPRY18], the celebrated construction of OptORAMa by Asharov, Komargodski, Lin, Nayak, Peserico, and Shi [AKL+20] matches this lower bound with amortized $O(\log N)$ overhead and $O(1)$ client space. Asharov, Komargodski, Lin, and Shi [AKLS21] de-amortize this construction to fully match the lower bound with worst-case $O(\log N)$ overhead.

Tampering adversaries. Many prior ORAM constructions, including OptORAMa [AKL+20] and its de-amortized counterpart [AKLS21], consider security against passive adversaries that can only observe access patterns. In reality, an adversary can potentially do a lot more. In many applications, the adversary can play an active role in learning information from access patterns by tampering with the memory [RFY+13, FRK+15, MPC+18]. A tampering adversary can clearly violate correctness by arbitrarily modifying the contents in the database. Critically, however, a tampering adversary can also breach *obliviousness*. In fact, we show in Section 2 that many ORAM constructions [CGLS17, CNS18, PPRY18, AKL+20, AKLS21] are generally *not* oblivious when interacting with a tampering adversary.

As such, we view obliviousness against non-tampering adversaries as the *honest-but-curious* definition of obliviousness, whereas we consider security against tampering adversaries to be the natural *maliciously secure* notion of obliviousness. For a maliciously secure ORAM, we would like the following properties to hold (see Section 3.3 for a formal definition):

- The user's view should be indistinguishable from interacting with an honest RAM database (up until an abort).
- A malicious adversary should not be able to learn anything from tampering with the database except for the number of user RAM queries, even via the timing of a client abort.

Existing constructions. The notion of maliciously secure ORAM (also called tamper-proof ORAM) was first proposed by Goldreich and Ostrovsky [GO96]. They argue that their ORAM constructions can be made maliciously secure with $O(1)$ blowup in overhead because it is possible to *time-stamp* all physical writes, i.e., to compute the number of times a given physical (not logical) address has been written to. Authenticating time-stamped messages prevents replay attacks, forcing the adversary to always respond honestly. However, their construction has overhead $O(\log^3 N)$.

Other works [SDS+18, RFY+13] use the tree-based paradigm to construct maliciously secure ORAMs. These works use collision-resistant hash functions (CRHFs) to impose a Merkle tree [Mer90] on the memory, with the root node stored locally by the client. Crucially, this does not add any asymptotic overhead since a Merkle tree can be naturally super-imposed on the underlying tree. However, since these constructions follow the tree-based paradigm, they do not achieve the optimal $O(\log N)$ overhead for general word sizes.

Unfortunately, in Section 2, we argue that these techniques do not work for the existing optimal ORAM construction [AKLS21]. Indeed, there is no known $O(\log N)$ overhead ORAM that is maliciously secure. There are also no known lower bounds showing separations between the overhead needed for honest-but-curious ORAMs and maliciously secure ORAM. This motivates the following question:

Question 1.1. *Does there exist a maliciously secure ORAM construction with $O(\log N)$ overhead and $O(1)$ client storage?*

Generic compilers for malicious security. One can ask more generally if there is a generic way to compile *any* honest-but-curious ORAM into one that is maliciously secure. In this setting, a *compiler* Π is a layer between any honest-but-curious ORAM $\mathcal{C}$ and the server, in the following sense: the compiler Π takes queries from $\mathcal{C}$ and interacts with the server to generate a response for $\mathcal{C}$, and we want the composition of Π and $\mathcal{C}$ to be a maliciously secure ORAM.

One solution to this would be *memory checking*, a notion first defined by Blum, Evans, Gemmell, Kannan, and Naor [BEG+91]. At a high level, a memory checker can be seen as a layer between the user and an unreliable memory which verifies the correctness of the memory's answers to the user's requests, using small, private, and reliable space. In fact, one can view both time-stamping and Merkle tree verification as special cases of memory checking.

Memory checking is clearly such a compiler, as the client can use a memory checker to make sure the adversary always answers honestly. If we had an $O(1)$-overhead memory checker, then we could compile the optimal honest-but-curious ORAM constructions into ones that are maliciously secure with $O(\log N)$ overhead. However, best known memory checker constructions have a bandwidth of $O(\log N)$ [BEG+91, NN98, GTS01, HJ06, DNRV09, PT11].[1] Therefore, applying such a memory checker to an optimal ORAM construction with $\Theta(\log N)$ overhead would result in a maliciously secure ORAM with $O(\log^2 N)$ overhead. Moreover, Dwork, Naor, Rothblum, and Vaikuntanathan [DNRV09] show that deterministic and non-adaptive memory checkers – capturing

[1] *Bandwidth* refers to the ratio of the number of physical bits accessed to the number of bits being requested. While [PT11] achieves $O(\log N/\log\log N)$ overhead, its bandwidth is still $O(\log N)$ because the logical and physical word sizes differ by more than a constant factor. In our setting, the logical and physical word sizes will always differ by at most a constant factor, so overhead and bandwidth are asymptotically equivalent.

the known memory checking constructions – must have overhead $\Omega(\log N/\log\log N)$. Therefore, barring a major development in memory checking, this approach will not work.

But do compilers have to be memory checkers? Does a weaker compiler suffice? This leads us to the following question:

Question 1.2. *Does an $O(1)$-overhead compiler from honest-but-curious to maliciously secure ORAM give an $O(1)$-overhead memory checker?*

If so, this would present a memory checking barrier to generically compile any optimal honest-but-curious ORAM into an optimal maliciously secure ORAM.

1.1 Our Contributions

In this work, we resolve both questions. To answer Question 1.1, we construct MacORAMa, a maliciously secure oblivious RAM with worst-case $O(\log N)$ overhead.

Theorem 1.3 (Informal version of Theorem 9.5)**.** *Assuming the existence of one-way functions, for databases of size N and word size $\omega(\log N)$, there is an ORAM construction with $O(\log N)$ worst-case overhead, $O(N)$ server storage, and $O(1)$ client storage that is secure against $\mathsf{poly}(N)$-time adversaries that can tamper with the database. Moreover, if the client (but not the adversary) is given access to a random oracle, our construction is unconditionally statistically secure, even against computationally unbounded adversaries.*

We compare MacORAMa to previous ORAM constructions and lower bounds in Table 1.

For word size $\omega(\log N)$, this not only matches the lower bound known for honest-but-curious ORAM [LN18, KL21], but also shows that malicious security is possible with *no additional asymptotic overhead*. Interestingly, even though maliciously secure ORAM is a stronger notion than memory checking, our construction matches the best known memory checker constructions (e.g., [BEG+91, NN98, GTS01, HJ06, DNRV09, PT11]) in terms of bandwidth. In other words, maliciously secure ORAM is a stronger primitive than both honest-but-curious ORAM and memory checking, and our construction achieves the best known bandwidth for both simultaneously.

Also, we observe that the existence of one-way functions is necessary for maliciously secure ORAM. One can easily show that the need for hiding the data itself (which is typically done via secret-key encryption) implies the existence of one-way functions. Furthermore, Naor and Rothblum [NR09] show that any memory checker, and hence maliciously secure ORAM, with local space s and overhead q (in bits) such that $s \cdot q = o(N)$ implies the existence of (almost) one-way functions. Therefore, our construction uses provably minimal assumptions.

To resolve Question 1.2, we show that any generic compiler from an honest-but-curious ORAM to a maliciously secure one needs to essentially be a memory checker.

[2]While the access-pattern is unconditionally statistically secure, one ultimately needs the existence of one-way functions to hide (i.e., encrypt) the underlying data.

Construction	Client Storage	Server Storage	Overhead	Assumption	Maliciously Secure
[GO96]	$\Theta(1)$	$\Theta(N \log N)$	$\Theta(\log^3 N)^*$	OWF	Yes
Path ORAM [SDS+18]	$\omega(\log N)$	$\Theta(N)$	$\Omega(\log^{2-\epsilon} N)$	None[2]	No
Path ORAM [SDS+18]	$\omega(\log N)$	$\Theta(N)$	$\Omega(\log^{2-\epsilon} N)$	CRHF	Yes
PanORAMa [PPRY18]	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N \log\log N)^*$	OWF	No
OptORAMa [AKL+20]	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N)^*$	OWF	No
[AKLS21]	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N)$	OWF	No
Lower Bound [GO96, LN18, KL21]	$\Theta(1)$	$\Theta(N)$	$\Omega(\log N)$	–	–
MacORAMa (Our work)	$\Theta(1)$	$\Theta(N)$	$\Theta(\log N)$	OWF	Yes

Table 1: This table outlines some of the known ORAM constructions secure against $\mathsf{poly}(N)$*-time adversaries (i.e., for* $\lambda = N$*) for any word size* $w = \omega(\log N)$ *and* $w \leq (\log N)^{1+\epsilon}$*, for* $0 < \epsilon < 1$*. We justify this choice of word size in the malicious setting in Section 2.4. Client and server storage are measured in terms of the number of words. The assumption column refers to what assumptions the constructions use to prove security. "OWF" stands for the existence of one-way functions, and "CRHF" stands for the existence of collision-resistant hash functions. Overheads with an asterisk (*) superscript denote amortized overheads.*

Theorem 1.4 (Informal version of Theorem 5.3)**.** *If there exists a generic honest-but-curious to maliciously secure ORAM compiler with* $O(1)$ *blowup, there exists a memory checker*[3] *with* $O(1)$ *overhead.*

Therefore, the existence of $O(1)$-overhead memory checkers exactly characterizes the existence of an $O(1)$-blowup compiler from honest-but-curious to maliciously secure ORAM. However, since the best memory checkers have bandwidth $O(\log N)$, this is a barrier to an overhead-preserving generic compiler. As a result, to construct our maliciously secure ORAM with $O(\log N)$ overhead, we use the existing optimal construction in a white-box way.

Our techniques. To prove Theorem 1.3, we use more general notions of memory checking. Blum et al. [BEG+91] define an *online* memory checker as one which immediately verifies the correctness of every answer it gives the user and an *offline* memory checker as one that only needs to report that some error has occurred after a batch of requests. While all known online memory checkers incur an additional blowup of $\Omega(\log N)$, there exist offline memory checkers that achieve amortized $O(1)$ bandwidth [BEG+91, DNRV09]. Even though offline memory checking alone is insufficient to guarantee malicious security, we show how to intricately combine online and offline memory checking techniques to get $O(1)$ blowup and malicious security simultaneously. We describe these techniques in Section 2.3.

[3]Technically, we consider a slightly different notion of memory checking that we show is both necessary and sufficient for compiling honest-but-curious to maliciously secure ORAM. See Section 5 for more details.

We believe that our techniques in combining online and offline memory checking are general enough to extend to future hierarchical ORAM constructions. Furthermore, as memory checking has other applications in cryptography and beyond [BEG+91, CSG+05, ABC+07, JK07, OR07, SW13], our techniques may be of independent interest.

1.2 Related Work

Server-side computation. The lower bounds of Goldreich and Ostrovsky [GO96], Larsen and Nielsen [LN18], and Komargodski and Lin [KL21] assume that the server is a "passive storage," i.e., supports only reads and writes. Crucially, they assume that the honest server does not do any additional work. While we only consider the passive storage model, we point out that prior work has in fact leveraged server-side computation to construct maliciously secure ORAMs with smaller client communication overhead [AKST14, DvF+16, AFN+17, HGY20]. For example, Devadas et al. [DvF+16] construct Onion ORAM, which is a maliciously secure ORAM with $O(1)$ communication overhead for word size $\tilde{\omega}(\log^6 N)$ (where $\tilde{\omega}$ hides $\log \log N$ factors) by relying on server-side computation. Hoang et al. [HGY20] construct a *multi-server* maliciously secure ORAM with server-side computation achieving $O(1)$ communication overhead for word size $\omega(\log N)$. These constructions generally rely on poly-logarithmic server work.

However, ORAM constructions with server-side computation are unsuitable for the multi-party computation (MPC) setting. For example, ORAMs can be used to construct MPC protocols [GKK+12, LO13, Tof14, KS14, WHC+14, LWN+15], but any server-side work has to be computed using an MPC protocol with poly-logarithmic circuit size and is thus expensive.

Multi-client maliciously secure ORAM. Several works have studied the problem of a multi-client ORAM [FWC+12, SS13, LO13, MMRS15, BMN17, MMRS17, CFLM20]. In this setting, data owners can delegate access to their database to third party clients, while preserving the privacy of the clients. In particular, the access patterns must remain hidden not only from the server, but also from other clients.

This problem has been studied in both the honest-but-curious and malicious setting. Blass et al. [BMN17] consider the problem in the setting where only the server is malicious. Maffei et al. [MMRS15, MMRS17] further consider the model where both the server and the clients are malicious. In the setting of the latter work, one not only has to protect against tampering servers, but also against tampering clients to ensure such clients do not compromise the privacy of other clients.

However, most of these constructions at the core either rely on tree-based ORAM schemes interleaved with Merkle trees or apply a Merkle tree on top of a hierarchical scheme. Therefore, these do not achieve the desired efficiency for general word sizes.

Weaker notions of malicious security. The work of Fletcher et al. [FRK+15] relies on the tree-based paradigm and recursively uses PRFs and authentication to cleverly compress the position map and to time-stamp in an alternate way. While the construction is extremely efficient in practice, they achieve a weaker notion of malicious security. Their construction promises obliviousness against honest-but-curious adversaries but guarantees only correctness against tampering

adversaries. In particular, the timing of the client's abort in the presence of a tampering adversary can leak information about the user's access patterns.

1.3 Organization

In Section 2, we give a technical overview of our result. In Section 3, we define the cryptographic primitives that we use and our definitions of malicious security. In Section 4, we define memory checking, and we give constructions for both online and offline memory checking. In Section 5, we define a weaker notion of memory checking that we call separated memory checkers, and we show that any generic compiler must satisfy this notion of memory checking. In Section 6, we define a notion of *write-deterministic* algorithms, and we argue that algorithms that are write-deterministic can be made maliciously secure with $O(1)$ blowup in overhead. In Section 7, we show how the building blocks for the final ORAM construction can be made maliciously secure. In Section 8, we give a maliciously secure hash table construction that matches the efficiency of previous honest-but-curious constructions. Finally, in Section 9, we give our ORAM construction and its proof of security. We defer many technical definitions, theorems, and proofs to the appendix.

2 Technical Overview

2.1 The Hierarchical ORAM Paradigm

The hierarchical ORAM framework was first introduced by Goldreich and Ostrovsky [GO96]. For a database D of size N, the corresponding hierarchical ORAM contains $L = \log_2 N$ *oblivious hash tables* $H_1, \ldots, H_L$, where H_i contains 2^i data items. Upon receiving a read or write to some address $\mathsf{addr} \in D$, the ORAM proceeds as follows:

- **Look-up phase:** Perform a hash table lookup for addr in each level $H_1, \ldots, H_L$ sequentially, until the key addr is found. If addr is found in level H_i, look up dummies in the subsequent layers $H_{i+1}, \ldots, H_L$. If the operation is a read, copy the data found in H_i to H_1 and return it. If it is a write, write the new value to H_1 that was provided as part of the access.
- **Rebuild phase:** Find the lowest level (i.e., smallest index) ℓ which is empty. If all levels are non-empty, set $\ell := L$. Merge all the elements in the first $\ell - 1$ layers $\bigcup_{1 \le j \le \ell - 1} H_i$ to layer ℓ, while only keeping the copy of any addr in the lowest possible level. Construct a new oblivious hash table at level ℓ with these contents. Now $H_1, \ldots, H_{\ell-1}$ are empty, and H_ℓ is non-empty.

For each access to D, we perform one hash table lookup in each H_i and one write to H_1. For these lookups to be efficient, we can use an *oblivious Cuckoo hashing* scheme [PR04, GM11, CGLS17] to achieve essentially $O(1)$ lookup time. Moreover, in the rebuild phase, the ith level is rebuilt every 2^i accesses. If the rebuild time for level i takes time $T(2^i)$, then the total number of queries for t accesses to D is essentially given by $O(t \log(N)) + \sum_{i=1}^{\log N} \lceil \frac{t}{2^i} \rceil \cdot T(2^i)$. In the original work of Goldreich and Ostrovsky [GO96], they show how to obtain $T(n) = O(n \log^2 n)$, which gives an amortized query complexity of $O(\log^3 N)$. Patel, Persiano, Raykova, and Yeo [PPRY18] improve this to $T(n) = O(n \log \log n)$, giving an amortized query complexity of $O(\log N \log \log N)$. Finally,

the work of Asharov et al. [AKL+20] shows how to reduce to $T(n) = O(n)$, giving an overall number of queries of

$$\sum_{i=1}^{\log N} \left\lceil \frac{t}{2^i} \right\rceil \cdot O(2^i) = O(t \log N),$$

resulting in an amortized query complexity of $O(\log N)$. This complexity was then de-amortized in the work of Asharov et al. [AKLS21] to obtain $O(\log N)$ worst-case query complexity.

The main invariant maintained in the hierarchical construction is that any key corresponding to a real address will be looked up only once in the life span of an oblivious hash table H_i. To see this, note that once we find a key addr in level i, we search for dummies in subsequent layers. We then write back addr to H_1, and addr will always exist in a lower level than H_i until H_i is rebuilt. Therefore, the hash tables only need to guarantee obliviousness when this invariant is met.

2.2 Insufficiency of Standard Techniques

2.2.1 Authentication

For static databases (i.e., ones that do not support writes), one could use *message authentication codes* (MACs). MACs add tags to data (e.g., to ciphertexts) to make them difficult to modify or forge. However, when the database supports both reads and writes, MACs may not be sufficient because MACs do not protect against *replay attacks*, attacks where the server responds with old authenticated content. In particular, since ORAMs have to support dynamic updates, authentication alone is not sufficient. We show an explicit attack on hierarchical ORAMs including OptORAMa [AKL+20, AKLS21] and PanORAMa [PPRY18] even when MACs are added.

Replay attack. First, suppose the user requests a read to address addr, and addr is found at hash table H_i. Consider a malicious database that effectively does not write the value of addr back to H_1. Now, suppose the user requests a write to address addr', and suppose that no rebuilding has happened between the two accesses (e.g., this happens when the number of total accesses so far is odd). If $\mathsf{addr}' = \mathsf{addr}$, since addr was not written back to H_1, the user looks up addr in all layers up to H_i, just as in the previous access. Therefore, the access pattern to tables $H_1, \cdots, H_i$ in both cases will look the same. On the other hand, if $\mathsf{addr}' \neq \mathsf{addr}$, then the access pattern is very likely to be different. Hence, the adversary is able to distinguish between access patterns $A_1 = \{(\mathsf{read}, \mathsf{addr}), (\mathsf{write}, \mathsf{addr})\}$, and $A_2 = \{(\mathsf{read}, \mathsf{addr}), (\mathsf{write}, \mathsf{addr}')\}$, which breaks the obliviousness of the ORAM.

More generally, replay attacks can break the invariant that any key is looked up exactly once in the life span of an oblivious hash table. As a succinct summary, *obliviousness* of hash table lookups depends on the *correctness* of previous hash table lookups.

2.2.2 Time-stamping

Goldreich and Ostrovsky [GO96] argue that their constructions are maliciously secure because it is possible to *time-stamp* all physical writes. Roughly speaking, time-stamping means that at any

point in time, the client is able to locally compute the number of times any physical (not logical) address has been overwritten. If the client uses authenticated encryption to encrypt the data along with its time-stamp, any replay attack can be easily detected with essentially no blowup in overhead or local storage. In particular, since the hash tables in [GO96] are constructed using several AKS oblivious sorts [AKS83], it is in fact time-stampable. Therefore, their construction gives a maliciously secure ORAM construction which achieves $O(\log^3 N)$ overhead. However, optimal ORAM schemes [AKL+20, AKLS21] do not seem to be time-stampable in the same way because the access patterns of the writes are not deterministic.

The Marking Problem. Consider the following arbitrary sequence of RAM operations, inspired by Lemma 3.3 of Dwork et al. [DNRV09].

- Initialize a database of size n and write 0 to all of the indices.
- Choose arbitrary $S \subseteq [n]$ of size $n/2$ and write 1 to all indices in S.

Claim 2.1. *The Marking Problem as defined above cannot be time-stamped with $o(n)$ bits of memory.*

Proof. A time-stamping function $T(i, t)$ outputs the number of times a given index i has been written to up to time t. Note that $T(i, 3n/2) = 2$ if and only $i \in S$. Therefore, being able to compute $T(\cdot, \cdot)$ is sufficient to recover the set S. Since there are $2^{(1-o(1))n}$ possibilities for S, is clear that computing $T(\cdot, \cdot)$ must require $(1 - o(1)) \cdot n$ bits of space. □

Balls-in-Bins Hashing. Consider the following balls-in-bins hashing algorithm for a list of values.

- Iterate over the list of n "balls" labeled by keys $k_1, k_2, \ldots, k_n$, while assigning a random value $r_i \leftarrow [B]$ to k_i.[4]
- Initialize empty buckets $L_1, \ldots L_B$.
- Iterate over $i \in [n]$, and write k_i into the next available slot in bucket L_{r_i}.

Claim 2.2. *The Balls-in-Bins Hashing Problem as defined above cannot be time-stamped in fewer than n bits of memory for $B > 1$.*

Proof. By a similar argument as in the marking problem, one can argue that a time-stamping function can be used to recover all of the random r_i values, which has entropy $n \log_2(B) \geq n$ (for $B \geq 2$). Thus, for any $B > 1$, this sequence of operations cannot be time-stamped with fewer than n bits of memory. □

While these might seem like contrived examples, these exact sequences of RAM operations shows up in OptORAMa and its de-amortized version [AKL+20, AKLS21] within the hash table implementation.

[4] Even if one generates r_i values via something like $\mathsf{PRF}(k_i)$, a similar lower bound applies since the ordering of k_i is random.

In particular, whenever an address is visited in the hash table H_i at level i of the hierarchical ORAM, Asharov et al. [AKLS21] mark the visited elements and then remove these elements during the rebuild phase. Crucially, these hash table constructions [AKL+20, AKLS21] work off of Patel et al.'s [PPRY18] idea of reusing the residual randomness of *unvisited* elements to attain linear rebuild time. Therefore, it is important that all marked elements are memory checked and removed to achieve this guarantee. Since the sequence of logical addresses read could be arbitrary, by Claim 2.1, it is not possible to time-stamp this sequence in low space.

Looking ahead, we modify the oblivious hash table constructions of OptORAMa and its de-amortized version [AKL+20, AKLS21] to mark in a different way that is still query and space efficient but can now be made safe against malicious servers.

The constructions of Asharov et al. [AKL+20, AKLS21] also use balls-in-bins hashing in the construction of the hash table. Intuitively, if a malicious adversary deletes elements during the balls-in-bins hashing, some elements will be lost during the hash table construction. An adversary can exploit this to detect repeated access patterns in a similar manner as discussed in our earlier replay attack.

2.2.3 Merkle Trees

A common technique for checking consistency of data is Merkle trees [Mer90], which are used in applications such as succinct argument systems for NP [Kil92], trusted hardware [CD16], and blockchains. At a high level, a Merkle tree stores the database at the leaves of a binary tree, and each internal node contains hashes (e.g., using a collision resistant hash function) of its children nodes. One only has to locally store the root of the binary tree to be able to check consistency. To check a particular address of the database, one can query all of the nodes on the path to the root and check consistency by also querying all the children of the nodes on the path. Since the height of the tree is $\log_2 N$, one would have to make $O(\log N)$ queries to check each location in memory.

While this would incur a $O(\log N)$ multiplicative overhead when applied to existing ORAM constructions in a black-box manner, Stefanov et al. [SDS+18] noticed that one can capitalize on the tree-based structure of path ORAM to superimpose a Merkle tree with $O(1)$ blowup. However, since the hierarchical ORAM constructions do not have a similar tree-based structure, it is not clear that a similar technique can be applied.

2.2.4 Memory Checking

More generally, one could use *memory checking* techniques [BEG+91] (defined in Section 4) to construct maliciously secure ORAMs. However, best known memory checkers are in fact tree-based, and they all have $O(\log N)$ bandwidth [BEG+91, DNRV09, PT11]. Therefore, generically applying existing memory checkers to optimal ORAM constructions will not give the desired overhead.

2.2.5 Generic Compilers

As posed in Question 1.2, one might ask if a more efficient generic compiler from honest-but-curious to maliciously secure ORAM exists. In Section 5, we show that such a compiler must also essentially

be a memory checker. The idea is as follows. Intuitively, one could embed an adversarially chosen access pattern into an honest-but-curious ORAM and adversarially tamper with the database to force an incorrect response from the compiler. At this point, the (adversarially constructed) honest-but-curious ORAM can misbehave arbitrarily. We formalize this idea in Theorem 5.8.

More specifically, in Section 5, we define a variant of memory checking that we show is *equivalent* to a generic compiler. While the standard definition of memory checking allows user queries to depend adaptively on the memory checker's server access pattern, our definition of memory checking requires security only when the user queries are independent of the memory checker's server access pattern. For some quick intuition as to why we need this definition, this variant of memory checking allows us to separate out and embed adversarial user queries within an honest-but-curious ORAM. Conversely, to show that this variant notion of memory checking is also *sufficient* for ORAM (Corollary 5.9), we crucially rely on obliviousness.

2.3 Our Techniques

As our starting point, we use the worst-case $O(\log N)$ overhead honest-but-curious ORAM construction of Asharov et al. [AKLS21]. Our main technique is carefully combining various forms of memory checking. Blum et al. [BEG+91] make the distinction between *online* and *offline* memory checkers. An *online memory checker* immediately verifies the correctness of every answer from the database. An *offline memory checker* is a batched version of online memory checking that only detects if *some* error has occurred after a long sequence of operations. We define these notions of memory checking formally in Section 4.

Online memory checking. Online memory checking is sufficient to guarantee malicious security, as any malicious response from the adversary can be immediately detected (as we prove in Theorem 4.5). Time-stamping is an online memory checking strategy that has $O(1)$ overhead. While most of the construction is time-stampable, as argued earlier, some parts of the algorithm have access patterns that are provably not time-stampable. Since the best known general online memory checkers have bandwidth $O(\log N)$, online memory checking alone does not seem sufficient to construct a maliciously secure ORAM with $O(\log N)$ overhead.

2.3.1 Offline Memory Checking

In offline memory checking, we require detection of an incorrect response only at the end of a given computation. It is possible with essentially only $O(1)$ amortized overhead [BEG+91, DNRV09]. However, offline memory checking does not necessarily prevent leakage. For example, the replay attack shows that if the outputs of the hash table lookups are tampered with, privacy leakage will occur before offline memory checking catches this error. To combat this issue, we characterize situations where offline memory checking is sufficient.

Access-deterministic algorithms. Many sub-protocols of the construction have deterministic access patterns, i.e., access patterns that have no dependence on the input or any randomness. For example, in the AKS sorting network [AKS83], the comparisons made by the algorithm are

entirely determined by the edges of an expander graph, which is independent of the input array. In fact, many algorithms in OptORAMa, including tight compacting a bit array and interspersing two randomly shuffled arrays [AKL+20], have this flavor.

Since the access patterns of such algorithms are independent of the input in the honest-but-curious setting, it seems as though directly offline checking these algorithms is sufficient to make them maliciously secure. However, this is not always the case; the security of offline memory checking crucially depends on the implementation.

For a concrete example, consider the oblivious sorting network of Ajtai et al. [AKS83]. This algorithm, along with several algorithms in OptORAMa [AKL+20] (e.g., loose compaction), cleverly uses edges of bipartite expander graphs to make comparisons. A few steps of an implementation could go as follows:

1. Use space on the server to compute and store a bipartite expander graph $G = (V, E)$.
2. Iterate over the edge set E, and make comparisons according to E.

In this implementation, if a tampering adversary were to replace the edge E with a set $E' = \{e'_1, e'_2, \ldots, e'_m\}$ of secret values, the comparisons will leak the contents of $e'_1, e'_2, \ldots, e'_m$. If one were to offline-check these steps, the contents of e'_i would be leaked, which would break malicious security. However, if the algorithm were instead to *locally* compute the sequence E in a low space way (on the fly, for example), an offline-check would be sufficient, but in general, computing such an E in low space may be impossible or at least complex to describe. Therefore, security of offline checking these access-deterministic algorithms is quite sensitive to implementation details.

To bypass these subtleties, we show (in Theorem 6.2) a generic way to convert access-deterministic algorithms into maliciously secure ones without worrying about implementation details.

We do this by observing that if we had access to some time-stamp *oracle*, a similar argument as the one by Goldreich and Ostrovsky [GO96] can be used to check the reads of the algorithm in an online way. We instantiate this time-stamp oracle by first running and offline memory-checking the honest-but-curious algorithm on some fixed dummy input (independent of the real input) to essentially build a time-stamp array $\boldsymbol{T}$ on the server. Then, we use $\boldsymbol{T}$ as a time-stamp oracle to verify all the reads in the real execution of the algorithm. This only blows up the run-time of the algorithm by a multiplicative $O(1)$ factor. We prove this formally in Theorem 6.2. With this theorem, in Section 7, we compile many honest-but-curious building blocks from OptORAMa in an implementation-independent way into ones that are maliciously secure with $O(1)$ blowup.

One can view this as a strengthening of Goldreich and Ostrovsky's [GO96] notion of a time-stampable simulation, since an algorithm being access-deterministic, or more accurately, *write-deterministic* which we define in Section 6, is essentially equivalent to the existence of a (not necessarily efficiently computable) time-stamp function. However, Goldreich and Ostrovsky's algorithm [GO96] needs the time-stamp function to be efficiently computable in low space while ours just needs such a function to exist.

Access patterns with random leakage. Some subprotocols in Asharov et al. [AKLS21] are in fact not access-deterministic. However, many such protocols only leak random values within intermediate stages. One such example is balls-in-bins hashing. Since the only values leaked in the

process are the random values associated to each ball, this can be simulated and has no dependence on the input. Therefore, offline memory checking suffices in these cases as well. We use this technique crucially in our hash table construction in Section 8 as well as in several of our building blocks in Section 7. For a self-contained example, see Algorithm C.10 that obliviously implements a random permutation.

Conditionally offline-safe access patterns. Some subprotocols S in Asharov et al. [AKLS21] are neither time-stampable, access-deterministic, nor have only random leakage. However, the following is often true:

- These subprotocols S *are* offline-safe if we assume some other portion P of the memory is tamper-proof.
- These portions of memory P are time-stampable, so we can use authentication to effectively assume that these portions cannot be tampered with.

As a concrete example, consider the *routing* problem, where there are there n balls $x_1, \cdots, x_n$, and we need to route x_i to some publicly known bins L_{r_i} for $r_i \in [B]$. (That is, r_i does not need to be hidden to the adversary.) If the portion of memory P that holds all of the r_i is tamper-proof, then the access pattern, namely mapping ball x_i to bin L_{r_i}, is completely fixed. However, if the adversary somehow modifies P to convert r_i to some r_i' that would reveal secret information, then the adversary could learn r_i' by observing the access to bin $L_{r_i'}$. If P is time-stampable, then this attack is preventable as one could ensure that any retrieved r_i value is correct, making the routing access pattern completely fixed. Therefore, offline memory checking the memory of the addresses being routed to is sufficient.

Therefore, if we online memory check the time-stampable portions of memory, namely P, and simultaneously offline memory check the conditionally offline-safe portions, namely S, we can guarantee security against tampering adversaries.

Post-verifiable offline checking. Another issue is that as is, an adversary can tamper with the database after an offline check was conducted, making any reads to previously offline checked memory unreliable. To combat this, we modify the offline memory checking protocol of Blum et al. [BEG+91] in Algorithm 4.10 to leave the memory in a *post-verifiable* state (using similar ideas as [AEK+17]). In particular, after an offline check of a batch of queries, the algorithm leaves the memory in a state that can be *online-checked* with query complexity $O(1)$ during future reads. This ensures that the contents of the memory cannot be tampered with after the offline check.

2.4 Word Size

Our results work for arbitrary word size w as long as $w = \omega(\log \lambda) = \omega(\log N)$ (for $N \leq \mathsf{poly}(\lambda)$, as it is in our setting) and $w \leq \mathsf{poly}(\lambda)$. Most prior work in the honest-but-curious regime focus on the setting where $w = \Theta(\log N)$. However, all existing techniques going from honest-but-curious to maliciously secure ORAM that preserve overhead require increasing the word size from $\Theta(\log N)$ to $\omega(\log N)$. For example, time-stamping requires separate MACs for each address, which each

need to have length $\omega(\log \lambda)$. Also, collision-resistant hash functions (CRHFs), as used in Path ORAM [SDS+18, RFY+13], need to have output length $\omega(\log \lambda)$ to avoid birthday attacks.

In fact, we now give some barriers for going below word size $\omega(\log \lambda)$:

- The existing constructions of online memory checkers with $O(\log N)$ bandwidth need word size $\omega(\log \lambda)$ to guarantee $1 - \mathsf{negl}(\lambda)$ soundness [BEG+91, NN98, GTS01, HJ06, DNRV09, PT11]. Since maliciously secure ORAM is a stronger primitive than online memory checking, this poses a barrier to constructing maliciously secure ORAM with $O(\log N)$ overhead and word size.
- Any online memory checker with $O(1)$ overhead and sublinear local space (i.e., in the computational soundness regime [NR09]) *must* have word size $\omega(\log \lambda)$, as otherwise, the adversary could simply return random values and break soundness with probability $1/\mathsf{poly}(\lambda)$ infinitely often. This captures the cases of time-stamping and CRHFs above.
- In the hierarchical paradigm, the replay attack shows that the correctness of each of the $\log N$ hash table lookups is necessary to guarantee obliviousness of the subsequent lookups. As a result, in the hierarchical paradigm, it seems that $O(1)$-overhead memory checking is necessary for each hash table lookup, and thus, making $\omega(\log \lambda)$ bits of communication necessary for each of the $\log N$ hash tables due to the previous argument.

3 Preliminaries

For a natural number $N \in \mathbb{N}$, we let $[N]$ denote the set $\{1, \ldots, N\}$. Throughout, we let λ denote the security parameter. We say a function $f : \mathbb{N} \to \mathbb{N}$ is *negligible* if for all constants $c \geq 0$, $\lim_{n \to \infty} f(n) \cdot n^c = 0$. We denote a negligible function in input λ by $\mathsf{negl}(\lambda)$. We say an algorithm is PPT if it runs in (non-uniform) probabilistic polynomial time.

For a (finite) set S, we use the notation $x \leftarrow S$ as shorthand for saying x is a random variable sampled uniformly from S. We abuse notation and write $x \leftarrow B()$ for saying that x is a sample from the output of a randomized algorithm B. For a stateful variable x, we often abuse notation and write $x \leftarrow y$ to denote the operation of updating the variable x to have value y. For $x, y \in \{0,1\}^n$, we use the notation $x \oplus y$ to denote bit-wise XOR of x and y, and for $x, y \in \{0,1\}^*$ we use the notation $x||y$ and (x, y) to denote concatenation of x and y. We use the notation $\varnothing$ to denote an empty list or placeholder value, with the exact meaning being clear from context.

3.1 Random Access Memory

We work in the standard word RAM model throughout. Unless specified otherwise, the underlying RAM functionality we would like to make oblivious has N logical addresses and each memory cell indexed by $\mathsf{addr} \in [N]$ contains a word of size w, i.e., an element of $\{0,1\}^w$. (See Functionality 3.8 for more details.) For adversaries running in $\mathsf{poly}(\lambda)$ time to be able to store an N-word dictionary, we assume $w, N \leq \mathsf{poly}(\lambda)$.

To obliviously implement this RAM functionality (and other functionalities along the way), it will be convenient to work in a RAM model with a slightly larger word size $\overline{w}$ such that $\overline{w} = O(w)$.

For clarity, we call w the *logical* or *plain-text* word size, and $\overline{w}$ the *physical* word size. Explicitly, we will set $\overline{w} = 10w$ such that $\overline{w} = \omega(\log \lambda)$ and $\overline{w} \leq \mathsf{poly}(\lambda)$, but we urge the reader to think of $\overline{w}$ as being an arbitrarily small function that is $\omega(\log \lambda)$. Since $\log(N) \leq O(\log \lambda) < \omega(\log \lambda) = \overline{w}$, physical words can contain a memory address, and we need $\overline{w} = \omega(\log \lambda)$ for security reasons so that ciphertexts and authentication tags can fit in a word and remain secure against $\mathsf{poly}(\lambda)$-time adversaries. (In particular, $2^{-\overline{w}} = \mathsf{negl}(\lambda)$.) See Section 2.4 for more details on this choice of word size.

As is standard in previous works, we assume that word-level addition and Boolean operations, and additionally pseudorandom function (PRF) evaluations, can be done in unit cost.

3.2 Cryptographic Primitives

In this section, we define the cryptographic primitives we need, such as pseudorandom function families (PRFs), symmetric-key encryption, and message authentication codes (MACs).

Definition 3.1 (Pseudorandom functions (PRFs))**.** *Let* PRF *be an efficiently computable function family indexed by keys* $\mathsf{sk} \in \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}$*, where each* $\mathsf{PRF}_{\mathsf{sk}}$ *takes as input a word* $x \in \{0,1\}^{\overline{w}}$ *and outputs a value* $y \in \{0,1\}^{\overline{w}}$*. A function family* PRF *is secure if for every (non-uniform) PPT* $\mathcal{A}$*, it holds that*

$$\left| \Pr_{\mathsf{sk} \leftarrow \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}} \left[\mathcal{A}^{\mathsf{PRF}_{\mathsf{sk}}(\cdot)}(1^\lambda) = 1 \right] - \Pr_{f \leftarrow F_{\overline{w}}} \left[\mathcal{A}^{f(\cdot)}(1^\lambda) = 1 \right] \right| \leq \mathsf{negl}(\lambda)$$

for all large enough $\lambda \in \mathbb{N}$*, where* $F_{\overline{w}}$ *is the set of all functions that map* $\{0,1\}^{\overline{w}}$ *into* $\{0,1\}^{\overline{w}}$*.*

Often, we only need a PRF with smaller domain or codomain than $\{0,1\}^{\overline{w}}$. We abuse notation and use the same PRF, where we implicitly either pad the input with 0s or ignore a suffix of the PRF output.

Since $w < \overline{w} \leq \mathsf{poly}(\lambda)$, one can easily modify the [GGM86] construction of PRFs to have domain and codomain $\{0,1\}^{\overline{w}}$. Moreover, these PRFs exist as long as there is a length-doubling cryptographic pseudorandom generator (PRG) with seed length $\ell_{\mathsf{skPRF}}(\lambda)$ that is secure against $\mathsf{poly}(\lambda)$-time adversaries, which is implied by a related quantitative statement regarding the existence of one-way functions [HILL99]. Therefore, throughout this paper, we assume there is a PRF with key-length $\ell_{\mathsf{skPRF}}(\lambda)$.

While $\ell_{\mathsf{skPRF}}(\lambda)$ captures the strength of the PRF, we emphasize that, as standard in the ORAM literature, we do not count storage of the PRF key as part of client's space. As such, the existence of a polynomially-secure one-way function is sufficient for our results. (In some of our theorem statements, for clarity, we remind the reader of the need for a PRF key in parentheses.)

We now define symmetric-key encryption, where plain-text messages have size w. Throughout, we have $w = \frac{1}{10} \cdot \overline{w} = \omega(\log \lambda)$.

Definition 3.2 (Encryption)**.** *A* symmetric-key *or* private-key encryption *scheme* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is a triple of PPT algorithms with the following syntax:*

- $\mathsf{Gen}(1^\lambda)$*: Outputs a key* $k \in \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}$*.*

- $\mathsf{Enc}_k(m)$*: Generates a ciphertext* $\mathsf{ct} \in \{0,1\}^{\overline{w}/5}$ *using the key* k *corresponding to the input message* $m \in \{0,1\}^w$*.*
- $\mathsf{Dec}_k(\mathsf{ct})$*: Deterministic decryption algorithm which outputs a message* m *corresponding to the decryption of* ct *with key* k*.*

Moreover, the following properties should hold:

- **Correctness:** *For all* $m \in \{0,1\}^w$*, it holds that* $\Pr[\mathsf{Dec}_k(\mathsf{ct}) = m] = 1$*, where the probability is taken over* $k \leftarrow \mathsf{Gen}(1^\lambda)$ *and* $\mathsf{ct} \leftarrow \mathsf{Enc}_k(m)$*.*
- **IND-CPA Security:** *For all stateful PPT adversaries* $\mathcal{A}$*, the following holds. For* $k \leftarrow \mathsf{Gen}(1^\lambda)$*,* $\mathcal{A}^{\mathsf{Enc}_k(\cdot)}(1^\lambda) \to (m_0, m_1)$*, where* $|m_0| = |m_1| = w$*,*

$$\left|\Pr\left[\mathcal{A}^{\mathsf{Enc}_k(\cdot)}\left(1^\lambda, \mathsf{Enc}_k(m_0)\right) = 1\right] - \Pr\left[\mathcal{A}^{\mathsf{Enc}_k(\cdot)}\left(1^\lambda, \mathsf{Enc}_k(m_1)\right) = 1\right]\right| \leq \mathsf{negl}(\lambda).$$

We use the standard construction of symmetric-key encryption from PRFs (e.g., from [Gol09]), as follows. $\mathsf{Gen}(1^\lambda)$ outputs uniformly random $k \leftarrow \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}$, $\mathsf{Enc}_k(m)$ outputs $\mathsf{ct} = (r, \mathsf{PRF}_k(r) \oplus m)$ for $r \leftarrow \{0,1\}^{\overline{w}/5-w}$, and for $\mathsf{ct} = (r, z)$, $\mathsf{Dec}_k(\mathsf{ct}) = z \oplus \mathsf{PRF}_k(r)$. Correctness immediately follows, and IND-CPA security follows from PRF security and because the probability of reuse of an r over $\mathsf{poly}(\lambda)$ ciphertexts is $\mathsf{poly}(\lambda) \cdot 2^{-|r|} = \mathsf{poly}(\lambda) \cdot 2^{-\Omega(\overline{w})} = \mathsf{negl}(\lambda)$.

In our security proofs, we will often use a variant of IND-CPA security as defined above that we call *adaptive* IND-CPA security. The game is informally defined as follows, indexed by $b \in \{0,1\}$:

- The challenger samples some key $k \leftarrow \mathsf{Gen}(1^\lambda)$.
- For $\mathsf{poly}(\lambda)$ iterations, the adversary $\mathcal{A}'$ adaptively sends messages m_0 and m_1, and the challenger sends back $\mathsf{Enc}_k(m_b)$.

A standard hybrid argument can be used to show that an adversary that has a non-negligible distinguishing advantage in this adaptive IND-CPA game between $b = 0$ and $b = 1$ will also break the standard IND-CPA security notion of $(\mathsf{Enc}, \mathsf{Dec}, \mathsf{Gen})$.

Definition 3.3 (MACs)**.** *A* message authentication code (MAC) *scheme* $(\mathsf{MACGen}, \mathsf{MAC}, \mathsf{MACVer})$ *is a triple of PPT algorithms with the following syntax:*

- $\mathsf{MACGen}(1^\lambda)$*: Outputs a key* $k \in \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}$*.*
- $\mathsf{MAC}_k(m)$*: Outputs an authentication* $\mathsf{tag} \in \{0,1\}^{\overline{w}/5}$ *on the key* k *corresponding to the input message* $m \in \{0,1\}^{\overline{w}}$*.*
- $\mathsf{MACVer}_k(m, \mathsf{tag})$*: Outputs 1 or* $\perp$*.*

Moreover, the following properties should hold:

- **Correctness:** *For all* $m \in \{0,1\}^{\overline{w}}$*, it holds that* $\Pr[\mathsf{MACVer}_k(m, \mathsf{tag}) = 1] = 1$*, where the probability is taken over* $k \leftarrow \mathsf{MACGen}(1^\lambda)$ *and* $\mathsf{tag} \leftarrow \mathsf{MAC}_k(m)$*.*

- **Unforgeability:** *For all PPT adversaries $\mathcal{A}$ with oracle access to $\mathsf{MAC}_k(\cdot)$ and $\mathsf{MACVer}_k(\cdot,\cdot)$, let $S_{\mathcal{A}}$ be the random variable corresponding to the set of m_i queried by $\mathcal{A}$ to the $\mathsf{MAC}_k(\cdot)$ oracle. Then, for $k \leftarrow \mathsf{MACGen}(1^\lambda)$ and $\mathcal{A}^{\mathsf{MAC}_k(\cdot),\mathsf{MACVer}_k(\cdot,\cdot)}(1^\lambda) \to (m^*, \mathsf{tag}^*)$ where $|m^*| = \overline{w}$, we require that*

$$\Pr\left[\mathsf{MACVer}_k(m^*, \mathsf{tag}^*) = 1 \wedge m^* \notin S_{\mathcal{A}}\right] = \mathsf{negl}(\lambda).$$

We use the standard construction of MACs from PRFs (e.g., from [Gol09]), as follows. $\mathsf{MACGen}(1^\lambda)$ outputs uniformly random $k \leftarrow \{0,1\}^{\ell_{\mathsf{skPRF}}(\lambda)}$, $\mathsf{MAC}_k(m)$ outputs $\mathsf{tag} = \mathsf{PRF}_k(m)$, and lastly, $\mathsf{MACVer}_k(m, \mathsf{tag}) = 1$ if and only if $\mathsf{PRF}_k(m) = \mathsf{tag}$. Correctness immediately follows, and unforgeability follows from PRF security because both the MAC_k and MACVer_k oracles can be instantiated with a PRF oracle, and the probability of correctly guessing a new output from a random function is $2^{-\overline{w}/5} = 2^{-\omega(\log\lambda)} = \mathsf{negl}(\lambda)$.

Because ciphertexts and MACs each have output length $\overline{w}/5$, authenticated encryption of the form $(\mathsf{ct}, \mathsf{MAC}_{k'}(\mathsf{ct}))$ where $\mathsf{ct} \leftarrow \mathsf{Enc}_k(\mathsf{data})$ for $\mathsf{data} \in \{0,1\}^w$ easily fits in a physical word of bit length $\overline{w}$.

3.3 Maliciously Secure Oblivious Implementations

The general goal we consider is constructing a RAM client $\mathcal{C}$ that *obliviously* implements a given functionality $\mathcal{F}$, *even* when interacting with a possibly malicious RAM server $\mathcal{A}$. That is, a space-constrained $\mathcal{C}$ gives RAM instructions to $\mathcal{A}$, who may behave arbitrarily and give tampered database responses along the way. Throughout, the client $\mathcal{C}$ has small, private, incorruptible local space. The client's goal is to use the large public database to compute $\mathcal{F}(x)$ given input x while making sure x is not leaked in any way during the process, even by a tampering adversary.

When $\mathcal{A}$ behaves dishonestly, we require that the client either aborts or outputs the correct answer anyway. That is, our security definition simultaneously handles both correctness of the client's output and obliviousness of the client's input x.

3.3.1 Stateless Functionalities

For (possibly randomized) stateless functionalities $\mathcal{F}$, our notion of maliciously secure obliviousness can be described as follows. We have a client $\mathcal{C}$ making RAM queries $\mathsf{query} = (\mathsf{op}, \mathsf{addr}, \mathsf{data}) \in \{\mathsf{read}, \mathsf{write}\} \times [N] \times (\{0,1\}^{\overline{w}} \cup \{\perp\})$ to a possibly malicious RAM memory held by $\mathcal{A}$. The client may output $\mathsf{flag} = \mathsf{true}$ to indicate detection of malicious activity, but if not, it generates some out that should be distributed according to $\mathcal{F}(x)$.

Of course, one possibility for the adversary $\mathcal{A}$ is to be *honest-but-curious*, where it always returns data^* correctly according to a RAM. That is, an honest-but-curious $\mathcal{A}$ keeps track of N-word memory $\mathsf{mem} \in \left(\{0,1\}^{\overline{w}}\right)^N$, and for each $\mathsf{query} = (\mathsf{write}, \mathsf{addr}, \mathsf{data})$, $\mathcal{A}$ updates $\mathsf{mem}[\mathsf{addr}] \leftarrow \mathsf{data}$ (and returns $\mathsf{data}^* \leftarrow \perp$), and for each $\mathsf{query} = (\mathsf{read}, \mathsf{addr}, \mathsf{data})$, $\mathcal{A}$ returns $\mathsf{data}^* \leftarrow \mathsf{mem}[\mathsf{addr}]$ (and ignores data). Restricting $\mathcal{A}$ to be honest-but-curious is the setting considered in many previous works.

A bit more formally, our malicious security notion is that the joint distribution of the access pattern and the output of $\mathcal{C}$, when interacting adaptively with $\mathcal{A}$, is indistinguishable from the output of a

simulator and $\mathcal{F}(x)$, where the simulator crucially does not have access to x. We formalize this in the real-ideal paradigm via the following definition.

Experiment 3.4 $\textsc{Real}(\mathcal{C}, \mathcal{A})$.

$x \leftarrow \mathcal{A}\left(1^\lambda\right)$
$\boxed{\mathsf{out} \leftarrow \bot}$
$\mathsf{data}^* \leftarrow \bot$
while $\boxed{\mathsf{out} = \bot}$ **do**
  $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{out}) \leftarrow \mathcal{C}\left(1^\lambda, x, \mathsf{data}^*\right)}$
  if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
  $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathsf{out}}\right)$

Experiment 3.5 $\textsc{Ideal}(\mathcal{F}, \mathcal{S}, \mathcal{A})$.

$x \leftarrow \mathcal{A}\left(1^\lambda\right)$
$\boxed{\mathsf{done} \leftarrow \mathsf{false}}$
$\mathsf{data}^* \leftarrow \bot$
while $\boxed{\mathsf{done} = \mathsf{false}}$ **do**
  $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{done}) \leftarrow \mathcal{S}\left(1^\lambda, \mathsf{data}^*\right)}$
  if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
  $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathcal{F}(x)}\right)$

Definition 3.6. *For a (stateless) RAM functionality $\mathcal{F}$, we say a stateful RAM machine $\mathcal{C}_\mathcal{F}$ is a* $(1-\delta)$-maliciously secure oblivious implementation *of $\mathcal{F}$ if the following two conditions hold:*

1. **Obliviousness & Correctness**: *There is a (stateful) PPT simulator $\mathcal{S}$ such that for all (stateful) PPT $\mathcal{A}$, the adversary $\mathcal{A}$ distinguishes between* $\textsc{Real}(x, \mathcal{C}_\mathcal{F}, \mathcal{A})$ *(Experiment 3.4) and* $\textsc{Ideal}\,(\mathcal{F}, \mathcal{S}, \mathcal{A})$ *(Experiment 3.5) with advantage at most δ. That is,*

$$|\Pr[\textsc{Real}(\mathcal{C}, \mathcal{A}) = 1] - \Pr[\textsc{Ideal}(x, \mathcal{F}, \mathcal{S}, \mathcal{A}) = 1]| \leq \delta.$$

2. **Completeness**: *For all (stateful) honest-but-curious servers $\mathcal{A}$, with probability $1-\delta$, the client $\mathcal{C}_\mathcal{F}$ never aborts, i.e., never sets* flag *to* true *throughout the whole execution of the real experiment.*

When δ is not specified, we take it to mean an arbitrary negligible function in λ.

Another interpretation of this security definition is that we require the existence of a universal compiler (namely, the simulator $\mathcal{S}$) from real-world adversaries into ideal-world adversaries, where ideal-world adversaries only have the ability to decide whether to abort.

Sometimes, instead of implementing a (stateless) functionality, we require a weaker property that the output of a client satisfies some property that is not necessarily unique. An example of this is TightCompaction (see Appendix C.7) that takes an input array of reals and dummies and satisfies the property that the real elements are all moved to the front in some order. The exact nature of the outputs of such an algorithm might be difficult to describe, so for these settings, we instead define the functionality $\mathcal{F}$ as the algorithm itself:

Definition 3.7. *We say that a (stateless) RAM machine $\mathcal{C}$ is a* $(1-\delta)$-maliciously secure oblivious algorithm *if $\mathcal{C}$ is a $(1-\delta)$-maliciously secure oblivious implementation of $\mathcal{C}$, in the sense of Definition 3.6. In this setting, the definition of $\mathcal{C}$ as a functionality is defined as the output of $\mathcal{C}$ when interacting with an honest server.*

Informally, if the output of $\mathcal{C}$ when interacting with an honest-but-curious adversary has some desirable property (e.g., as in $\mathsf{TightCompaction}$), then this definition says that $\mathcal{C}$ is also a maliciously secure implementation of a "functionality" with this property too.

3.3.2 Reactive Functionalities

Loosely speaking, a reactive functionality $\mathcal{F}$ is an interactive functionality that holds some internal state, and whenever it takes in a command cmd and input x, it gives some (possibly randomized) output $\mathcal{F}(\mathsf{cmd}, x)$, where the notational dependence on the internal hidden state is suppressed. One way to think of a reactive functionality is as a specification for a data structure problem (i.e., the desired behavior of a data structure), where the various types of queries are specified by cmd and the input to those queries are denoted by x. We note that stateless functionalities are special cases of reactive functionalities, so any theorem statements regarding reactive functionalities immediately apply to stateless ones as well. Throughout, we assume that all functionalities considered are computable in probabilistic polynomial time, i.e., have some (non-oblivious) implementation where the output to any given query is computable in probabilistic polynomial time in λ.

$\mathcal{F}_{\mathsf{RAM}}^{N,w}$ is an important example of a reactive functionality, and we describe it in Functionality 3.8.

Functionality 3.8 $\mathcal{F}_{\mathsf{RAM}}^{N,w}$: The RAM Functionality.

Syntax:

- $\mathsf{cmd} \in \{\mathsf{read}, \mathsf{write}\}$;
- $x = (\mathsf{addr}, \mathsf{data}) \in [N] \times \{0,1\}^w$, where addr is an index into an N-entry RAM database, and data contains a word (to be used only if $\mathsf{cmd} = \mathsf{write}$).

Internal State: A memory array mem with N entries, each containing values in $\{0,1\}^w$, all initialized to 0^w.

Command $\mathcal{F}_{\mathsf{RAM}}^{N,w}(\mathsf{cmd}, x)$:

- If $\mathsf{cmd} = \mathsf{read}$, return $\mathsf{mem}[\mathsf{addr}]$, the word located in mem at address $\mathsf{addr} \in [N]$.
- If $\mathsf{cmd} = \mathsf{write}$, update mem by $\mathsf{mem}[\mathsf{addr}] \leftarrow \mathsf{data}$, and nothing is returned (i.e., only the internal hidden state is modified).

That is, $\mathcal{F}_{\mathsf{RAM}}^{N,w}$ precisely describes the data structure problem of random access memory with N addresses each holding a word of size w. In our final ORAM construction, we implement a logical RAM with plain-text word size w by using words of size $\overline{w} = O(w)$ when interacting with the physical storage.

Other examples of reactive functionalities that we will use include hash tables (see Functionality 8.1) and dictionaries (see Appendix C.3).

Just like for stateless functionalities, we formalize security for reactive functionalities by a simulation-based definition and require indistinguishability between a real-world and ideal-world experiment, as in Experiments 3.9 and 3.10. At a high level, for a fixed functionality $\mathcal{F}$, in the experiment, the (stateful) adversary $\mathcal{A}$ gets to adaptively choose (cmd, x) for the (stateful) client $\mathcal{C}$. In Experiments 3.9 and 3.10, the outer "while" loop corresponds to the adversary issuing mul-

tiple commands to the client, and the inner "while" loop corresponds to the client and adversary performing the RAM computation of the given command.

- In the real world, the client $\mathcal{C}$, based on its knowledge of cmd and x, sends RAM queries $\mathsf{query} = (\mathsf{op}, \mathsf{addr}, \mathsf{data}) \in \{\mathsf{read}, \mathsf{write}\} \times [N] \times \{0,1\}^{\overline{w}}$ to the possibly malicious adversary $\mathcal{A}$. The adversary $\mathcal{A}$ can adaptively respond to each query with a possibly incorrect $\mathsf{data}^* \in \{0,1\}^{\overline{w}}$. At any point, $\mathcal{C}$ can abort (by setting $\mathsf{flag} = \mathsf{true}$, at which point the whole experiment ends), or it can claim it is finished and output out. This continues until either (a) $\mathcal{A}$ stops issuing commands or (b) $\mathcal{C}$ aborts at some point.
- In the ideal world, we require the existence of a simulator $\mathcal{S}$ that can simulate all the query and abort behavior of $\mathcal{C}$ without knowing x. We do not require the simulator to compute out, as this is in general impossible without knowing x. We instead (implicitly) set $\mathsf{out} = \mathcal{F}(\mathsf{cmd}, x)$ and give this output to the adversary $\mathcal{A}$ in issuing its next command.

Our security definition is informally that no efficient $\mathcal{A}$ can distinguish between the two worlds. Since $\mathcal{A}$ both sees the access pattern of $\mathcal{C}$ and the output of the computation, this indistinguishability asserts both obliviousness and correctness. We present the formal definition in Definition 3.11.

Experiment 3.9 $\textsc{Real}(\mathcal{C}, \mathcal{A})$.

$(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda\right)$
while $\mathsf{cmd} \neq \perp$ **do**
  $\boxed{\mathsf{out} \leftarrow \perp}$
  $\mathsf{data}^* \leftarrow \perp$
  while $\boxed{\mathsf{out} = \perp}$ **do**
    $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{out}) \leftarrow \mathcal{C}\left(1^\lambda, \mathsf{cmd}, x, \mathsf{data}^*\right)}$
    if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
    $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
  end while
  $(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathsf{out}}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda\right)$

Experiment 3.10 $\textsc{Ideal}(\mathcal{F}, \mathcal{S}, \mathcal{A})$.

$(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda\right)$
while $\mathsf{cmd} \neq \perp$ **do**
  $\boxed{\mathsf{done} \leftarrow \mathsf{false}}$
  $\mathsf{data}^* \leftarrow \perp$
  while $\boxed{\mathsf{done} = \mathsf{false}}$ **do**
    $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{done}) \leftarrow \mathcal{S}\left(1^\lambda, \mathsf{cmd}, \mathsf{data}^*\right)}$
    if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
    $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
  end while
  $(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathcal{F}(\mathsf{cmd}, x)}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda\right)$

Definition 3.11. *For a reactive functionality $\mathcal{F}$, we say a (stateful) RAM machine $\mathcal{C}_\mathcal{F}$ is a* $(1-\delta)$-maliciously secure oblivious implementation *of a reactive functionality $\mathcal{F}$ if the following two conditions hold:*

1. **Obliviousness & Correctness**: *There is a (stateful) PPT simulator $\mathcal{S}$ such that for all (stateful) PPT $\mathcal{A}$, the adversary $\mathcal{A}$ distinguishes between* $\textsc{Real}(\mathcal{C}_\mathcal{F}, \mathcal{A})$ *(Experiment 3.9) and* $\textsc{Ideal}\,(\mathcal{F}, \mathcal{S}, \mathcal{A})$ *(Experiment 3.10) with advantage at most δ.*
2. **Completeness**: *For all (stateful) honest-but-curious $\mathcal{A}$, with probability $1-\delta$, the client $\mathcal{C}_\mathcal{F}$ never aborts, i.e., never sets* flag *to* true *throughout the whole execution of the real experiment.*

When δ is not specified, it is taken to mean an arbitrary negligible function in λ.

For both stateless and reactive functionalities, we focus on two main complexity measures of implementations: *local space complexity* and *query complexity*. Local space complexity denotes the number of words an implementation $\mathcal{C}$ locally (and privately) stores. For stateless functionalities, the query complexity refers to the number of RAM queries made by $\mathcal{C}$ used to generate out. For reactive functionalities, the query complexity for a specific cmd refers to the number of RAM queries made by $\mathcal{C}$ needed to generate out for cmd. We also use the terms *time*, *run-time*, and query complexity interchangeably. Another complexity measure we consider for implementations is *server space complexity*, which refers to the amount of space needed on the server by the implementation.

A special case of this is oblivious RAM (ORAM), where the reactive functionality being implemented is the RAM functionality itself:

Definition 3.12 (Oblivious RAM)**.** *We say that $\mathcal{C}$ is a* $(1-\delta)$-maliciously secure oblivious RAM (ORAM) *(or honest-but-curious* $(1-\delta)$*-ORAM) for a database of size N with word size w if $\mathcal{C}$ is a* $(1-\delta)$*-maliciously secure oblivious (or honest-but-curious* $(1-\delta)$*-oblivious, respectively) implementation of $\mathcal{F}_{\mathsf{RAM}}^{N,w}$. When δ is unspecified, we take it to mean an arbitrary function negligible in λ.*

We define the *overhead* of an ORAM $\mathcal{C}$ to be the maximum of the query complexities of the read and write commands for $\mathcal{C}$. In fact, our ORAM construction will also hide whether there the logical query is a read or write, making read or write part of the input, not the command. In such a case, the notion of query complexity for an ORAM is uniquely defined. Because one can generically show that these definitions of ORAM are equivalent up to a multiplicative factor of two in the overhead, we use these definitions of ORAM interchangeably.

Additionally, we can assume without loss of generality that a universal simulator $\mathcal{S}$ for an ORAM $\mathcal{C}$ has the same space complexity as $\mathcal{C}$ (up to multiplicative $O(1)$ terms), as running $\mathcal{C}$ on some fixed dummy inputs is itself a simulator. (In fact, this argument further holds for any functionality $\mathcal{F}$ without input assumptions.)

Comparison to prior definitions. First, we observe that maliciously secure oblivious simulation (Definition 3.11) is a strengthening of the oblivious simulation definition given in [AKL+20, AKLS21], as now the adversary $\mathcal{A}$ need not respond to queries truthfully. In fact, oblivious simulation in [AKL+20, AKLS21] is essentially equivalent to our security definition where we now restrict to honest-but-curious adversaries $\mathcal{A}$. In our terminology, their notion of obliviousness can essentially be defined as follows:

Definition 3.13. *For a reactive functionality $\mathcal{F}$, we say a RAM machine $\mathcal{C}_{\mathcal{F}}$ is an* honest-but-curious $(1-\delta)$-oblivious implementation *of a reactive functionality $\mathcal{F}$ if Definition 3.11 holds with parameter* $(1-\delta)$*, except that Condition 1 is required to hold only against honest-but-curious PPT $\mathcal{A}$ (instead of arbitrary PPT $\mathcal{A}$).*

Definition 3.3 of [AKL+20] gives a similar definition of honest-but-curious obliviousness, with the main difference being that the adversary only observes the addresses being accessed, not any of

the data values. When combined with symmetric-key encryption in a straightforward way, their definition is equivalent to Definition 3.13, with an extra additive $\mathsf{negl}(\lambda)$ loss in obliviousness.

Second, we note that our definition of maliciously secure ORAM is also a strengthening of online memory checking [BEG+91] (as defined in Definition 4.1), as we require the client to satisfy the completeness condition and either abort or return the correct answer with high probability.

Input and output tapes. In our security definitions, when $(x, \mathsf{cmd}) \leftarrow \mathcal{A}(1^\lambda)$ is chosen adaptively by $\mathcal{A}$ and x is given to $\mathcal{C}$, it is often the case that x is long and cannot be stored locally by the client in small space. Similarly, out may also be long and unable to stored locally in small space.

To handle this, we assume that $\mathcal{C}$ is given two special tapes, specified as follows:

- T_{in} is a read-only RAM tape that can never be modified by $\mathcal{A}$ after x is initially written to it by $\mathcal{A}$. The adversary $\mathcal{A}$ can see the access pattern for this tape (i.e., the addresses read by $\mathcal{C}$) as they occur, and each read to T_{in} counts as a query as part of the query complexity of $\mathcal{C}$.
- $\mathsf{T}_{\mathsf{out}}$ is a write-once, write-only RAM tape for $\mathcal{C}$ to write out. The adversary $\mathcal{A}$ cannot modify the contents of this tape, but it can see the access pattern (i.e., the addresses written to by $\mathcal{C}$) as they occur, and each write to $\mathsf{T}_{\mathsf{out}}$ counts as a query as part of the query complexity of $\mathcal{C}$. (In the security game, $\mathsf{out} = \perp$ is just syntactic shorthand for the $\mathcal{C}$ saying it is not yet finished. It is possible that some of $\mathsf{T}_{\mathsf{out}}$ may be written to while $\mathsf{out} = \perp$ still holds.)

Crucially, as is standard in the setting of low-space computation, we do not consider the size of either tape as part of the local space complexity of $\mathcal{C}$.

Input assumptions. Just like in [AKL+20], we sometimes assume that inputs to commands of functionalities satisfy certain assumptions (e.g., having no duplicates, being sorted according to some key, or being randomly shuffled). This can be formalized as a (potentially randomized) map $\mathcal{X}$, where now $(\mathsf{cmd}, x) \leftarrow \mathcal{A}(1^\lambda)$ is mapped to $(\mathsf{cmd}, \mathcal{X}(\mathsf{cmd}, x))$ before being given to $\mathcal{C}$.

Hybrid model and composition. In Appendix A, we define a hybrid model and prove a concurrent composition theorem for our security notion following the universal composability (UC) framework of Canetti [Can20] since our simulations are straight-line and universal. This allows us to modularly compose maliciously secure oblivious implementations with each other safely.

4 Memory Checking

4.1 Definitions

We recall the notion of memory checking from Blum et al. [BEG+91]. A memory checker M can be defined as a probablistic RAM program that interacts with a user $\mathcal{C}$ and server $\mathcal{A}$, where $\mathcal{C}$ is performing a RAM computation with memory held by $\mathcal{A}$. Specifically, without a memory checker, $\mathcal{C}$ sends $\mathsf{query}_{\mathcal{C}} = (\mathsf{cmd}, \mathsf{addr}, \mathsf{data}) \in \{\mathsf{read}, \mathsf{write}\} \times [N] \times (\{0,1\}^{\overline{w}} \cup \{\perp\})$ to $\mathcal{A}$, who may or may

not correctly follow the RAM command, i.e., may send the wrong word back to $\mathcal{C}$ when $\mathsf{cmd} = \mathsf{read}$. With memory checker M, M now serves as an intermediary between $\mathcal{C}$ and $\mathcal{A}$ that takes in each $\mathsf{query}_{\mathcal{C}}$ from $\mathcal{C}$ and generates and sends (possibly multiple and adaptive) queries to $\mathcal{A}$. Whenever $\mathsf{cmd} = \mathsf{read}$, $\mathcal{C}$ once again generates and sends (possibly multiple and adaptive) queries to $\mathcal{A}$, and M is then required to either respond to $\mathcal{C}$ with some word or abort by sending $\perp$ to indicate a malicious $\mathcal{A}$. Once the memory checker aborts, the protocol is done. This continues in rounds until $\mathcal{C}$ is done sending queries, of which there are at most $\mathsf{poly}(\lambda)$.

Definition 4.1 (Online Memory Checker)**.** *We say that M is an* online memory checker *for $\mathcal{C}$ if the following two properties hold:*

1. **Completeness**: *If $\mathcal{A}$ is honest (i.e., behaves according to $\mathcal{F}_{\mathsf{RAM}}$), then M never aborts and the responses that M sends to $\mathcal{C}$ are all correct with probability $1 - \mathsf{negl}(\lambda)$.*
2. **Soundness**: *For all PPT $\mathcal{A}$, the probability that M ever sends some incorrect response to $\mathcal{C}$ is $\mathsf{negl}(\lambda)$. That is, for each request from $\mathcal{C}$, if $\mathcal{A}$ sends an incorrect response to M, M can either independently recover the correct answer and send it to $\mathcal{C}$, or it can abort by sending $\perp$ to $\mathcal{C}$.*

We call this memory checker "online" because the memory checker must be able to catch incorrect responses from M as soon they are sent. On the other hand, one can define the notion of an "offline" memory checker:

Definition 4.2 (Offline Memory Checker)**.** *We say that M is an* offline memory checker *for $\mathcal{C}$ if the following two properties hold:*

1. **Completeness**: *If $\mathcal{A}$ is honest (i.e., behaves according to $\mathcal{F}_{\mathsf{RAM}}$), then M never aborts, and the responses that M sends to $\mathcal{C}$ are all correct with probability $1 - \mathsf{negl}(\lambda)$.*
2. **Soundness**: *For all PPT $\mathcal{A}$, if M ever sends an incorrect response to $\mathcal{C}$, it must abort by the end of the last request made by $\mathcal{C}$ with probably at least $1 - \mathsf{negl}(\lambda)$.*[5]

That is, M may send many incorrect responses to $\mathcal{C}$, but if it does, by the end of the computation, M detects that there was an error at some point. We emphasize that M does not need to know where or when an error occurred.

Note that when we say M is an online or offline memory checker for an implementation $\mathcal{C}$, we mean this is true against adversaries that can both tamper with the server and adaptively feed commands and inputs into $\mathcal{C}$.

There are two main complexity measures we associate with (both online and offline) memory checkers: *(local) space complexity* and *query complexity* (or *overhead*). *Space complexity* is simply the amount of space (in words) used by the memory checker M, and worst-case (or amortized) *query complexity* is the worst-case (or amortized, respectively) number of requests made by M per request of $\mathcal{C}$ throughout the computation. We also sometimes explicitly consider the *server space complexity* of memory checkers, which refers to how many physical words are needed to be stored on the server. Unless specified otherwise, for a user $\mathcal{C}$ generating queries according to a RAM of size N, the server space complexity of memory checkers will be $O(N)$.

[5] More formally, for the last request, the user $\mathcal{C}$ must send some "last request" symbol along with its query to indicate to M that it is the final request. We omit this technicality for simplicity.

4.2 Online Memory Checking and Time-Stamping

Definition 4.3 (Definition 2.3.3.1 of [GO96])**.** *Let $\mathcal{C}$ be a RAM program on a memory of size N, and let* count *be a global counter of the number of RAM queries made by $\mathcal{C}$. We say that $\mathcal{C}$ is* time-stampable *if there exists a function $TS(\cdot,\cdot)$ locally computable in $O(1)$ space and word operations such that for $\mathsf{addr} \in [N]$, $TS(\mathsf{count}, \mathsf{addr}) \in \mathbb{N}$ is exactly the number of times that* addr *has been written to among the client's first* count *queries.*

In particular, when we say an implementation $\mathcal{C}$ is time-stampable, we mean there is a time-stamp function that holds against any adversary issuing commands and inputs given to $\mathcal{C}$.

We note that the lemma below is essentially already implicit in Theorem 3.2.1 of [GO96]. We give the proof for completeness and to show compatibility with our definitions.

Lemma 4.4. *If $\mathcal{C}$ is time-stampable on a memory of size N, then there exists an online memory checker for $\mathcal{C}$ with worst-case query complexity 1, local space complexity $O(1)$ (and one PRF key), and server space complexity $O(N)$.*

The proof of this lemma has been deferred to Appendix B.

We now show that online memory checkers are sufficient to compile honest-but-curious oblivious implementations to maliciously secure oblivious implementations. For generality and for our applications, we show this holds in the $\mathcal{G}$-hybrid model for an arbitrary functionality $\mathcal{G}$, but for intuition, one can consider $\mathcal{G}$ to be the empty functionality and ignore the hybrid model throughout. (For more details about the hybrid model, see Appendix A.)

Theorem 4.5. *Suppose that $\mathcal{C}$ is an honest-but-curious $(1-\delta)$-oblivious implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with client space complexity c and query complexity q, and suppose that $\mathcal{C}$ has an online memory checker M with local space complexity c_M and query complexity q_M. Then, the composition of M and $\mathcal{C}$ is a $(1-\delta-\mathsf{negl}(\lambda))$-maliciously secure oblivious implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with client space complexity $c + c_M$ and query complexity $q \cdot q_M$.*

The proof of this theorem has been deferred to Appendix B.

Now, combining Lemma 4.4 and Theorem 4.5, we immediately get the following:

Corollary 4.6. *Assuming there exist one-way functions, if $\mathcal{C}$ is a time-stampable honest-but-curious oblivious implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with client space complexity $s_{\mathcal{C}}$ and query complexity $q_{\mathcal{C}}$, then there exists a maliciously secure oblivious implementation $\mathcal{C}'$ of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with client space complexity $s_{\mathcal{C}} + O(1)$ words (and one PRF key) and query complexity $q_{\mathcal{C}}$.*

We also use the fact that online memory checkers exist with $O(\log N)$ overhead without any need for time-stampability (e.g., [BEG+91, DNRV09]).

Theorem 4.7 (Section 5.1.2 in [BEG+91])**.** *Assuming there exist one-way functions, for word size $\omega(\log \lambda)$, there is an online memory checker for RAM databases with N words with query complexity $O(\log N)$, local space complexity $O(1)$ (and one PRF key), and server space complexity $O(N)$.*

With this online memory checker and Theorem 4.5, we now get the following:

Corollary 4.8. *If $\mathcal{C}$ is an honest-but-curious $(1-\delta)$-oblivious implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with client space complexity $s_{\mathcal{C}}$ and query complexity $q_{\mathcal{C}}$ for a RAM server with N words, then there exists a $(1-\delta-\mathsf{negl}(\lambda))$-maliciously secure oblivious implementation $\mathcal{C}'$ of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with query complexity $O(q_{\mathcal{C}} \cdot \log(N))$, client space complexity $s_{\mathcal{C}} + O(1)$ words (and one PRF key), and server space complexity $O(N)$.*

4.3 Offline Memory Checking

Here, we present an offline memory checker that can be seen as an adaptation of the offline memory checker of Blum et al. [BEG+91]. While we assume the existence of one-way functions (which we independently need for our ORAM construction), we believe our construction is simpler because we use a simple counting argument and MACs instead of ϵ-biased hash functions. We also show how we can make our algorithm *post-verifiable* using MACs.

Post-verifiability. For our usage of offline memory checkers, it will be useful to be able to verify later reads in an online way after an offline check. That is, after an offline memory checker either aborts or confirms that the responses to $\mathcal{C}$ are correct, as long as no writes are made after that, it will be useful to have an $O(1)$-query complexity way to online check the reads. We give a formal definition below.

Definition 4.9 (Post-Verifiable Offline Memory Checker)**.** *Suppose that $\mathcal{C}$ can be split chronologically into two phases, with the first phase happening before the second:*

- *Read/Write Phase: An arbitrary sequence of read and write queries.*
- *Read-Only Phase: An arbitrary sequence of read queries.*

We say that M is a post-verifiable offline memory checker *for $\mathcal{C}$ if the following two properties hold:*

1. **Completeness**: *If $\mathcal{A}$ is honest (i.e., behaves according to $\mathcal{F}_{\mathsf{RAM}}$), then M never aborts, and the responses that M sends to $\mathcal{C}$ are all correct with probability $1-\mathsf{negl}(\lambda)$. (If it does so with probability 1 instead of $1-\mathsf{negl}(\lambda)$, we say M satisfies* perfect completeness.*)*
2. **Soundness**: *For all PPT $\mathcal{A}$, if M ever sends an incorrect response to $\mathcal{C}$ during the Read/Write Phase, it must abort by the end of the read/write phase with probably at least $1-\mathsf{negl}(\lambda)$.*[6] *Furthermore, the probability that M ever sends $\mathcal{C}$ an incorrect response during the Read-Only Phase write is $\mathsf{negl}(\lambda)$.*

We describe the algorithm in Algorithm 4.10 in Appendix B. For the Read/Write Phase, for simplicity of notation, let P denote the adaptive sequence of operations. We can model this by considering a trusted tape $\boldsymbol{I}$ with an associated online memory checker, and a work tape $\boldsymbol{W}$ where the all writes in P will be executed. For example, one can think of $\boldsymbol{I}$ as the input tape to a client or a portion of distinct online-checkable memory.

[6] More formally, at the end of the Read/Write Phase, the user $\mathcal{C}$ must send some "end of phase" symbol along with its query to indicate to M that there will be no more writes. We omit this technicality for simplicity. Any writes after this point will be considered an inadmissible user.

Algorithm 4.10 Offline memory checker.

Initial State:

- $\boldsymbol{I}$: Trusted tape with an associated online memory checker M (e.g., holding an input x).

Final State:

- $\boldsymbol{W}$: Work tape is left in a verifiable read-only state.

Local State: Counters T and T', a MAC secret key k, and a Read-Only Phase MAC secret key k'.

Algorithm: Read/Write Phase.

- Sample a key $k \leftarrow \mathsf{MACGen}(1^\lambda)$ for a MAC. Iterate over $\boldsymbol{W}$, initialize every $\boldsymbol{W}[\mathsf{addr}]$ to $(\varnothing, \mathsf{addr}, 0)$ with authentication key k. Initialize $T \leftarrow 0$.
- When the user sends the request $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$:
  - If $\mathsf{addr} \in \boldsymbol{I}$:
    - If $\mathsf{op} = \mathsf{write}$, output $\perp$.
    - If $\mathsf{op} = \mathsf{read}$:
      - Send $\mathsf{query} = (\mathsf{read}, \mathsf{addr}, \perp)$ to $\mathcal{A}$.
      - Run M on $\mathcal{A}(1^\lambda, \mathsf{query})$ and output $\perp$ if M aborts.
      - Otherwise, if M responds with data from $\mathcal{A}(1^\lambda, \mathsf{query})$, return data to $\mathcal{C}$.
  - If $\mathsf{addr} \in \boldsymbol{W}$:
    - Send $\mathsf{query} = (\mathsf{read}, \mathsf{addr}, \perp)$ to $\mathcal{A}$ to receive $(\mathsf{data}^*, \sigma)$
    - Compute $\mathsf{MACVer}_k(\mathsf{data}^*, \sigma)$. If verification fails, output $\perp$.
    - Unpack $(\mathsf{data}_{old}, \mathsf{addr}', \mathsf{count}) \leftarrow \mathsf{data}^*$.
    - If $\mathsf{addr}' \neq \mathsf{addr}$, output $\perp$.
    - If $\mathsf{op} = \mathsf{read}$, set $\mathsf{data}' = (\mathsf{data}_{old}, \mathsf{addr}, \mathsf{count} + 1)$.
    - If $\mathsf{op} = \mathsf{write}$, set $\mathsf{data}' = (\mathsf{data}, \mathsf{addr}, \mathsf{count} + 1)$.
    - Compute $\sigma' \leftarrow \mathsf{MAC}_k(\mathsf{data}')$.
    - Send $\mathsf{query} = (\mathsf{write}, \mathsf{addr}, (\mathsf{data}', \sigma'))$ to $\mathcal{A}$.
    - Send data_{old} to $\mathcal{C}$ if $\mathsf{op} = \mathsf{read}$.
    - Increment T.
- At the very end of the Read/Write Phase, initialize a counter $T' = 0$. Generate a new MAC key $k' \leftarrow \mathsf{MACGen}(1^\lambda)$. Iterate over $\mathsf{addr} \in \boldsymbol{W}$ and do the following:
  - Obtain $(\mathsf{data}^*, \sigma) \leftarrow \mathcal{A}(1^\lambda, \mathsf{query})$.
  - If $\mathsf{MACVer}_k(\mathsf{data}^*, \sigma) = \perp$, output $\perp$.
  - Unpack $(\mathsf{data}, \mathsf{addr}', \mathsf{count}) \leftarrow \mathsf{data}^*$.
  - If $\mathsf{addr}' \neq \mathsf{addr}$, output $\perp$.
  - Set $\mathsf{data}' = (\mathsf{data}, \mathsf{addr})$ and compute $\sigma' \leftarrow \mathsf{MAC}_{k'}(\mathsf{data}')$.
  - Send $\mathsf{query} = (\mathsf{write}, \mathsf{addr}, (\mathsf{data}', \sigma'))$ to $\mathcal{A}$.
  - Increment $T' \leftarrow T' + \mathsf{count}$.
- At the end, accept if and only if $T = T'$.

Algorithm: Read-Only Phase.

- When the user sends the request $(\mathsf{read}, \mathsf{addr}, \perp)$, send the identical query $(\mathsf{read}, \mathsf{addr}, \perp)$ to $\mathcal{A}$ to receive $(\mathsf{data}', \sigma')$, where $\mathsf{data}' = (\mathsf{data}, \mathsf{addr}')$.
- If $\mathsf{addr}' \neq \mathsf{addr}$ or $\mathsf{MACVer}_{k'}(\mathsf{data}', \sigma') = \perp$, output $\perp$.

- Otherwise, send data to $\mathcal{C}$.

Theorem 4.11. *There is a post-verifiable offline memory checker with perfect completeness for any $\mathcal{C}$ (controlled adaptively by $\mathcal{A}$) with amortized query complexity $O(1 + |\boldsymbol{W}|/|P|)$ in the Read/Write Phase and worst-case query complexity 1 in the Read-Only Phase. In particular, if $|\boldsymbol{W}| = O(|P|)$, the Read/Write Phase has amortized query complexity $O(1)$. Moreover, the local space of this memory checker is $O(1)$ words (and one PRF key), and the server space complexity (ignoring $\boldsymbol{I}$) is $O(|\boldsymbol{W}|)$.*

Proof. First, we prove the following lemma.

Lemma 4.12. *For the Read/Write Phase of Algorithm 4.10, if the memory functions correctly (i.e., the server is honest), then $T = T'$. Otherwise, $T \neq T'$ with probability $1 - \mathsf{negl}(\lambda)$.*

Proof. Let w_{addr} denote the number of times the memory checking algorithm sends a write query to $\mathcal{A}$ at addr, and let c_{addr} denote the value of count associated to addr during the final iteration over $\boldsymbol{W}$. Note that $T = T'$ if and only if $\sum_{\mathsf{addr}} w_{\mathsf{addr}} = \sum_{\mathsf{addr}} c_{\mathsf{addr}}$.

By unforgeability of MACs, every read must correspond to some authenticated write by the memory checker with probability $1 - \mathsf{negl}(\lambda)$. For the remainder of the argument, suppose that every read does in fact correspond to some authenticated write.

Fix some addr, and let $\alpha = w_{\mathsf{addr}}$. Let count_i denote the count associated to the ith write to addr. Define $c_0 = 0$, and for $1 \leq i \leq \alpha$, and let c_i denote the largest value of count associated to addr after the ith write across all the writes so far, i.e., $c_i = \max_{j \leq i} \mathsf{count}_j$.

Clearly, the sequence $c_0, c_1, \ldots, c_\alpha$ is nondecreasing. Moreover, for every write, since count_i is determined by the previous count value read, it has the form $\mathsf{count}_i = \mathsf{count}_j + 1$ for some $j < i$. Therefore, it is easy to see that $c_i \leq c_{i-1} + 1$. Moreover, equality holds only if and only if the read immediately preceding the ith write has count $\mathsf{count}_j = c_{i-1}$ for $j < i$.

By repeatedly applying this inequality, we have that

$$c_i \leq c_{i-1} + 1 \leq \cdots \leq c_0 + i = i,$$

where equality holds if and only if $c_j = j$ for all $0 \leq j \leq i$. In particular, $c_\alpha \leq \alpha$.

Since $c_i = \max_{j \leq i} \mathsf{count}_j \leq i$, note that $c_{i+1} = \max_{j \leq i+1} \mathsf{count}_j = i+1$ if and only if $\mathsf{count}_{i+1} = i+1$. Suppose $c_\alpha = \alpha$. Then, we must have $c_i = i$ for all $i \leq \alpha$, and in particular, $\mathsf{count}_i = i$ for all $i \leq \alpha$. Therefore, every read to addr must in fact correspond to the most recent contents of addr.

Suppose $c_{\mathsf{addr}} = w_{\mathsf{addr}} = \alpha$. Since c_{addr} must correspond to some count_i, we have that $c_{\mathsf{addr}} \leq c_\alpha = \alpha$, where equality holds if and only if $c_{\mathsf{addr}} = \mathsf{count}_\alpha = \alpha$. In particular, the memory functioned correctly on addr.

Now, note that $\sum c_{\mathsf{addr}} \leq \sum w_{\mathsf{addr}}$, where equality holds if and only if $c_{\mathsf{addr}} = w_{\mathsf{addr}}$ for all addr. In particular, the memory functioned correctly on all addr, as desired. □

Now, we consider the completeness of the memory checker. If the memory functions correctly, then all of authentication checks will pass. Moreover, by Lemma 4.12, we have that $T = T'$ at the end of the algorithm. Therefore, the memory checker will always accept in this case.

Now, we consider the soundness of the checker. Since the memory checker in Algorithm 4.10 accepts only if $T = T'$, by Lemma 4.12, the checker aborts with probability $1 - \mathsf{negl}(\lambda)$ if the memory does not function correctly. Therefore, this gives us that the memory checker is in fact sound in the Read/Write Phase. Soundness of the Read-Only Phase (with probability $1 - \mathsf{negl}(\lambda)$) directly follows from MAC unforgeability with key k', as each address gets at most one tag from the MAC.

Now, for efficiency, note that the total number of queries is $O(P)$ to execute the commands of $\mathcal{C}$, and then the memory checker does an additional linear time iteration over $\boldsymbol{W}$. Therefore, the overall number of queries in the Read/Write Phase is $O(|P| + |\boldsymbol{W}|)$, making the amortized query complexity $O(1) + O(|P|/|\boldsymbol{W}|)$. The query complexity of the Read-Only Phase is clearly 1 since the algorithm just makes one read to $\mathcal{A}$. Moreover, $O(1)$ words are sufficient for the counters, and both MACs can be implemented with one joint PRF key. □

5 Separated Memory Checkers

In this section, we define a variant of online memory checking that we show is both *necessary and sufficient* for generically compiling honest-but-curious ORAM clients into maliciously secure ones. At a high level, such an online memory checker M will be secure for arbitrary small-space RAM users $\mathcal{C}$ that do not interactively collude with the tampering server. As such, we call such memory checkers *separated memory checkers*. We give a formal definition below:

Definition 5.1 (Separated Memory Checker)**.** *We say that M is a* separated memory checker for users with space c *(measured in words) if the following two properties hold:*

1. **Completeness**: *If $\mathcal{A}$ is honest (i.e., behaves according to $\mathcal{F}_{\mathsf{RAM}}$), then for all requests generated by a user $\mathcal{C}$ with space c, where $\mathcal{C}$ cannot see physical queries made by M, M never aborts with probability $1 - \mathsf{negl}(\lambda)$.*
2. **Soundness**: *For all PPT $\mathcal{A}$ and all c-space users $\mathcal{C}$, where $\mathcal{A}$ and $\mathcal{C}$ do not communicate throughout the entire protocol, the probability that M ever sends some incorrect response to $\mathcal{C}$ is $\mathsf{negl}(\lambda)$. That is, for each request from $\mathcal{C}$, if $\mathcal{A}$ sends an incorrect response to M, M can either independently recover the correct answer and send it to $\mathcal{C}$, or it can abort by sending $\perp$ to $\mathcal{C}$.*

The differences between this definition and the earlier memory checking definitions (Definitions 4.1 and 4.2) are as follows:

- While online memory checkers according to Definition 4.1 are defined in terms of specific users $\mathcal{C}$, as is convenient when describing time-stamping, in Definition 5.1, these memory checkers are defined for general low-space users $\mathcal{C}$.
- While offline memory checkers according to Definition 4.2 are phrased in terms of users $\mathcal{C}$ that are adaptively controlled by the server $\mathcal{A}$, here we explicitly separate $\mathcal{C}$ and $\mathcal{A}$ throughout the interactive protocol. However, since we quantify over all low-space $\mathcal{C}$ and PPT $\mathcal{A}$, the user and server can effectively collude beforehand to try to break the memory checker. (Looking ahead, for any M that does not satisfy this new definition, this separation condition will allow us to isolate the malicious user and embed it inside an honest-but-curious ORAM construction.)

- We need completeness and soundness only against users with space c, as we only consider applying these memory checkers where the users are ORAM clients that have low space complexity.

Remark 5.2. *We note that the lower bound of Dwork et al. [DNRV09] can be adapted to show that that deterministic, non-adaptive separated memory checkers must also have* $\Omega(\log N / \log \log N)$ *overhead. Since we restrict the adversary to be separated, this definition is weaker than general purpose online memory checking, e.g., as in Blum et al. [BEG+91].*

Throughout this section, for simplicity, we will consider ORAMs for a database of size N to have physical server space $O(N)$, as constructions with optimal overhead achieve this server space complexity. First, we show that a generic compiler from honest-but-curious ORAM to maliciously secure ORAM must be a separated memory checker.

Theorem 5.3. *Suppose RAM program* Π *is such that that for all honest-but-curious ORAM* $\mathcal{C}$ *for a database of size* N *with local space at most* c*, the composition of* Π *and* $\mathcal{C}$ *is a maliciously secure ORAM size* N*. If* Π *has worst-case blowup in query complexity* ℓ*, then there is a separated memory checker for users with space* $O(c)$ *for a database of size* N *that has worst-case query complexity* $O(\ell)$*.*

In particular, if there does not exist a separated memory checker for users with space $O(1)$ *(and a PRF key) with worst-case blowup in query complexity* $O(1)$*, then there is no worst-case* $O(1)$*-blowup* Π *such that composing an arbitrary honest-but-curious,* $O(1)$ *local space ORAM with* Π *will be maliciously secure.*

For intuition, we first provide a sketch of the proof.

Proof sketch. We fix an honest-but-curious ORAM $\mathcal{C}$ with local space at most c and query complexity ℓ, and we construct a family of honest-but-curious ORAMs $\{\mathcal{C}'_{\mathcal{P}}\}_{\mathcal{P}}$ by interleaving $\mathcal{C}$ with arbitrary RAM users $\mathcal{P}$ that use space c, so that each $\mathcal{C}'_{\mathcal{P}}$ has local space complexity $O(c)$ and query complexity $O(\ell)$ (as shown in Algorithm 5.4). Furthermore, we augment $\mathcal{C}'_{\mathcal{P}}$ so that it detects any incorrect responses to queries from $\mathcal{P}$ with non-negligible probability, at which point it behaves arbitrarily (e.g., gives an incorrect response). We then construct a separated memory checker for $O(c)$ space users Π' by combining Π and $\mathcal{C}$ (as shown in Algorithm 5.6). We argue that if there is some $\mathcal{P}$ for which Π' is not a memory checker, then composing Π with $\mathcal{C}'_{\mathcal{P}}$ is not maliciously secure, which is a contradiction.

Now, we provide a formal proof.

Proof of Theorem 5.3. Fix an arbitrary program $\mathcal{P}$ with space at most c. Consider $\mathcal{C}'_{\mathcal{P}}$ defined as follows which runs $\mathcal{C}$ and $\mathcal{P}$ locally as described in Algorithm 5.4.

Algorithm 5.4 Honest-but-curious ORAM $\mathcal{C}'_{\mathcal{P}}$ constructed using $\mathcal{C}$ and $\mathcal{P}$ as sub-routines.

- Allocate logical server spaces $S_{\mathcal{C}}$ and $S_{\mathcal{P}}$ each of size N for $\mathcal{C}$ and $\mathcal{P}$, respectively.
- Choose a uniformly random address $\mathsf{addr}_{\mathsf{test}} \leftarrow S_{\mathcal{P}}$. Throughout the algorithm, $\mathcal{C}'_{\mathcal{P}}$ will use $O(1)$ local space to additionally locally keep track of all updates that $\mathcal{P}$ makes to $\mathsf{addr}_{\mathsf{test}}$.

- For every logical query $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ to $\mathcal{C}'_{\mathcal{P}}$,
  - Feed $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ to $\mathcal{C}$.
  - While $\mathcal{C}$ generates access $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$:
    - Conduct the access $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$, while treating addr' as an address in $S_{\mathcal{C}}$.
    - If $\mathcal{P}$ is not finished, continue running $\mathcal{P}$ to generate a new physical query $(\mathsf{op}'', \mathsf{addr}'', \mathsf{data}'')$. Conduct access $(\mathsf{op}'', \mathsf{addr}'', \mathsf{data}'')$ while treating addr'' as an address in $S_{\mathcal{P}}$. If $\mathsf{addr}'' = \mathsf{addr}_{\mathsf{test}}$:
      - If the operation was $(\mathsf{write}, \mathsf{addr}_{\mathsf{test}}, \mathsf{data}'')$ from program $\mathcal{P}$, update the locally stored value of $\mathsf{addr}_{\mathsf{test}}$ with data''.
      - If the operation was $(\mathsf{read}, \mathsf{addr}_{\mathsf{test}}, \perp)$, compare the value from the server with the locally stored value. If the values are not equal, return 1^w to the user.
  - When $\mathcal{C}$ outputs out, $\mathcal{C}'_{\mathcal{P}}$ also returns out to the user.

Claim 5.5. *If $\mathcal{C}$ is an honest-but-curious ORAM with space complexity c and overhead q, then $\mathcal{C}'_{\mathcal{P}}$ is an honest-but-curious ORAM with space complexity $O(c)$ and overhead $2q$.*

Proof. It suffices to give a simulator $\mathcal{S}'_{\mathcal{P}}$. Define $\mathcal{S}'_{\mathcal{P}}$ to be $\mathcal{C}'_{\mathcal{P}}$, where all uses of $\mathcal{C}$ are replaced with the simulator $\mathcal{S}$ for $\mathcal{C}$. First, note that for honest servers, $\mathcal{C}'_{\mathcal{P}}$ has no abort condition. Indistinguishability between the real and ideal worlds directly follows because honest-but-curious security of $\mathcal{C}$ because usage of $\mathcal{C}$ is black-box, because program $\mathcal{P}$ has no access to user requests $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$, and because the value locally stored for $\mathsf{addr}_{\mathsf{test}}$ will always be correct when interacting with an honest server. □

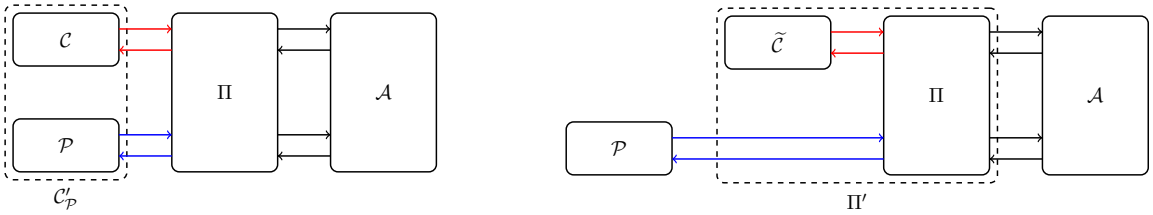
Suppose Π compiles all honest-but-curious ORAMs with space at most $O(c)$ into one that is maliciously secure. In particular, for all programs $\mathcal{P}$ with space at most c, the composition of Π and $\mathcal{C}'_{\mathcal{P}}$ is maliciously secure.

Let $\widetilde{\mathcal{C}}$ be $\mathcal{C}$ where the logical queries are fixed to first feeding it the query $(\mathsf{write}, 1, 0^w)$ and then repeatedly feeding it the fixed queries $(\mathsf{read}, 1, \perp)$. (As an aside, we remark that $\widetilde{\mathcal{C}}$ is a simulator for $\mathcal{C}$, although we will not need this fact.) Consider the following program Π' that uses Π and $\widetilde{\mathcal{C}}$ as sub-routines:

Algorithm 5.6 Candidate separated memory checker Π' with blow-up $O(\ell)$.

- Allocate logical server spaces $S_{\widetilde{\mathcal{C}}}$ and $S_{\mathcal{U}}$ for $\widetilde{\mathcal{C}}$ and the user $\mathcal{U}$, respectively.
- For every logical query $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ to Π',
  - Run $\widetilde{\mathcal{C}}$ to generate a new physical query $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$.
  - Feed $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$ from $\widetilde{\mathcal{C}}$ into Π, and perform the corresponding physical accesses of Π to server space $S_{\widetilde{\mathcal{C}}}$. Return the output of Π to $\widetilde{\mathcal{C}}$.
  - Feed $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ from the user $\mathcal{U}$ into Π, and perform corresponding accesses of Π to server space $S_{\mathcal{U}}$. Return the output of Π to the user.

To give some intuition, the composition of Π' with user $\mathcal{P}$ is identical to the composition of Π and $\mathcal{C}'_{\mathcal{P}}$, where the user queries to $\mathcal{C}'_{\mathcal{P}}$ are fixed and the outputs of $\mathcal{C}'_{\mathcal{P}}$ are possibly incorrect. We will think of Π' as a separated memory checker and $\mathcal{P}$ as its user, and we will think of $\mathcal{C}'_{\mathcal{P}}$ as an honest-but-curious ORAM that Π compiles into a malicious one. The equivalence between these compositions allows us to reason back and forth between memory checking and ORAM security. We illustrate this pictorially in Figure 1.



(a) Pictorial depiction of $\mathcal{C}'_{\mathcal{P}}$ from Algorithm 5.4.

(b) Pictorial depiction of Π' from Algorithm 5.6.

Figure 1: Comparing interaction of $\mathcal{C}'_{\mathcal{P}}$ and Π with the interaction of $\mathcal{P}$ and Π'.

Claim 5.7. *Π' is a separated memory checker with blow-up $O(\ell)$ for all RAM programs with space at most c.*

Proof. Suppose that Π' is *not* a separated memory checker for users with space $O(c)$. Then, either completeness or soundness does not hold for Π' with respect to some user program $\mathcal{P}$ with space at most c. We now argue that the composition of Π and $\mathcal{C}'_{\mathcal{P}}$ (as defined in Algorithm 5.4) is not a maliciously secure ORAM despite $\mathcal{C}$ being an honest-but-curious ORAM.

Suppose Π' is not complete for some c-space user program $\mathcal{P}$. Then, this means that the sub-routine Π in Π' aborts against an honest server with non-negligible probability infinitely often.

Consider $\mathcal{C}'_{\mathcal{P}}$ on the sequence where we first feed it the logical query $(\mathsf{write}, 1, 0^w)$ and then repeatedly feed it queries $(\mathsf{read}, 1, \perp)$ (i.e., identical to $\widetilde{\mathcal{C}}$), and consider Π as a compiler in this setting. Note that the distribution of queries that Π receives as a compiler is identical to the distribution of queries that Π receives as a sub-routine of Π'. Therefore, Π will abort with non-negligible probability infinitely often as a compiler for $\mathcal{C}'_{\mathcal{P}}$, which contradicts the completeness of Π composed with $\mathcal{C}'_{\mathcal{P}}$ (as a maliciously secure ORAM).

Now suppose Π' is not sound for some c-space user program $\mathcal{P}$. Then, there exists some adversarial server $\mathcal{A}$ such that $\mathcal{P}$ and $\mathcal{A}$ can force Π' to make a mistake with non-negligible probability $\epsilon(\lambda)$ infinitely often.

Consider $\mathcal{C}'_{\mathcal{P}}$ on the sequence where we first feed it the logical query $(\mathsf{write}, 1, 0^w)$ and then repeatedly feed it queries $(\mathsf{read}, 1, \perp)$ (i.e., identical to $\widetilde{\mathcal{C}}$), and consider Π as a compiler in this setting. Up until the first mistake that Π' makes to $\mathcal{P}$, the distribution of queries to Π is the same, so the probability that Π makes a mistake is also at least $\epsilon(\lambda)$ (infinitely often). Note that $\mathsf{addr}_{\mathsf{test}}$, as sampled uniformly by $\mathcal{C}'_{\mathcal{P}}$, is equal to the logical address of the the first corrupted response for Π' with probability at least $\frac{1}{N}$. Therefore, $\mathcal{C}'_{\mathcal{P}}$ detects this mistake with probability at least $\epsilon(\lambda)/N \geq \epsilon(\lambda)/\mathsf{poly}(\lambda)$, which is non-negligible in λ infinitely often. As a result, $\mathcal{C}'_{\mathcal{P}}$ outputs 1^w with non-negligible probability infinitely often. As such, a malicious ORAM adversary can run the malicious server $\mathcal{A}$, feed the ORAM $(\mathsf{write}, 1, 0^w)$ and then repeatedly $(\mathsf{read}, 1, \perp)$ to distinguish

between the real and ideal worlds, as 1^w will be outputted with non-negligible probability in the real world but not the ideal world. Therefore, the obliviousness and correctness of Π composed with $\mathcal{C}'_{\mathcal{P}}$ (as a maliciously secure ORAM) is violated. □

Since Π' has the desired properties, the proof of Theorem 5.3 is complete. □

Now, we show that separated memory checkers are also sufficient to compile any honest-but-curious ORAM into one that is maliciously secure.

Theorem 5.8. *Suppose that M is a separated memory checker for users with space $O(c)$. Then, for any honest-but-curious ORAM $\mathcal{C}$ with client space complexity c, M is an online memory checker for $\mathcal{C}$, in the sense of Definition 4.1.*

For intuition, we provide a sketch of the proof.

Proof sketch. Since $\mathcal{C}$ has space complexity c, there exists a simulator $\mathcal{S}$ for $\mathcal{C}$ with space complexity $O(c)$. In particular, M is a good memory checker for $\mathcal{S}$ since $\mathcal{S}$ is *separated* from the ORAM user. Hence, $\mathcal{C}$ with arbitrary inputs is indistinguishable from a separated RAM user, namely $\mathcal{S}$. Therefore, we argue that if M doesn't satisfy either completeness or soundness for $\mathcal{C}$, one can use M to construct an *honest-but-curious* adversary $\mathcal{A}$ (i.e., one that does not send incorrect responses) that distinguishes $\textsc{Real}(\mathcal{C}, \mathcal{A})$ and $\textsc{Ideal}(\mathcal{F}_{\mathsf{RAM}}, \mathcal{S}, \mathcal{A})$.

Proof of Theorem 5.8. Supposing otherwise, there exists an honest-but-curious ORAM $\mathcal{C}$ with local space c for which either online memory checking soundness or completeness of M does not hold for $\mathcal{C}$. Note that without loss of generality, there exists an honest-but-curious simulator $\mathcal{S}$ for $\mathcal{C}$ with space $O(c)$. This is due to the fact that one can define the simulator to simply be $\mathcal{C}$ run on a fixed sequence, e.g. $(\mathsf{write}, \mathsf{addr}, 0)$ for some fixed address addr.

Soundness. Suppose that M is not sound for $\mathcal{C}$. In other words, there exists an adversary $\mathcal{A}_M$ that controls both the logical queries sent to $\mathcal{C}$ as well as the server's responses to M that *forces M to send incorrect responses* with non-negligible probability infinitely often. Consider the following adversary $\mathcal{A}_{\mathcal{C}}$ against the implementation $\mathcal{C}$:

1. Allocate server space D for $\mathcal{C}$. This space will not be tampered with.
2. Independently, internally within $\mathcal{A}_{\mathcal{C}}$, allocate physical server space to run M. Use $\mathcal{A}_M$ to control this server that M is interacting with.
3. Initialize $\mathsf{flag} = \mathsf{false}$.
4. Until $\mathcal{A}_M$ aborts, generate $(\mathsf{op}, \mathsf{addr}, \mathsf{data}) \leftarrow \mathcal{A}_M$, and send it to $\mathcal{C}$. For every access $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$ from $\mathcal{C}$,
   (a) Perform the command honestly on server space D, and generate the honest output out (to later send to the client).
   (b) Feed $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$ to M and have M interact with $\mathcal{A}_M$ as the server.
   (c) If $\mathsf{op} = \mathsf{read}$, compare the response from D with the response from M. If the responses do not match, set $\mathsf{flag} = \mathsf{true}$, and stop running M.

(d) Send the honest output out from Step 4a to the $\mathcal{C}$.

5. If $\mathsf{flag} = \mathsf{true}$, output 1. Else, output 0.

We claim that $\mathcal{A}_\mathcal{C}$ breaks the honest-but-curious security of $\mathcal{C}$. First, note that all responses to $\mathcal{C}$ are in fact honest, and hence $\mathcal{A}_\mathcal{C}$ is in fact an honest-but-curious adversary. In $\textsc{Real}(\mathcal{C}, \mathcal{A}_\mathcal{C})$, M sends an incorrect response with non-negligible probability infinitely often by definition of $\mathcal{A}_M$, i.e., $\mathcal{A}_\mathcal{C}$ outputs 1 with non-negligible probability infinitely often. On the other hand, in $\textsc{Ideal}(\mathcal{F}_{\mathsf{RAM}}^{N,w}, \mathcal{S}, \mathcal{A}_\mathcal{C})$, M is instead interacting with $\mathcal{S}$, which is in fact separated from $\mathcal{A}_M$. By soundness of M as a separated memory checker, M sends an incorrect with negligible probability, i.e., $\mathcal{A}_\mathcal{C}$ outputs 1 with negligible probability. Therefore, $\mathcal{A}_\mathcal{C}$ distinguishes the two worlds with non-negligible probability infinitely often.

Completeness. Suppose that M is not complete for $\mathcal{C}$. In other words, there exists an adversary $\mathcal{A}_M$ that controls logical queries sent to $\mathcal{C}$ and sees (but does not control) physical server queries that *forces M to either abort or make a mistake* with non-negligible probability infinitely often. Consider the following adversary $\mathcal{A}_\mathcal{C}$:

1. Allocate server space D for $\mathcal{C}$. This space will not be tampered with.
2. Independently, internally within $\mathcal{A}_\mathcal{C}$, allocate physical server space D_M to run M. This space will not be tampered with.
3. Initialize $\mathsf{flag} = \mathsf{false}$.
4. Until $\mathcal{A}_M$ aborts, generate $(\mathsf{op}, \mathsf{addr}, \mathsf{data}) \leftarrow \mathcal{A}_M$, and send it to $\mathcal{C}$. For every access $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$ from $\mathcal{C}$,
   (a) Perform the command honestly on server space D, and generate the honest output out (to later send to the client).
   (b) Feed $(\mathsf{op}', \mathsf{addr}', \mathsf{data}')$ to M and have M interact with D_M as the server.
   (c) If M aborts, set $\mathsf{flag} = \mathsf{true}$ and stop running M.
   (d) If $\mathsf{op} = \mathsf{read}$, compare the response from D with the response from M. If the responses do not match, set $\mathsf{flag} = \mathsf{true}$, and stop running M.
   (e) Send the honest output out from Step 4a to the $\mathcal{C}$.
5. If $\mathsf{flag} = \mathsf{true}$, output 1. Else, output 0.

We claim that $\mathcal{A}_\mathcal{C}$ breaks the honest-but-curious security of $\mathcal{C}$. First, note that all responses to $\mathcal{C}$ are in fact honest, and hence $\mathcal{A}_\mathcal{C}$ is in fact an honest-but-curious adversary. In $\textsc{Real}(\mathcal{C}, \mathcal{A}_\mathcal{C})$, M aborts or sends an incorrect response with non-negligible probability infinitely often by definition of $\mathcal{A}_M$. On the other hand, in $\textsc{Ideal}(\mathcal{F}_{\mathsf{RAM}}^{N,w}, \mathcal{S}, \mathcal{A}_\mathcal{C})$, M is instead interacting with $\mathcal{S}$, which is in fact separated from $\mathcal{A}_M$. By completeness and soundness of M as a separated memory checker, M aborts or sends an incorrect response only with negligible probability. Therefore, $\mathcal{A}_\mathcal{C}$ distinguishes the two worlds with non-negligible probability infinitely often. □

Corollary 5.9. *Suppose that $\mathcal{C}$ is an honest-but-curious ORAM with client space complexity c, and suppose there is a separated memory checker M for users with space $O(c)$. Then, the composition of M and $\mathcal{C}$ is a maliciously secure ORAM.*

Proof. This immediately follows from combining Theorem 5.8 and Theorem 4.5. □

6 Write-Deterministic Implementations

In this section, we formalize how an *access-deterministic* honest-but-curious oblivious algorithm can be made oblivious against malicious servers with $O(1)$ blowup in overhead. In fact, we only need the *writes* to be deterministic, meaning that the reads can occur to arbitrary addresses. This motivates the following definition.

Definition 6.1. *We say a RAM program $\mathcal{C}$ is* $(1-\epsilon)$-write-deterministic *if there exists a fixed set $S_{\mathcal{C}}$ of time and address pairs such that for all x, whenever $\mathcal{C}$ does not abort,*

$$\{(t, \mathsf{addr}) : \textit{on input } x, \mathcal{C} \textit{ makes a write to } \mathsf{addr} \textit{ at time } t\} = S_{\mathcal{C}}$$

when $\mathcal{C}$ is interacting with an honest RAM server, and for all x, $\mathcal{C}$ aborts with probability at most ϵ when interacting with an honest server. That is, for all x, the timing and locations of writes that $\mathcal{C}$ makes are deterministic and completely independent of x. (This definition says nothing about the content of the writes or the contents or addresses of the reads, but it does implicitly fix the timing of the reads.) Whenever ϵ is unspecified, we take it to mean $\epsilon = 0$.

Since the writes are deterministic, this definition can be seen as asserting the existence of a time-stamp function (Definition 4.3), or "time-labeling" in the sense of Goldreich and Ostrovsky [GO96], but without any requirement that such a function is computable in low space. We now show that this notion (without any efficiency requirements) is sufficient to get malicious security.

As mentioned in Section 2, offline memory checking a write-deterministic honest-but-curious oblivious algorithm directly may not be maliciously secure. We circumvent this by first running the honest-but-curious algorithm on some fixed dummy input (independent of the real input) to essentially build a time-stamp array. By offline-checking up until the array is built, we ensure correctness of the time-stamp array. Note that obliviousness is guaranteed throughout the offline check because nothing about the real input is used when building the array. After this check is finished, the algorithm now runs on the real input, using the post-verifiability of the time-stamp array to verify time-stamps in MACs in an online way. This allows one to catch any replay attacks by the adversarial server immediately. This is detailed in Algorithm 6.3.

Theorem 6.2. *Suppose $\mathcal{C}_{\mathcal{F}}$ is a $(1-\delta)$-honest-but-curious oblivious, $(1-\epsilon)$-write-deterministic implementation for a stateless functionality $\mathcal{F}$ with client space complexity c, server space s and query complexity q. Then, there exists a $(1 - O(\delta) - O(\epsilon) - \mathsf{negl}(\lambda))$-maliciously secure oblivious implementation $\mathcal{C}'_{\mathcal{F}}$ for $\mathcal{F}$ with query complexity $O(q+s)$, server space complexity $O(q+s)$, and client space complexity $O(c)$ (and one PRF key).*

Proof. We claim that $\mathcal{C}'_{\mathcal{F}}$ in Algorithm 6.3 is a maliciously secure implementation of $\mathcal{F}$ with the desired efficiency.

Algorithm 6.3 Maliciously secure implementation $\mathcal{C}'_{\mathcal{F}}$.

Input: x.

Output: $\mathcal{F}(x)$.
Memory checking: In the grayed-out portion of the algorithm, we use the post-verifiable offline memory checker from Theorem 4.11. If the memory checker aborts, so does this algorithm. Everything in the grayed-out box constitutes the Read/Write Phase, and Line 5 is the Read-Only Phase.
Authentication: Outside the grayed-out portion of the algorithm, for every write query $(\mathsf{write}, \mathsf{addr}, \mathsf{data}')$ generated by our algorithm $\mathcal{C}'_{\mathcal{F}}$, data' is replaced with $\mathsf{data}'' := (\mathsf{data}', \sigma \leftarrow \mathsf{MAC}_{\mathsf{sk}}(\mathsf{data}', \mathsf{addr}))$. Every read query $(\mathsf{read}, \mathsf{addr}, \perp)$ is passed through authentication, namely unpacking $\mathsf{data}'' = (\mathsf{data}', \sigma^*)$, checking $\mathsf{MACVer}_{\mathsf{sk}}((\mathsf{data}', \mathsf{addr}), \sigma^*) = 1$, aborting if verification fails, and returning data' if not. For readability, we do not explicitly write these authentication tags and checks.
The algorithm.

1. Initialize the following arrays:
   - Array $\boldsymbol{N}$ containing q elements, initialized to $\varnothing$. If the offline check passes, for a write at time ctr to addr, $\boldsymbol{N}[\mathsf{ctr}]$ will contain the subsequent time addr is written to (if such a time exists), and otherwise $\boldsymbol{N}[\mathsf{ctr}] = \varnothing$.
   - Array $\boldsymbol{W}$ of size s initialized to $(\varnothing, \varnothing)$. This will be used as the work-space for $\mathcal{C}_{\mathcal{F}}$.
2. Make copies $\boldsymbol{N}^{\mathsf{off}}$ and $\boldsymbol{W}^{\mathsf{off}}$ of $\boldsymbol{N}$ and $\boldsymbol{W}$ respectively.
3. Initialize a counter $\mathsf{ctr} = 0$ locally (this will be used to keep track of the number of reads and writes).
4. Let $\mathcal{C}_0$ be a modification of $\mathcal{C}_{\mathcal{F}}$ where the input is hard-coded to some fixed, well-formed dummy input (e.g., 0^n), independent of the real input x. Run $\mathcal{C}_0$ using $\boldsymbol{W}^{\mathsf{off}}$ as its designated work space on the server, aborting if it ever makes more than q queries. Now, we convert a query $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ generated by $\mathcal{C}_0$ (interpreting all addr to be in $\boldsymbol{W}^{\mathsf{off}}$) into the following sequence queries:
   - If $\mathsf{op} = \mathsf{write}$:
     - Retrieve $\boldsymbol{W}^{\mathsf{off}}[\mathsf{addr}]$ from $\mathcal{A}$.
     - Let $\mathsf{data}' \leftarrow \mathcal{A}(1^\lambda, \mathsf{query})$. Unpack $(\mathsf{data}_{old}, \mathsf{ctr}') \leftarrow \mathsf{data}'$.
     - Set $\boldsymbol{N}^{\mathsf{off}}[\mathsf{ctr}'] = \mathsf{ctr}$.
     - Write data to $\boldsymbol{W}^{\mathsf{off}}[\mathsf{addr}]$.
   - If $\mathsf{op} = \mathsf{read}$:
     - Retrieve $\boldsymbol{W}^{\mathsf{off}}[\mathsf{addr}]$ from $\mathcal{A}$ and send it to $\mathcal{C}_0$.
   - Increment ctr.
5. Copy the contents of $\boldsymbol{N}^{\mathsf{off}}$ to $\boldsymbol{N}$ using the post-verifiability of the offline checker. Note that $\boldsymbol{W}$ is still all $(\varnothing, \varnothing)$.
6. Re-initialize counter $\mathsf{ctr} = 0$, and keep track of this locally.
7. Now, we run $\mathcal{C}_{\mathcal{F}}$ on input x. For every $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ sent by $\mathcal{C}_{\mathcal{F}}$:
   - If $\mathsf{op} = \mathsf{write}$:
     - Write $(\mathsf{data}, \mathsf{ctr}) \rightarrow \boldsymbol{W}[\mathsf{addr}]$.
   - If $\mathsf{op} = \mathsf{read}$:
     - Read $\boldsymbol{W}[\mathsf{addr}]$ to retrieve $(\mathsf{data}', \mathsf{ctr}')$ from $\mathcal{A}$.
     - Read $\boldsymbol{N}[\mathsf{ctr}']$ to verify that $\boldsymbol{N}[\mathsf{ctr}'] > \mathsf{ctr}$, and abort if this does not hold.
     - Send data' to $\mathcal{C}_{\mathcal{F}}$.

- Increment ctr.

8. Output whatever $\mathcal{C}_{\mathcal{F}}$ outputs.

Consider the following simulator $\mathcal{S}$ that is identical to $\mathcal{C}'_{\mathcal{F}}$, except it runs $\mathcal{S}_{\mathcal{F}}$ instead of $\mathcal{C}_{\mathcal{F}}$ at Step 7, where $\mathcal{S}_{\mathcal{F}}$ is the honest-but-curious simulator for $\mathcal{C}_{\mathcal{F}}$. Now we show that no PPT $\mathcal{A}$ can distinguish between $\textsc{Real}(\mathcal{C}'_{\mathcal{F}}, \mathcal{A})$ and $\textsc{Ideal}(\mathcal{F}, \mathcal{S}, \mathcal{A})$ with probability greater than $\delta + \mathsf{negl}(\lambda)$.

Experiment Hybrid_0: The real world $\textsc{Real}(\mathcal{C}'_{\mathcal{F}}, \mathcal{A})$.

Experiment Hybrid_1: The experiment is the same as the previous hybrid, except we replace the offline memory checker M with an idealized version M'. Specifically, M' behaves identically to M, except ensures with probability 1 (instead of $1 - \mathsf{negl}(\lambda)$) that the soundness property holds, i.e., if M' doesn't abort after the first phase (Read/Write), then all responses M' has previously given are correct, and all data^* sent back to $\mathcal{C}'_{\mathcal{F}}$ by M' in the second phase (Read-Only) are correct.

Claim 6.4. $|\Pr[\mathsf{Hybrid}_1 = 1] - \Pr[\mathsf{Hybrid}_0 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This directly follows from soundness of the offline memory checker M. □

Experiment Hybrid_2: In this hybrid, we modify the client and augment it with additional space to check that all ciphertexts passing the authentication verification have been generated by the MAC before, aborting if this is not the case. (Note that in the real world, a client cannot generally do this because this takes $\Omega(N)$ space.)

Claim 6.5. $|\Pr[\mathsf{Hybrid}_2 = 1] - \Pr[\mathsf{Hybrid}_1 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This directly follows from unforgeability of MACs, as any data passing the verification check must have been queried to the $\mathsf{MAC}_{\mathsf{sk}}(\cdot)$ oracle (with $1 - \mathsf{negl}(\lambda)$ probability). □

Experiment Hybrid_3: In this hybrid, we ensure with probability 1 that in Step 7, any data' sent back to $\mathcal{C}_{\mathcal{F}}$ is correct. (This can be checked in the hybrid by keeping an auxiliary, honest version of $\boldsymbol{W}$.)

Claim 6.6. $\Pr[\mathsf{Hybrid}_3 = 1] = \Pr[\mathsf{Hybrid}_2 = 1]$.

Proof. If $\mathcal{C}'_{\mathcal{F}}$ aborts before Step 7, then the hybrids are identical. If $\mathcal{C}'_{\mathcal{F}}$ does not abort before Step 7, by perfect unforgeability of the MAC and perfect soundness of the memory checker (from the previous hybrids), we know that the contents of $\boldsymbol{N}$ are correct with respect to $\mathcal{C}_0$, i.e., if addr is written to at time ctr, $\boldsymbol{N}[\mathsf{ctr}]$ contains the subsequent time addr is written to (if one exists) with respect to $\mathcal{C}_0$.

Now, we proceed by induction on each step of the execution of $\mathcal{C}_{\mathcal{F}}$ on input x. The base case is that no data' is returned incorrectly before the first read, which immediately holds. For the inductive step, at a given time ctr, by write-determinism of $\mathcal{C}_{\mathcal{F}}$ and by the inductive hypothesis, we know that the write sequences of $\mathcal{C}_0$ and $\mathcal{C}_{\mathcal{F}}$ are identical since the inductive hypothesis asserts that all data' so far have been correct, so it is interacting with an honest server. Therefore, $\boldsymbol{N}$ (up to time ctr) must be correct for $\mathcal{C}_{\mathcal{F}}$ as well. Now, by perfect unforgeability of the MAC, we know that the only possible way data' could be wrong for a read to addr is via a replay attack, where some incorrect

$(\mathsf{data}^*, \mathsf{ctr}^*)$ is returned by $\mathcal{A}$ for $\mathsf{ctr}^* < \mathsf{ctr}'$, where ctr^* is an old time of a write to addr and ctr' is the true time of the most recent write to addr before the current time ctr. However, by correctness of $\boldsymbol{N}$, since there was a write to addr at time ctr', we know $\boldsymbol{N}[\mathsf{ctr}^*] \leq \mathsf{ctr}' < \mathsf{ctr}$, so $\mathcal{C}'_{\mathcal{F}}$ must have aborted within Step 7 before sending back data^*. This completes the inductive argument, showing that all data' sent back to $\mathcal{C}_{\mathcal{F}}$ are correct. □

Experiment Hybrid_4: Replace $\mathcal{C}_{\mathcal{F}}$ in Step 7 with $\mathcal{S}_{\mathcal{F}}$, and replace the output of $\mathcal{C}_{\mathcal{F}}$ (and hence $\mathcal{C}'_{\mathcal{F}}$) with the output of the functionality $\mathcal{F}$.

Claim 6.7. $|\Pr[\mathsf{Hybrid}_4 = 1] - \Pr[\mathsf{Hybrid}_3 = 1]| \leq \delta$.

Proof. Since all data' sent back to $\mathcal{C}_{\mathcal{F}}$ (in Hybrid_3) or $\mathcal{S}_{\mathcal{F}}$ (in Hybrid_4) are correct by Hybrid_3, by invoking the honest-but-curious $(1-\delta)$-obliviousness of $\mathcal{C}_{\mathcal{F}}$, the two hybrids are $(1-\delta)$-indistinguishable. □

Experiment Hybrid_5: In this hybrid, we unravel the hybrids 0-3, namely:

- Remove the perfect data' verification in Hybrid_3.
- Conduct the MAC check rather than saving all the tags for authentication.
- Replace the idealized offline checker M' with the original version M.

Claim 6.8. $|\Pr[\mathsf{Hybrid}_5 = 1] - \Pr[\mathsf{Hybrid}_4 = 1]| \leq \mathsf{negl}(\lambda)$.

Note that Hybrid_5 is in fact exactly $\textsc{Ideal}(\mathcal{F}, \mathcal{S}, \mathcal{A})$, thereby completing the proof.

Next, we show completeness. Since the algorithm is $(1-\epsilon)$-write-deterministic, we know that both $\mathcal{C}_0$ and $\mathcal{C}$ each abort with probability at most ϵ against an honest server. Therefore, by a union bound, with probability at least $1-2\epsilon$, neither of them abort against an honest server. Since our offline memory checker has completeness $1-\mathsf{negl}(\lambda)$ (in fact, completeness 1), this implies that $\mathcal{C}'_{\mathcal{F}}$ overall has completeness $1-2\epsilon-\mathsf{negl}(\lambda)$.

Lastly, we argue that $\mathcal{C}'_{\mathcal{F}}$ has the desired efficiency. The server space complexity is now $O(q+s)$, since we use $O(q)$ space to store the array $\boldsymbol{N}$ (see Remark 6.9) and $O(s)$ space to store the work space of $\mathcal{C}_0$ and $\mathcal{C}_{\mathcal{F}}$. Furthermore, the total number of queries made by our offline memory checker is $O(q+(q+s)) = O(q+s)$, since the number of queries is $O(q)$ and the total work-tape size being offline-checked is $O(q+s)$. The number of queries made by running $\mathcal{C}_0$ and $\mathcal{C}_{\mathcal{F}}$ is $O(q)$, bringing the total query complexity of $\mathcal{C}'_{\mathcal{F}}$ to $O((q+s)+q) = O(q+s)$. Lastly, the client local space complexity is $O(c)$ since we run $\mathcal{C}_0$ and $\mathcal{C}_{\mathcal{F}}$, and we can run the memory checker and MAC with one PRF key (and $O(1)$ words to store all necessary counters). □

Remark 6.9. *The server space complexity of $\mathcal{C}'_{\mathcal{F}}$ in Theorem 6.2 can be reduced to $O(s)$ generating the data structure $\boldsymbol{N}$ on the fly by re-running Steps 4 through 7 every $O(s)$ steps. In our case, the bound of $O(q+s)$ on the server space complexity is sufficient to achieve our $O(N)$ server space ORAM construction, so we give the simpler construction and proof for readability.*

In Section 7, we argue that many of our ORAM building blocks such as sorting, tight compaction and Cuckoo hashing are in fact access-deterministic and therefore write-deterministic. Therefore, these can be made maliciously secure with low overhead by directly applying Theorem 6.2.

7 Overview of Maliciously Secure Building Blocks

Our ORAM construction relies on several oblivious building blocks. We describe how we modify existing algorithms for these building blocks to be maliciously secure with small (usually $O(1)$) *blowup*, i.e., multiplicative increase in query complexity, relative to the honest-but-curious counterparts. For the explicit constructions and proofs, see Appendix C.

- **Maliciously Secure Oblivious RAM with $O(\log^4 n)$ overhead (Appendix C.1)**

  We modify the existing perfectly secure honest-but-curious ORAM with $O(\log^3 n)$ overhead due to Chan et al. [CNS18] to obtain a maliciously secure ORAM with $O(\log^4 n)$ overhead with only a $\mathsf{negl}(\lambda)$ security loss independently of n, as long as $n \leq \mathsf{poly}(\lambda)$.

- **Oblivious Sorting Algorithms (Appendix C.2)**

  We argue that the sorting algorithm of Ajtai et al. [AKS83] and the packed sorting algorithm of Asharov et al. [Bat68, AKLS21] are access-deterministic, and thus can be made maliciously secure with $O(1)$ blowup in query complexity.

- **Oblivious Two-Key Dictionary (Appendix C.3)**

  Similarly to Asharov et al. [AKLS21], we give a maliciously secure implementation of a dictionary where each element has two keys such that one can pop elements with respect to either key. We do this by constructing a non-oblivious data structure and then composing our maliciously secure ORAM from C.1 with this data structure.

- **Oblivious Random Permutation (Appendix C.4)**

  We modify the random shuffling algorithm of Chan et al. [CCS17] to be maliciously secure with $O(1)$ blowup using offline checking.

- **Oblivious Bin Placement (Appendix C.5)**

  A placement algorithm was proposed by Chan et al. [CGLS17] to obliviously route elements in an array. We show that this can be made maliciously secure with $O(1)$ blowup by arguing that it is time-stampable.

- **Oblivious Balls-into-Bins Sampling (Appendix C.6)**

  We adapt the algorithm of Asharov et al. [AKL$^+$20] by applying online memory checking techniques to provide a maliciously secure algorithm for sampling balls-in-bins loads efficiently.

- **Tight Compaction (Appendix C.7)**

  Tight compaction takes as input an array where some elements are marked and outputs a permutation of the array so that all the marked elements appear before the unmarked elements. Since the linear-time algorithm of Asharov et al. [AKL$^+$20] is access-deterministic, we can apply Theorem 6.2.

- **Intersperse (Appendix C.7)**

  Intersperse is an algorithm which, given two randomly shuffled arrays, outputs a random shuffle of the concatenation of the two arrays. Asharov et al. [AKL$^+$20] give a access-deterministic linear-time algorithm to do this, so we apply Theorem 6.2 to argue that it can be made maliciously secure.

- **Perfect Random Permutation (Appendix C.7)**

  Asharov et al. [AKL+20] give an algorithm to randomly shuffle lists with $O(n \log n)$ queries with no failure probability. Since the algorithm is access-deterministic, we can once again apply Theorem 6.2 to make it maliciously secure with $\mathsf{negl}(\lambda)$ security loss, independent of the size of n.

- **Oblivious Cuckoo Hashing (Appendix C.8)**

  Chan et al. [CGLS17] give an oblivious algorithm to compute Cuckoo hash tables with essentially $O(1)$ lookup time. We argue this algorithm can be made maliciously secure with $O(1)$ blowup because it is access-deterministic.

- **Deduplication (Appendix C.9)**

  Asharov et al. [AKLS21] give a linear-time oblivious algorithm to take a union of two randomly shuffled arrays while removing duplicates. We make this maliciously secure with $O(1)$ blowup using our maliciously secure hash table construction as given in Section 8.

8 Maliciously Secure Oblivious Hash Table

In this section, we construct an efficient maliciously secure oblivious implementation of the hash table functionality $\mathcal{F}_{\mathsf{HT}}$ (Functionality 8.1). Following previous works, in our ORAM construction, we use this hash table implementation to implement each layer of the hierarchical ORAM.

The works of Asharov et al. [AKL+20, AKLS21] provide an implementation CombHT of $\mathcal{F}_{\mathsf{HT}}$ with the following properties:

- The input array of size n satisfies $\log^{11} \lambda \leq n \leq \mathsf{poly}(\lambda)$.
- Both $\mathsf{CombHT.Build}()$ and $\mathsf{CombHT.Extract}()$ run in $O(n)$ time.
- Each $\mathsf{CombHT.Lookup}()$ call takes $O(1)$ time ignoring linear scans over a $O(\log \lambda)$-sized stash, which in the final ORAM construction will end up being amortized over many lookups.

At a high level, this is achieved by hashing in two levels: the first level of hashing is a standard balls and bins hashing into $\frac{n}{\mathsf{polylog}(\lambda)}$ bins (which they call BigHT), and each bin is then implemented as a $\mathsf{polylog}(\lambda)$ size Cuckoo hash table (called $\mathsf{SmallHT}$). In particular, they present CombHT in the $\mathsf{SmallHT}$-hybrid model.

However, as presented, it is not clear how the CombHT construction in [AKL+20, AKLS21] can be made obliviously secure with $O(1)$ multiplicative overhead. If each $\mathsf{SmallHT}$ instance is used in a black-box way, by a variant of the marking lower bound in Section 2.2.2, online memory checking which indices were accessed in each $\mathsf{SmallHT}$ instance (as needed to remove accessed elements when implementing $\mathsf{SmallHT.Extract}()$) is not possible in low space.

Therefore, to make a maliciously secure version of CombHT with small local space, we combine the $\mathsf{SmallHT}$ instances in a non-black-box way. Specifically, in MalHT, we time-stamp a list of all accessed elements across $\mathsf{SmallHT}$ instances as the lookups occur, and then to support $\mathsf{MalHT.Extract}()$, we do an offline memory check, as the access pattern will be conditionally write-deterministic on the time-stamped list of lookups.

Functionality 8.1 $\mathcal{F}_{\mathsf{HT}}^{n}$: Hash Table Functionality for Non-Recurrent Lookups

Command $\mathcal{F}_{\mathsf{HT}}^{n}.\mathsf{Build}()$:

- **Input:** An array $\boldsymbol{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted $(k_i, v_i) \in \{0,1\}^D \times \{0,1\}^D$ where $D := O(1) \cdot w$ where w is the plaintext word size.
- **Input assumptions:** The elements in the array are uniformly shuffled, and all real keys contained in $\{k_1, k_2, \ldots, k_n\}$ are distinct.
- **The procedure:**
  - Initialize a list $\boldsymbol{P} \leftarrow \varnothing$.
  - Initialize the internal $\mathsf{state} := (\boldsymbol{I}, \boldsymbol{P})$.

Command $\mathcal{F}_{\mathsf{HT}}^{n}.\mathsf{Lookup}()$:

- **Input:** a key $k \in \{0,1\}^D \cup \{\perp\}$.
- **The procedure:**
  - Parse the internal state as $\mathsf{state} = (\boldsymbol{I}, \boldsymbol{P})$.
  - If $k \in \boldsymbol{P}$, set $v^* = \perp$.
  - If $k = \perp$ or $k \notin \boldsymbol{I}$, set $v^* = \perp$.
  - Otherwise, set $v^* = v$ where v is the value that corresponds to the key k in $\boldsymbol{I}'$.
  - Update $\boldsymbol{P} \leftarrow \boldsymbol{P} \cup \{(k, v)\}$.
- **Output:** The element v^*.

Command $\mathcal{F}_{\mathsf{HT}}^{n}.\mathsf{Extract}()$:

- **Input:** There is no input to this command.
- **The procedure:**
  - Parse the internal state $\mathsf{state} = (\boldsymbol{I}, \boldsymbol{P})$.
  - Define an array $\boldsymbol{J} = \{a'_1, a'_2, \ldots, a'_n\}$ from $\boldsymbol{I}$ as follows: For $i \in [n]$, set $a'_i = a_i$ if $a_i \notin \boldsymbol{P}$, otherwise set $a'_i = \mathsf{dummy}$.
  - Shuffle $\boldsymbol{J}$ uniformly.
- **Output:** The array $\boldsymbol{J}$.

Now, we present MalHT, a maliciously secure implementation of Functionality 8.1.

Algorithm 8.2 $\mathsf{MalHT.Build}()$: Hash table for shuffled inputs.

Input: An array $\boldsymbol{I} = (a_1, \ldots, a_n)$ containing n elements, where each a_i is either dummy or a (key, value) pair denoted (k_i, v_i) where both the key k and the value v are D-bit strings, where $D := O(1) \cdot w$.

Input assumptions: The elements in the array are uniformly shuffled, and all real keys contained in $\{k_1, k_2, \ldots, k_n\}$ are distinct.

Secret key: Sample a random PRF secret key sk. Use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\texttt{Enc}\text{”}||\cdot)$ for all ciphertexts, and use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\texttt{MAC}\text{”}||\cdot)$ for all MACs, where we now abuse notation and overload sk to denote the secret key for both.

Authenticated Encryption: Unless otherwise specified, for every write query $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$, data is replaced with $\mathsf{data}' := (\mathsf{ct} \leftarrow \mathsf{Enc}_{\mathsf{sk}}(\mathsf{data}), \sigma \leftarrow \mathsf{MAC}_{\mathsf{sk}}(\mathsf{ct}, \mathsf{addr}))$.

Reads are passed through authenticated decryption, namely unpacking $\mathsf{data}' = (\mathsf{ct}^*, \sigma^*)$, checking $\mathsf{MACVer}_{\mathsf{sk}}((\mathsf{ct}^*, \mathsf{addr}), \sigma^*) = 1$, aborting if verification fails, and otherwise returning $\mathsf{data}^* = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}^*)$.

Memory checking: To make the algorithm maliciously obliviously secure, we combine both online and offline memory checking components. All portions that are post-verifiably offline memory checked are boxed out in gray (corresponding to the Read/Write phase), and all portions of memory that are not online-checkable via time-stamping will be indicated with superscript off. When offline memory checking, after the Read/Write phase, for simplicity we use post-verifiability to immediately copy the contents of memory sequentially to fresh "online checkable" portions of memory with corresponding time-stamps. We time-stamp all other parts of this implementation.

The algorithm:

1. Let $\mu := \log^9 \lambda$, $\epsilon := \frac{1}{\log^2 \lambda}$, $\delta := e^{-\log \lambda \cdot \log\log \lambda}$, and $B := \lceil n/\mu \rceil$.
2. Initialize arrays $\mathsf{Bin}_1^{\mathsf{off}}, \mathsf{Bin}_2^{\mathsf{off}}, \ldots, \mathsf{Bin}_B^{\mathsf{off}}$ of size $\mu + 0.5 \cdot \epsilon\mu$ to $\varnothing$.
3. Initialize counters $c_1^{\mathsf{off}}, \ldots, c_B^{\mathsf{off}} := 0$.
4. *Balls in bins hashing to leave results in a post-verifiable state:*
   - For $i = 1, 2, \ldots, n$, throw real items $a_i = (k_i, v_i)$ into bin $\mathsf{Bin}_j^{\mathsf{off}}$ where $j = \mathsf{PRF}_{\mathsf{sk}}(0||k_i) \pmod B$. If $a_i = \mathsf{dummy}$, throw it into a uniformly random bin. Concretely:
     - Every bin $\mathsf{Bin}_j^{\mathsf{off}}$ has an associated counter c_j^{off}.
     - Retrieve counter c_j^{off}. If $c_j^{\mathsf{off}} > |\mathsf{Bin}_j^{\mathsf{off}}|$, output $\perp$.
     - Place a_i in position $\mathsf{Bin}_j^{\mathsf{off}}[c_j^{\mathsf{off}}]$.
     - Increment c_j^{off}.
   - Initialize lists $\mathsf{Bin}_1, \ldots, \mathsf{Bin}_B$ to $\varnothing$.
   - For $i = 1, \ldots, B$, using the post-verifiability of the offline memory checker, copy over the contents of $\mathsf{Bin}_i^{\mathsf{off}}$ into Bin_i sequentially with appropriate timestamps.
5. *Sampling secret loads*:
   - Sample $(L_1, \ldots, L_B) \leftarrow \mathsf{SampleBinLoads}_{B,\delta}(n')$, where $n' = n \cdot (1 - \epsilon)$.
   - For any $i \in [B]$, if $||\mathsf{Bin}_i| - \mu| > 0.5\epsilon\mu$ or $\left|L_i - \frac{n'}{B}\right| > 0.5\epsilon\mu$, output $\perp$.
6. *Creating major bins:*
   - Initialize new bins $\mathsf{Bin}'_1, \ldots, \mathsf{Bin}'_B$, each of size μ.
   - For each $1 \le i \le B$, iterate in parallel over all of Bin_i and Bin'_i, and copy over the first L_i elements from Bin_i to Bin'_i, and fill the remaining $\mu - L_i$ slots of Bin'_i with dummy.
7. *Creating overflow pile.*
   - Iterate over all of Bin_i, and replace the first L_i positions with dummy, and rewrite the unmodified contents.
   - Concatenate $X = \mathsf{Bin}_1||\mathsf{Bin}_2||\ldots||\mathsf{Bin}_B$.
   - Run $Y \leftarrow \mathsf{TightCompaction}(X)$ to move real elements to the front.
   - Truncate Y to length $2\epsilon \cdot n$.
8. *Prepare Cuckoo hash tables for efficient lookup:* For each $i = 1, 2, \ldots, B$:
   - Obtain the Cuckoo hashing initialized array $\boldsymbol{X}_i \leftarrow \mathsf{CuckooInit}(\mathsf{Bin}'_i)$ by calling the subroutine in Algorithm C.35 (note that this is not a hybrid call; we instead separate the pseudocode for readability). This subroutine intersperses dummies to pad the array to length $c_{\mathsf{cuckoo}} \cdot |\mathsf{Bin}_i| + \log \lambda$.

- Iterate over indices j in $\boldsymbol{X}_i$ and create the metadata array $\mathbf{MD}_{\boldsymbol{X}_i}$ and do the following:
  - If $\boldsymbol{X}_i[j] = (k_j, v_j)$ is a real element, let $\mathbf{MD}_{\boldsymbol{X}_i}[j] := (\mathsf{choice}_{1,j}, \mathsf{choice}_{2,j})$ where $(\mathsf{choice}_{1,j}, \mathsf{choice}_{2,j}) \leftarrow \mathsf{PRF}_{\mathsf{sk}}(i||k_j)$.
  - If $\boldsymbol{X}_i[j]$ is a dummy, let $\mathbf{MD}_{\boldsymbol{X}_i}[j] := (\perp, \perp)$.
- Call the subroutine $\mathbf{Assign}_{\boldsymbol{X}_i} \leftarrow \mathsf{CuckooMD}(\mathbf{MD}_{\boldsymbol{X}_i})$ (Algorithm C.37) to obtain the Cuckoo hashing assignment, where we use the *packed* $\mathsf{cuckooAssign}$ algorithm from Corollary C.34.
- Initialize $\mathsf{CBin}_i^{\mathsf{off}}$ to be an empty array of size $\mu \cdot c_{\mathsf{cuckoo}}$, and initialize $\mathsf{S}_i^{\mathsf{off}}$ to be the stash of size $\log \lambda$ associated with $\mathsf{CBin}_i^{\mathsf{off}}$.
- Route the elements of $\boldsymbol{X}_i$ according to $\mathbf{Assign}_{\boldsymbol{X}_i}$ to $\mathsf{CBin}_i^{\mathsf{off}} \cup \mathsf{S}_i^{\mathsf{off}}$ in the clear. More concretely, for $j = 1, 2, \ldots, |\boldsymbol{X}_i|$:
  - Let $(k, v) \leftarrow \boldsymbol{X}_i[j]$ (note that k may be a dummy).
  - Let $\mathsf{addr} \leftarrow \mathbf{Assign}_{\boldsymbol{X}_i}[j]$.
  - If $\mathsf{addr} \in \mathsf{CBin}_i^{\mathsf{off}}$, then write $\mathsf{CBin}_i^{\mathsf{off}}[\mathsf{addr}] := (k, v)$.
  - If $\mathsf{addr} \in \mathsf{S}_i^{\mathsf{off}}$, then write $\mathsf{S}_i^{\mathsf{off}}[\mathsf{addr}] := (k, v)$.
- Initialize CBin_i to be an empty array of size $\mu \cdot c_{\mathsf{cuckoo}}$, and initialize S_i to be the stash of size $\log \lambda$ associated with CBin_i.
- Using the post-verifiability of the offline memory checker, copy $\mathsf{CBin}_i^{\mathsf{off}} \cup \mathsf{S}_i^{\mathsf{off}}$ into $\mathsf{CBin}_i \cup \mathsf{S}_i$ sequentially with appropriate timestamps.

9. *Prepare Overflow Cuckoo hash table for efficient lookup:*
   - Run $\mathsf{cuckooAssign}(Y)$ (without packed sorting) from Theorem C.31 with parameter δ and $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{OF}\text{”}||\cdot)$ to obtain the table and stash $(\mathsf{OF}, \mathsf{OF}_{\mathsf{S}})$. Since $|Y| \leq 2\epsilon \cdot n = \frac{2n}{\log^2 \lambda}$, we have that the query and space complexity of this step is $O(|Y| \log |Y|) = O(n)$ since $n \leq \mathsf{poly}(\lambda)$.
10. *Prepare Stash Cuckoo hash table for efficient lookup:*
    - Let $\mathsf{S} = \bigcup_{i=1}^{B} \mathsf{S}_i$, i.e., the union of all stashes. Note that $|\mathsf{S}| \leq |B| \cdot \log \lambda \leq O\left(\frac{n}{\log^8 \lambda}\right)$.
    - Run $\mathsf{cuckooAssign}(\mathsf{S})$ (without packed sorting) from Theorem C.31 with parameter δ and $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{SecS}\text{”}||\cdot)$ to obtain the table and stash $(\mathsf{SecS}, \mathsf{SecS}_S)$. Since $|S| = O\left(\frac{n}{\log^8 \lambda}\right)$, we have that the query and space complexity of this step is $O(|S| \log |S|) = O(n)$.

Output: This command has no output.

State on Server:

- Bins
  - Main Cuckoo bins: $\mathsf{CBin}_1, \ldots, \mathsf{CBin}_B$.
  - Cuckoo hash table for overflow pile: OF.
  - Cuckoo hash table for combined stash: SecS.
- Two lists OF_{S} and $\mathsf{SecS}_{\mathsf{S}}$ to lookup elements in the leftover stash.
- Array $\boldsymbol{P}$ of size n initialized $\varnothing$. This array will be used to track all lookups made to $\mathsf{CBin}_1, \ldots, \mathsf{CBin}_B, \mathsf{OF}, \mathsf{SecS}$.

Local State:

- Secret key: sk.
- Counter c initialized to 0. This counter will be used to track the number of lookups made.

Algorithm 8.3 MalHT.Lookup(k)

State on Server:

- Bins: $(\mathsf{CBin}_1, \ldots, \mathsf{CBin}_B, \mathsf{OF}, \mathsf{SecS}, \mathsf{OF_S}, \mathsf{SecS_S})$.
- Array $\boldsymbol{P}$ of all lookups made.

Local State: Counter c and secret key sk.
Input: Key k to look for (which might be $\perp$ for dummy lookups).
Input Assumption: The key k has not been previously looked up, i.e., $k \notin \boldsymbol{P}$.
The algorithm:

1. Initialize $v := \perp$ and list $A_k \leftarrow \varnothing$.
2. If $k = \perp$:
   - Iterate linearly over $\mathsf{OF_S}$, and write every element back unmodified.
   - Iterate linearly over $\mathsf{SecS_S}$, and write every element back unmodified.
   - Lookup two random locations in OF, and add the locations to A_k.
   - Lookup two random locations in S, and add the locations to A_k.
   - Choose random bin $i \leftarrow [B]$, and lookup two random locations in CBin_i, and add the locations to A_k.
3. If $k \neq \perp$:
   - *Lookup in the stashes:*
     - Iterate linearly over $\mathsf{OF_S}$. If $\mathsf{OF_S}[j]$ contains k, write $\perp$ back at $\mathsf{OF_S}[j]$, and store the value of v. Otherwise, perform a dummy write to $\mathsf{OF_S}[j]$.
     - Iterate linearly over $\mathsf{SecS_S}$. If $\mathsf{SecS_S}[j]$ contains k, write $\perp$ back at $\mathsf{SecS_S}[j]$, and store the value of v. Otherwise, perform a dummy write to $\mathsf{SecS_S}[j]$.
     - Update the value of v.
   - *Lookup in overflow pile:*
     - If $v \neq \perp$ (i.e., v was found in the stashes), look up two random locations in OF.
     - If $v = \perp$, let $\mathsf{choice}_{1,\mathsf{OF}}, \mathsf{choice}_{2,\mathsf{OF}} \leftarrow \mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{OF}\text{”}||k)$. Lookup $\mathsf{OF}[\mathsf{choice}_{1,\mathsf{OF}}]$ and $\mathsf{OF}[\mathsf{choice}_{2,\mathsf{OF}}]$. If key k lies in either of them, set v to be the corresponding value. Add both addresses to A_k.
   - *Lookup in combined stash:*
     - If $v \neq \perp$, look up two random locations in SecS. Add the locations to A_k.
     - If $v = \perp$, compute $\mathsf{choice}_{1,\mathsf{SecS}}, \mathsf{choice}_{2,\mathsf{SecS}} \leftarrow \mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{SecS}\text{”}||k)$. Lookup $\mathsf{SecS}[\mathsf{choice}_{1,\mathsf{SecS}}]$ and $\mathsf{SecS}[\mathsf{choice}_{2,\mathsf{SecS}}]$. If key k lies in either of them, set v to be the corresponding value. Add both addresses to A_k.
   - *Lookup in bins:*
     - If $v \neq \perp$, choose a random bin $i \leftarrow [B]$ and look up two random locations in CBin_i. Add both addresses to A_k.
     - If $v = \perp$,
       - Compute $i \leftarrow \mathsf{PRF}_{\mathsf{sk}}(0||k)$.
       - Let $\mathsf{choice}_1, \mathsf{choice}_2 \leftarrow \mathsf{PRF}_{\mathsf{sk}}(i||k)$.
       - Lookup $\mathsf{CBin}_i[\mathsf{choice}_1]$ and $\mathsf{CBin}_i[\mathsf{choice}_2]$. If key k lies in either of them, set v to be the corresponding value. Add both addresses to A_k.

Output: The value v.

Final State on Server:

- Bins: $(\mathsf{CBin}_1, \ldots, \mathsf{CBin}_B, \mathsf{OF}, \mathsf{SecS})$. Note that these remain *unchanged*.
- Two lists $\mathsf{OF_S}$ and $\mathsf{SecS_S}$ to look up elements in the leftover stash.
- Update $\boldsymbol{P}[c] \leftarrow (k, A_k)$. Here, A_k is a list of all 6 addresses (two addresses in OF, two addresses in SecS, and two addresses in CBin_i for some i) that were looked up in the table.

Final Local State: Increment $c \leftarrow c + 1$.

Algorithm 8.4 MalHT.Extract()

State on Server:

- Bins: $(\mathsf{CBin}_1, \ldots, \mathsf{CBin}_B, \mathsf{OF}, \mathsf{SecS}, \mathsf{OF_S}, \mathsf{SecS_S})$.
- Array $\boldsymbol{P}$ of all lookups made.

Local State: Counter c and secret key sk.
Input: This command has no input.
The algorithm:

1. Let $L = \mathsf{CBin}_1||\mathsf{CBin}_2||\ldots||\mathsf{CBin}_B||\mathsf{OF}||\mathsf{SecS}$.
2. Initialize T^{off} to be an array of size $|L|$.
3. Copy the elements from L into T^{off}.
4. Iterate over $\boldsymbol{P}$, and sequentially do following:
   - If $\boldsymbol{P}[i] = (\perp, A_k)$, then access all addresses in A_k, and write back the contents without modifications.
   - If $\boldsymbol{P}[i] = (k, A_k)$ where $k \neq \perp$, access every $\mathsf{addr} \in A_k$:
     - If key k lies in $T^{\mathsf{off}}[\mathsf{addr}]$, rewrite the entry within T^{off} to $\perp$.
     - If key k does not lie in $T^{\mathsf{off}}[\mathsf{addr}]$, write back the contents within T^{off} without modifications.
5. Initialize an array T of size $|L|$ to $\varnothing$.
6. Using the post-verifiability of the offline memory checker, copy the contents of T^{off} into T sequentially with appropriate timestamps.
7. Let $C||\mathsf{OF}'||\mathsf{SecS}' := T$, where C, OF' and SecS' correspond to the marked versions of $\mathsf{CBin}_1, \cdots, \mathsf{CBin}_B$, OF and SecS respectively.
8. Let $R \leftarrow \mathsf{PerfectORP}(\mathsf{OF}'||\mathsf{SecS}')$.
9. Let $T \leftarrow \mathsf{Intersperse}(C||R; |C|, |R|)$.
10. Let $T_{\mathsf{S}} \leftarrow \mathsf{PerfectORP}(\mathsf{OF_S}||\mathsf{SecS_S})$.
11. Let $T' \leftarrow \mathsf{Intersperse}(T||T_{\mathsf{S}}; |T|, |T_{\mathsf{S}}|)$.
12. Let $T'' \leftarrow \mathsf{TightCompaction}(T')$ considering all entries overwritten with $\perp$ as 0-balls (i.e., moved to end of array), and truncate the array to size n.
13. Let $X \leftarrow \mathsf{IntersperseRD}_n(T'')$.

Output: The array X.
Final State on Server: Reset the state on the server to $\varnothing$.

Malicious security. To make the algorithm maliciously obliviously secure, we combine both online and offline memory checking components. All portions that are offline memory checked are boxed out in gray, and all portions of memory that are not online-checkable via time-stamping will be indicated with superscript off. All such portions of memory are left in a post-verifiable state at the end of the offline checking portions after Algorithm 4.10. We then immediately copy the contents of this memory sequentially to fresh "online checkable" portions of memory with corresponding time-stamps. All other parts of this implementation can be time-stamped in the hybrid model. To see this, note that in both $\mathsf{MalHT.Build}()$ and $\mathsf{MalHT.Extract}()$, all portions which are not offline-checked are linear scans over contiguous portions of memory. In $\mathsf{MalHT.Lookup}()$, all of $\mathsf{OF_S}$ and $\mathsf{SecS_S}$ is updated every time, and hence can be time-stamped with just the knowledge of the number of lookups c. Moreover, only $\boldsymbol{P}[c]$ is modified in $\boldsymbol{P}$. Therefore, $\boldsymbol{P}$ can also be time-stamped.

We highlight the main reasons why it is safe to offline-check various portions of the algorithm.

- Step 4 of $\mathsf{MalHT.Build}()$, as discussed in Section 2.2.2, does not seem time-stampable. Nonetheless, offline checking is safe because the values leaked are either of the form $\mathsf{PRF_{sk}}(0||k_i)$ (with no duplicates) or uniformly random.
- In Step 8 of $\mathsf{MalHT.Build}()$, we use the fact that $\mathbf{Assign}_{\boldsymbol{X}_i}$ is time-stamped and tamper-proof. Moreover, since the values of $\mathbf{Assign}_{\boldsymbol{X}_i}$ are safe to leak (because the input to $\mathsf{MalHT.Build}()$ is randomly shuffled), routing according to $\mathbf{Assign}_{\boldsymbol{X}_i}$ is *conditionally offline-safe*, as discussed in Section 2.3.
- Note that the array $\boldsymbol{P}$ was time-stamped as it was constructed during $\mathsf{MalHT.Lookup}()$ calls. Therefore, in Steps 3 and 4 of $\mathsf{MalHT.Extract}()$, we use the fact that $\boldsymbol{P}$ is time-stamped and safe to leak (since $\boldsymbol{P}$ contains known addresses) to once again argue that it is conditionally offline-safe. Note that time-stamping $\boldsymbol{P}$ was not directly possible in the CombHT construction of Asharov et al. [AKL+20] since accesses to the small bins were abstracted out with $\mathsf{SmallHT}$ calls.

Theorem 8.5. *Suppose* $\log^{11}\lambda \le n \le \mathsf{poly}(\lambda)$. MalHT *is a maliciously secure implementation of* $\mathcal{F}_{\mathsf{HT}}$. *Moreover, the run-time of the algorithm is* $O(n)$ *for* Build, $O(\log\lambda)$ *for* Lookup *and* $O(n)$ *for* $\mathsf{Extract}$.

Proof. We describe the simulator $\mathcal{S}$ for MalHT in the ($\mathsf{Intersperse}$, $\mathsf{IntersperseRD}$, $\mathsf{TightCompaction}$, $\mathsf{PerfectORP}$, $\mathsf{SampleBinLoads}$, $\mathsf{cuckooAssign}$, $\mathsf{packedcuckooAssign}$, $\mathcal{F}_{\mathsf{Sort}}$, $\mathcal{F}_{\mathsf{Placement}}$)-hybrid.

1. **Simulating Build:** Run the real algorithm on input $\boldsymbol{I}$ of only dummies and generates the initial state on the server.
2. **Simulating Lookup:** Run the real lookup algorithm on input $\perp$ with the simulator's state on the server.
3. **Simulating Extract:** Run the real algorithm on the simulator's state on the server.

We show that an adversary $\mathcal{A}$ cannot distinguish $\textsc{Real}(\mathcal{C}_{\mathsf{MalHT}}, \mathcal{A})$ and $\textsc{Ideal}(\mathcal{F}_{\mathsf{HT}}, \mathcal{S}, \mathcal{A})$.

Experiment Hybrid_0**:** The real view $\textsc{Real}(\mathcal{C}_{\mathsf{MalHT}}, \mathcal{A})$.

Experiment Hybrid_1**:** This experiment is the same as Hybrid_0, except every invocation of $\mathsf{PRF_{sk}}(\cdot)$ is replaced with a random oracle call $\mathcal{O}(\mathsf{sk}||\cdot)$.

Claim 8.6. $|\Pr[\mathsf{Hybrid}_1 = 1] - \Pr[\mathsf{Hybrid}_0 = 1]| \leq \mathsf{negl}(\lambda)$.

This holds by PRF security.

Experiment Hybrid_2: This experiment is the same as the previous hybrid, except we replace the online checker M with an idealized version M' which behaves identically to M, except ensures (with probability 1) that any data^* sent back by M is correct.

Claim 8.7. $|\Pr[\mathsf{Hybrid}_2 = 1] - \Pr[\mathsf{Hybrid}_1 = 1]| \leq \mathsf{negl}(\lambda)$.

This follows directly from the soundness guarantee of online memory checkers.

Experiment Hybrid_3: This experiment is the same as the previous hybrid, except we replace the post-verifiable offline checker M_{off} with an idealized version M'_{off} which behaves identically to M_{off} except has soundness 1 instead of $1 - \mathsf{negl}(\lambda)$.

Claim 8.8. $|\Pr[\mathsf{Hybrid}_3 = 1] - \Pr[\mathsf{Hybrid}_2 = 1]| \leq \mathsf{negl}(\lambda)$.

This follows directly from the soundness guarantee of the post-verifiable offline memory checker.

Experiment Hybrid_4: In this hybrid, we modify the client and augment it with additional space to check that all ciphertexts passing the authentication verification have been generated by the MAC before, aborting if this is not the case.

Claim 8.9. $|\Pr[\mathsf{Hybrid}_4 = 1] - \Pr[\mathsf{Hybrid}_3 = 1]| \leq \mathsf{negl}(\lambda)$.

This follows from the unforgeability of our MAC scheme.

Experiment Hybrid_5: In this hybrid, we replace decryption with a local database D. Specifically, for each ciphertext ct generated with underlying message m, we locally store $D[\mathsf{ct}] \leftarrow m$, and upon retrieving a ciphertext ct from the memory checker, instead of decrypting, we retrieve the message from the locally stored $D[\mathsf{ct}]$.

Claim 8.10. $|\Pr[\mathsf{Hybrid}_5 = 1] - \Pr[\mathsf{Hybrid}_4 = 1]| = 0$.

Proof. This follows by perfect correctness of our encryption scheme and the perfect unforgeability of the MAC. □

Experiment Hybrid_6: In this hybrid, we replace all ct_i corresponding to data_i with encryptions of 0 as sent to the memory checkers. However, we still read and write to the local database D as before using the true value of the messages (i.e., not all 0s).

Claim 8.11. $|\Pr[\mathsf{Hybrid}_6 = 1] - \Pr[\mathsf{Hybrid}_5 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. Suppose an adversary can distinguish the two views. We construct an adversary $\mathcal{A}'$ against the adaptive IND-CPA game, as defined in Section 3.2. Specifically, we will let $\mathcal{A}'$ be the client in Hybrid_5 and Hybrid_6 as follows:

- For each write operation $(\mathsf{write}, \mathsf{addr}_i, \mathsf{data}_i)$, the $\mathcal{A}'$ sends $m_0 = \mathsf{data}_i$ and $m_1 = 0$ to the adaptive IND-CPA challenger to get back a ciphertext ct_i. $\mathcal{A}'$ updates its dictionary $D[\mathsf{ct}_i] = \mathsf{data}_i$.

- For each read operation, upon receiving ct_i, we let $\mathsf{data} \leftarrow D[\mathsf{ct}]$ be the result of the decryption as given to $\mathcal{C}$ in Hybrid_5. Note that ct must exist in D by the assumption in Hybrid_5 that all ciphertexts that passed the authentication check have already been seen before.

The case $b = 0$ corresponds exactly to Hybrid_5 and the case $b = 1$ is exactly Hybrid_6. Therefore, an adversary that distinguishes the two hybrids in fact breaks the IND-CPA security of the encryption scheme. □

Experiment Hybrid_7: In this experiment, the outputs of the commands come from the true functionality instead of $\mathcal{C}$.

Claim 8.12. $\Pr[\mathsf{Hybrid}_7 = 1] = \Pr[\mathsf{Hybrid}_6 = 1]$.

Proof. Note that the view after $\mathsf{MalHT.Build}()$ is identical since $\mathsf{MalHT.Build}()$ has no output.

If $\mathsf{MalHT.Lookup}(k)$ is called, if $\mathcal{C}$ aborts before reaching the end of the algorithm, the view is identical. Otherwise, by invoking the perfect soundness of the online checker M' and the offline checker M'_{off}, we know that both $\mathsf{MalHT.Build}()$ and $\mathsf{MalHT.Lookup}()$ must have been executed honestly. Therefore, by the correctness of the algorithm, the output in fact corresponds to the value of key k.

If $\mathsf{MalHT.Extract}()$ is called, if $\mathcal{C}$ aborts before reaching the end of the algorithm, the view is identical. Otherwise, by the perfect soundness of the online checker M' and the offline checker M'_{off}, it suffices to consider the case where $\mathcal{A}$ honestly answers the queries of $\mathcal{C}$.

In particular, it suffices to argue that an honest-but-curious execution of the implementation results in an output which is a uniform permutation of the unvisited items that is independent of the (content-less) access pattern. This follows from combining the proofs of Claim C.5 and C.10 of [AKL+20]. □

Experiment Hybrid_8: The experiment is the same as the previous hybrid, except in $\mathsf{MalHT.Extract}()$, arrays L, OF_{S}, $\mathsf{SecS}_{\mathsf{S}}$ are all now replaced with arrays of dummies. Recall that all encryptions are of 0 and that the output crucially comes from the functionality rather than the implementation. Therefore, it suffices to argue that the distribution of the sequence of addresses accessed does not change.

Claim 8.13. $\Pr[\mathsf{Hybrid}_8 = 1] = \Pr[\mathsf{Hybrid}_7 = 1]$.

Proof. Note that the access pattern of $\mathsf{MalHT.Extract}()$ is determined entirely by the address values of $\boldsymbol{P}$. Note that by replacing the contents of L with dummies, T^{off} is also replaced with dummies. Clearly, if $\boldsymbol{P} = (\perp, A_k)$ or if $\boldsymbol{P} = (k, A_k)$ where k was not found, the accesses corresponding to $\boldsymbol{P}$ are identical. If $\boldsymbol{P} = (k, A_k)$ where k was found during $\mathsf{MalHT.Lookup}(k)$ at address addr, then the only difference is that the key k will not be found at $T^{\mathsf{off}}[\mathsf{addr}]$. Thus, the contents are still written back to the same addresses without modifications. Therefore, the access patterns in Hybrid_7 and Hybrid_8 are identical. □

Experiment Hybrid_9: The experiment is the same as the previous hybrid, except that we modify it so that in $\mathsf{MalHT.Lookup}()$, we replace every call $\mathsf{MalHT.Lookup}(k)$ for some real key k with a dummy lookup $\mathsf{MalHT.Lookup}(\perp)$.

Claim 8.14. $|\Pr[\mathsf{Hybrid}_9 = 1] - \Pr[\mathsf{Hybrid}_8 = 1]| \leq n \cdot e^{-\Omega(\log^5 \lambda)}$.

Proof. If $\mathcal{C}$ aborts during $\mathsf{MalHT.Build}()$, the view is identical. Suppose $\mathsf{MalHT.Build}()$ is executed to completion. Note that if $\mathcal{A}$ does not execute $\mathsf{Build}()$ honestly, the memory checkers M' and M'_{off} would have aborted because they have perfect soundness. Therefore, we may assume that $\mathsf{MalHT.Build}()$ was honestly implemented.

Since we showed in the previous claim that the access pattern in $\mathsf{MalHT.Extract}()$ is determined entirely by the access pattern in $\mathsf{MalHT.Lookup}()$, it suffices to consider the joint distribution of the access patterns in Build and Lookup in the two hybrids. This was shown to be close up to a factor of $n \cdot e^{-\Omega(\log^5 \lambda)}$ in Claim C.6 of Asharov et al. [AKL+20]. □

Experiment Hybrid_{10}: The experiment is the same as the previous hybrid, except the client $\mathcal{C}$ runs $\mathsf{MalHT.Build}()$ on input $\boldsymbol{I}$ consisting only dummies.

Claim 8.15. $\Pr[\mathsf{Hybrid}_{10} = 1] = \Pr[\mathsf{Hybrid}_9 = 1]$.

Proof. Suppose that $\mathcal{C}$ does not abort before Step 4 of $\mathsf{MalHT.Build}()$. Note that since we replaced $\mathsf{PRF}_{\mathsf{sk}}(0||\cdot)$ with $\mathcal{O}(\mathsf{sk}||0||\cdot)$, and because there are no duplicate keys in the input, the view of Hybrid_9 at Step 4 corresponds to placing balls into bins sampled using independent randomness. This is the exact view in Hybrid_{10}, and therefore, the views of Step 4 are identical in both hybrids.

After Step 4, until Step 8, none of the access patterns are determined by the contents of the bins. Hence, the view until Step 8 is identical.

In each iteration of Step 8 of $\mathsf{MalHT.Build}()$, until the offline memory checking portion, all access patterns are linear scans over portions of memory. In the offline memory checking portion, since $\mathbf{Assign}_{\boldsymbol{X}_i}$ is in an online-checkable portion of memory which can be verified with probability 1 by the idealized online checker M', the access pattern is determined entirely by $\mathbf{Assign}_{\boldsymbol{X}_i}$. Hence, since $\mathbf{Assign}_{\boldsymbol{X}_i}$ was obtained through an indiscriminate hashing scheme, this access pattern looks identical in both worlds. Steps 9 and 10 are simply hybrid calls and are hence secure.

Since the remainder of the algorithm once again simply consists of linear scans of portions of memory, the view of the adversary in both worlds is identical. □

Experiment Hybrid_{11}: This experiment is the same as the previous hybrid, except that the $\mathcal{C}$ does the following:

- Encrypt the true data_i (but still using dummy inputs) corresponding to the execution rather than only encrypting 0.
- Use the decryption algorithm rather than maintaining a dictionary of ciphertexts.
- Check the MAC verifications rather than keeping a list of all ciphertexts and MACs.
- Replace the idealized offline memory checker M'_{off} with the actual checker M_{off}.
- Replace the idealized online memory checker M' with the actual checker M.

- Replace every invocation of $\mathcal{O}(\mathsf{sk}||\cdot)$ with $\mathsf{PRF}_{\mathsf{sk}}(\cdot)$.

Claim 8.16. $|\Pr[\mathsf{Hybrid}_{11} = 1] - \Pr[\mathsf{Hybrid}_{10} = 1]| \leq \mathsf{negl}(\lambda)$.

Since this is simply unraveling Hybrids 0-6 in reverse, this claim follows by repeating the same arguments. Note that this view corresponds exactly to the view in $\textsc{Ideal}(\mathcal{F}_{\mathsf{HT}}, \mathcal{S}, \mathcal{A})$, thereby completing the proof.

Completeness. In the hybrid model, the probability of failure when interacting with an honest server comes only from Step 5 of Build. The probability of aborting is at most $n \cdot e^{-\Omega(\log^5 \lambda)} = \mathsf{negl}(\lambda)$ since $n \leq \mathsf{poly}(\lambda)$.

Efficiency. MalHT.Lookup() takes $O(\log \lambda)$ time since it involves two linear scans over $O(\log \lambda)$ size stashes, and six lookups. In Build,

- In Step 4, the algorithm takes $O(n)$, and the workspace that needs to be offline checked has length $\lceil \frac{n}{\mu} \rceil \cdot (\mu + 0.5\epsilon\mu) \leq 2n$. Therefore, the run-time of the offline check is $O(n)$.
- The subroutine $\mathsf{SampleBinLoads}_{B,\delta}(n')$ takes $O(B \cdot \log^5(1/\delta) \log\log(1/\delta)) \leq O(\lceil n/\log^9 \lambda \rceil \cdot \log^6 \lambda \cdot \log\log \lambda \rceil) = O(n/\log^2 \lambda)$ time by Claim C.22.
- In Step 8, first note that Algorithms C.35 and C.37 are implemented using packed sorting on blocks of size $O(\log\log \lambda)$ (because the PRF values are in the range $O(\log^9 \lambda)$) on $O(\log^9 \lambda)$ elements the run-time is linear in each bucket.
- In the offline checked portion of Step 8, the size of the work space is $O(\mu + \log \lambda)$, and hence the whole offline check can be done in $O(\mu + \log \lambda)$ query complexity per bin. Overall, this comes to $O(n)$ query complexity.
- In Step 9, the size of the overflow pile is $O(\epsilon n)$. Therefore, applying the parameters of Corollary C.34, we see that since we apply $\delta = e^{-\log \lambda \cdot \log\log \lambda}$, the algorithm is $(1 - \mathsf{negl}(\lambda))$-maliciously secure. Moreover, the query complexity and space complexity is $O(\epsilon n \cdot \log(\epsilon n)) = O(n)$ since $n \leq \mathsf{poly}(\lambda)$ and $\epsilon = \frac{1}{\log^2 \lambda}$.
- Similarly, in Step 10, the size of S is $O(n/\mu \cdot \log \lambda) = O(n/\log^8 \lambda)$. Therefore, applying the same argument as for the overflow pile, this query complexity and space complexity is bounded by $O(n)$.

In Extract, the query complexity is clearly linear in the size of L everywhere, except when we randomly shuffle to obtain R and T_{S}. However, since we are shuffling lists of size and $|R| = O(n/\log^2 n)$ and $|T_{\mathsf{S}}| = O(\log \lambda)$, these steps take at most $O(n)$ query complexity and space complexity. □

9 Maliciously Secure Optimal ORAM Construction

In this section, we put together our building blocks from Section 7 as well as our MalHT construction in Section 8 to obtain our final construction.

9.1 A Warm-Up Construction

As a warm-up, we first propose a less efficient maliciously secure ORAM which achieves $O(\log N \cdot \log \lambda + \mathsf{poly}(\log\log\lambda))$ overhead and argue that it is secure. Then, we argue how to modify this to reduce the overhead to $O(\log N + \mathsf{poly}(\log\log\lambda))$. We present our final ORAM construction in Algorithm D.3 in Appendix D.

The overall structure of the algorithm follows the hierarchical paradigm, with the main difference being that there are multiple copies of each level in order to de-amortize the reshuffling work. At a high level, there are "active layers" where lookups are conducted while the other layers are rebuilt.

Structure of construction. We set $\ell = \lceil 11 \log\log\lambda \rceil$ and $L = \lceil \log N \rceil$. Our Oblivious RAM will have the following data structures.

- Two $\mathcal{F}_{\mathsf{2KeyDict}}$ (as discussed in Theorem C.8) instances A_ℓ and B_ℓ with capacity $2^{\ell+1}$ elements.
- For each level $i \in \{\ell+1, \ldots, L\}$ contains four white-box $\mathcal{F}_{\mathsf{HT}}$ instances. We denote the levels as $(\mathsf{A}^{\mathsf{HF}}_{\ell+1}, \ldots, \mathsf{A}^{\mathsf{HF}}_{L})$, $(\mathsf{A}^{\mathsf{F}}_{\ell+1}, \ldots, \mathsf{A}^{\mathsf{F}}_{L})$, $(\mathsf{B}^{\mathsf{HF}}_{\ell+1}, \ldots, \mathsf{B}^{\mathsf{HF}}_{L})$ and $(\mathsf{B}^{\mathsf{F}}_{\ell+1}, \ldots, \mathsf{B}^{\mathsf{F}}_{L})$. Here, the superscripts$\mathsf{F}$ and HF are used to denote that the corresponding hash-tables are either full or half-full respectively. For more details, see Asharov et al. [AKLS21].
- Pointers $\mathsf{A}_{\ell+1}, \ldots, \mathsf{A}_L$ and $\mathsf{B}_{\ell+1}, \ldots, \mathsf{B}_L$, where each A_i points to either $\{\mathsf{A}^{\mathsf{HF}}_i, \mathsf{A}^{\mathsf{F}}_i, \mathtt{Null}\}$ and each B_i points to $\{\mathsf{B}^{\mathsf{HF}}_i, \mathsf{B}^{\mathsf{F}}_i, \mathtt{Null}\}$, where $\mathtt{Null}$ is a null pointer.
- A global counter ctr initialized to 0.

The algorithm will also maintain a running task list calls tasks Tasks. At the end of each access, it will perform part roughly $O(1)$ steps of computation for each task in the list.

Algorithm 9.1 A Less Efficient Oblivious RAM: $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$. This algorithm is directly adapted from Construction 5.2 of [AKLS21].

Input: $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
Initialization: Initialize $\mathsf{ctr} \leftarrow 0$, and initialize all data structures to be empty. Initialize 0 to keep track of the number of writes.
Secret key: Sample a random PRF secret key sk. Use $\mathsf{PRF}_{\mathsf{sk}}(\text{"}\mathtt{Enc}\text{"}||\cdot)$ for all encryptions, and use $\mathsf{PRF}_{\mathsf{sk}}(\text{"}\mathtt{MAC}\text{"}||\cdot)$ for all MACs.
Authenticated Encryption: For every write query $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$, data is replaced with $\mathsf{data}' := (\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathsf{data}), \sigma = \mathsf{MAC}_{\mathsf{sk}}(\mathsf{ct}, \mathsf{addr}))$. Reads are passed through authenticated decryption, namely unpacking $\mathsf{data}' = (\mathsf{ct}^*, \sigma^*)$, checking $\mathsf{MACVer}_{\mathsf{sk}}((\mathsf{ct}^*, \mathsf{addr}), \sigma^*) = 1$, aborting if verification fails, and otherwise returning $\mathsf{data}^* = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}^*)$.
Memory checking: Note that the algorithm only makes hybrid calls, and updates three values regularly: $\mathsf{fetched}, \mathsf{found}$ and data^*. Therefore, the client can simply locally store the variables. All other operations involve copying and writing memory from hybrid input and output tapes, which can be time-stamped.
The algorithm:
Lookup:

- Initialize $\mathsf{found} = \mathsf{false}$, $\mathsf{data}^* = \perp$.

- Perform $\mathsf{fetched} \leftarrow \mathsf{A}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
- If $\mathsf{fetched} \neq \perp$: then call $\mathsf{B}_\ell.\mathsf{PopKey}(\perp)$. Otherwise, $\mathsf{fetched} := \mathsf{B}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
- If $\mathsf{fetched} \neq \perp$, set $\mathsf{found} = \mathsf{true}$.
- For each $i \in \{\ell+1, \ldots, L\}$ in increasing order:
  - If A_i is `Null`, let the output of $\mathsf{A}_i.\mathsf{Lookup}()$ be $\perp$.
  - If B_i is `Null`, let the output of $\mathsf{B}_i.\mathsf{Lookup}()$ be $\perp$.
  - If $\mathsf{found} = \mathsf{false}$:
    - Set $\mathsf{fetched} := \mathsf{A}_i.\mathsf{Lookup}(\mathsf{addr})$.
    - If $\mathsf{fetched} \neq \perp$, then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
  - Else, perform $\mathsf{A}_i.\mathsf{Lookup}(\perp)$.
  - If $\mathsf{found} = \mathsf{false}$:
    - Set $\mathsf{fetched} := \mathsf{B}_i.\mathsf{Lookup}(\mathsf{addr})$.
    - If $\mathsf{fetched} \neq \perp$ then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.
  - Else, perform $\mathsf{B}_i.\mathsf{Lookup}(\perp)$.

Update:

- If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time addr is being accessed, set $\mathsf{data}^* = 0$.
- Let $(k, v) := (\mathsf{addr}, \mathsf{data}^*)$ if this is a read operation; else let $(k, v) := (\mathsf{addr}, \mathsf{data})$.
- Insert (k, v) into A_ℓ and B_ℓ using $\mathsf{Insert}(k, \mathsf{ctr} \pmod{2^{\ell+1}}, v)$.

Rebuild:

- Increment ctr by 1.
- For $i \in \{\ell+1, \ldots, L\}$:
  - If $\mathsf{ctr} \equiv 0 \mod 2^{i-2}$, then continue to 1 of the 4 following cases:

If $\mathsf{ctr} \equiv$	$0 \mod 2^i$	$2^{i-2} \mod 2^i$	$2 \cdot 2^{i-2} \mod 2^i$	$3 \cdot 2^{i-2} \mod 2^i$
Set $\mathsf{A}_i :=$	`Null`	$\mathsf{A}_i^{\mathsf{HF}}$	`Null`	$\mathsf{A}_i^{\mathsf{HF}}$
Set $\mathsf{B}_i :=$	$\mathsf{B}_i^{\mathsf{F}}$	$\mathsf{B}_i^{\mathsf{F}}$	$\mathsf{B}_i^{\mathsf{HF}}$	`Null`
Start	$\mathsf{RebuildHF}(\mathsf{A}_i^{\mathsf{HF}})$	$\mathsf{RebuildHF}(\mathsf{B}_i^{\mathsf{HF}})$	$\mathsf{RebuildF}(\mathsf{A}_i^{\mathsf{F}})$	$\mathsf{RebuildF}(\mathsf{B}_i^{\mathsf{F}})$

    Here, starting a task means that we will add the task to the list Tasks. Refer to Algorithms 9.2 and 9.3 for $\mathsf{RebuildF}$ and $\mathsf{RebuildHF}$ respectively.
- For every task $t \in \mathsf{Tasks}$, execute $t.\mathsf{eachEpoch}$ steps of the task.
- Return v.

Algorithm 9.2 Warm-up $\mathsf{RebuildF}(\mathsf{C}_i^{\mathsf{F}})$

Input: The task has input $\mathsf{C}_i^{\mathsf{F}} \in \{\mathsf{A}_i^{\mathsf{F}}, \mathsf{B}_i^{\mathsf{F}}\}$.
Property $\mathsf{eachEpoch}$: The total time allocated to this task is 2^{i-2}.

- If $i = \ell + 1$: Let $W \in O(2^{\ell+1} \cdot \mathsf{poly}(\log \log N))$ bound the work done by this rebuild. Set $\mathsf{eachEpoch} = W/2^{i-2}$.
- If $i > \ell+1$: Let $W \in O(2^i)$ bound the work done by this rebuild algorithm. Set $\mathsf{eachEpoch} = W/2^{i-2}$.

The algorithm:

- If $i = \ell + 1$:
  - Run $\mathsf{C}_{i-1}.\mathsf{PopTime}(0, 2^\ell - 1)$ repeatedly for 2^ℓ times. Call the output list X'.
  - Obtain $\mathsf{X} \leftarrow \mathsf{Shuffle}(\mathsf{X}')$.
- If $\ell + 2 \le i \le L$: Obtain $\mathsf{X} \leftarrow \mathsf{C}^{\mathsf{F}}_{i-1}.\mathsf{Extract}()$.
- Call $\mathsf{Z} \leftarrow \mathsf{Dedup}(\mathsf{X}, \mathsf{Y})$.
- Call $\mathsf{C}^{\mathsf{F}}_i.\mathsf{Build}(\mathsf{Z})$.

Algorithm 9.3 Warm-up $\mathsf{RebuildHF}(\mathsf{C}^{\mathsf{HF}}_i)$

Input: The task gets as input a table $\mathsf{C}^{\mathsf{HF}}_i \in \{\mathsf{A}^{\mathsf{HF}}_i, \mathsf{B}^{\mathsf{HF}}_i\}$, for some index $i \in \{\ell + 1, \ldots, L\}$.
Property $\mathsf{eachEpoch}$: The total time allocated to this task is 2^{i-2}.

- If $i = \ell + 1$: Let $W \in O(2^{\ell+1} \cdot \mathsf{poly}(\log \log N))$ bound the work done by this rebuild. Set $\mathsf{eachEpoch} = W/2^{i-2}$.
- If $i > \ell+1$: Let $W \in O(2^i)$ bound the work done by this rebuild algorithm. Set $\mathsf{eachEpoch} = W/2^{i-2}$.

The algorithm:

- If $i = L$:
  - Run $\mathsf{X} \leftarrow \mathsf{C}^{\mathsf{F}}_{L-1}.\mathsf{Extract}()$ and $\mathsf{Y} \leftarrow \mathsf{C}^{\mathsf{F}}_L.\mathsf{Extract}()$.
  - Let $\mathsf{Z} \leftarrow \mathsf{Dedup}(\mathsf{X}, \mathsf{Y})$.
  - Run $\mathsf{C}^{\mathsf{HF}}_L.\mathsf{Build}(\mathsf{Z})$.
- If $\ell + 1 \le i \le L - 1$:
  - If $i = \ell + 1$:
    - Run $\mathsf{C}_\ell.\mathsf{PopTime}(2^\ell, 2^{\ell-1})$ repeatedly for 2^ℓ iterations. Call the output list X'.
    - Obtain $\mathsf{X} \leftarrow \mathsf{Shuffle}(\mathsf{X}')$.
  - If $\ell + 2 \le i \le L - 1$:
    - Let $\mathsf{X} \leftarrow \mathsf{C}^{\mathsf{F}}_{i-1}.\mathsf{Extract}()$.
  - Initialize an array Y of 2^{i-1} dummies.
  - Obtain $\mathsf{Z} \leftarrow \mathsf{IntersperseRD}(\mathsf{X}, \mathsf{Y})$.
  - Call $\mathsf{C}^{\mathsf{HF}}_i.\mathsf{Build}(\mathsf{Z})$.

Theorem 9.4. *Let N be the capacity of the database, and let $\lambda \in \mathbb{N}$ be a security parameter such that $N \le \mathsf{poly}(\lambda)$. Then, Algorithm 9.1 is a maliciously secure oblivious implementation of $\mathcal{F}_{\mathsf{RAM}}$, and each* Access *has worst-case query complexity $O(\log N \cdot \log \lambda + \log^5 \log \lambda)$.*

Proof. Asharov et al. [AKLS21] show (in Theorem 5.3) that Access is an honest-but-curious oblivious implementation of $\mathcal{F}_{\mathsf{RAM}}$. Since the implementation is time-stampable in the ($\mathcal{F}_{\mathsf{2KeyDict}}$, $\mathcal{F}_{\mathsf{Dedup}}$, $\mathcal{F}_{\mathsf{Shuffle}}$, $\mathcal{F}_{\mathsf{HT}}$, Intersperse)-hybrid model, by Corollary 4.6, Access is in fact maliciously secure.[7]

Completeness. In the hybrid model, it is easy to see that the algorithm has perfect completeness.

[7]Technically, since we pause and resume hybrid computations, we have to modify the functionality descriptions to handle these pauses in a black-box way. This can be easily handled, but for simplicity, we ignore this technicality.

Efficiency. Each *lookup phase* makes two calls to $\mathcal{F}_{\mathsf{2KeyDict}}$ with $O(\log^{12}\lambda)$ elements. Therefore, by applying Theorem C.8, we see that this takes $O(\log^5 \log \lambda)$ time. Moreover, we make $O(\log N)$ lookups to MalHT hash tables A_i and B_i, and each lookup takes $O(1)+O(\log \lambda)$ time, where the $O(1)$ lookup comes from the lookups to Cuckoo hash tables, and the $O(\log \lambda)$ comes from scanning the corresponding $\mathsf{OF_S}$ and $\mathsf{SecS_S}$ stashes. Therefore, this gives us $O(\log N \cdot \log \lambda + \log^5 \log \lambda)$ time. Each *write back* phase takes $O(\log^5 \log \lambda)$ time since it includes two writes to the dictionaries. Finally, each task $t \in \mathsf{Tasks}$, $t.\mathsf{eachEpoch} = O(1)$ except for $\mathsf{RebuildF}(C_\ell^{\mathsf{F}})$, and $\mathsf{RebuildHF}(C_\ell^{\mathsf{HF}})$, which take $O(\log^5 \log \lambda)$ time (since each dictionary pop takes $O(\log^5 \log \lambda)$ time). Hence, the overall run-time is $O(\log N \cdot \log \lambda + \log^5 \log \lambda)$, as desired. □

9.2 Final Construction

Theorem 9.5 (Restatement of Theorem 1.3)**.** *Let N be the capacity of the database, and let $\lambda \in \mathbb{N}$ be a security parameter such that $N \leq \mathsf{poly}(\lambda)$. Then, for word size $w = \omega(\log \lambda)$, Algorithm D.3 is a maliciously secure oblivious implementation of $\mathcal{F}_{\mathsf{RAM}}^{N,w}$, and each Access has worst-case query complexity $O(\log N + \log^5 \log \lambda)$. The local space complexity is $O(1)$ (and one PRF key), and the server space complexity is $O(N)$. Moreover, if the client (but not the adversary) has access to a random oracle, this implementation is statistically secure, even against computationally unbounded adversaries.*

The construction and proof have been deferred to Appendix D.

Acknowledgments

We are extremely grateful to Vinod Vaikuntanathan for suggesting this problem to us, engaging in many insightful discussions, and giving detailed feedback on our manuscript. We thank Ran Canetti for helpful discussions about universal composability. We also thank Moni Naor for helpful discussions about memory checking. We thank Alexandra Henzinger for giving valuable feedback on the manuscript.

References

[ABC+07] Giuseppe Ateniese, Randal C. Burns, Reza Curtmola, Joseph Herring, Lea Kissner, Zachary N. J. Peterson, and Dawn Song. Provable data possession at untrusted stores. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 598–609. ACM Press, October 2007.

[AEK+17] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. Concerto: A high concurrency key-value store with integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 251–266, 2017.

[AFN+17] Ittai Abraham, Christopher W. Fletcher, Kartik Nayak, Benny Pinkas, and Ling Ren. Asymptotically tight bounds for composing ORAM with PIR. In Serge Fehr, editor,

PKC 2017, Part I, volume 10174 of *LNCS*, pages 91–120. Springer, Heidelberg, March 2017.

[AKL+20] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. OptORAMa: Optimal oblivious RAM. In Anne Canteaut and Yuval Ishai, editors, *EUROCRYPT 2020, Part II*, volume 12106 of *LNCS*, pages 403–432. Springer, Heidelberg, May 2020.

[AKLS21] Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, and Elaine Shi. Oblivious RAM with worst-case logarithmic overhead. In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 610–640, Virtual Event, August 2021. Springer, Heidelberg.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An $O(n \log n)$ sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, 1983.

[AKST14] Daniel Apon, Jonathan Katz, Elaine Shi, and Aishwarya Thiruvengadam. Verifiable oblivious storage. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 131–148. Springer, Heidelberg, March 2014.

[Bat68] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, Spring Joint Computer Conference*, AFIPS '68 (Spring), page 307–314, New York, NY, USA, 1968. Association for Computing Machinery.

[BEG+91] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991.

[BKP+14] Karl Bringmann, Fabian Kuhn, Konstantinos Panagiotou, Ueli Peter, and Henning Thomas. Internal DLA: Efficient simulation of a physical growth model - (extended abstract). In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *ICALP 2014, Part I*, volume 8572 of *LNCS*, pages 247–258. Springer, Heidelberg, July 2014.

[BMN17] Erik-Oliver Blass, Travis Mayberry, and Guevara Noubir. Multi-client oblivious RAM secure against malicious servers. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 686–707. Springer, Heidelberg, July 2017.

[BNP+15] Vincent Bindschaedler, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, and Yan Huang. Practicing oblivious access on cloud storage: the gap, the fallacy, and the new way forward. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 837–849. ACM Press, October 2015.

[Can20] Ran Canetti. Universally composable security. *J. ACM*, 67(5), September 2020.

[CCS17] T.-H. Hubert Chan, Kai-Min Chung, and Elaine Shi. On the depth of oblivious parallel RAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 567–597. Springer, Heidelberg, December 2017.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint.iacr.org/2016/086.

[CFLM20] Sherman S. M. Chow, Katharina Fech, Russell W. F. Lai, and Giulio Malavolta. Multi-client oblivious RAM with poly-logarithmic communication. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part II*, volume 12492 of *LNCS*, pages 160–190. Springer, Heidelberg, December 2020.

[CGLS17] T.-H. Hubert Chan, Yue Guo, Wei-Kai Lin, and Elaine Shi. Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In Tsuyoshi Takagi and Thomas Peyrin, editors, *ASIACRYPT 2017, Part I*, volume 10624 of *LNCS*, pages 660–690. Springer, Heidelberg, December 2017.

[CGPR15] David Cash, Paul Grubbs, Jason Perry, and Thomas Ristenpart. Leakage-abuse attacks against searchable encryption. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015*, pages 668–679. ACM Press, October 2015.

[CNS18] T.-H. Hubert Chan, Kartik Nayak, and Elaine Shi. Perfectly secure oblivious parallel RAM. In Amos Beimel and Stefan Dziembowski, editors, *TCC 2018, Part II*, volume 11240 of *LNCS*, pages 636–668. Springer, Heidelberg, November 2018.

[Con22] Graeme Connell. Technology deep dive: Building a faster oram layer for enclaves. https://signal.org/blog/building-faster-oram/, 2022.

[CSG+05] Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *2005 IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society Press, May 2005.

[DFD+21] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. Snoopy: Surpassing the scalability bottleneck of oblivious storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, pages 655–671, 2021.

[DMN11] Ivan Damgård, Sigurd Meldgaard, and Jesper Buus Nielsen. Perfectly secure oblivious RAM without random oracles. In Yuval Ishai, editor, *TCC 2011*, volume 6597 of *LNCS*, pages 144–163. Springer, Heidelberg, March 2011.

[DNRV09] Cynthia Dwork, Moni Naor, Guy N. Rothblum, and Vinod Vaikuntanathan. How efficient can memory checking be? In Omer Reingold, editor, *TCC 2009*, volume 5444 of *LNCS*, pages 503–520. Springer, Heidelberg, March 2009.

[DvF+16] Srinivas Devadas, Marten van Dijk, Christopher W. Fletcher, Ling Ren, Elaine Shi, and Daniel Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 145–174. Springer, Heidelberg, January 2016.

[FDD12] Christopher W Fletcher, Marten van Dijk, and Srinivas Devadas. A secure processor architecture for encrypted computation on untrusted programs. In *Proceedings of the seventh ACM workshop on Scalable trusted computing*, pages 3–8, 2012.

[FRK+15] Christopher W. Fletcher, Ling Ren, Albert Kwon, Marten van Dijk, and Srinivas Devadas. Freecursive oram: [nearly] free recursion and integrity verification for position-based oblivious ram. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, page 103–116, New York, NY, USA, 2015. Association for Computing Machinery.

[FWC+12] Martin Franz, Peter Williams, Bogdan Carbunar, Stefan Katzenbeisser, Andreas Peter, Radu Sion, and Miroslava Sotáková. Oblivious outsourced storage with delegation. In George Danezis, editor, *FC 2011*, volume 7035 of *LNCS*, pages 127–140. Springer, Heidelberg, February / March 2012.

[GGM86] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[GHJR15] Craig Gentry, Shai Halevi, Charanjit S. Jutla, and Mariana Raykova. Private database access with HE-over-ORAM architecture. In Tal Malkin, Vladimir Kolesnikov, Allison Bishop Lewko, and Michalis Polychronakis, editors, *ACNS 15*, volume 9092 of *LNCS*, pages 172–191. Springer, Heidelberg, June 2015.

[GKK+12] S. Dov Gordon, Jonathan Katz, Vladimir Kolesnikov, Fernando Krell, Tal Malkin, Mariana Raykova, and Yevgeniy Vahlis. Secure two-party computation in sublinear (amortized) time. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *ACM CCS 2012*, pages 513–524. ACM Press, October 2012.

[GM11] Michael T. Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *ICALP 2011, Part II*, volume 6756 of *LNCS*, pages 576–587. Springer, Heidelberg, July 2011.

[GO96] Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.

[Gol09] Oded Goldreich. *Foundations of cryptography: volume 2, basic applications*. Cambridge university press, 2009.

[GTS01] Michael T Goodrich, Roberto Tamassia, and Andrew Schwerin. Implementation of an authenticated dictionary with skip lists and commutative hashing. In *Proceedings DARPA Information Survivability Conference and Exposition II. DISCEX'01*, volume 2, pages 68–82. IEEE, 2001.

[HGY20] Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. MACAO: A maliciously-secure and client-efficient active ORAM framework. In *NDSS 2020*. The Internet Society, February 2020.

[HILL99] Johan Håstad, Russell Impagliazzo, Leonid A. Levin, and Michael Luby. A pseudorandom generator from any one-way function. *SIAM Journal on Computing*, 28(4):1364–1396, 1999.

[HJ06] W. Eric Hall and Charanjit S. Jutla. Parallelizable authentication trees. In Bart Preneel and Stafford Tavares, editors, *SAC 2005*, volume 3897 of *LNCS*, pages 95–109. Springer, Heidelberg, August 2006.

[IKK12] Mohammad Saiful Islam, Mehmet Kuzu, and Murat Kantarcioglu. Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In *NDSS 2012*. The Internet Society, February 2012.

[JK07] Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007.

[Kil92] Joe Kilian. A note on efficient zero-knowledge proofs and arguments (extended abstract). In *24th ACM STOC*, pages 723–732. ACM Press, May 1992.

[KL21] Ilan Komargodski and Wei-Kai Lin. A logarithmic lower bound for oblivious RAM (for all parameters). In Tal Malkin and Chris Peikert, editors, *CRYPTO 2021, Part IV*, volume 12828 of *LNCS*, pages 579–609, Virtual Event, August 2021. Springer, Heidelberg.

[KLO12] Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Yuval Rabani, editor, *23rd SODA*, pages 143–156. ACM-SIAM, January 2012.

[KMW10] Adam Kirsch, Michael Mitzenmacher, and Udi Wieder. More robust hashing: Cuckoo hashing with a stash. *SIAM Journal on Computing*, 39(4):1543–1561, 2010.

[KS14] Marcel Keller and Peter Scholl. Efficient, oblivious data structures for MPC. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part II*, volume 8874 of *LNCS*, pages 506–525. Springer, Heidelberg, December 2014.

[LN18] Kasper Green Larsen and Jesper Buus Nielsen. Yes, there is an oblivious RAM lower bound! In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part II*, volume 10992 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2018.

[LO13] Steve Lu and Rafail Ostrovsky. Distributed oblivious RAM for secure two-party computation. In Amit Sahai, editor, *TCC 2013*, volume 7785 of *LNCS*, pages 377–396. Springer, Heidelberg, March 2013.

[LWN+15] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. ObliVM: A programming framework for secure computation. In *2015 IEEE Symposium on Security and Privacy*, pages 359–376. IEEE Computer Society Press, May 2015.

[Mer90] Ralph C. Merkle. A certified digital signature. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 218–238. Springer, Heidelberg, August 1990.

[MMRS15] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358. IEEE Computer Society Press, May 2015.

[MMRS17] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Maliciously secure multi-client ORAM. In Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi, editors, *ACNS 17*, volume 10355 of *LNCS*, pages 645–664. Springer, Heidelberg, July 2017.

[MPC+18] Pratyush Mishra, Rishabh Poddar, Jerry Chen, Alessandro Chiesa, and Raluca Ada Popa. Oblix: An efficient oblivious search index. In *2018 IEEE Symposium on Security and Privacy*, pages 279–296. IEEE Computer Society Press, May 2018.

[NN98] Kobbi Nissim and Moni Naor. Certificate revocation and certificate update. In Aviel D. Rubin, editor, *USENIX Security 98*. USENIX Association, January 1998.

[NR09] Moni Naor and Guy N Rothblum. The complexity of online memory checking. *Journal of the ACM (JACM)*, 56(1):1–46, 2009.

[OR07] Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In Niels Provos, editor, *USENIX Security 2007*. USENIX Association, August 2007.

[OS97] Rafail Ostrovsky and Victor Shoup. Private information storage (extended abstract). In *29th ACM STOC*, pages 294–303. ACM Press, May 1997.

[PPRY18] Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. PanORAMa: Oblivious RAM with logarithmic overhead. In Mikkel Thorup, editor, *59th FOCS*, pages 871–882. IEEE Computer Society Press, October 2018.

[PR04] Rasmus Pagh and Flemming Friche Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, 2004.

[PT11] Charalampos Papamanthou and Roberto Tamassia. Optimal and parallel online memory checking. Cryptology ePrint Archive, Report 2011/102, 2011. `https://eprint.iacr.org/2011/102`.

[RFY+13] Ling Ren, Christopher W. Fletcher, Xiangyao Yu, Marten van Dijk, and Srinivas Devadas. Integrity verification for path oblivious-ram. In *2013 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6, 2013.

[SCSL11] Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with $O((\log N)^3)$ worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *ASIACRYPT 2011*, volume 7073 of *LNCS*, pages 197–214. Springer, Heidelberg, December 2011.

[SDS+18] Emil Stefanov, Marten Van Dijk, Elaine Shi, T.-H. Hubert Chan, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: An extremely simple oblivious ram protocol. *J. ACM*, 65(4), apr 2018.

[SS13] Emil Stefanov and Elaine Shi. Multi-cloud oblivious storage. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 247–258. ACM Press, November 2013.

[SW13] Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of Cryptology*, 26(3):442–483, July 2013.

[Tof14] Tomas Toft. A secure priority queue; or: On secure datastructures from multiparty computation. In Hyang-Sook Lee and Dong-Guk Han, editors, *ICISC 13*, volume 8565 of *LNCS*, pages 20–33. Springer, Heidelberg, November 2014.

[WHC+14] Xiao Shaun Wang, Yan Huang, T.-H. Hubert Chan, abhi shelat, and Elaine Shi. SCORAM: Oblivious RAM for secure computation. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014*, pages 191–202. ACM Press, November 2014.

[ZWR+16] Samee Zahur, Xiao Shaun Wang, Mariana Raykova, Adria Gascón, Jack Doerner, David Evans, and Jonathan Katz. Revisiting square-root ORAM: Efficient random access in multi-party computation. In *2016 IEEE Symposium on Security and Privacy*, pages 218–234. IEEE Computer Society Press, May 2016.

A Additional Preliminaries

A.1 Hybrid Model

To more formally state and prove our composition theorem that allows us to prove malicious security in a modular way, we define a hybrid model for our computations. Let $\mathcal{G} = (\mathcal{G}_1, \cdots, \mathcal{G}_\ell)$ be a sequence of reactive functionalities. In the $\mathcal{G}$-hybrid model, the client can perform computations assuming it has oracle access to each functionality $\mathcal{G}_i$ for $i \in [\ell]$. More formally, in the real world, we have the following model. Each query can now either be $(\mathsf{op}, \mathsf{addr}, \mathsf{data})$ as before, or a command and input for some $\mathcal{G}_i$, denoted $(i, \mathsf{cmd}_{\mathcal{G}_i}, x_{\mathcal{G}_i})$.

Experiment A.1 $\textsc{Real}(\mathcal{C}, \mathcal{A})^{\mathcal{G}}$ Hybrid.

$(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda\right)$
while $\mathsf{cmd} \neq \perp$ **do**
  $\boxed{\mathsf{out} \leftarrow \perp}$
  $\mathsf{data}^* \leftarrow \perp$
  $\boxed{\mathsf{out}_{\mathcal{G}} = (\mathsf{out}_{\mathcal{G}_i})_{i \in [\ell]} \leftarrow \{\perp\}^\ell}$
  while $\boxed{\mathsf{out} = \perp}$ **do**
    $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{out}) \leftarrow \mathcal{C}\left(1^\lambda, \mathsf{cmd}, x, \mathsf{data}^*, \mathsf{out}_{\mathcal{G}}\right)}$
    if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
    if $\boxed{\mathsf{query} = (i, \mathsf{cmd}_{\mathcal{G}_i}, x_{\mathcal{G}_i})}$ **then**
      $\mathcal{A}\left(1^\lambda, i, \mathsf{cmd}_{\mathcal{G}_i}\right)$
      $\boxed{\mathsf{out}_{\mathcal{G}_i} \leftarrow \mathcal{G}_i(\mathsf{cmd}_{\mathcal{G}_i}, x_{\mathcal{G}_i})}$
      $\mathsf{data}^* \leftarrow \perp$
    else
      $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
    end if
  end while
  $(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathsf{out}}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda\right)$

Experiment A.2 $\textsc{Ideal}(\mathcal{F}, \mathcal{S}, \mathcal{A})^{\mathcal{G}}$ Hybrid.

$(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda\right)$
while $\mathsf{cmd} \neq \perp$ **do**
  $\boxed{\mathsf{done} \leftarrow \mathsf{false}}$
  $\mathsf{data}^* \leftarrow \perp$
  while $\boxed{\mathsf{done} = \mathsf{false}}$ **do**
    $\boxed{(\mathsf{query}, \mathsf{flag}, \mathsf{done}) \leftarrow \mathcal{S}\left(1^\lambda, \mathsf{cmd}, \mathsf{data}^*\right)}$
    if $\mathsf{flag} = \mathsf{true}$ **then return** $b \leftarrow \mathcal{A}\left(1^\lambda\right)$
    if $\boxed{\mathsf{query} = (i, \mathsf{cmd}_{\mathcal{G}_i})}$ **then**
      $\mathcal{A}\left(1^\lambda, i, \mathsf{cmd}_{\mathcal{G}_i}\right)$
      $\mathsf{data}^* \leftarrow \perp$
    else
      $\mathsf{data}^* \leftarrow \mathcal{A}\left(1^\lambda, \mathsf{query}\right)$
    end if
  end while
  $(\mathsf{cmd}, x) \leftarrow \mathcal{A}\left(1^\lambda, \boxed{\mathcal{F}(\mathsf{cmd}, x)}\right)$
end while
return $b \leftarrow \mathcal{A}\left(1^\lambda\right)$

Hybrid input and output tapes. When in the hybrid model, we assume there are designated *hybrid input and output tapes* for each functionality $\mathcal{G}_i$ where $i \in [\ell]$, specified as follows:

- $\mathsf{H}^i_{\mathsf{in}}$ is a write-once, write-only RAM tape that $\mathcal{C}_{\mathcal{F}}$ can write $x_{\mathcal{G}_i}$ on. (By write-once, we mean that each address of the RAM tape is written to at most once.) The adversary $\mathcal{A}$ cannot see or modify the contents of this tape, but it can see the access pattern (i.e., the addresses written to by $\mathcal{C}_{\mathcal{F}}$) as they occur.
- $\mathsf{H}^i_{\mathsf{out}}$ is a read-only RAM tape on which $\mathcal{C}_{\mathcal{F}}$ can access $\mathsf{out}_{\mathcal{G}_i} = \mathcal{G}_i(\mathsf{cmd}_{\mathcal{G}_i}, x_{\mathcal{G}_i})$. The adversary $\mathcal{A}$ cannot see or modify the contents of this tape, but it can see the access pattern (i.e., the addresses read by $\mathcal{C}_{\mathcal{F}}$) as they occur.

None of these tapes count towards the local space complexity of $\mathcal{C}_{\mathcal{F}}$.

Hybrid input assumptions. Just as stated before, inputs to $\mathcal{F}$ can have input assumptions. Now, however, since implementations $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model generate inputs $x_{\mathcal{G}_i}$ to $\mathcal{G}_i$, it must be the case that each $\mathcal{X}_{\mathcal{G}_i}$'s input assumption is satisfied for $x_{\mathcal{G}_i}$.

Definition A.3. *For a reactive RAM functionality $\mathcal{F}$ and a sequence of RAM functionalities $\mathcal{G} = (\mathcal{G}_1, \cdots, \mathcal{G}_\ell)$ with input assumptions $\mathcal{X} = (\mathcal{X}_1, \cdots, \mathcal{X}_\ell)$, we say a RAM machine $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ is a $(1-\delta)$-*maliciously secure oblivious implementation of a reactive functionality $\mathcal{F}$ in the $\mathcal{G}$-hybrid model *if calls to $\mathcal{G}$ satisfy their respective input assumptions and the following conditions hold:*

1. **Obliviousness & Correctness**: *There is a (stateful) PPT simulator $\mathcal{S}$ such that for all (stateful) PPT $\mathcal{A}$, the adversary $\mathcal{A}$ distinguishes between the* $\textsc{Real}(\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}, \mathcal{A})^{\mathcal{G}}$ *(Experiment A.1) and* $\textsc{Ideal}\,(\mathcal{F}, \mathcal{S}, \mathcal{A})^{\mathcal{G}}$ *(Experiment A.2) experiments with advantage at most δ.*
2. **Completeness**: *For all (stateful) honest-but-curious PPT $\mathcal{A}$, with probability $1-\delta$, the client $\mathcal{C}_{\mathcal{F}}$ never aborts, i.e., never sets* flag *to* true *throughout the whole execution of the real experiment.*

With this definition in hand, we now show how to combine an implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model with an implementation of $\mathcal{G}$ to get an implementation of $\mathcal{F}$ in the plain model (i.e., no hybrid functionalities). At a high level, the construction is directly compositing the implementations, with the only modification being that the hybrid input and output tapes now live on the server under authenticated encryption instead of being separate tapes. This change is necessary because an implementation in the plain model (as opposed to the hybrid model) does not have hybrid tapes, and the implementation cannot afford to store the contents of the hybrid input and output tapes locally.

Theorem A.4 (Concurrent Composition)**.** *Assume there exist one-way functions. Let $\mathcal{G} = (\mathcal{G}_1, \cdots, \mathcal{G}_\ell)$ be a sequence of functionalities, where $\ell = \mathsf{poly}(\lambda)$. Suppose that $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ is a $(1-\delta)$-maliciously secure oblivious implementation of $\mathcal{F}$ in the $\mathcal{G}$-hybrid model, and suppose that for each $i \in [\ell]$, $\mathcal{C}_{\mathcal{G}_i}$ is a $(1-\delta_i)$-maliciously secure oblivious implementation of $\mathcal{G}_i$ (in the plain model). Then, there is a client $\mathcal{C}_{\mathcal{F}}$ that is a $(1 - \mathsf{negl}(\lambda) - \delta - \sum_i \delta_i)$-maliciously secure oblivious implementation of $\mathcal{F}$ (in the plain model) with the following properties:*

- *The query complexity of $\mathcal{C}_{\mathcal{F}}$ is the query complexity of $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$, where each (single) hybrid call to $(i, \mathsf{cmd}_{\mathcal{G}_i})$ is replaced with the query complexity of $\mathcal{C}_{\mathcal{G}_i}$ for $\mathsf{cmd}_{\mathcal{G}_i}$.*
- *The local space complexity of $\mathcal{C}_{\mathcal{F}}$ is the sum of the local space complexities of $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ and the sum of the local space complexities of $\mathcal{C}_{\mathcal{G}_i}$ over $i \in [\ell]$, as well as the size of ℓ PRF keys.*
- *The server space complexity of $\mathcal{C}_{\mathcal{F}}$ is the sum of the server space needed for $\mathcal{C}_{\mathcal{F}}$, the sum of the server space needed for $\mathcal{C}_{\mathcal{G}_i}$, and the sum of the size of the hybrid tapes $\mathsf{H}_{\mathsf{in}}^i$ and $\mathsf{H}_{\mathsf{out}}^i$.*

Remark A.5. *The need for one-way functions here is only for space efficiency, to get rid of $\mathsf{H}_{\mathsf{in}}, \mathsf{H}_{\mathsf{out}}$ as used in the $\mathcal{G}$-hybrid model. Specifically, in $\mathcal{C}_{\mathcal{F}}$, these tapes between $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ and $\mathcal{C}_{\mathcal{G}}$ are no longer an input or output tape for $\mathcal{F}$, so it would count towards the space complexity of $\mathcal{C}_{\mathcal{F}}$. Therefore, it*

will now live in public memory and will have contents encrypted with authentication. Besides this change, the composition theorem is fully syntactic, following the UC framework of Canetti [Can20] since our simulations are straight-line and universal.

Remark A.6. *As long as $\ell \leq \mathsf{poly}(\lambda)$, one can replace the ℓ PRF keys with 1 PRF key as long as for each $i \in [l]$ and t, it is possible to determine how many times hybrid $\mathcal{G}_i$ has been invoked up to time t. To see this, let $T(i,t)$ be this quantity. Then, for a single PRF key k, the ith hybrid at time t can use $\mathsf{PRF}_k(i||T(i,t)||\cdot)$ as its effective PRF to generate the ciphertexts and MACs.*

Proof of Theorem A.4. For simplicity, we give the proof for the case where $\ell = 1$ and abuse notation so that $\mathcal{G} = \mathcal{G}_1$ is the only hybrid functionality. We define $\mathcal{C}_\mathcal{F}$ by composing $\mathcal{C}_\mathcal{F}^\mathcal{G}$ and $\mathcal{C}_\mathcal{G}$ in the natural way, as follows. With the exception of the $\mathsf{H}_{\mathsf{in}}, \mathsf{H}_{\mathsf{out}}$ tapes, which we will describe later, $\mathcal{C}_\mathcal{F}$ begins by running $\mathcal{C}_\mathcal{F}^\mathcal{G}$ and replaces each hybrid $\mathsf{query} = (\mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$ with a stateful invocation of $\mathcal{C}_\mathcal{G}(1^\lambda, \mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$. If $\mathcal{C}_\mathcal{G}$ ever sets $\mathsf{flag} = \mathsf{true}$, $\mathcal{C}_\mathcal{F}$ also sets $\mathsf{flag} = \mathsf{true}$ (i.e., the whole client aborts). When $\mathcal{C}_\mathcal{G}$ sets $\mathsf{out}_\mathcal{G} \neq \bot$, it continues $\mathcal{C}_\mathcal{F}$ with $\mathsf{out}_\mathcal{G}$ (and the previous (cmd, x) for $\mathcal{C}_\mathcal{F}^\mathcal{G}$ unchanged).

Now we describe how $\mathsf{H}_{\mathsf{in}} = \mathsf{T}_{\mathsf{in}}^\mathcal{G}$ and $\mathsf{H}_{\mathsf{out}} = \mathsf{T}_{\mathsf{out}}^\mathcal{G}$ are handled, as we can no longer afford to have special tapes that are hidden from the adversary that do not count towards space complexity. Instead, we put H_{in} and $\mathsf{H}_{\mathsf{out}}$ as part of the public tape (so that $\mathcal{A}$ can view and modify it), but with authenticated encryption. We define some part of the public tape (held by $\mathcal{A}$) to correspond to each $\mathsf{addr} \in \mathsf{H}_{\mathsf{in}} \cup \mathsf{H}_{\mathsf{out}}$ in a natural way, denoted $\mathsf{addr}_\mathcal{A}$.

Replacing H_{in}. For each call to $\mathcal{G}$, the client $\mathcal{C}_\mathcal{F}$ locally stores and samples fresh $k \leftarrow \mathsf{Gen}(1^\lambda)$ and $k' \leftarrow \mathsf{MACGen}(1^\lambda)$. For each $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$ for $\mathsf{addr} \in \mathsf{H}_{\mathsf{in}}$ in this hybrid call from $\mathcal{C}_\mathcal{F}^\mathcal{G}$, the client $\mathcal{C}_\mathcal{F}$ computes $\mathsf{ct} \leftarrow \mathsf{Enc}_k(\mathsf{data})$ and $\sigma \leftarrow \mathsf{MAC}_{k'}(\mathsf{addr}, \mathsf{ct})$, and sends the query $(\mathsf{write}, \mathsf{addr}_\mathcal{A}, (\mathsf{ct}, \sigma))$ to $\mathcal{A}$ instead of the write call to H_{in}. Then, for each $(\mathsf{read}, \mathsf{addr})$ to $x_\mathcal{G}$ for $\mathsf{addr} \in \mathsf{T}_{\mathsf{in}}^\mathcal{G} = \mathsf{H}_{\mathsf{in}}$ from $\mathcal{C}_\mathcal{G}$, the client $\mathcal{C}_\mathcal{F}$ instead queries $(\mathsf{read}, \mathsf{addr}_\mathcal{A})$ to $\mathcal{A}$, and on receiving $\mathsf{data}^* = (\mathsf{ct}, \sigma)$, the client $\mathcal{C}_\mathcal{F}$ first checks $\mathsf{MACVer}_{k'}((\mathsf{addr}, \mathsf{ct}), \sigma) = 1$ (setting $\mathsf{flag} = \mathsf{true}$ to abort if not) and gives back $\mathsf{data} \leftarrow \mathsf{Dec}_k(\mathsf{ct})$ to $\mathcal{C}_\mathcal{G}$ as the result of the read.

Replacing $\mathsf{H}_{\mathsf{out}}$. For each $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$ for $\mathsf{addr} \in \mathsf{T}_{\mathsf{out}}^\mathcal{G} = \mathsf{H}_{\mathsf{out}}$ to generate $\mathsf{out}_\mathcal{G}$, the client $\mathcal{C}_\mathcal{F}$ computes $\mathsf{ct} \leftarrow \mathsf{Enc}_k(\mathsf{data})$ and $\sigma \leftarrow \mathsf{MAC}_{k'}(\mathsf{addr}, \mathsf{ct})$, and sends the query $(\mathsf{write}, \mathsf{addr}_\mathcal{A}, (\mathsf{ct}, \sigma))$ to $\mathcal{A}$ instead of the write call to $\mathsf{H}_{\mathsf{out}}$. Then, for each $(\mathsf{read}, \mathsf{addr})$ to $\mathsf{out}_\mathcal{G}$ for $\mathsf{addr} \in \mathsf{H}_{\mathsf{out}}$ from $\mathcal{C}_\mathcal{F}^\mathcal{G}$, the client $\mathcal{C}_\mathcal{F}$ instead queries $(\mathsf{read}, \mathsf{addr}_\mathcal{A})$ to $\mathcal{A}$, and on receiving $\mathsf{data}^* = (\mathsf{ct}, \sigma)$, the client $\mathcal{C}_\mathcal{F}$ first checks $\mathsf{MACVer}_{k'}((\mathsf{addr}, \mathsf{ct}), \sigma) = 1$ (setting $\mathsf{flag} = \mathsf{true}$ to abort if not) and gives back $\mathsf{data} \leftarrow \mathsf{Dec}_k(\mathsf{ct})$ to $\mathcal{C}_\mathcal{F}$ as the result of the read.

This completes the description of $\mathcal{C}_\mathcal{F}$. The description of the universal simulator $\mathcal{S}_\mathcal{F}$, in terms of universal simulators $\mathcal{S}_\mathcal{F}^\mathcal{G}$ and $\mathcal{S}_\mathcal{G}$, is very similar to the description of $\mathcal{C}_\mathcal{F}$ in terms of $\mathcal{C}_\mathcal{F}^\mathcal{G}$ and $\mathcal{C}_\mathcal{G}$. The only meaningful difference is for accesses to $\mathsf{H}_{\mathsf{in}}, \mathsf{H}_{\mathsf{out}}$. Each simulated $(\mathsf{write}, \mathsf{addr})$ to $\mathsf{addr} \in \mathsf{H}_{\mathsf{in}} \cup \mathsf{H}_{\mathsf{out}}$ (by $\mathcal{S}_\mathcal{F}^\mathcal{G}$ or $\mathcal{S}_\mathcal{G}$) is now given by sampling $\mathsf{ct} \leftarrow \mathsf{Enc}_k(0)$, $\sigma \leftarrow \mathsf{MAC}_{k'}(\mathsf{addr}, \mathsf{ct})$, and the resulting query is $(\mathsf{write}, \mathsf{addr}_\mathcal{A}, (\mathsf{ct}, \sigma))$, where keys k, k' are sampled just like in $\mathcal{C}_\mathcal{F}$. Reading from $\mathsf{H}_{\mathsf{in}}, \mathsf{H}_{\mathsf{out}}$ is identical to $\mathcal{C}_\mathcal{F}$, except that the resulting data is not needed to keep running the simulators.

Now, we prove that $\mathcal{C}_\mathcal{F}$ is a maliciously secure oblivious implementation of $\mathcal{F}$. To see completeness, observe that $\mathcal{C}_\mathcal{F}^\mathcal{G}$ and $\mathcal{C}_\mathcal{G}$ are $(1-\delta)$- and $(1-\delta_1)$-complete respectively, and since decryption and

authentication have perfect correctness, $(1-\delta-\delta_1)$-completeness of the whole implementation follows.

To see obliviousness and correctness, we argue indistinguishability between the real and ideal worlds through following hybrids.

Experiment Hybrid_0: $\textsc{Real}(\mathcal{C}_\mathcal{F}, \mathcal{A})$, the real experiment.

Experiment Hybrid_1: In this hybrid, we assume no forgeries occur (say, by overwriting the responses from $\mathcal{A}$ to never forge). Note that since tapes $\mathsf{H}_\mathsf{in}, \mathsf{H}_\mathsf{out}$ were write-once, and since addresses were included in the MAC, the adversary now cannot have changed any ciphertexts on the hybrid tapes without a client abort.

Claim A.7. $|\Pr[\mathsf{Hybrid}_1 = 1] - \Pr[\mathsf{Hybrid}_0 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This directly follows from unforgeability of MACs. □

Experiment Hybrid_2: In this hybrid, we replace $\mathcal{C}_\mathcal{G}$ with $\mathcal{S}_\mathcal{G}$ and the corresponding outputs of $\mathcal{C}_\mathcal{G}$ with $\mathcal{G}(\mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$. At this point, the contents of the H_in tape are not used since we are using the simulator $\mathcal{S}_\mathcal{G}$ instead of $\mathcal{C}_\mathcal{G}$, and the contents written to H_out are now generated directly from $\mathcal{G}(\mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$ instead of $\mathcal{C}_\mathcal{G}$.

Claim A.8. $|\Pr[\mathsf{Hybrid}_2 = 1] - \Pr[\mathsf{Hybrid}_1 = 1]| \leq \delta_1$.

Proof. We do this via a reduction to security of $\mathcal{C}_\mathcal{G}$. Suppose there exists some adversary $\mathcal{A}$ distinguishing this and the previous hybrid. Now, we can construct an adversary $\mathcal{A}'$ breaking security of $\mathcal{C}_\mathcal{G}$. Specifically, adversary $\mathcal{A}'$ will run both $\mathcal{A}$ and $\mathcal{C}_\mathcal{F}^\mathcal{G}$, where the input tape $\mathsf{T}_\mathsf{in}^\mathcal{G}$ and output tape $\mathsf{T}_\mathsf{out}^\mathcal{G}$ of $\mathcal{C}_\mathcal{G}$ are given by decrypting the version of $\mathsf{H}_\mathsf{in}, \mathsf{H}_\mathsf{out}$ in the public tape. Since $\mathcal{C}_\mathcal{F}^\mathcal{G}$ is write-once to H_in and $\mathcal{A}$ never modifies H_in, the input tape to $\mathcal{C}_\mathcal{G}$ is unmodified by $\mathcal{A}'$ after the initial $x_\mathcal{G}$ is written to it. Similarly, since $\mathcal{C}_\mathcal{F}^\mathcal{G}$ has read-only access to H_out and $\mathcal{A}$ never modifies H_out, the output tape to $\mathcal{C}_\mathcal{G}$ is never modified by $\mathcal{A}'$. Furthermore, since $\mathcal{C}_\mathcal{F}^\mathcal{G}$ must satisfy input assumption $\mathcal{X}_\mathcal{G}$, we know the input assumption for $\mathcal{C}_\mathcal{G}$ is satisfied. Therefore, $\mathcal{A}'$ is a well-formed adversary that distinguishes between $\textsc{Real}(\mathcal{C}_\mathcal{G}, \mathcal{A}')$ and $\textsc{Ideal}(\mathcal{G}, \mathcal{S}_\mathcal{G}, \mathcal{A}')$, completing the reduction. □

Experiment Hybrid_3: In this hybrid, the client $\mathcal{C}_\mathcal{F}$ now stores $x_\mathcal{G}$ and corresponding output $\mathcal{G}(\mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$ locally (in addition to being encrypted with authentication on the hybrid tapes). While $\mathcal{C}_\mathcal{F}$ still makes the identical read and write queries to the hybrid tapes as before, the decrypted data returned from any reads to H_out is ignored, and instead, the data of the corresponding address is read locally from the local version stored by the client.

Claim A.9. $\Pr[\mathsf{Hybrid}_3 = 1] = \Pr[\mathsf{Hybrid}_2 = 1]$.

Proof. By perfect unforgeability of the MAC from an earlier hybrid and the write-once property on the hybrid output tape, the versions of $\mathcal{G}(\mathsf{cmd}_\mathcal{G}, x_\mathcal{G})$ stored on H_out and locally will always be identical (if no abort has already occurred). □

Experiment Hybrid_4: In this hybrid, we replace each ciphertext given to $\mathcal{A}$ on either of the hybrid tapes with a fresh encryption of 0. Note that due to the previous hybrids, none of the values on the hybrid tapes are used.

Claim A.10. $|\Pr[\mathsf{Hybrid}_4 = 1] - \Pr[\mathsf{Hybrid}_3 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This directly follows from adaptive IND-CPA security, as none of the program logic in these hybrids depends on any of the contents that are encrypted. In particular, decryption is not necessary as the output $\mathcal{G}(\mathsf{cmd}_{\mathcal{G}}, x_{\mathcal{G}})$ is stored locally in plaintext. □

Experiment Hybrid_5: Now, we swap $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ with $\mathcal{S}_{\mathcal{F}}^{\mathcal{G}}$ and corresponding outputs of $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ with $\mathcal{F}(\mathsf{cmd}, x)$. We no longer locally store $x_{\mathcal{G}}$ or $\mathcal{G}(\mathsf{cmd}_{\mathcal{G}}, x_{\mathcal{G}})$ because they are not generated or necessary for the simulators $\mathcal{S}_{\mathcal{F}}^{\mathcal{G}}$ or $\mathcal{S}_{\mathcal{G}}$.

Claim A.11. $|\Pr[\mathsf{Hybrid}_5 = 1] - \Pr[\mathsf{Hybrid}_4 = 1]| \leq \delta$.

Proof. We do this via a reduction to security of $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$. Suppose there exists some adversary $\mathcal{A}$ distinguishing this and the previous hybrid. Then, we can construct an adversary $\mathcal{A}'$ breaking security of $\mathcal{C}_{\mathcal{F}}^{\mathcal{G}}$ (in the $\mathcal{G}$-hybrid model). Specifically, the adversary $\mathcal{A}'$ will combine $\mathcal{A}$ and $\mathcal{S}_{\mathcal{G}}$, and whenever $\mathcal{A}$ needs a value from H_{in} or $\mathsf{H}_{\mathsf{out}}$, it can generate it on its own by encrypting 0 and authenticating it. Since the content-less accesses of H_{in} and $\mathsf{H}_{\mathsf{out}}$ are visible to $\mathcal{A}'$ in the security game, $\mathcal{A}'$ can successfully run $\mathcal{A}$ internally. Moreover, $\mathcal{A}'$ cannot change any contents of the ciphertexts since we assume no MAC forgeries happen. □

Experiment Hybrid_6: Lastly, we can assume that forgeries can happen once again. Moreover, this is exactly the ideal model $\textsc{Ideal}(\mathcal{F}, \mathcal{S}_{\mathcal{F}}, \mathcal{A})$.

Claim A.12. $|\Pr[\mathsf{Hybrid}_6 = 1] - \Pr[\mathsf{Hybrid}_5 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This is indistinguishable from the previous hybrid by unforgeability of the MAC. □

□

B Deferred proofs from Section 4

Proof of Lemma 4.4. Let $(\mathsf{MACGen}, \mathsf{MAC}, \mathsf{MACVer})$ be a MAC scheme. Before receiving any queries, the checker M initializes a counter $\mathsf{count} \leftarrow 0$ and generates $k \leftarrow \mathsf{MACGen}(1^\lambda)$. After each query from $\mathcal{C}$, M increments count by 1.

Write Queries: For each $\mathsf{query} = (\mathsf{write}, \mathsf{addr}, \mathsf{data})$ from $\mathcal{C}$, M sends the modified $\mathsf{query}' = (\mathsf{write}, \mathsf{addr}, \mathsf{data}')$ to the adversary $\mathcal{A}$, where

$$\begin{aligned} \mathsf{tag} &\leftarrow \mathsf{MAC}_k(\mathsf{addr}, \mathsf{data}, TS(\mathsf{count}, \mathsf{addr})), \\ \mathsf{data}' &= (\mathsf{data}, \mathsf{tag})\,. \end{aligned}$$

Read Queries: For each $\mathsf{query} = (\mathsf{read}, \mathsf{addr}, \perp)$ from $\mathcal{C}$, M sends the query as is to $\mathcal{A}$, and on response $\mathsf{data}' = (\mathsf{data}^*, \mathsf{tag}^*)$ from $\mathcal{A}$, the checker M ensures

$$\mathsf{MACVer}_k(\mathsf{tag}^*, (\mathsf{addr}, \mathsf{data}^*, TS(\mathsf{count}, \mathsf{addr}))) = 1,$$

and if so, passes on data^* to $\mathcal{C}$. If the verification fails, M sends $\perp$ to $\mathcal{C}$ and aborts.

Clearly, M has worst-case query complexity 1, and M can use one word to store count so its local space complexity is $O(1)$ (ignoring the PRF key).

We now show that M satisfies the completeness and soundness conditions as required by Definition 4.1. Completeness is as follows. By perfect correctness of our MAC scheme, it suffices to show that $TS(\mathsf{count}_{old}, \mathsf{addr}) = TS(\mathsf{count}_{new}, \mathsf{addr})$, where count_{old} was the time of the most recent write to addr up until time count_{new}. This holds by the definition of TS being a time-stamp function, as no writes to addr occur between count_{old} and count_{new}.

Soundness follows from unforgeability of the MAC scheme and properties of TS. Specifically, if the adversary ever forces the memory checker to output some incorrect data^* at some addr, it must be the case that $\mathsf{data}' = (\mathsf{data}^*, \mathsf{tag}^*)$ from $\mathcal{A}$ satisfies

$$\mathsf{MACVer}_k(\mathsf{tag}^*, (\mathsf{addr}, \mathsf{data}^*, TS(\mathsf{count}, \mathsf{addr}))) = 1$$

where $\mathsf{data}^* \neq \mathsf{data}$ (where data is correct). Then, $(\mathsf{tag}^*, (\mathsf{addr}, \mathsf{data}^*, TS(\mathsf{count}, \mathsf{addr})))$ is a forgery, as $TS(\cdot, \mathsf{addr})$ increments by 1 for each write to addr, so we can construct a MAC adversary where the only oracle query of the form $\mathsf{MAC}_k(\mathsf{addr}, z, TS(\mathsf{count}, \mathsf{addr}))$ (for the fixed count and addr but varying z) is for the unique value $z = \mathsf{data} \neq \mathsf{data}^*$. Therefore, by MAC unforgeability, soundness holds. □

Proof of Theorem 4.5. We define $\mathcal{C}'$ to be the composition of M and $\mathcal{C}$. That is, for each non-hybrid query that $\mathcal{C}$ generates, it is sent to M, and the resulting q_M queries generated from M will be sent to $\mathcal{A}'$, the adversarial server interacting with $\mathcal{C}'$. (For simplicity of notation, we assume that M always makes exactly q_M queries to the untrusted memory per client request.) For each response from $\mathcal{A}'$, M will either abort or not. If M aborts, $\mathcal{C}'$ will set $\mathsf{flag} = \mathsf{true}$ and immediately abort, but if M instead sends back some data^* (which must be correct with high probability by soundness of online memory checking), then data^* is sent back to $\mathcal{C}$. All hybrid reads and writes in $\mathcal{C}'$ will be the same as in $\mathcal{C}$ as hybrid tapes cannot be modified by the adversary. That is, the memory checker does not modify any hybrid calls.

First, it is easy to see that the query complexity of $\mathcal{C}'$ is at most $q \cdot q_M$, and it is also immediate to see that the local space complexity of $\mathcal{C}'$ will have an additive client space overhead of c_M.

Next, we show completeness of $\mathcal{C}'$ (in the sense of Definition 3.11). By the completeness condition of M (in the sense of Definition 4.1), for honest $\mathcal{A}'$, it follows that M never aborts and always sends $\mathcal{C}$ the correct word data^* back to $\mathcal{C}$ with $1 - \mathsf{negl}(\lambda)$ probability. Now, by a simple hybrid argument, we can invoke the completeness of $\mathcal{C}$ (in the sense of Definition 3.13), as the effective adversary $\mathcal{A}$ (namely the composition of $\mathcal{A}'$ and M) is honest-but-curious, so we know that $\mathcal{C}$ will only abort with $\mathsf{negl}(\lambda)$ probability. Therefore, $\mathcal{C}'$ only aborts with $\mathsf{negl}(\lambda)$ probability against an honest-but-curious $\mathcal{A}'$.

Finally, we argue obliviousness and correctness of $\mathcal{C}'$ by a hybrid argument as follows.

Experiment Hybrid_0: The real world $\textsc{Real}(\mathcal{C}', \mathcal{A}')^{\mathcal{G}}$.

Experiment Hybrid_1: In this hybrid, we replace the online memory checker M with an idealized version M' which behaves identically to M, except ensures with probability 1 (instead of $1 - \mathsf{negl}(\lambda)$)

that any data^* sent back to $\mathcal{C}$ by M is correct. (Note that such an M' exists with run-time and space at most $\mathsf{poly}(N) \leq \mathsf{poly}(\lambda)$.)

Claim B.1. $|\Pr[\mathsf{Hybrid}_1 = 1] - \Pr[\mathsf{Hybrid}_0 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. This directly follows from the soundness of M. That is, any $\mathsf{poly}(\lambda)$-time adversary distinguishing Hybrid_1 and Hybrid_0 can be used to break soundness of M with the same advantage. □

Experiment Hybrid_2: In this hybrid, we replace $\mathcal{C}$ with $\mathcal{S}$ and also replace any out generated by $\mathcal{C}$ with $\mathcal{F}(\mathsf{cmd}, x)$. Note that this implicitly defines a universal simulator $\mathcal{S}'$ for $\mathcal{C}'$, making this experiment identical to $\textsc{Ideal}(\mathcal{F}, \mathcal{S}', \mathcal{A}')^{\mathcal{G}}$.

Claim B.2. $|\Pr[\mathsf{Hybrid}_2 = 1] - \Pr[\mathsf{Hybrid}_1 = 1]| \leq \delta$.

Proof. We prove this by reduction to honest-but-curious obliviousness and correctness of $\mathcal{C}$ with simulator $\mathcal{S}$. Specifically, suppose $\mathsf{poly}(\lambda)$-time $\mathcal{A}'$ distinguishes between Hybrid_2 and Hybrid_1. Then, combining $\mathcal{A}'$ with our idealized memory checker M' gives a $\mathsf{poly}(\lambda')$ adversary against $\mathcal{C}$ and $\mathcal{S}$.

Let $\mathcal{A}$ be the composition of $\mathcal{A}'$ and M'. Since the previous hybrid uses an idealized memory checker M', we know that $\mathcal{A}$ always either aborts on its own or gives the correct answer to $\mathcal{C}$ or $\mathcal{S}$. Therefore, $\mathcal{A}$ is a (possibly aborting) honest-but-curious adversary against $\mathcal{C}$ and $\mathcal{S}$.[8] In other words, $\mathsf{Hybrid}_1 = \textsc{Real}(\mathcal{C}, \mathcal{A})^{\mathcal{G}}$ and $\mathsf{Hybrid}_2 = \textsc{Ideal}(\mathcal{S}, \mathcal{F}, \mathcal{A})^{\mathcal{G}}$. Therefore, the claim follows by honest-but-curious obliviousness and correctness of $\mathcal{C}$ with simulator $\mathcal{S}$. □

□

C Maliciously Secure Building Blocks

In this section, we describe the building blocks that we use to construct our ORAM. While the implementations presented were shown to be oblivious in the honest-but-curious model in [AKL+20], we modify them to be secure in the malicious setting.

Throughout this section, we assume that the client always encrypts and authenticates its writes to the adversary with its own key (independent of any memory checkers). Similarly, the client passes all reads through the verification and decryption, and outputs $\perp$ if any authentication fails.

C.1 Maliciously Secure Oblivious RAM with $O\left(\log^4 N\right)$ Overhead

As a first building block, we need a maliciously secure oblivious RAM with worst-case $\mathsf{polylog}(N)$ overhead per access.

[8] Any abort behavior from $\mathcal{A}$ can be emulated by simply running the rest of the protocol honestly and ignoring everything after the abort.

Theorem C.1 ([OS97, CNS18], Theorem 4.8 of [AKL+20], Theorem 3.4 of [AKL+20])**.** *For RAM databases of size n, there is a perfectly honest-but-curious secure ORAM with $O(\log^3 n)$ worst-case bandwidth, where the adversary does not have access to any data (i.e., only the commands, inputs, outputs, and access pattern), or equivalently perfectly secure in the sense of Definition 3.3 of [AKL+20].*

By adding symmetric-key encryption to make it honest-but-curious according to Definition 3.13, we go from perfect loss to $\mathsf{negl}(\lambda)$ loss independent of n, as long as $n \leq \mathsf{poly}(\lambda)$. Lastly, we combine with Corollary 4.8 to get a maliciously secure ORAM with an additional $O(\log n)$ overhead.

Corollary C.2. *For RAM databases of size n, there is a $(1 - \mathsf{negl}(\lambda))$-maliciously secure ORAM with $O(\log^4 n)$ worst-case overhead in the sense of Definition 3.13, as long as $n \leq \mathsf{poly}(\lambda)$.*

C.2 Oblivious Sorting

The work of Ajtai, Komlós, and Szemerédi [AKS83] showed that there is a comparator-based circuit with $O(n \log n)$ comparators that can sort an array of length n.

Theorem C.3 ([AKS83])**.** *There is a deterministic oblivious sorting algorithm that sorts n elements in $O(n \log n)$ time.*

Since this algorithm can be described using a comparison-based circuit, the list of comparisons is entirely determined by the circuit, and therefore the algorithm is access-deterministic. Therefore, applying Theorem 6.2 gives us an immediate corollary.

Corollary C.4. *There is a maliciously secure oblivious implementation of $\mathcal{F}_{\mathsf{Sort}}$ with query complexity and server space complexity $O(n \log n)$.*

Similarly, [AKL+20] show a modified version Batcher's bitonic sort [Bat68] which gives a more efficient run-time when each memory word can hold up to $B > 1$ elements.

Theorem C.5 ([AKL+20])**.** *There is a deterministic packed oblivious sorting algorithm that sorts n elements in $O(\frac{n}{B} \log^2 n)$ time, where B denotes the number of elements each memory word can pack.*

Once again, this algorithm can be described using a circuit, so it is also implicitly access-deterministic. Therefore, by Theorem 6.2 we have the following corollary.

Corollary C.6. *There is a maliciously secure oblivious implementation of $\mathcal{F}_{\mathsf{Sort}}$ with query complexity and server space complexity $O(\frac{n}{B} \log^2 n)$, where B denotes the number of elements each memory word can pack.*

C.3 Oblivious Two-Key Dictionary

In this section, we construct a maliciously secure two-key dictionary similarly to Asharov et al. [AKL+20], where an element is keyed by pair of keys from $[K] \times [T]$. At a high level, it supports popping elements according to either key. We give the full functionality description in Functionality C.7.

Functionality C.7 $\mathcal{F}_{\mathsf{2KeyDict}}$ (Description from Functionality 3.3 of [AKLS21].)

$\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{Init}()$:

- **Input:** This operation has no input.
- **The procedure:** Allocate an empty list M indexed by $k \in [K]$, where all $M[k]$ are initialized to $\perp$.
- **Output:** This operation has no output.

$\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{Insert}(k, t, v)$:

- **Input:** A key k, time $t \in \mathbb{N}$, and a value v where k might be $\perp$, i.e., a dummy.
- **The procedure:** If $k \neq \perp$, set $M[k] := (t, v)$.
- **Output:** This operation has no output.

$\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{PopKey}(k)$:

- **Input:** A key k (which might be $\perp$, i.e., dummy).
- **The procedure:** Set $(t^*, v^*) \leftarrow M[k]$, and set $M[k] := \perp$.
- **Output:** The value v^*.

$\mathcal{F}_{\mathsf{2KeyDict}}.\mathsf{PopTime}(t_1, t_2)$:

- **Input:** Times $t_1, t_2 \in [T]$ such that $t_1 \leq t_2$.
- **The procedure:**
  - Let k be the smallest index such that $M[k] = (t^*, v^*)$ for some $t^* \in [t_1, t_2]$ and set $M[k] = \perp$ if such a k exists. Otherwise, if no such k exists, set $v^* := \perp$.
- **Output:** The value v^*.

Theorem C.8 (Modification of Theorem 3.4 of [AKLS21])**.** *For all $n \leq \mathsf{poly}(\lambda)$, there exists a $(1 - \mathsf{negl}(\lambda))$-maliciously secure oblivious implementation of $\mathcal{F}^n_{\mathsf{Dict}}$ such that* $\mathsf{Insert}, \mathsf{PopKey}$ *and* $\mathsf{PopTime}$ *take $O(\log^5 n)$ time in the worst case.*

Proof. Note that we can non-obliviously implement $\mathcal{F}_{\mathsf{2KeyDict}}$ by instantiating two balanced binary search trees, where the first tree orders elements according to k, and the second tree orders elements according to t. Each of these operations can be done with $O(\log n)$ overhead. Now, we can apply the maliciously secure ORAM of Corollary C.2 to obtain a maliciously secure implementation with overall $O(\log^5 n)$ multiplicative overhead in the operations. □

C.4 Oblivious Random Permutation

Let $\mathcal{F}^n_{\mathsf{Shuffle}}$ be the functionality that randomly permutes a given array.

Functionality C.9 $\mathcal{F}^n_{\mathsf{Shuffle}}(\boldsymbol{I})$: Randomly permute a given array.

Input: An array $\boldsymbol{I}$ of size n.
The procedure:

- Choose a permutation $\pi : [n] \to [n]$ uniformly at random.
- Initialize an array $\boldsymbol{B}$ of size n. Assign $\boldsymbol{B}[i] = \boldsymbol{I}[\pi(i)]$ for all $i \in [n]$.

Output: The array $\boldsymbol{B}$.

We give an implementation of $\mathcal{F}^n_{\mathsf{Shuffle}}$ in Algorithm C.10. This algorithm was given in [AKL+20] and is an adaptation of an algorithm from [CCS17], but we add memory checking to make it maliciously secure.

Algorithm C.10 ORP: Oblivious Random Permutation. This algorithm is adapted from Theorem 4.3 of Asharov et al. [AKL+20].

Input: An input array $\boldsymbol{I}$ containing n balls, each encoded with D bits.
Memory checking: All portions which are grayed out will be offline memory checked. All other operations can be time-stamped and online-checked since they only involve linear scans of contiguous memory. The portions of memory that are not online checked are denoted by superscript off.
The algorithm.

1. Iterate over $\boldsymbol{I}$ and assign each element an $8 \log n$-bit random label drawn uniformly from $\{0,1\}^{8 \log n}$.
2. Run $\boldsymbol{R} \leftarrow \mathsf{Sort}(\boldsymbol{I})$, where the sort is according to the random labels assigned to the elements. We break ties between balls x_i and x_j with the same random label according to whether $i < j$, where $i, j \in [n]$ are the indices of the balls in $\boldsymbol{I}$. We stress that this does not depend on the values of either x_i or x_j.
3. Initialize empty lists $\boldsymbol{J}^{\mathsf{off}}$ and $\boldsymbol{X}^{\mathsf{off}}$ of size n.
4. Linearly scan $\boldsymbol{R}$, writing two arrays as follows. (Note that this can be done in linear time because leakage of the indices of colliding elements does not break obliviousness.)
   - A list $\boldsymbol{J}^{\mathsf{off}}$ of the indices of all elements in $\boldsymbol{R}$ that have collisions in the random labels.
   - A list $\boldsymbol{X}^{\mathsf{off}}$ containing all the colliding elements in $\boldsymbol{R}$.
5. If the number of elements r in $\boldsymbol{X}^{\mathsf{off}}$ is greater than $\sqrt{n}$, output $\mathsf{Overflow}$ and abort.
6. Initialize an array $\boldsymbol{Y}^{\mathsf{off}}$ of size r.
7. Run a naïve quadratic oblivious shuffle algorithm on $\boldsymbol{X}^{\mathsf{off}}$ to obtain a shuffled array $\boldsymbol{Y}^{\mathsf{off}}$. To do this, iterate over $i \in [r]$:
   - Assign $\boldsymbol{X}^{\mathsf{off}}[i]$ a tag uniformly from $t \leftarrow [r - i + 1]$.
   - Iterate over all of $\boldsymbol{Y}^{\mathsf{off}}$, and write $\boldsymbol{X}^{\mathsf{off}}[i]$ at the t-th unoccupied location of $\boldsymbol{Y}^{\mathsf{off}}$ (while performing dummy writes to all other indices of $\boldsymbol{Y}^{\mathsf{off}}$).
8. Initialize an array $\boldsymbol{R}^{\mathsf{off}}$ of size $|\boldsymbol{R}|$.
9. Copy the contents of $\boldsymbol{R}$ to $\boldsymbol{R}^{\mathsf{off}}$ sequentially.
10. Iterate over $\boldsymbol{J}^{\mathsf{off}}$ and route element $\boldsymbol{Y}^{\mathsf{off}}[i]$ to address $\boldsymbol{R}^{\mathsf{off}}[\boldsymbol{J}^{\mathsf{off}}[i]]$.
11. Copy the contents of $\boldsymbol{R}^{\mathsf{off}}$ to $\boldsymbol{R}$ sequentially, using the post-verifiability of $\boldsymbol{R}^{\mathsf{off}}$ from our offline memory checker. (As usual, the writes to $\boldsymbol{R}$ are time-stamped.)

12. Iterate over $\boldsymbol{R}$, and remove the random labels we had assigned at Step 1.

Output: The array $\boldsymbol{R}$.

Theorem C.11 (Theorem 4.3 of [AKL+20])**.** *Let D denote the number of bits it takes to encode an element. Then, Algorithm C.10 (without memory checking) is a $(1 - e^{-\sqrt{n}})$-oblivious honest-but-curious implementation of $\mathcal{F}^n_{\mathsf{Shuffle}}$, and runs in time $O(T^{D+8\log n}_{\mathsf{Sort}}(n) + n)$, where $T^{\ell}_{\mathsf{Sort}}(n)$ is the upper bound on the time it takes to sort n elements of size ℓ bits.*

Theorem C.12. *Algorithm C.10 is a $(1 - e^{-\sqrt{n}} - \mathsf{negl}(\lambda))$-maliciously secure oblivious implementation of $\mathcal{F}^n_{\mathsf{Shuffle}}$ with query complexity $O(n + T^{D+8\log n}_{\mathsf{Sort}}(n))$, where $T^{\ell}_{\mathsf{Sort}}(n)$ is the upper bound on the time it takes to sort n elements of size ℓ bits. In particular, for $n \geq \log^3(\lambda)$, Algorithm C.10 is $(1 - \mathsf{negl}(\lambda))$-maliciously secure.*

Proof. To see completeness (in the sense of Definition 3.6), we defer to honest-but-curious security C.11 to see that it aborts with probability at most $e^{-\sqrt{n}}$ against an honest-but-curious adversary.

To see correctness and obliviousness, consider the simulator which simply runs ORP on an input of n dummies. We show that no adversary can distinguish $\textsc{Real}(\mathcal{C}_{\mathsf{ORP}}, \mathcal{A})$ and $\textsc{Ideal}(\mathcal{F}^n_{\mathsf{Shuffle}}, \mathcal{S}, \mathcal{A})$, where $\mathcal{C}_{\mathsf{ORP}}$ runs the implementation of ORP as in Algorithm C.10.

Experiment Hybrid_0: The view $\textsc{Real}(\mathcal{C}_{\mathsf{ORP}}, \mathcal{A})$.

Experiment Hybrid_1: We replace the online memory checker M with an idealized version M' which behaves identically to M, except ensures (with probability 1) that any data^* sent back to $\mathcal{C}$ is correct and aborts otherwise. Note that this can be achieved by simply having M' store a copy of the database which it updates locally (M has a sublinear space restriction while M' does not).

This is negligibly close to Hybrid_0 by the soundness guarantee of the online memory checker.

Experiment Hybrid_2: We replace the offline memory checker M_{off} with an idealized version M'_{off} which rejects any executions with errors with probability 1.

This is negligibly close to Hybrid_1 by the soundness guarantee of the offline memory checker.

Experiment Hybrid_3: We modify the client to augment it with additional space to check that all ciphertexts passing the authentication verfication have been generated by the MAC before, aborting if this is not the case.

This is negligibly close to Hybrid_2 by unforgeability of the MAC scheme.

Experiment Hybrid_4: In this hybrid, the client keeps a local dictionary D that keeps track of all the ciphertexts ct_i and the corresponding plaintexts, i.e., sets $D[\mathsf{ct}_i] \leftarrow \mathsf{data}_i$. Now, instead of using the decryption algorithm, the client looks up the dictionary $D[\mathsf{ct}_i]$ instead (note that the client only decrypts the data after checking the MAC against the ciphertext and address).

Claim C.13. $|\Pr[\mathsf{Hybrid}_4 = 1] - \Pr[\mathsf{Hybrid}_3 = 1]| = 0$.

Proof. By perfect MAC unforgeability after Hybrid_3 and perfect correctness of encryption, the view of the adversary in both worlds is identical. □

Experiment Hybrid_5: This hybrid is identical to Hybrid_4 except we replace all ciphertexts ct_i corresponding to data_i with encryptions of 0, still keeping track of the dictionary $D[\mathsf{ct}_i] \leftarrow \mathsf{data}_i$.

Claim C.14. $|\Pr[\mathsf{Hybrid}_5 = 1] - \Pr[\mathsf{Hybrid}_4 = 1]| \leq \mathsf{negl}(\lambda)$.

Proof. We construct an adversary $\mathcal{A}'$ against the adaptive IND-CPA game, as defined in Section 3.2. Specifically, we let $\mathcal{A}'$ be everything besides the encryption layer in Hybrid_4 and Hybrid_5 except with the encryptions chosen as follows:

- For encryptions, instead of encrypting data_i, it sends $m_0 = \mathsf{data}_i$ and $m_1 = 0$ to the adaptive IND-CPA challenger to get back a ciphertext $\mathsf{ct}_i \leftarrow \mathsf{Enc}_k(m_b)$, which it passes through Hybrid_4 in the usual way. As before, we keep track of a dictionary D and set $D[\mathsf{ct}_i] \leftarrow \mathsf{data}_i$ for each ciphertext generated by the challenger.
- For decryptions, as before, when receiving ct_i, we let $\mathsf{data}_i \leftarrow D[\mathsf{ct}_i]$ be the result of the decryption as given to the client. Note that ct_i must exist in D by the assumption in Hybrid_3 that all ciphertexts that passed the authentication check have already been seen before. (The reason for this D is that the IND-CPA security game does not give access to a decryption oracle. This is why we use authenticated encryption as opposed to just encryption.)

When $b = 0$, this is exactly the view of Hybrid_4. When $b = 1$, this is exactly the view of Hybrid_5. □

Experiment Hybrid_6: The same as Hybrid_5, except we replace the input to the algorithm with dummies, and we replace the output to come directly from $\mathcal{F}^n_{\mathsf{Shuffle}}$ rather than from the algorithm.

Claim C.15. $\Pr[\mathsf{Hybrid}_6 = 1] = \Pr[\mathsf{Hybrid}_5 = 1]$.

Since the data_i component of the accesses no longer have any dependence on the input $\boldsymbol{I}$ due to Hybrid_5, it suffices to show the joint distribution of the real access pattern and the output permutation is close to the joint distribution of the dummy access pattern and an independent random permutation. Since the access pattern has no dependence on $\boldsymbol{I}$, the behavior up to and including the abort conditions are identical between the two hybrids. Therefore, it suffices to consider the case when there is no abort. In that case, the perfect soundness of our idealized memory checkers allows us consider the honest-but-curious case, as the algorithm must have been executed honestly with probability 1. We invoke the honest-but-curious security argument from Lemma 11 of [CCS17] to see that the joint distribution of the real access pattern and output permutation is equal to the joint distribution of the dummy access pattern and independent random permutation.

Experiment Hybrid_7: In this hybrid, we essentially unravel Hybrids 0-5.

- Replace the encryptions of zero with the encryptions of the true values.
- Conduct the MAC check rather than saving all the ciphertexts passing through for authentication verification.
- Replace the idealized online and offline checkers M' and M'_{off} with the original versions, M and M_{off}.

Claim C.16. $|\Pr[\mathsf{Hybrid}_7 = 1] - \Pr[\mathsf{Hybrid}_6 = 1]| \leq \mathsf{negl}(\lambda)$.

Note that Hybrid_7 is in fact exactly $\text{IDEAL}(\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{S}, \mathcal{A})$, therefore completing the proof.

Since all offline memory checks are conducted over $O(n)$ memory, the query complexity of the algorithm simply has an additive factor of $O(n)$. □

C.5 Oblivious Bin Placement

In this section, we consider the functionality $\mathcal{F}_{\mathsf{Placement}}$ where we are given an array $\boldsymbol{I}$ of n real and dummy elements such that each element has a tag from $\{1, 2, \ldots, n\} \cup \{\perp\}$, and we have to output an array $\boldsymbol{O}$ such that the real elements are at the index corresponding to their tag.

Chan et al. [CGLS17] show in Section 3.2 of their paper that a more generalized version of this algorithm can be implemented with $O(1)$ oblivious sorts, and we explicitly state a special case of their algorithm, Algorithm C.17, for completeness.

Algorithm C.17 Placement: An algorithm to obliviously route elements in an array. This algorithm is a special case of the oblivious bin placement algorithm from [CGLS17].

Input: An array $\boldsymbol{I}$ of n real and dummy elements such that each element has a tag from $\{1, 2, \ldots, n\} \cup \{\perp\}$. Every real element is guaranteed to have a distinct tag, and every dummy element is tagged with $\perp$.
Memory checking: Since the algorithm only consists of linear iterations over contiguous memory and hybrid calls to $\mathcal{F}_{\mathsf{Sort}}$, it is time-stampable. Hence, we online check every query.
The algorithm:

- Create an array of length n of filler elements $\boldsymbol{F}$ where $\boldsymbol{F}[i] := (\mathsf{fill}, i)$, i.e., the tag of this filler element is i.
- Let $\boldsymbol{A} = \boldsymbol{I} || \boldsymbol{F}$.
- Let $\boldsymbol{A}' \leftarrow \mathsf{Sort}(\boldsymbol{A})$, where the elements are sorted by increasing order of tag, treating $\perp$ as largest (i.e., all dummy elements appear at the end). Ties among tags are broken so that the real elements appear before fillers.
- Iterate over $\boldsymbol{A}'$ linearly, and do the following:
  - If $\boldsymbol{A}'[j]$ is a real element, mark it as "`good`".
  - If $\boldsymbol{A}'[j] = (\mathsf{fill}, i)$,
    - If $\boldsymbol{A}'[j-1]$ was tagged with i, mark $\boldsymbol{A}'[j]$ as "`excess`".
    - Otherwise, mark $\boldsymbol{A}'[j]$ as "`good`".
  - If $\boldsymbol{A}'[j] = \mathsf{dummy}$, mark it as "`excess`".
- Let $\boldsymbol{J}' \leftarrow \mathsf{Sort}(\boldsymbol{A}')$, where we sort by prioritizing elements marked as "`good`" over elements marked as "`excess`" and among them breaking ties according to the tags.
- Obtain $\boldsymbol{J}$ by truncating $\boldsymbol{J}'$ to length n.

Output: Output array $\boldsymbol{J}$.

Claim C.18. Placement *(Algorithm C.17) is a maliciously secure oblivious implementation of* $\mathcal{F}_{\mathsf{Placement}}$. *Using a standard oblivious implementation of* Sort*, we obtain a query and server space complexity of* $O(n \log n)$. *Moreover, by using a packed implementation of* Sort *as in Corollary C.6, we obtain a query and server space complexity of* $O(\frac{nD}{w} \cdot \log^2 n + n)$*, where* D *denotes the number of bits to encode a single element.*

Proof. As shown by Chan et al. [CGLS17], this algorithm is an honest-but-curious oblivious implementation of $\mathcal{F}_{\mathsf{Placement}}$. Moreover, since the algorithm comprises of hybrid calls to Sort and linear scans of $\boldsymbol{A}$ and $\boldsymbol{A}'$, it is time-stampable in the $\mathcal{F}_{\mathsf{Sort}}$-hybrid model. Therefore, by Corollary 4.6, we have the desired result. □

C.6 Oblivious Balls-into-Bins Sampling

In this section, we consider the functionality $\mathcal{F}^{n,m}_{\mathsf{throw\text{-}balls}}$ which throws n balls into m bins uniformly at random and outputs the bin loads. To obtain an efficient oblivious implementation, Asharov et al. [AKL+20] use the following approximation for $\mathcal{F}^{n}_{\mathsf{binomial}}$, the functionality that outputs a sample from $\mathsf{Binomial}(n, 1/2)$.

Theorem C.19 (Theorem 4.18 of [AKL+20], Theorem 5 of [BKP+14])**.** *For any $n = 2^{O(w)}$, there is a $(1-n\cdot\delta)$-oblivious algorithm $\mathsf{SampleApproxBinomial}_\delta$ that implements the functionality $\mathcal{F}^{n}_{\mathsf{binomial}}$ in time $O(\log^5(1/\delta))$.*

By generically applying an online memory checker as in Corollary 4.8, this can be made maliciously secure with an extra $\log\log(1/\delta)$ factor blowup.

Corollary C.20. *For any $n = 2^{O(w)}$, there is a $(1 - n\cdot\delta - \mathsf{negl}(\lambda))$-maliciously secure oblivious algorithm $\mathsf{SampleApproxBinomial}_\delta$ that implements the functionality $\mathcal{F}^{n}_{\mathsf{binomial}}$ in time $O(\log^5(1/\delta)\cdot \log\log(1/\delta))$.*

From now on, when we refer to $\mathsf{SampleApproxBinomial}_\delta$, we refer to the maliciously secure oblivious implementation.

Now, we can implement $\mathcal{F}^{n,m}_{\mathsf{throw\text{-}balls}}$ using Algorithm 4.19 of Asharov et al. [AKL+20].

Algorithm C.21 $\mathsf{SampleBinLoads}_{m,\delta}(n)$. This is Algorithm 4.19 in [AKL+20].

Input: A secret number of balls $n \in \mathbb{N}$.
Public parameters: The number of bins $m \in \mathbb{N}$, which is a power of 2.
Memory checking: Since the algorithm only consists of linear iterations over contiguous memory, it is time-stampable. Hence, we online check every query.
The algorithm:

- (Base case.) If $m = 1$, output n. Otherwise, continue with the following.
- Sample a binomial random variable $X \leftarrow \mathsf{SampleApproxBinomial}_\delta(n)$, where X is the total number of balls in the first $m/2$ bins. Recursively call $L_1 \leftarrow \mathsf{SampleBinLoads}_{m/2,\delta}(X)$ and $L_2 \leftarrow \mathsf{SampleBinLoads}_{m/2,\delta}(n - X)$.
- Output the concatenated array $L_1||L_2$.

Claim C.22 (Adaptation of Theorem 4.20 of [AKL+20])**.** *For any integer $n = 2^{O(w)}$, m a power of 2, and $m \le \mathsf{poly}(\lambda)$, $\mathsf{SampleBinLoads}_{m,\delta}$ is a $(1 - m\cdot n\cdot\delta - \mathsf{negl}(\lambda))$-maliciously secure oblivious implementation of the functionality $\mathcal{F}^{n,m}_{\mathsf{throw\text{-}balls}}$, with query complexity $O(m\cdot\log^5(1/\delta)\cdot\log\log(1/\delta))$.*

Proof. By Theorem 4.20 in [AKL+20], we know that this is an honest-but-curious implementation of $\mathcal{F}^{n,m}_{\mathsf{throw\text{-}balls}}$. When $m = 1$, the implementation is clearly secure. Now, suppose the implementation is secure for all $m \leq m'$. Consider $m = 2m'$. Then, note that in the $\mathsf{SampleBinLoads}_{m/2,\delta}$-hybrid model, the algorithm only consists of linear scans and is therefore time-stampable. Therefore, by Corollary 4.6, we have that $\mathsf{SampleBinLoads}_{m,\delta}$ is a maliciously secure oblivious implementation of $\mathcal{F}^{n,m}_{\mathsf{throw\text{-}balls}}$ in the $\mathcal{F}_{\mathsf{binomial}}$-hybrid model. Now, we apply Corollary C.20 to give us the desired run-time. □

C.7 Tight Compaction, Intersperse and Perfect Random Permutation

Tight compaction. Suppose we are given an input array given n balls, each of which are labelled either 0 or 1. The problem of outputting a permutation of the input array such that the 1-balls are moved to the front of the array is known as *tight compaction*. Asharov et al. [AKL+20] given a deterministic linear-time oblivious implementation of tight compaction.

Theorem C.23 (Theorem 1.2 in [AKL+20])**.** *Algorithm 5.3 in [AKL+20] is an access-deterministic, honest-but-curious oblivious algorithm for tight compaction with query complexity $O(n)$ which compacts any input array of n elements.*

We now argue that this algorithm can be made maliciously obliviously secure.

Corollary C.24. *There is a maliciously secure oblivious algorithm $\mathsf{TightCompaction}_n(\boldsymbol{I})$ for tight compaction with query and server space complexity $O(n)$.*

Proof. By Theorem C.23, we know there exists an access-deterministic, honest-but-curious oblivious algorithm with the desired efficiency. Therefore, by applying Theorem 6.2, we have a maliciously secure oblivious algorithm with the same efficiency. □

Intersperse. Now, we consider the intersperse algorithm from OptORAMa [AKL+20]. Informally, intersperse does the following:

- **Input:** An array $\boldsymbol{I} := \boldsymbol{I}_0 || \boldsymbol{I}_1$ of size n, where $|I_b| = n_b$ and $n = n_0 + n_1$. Note that $n := n_0 + n_1$, but both n_0 and n_1 are hidden.
- **Output:** An array $\boldsymbol{B}$ of size n that contains all elements of $\boldsymbol{I}_0$ and $\boldsymbol{I}_1$. Each position in $\boldsymbol{B}$ holds an element from either $\boldsymbol{I}_0$ or $\boldsymbol{I}_1$, chosen uniformly at random (and these random choices are hidden from the adversary).

In the special case where the inputs $\boldsymbol{I}_0$ and $\boldsymbol{I}_1$ are randomly shuffled, this algorithm realizes $\mathcal{F}^n_{\mathsf{Shuffle}}$. Asharov et al. [AKL+20] give such an algorithm.

Theorem C.25 (Claim 6.3 in [AKL+20])**.** *There exists an access-deterministic, honest-but-curious oblivious intersperse algorithm (Algorithm 6.1, [AKL+20]) with $O(n)$ query complexity and server space complexity.*

Since the algorithm is write-deterministic, by applying Theorem 6.2, we have the following corollary.

Corollary C.26. *There is a maliciously secure oblivious algorithm* $\mathsf{Intersperse}_n(\boldsymbol{I}_0||\boldsymbol{I}_1; n_0, n_1)$ *for interspersing arrays with* $O(n)$ *query complexity and* $O(n)$ *server space complexity.*

Another useful building block is a related algorithm called real-dummy intersperse. Informally, real-dummy intersperse does the following:

- **Input:** An array $\boldsymbol{I}$ of n elements, where each element is tagged either real or dummy.
- **Output:** The output is an array $\boldsymbol{B}$ of size $|\boldsymbol{I}|$ that is a permutation of $\boldsymbol{I}$. If the real elements in $\boldsymbol{I}$ are randomly shuffled, then $\boldsymbol{B}$ will also be randomly shuffled.

Theorem C.27 (Claim 6.7 of [AKL+20])**.** *There is an access-deterministic, honest-but-curious oblivious algorithm for real-dummy intersperse with* $O(n)$ *query complexity and server space complexity.*

Since the algorithm is access-deterministic, by applying Theorem 6.2, we have the following corollary.

Corollary C.28. *There is a maliciously secure oblivious algorithm* $\mathsf{IntersperseRD}_n(\boldsymbol{I})$ *for interspersing reals and dummies with* $O(n)$ *query complexity and server space complexity.*

Perfect oblivious random permutation. Asharov et al. [AKL+20] give a divide-and-conquer approach to generate a random permutation $\pi : [n] \to [n]$ array with perfect obliviousness. With encryption to hide the data, this becomes honest-but-curious $(1 - \mathsf{negl}(\lambda))$-oblivious, independently of n as long as $n \leq \mathsf{poly}(\lambda)$.

Theorem C.29 (Theorem 4.6 of [AKL+20])**.** *There is an access-deterministic, honest-but-curious oblivious implementation (Algorithm 6.8 in [AKL+20]) of* $\mathcal{F}^n_{\mathsf{Shuffle}}$ *with* $O(n \log n)$ *query complexity and* $O(n)$ *server space complexity.*

Now, we again apply Theorem 6.2 to obtain the following corollary.

Corollary C.30. *There is a maliciously secure oblivious implementation* $\mathsf{PerfectORP}_n(\boldsymbol{I})$ *of* $\mathcal{F}^n_{\mathsf{Shuffle}}$ *with query complexity* $O(n \log n)$ *and server space complexity* $O(n \log n)$*.*

Note that the server space complexity is now $O(n \log n)$ instead of $O(n)$ because of Theorem 6.2 and the fact that the query complexity of the underlying honest-but-curious algorithm is $O(n \log n)$.

C.8 Oblivious Cuckoo Hashing

Cuckoo hashing [PR04] is a hashing technique with $O(1)$ blow-up in space complexity and essentially $O(1)$ lookup time. In this section, we outline an oblivious way to construct these Cuckoo hash tables following the works of Chan et al. [CGLS17] and Asharov et al. [AKL+20], and we show how to make the construction maliciously secure.

At a high level, in the general Cuckoo hashing problem, we would like to place n balls into a table of size $c_{\mathsf{cuckoo}} \cdot n$, where $c_{\mathsf{cuckoo}} > 1$ is some fixed constant. Each ball receives two independent bin choices in the range $[c_{\mathsf{cuckoo}} \cdot n]$. During the build phase, the algorithm picks either a bin-choice for

each ball, or places it in an overflow stash S. If we allow a stash of size s, Kirsch et al. [KMW10] show that one can successfully find a cuckoo hashing assignment without collisions with probability $1-n^{-\Omega(s)}$. For the rest of this section, let $n_{\mathsf{cuckoo}} = c_{\mathsf{cuckoo}} \cdot n + \log \lambda$. In other words, we are setting the stash size $s = \log \lambda$.

In general, the output of a Cuckoo hashing table could have bin assignments which depend on the keys of the balls, not just the bin-choice metadata array. Asharov et al. [AKL+20] define a notion of *indiscriminate hashing*, where the location of any real ball does not depend on the key, and is fully determined by its relative index in the input array as well as the bin-choice metadata array $\boldsymbol{I}$. This additional property is required for obliviousness of MalHT.

As discussed in Appendix B in [AKL+20], finding a Cuckoo hash assignment can be implemented through a constant number of rounds of oblivious BFS which in turn just use (access-deterministic) oblivious sorts. Therefore, by inspection, it is clear that these algorithms are access-deterministic. We recap the theorems here.

Theorem C.31 (Corollary 4.11 in [AKL+20])**.** *Suppose that $\delta > 0$, $n \geq \log^8(1/\delta)$, and the stash size s satisfies $s \geq \log(1/\delta)/n$. Then, there is an indiscriminate Cuckoo hashing assignment algorithm which is $(1-O(\delta))$-access-deterministic and honest-but-curious oblivious with query complexity $O(n \log n)$ when $w = \Omega(\log n)$.*

Since the algorithm is write-deterministic, we obtain the following corollary by applying Theorem 6.2.

Corollary C.32. *Suppose that $\delta > 0$, $n \geq \log^8(1/\delta)$, and the stash size s satisfies $s \geq \log(1/\delta)/n$. Then, there is an oblivious indiscriminate Cuckoo hashing algorithm* $\mathsf{cuckooAssign}$ *which is $(1 - O(\delta) - \mathsf{negl}(\lambda))$-maliciously secure with query and server space complexity $O(n \log n)$ when $w = \Omega(\log n)$.*

If a single plaintext word can contain many elements of length $\ell := 8 \log_2 n_{\mathsf{cuckoo}}$ bits, Asharov et al. [AKL+20] use packed oblivious sorting in the cuckoo assignment algorithm to obtain the following run-time.

Theorem C.33 (Packed oblivious Cuckoo assignment, Corollary 4.12 in [AKL+20])**.** *Suppose that $\delta > 0$, $n \geq \log^8(1/\delta)$, and the stash size s satisfies $s \geq \log(1/\delta)/n$. Suppose each element can be expressed in $\ell := 8 \cdot \log_2 n_{\mathsf{cuckoo}}$ bits. Then, there is an indiscriminate* packed *Cuckoo hashing algorithm which is $(1 - O(\delta))$-access-deterministic, honest-but-curious oblivious with query complexity $O(n_{\mathsf{cuckoo}} + (n_{\mathsf{cuckoo}}/w) \cdot \log^3 n_{\mathsf{cuckoo}})$.*

Once again, since the algorithm is write-deterministic, we obtain the following corollary by applying Theorem 6.2.

Corollary C.34. *Suppose that $\delta > 0$, $n \geq \log^8(1/\delta)$, and the stash size s satisfies $s \geq \log(1/\delta)/n$. Suppose each element can be expressed in $\ell := 8 \cdot \log_2 n_{\mathsf{cuckoo}}$ bits. Then, there is an oblivious indiscriminate* packed *Cuckoo hashing algorithm which we denote by* $\mathsf{packedcuckooAssign}$ *which is $(1-O(\delta)-\mathsf{negl}(\lambda))$-maliciously secure with query and server space complexity $O(n_{\mathsf{cuckoo}}+(n_{\mathsf{cuckoo}}/w)\cdot \log^3 n_{\mathsf{cuckoo}})$.*

C.8.1 Oblivious Cuckoo Hashing Subroutines

In order to achieve essentially linear-time Cuckoo hashing, Asharov et al. [AKL+20] give subroutines that capitalize on *packed sorting*. We give the subroutines in this section. Note that both of these algorithms are clearly time-stampable in their respective hybrids. We emphasize that we use these as *subroutines* (not hybrid calls) in our MalHT construction.

At a high level, we are given n balls, and we first randomly shuffle it with $n_{\mathsf{cuckoo}} - n$ dummy elements. Then, we abstract out a "metadata array" corresponding to the bin choices, given by $\mathbf{MD} = \{(u_i, v_i)\}_{i \in n_{\mathsf{cuckoo}}}$. For real balls, the pair (u_i, v_i) contains two bin choices from $[n_{\mathsf{cuckoo}}]$. For dummy balls, $(u_i, v_i) = (\perp, \perp)$. This subroutine, $\mathsf{CuckooInit}$, is concretely described in Algorithm C.35.

Then, we compute a valid Cuckoo hashing assignment for all the real balls in the input by calling $\mathsf{packedcuckooAssign}$ as in Corollary C.34. And then, we allocate the dummy elements in a random order to the empty slots in the assignment. This subroutine, $\mathsf{CuckooMD}$, is described concretely in Algorithm C.37.

Algorithm C.35 $\mathsf{CuckooInit}_n$: Initializing and shuffling the array to be the correct length for Cuckoo Hashing. This is Algorithm 8.1 in [AKL+20].

Input: An input array $\boldsymbol{I}$ of length n consisting of real and dummy elements.
Memory checking: It is clear that the following pseudocode can be time-stamped in the $(\mathsf{ORP}_n, \mathsf{TightCompaction}, \mathsf{Intersperse})$-hybrid model. As a subroutine in MalHT (Algorithm 8.2), we therefore time-stamp the following subroutine.
The algorithm:

- Count the number of real elements in $\boldsymbol{I}$. Let n_R be the result.
- Write down a metadata array $\mathbf{MD}$ of length n_{cuckoo}, where the first n_R elements contain only a symbol real, and the remaining $n_{\mathsf{cuckoo}} - n_R$ elements are of the form $(\perp, 1), (\perp, 2), \ldots, (\perp, n_{\mathsf{cuckoo}} - n_R)$ to represent dummies.
- Call $\mathbf{MD}' \leftarrow \mathsf{ORP}_n(\mathbf{MD})$, packing $O\left(\frac{w}{\log n}\right)$ elements into a single memory word.
- Run $\mathsf{TightCompaction}_{n_{\mathsf{cuckoo}}}(\mathbf{MD}')$ to move the real elements to the front and the dummy elements to the end.
- Run $\boldsymbol{I}' \leftarrow \mathsf{TightCompaction}_{n_{\mathsf{cuckoo}}}(\boldsymbol{I})$ to move the real elements of the original array (with the values from the input) to the front.
- Initialize an array $\boldsymbol{J}$ of size n_{cuckoo}.
- For $i = 1, 2, \ldots, n_{\mathsf{cuckoo}}$:
  - If $i \le n_R$, read $\mathsf{data} \leftarrow \boldsymbol{I}'[i]$, access $\mathbf{MD}'[i]$, and write data to $\boldsymbol{J}[i]$.
  - If $i > n_R$, access $\boldsymbol{I}'[i]$, read $\mathsf{data} \leftarrow \mathbf{MD}'[i]$ (with w bits), and write data to $\boldsymbol{J}[i]$.
- Run $\boldsymbol{J}' \leftarrow \mathsf{Intersperse}_{n_{\mathsf{cuckoo}}}(\boldsymbol{J}, n_R, n_{\mathsf{cuckoo}} - n_R)$.

Output: The array $\boldsymbol{J}'$.

Claim C.36 (Claim 8.2 of [AKL+20])**.** *Algorithm C.35 fails with probability at most $e^{-\Omega(\sqrt{n})}$ and runs in $O(n + \frac{n}{w} \cdot \log^3 n)$ time. When $n = \log^9 \lambda$ and $w \ge \log^3 \log \lambda$, the algorithm runs in time $O(n)$ time and fails with probability $e^{-\Omega(\log^{9/2} \lambda)} = \mathsf{negl}(\lambda)$.*

Algorithm C.37 $\mathsf{CuckooMD}$: Compute the bucket assignment of a Cuckoo hashing instance. This algorithm is Algorithm 8.3 in [AKL+20].

Input: An array $\mathbf{MD}_{\boldsymbol{X}}$ of length $n_{\mathsf{cuckoo}} = c_{\mathsf{cuckoo}} \cdot n + \log \lambda$, where each element is either dummy or a pair $(\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$ where $\mathsf{choice}_{i,b} \in [c_{\mathsf{cuckoo}} \cdot n]$ for all $b \in \{1, 2\}$, and the number of real pairs is at most n.

Memory checking: It is clear that the following pseudocode can be time-stamped in the $(\mathsf{cuckooAssign}, \mathsf{Placement}, \mathsf{Sort})$-hybrid model. As a subroutine in MalHT (Algorithm 8.2), we can therefore time-stamp this subroutine.

The algorithm.

1. Run $\mathsf{packedcuckooAssign}$ as promised in Corollary C.34 with parameter $\delta = e^{-\log \lambda \log\log \lambda}$, and let $\mathbf{Assign}_{\boldsymbol{X}}$ be the result. In particular $\mathbf{Assign}_{\boldsymbol{X}}$ has the following properties:
   - If $\mathbf{MD}_{\boldsymbol{X}}[i] = (\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2})$, then $\mathbf{Assign}_{\boldsymbol{X}}[i] \in \{\mathsf{choice}_{i,1}, \mathsf{choice}_{i,2}\} \cup \mathsf{S}_{\mathsf{stash}}$.
   - If $\mathbf{MD}_{\boldsymbol{X}}[i] = \mathsf{dummy}$, we have that $\mathbf{Assign}_{\boldsymbol{X}}[i] = \perp$.
2. Run $\mathbf{Occupied} \leftarrow \mathsf{Placement}(\mathbf{Assign}_{\boldsymbol{X}})$. In other words, $\mathbf{Occupied}[j] = i$ if $\mathbf{Assign}_{\boldsymbol{X}}[i] = j$, and $\mathbf{Occupied}[j] = \perp$ otherwise.
3. Iterate over $\mathbf{Assign}_{\boldsymbol{X}}$, and tag the ith element with tag i.
4. Run $\widetilde{\mathbf{Assign}} \leftarrow \mathsf{Sort}(\mathbf{Assign}_{\boldsymbol{X}})$ where we sort so that the real elements are in the front, and the dummy elements appear at the end ordered by the dummy index (where the dummy indices were computed in Algorithm C.35).
5. Iterate over $\mathbf{Occupied}$ and label the ith element with tag i.
6. Run $\widetilde{\mathbf{Occupied}} \leftarrow \mathsf{Sort}(\mathbf{Occupied})$, where we sort so that the occupied bins appear first, and the empty bins appear at the end.
7. For $j = 1, 2, \ldots, n_{\mathsf{cuckoo}}$:
   - If $\widetilde{\mathbf{Assign}}[j]$ is a real element, perform a dummy access to $\widetilde{\mathbf{Occupied}}[j]$ and a dummy write back to $\widetilde{\mathbf{Assign}}[j]$ same contents.
   - If $\widetilde{\mathbf{Assign}}[j]$ is a dummy, update the bin for $\widetilde{\mathbf{Assign}}[j]$ to be the corresponding tag of $\widetilde{\mathbf{Occupied}}[j]$.
8. Run $\mathbf{Assign}_{\boldsymbol{X}} \leftarrow \mathsf{Sort}(\widetilde{\mathbf{Assign}})$ where we sort according to the label in Step 3.
9. **Output:** The array $\mathbf{Assign}_{\boldsymbol{X}}$.

Claim C.38 (Claim 8.4 in [AKL+20])**.** *For $n \geq \log^9 \lambda$, Algorithm C.37 fails with probability at most $e^{-\Omega(\log \lambda \cdot \log\log \lambda)} = \mathsf{negl}(\lambda)$, and completes in $O\left(n \cdot \left(1 + \frac{\log^3 n}{w}\right)\right)$ time. For $n = \log^9 \lambda$ and $w \geq \log^3 \log \lambda$, it runs in $O(n)$ time.*

C.9 Deduplication

One additional building block that we need is linear-time *deduplication*, as constructed by Asharov et al. [AKLS21]. We first describe the functionality $\mathcal{F}_{\mathsf{Dedup}}$.

Functionality C.39 $\mathcal{F}^n_{\mathsf{Dedup}}(X_1, X_2)$

Input: Arrays X_1, X_2 of size n and $2n$ respectively. At least half of the elements of X_2 are dummies.
Input Assumptions: The arrays X_1 and X_2 are randomly shuffled, and the real elements among each X_b have no duplicate keys.
Output: A randomly shuffled output array of length $2n$ containing all the real elements of X_1 and X_2 with no duplicates. When there are duplicates, the copy in X_1 is preferred.

A first attempt at doing this obliviously would be to hash one table using $\mathcal{F}_{\mathsf{HT}}$, and iterate over the second table and perform lookups to the first table while removing any duplicates. However, our hash table functionality MalHT does not have constant lookup time, so this algorithm would have a run-time of $O(n \cdot \log \lambda)$. In order to obtain a truly linear time algorithm, we use MalHT in a *white-box* way, similarly to how the deduplication algorithm in Asharov et al. [AKL+20] uses CombHT in a white-box way.

Algorithm C.40 $\mathsf{Dedup}(X_1, X_2)$. This algorithm is adapted from Theorem 4.1 of Asharov et al. [AKLS21].

Input Assumption: Arrays X_1 and X_2 have size n and $2n$ respectively, and they are independently, randomly shuffled. The keys within each array are unique (but crucially, there may be duplicates between X_1 and X_2). At least n elements in X_2 are dummies.
Secret key: Sample a random PRF secret key sk. Use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\texttt{Enc}\text{”}||\cdot)$ for all encryptions, and use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\texttt{MAC}\text{”}||\cdot)$ for all MACs.
Authenticated Encryption: For every write query $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$, data is replaced with $\mathsf{data}' := (\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathsf{data}), \sigma = \mathsf{MAC}_{\mathsf{sk}}(\mathsf{ct}, \mathsf{addr}))$. Reads are passed through authenticated decryption, namely unpacking $\mathsf{data}' = (\mathsf{ct}^*, \sigma^*)$, checking $\mathsf{MACVer}_{\mathsf{sk}}((\mathsf{ct}^*, \mathsf{addr}), \sigma^*) = 1$, aborting if verification fails, and otherwise returning $\mathsf{data}^* = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}^*)$.
The algorithm.

1. Initialize T_1 to be a MalHT instance. Perform $\mathsf{T}_1.\mathsf{Build}(X_1)$. (We call this a MalHT instance for ease of understanding, but in reality, this is a black-box hybrid call to $\mathcal{F}_{\mathsf{HT}}$.)
2. Initialize $\mathsf{T}_2^{\mathsf{whitebox}}$ to be a *white-box* MalHT instance. Here, we will not be accessing MalHT in the hybrid model, but rather we use it in a white-box way. In particular, we have access to the secret state of $\mathsf{T}_2^{\mathsf{whitebox}}$ Call $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Build}(X_2)$, and let $\mathsf{OF}_{2,\mathsf{S}}, \mathsf{SecS}_{2,\mathsf{S}}$ denote the overflow and the additional stashes of $\mathsf{T}_2^{\mathsf{whitebox}}$ respectively.
3. Initialize an empty array L.
4. Linearly scan $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{SecS}_{2,\mathsf{S}}$, and for each (k, v):
   - Perform $(k', v') \leftarrow \mathsf{T}_1.\mathsf{Lookup}(k)$.
   - If k is found in T_1, mark the element as $\perp$ in $\mathsf{OF}_{2,\mathsf{S}}$ or $\mathsf{SecS}_{2,\mathsf{S}}$ accordingly. Otherwise, perform a dummy write (with unmodified contents).
   - Write (k', v') (even if $k' = \perp$) in the next slot of L, along with the tag “$\texttt{OF}$” or “$\texttt{SecS}$” depending on where k is from.
5. Run $L' \leftarrow \mathsf{PerfectORP}(L)$.
6. Run $\mathsf{S}'_1 \leftarrow \mathsf{T}_1.\mathsf{Extract}()$, and run $\mathsf{S}_1 \leftarrow \mathsf{Intersperse}(\mathsf{S}'_1||L')$.

7. Linearly scan S_1, and for each (k, v) perform the following lookups in T_2. Note that here, we perform $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Lookup}()$ as described in $\mathsf{MalHT.Lookup}()$, except we do not perform the lookups in the overflow stash $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{SecS}_{2,\mathsf{S}}$.
   - If k is marked as "$\mathtt{OF}$", we pretend that k was in fact found in $\mathsf{OF}_{2,\mathsf{S}}$ and proceed with the remainder of the usual $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Lookup}(k)$ call.
   - If k is marked as "$\mathtt{SecS}$", we pretend that k was in fact found in $\mathsf{SecS}_{2,S}$ and proceed with the remainder of the usual $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Lookup}(k)$ call.
   - If k is not marked, then we pretend that k was not found in either of $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{SecS}_{2,\mathsf{S}}$, and proceed with the rest of the real lookup $(k', v') \leftarrow \mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Lookup}(k)$.
8. Perform $\mathsf{S}_2 \leftarrow \mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Extract}()$ (including $\mathsf{OF}_{2,\mathsf{S}}$ and $\mathsf{SecS}_{2,\mathsf{S}}$, treating the elements marked as $\perp$ in Step 4 as accessed).
9. Run $Z' \leftarrow \mathsf{Intersperse}(S_1 || S_2)$.
10. Run $Z'' \leftarrow \mathsf{TightCompaction}(Z')$ to move all dummy elements to the end. Truncate the array to be of size $2n$.
11. Run $Z \leftarrow \mathsf{IntersperseRD}(Z'')$ to randomly shuffle Z''.
12. Output Z.

Malicious security. Other than the white-box calls to MalHT, every operation is either a linear scan over a portion of memory, or a hybrid call. Therefore, the non-hybrid calls can be time-stamped and thus checked in an online way. We refer to this memory checker as M. For the white-box calls to MalHT, we use online and offline memory checking just as described in MalHT. We refer to this combined memory checker for MalHT as M_{HT}.

Theorem C.41. *Suppose* $\log^9 \lambda \le n \le \mathsf{poly}(\lambda)$. Dedup *is a maliciously secure oblivious implementation of* $\mathcal{F}_{\mathsf{Dedup}}$ *with query complexity* $O(n)$.

Proof. Consider the simulator $\mathcal{S}$ which works as follows in the $(\mathcal{F}_{\mathsf{Shuffle}}, \mathcal{F}_{\mathsf{Intersperse}}, \mathcal{F}_{\mathsf{IntersperseRD}}, \mathcal{F}_{\mathsf{HT}})$-hybrid model: $\mathcal{S}$ simulates the access pattern of Dedup on inputs X_1 and X_2 consisting of only dummies. For every call to $\mathsf{T}_2^{\mathsf{whitebox}}$, we do the following:

- When Dedup calls $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Build}()$, run the code of $\mathsf{MalHT.Build}()$ on $2n$ dummies.
- When Dedup calls $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Lookup}()$, run the code of $\mathsf{MalHT.Lookup}(\perp)$ while skipping the linear scan over the stashes.
- When Dedup calls $\mathsf{T}_2^{\mathsf{whitebox}}.\mathsf{Extract}()$, run the code of $\mathsf{MalHT.Extract}()$ on a secret state of only dummies.

We argue that $\textsc{Real}(\mathcal{C}_{\mathsf{Dedup}}, \mathcal{A})$ and $\textsc{Ideal}(\mathcal{F}_{\mathsf{Dedup}}, \mathcal{S}, \mathcal{A})$ are computationally indistinguishable.

The sequence of hybrids in proving obliviousness and correctness are the same as in Theorem 8.5, with the following modifications:

- We time-stamp all non-hybrid portions of our algorithm that do not deal with $\mathsf{T}_2^{\mathsf{whitebox}}$ or its stashes. We combine this time-stamping with the online memory checker (via time-stamping) as used in Theorem 8.5.

- Similarly, we time-stamp the linear scans of $\mathsf{OF}_{2,\mathsf{S}}, \mathsf{SecS}_{2,\mathsf{S}}$. We also combine this time-stamping with the online memory checker (via time-stamping) as used in Theorem 8.5.
- In Step 7, the combined lookups of the linear scans of the stashes and $\mathsf{T}_2^{\mathsf{whitebox}}$ (without the stashes) look exactly like a $\mathsf{MalHT.Lookup}$ call.
- We invoke the $\mathsf{Intersperse}, \mathsf{TightCompaction}, \mathsf{IntersperseRD}$ hybrid calls to ensure that the output looks randomly shuffled.
- When we replace the output of the algorithm with the output of the real functionality, we invoke the honest-but-curious correctness argument from Theorem 4.1 of Asharov et al. [AKLS21].

Completeness. In the hybrid model, the probability of failure when interacting with an honest server comes only from white-box use of $\mathsf{T}_2^{\mathsf{whitebox}}$ in Step 5 of $\mathsf{MalHT.Build}()$. The probability of aborting is at most $n \cdot e^{-\Omega(\log^5 \lambda)} = \mathsf{negl}(\lambda)$ since $n \leq \mathsf{poly}(\lambda)$.

Efficiency. The run-time of Step 4 is $O(\log \lambda \cdot \log \lambda) = O(\log^2 \lambda)$. The run-time of Step 7 is $O(n) \cdot O(1) = O(n)$ since we are only doing the cuckoo hash lookups of $\mathsf{MalHT.Lookup}()$. Finally, the run-time of Step 8 is also $O(n)$. By plugging in the run-times of MalHT, $\mathsf{Intersperse}$, $\mathsf{PerfectORP}$ and $\mathsf{TightCompaction}$, we get an overall run-time of $O(n)$. □

D Final ORAM Construction

In this section, we give the construction of an optimal ORAM which is maliciously secure, very closely following the construction of Asharov et al. [AKLS21]. The main bottleneck in our warm-up construction was the extra $O(\log \lambda)$ factor incurred from every lookup in $\mathcal{F}_{\mathsf{HT}}$. To avoid this, we combine all the stashes to amortize the lookup time across all the stashes.

D.1 Oblivious Leveled Dictionary

To help amortize the cost of searching the stashes, we follow the construction of Asharov et al. [AKLS21] and use a leveled dictionary. We use this leveled dictionary to find a copy of an index k residing in the smallest "level". It also supports popping elements from a specific level.

Theorem D.2. *Assume the tuple* $(k, \mathsf{level}, \mathsf{whichStash}, \mathsf{data})$ *can be stored in* $O(1)$ *memory words. Suppose* $n \leq \mathsf{poly}(\lambda)$*. Then, there exists a maliciously secure oblivious implementation of* $\mathcal{F}_{\mathsf{LevelDict}}$ *that supports* n *elements such that* $\mathsf{Insert}, \mathsf{Lookup}$*, and* $\mathsf{PopLevel}$ *can be done in* $O(\log^5 n)$ *time in the worst-case.*

The proof of this fact follows similarly to the proof of Theorem C.8.

Functionality D.1 $\mathcal{F}_{\mathsf{LevelDict}}$: Leveled dictionary

$\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{Init}()$:

- **Input:** This operation has no input.
- **The procedure:** Initialize a 2-dimensional list indexed by $(k, \mathsf{level}) \in [K] \times [L]$ for the given key space K and level space L. Initialize all $M[k, \mathsf{level}] = \perp$.
- **Output:** This operation has no output.

$\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{Insert}(k, \mathsf{level}, \mathsf{whichStash}, \mathsf{data})$:

- **Input:** Key k, a level level, flag $\mathsf{whichStash} \in \{\text{“}\texttt{OF}\text{”}, \text{“}\texttt{Stash}\text{”}\}$, and a value data. Note that k can be $\perp$ (i.e., dummy).
- **The procedure:** If $k \neq \perp$, set $M[k, \mathsf{level}] := (\mathsf{whichStash}, \mathsf{data})$.
- **Output:** The Insert operation has no output.

$\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{Lookup}(k)$:

- **Input:** A key k that might be $\perp$ (i.e., dummy).
- **The procedure:**
  - Let level be the smallest index such that $M[k, \mathsf{level}] \neq \perp$.
  - If no such level exists, output $\perp$.
  - Otherwise, set $(\mathsf{whichStash}^*, \mathsf{data}^*) := M[k, \mathsf{level}]$.
- **Output:** The tuple $(\mathsf{level}, \mathsf{whichStash}^*, \mathsf{data}^*)$.

$\mathcal{F}_{\mathsf{LevelDict}}.\mathsf{PopLevel}(\mathsf{level}, \mathsf{whichStash})$:

- **Input:** A level level and a flag $\mathsf{whichStash}$.
- **The procedure:**
  - Let $k \in K$ be the smallest index such that $M[k, \mathsf{level}] = (\mathsf{whichStash}, \cdot)$.
  - If no such k exists, set $\mathsf{data}^* := \perp$.
  - Otherwise, set $\mathsf{data}^* \leftarrow M[k, \mathsf{level}]$ and then set $M[k, \mathsf{level}] := \perp$.
- **Output:** The tuple (k, data^*).

D.2 Algorithm Description

In this section, we present our optimal ORAM construction, with the differences from the warm-up construction boxed and highlighted in purple. At a high level, the main difference is that we now use MalHT in a white-box way and amortize the cost of scanning the stashes by combining all the stashes from all of the levels. This algorithm is directly adapted from [AKLS21].

Structure. The structure of the ORAM is quite similar to the warm-up ORAM, except now we use our MalHT construction in a *white-box* way. In particular, we have access to the overflow stash and the combined stash, $\mathsf{OF_S}$ and $\mathsf{SecS_S}$. We combine all such stashes into leveled dictionaries, and amortize the cost of searching through them over all hash tables. Therefore, instead of paying $O(\log \lambda)$ time per $\mathsf{MalHT.Lookup}()$ call, we search all overflow piles at once. Our setup is as follows:

- Two $\mathcal{F}_{\mathsf{2KeyDict}}$ instances A_ℓ and B_ℓ with capacity $2^{\ell+1}$ elements.

- Four $\mathcal{F}_{\mathsf{LevelDict}}$ instances $\mathsf{Stash}_{\mathsf{C}}^{\mathsf{X}}$ for $\mathsf{C} \in \{\mathsf{A}, \mathsf{B}\}$ and $\mathsf{X} \in \{\mathsf{HF}, \mathsf{F}\}$. This functionality is described in Appendix D.1.
- For each level $i \in \{\ell+1, \ldots, L\}$ contains four *white-box* MalHT instances. We denote the levels as $(\mathsf{A}_{\ell+1}^{\mathsf{HF}}, \ldots, \mathsf{A}_L^{\mathsf{HF}})$, $(\mathsf{A}_{\ell+1}^{\mathsf{F}}, \ldots, \mathsf{A}_L^{\mathsf{F}})$, $(\mathsf{B}_{\ell+1}^{\mathsf{HF}}, \ldots, \mathsf{B}_L^{\mathsf{HF}})$ and $(\mathsf{B}_{\ell+1}^{\mathsf{F}}, \ldots, \mathsf{B}_L^{\mathsf{F}})$.
- Pointers $\mathsf{A}_\ell, \ldots, \mathsf{A}_L$ and $\mathsf{B}_\ell, \ldots, \mathsf{B}_L$, where each A_i points to either $\{\mathsf{A}_i^{\mathsf{HF}}, \mathsf{A}_i^{\mathsf{F}}, \mathtt{Null}\}$ and each B_i points to $\{\mathsf{B}_i^{\mathsf{HF}}, \mathsf{B}_i^{\mathsf{F}}, \mathtt{Null}\}$, where $\mathtt{Null}$ is a null pointer.
- A global counter ctr initialized to 0.

The algorithm will also maintain a running task list calls tasks Tasks. At the end of each access, it will perform part roughly $O(1)$ computation for each task in the list.

Algorithm D.3 Oblivious RAM: $\mathsf{Access}(\mathsf{op}, \mathsf{addr}, \mathsf{data})$. This algorithm is directly adapted from [AKLS21], except we replace CombHT calls with MalHT calls.

Input: $\mathsf{op} \in \{\mathsf{read}, \mathsf{write}\}$, $\mathsf{addr} \in [N]$ and $\mathsf{data} \in \{0,1\}^w$.
Initialization: Initialize $\mathsf{ctr} \leftarrow 0$, and initialize all data structures to be empty.
Secret key: Sample a random PRF secret key sk. Use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{Enc}\text{”}||\cdot)$ for all encryptions, and use $\mathsf{PRF}_{\mathsf{sk}}(\text{“}\mathtt{MAC}\text{”}||\cdot)$ for all MACs.
Authenticated Encryption: For every write query $(\mathsf{write}, \mathsf{addr}, \mathsf{data})$, data is replaced with $\mathsf{data}' := (\mathsf{ct} = \mathsf{Enc}_{\mathsf{sk}}(\mathsf{data}), \sigma = \mathsf{MAC}_{\mathsf{sk}}(\mathsf{ct}, \mathsf{addr}))$. Reads are passed through authenticated decryption, namely unpacking $\mathsf{data}' = (\mathsf{ct}^*, \sigma^*)$, checking $\mathsf{MACVer}_{\mathsf{sk}}((\mathsf{ct}^*, \mathsf{addr}), \sigma^*) = 1$, aborting if verification fails, and otherwise returning $\mathsf{data}^* = \mathsf{Dec}_{\mathsf{sk}}(\mathsf{ct}^*)$.
Memory checking: Note that the algorithm only makes hybrid calls and updates the following $O(1)$ values regularly: $\mathsf{inStash}$ values, $\mathsf{fetched}$ values, found, and data^*. Therefore, the client can simply locally store the variables.
The algorithm:
Lookup:

- Initialize $\mathsf{found} = \mathsf{false}$, $\mathsf{data}^* = \perp$.
- Perform $\mathsf{fetched}_{\mathsf{A}} := \mathsf{A}_\ell.\mathsf{PopKey}(\mathsf{addr})$.
  - If $\mathsf{fetched}_{\mathsf{A}} \neq \perp$, then set $\mathsf{found} = \mathsf{true}$, $\mathsf{data}^* = \mathsf{fetched}_{\mathsf{A}}$, and perform $\mathsf{B}_\ell.\mathsf{PopKey}(\perp)$.
  - Otherwise, perform $\mathsf{fetched}_{\mathsf{B}} := \mathsf{B}_\ell.\mathsf{PopKey}(\mathsf{addr})$. If $\mathsf{fetched}_{\mathsf{B}} \neq \perp$, the set $\mathsf{found} = \mathsf{true}$, and $\mathsf{data}^* = \mathsf{fetched}_{\mathsf{B}}$.
- Perform:
  - $\mathsf{inStash}_{\mathsf{A}}^{\mathsf{HF}} \leftarrow \mathsf{Stash}_{\mathsf{A}}^{\mathsf{HF}}.\mathsf{Lookup}(k)$.
  - $\mathsf{inStash}_{\mathsf{A}}^{\mathsf{F}} \leftarrow \mathsf{Stash}_{\mathsf{A}}^{\mathsf{F}}.\mathsf{Lookup}(k)$.
  - $\mathsf{inStash}_{\mathsf{B}}^{\mathsf{HF}} \leftarrow \mathsf{Stash}_{\mathsf{B}}^{\mathsf{HF}}.\mathsf{Lookup}(k)$.
  - $\mathsf{inStash}_{\mathsf{B}}^{\mathsf{F}} \leftarrow \mathsf{Stash}_{\mathsf{B}}^{\mathsf{F}}.\mathsf{Lookup}(k)$.
- For each $i \in \{\ell+1, \ldots, L\}$ in increasing order:
  - If A_i is $\mathtt{Null}$, let the output of $\mathsf{A}_i.\mathsf{Lookup}()$ be $\perp$.
  - If B_i is $\mathtt{Null}$, let the output of $\mathsf{B}_i.\mathsf{Lookup}()$ be $\perp$.
  - If $\mathsf{found} = \mathsf{false}$:
    - If $\mathsf{A}_i = \mathsf{A}_i^{\mathsf{HF}}$, and $\mathsf{inStash}_{\mathsf{A}}^{\mathsf{HF}}.\mathsf{level} = i$, then set $\mathsf{inStash} = \mathsf{inStash}_{\mathsf{A}}^{\mathsf{HF}}$.
    - If $\mathsf{A}_i = \mathsf{A}_i^{\mathsf{F}}$ and $\mathsf{inStash}_{\mathsf{A}}^{\mathsf{F}}.\mathsf{level} = i$, then set $\mathsf{inStash} = \mathsf{inStash}_{\mathsf{A}}^{\mathsf{F}}$.
    - Otherwise, set $\mathsf{inStash} = \perp$.

* Set $\mathsf{fetched} := \mathsf{A}_i.\mathsf{Lookup}(\mathsf{addr})$ with the following modifications:
  - Do not scan $\mathsf{OF_S}$ or $\mathsf{SecS_S}$.
  - Instead, when the lookup visits $\mathsf{OF_S}$, if $\mathsf{inStash.whichStash} =$ "OF", then use $\mathsf{inStash.data}$ as found in OF. Otherwise, proceed with the lookup as if it was not found in $\mathsf{OF_S}$.
  - Similarly, when the lookup visits $\mathsf{SecS_S}$, if $\mathsf{inStash.whichStash} =$ "SecS", then use $\mathsf{inStash.data}$ as found in SecS. Otherwise, proceed with the lookup as if it was not found in $\mathsf{SecS_S}$.
* If $\mathsf{fetched} \neq \perp$, then set $\mathsf{found} := \mathsf{true}$ and $\mathsf{data}^* := \mathsf{fetched}$.

– Else, perform $\mathsf{A}_i.\mathsf{Lookup}(\perp)$.
– If $\mathsf{found} = \mathsf{false}$:
  * Repeat the same for $\mathsf{B}_i.\mathsf{Lookup}(\mathsf{addr})$ just as in $\mathsf{A}_i.\mathsf{Lookup}(\mathsf{addr})$, while simulating the lookup in the stashes according to $\mathsf{inStash}_\mathsf{B}^\mathsf{HF}$ and $\mathsf{inStash}_\mathsf{B}^\mathsf{F}$.
– Else, perform $\mathsf{B}_i.\mathsf{Lookup}(\perp)$.

Update:

- If $\mathsf{found} = \mathsf{false}$, i.e., this is the first time addr is being accessed, set $\mathsf{data}^* = 0$.
- Let $(k, v) := (\mathsf{addr}, \mathsf{data}^*)$ if this is a read operation; else let $(k, v) := (\mathsf{addr}, \mathsf{data})$.
- Insert (k, v) into A_ℓ and B_ℓ using $\mathsf{Insert}(k, \mathsf{ctr} \pmod{2^{\ell+1}}, v)$.

Rebuild:

- Increment ctr by 1.
- For $i \in \{\ell + 1, \ldots, L\}$:
  – If $\mathsf{ctr} \equiv 0 \mod 2^{i-2}$, then continue to 1-out-of-4 case:

If $\mathsf{ctr} \equiv$	$0 \mod 2^i$	$2^{i-2} \mod 2^i$	$2 \cdot 2^{i-2} \mod 2^i$	$3 \cdot 2^{i-2} \mod 2^i$
Set $\mathsf{A}_i :=$	Null	A_i^HF	Null	A_i^HF.
Set $\mathsf{B}_i :=$	B_i^F	B_i^F	B_i^HF	Null
Start	$\mathsf{RebuildHF}(\mathsf{A}_i^\mathsf{HF})$	$\mathsf{RebuildHF}(\mathsf{B}_i^\mathsf{HF})$	$\mathsf{RebuildF}(\mathsf{A}_i^\mathsf{F})$	$\mathsf{RebuildF}(\mathsf{B}_i^\mathsf{F})$

  Here, starting a task means that we will add the task to the list Tasks.
  – For every task $t \in \mathsf{Tasks}$, execute $t.\mathsf{eachEpoch}$ steps of the task.
  – Return v.

Algorithm D.4 $\mathsf{RebuildF}(\mathsf{C}_i^\mathsf{F})$

Input: The task has input $\mathsf{C}_i^\mathsf{F} \in \{\mathsf{A}_i^\mathsf{F}, \mathsf{B}_i^\mathsf{F}\}$.
Property $\mathsf{eachEpoch}$: The total time allocated to this task is 2^{i-2}.

- If $i = \ell + 1$: Let $W \in O(2^{\ell+1} \cdot \mathsf{poly}(\log \log N))$ bound the work done by this rebuild. Set $\mathsf{eachEpoch} = W/2^{i-2}$.
- If $i > \ell+1$: Let $W \in O(2^i)$ bound the work done by this rebuild algorithm. Set $\mathsf{eachEpoch} = W/2^{i-2}$.

The algorithm:

- If $i = \ell + 1$:

- Run $\mathsf{C}_{i-1}.\mathsf{PopTime}(0, 2^\ell - 1)$ repeatedly for 2^ℓ times. Call the output list X'.
- Obtain $\mathsf{X} \leftarrow \mathsf{Shuffle}(\mathsf{X}')$.

- If $\ell + 2 \le i \le L$: Obtain $\mathsf{X} \leftarrow \mathsf{C}^{\mathsf{F}}_{i-1}.\mathsf{Extract}()$.
- Call $\mathsf{Z} \leftarrow \mathsf{Dedup}(\mathsf{X}, \mathsf{Y})$.
- Create a new table $\mathsf{OF_S}, \mathsf{SecS_S} \leftarrow \mathsf{C}^{\mathsf{F}}_i.\mathsf{Build}(\mathsf{Z})$. Here, we are using MalHT in a white-box way, and accessing the portion of the secret states corresponding to the overflow stash and the combined stash).
- For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{OF_S}$, call $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{“}\mathtt{OF}\text{”}, \mathsf{data})$.
- For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{SecS_S}$, call $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{“}\mathtt{SecS}\text{”}, \mathsf{data})$.

Algorithm D.5 $\mathsf{RebuildHF}(\mathsf{C}^{\mathsf{HF}}_i)$

Input: The task gets as input a table $\mathsf{C}^{\mathsf{HF}}_i \in \{\mathsf{A}^{\mathsf{HF}}_i, \mathsf{B}^{\mathsf{HF}}_i\}$, for some index $i \in \{\ell + 1, \ldots, L\}$.
Property eachEpoch: The total time allocated to this task is 2^{i-2}.

- If $i = \ell + 1$: Let $W \in O(2^{\ell+1} \cdot \mathsf{poly}(\log \log N))$ bound the work done by this rebuild. Set $\mathsf{eachEpoch} = W/2^{i-2}$.
- If $i > \ell+1$: Let $W \in O(2^i)$ bound the work done by this rebuild algorithm. Set $\mathsf{eachEpoch} = W/2^{i-2}$.

The algorithm.

- If $i = L$:
  - For both $j = L - 1$ and $j = L$, for $O(\log \lambda)$ iterations, run $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(j, \text{“}\mathtt{OF}\text{”})$. Call the output list $\mathsf{OF}_{\mathsf{S},j}$.
  - For both $j = L - 1$ and $j = L$, for $O(\log \lambda)$ iterations, run $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(j, \text{“}\mathtt{SecS}\text{”})$. Call the output list $\mathsf{SecS}_{\mathsf{S},j}$.
  - Run $\mathsf{X} \leftarrow \mathsf{C}^{\mathsf{F}}_{L-1}.\mathsf{Extract}()$ and $\mathsf{Y} \leftarrow \mathsf{C}^{\mathsf{F}}_{L}.\mathsf{Extract}()$. Here, we include $\mathsf{OF}_{\mathsf{S},j}$ and $\mathsf{SecS}_{\mathsf{S},j}$ in the extract functionality.
  - Let $\mathsf{Z} \leftarrow \mathsf{Dedup}(\mathsf{X}', \mathsf{Y}')$.
  - Run $\mathsf{OF_S}, \mathsf{SecS_S} \leftarrow \mathsf{C}^{\mathsf{HF}}_{L}.\mathsf{Build}(Z)$.
  - For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{OF_S}$, call $\mathsf{Stash}^{\mathsf{HF}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, L, \text{“}\mathtt{OF}\text{”}, \mathsf{data})$.
  - For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{SecS_S}$, call $\mathsf{Stash}^{\mathsf{HF}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, L, \text{“}\mathtt{SecS}\text{”}, \mathsf{data})$.
- For $\ell + 1 \le i \le L - 1$:
  - If $i = \ell + 1$:
    - Run $\mathsf{C}_\ell.\mathsf{PopTime}(2^\ell, 2^{\ell-1})$ repeatedly for 2^ℓ iterations. Call the output list X.
    - Obtain $\mathsf{X}' \leftarrow \mathsf{Shuffle}(\mathsf{X})$.
  - If $\ell + 2 \le i \le L - 1$:
    - For $O(\log \lambda)$ iterations, run $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(i - 1, \text{“}\mathtt{OF}\text{”})$. Call the output list $\mathsf{OF_S}$.
    - For $O(\log \lambda)$ iterations, run $\mathsf{Stash}^{\mathsf{F}}_{\mathsf{C}}.\mathsf{PopLevel}(i - 1, \text{“}\mathtt{SecS}\text{”})$. Call the output list $\mathsf{SecS_S}$.
    - Let $\mathsf{L} = \mathsf{C}^{\mathsf{F}}_{i-1}.\mathsf{Extract}()$ (including $\mathsf{OF_S}$ and $\mathsf{SecS_S}$).
  - Initialize an array Y of 2^{i-1} dummies.
  - Obtain $\mathsf{Z} \leftarrow \mathsf{IntersperseRD}(\mathsf{X}||\mathsf{Y})$.
  - Create a new table $\mathsf{OF_S}, \mathsf{SecS_S} \leftarrow \mathsf{C}^{\mathsf{HF}}_i.\mathsf{Build}(\mathsf{Z})$.
  - For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{OF_S}$, call $\mathsf{Stash}^{\mathsf{HF}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{“}\mathtt{OF}\text{”}, \mathsf{data})$.
  - For all $(\mathsf{addr}, \mathsf{data}) \in \mathsf{SecS_S}$, call $\mathsf{Stash}^{\mathsf{HF}}_{\mathsf{C}}.\mathsf{Insert}(\mathsf{addr}, i, \text{“}\mathtt{SecS}\text{”}, \mathsf{data})$.

Malicious security. Other than the white-box calls to MalHT, every operation is either a linear scan over a portion of memory, or a hybrid call. Therefore, these can be time-stamped, and checked in an online way. We refer to this memory checker as M. For the white-box calls to MalHT, we use online and offline memory checking just as described in MalHT. We refer to this combined memory checker for MalHT as M_{HT}.

Proof of Theorem 9.5. In the $(\mathcal{F}_{\mathsf{LevelDict}}, \mathcal{F}_{\mathsf{2KeyDict}}, \mathcal{F}_{\mathsf{Shuffle}})$-hybrid model (we treat the MalHT instances as "white-box"), the observed access pattern is a deterministic change in the lookup phase (i.e., skipping the online-checkable linear scan of the stashes in $\mathsf{MalHT.Lookup}()$) from the warm-up construction. Moreover, in the rebuild phases, we first extract the elements from the stashes of the corresponding layers, and then proceed with the rest of $\mathsf{MalHT.Extract}()$ as per usual. Since $\mathsf{MalHT.Lookup}()$ is online checked, the access pattern of $\mathsf{MalHT.Extract}()$ as used in this white-box way is identical to the black-box use of $\mathsf{MalHT.Extract}()$ in the warm-up construction.

Completeness. In the hybrid model, the probability of failure when interacting with an honest server comes only from white-box use of MalHT at each level. The probability of aborting in any of the $O(\log N)$ instances is at most $O(\log N) \cdot N \cdot e^{-\Omega(\log^5 \lambda)} = \mathsf{negl}(\lambda)$ since $N \leq \mathsf{poly}(\lambda)$.

Efficiency. The efficiency analysis of the is algorithm is very similar to that in Theorem 9.1. At a high level, the main difference in the lookup phase are that we additionally scan $O(1)$ leveled-dictionaries, so this adds a factor of $O(\log^5 \log \lambda)$. Also, each MalHT lookup now only takes $O(1)$ time rather than $O(\log \lambda)$ since we skip the linear scan over the stashes. Hence, the lookup phase now takes time $O(\log N + \log^5 \log \lambda)$ rather than $O(\log N \cdot \log \lambda + \log^5 \log \lambda)$.

In each $\mathsf{RebuildF}(\mathsf{C}_i^{\mathsf{F}})$ call, we add $O(\log \lambda)$ insert calls to the leveled dictionary, which each take $O(\log^5 \log \lambda)$. Asymptotically, this does not increase the run-time of $\mathsf{RebuildF}$ by more than an $O(1)$ multiplicative factor.

In $\mathsf{RebuildHF}(\mathsf{C}_i^{\mathsf{HF}})$, we additionally also call $\mathsf{PopLevel}$ to the leveled dictionary $O(\log \lambda)$ time. This also adds a factor of $O(\log \lambda \cdot \log^5 \log \lambda)$, which does not increase the run-time of $\mathsf{RebuildHF}$ by more than an $O(1)$ multiplicative factor. □

D.3 Reducing Client Space

Here, we show how to implement our ORAM while using only $O(1)$ words of local storage. A priori, this is not the case for a few reasons:

1. Concurrent composition (Theorem A.4) stores a different PRF key for each hybrid being invoked.
2. Concurrent composition (Theorem A.4) adds up the space complexities of each implementation.

Thankfully, both of these issues can be remedied by the fact that the timing of which hybrids are called and when is computable in low space given only the global counter (note that this is not true for OptORAMa, see Remark D.6). Item 1 can immediately be salvaged by Remark A.6 as

the timing of when we call hybrids is always easily computable given the current global counter. For Item 2, first notice that each implementation on its own when viewed as a node in the hybrid dependency tree has $O(1)$ space, ignoring PRF keys. Second, since the schedule of hybrid calls is easily computable, the PRF keys can be consolidated into one global PRF key easily.

Lastly, we explain how to effectively use the untrusted server as a way to store what would be the local space of each of these implementations (specifically, using encryption and authentication). Consider some node in the hybrid dependency tree that has $O(1)$ space on its own. This node will have $O(1)$ dedicated server space to encrypt and authenticate its local space contents. When starting or resuming a computation, this node will download these $O(1)$ words from the server (assuming authentication passes), perform the computation, and then re-encrypt and authenticate its space contents to the server. Because the hybrid call scheduling is easily computable given the current counter, these authentications can be time-stamped, ensuring that there are no possible replay attacks. Moreover, IND-CPA security guarantees that nothing about the space contents is revealed to the adversary. Since there are $o(n)$ total nodes in the hybrid dependency tree, the total amount of additional server storage this would require is $o(n)$.

Remark D.6. *The hybrid dependency tree for OptORAMa [AKL+20] cannot be determined in low space because the* SmallHT *hybrid calls are in an arbitrary order. However, achieving $O(1)$ overall space complexity is still possible in their setting because the adversary cannot tamper with the server.*