Snapshot-Oblivious RAMs: Sub-Logarithmic Efficiency for Short Transcripts

Yang Du[1], Daniel Genkin[2], and Paul Grubbs[3]

[1]University of Michigan, duyung@umich.edu
[2]Georgia Tech, genkin@gatech.edu
[3]University of Michigan, paulgrub@umich.edu

Abstract. Oblivious RAM (ORAM) is a powerful technique to prevent harmful data breaches. Despite tremendous progress in improving the concrete performance of ORAM, it remains too slow for use in many practical settings; recent breakthroughs in lower bounds indicate this inefficiency is inherent for ORAM and even some natural relaxations.

This work introduces snapshot-oblivious RAMs, a new secure memory access primitive. Snapshot-oblivious RAMs bypass lower bounds by providing security only for transcripts whose length (call it c) is fixed and known ahead of time. Intuitively, snapshot-oblivious RAMs provide strong security for attacks of short duration, such as the snapshot attacks targeted by many encrypted databases.

We give an ORAM-style definition of this new primitive, and present several constructions. The underlying design principle of our constructions is to store the history of recent operations in a data structure that can be accessed obliviously. We instantiate this paradigm with data structures that remain on the client, giving a snapshot-oblivious RAM with constant bandwidth overhead. We also show how these data structures can be stored on the server and accessed using oblivious memory primitives. Our most efficient instantiation achieves $\mathcal{O}(\log c)$ bandwidth overhead. By extending recent ORAM lower bounds, we show this performance is asymptotically optimal. Along the way, we define a new *hash queue* data structure—essentially, a dictionary whose elements can be modified in a first-in-first-out fashion—which may be of independent interest.

1 Introduction

Users of cloud computing services trust providers to store sensitive data. Encryption can protect the data itself, but cannot prevent information from being disclosed by attacks on metadata like the memory access patterns. A long line of work has conclusively demonstrated that access pattern attacks can be used to reveal sensitive information. In some settings, access patterns alone can be used to completely decrypt data [34,32,9,22,28,23,24,37,14,36].

Oblivious RAM (ORAM) is a technique that can hide memory access patterns and therefore prevent these kinds of harmful attacks. ORAM is quite useful, but its strong security guarantees come at a cost, both asymptotic and concrete. With the best known constructions [4] achieving $\mathcal{O}(\log n)$ overhead for an n-entry memory, and with a matching $\Omega(\log n)$ lower bound by [38], it seems impossible to have an ORAM scheme where the cost of each memory access does not depend on the total memory size.

Unfortunately, even relaxing security requirements does not allow bypassing the $\Omega(\log n)$ lower bound. Indeed, similar lower-bounds have been shown for differentially oblivious RAMs [42], or even when the memory access pattern is known ahead of time [6,19]. The attempt to gain efficiency in various settings has led to primitives such as structured/searchable encryption [13,48,11], which allows for fast database lookup at the cost of allowing attacks in some settings [9]. Alternatively, prior works have assumed the a-priori knowledge of a certain distribution of memory accesses [21], or provided an ORAM-based mechanisms for adjusting searchable encryption leakage [15].

Motivated by the goal of securing worst-case memory access patterns without dependence on the size of the entire memory, in this paper we tackle the following question:

How can we sidestep the $\Omega(\log n)$ lower bound, while providing a meaningful and general security guarantee for memory access patterns?

1.1 Our Contributions

We begin with the observation that many attacks on real systems follow a common pattern: an attacker gains access to an already-running system, is present in the system for a relatively short time, then either leaves or loses access because the attack was detected. The Verizon Data Breach Incident Report (DBIR) underscores the commonality of these kinds of attacks: for example, in 2021 it found nearly five thousand incidents of "Basic Web Application Attacks", simple attacks in which an attacker compromises the web application and quickly performs only a few actions, such as downloading emails. DBIR also found that roughly 50% of detected security incidents were detected within a few days [1]. A limiting case of this model is the so-called "snapshot" threat model targeted by many encrypted databases, where the attacker obtains only a one-time snapshot of the database system, giving it only the currently-running queries [23].

Thus, for encrypted memory primitives it makes sense to consider an attack model where the attacker sees only a "window" of memory access patterns of bounded size; however, the attacker cannot see the system's memory access pattern before the attack began, nor can it see the access pattern after the attack has concluded. Thus, we define the notion of c-Snapshot ORAM, which maintains ORAM-like security guarantees but against a weaker adversary which is limited to observing only c memory operations.

Definition 1 (informal). *We say a RAM emulator RE is c-snapshot oblivious in case the following holds. For any two sequences of operations $\overrightarrow{op}^1, \overrightarrow{op}^2$ of the same length, and for any subsequences of c operations: $\overrightarrow{op}^1_c \subseteq \overrightarrow{op}^1$, $\overrightarrow{op}^2_c \subseteq \overrightarrow{op}^2$, it holds that the access patterns seen while executing $\overrightarrow{op}^1_c$ and $\overrightarrow{op}^2_c$ are computationally indistinguishable.*

Next, with Definition 1 in hand, we then present our first c-Snapshot ORAM construction where the client's overhead is polylogarithmic in c but independent of n. More formally,

Theorem 1 (informal). *There exists a c-snapshot oblivious RAM emulator with $\mathcal{O}(\log^2 c)$ bandwidth overhead, using $\tilde{\mathcal{O}}(\log c)$ client storage.*

In particular, Theorem 1 offers the "best of both words" ORAM construction, as the client obtains a meaningful security guarantee against realistic adversaries while having its overhead not depend on n. Next, we proceed to reduce the client's storage to constant,

while maintaining polylogarithmic (in c) overhead for the server. We achieve this in the amortized setting. See Theorem 2 below.

Theorem 2 (informal). *There exists a c-snapshot oblivious RAM emulator with $\mathcal{O}(\log c)$ amortized bandwidth overhead, using constant client storage.*

Finally, we proceed to find the lower bound for c-snapshot ORAMs. Here, we show that any c-snapshot secure construction with constant storage must have an $\Omega(\log c)$ amortized bandwidth overhead. In particular, this makes the construction in Theorem 2 asymptotically optimal.

Theorem 3. *Any c-snapshot oblivious RAM emulator using constant client storage, must have a lower bound of $\Omega(\log c)$ amortized bandwidth overhead.*

1.2 Technical Overview

Motivated by the challenge of bypassing the ORAM lower bound while still providing meaningful security guarantees in a natural setting, in Section 3 we begin by presenting our definition of a c-snapshot ORAM. Our aim is to provide security against an adversary that is capable of only seeing a window of at most c operations. We formalize this with an IND-CPA style game in which the adversary needs to distinguish which of two chosen transcripts were executed, given only the access patterns of the last c operations and the state of the memory before these operations. We also prove our definition has several desirable properties: notably, c-snapshot obliviousness implies security for smaller snapshots as well.

In this paper, we do not assume any encryption on the memory content and let adversary only see the accessed address. In practice, we can either use a standard "read, re-encrypt, write back" paradigm, or secret-sharing under multi-party setting.

A folklore 1-snapshot oblivious scheme. With the definition of c-snapshot ORAM in hand, we proceed to analyze a folklore RAM emulator which simply permutes memory addresses using a PRP, while hiding the operation type by performing a read and a write for both operation types. As we show in Section 4, this results in a 1-snapshot ORAM, as the adversary only sees an access to a single pseudorandom memory location.

Getting $c > 1$. Moving to the more general goal of c-snapshot obliviousness, we proceed to hide repeated accesses to the same memory locations by the client using a size-c queue. More specifically, we ask the ORAM client to maintain a queue of size $\mathcal{O}(c)$, which intuitively acts as a cache for the last c accesses. While addresses the are not present in the queue are fetched from the server's memory, we access a dummy element in case the address is present. Notably, as the attacker only sees a window of c, we do not need to re-shuffle, as any eviction of the queue is guaranteed to be touching an address which was last accessed more than c operations ago. This ensures that any address is accessed at most once in every size-c window, intuitively mimicking the 1-snapshot ORAM construction. See Section 5 for details.

Achieving polylogarithmic storage. Our next step is to reduce the storage required by the client from $O(c)$ to $\mathsf{polylog}(c)$. An intuitive approach will be to recursively delegate the client's storage to the server using an oblivious RAM. Because storage complexity of the construction in Section 5 in linear in c, such a recursive composition will result in reducing the client's storage overhead.

In Sections 6 and 7.1 we present different constructions using a custom data structure we call an Oblivious Hash Queue (OHQ). More specifically, we begin by observing that obliviously delegating the client's queue to the server is simpler than general ORAM, as the queue only supports a limited set of operations. By efficiently solving the oblivious queue delegation problem, in Section 6 we are able to obtain c-snapshot oblivious construction with $\mathcal{O}(\log^2 c)$ bandwidth overhead, using $\tilde{\mathcal{O}}(\log c)$ client storage. Further refining our OHQ technique, in Section 7.1 we obtain a construction with $\mathcal{O}(\log c)$ amortized bandwidth and constant client overhead, albeit with a worse concrete efficiency compared to the construction in Section 6.

A matching lower bound. Directly following from Larsen and Nielsen lower bound [38], in Section 7.3 we show a lower bound for obtaining c-snapshot ORAM, proving that every secure construction must have an $\Omega(\log c)$ amortized bandwidth overhead. We reuse Larsen and Nielsen's result in the c-snapshot security setting. This essentially proves the asymptotic optimality of the construction in Section 7.1, limiting future improvements to lower order terms.

1.3 Related Work

ORAM. There are two kinds of oblivious RAM: hierarchical ORAM, initially proposed in [19] and following works [19,35,43,20,41,4], and tree based ORAM, proposed by Shi et al. in [47] and followed by [47,17,12,49,44,51]. Computationally secure ORAM is optimized by [4] with an amortized bandwidth overhead of $\mathcal{O}(\log n)$, and de-amortized by [5]. These above ORAM constructions satisfies the most strict security definition (see Section 2.2). ORAM can be more efficient if it is designed for a specific usage, such as oblivious data structure [52] and zero-knowledge ORAM [26,27].

Variants of the basic ORAM model include the offline setting and the balls-in-bins model. Boyle and Naor [6] showed how to construct an ORAM scheme in the offline setting. Jafargholi et al. [30] gave a statistically secure offline ORAM with $\Omega(\log n)$ overhead, using an oblivious priority queue. Read only ORAM [53] supports only read operation in the online setting. If we remove the ball-in-bin model, ORAM efficiency can be enhanced given server computation ability [2,16,40,25]. Differentially private ORAM [50] further weakens the security requirement by requiring that the access patterns of adjacent transcripts (vs. any two transcripts) are statistically close.

Structured encryption. Most of searchable encryption and structured encryption schemes [31,18,21,15] assumes a fully persistent adversary. But there are works assuming non-persistent adversary such as [3]. In their setting, adversary is only observing snapshots of database but not access pattern of queries. A line of works on leakage suppression [31] uses a cache to store most recent accessed queries, and retrieve from cache if queried again. However this does not allow writing things back to main memory unless a rebuild, which incurs an amortized $\Omega(\log n)$ overhead. Our schemes (Section 5, 6) allow writes back to main memory because we require security to hold only for a short operation sequence. A follow-up on leakage suppression [18] allows addition and deletion of keys in a multimap.

A recent line of work has studied intermediate security for persistent adversaries that is stronger than typical structured encryption but weaker than ORAM. For example, Pancake [21] shows how to do efficient key-value lookups with access pattern hiding in

a setting where the distribution of queries is known a priori, and queries are independent. SEAL [15] combines structured encryption with ORAM, allowing more fine-grained tradeoffs between access pattern hiding (against persistent adversaries) and efficiency.

Lower bounds. Larsen and Nielsen [38] gave the first cell-probe lower bound $\Omega(\log n)$ for online ORAM, answering the question asked in [6], which also reduced lower bound for offline ORAM to sorting circuits. A follow-up work from Jacob et al. [29] gave lower bounds for oblivious data structures. A recent work [33] generalized the overhead to both online and offline ORAM. Weiss and Wichs [53] showed lower bound for read only online ORAM, and Persiano and Yeo [42] gave a $\Omega(n)$ lower bound for differentially private RAM. Larsen et al. [39] gave an ORAM lower bound under multi-server setting. Recently, Patel et al. showed there is an inherent inefficiency in encrypted multi-maps with even decoupled key-equality pattern leakage, which leads to a $\Omega(\log n)$ overhead in the leakage cell probe model. Cash et al. gave lower bound for one-round ORAM [8], which requires either $\Omega(\sqrt{N})$ bandwidth overhead or $\Omega(\sqrt{N})$ client storage. Our snapshot oblivious RAM (Section 5.2) is also one-round but has constant overhead and needs $\Theta(c)$ client storage.

2 Preliminaries

2.1 Pseudorandom Permutation

Definition 2. ***(Pseudorandom permutation)*** *A Pseudorandom permutation (PRP) is a function family $E : \mathcal{K} \times \{0,1\}^n \to \{0,1\}^n$. We define the PRP security game $\mathbf{PRP}(E, \mathcal{A}, i)$. First, a key k is randomly generated from $\mathcal{K}$ and a random permutation π is randomly generated from all permutations of n elements $\mathsf{Perms}(n)$. The adversary has access to an oracle $\mathcal{O}_i^k$. When the adversary queries a string s, it receives either $E_k(s)$ in the case $i = 0$ or $\pi(s)$ in the case $i = 1$. Finally the adversary outputs a bit b. We say that E is a secure PRP if for all nuPPT adversaries $\mathcal{A}$ playing the PRP security game.*

$$\mathbf{Adv}_E^{prp}(\mathcal{A}) = \big|\Pr[\mathbf{PRP}(E, \mathcal{A}, 0) = 1] - \Pr[\mathbf{PRP}(E, \mathcal{A}, 1) = 1]\big|,$$

the advantage defined above is negligible.

2.2 ORAM

In this section, we describe the syntax of our execution model and RAM emulator. We then proceed to define the correctness requirements of RAM emulators, as well as their obliviousness security definitions.

Execution model and terminology. We define a random access memory (e.g., RAM) DB to be an array of M entries, where each entry contains at least $m \geq \lceil \log M \rceil$-bits. We define an operation to be a tuple (op, idx, val) where op is either read or write, idx is an integer between 0 and $M - 1$, and val is either a bit string of length m or the $\perp$ symbol. Finally, we define a transcript to be a sequence of operations.

A note on "blocks". Many works on ORAM [49,41,4] additionally define a "block" of memory to be a sequence of memory locations that can be accessed with unit cost. While we do not use blocks in this paper, and for simplicity assume that one operation

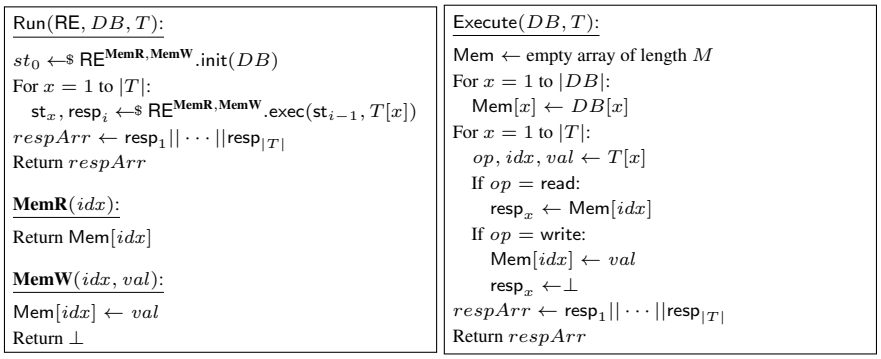

Fig. 1. RAM emulator correctness.

only reads or writes to a single memory location, we do note that our results can be easily extended to account for block memory accesses.

RAM emulators. A RAM emulator RE is a pair of algorithms $(\mathsf{init}, \mathsf{exec})$ that simulates a RAM. Both init and exec have oracle access to two procedures — **MemR** and **MemW** — that allow reading and writing to an array Mem of size M. Below, we will mostly leave implicit the length of each array entry, and simply assume they are large enough. (To draw an analogy to encrypted databases, RE is the "client" and the array Mem it reads and writes through its oracles is the "server".)

The randomized initialization procedure $\mathsf{RE}.\mathsf{init}(DB)$ takes an array of size N where each input is m bits long, representing the initial state of the memory, as input. It outputs an initial state st_0. The randomized execute procedure $\mathsf{RE}.\mathsf{exec}(\mathsf{st}, (op, idx, val))$ takes as input a state st and an operation. It executes the operation and outputs the result and a new state. (Below, in cases where the result is not used, we will omit it.)

Access pattern. We define an access pattern of an emulator RE on an array DB and transcript T to be the sequence of **MemR** and **MemW** oracle calls, and the first argument (accessed index) made by RE during init and while calling exec on each operation in the transcript. As an abuse of notation, we will sometimes use $\mathsf{RE}(DB, T)$ to refer to the access pattern corresponding to executing the operations in T on DB.

Correctness and efficiency. Intuitively, a RAM emulator RE should always return the same results as the "canonical" RAM implementation Execute outlined in Figure 1 (right). More formally, for a RAM emulator RE we define correctness using the pseudocode in Figure 1 (left). That is, we say that RE is correct if for any database DB and transcript T, the output of $\mathsf{Run}(\mathsf{RE}, DB, T)$ is equal to the output of $\mathsf{Execute}(DB, T)$ with probability 1 over the random choices made during init and exec.

Bandwidth overhead. One of the main measures of efficiency for RAM emulators is bandwidth overhead, namely the increase in memory usage compared to the baseline of just executing the transcript directly. Formally, for an emulator RE, database DB, and transcript T, we define the bandwidth overhead as $\mathsf{Ex}[|\mathsf{RE}(DB, T)|/|T|]$ where the expectation is taken over the randomness of RE.

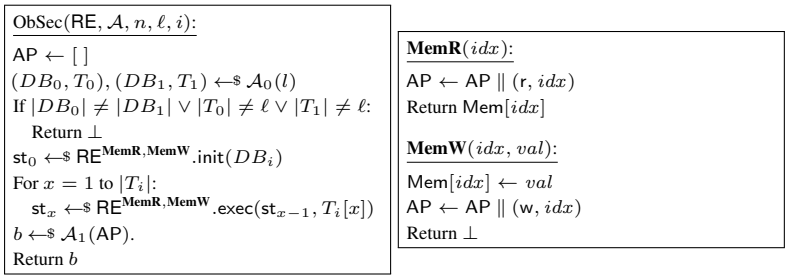

Fig. 2. ORAM security game definition in pseudocode.

Oblivious RAM Emulators. Next we define the notion of obliviousness for RAM emulators, see Figure 2. In this pseudocode, the adversary has two stages. The first stage adversary $\mathcal{A}_0$ chooses the arrays (databases) DB_0, DB_1 and the transcripts T_0, T_1. Next, the second stage adversary $\mathcal{A}_1$ tries to guess the bit b. We note that $\mathcal{A}_1$ is not given access to the contents of memory: all its input AP contains is the memory address accessed by each oracle call, and its type (r or w). This is make the definition agnostic to the way the memory contents are hidden—i.e., our definition can just as easily apply to a setting where the memory is encrypted as it can to one where RE is run in multi-party computation.

Definition 3. *(Oblivious RAM emulator security) We define the ObSec advantage of an adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ against RAM emulator RE as*

$$\mathbf{Adv}^{obl}_{\mathsf{RE}}(\mathcal{A}) = \left| \Pr[\mathit{ObSec}(\mathsf{RE}, \mathcal{A}, n, \ell, 0) = 1] - \Pr[\mathit{ObSec}(\mathsf{RE}, \mathcal{A}, n, \ell, 1) = 1] \right| .$$

We say the RAM emulator RE is computationally oblivious if for any nuPPT adversary $\mathcal{A}$, $\mathbf{Adv}^{obl}_{\mathsf{RE}}(\mathcal{A}) = \mathrm{negl}(n)$.

Semi-honest security. Finally, we note that because **MemR** and **MemW** read and write Mem, neither these ORAM definitions capture servers that modify memory contents or reply with stale values. Such attacks can be prevented using standard techniques [45].

2.3 Oblivious Maps

Below, we will use oblivious maps, which are oblivious data structures akin to ORAM but tailored for specific operation types (less generic than memory read/write).

As proposed in [52], we give oblivious map the following syntax. An oblivious map OM has an initialize function $\mathsf{OM.init}(N)$ which takes N as the maximum capacity and outputs an initial state. As with ORAMs, we view oblivious maps as having oracle access to **MemR** and **MemW** oracles to manipulate their memory. OM has an execution function that supports four operations: Find, Insert, Update, Delete. $\mathsf{OM.Find}(key)$ returns the value associated to key. $\mathsf{OM.Insert}(key, val)$ inserts the key value pair in to the map. $\mathsf{OM.Update}(key, val)$ replaces the value associated to key by val. $\mathsf{OM.Delete}(key)$ deletes the key value pair whose key is key. The execute function additionally inputs and outputs a state.

We require oblivious maps to satisfy a variant of the ORAM security definition defined above. Let OblivMapSec denote the security game. (We omit pseudocode since it is almost identical to ObSec.)

Definition 4. *(Oblivious map) We define the advantage of $\mathcal{A}$ against* OM *as*

$$\mathbf{Adv}^{\mathit{OMap}}_{\mathsf{OM}}(\mathcal{A}) = \big|\Pr[\mathit{OblivMapSec}(\mathsf{OM}, \mathcal{A}, N, \ell, 0) = 1] - \Pr[\mathit{OblivMapSec}(\mathsf{OM}, \mathcal{A}, N, \ell, 1) = 1]\big|.$$

If this advantage is negligible for all nuPPT adversaries, we say OM *is an oblivious map.*

3 Snapshot-Oblivious RAM Emulators

In this section, we introduce our new primitive: c-snapshot oblivious RAM emulators. (We will usually shorten this to c-snapshot ORAMs.) The syntax of the new primitive is similar to ORAM, but with one important change: we allow the init procedure to take, in addition to the initial array DB, a natural number c denoting the number of operations' access patterns the adversary gets to see. The syntax is otherwise unchanged. The correctness notion for RAM emulators must change slightly as well: for a RAM emulator to be correct, the correctness condition defined in Section 2 must hold with probability 1 for every possible choice of c.

c-snapshot obliviousness. Next we explain our new security notion, c-snapshot obliviousness. Before formally stating the definition, we will briefly discuss the space of possible definitions, and identify some desirable properties of a snapshot-obliviousness definition. First, we expect snapshot-obliviousness should be *strictly* weaker than plain obliviousness. Namely, any ORAM should be c-snapshot oblivious for any c. Second, for any $c' < c$, it should be the case that c-snapshot obliviousness implies c'-snapshot obliviousness. Finally, to meaningfully capture snapshot attacks on real systems, we would like snapshot-obliviousness to allow the adversary to see any c operations of its choosing, without restricting the adversary to any particular locations.

Our definition. We give the pseudocode of our definition in Figure 3. Like plain obliviousness, the definition allows the adversary to specify two pairs of an array and transcript. The game runs $\mathsf{RE}.\mathsf{init}$ on the ith pair using the oracles **MRH** and **MWH**, which allow the emulator to manipulate the memory Mem without recording the access patterns. Then the game runs $\mathsf{RE}.\mathsf{exec}$ on all but the last c operations of T_i, again without recording the access patterns. Next, the game proceeds to execute final c operations of the transcript via $\mathsf{RE}.\mathsf{exec}$, but this time using **MemR** and **MemW** which record their access patterns in AP. Finally, the game runs the second adversary $\mathcal{A}_1$ on the recorded access patterns AP, and (implicitly) the state of $\mathcal{A}_0$. $\mathcal{A}_1$ in turn is expected to correctly guess i.

Definition 5. ***(c-snapshot obliviousness)*** *Let RE be a RAM emulator and c be a fixed number, the c-SnapObSec advantage of the adversary $\mathcal{A} = (\mathcal{A}_0, \mathcal{A}_1)$ against RE is*

$$\mathbf{Adv}^{\mathit{snap}}_{\mathsf{RE}}(\mathcal{A}) = \big|\Pr[\mathit{SnapObSec}(\mathsf{RE}, \mathcal{A}, n, c, 0) = 1] - \Pr[\mathit{SnapObSec}(\mathsf{RE}, \mathcal{A}, n, c, 1) = 1]\big| .$$

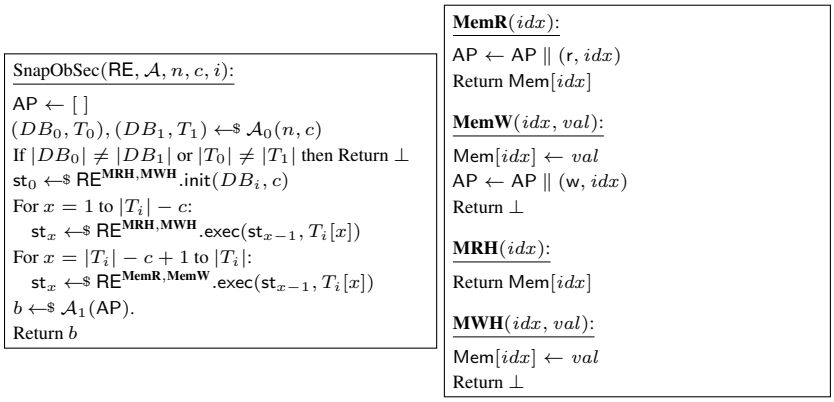

Fig. 3. SnapORAM security game.

The emulator RE *is said to be (computationally) c-snapshot oblivious if for any nuPPP adversary $\mathcal{A}$,* $\mathbf{Adv}^{snap}_{\mathsf{RE}}(\mathcal{A}) = \mathrm{negl}(n)$.

Comparing to obliviousness. We now argue that our c-snapshot obliviousness definition is a natural restriction of regular ORAM. In particular, if for a RAM emulator RE there exists a c and an adversary $\mathcal{A}$ with non-negligible c-SnapObSec advantage, we can build a reduction $\mathcal{B} = (\mathcal{B}_0, \mathcal{B}_1)$ that breaks ORAM security. The reduction $\mathcal{B}_0$ works by running $\mathcal{A}_0$ (with c as an argument) and outputting the two pairs it outputs. Then, $\mathcal{B}_1$ uses its access patterns AP to construct $\mathcal{A}_1$'s inputs. (Note that $\mathcal{A}_1$ takes the initial state of the memory Mem_0 as well as the access patterns of the last c operations; $\mathcal{B}_1$ can construct both with AP. Clearly, $\mathcal{A}$'s c-SnapObSec advantage is a lower bound on $\mathcal{B}$'s ORAM advantage.

Requiring equal length transcripts. In the SnapObSec game, as in ObSec above, we require the adversary to output two equal length transcripts. This restriction is necessary in ObSec to prevent a trivial distinguishing attack based on the transcript length. However, astute readers may notice that since an adversary can only view the access pattern of c operations, specifying two differing-length transcripts does not give a SnapObSec adversary a trivial win. The c-snapshot obliviousness definition could conceivably be strengthened by removing the restriction that the transcripts are of equal length. However, the security analyses of some c-snapshot ORAM constructions below —e.g., UHQoram in Section 7—would require a non-standard transcript-length-hiding property of an underlying ORAM. Lifting the length restriction is a good question for future work.

Observing the last c operations. Our c-snapshot obliviousness definition allows the adversary to design the whole transcript but restricts the observing window to be the last c operations at the end of the transcript. We claim this setting is as strong as allowing to put the observing window anywhere in the middle of the transcript. For a typical ORAM not handling batching transcripts, the way to access one physical memory position, though randomized, does not depend on the remaining transcripts after that. This means any operation after the observing window will not change the distribution of access

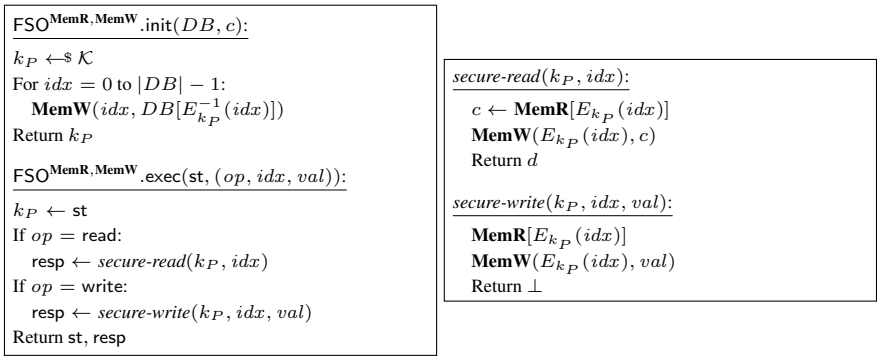

Fig. 4. The FSO RAM emulator, and the definition of *secure-read* and *secure-write*. (Note that both *secure-read* and *secure-write* implicitly have access to the same oracles as exec.)

patterns the adversary gets. Due to this independence, it is without loss of generality to put the c accesses at the end of the transcript.

c'-snapshot obliviousness for $c' < c$. The security definition immediately leads to a result that any snapshot-oblivious RAM emulator initialized with a DB and some number c is still secure if the adversary observes access pattern of c' operations and $c' < c$. We note, however, that this is different from saying any c-snapshot oblivious RAM emulator is c'-snapshot oblivious: this statement is not necessarily even correct. In SnapObSec game, the RE is initialized by a parameter c, so an adversary against a c-snapshot oblivious RAM emulator is getting access pattern from a RE is initialized by c. However, proving this would require building a reduction that wins the c-snapshot game given an adversary that wins the c'-snapshot game, and it's not clear if the adversary can simulate the view of a c'-snapshot adversary given its inputs (computed from a c-snapshot ORAM initialized with c fixed). We believe that for restricted classes of snapshot-oblivious RAMs, this statement is true, but we leave the details to future work.

4 FSO: A 1-Snapshot Oblivious RAM

Next we will give a "warm-up" analysis of a folklore snapshot-oblivious RAM, FSO, and show that it meets 1-snapshot obliviousness.

The scheme. In Figure 4, we give the pseudocode of FSO. It uses a pseudorandom permutation E. During init, FSO samples a PRP key, then loads the array into memory according to the permutation E. (The parameter c is ignored during init.) Then, it outputs the keys as its initial state.

During exec, the scheme performs either *secure-read* or *secure-write* depending on op. Both perform a *writeback* to hide the operation type: they first read index $E_{k_P}(idx)$ with **MemR**, and write it back to the same location with **MemW**. If the operation was a read, exec returns the value, else it returns nothing. Clearly, this scheme has both constant bandwidth overhead and constant client storage.

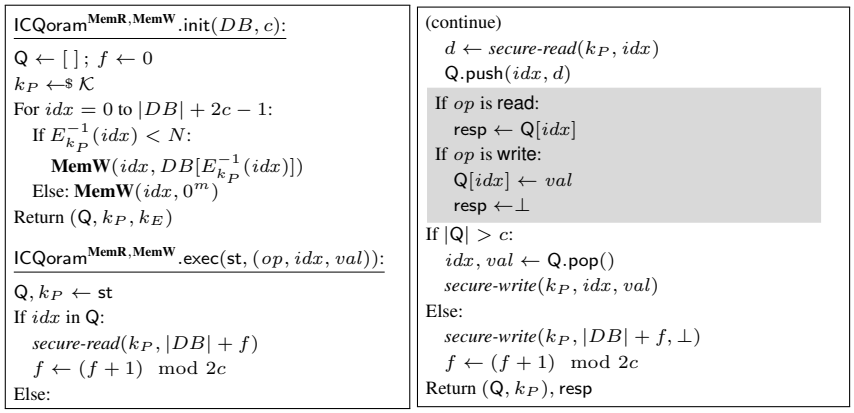

Fig. 5. ICQoram, an insecure queue-based scheme. The *secure-read* and *secure-write* procedures are as defined in Figure 4. Three stages are in execution function, the second one is shaded.

Security of FSO**.** The security of FSO for restricted adversaries seems to be folklore—see, e.g., Cash [7]— but to our knowledge has never been formally proven. We validate this folklore by showing FSO is c-snapshot oblivious for $c = 1$.

Theorem 4. *If E is a secure PRP, then* FSO *RAM emulator is 1-snapshot oblivious.*

Proof. We define G_0 to be the case that FSO initializes on DB_0 and executes on T_0. In G_1 FSO initializes on DB_1 and executes on T_1. We want to show that both G_0 and G_1 are indistinguishable from G_{hybrid} where the adversary observes read and write a same but random idx in the access pattern.

In $G_0, G_1, G_{\mathsf{hybrid}}$, the adversary observes $\mathsf{AP} = (\mathsf{r}, idx')||(\mathsf{w}, idx')$. The first part of AP comes from *secure-read* and the second part comes from *secure-write*.

The difference between G_i and G_{hybrid} is that the idx' in AP is $E_{k_P}(idx)$ in G_i, which is computed by a PRP; while in G_{hybrid}, it is truly random, or we can say it is from a random permutation π, $idx' = \pi(idx)$ for fixed idx. If G_i and G_{hybrid} is distinguishable, we can tell difference between PRP and truly random permutation by a simple reduction, $|\Pr[G_{i,\mathsf{hybrid}} = 1] - \Pr[G_{\mathsf{hybrid}} = 1]| \leq \mathbf{Adv}^{\mathrm{prp}}_E(\mathcal{C}_i)$. Therefore, by the 2-step reduction, $|\Pr[G_0 = 1] - \Pr[G_1 = 1]| \leq 2\mathbf{Adv}^{\mathrm{prp}}_E(\mathcal{C})$.

5 The c-queue Scheme

In the previous section, we showed a simple c-snapshot oblivious RAM. In this section we will show how to get $c > 1$. Before giving our construction, we will describe a natural approach that turns out to be insecure.

5.1 An Insecure Scheme

The FSO scheme in Section 4 is only 1-snapshot obliviousness because it leaks repetitions in accesses: reading the same "logical" address twice causes the scheme to make the

same physical accesses. To make a secure scheme for general c we'd like the property that physical accesses are all distinct whether or not logical accesses are.

A natural way to ensure this is to augment FSO with a queue of recent accesses. It keeps track of which entries were accessed in the last c operations, along with their values. If any recently-accessed entries are accessed again within c operations, the scheme reads them from the queue instead of from the remote memory. To prevent the server from learning if the queue was used, the scheme can access a fake element.

The ICQoram scheme in Figure 5 formalizes this idea. ICQoram.init works as in FSO, except it also adds $2c$ dummy elements. The procedure ICQoram.exec has three stages. First, it fetches address idx to the queue Q. If idx is already in the queue, it fetches a dummy element, otherwise reads idx into the queue. Second, it processes the operation (op, idx, val). If the operation is a write, it updates the value of idx in the queue; else, it stores val as the read's return value. Finally, ICQoram performs eviction. If the size of queue is greater than c, it writes the oldest element back to main memory, otherwise it writes a dummy element.

This scheme is fairly efficient: it requires $\mathcal{O}(c)$ additional storage in physical memory, $\mathcal{O}(c)$ additional client state, and has constant bandwidth overhead.

Security. The access pattern for each operation is one *secure-read* and one *secure-write*. If ICQoram could guarantee that for any c operations, the indices touched in the $2c$ *secure-read*s and *secure-write*s were different, it could be proven secure using a straightforward extension of the proof for FSO in Section 4.

However, this guarantee does not hold. ICQoram only makes sure the c *secure-read* have distinct indices; the c *secure-write* indices depend on what is residing in the queue in a way that can be exploited by an attacker to distinguish between two transcripts. We demonstrate this with a concrete example. (We remind the reader that although the attacker can only observe the access pattern of c operations, it can choose the entire transcript.) Let $c = 3$, $|DB| = 10$, and take the two transcripts

$$T_0 = \mathsf{read}(1), \mathsf{read}(2), \mathsf{read}(3), \mathsf{read}(4), \mathsf{read}(5),$$
$$T_1 = \mathsf{read}(1), \mathsf{read}(2), \mathsf{read}(3), \mathsf{read}(4), \mathsf{read}(1)\ .$$

At the end of the third operation, for both transcripts, there are three indices in the queue, $1, 2, 3$. Now we start the execution of the fourth and fifth operations. For T_0, the access pattern of last two operations is *secure-read*(4), *secure-write*(1), *secure-read*(5), *secure-write*(2). But access pattern of transcript T_1 is *secure-read*(4), *secure-write*(1), *secure-read*(1), *secure-write*(2). Since the adversary can see access pattern for the last three operations, it can tell T_0 or T_1 from whether the third to last *secure-write* touches the same address with the second to last *secure-read*.

5.2 CQoram: A c-Snapshot ORAM

Though ICQoram is insecure, the queue-based approach can be fixed. Fixing ICQoram is challenging because of a three-way tension between bounded state size, correctness, and security: to keep the queue's size bounded, elements in it must eventually be evicted. For correctness, the evicted element must be written back to its location in main memory; otherwise, an element updated while in the queue will not have the correct value in

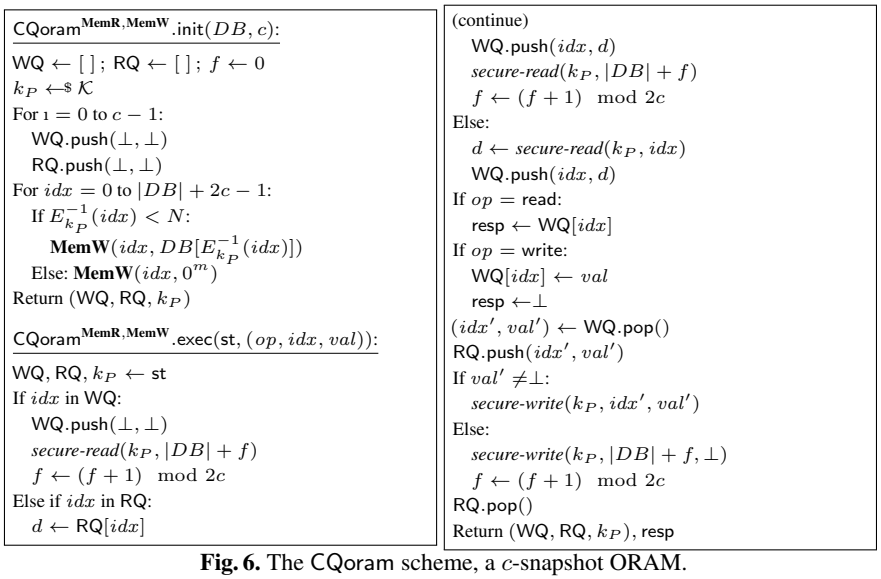

Fig. 6. The CQoram scheme, a c-snapshot ORAM.

the future. But to maintain security—namely, the invariant that all $2c$ accesses are distinct—this location must not be touched again after eviction.

We begin with the simple observation that a second "read-only" queue could be used to keep track of the elements that were recently evicted from the main queue. This could be checked during exec to prevent duplicate accesses, preventing the attack above. Our CQoram scheme will use this idea; as we will see, there are several important subtleties that must be dealt with. Notably, care must be taken if an element is written while it is in this secondary read queue.

The CQoram scheme. We give pseudocode of the scheme in Figure 6. As with FSO, CQoram uses a PRP E with key space $\mathcal{K}$. The CQoram.init procedure is nearly identical to ICQoram's init, except it initializes two queues—the write queue WQ and the read queue RQ—instead of just one, and fills the queues with dummies. The invariant of this scheme is that at the beginning and end of CQoram.exec, both two queues have exactly c elements, either real or dummy, in them.

As with ICQoram, the CQoram.exec procedure has three main phases. First, it checks both WQ and RQ for the index idx to be accessed; like ICQoram, if either queue contains idx it reads a dummy, else it reads idx from main memory. One important new step is in the second branch, which checks RQ. Here, if idx is found in RQ, it will move it and its value back into WQ to maintain the invariant that WQ always contains the element. (We do not need to delete the element from RQ—the copy in RQ will always be deleted before the element is evicted from WQ.)

The second phase is executing the operation on the element. This phase is the same as in ICQoram. The third phase of CQoram.exec, eviction, is necessarily quite different than in ICQoram. It begins by popping the front (oldest) element from WQ and pushing it into RQ anyway. Then it checks if that element is a dummy; if not, writes the element

back to main memory, otherwise writes a dummy. Finally, pop the front element from RQ and (implicitly) deletes it. Finally, we note that CQoram has the same asymptotic performance as ICQoram; concretely, CQoram requires twice as much client storage as ICQoram, but has identical bandwidth and storage overhead.

Security of CQoram**.** Next we prove that CQoram is a c-snapshot oblivious RAM emulator for any c. We begin with a lemma showing that any size-$2c$ subsequence of accesses made with CQoram are to distinct memory locations. Below, we will treat the pair of entries in AP made by our *secure-read* or *secure-write* procedures as one "access", since either procedure just performs a writeback—a read, then a write—on one memory location.

Lemma 1. *Let DB be an array of N m-bit strings, and T be a transcript of n operations. Let $x_1, x_2, \ldots, x_{2n}$ be random variables denoting the sequence of indices in* Mem *accessed by* CQoram *while executing T on DB. For any $i \in [1, 2n]$ let $\{x_i, \ldots, x_{i+2c-1}\}$ be the subsequence of at most $2c$ accesses starting with x_i. Then with probability 1 over the random coins of* CQoram*, all accesses in this subsequence are distinct.*

Proof. We prove this statement in two steps. First, we observe that it is sufficient to prove a weaker statement: namely that for any size-$2c$ sequence of physical accesses, the first access x_i occurs only once in that sequence. This implies all size-$2c$ subsequences are distinct because if there was a subsequence where this did not hold, there would also be a size-$2c$ subsequence where the first access occurred more than once in that subsequence.

Next we prove that the first access occurs only once. The $2c$ memory accesses are either "real" array values or dummies. We know that real values are at position $E_{k_P}(1), \ldots, E_{k_P}(N)$, and dummies are at position $E_{k_P}(N+1), \ldots, E_{k_P}(N+2c)$; thus, dummies cannot have the same address as real values, and so $x_i = x_{i+j}$ can only be the case if they are either both dummies or both real values.

Since the subsequence has $2c$ memory accesses there are at most $2c$ dummies being touched. During CQoram.init we add $2c$ dummies, and we use the counter f to make sure each dummy is accessed only once. Thus, if the accesses are both to dummy values, they must be distinct.

Now we only care about the case where x_i and x_{i+j} are both to real values, and let idx_i and idx_{i+j} be the corresponding real indices. First, we will state three facts about CQoram.exec. (1) Any access to a real value happens either because of *secure-read* or *secure-write*. (2) *secure-read*(idx_i) happens only if idx_i is neither in WQ or RQ. (3) *secure-write*(idx_i) happens only when idx_i is popped from WQ.

There are four cases to analyze.

- *secure-read*$(idx_i), \ldots,$ *secure-read*(idx_{i+j}) After idx_i is read, it is pushed into WQ. idx_i is popped after c new elements are pushed into WQ. Each operation will push exactly 1 element into WQ. Therefore, in the next $c-1$ operations, idx_i is always in WQ, so $idx_i \neq idx_{i+j}$ and $x_i = x_{i+j}$ for all j.
- *secure-read*$(idx_i), \ldots,$ *secure-write*(idx_{i+j}) After idx_i is read, it is pushed into WQ and it is written only when idx_i is popped out. Thus, in the next $c-1$ operations, idx_i is always in WQ, so $idx_i \neq idx_{i+j}$ for all j.
- *secure-write*$(idx_i), \ldots,$ *secure-read*(idx_{i+j}). First, idx_i is pushed into RQ after being written. We read the index idx_i from the memory only if it is not in WQ or

RQ. idx_i is popped only after c new elements are pushed into RQ. Each operation will push 1 element into RQ. Therefore, in the next $c-1$ operations, idx_i is always in RQ, and $idx_i \neq idx_{i+j}$ for all j. (Note that this is the case where ICQoram fails to prevent duplicate reads.)

– *secure-write*$(idx_i), \ldots,$ *secure-write*(idx_{i+j}). As above, idx_i is popped from WQ after being written. We write the index idx_i to the memory only if it is already in WQ. It takes one operation to read idx_i to WQ again and at least $c-1$ operations before being popped out, so $idx_i \neq idx_{i+j}$ for all j.

Thus, we have proved that x_i is only accessed once, and we are done. ∎

Theorem 5. *The* CQoram *scheme is a c-snapshot oblivious RAM emulator, for any c.*

Proof. Each operation has one secure-read and one secure-write, which writes a $(\mathsf{r}, idx)||(\mathsf{w}, idx)$ to the access pattern AP. In c operations, the $2c$ read/write indices are distinct by Lemma 1. Call these $x_1, \ldots, x_{2c}$. Then AP has $2c$ copies of $(\mathsf{r}, idx^*)||(\mathsf{w}, idx^*)$ where the $2c$ $idx^* = E_{k_P}(x_i)$ are distinct and pseudorandom, which are indistinguishable from a hybrid game that idx^* are $\pi(1), \cdots, \pi(2c)$ where π is a random permutation. ∎

Discussion The CQoram scheme has constant bandwidth overhead because each plaintext operation is done by one secure-read and one secure-write, each of which does two memory accesses. So $|\mathsf{CQoram}(DB, T)|/|T| = 4 = \mathcal{O}(1)$. But it needs $\mathcal{O}(c)$ client storage.

We can store the queue on the server, but during CQoram.exec, we need to check the queues' contents. This operation needs to iterate the entire queue, so it has to introduce a linear overhead in c. Therefore on each queue operation, we scan and update the entire queue, which gives us an $\mathcal{O}(c)$ bandwidth overhead and constant client storage. In the next section, we will present a much more efficient way to outsource the queue's storage to the server.

Readers may find that different from the ICQoram scheme, we pad the size of queues to c. Note that this does not fix the insecurity of ICQoram. Instead, if we choose to store the queues on the client's side, removing the paddings even enhances the efficiency. However, if we pop WQ only when $|\mathsf{WQ}| > c$, the latest version of some memory contents may be arbitrarily old. Suppose the transcript is repeatedly writing some values to address 1 to $c-1$, then these updated values are never uploaded because the queue has size $c-1$. Therefore if a client is shutdown unexpectedly, the "back-up" value on the server can be extremely out of date. Our CQoram scheme makes sure that every updated memory value will be uploaded to the server every c operations.

6 Oblivious Hash Queue based c-SnapORAM

As we described above, for the CQoram scheme, the read and write queues can be stored on the server and simply streamed to the client during each CQoram.exec. This allows constant client-side storage but incurs $\mathcal{O}(c)$ bandwidth overhead, which may be prohibitive if c is large.

To reduce this overhead, we could instead store the queues in a smaller ORAM. Since the amount of storage needed for the queues is only $\mathcal{O}(c)$, this would in principle allow us to reduce the overhead of CQoram exponentially, to something like $\mathcal{O}(\log c)$.

However, making this strategy work is quite challenging. The read and write queues in CQoram are used in several different ways in CQoram.exec: searching for (idx, val) pairs, updating their values, and pushing and popping elements in a first-in, first-out fashion. Ultimately, no existing data structure efficiently provides the combination of dictionary and queue properties we need, so we invent our own novel data structure, which we term the *hash queue*.

In this section, we will introduce the syntax of hash queue and give an oblivious hash queue security definition. We show how to build a c-snapshot ORAM (PHQoram) using an oblivious hash queue, and how to use oblivious map to build an oblivious hash queue (OMOHQ). The PHQoram construction has polylogarithmic bandwidth overhead, which will be further reduced in Section 7.

Definition 6. ***(Hash queue)*** *A hash queue is a pair of algorithms: an initialization function* $\mathsf{HQ}.\mathsf{init}(c)$ *and an execution function* $\mathsf{HQ}.\mathsf{exec}(op, args)$ *where* $args$ *is a tuple of arguments.*

A hash queue is initialized by calling its initialization function with argument c, which represents the maximum size of the hash queue. After initialization, the HQ.exec function takes a state as input and output, and supports the following four types of operation:

- $op = \mathsf{Find}$, $args = (key)$. The data structure searches on key and returns val if key is found, otherwise returns $\perp$.
- $op = \mathsf{Push}$, $args = (key, val)$. Insert the key value pair.
- $op = \mathsf{Access}$, $args = (op', key, val)$. If op' is read, searches for key and returns its value. If op' is write, searches on key and replaces its value by val and returns $\perp$. If key is not found, the data structure returns $\perp_0$, a reserved failure symbol distinct from $\perp$.
- $op = \mathsf{Pop}$, $args = ()$. Returns the oldest key-value pair and deletes it.

Below, we will abuse notation slightly and replace exec with the hash queue operation it executes. E.g., $\mathsf{HQ}.\mathsf{Find}(key)$ instead of $\mathsf{HQ}.\mathsf{exec}(\mathsf{Find}, (key))$.

6.1 Hash Queue Security

A natural security definition for hash queues is an ORAM-style notion that requires hiding everything except the operation count. This kind of definition is typical of other oblivious data structures [52]. However, such a definition is stronger than what we need: our goal is to replace the client-side queues in CQoram with hash queues; in CQoram (Figure 6). Notice that no matter what the transcript is, for each CQoram.exec, we always search idx in WQ, then execute push, modify, and pop in the WQ. Likewise, we search in RQ at the beginning (not always, but we can do a dummy search), then push and pop. That is to say the sequence of operation executed on a queue which will be replaced by an oblivious hash queue, is always the same and publicly known in advance. Because of such observation, we propose our first obliviousness definition. We give the pseudocode for our *public operation obliviousness* security notion for hash queues in Figure 7.

Similar to the security game of RAM emulators, we define both of init, exec as relative to a pair of oracles **MemR**, **MemW**. Cryptographic primitives like hash queue

PublicOpOblivHashSec($\mathsf{HQ}, \mathcal{A}, n, i$):
$\mathsf{AP} \leftarrow []$
$\mathsf{st}_0 \leftarrow\$ \mathsf{HQ}^{\mathbf{MemR},\mathbf{MemW}}.\mathsf{init}(n)$
$T_0, T_1 \leftarrow\$ \mathcal{A}_0(n)$
For $x = 1$ to ℓ:
  If $T_0[x].op \neq T_1[x].op$ then Return $\perp$
  $op, args \leftarrow T_i[x]$
  $\mathsf{st}_x \leftarrow\$ \mathsf{HQ}^{\mathbf{MemR},\mathbf{MemW}}.\mathsf{exec}(\mathsf{st}_{x-1}, op, args)$
$b \leftarrow\$ \mathcal{A}_1(\mathsf{AP})$
Return b

Fig. 7. Game defining public operation obliviousness for a hash queue. AP is modified by oracles **MemR**, **MemW** as defined in Figure 2 during the execution of exec function.

use the same **MemR**, **MemW** oracles to access the entire physical memory. To make sure the primitives do not overwrite others' memory, each primitive is allocated a primitive identifier pid and memory space when calling init. **MemR**, **MemW** implicitly take pid as an argument and add a proper offset to get the physical memory address.

Definition 7. ***(Public-operation oblivious hash queue)*** *For a two-part adversary $\mathcal{A}$ playing game defined in Figure 7, we define the public-operation obliviousness advantage of $\mathcal{A}$ against HQ as*

$$\mathbf{Adv}^{opo}_{\mathsf{HQ}}(\mathcal{A}) = \big|\Pr[\mathit{PublicOpOblivHashSec}(n, \mathsf{HQ}, \mathcal{A}, 0) = 1] - \mathit{PublicOpOblivHashSec}(n, \mathsf{HQ}, \mathcal{A}, 1) = 1\big|.$$

If for a hash queue HQ, for any nuPPT adversary $\mathcal{A}$, the above advantage is negligible, we say that HQ is public-operation oblivious.

The game is similar to our obliviousness notion for RAM emulators in Section 2. It lets the adversary $\mathcal{A}_0$ output two pairs of transcripts with the same "operation pattern", executes the ith transcript, and gives the $\mathcal{A}_1$ the access patterns and outputs its guess b.

6.2 A c-Snapshot ORAM From Hash Queues

Next we describe a generic transformation that builds a c-snapshot ORAM from any hash queue meeting the public-operation obliviousness property defined above. (In the next subsection, we will construct a hash queue which enjoys this property.) We call our construction PHQoram, and give its pseudocode in Figure 8. At a high level, PHQoram follows the strategy we outlined above of outsourcing CQoram's read and write queues to the server. PHQoram replaces RQ with a hash queue rOHQ, and likewise replaces WQ with a hash queue wOHQ. The procedure PHQoram.init initializes the two queues independently in non-overlapping regions of Mem (handled by **MemR**, **MemW** oracles), then samples a PRP key and fills the rest of Mem with DB entries and dummy elements. The procedure PHQoram.exec works similarly to CQoram.exec, with a few important differences. Most notably, it executes both wOHQ.Find and rOHQ.Find, whereas CQoram does not check RQ if the index is found in WQ. This prevents leaking the hash queue contents based on the number of accesses to each hash queue.

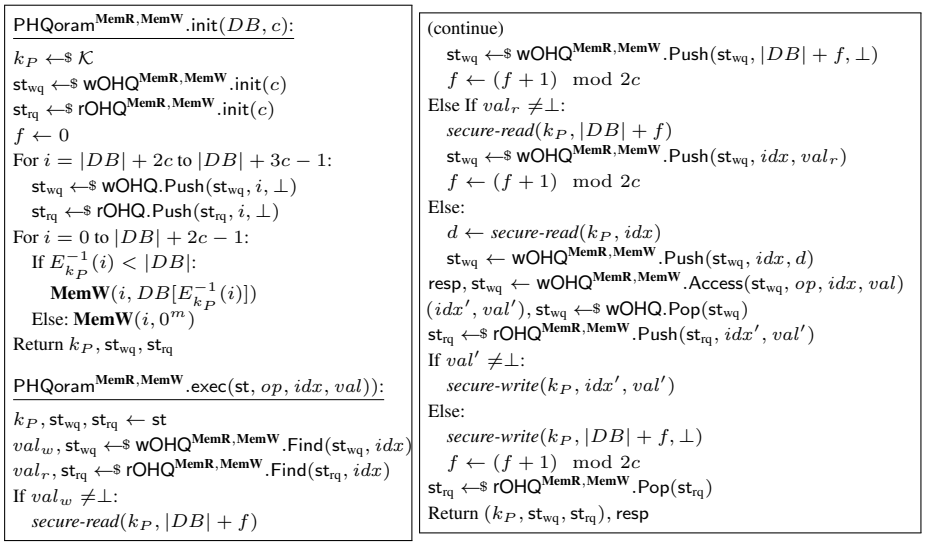

Fig. 8. Construction of PHQoram c-Snapshot ORAM emulator in pseudocode. The exec procedure starts on the left and continues on the right.

It is not too hard to see that if wOHQ, rOHQ has bandwidth overhead $g(c)$ for each operation, then PHQoram has bandwidth overhead $\mathcal{O}(g(c))$. Regardless of which branch is taken, PHQoram does the following things on each RAM operation:

wOHQ.Find, rOHQ.Find, *secure-read*, wOHQ.Push, wOHQ.Access, wOHQ.Pop, rOHQ.Push, *secure-write*, rOHQ.Pop.

Since there are a constant number (7) of hash queue operations each with $g(c)$ overhead and a constant number (2) of accesses to the "main" memory with $\mathcal{O}(1)$ overhead, the overall bandwidth overhead is $\mathcal{O}(g(c))$.

Theorem 6. *Let E be a secure PRP and wOHQ, rOHQ be public-operation oblivious hash queues. Then the* PHQoram *scheme in Figure 8 is a c-snapshot oblivious RAM emulator.*

Proof. We will prove c-snapshot obliviousness by reduction. The high-level strategy is as follows: first, we will perform two game hops to "decouple" the operations made against the two hash queues from the adversary's chosen transcripts in SnapObSec. (Specifically, we will simply execute the same operation sequence on wOHQ and rOHQ, but with dummy arguments.) In these hybrid games we will ensure the correctness of the distribution of accesses to the main memory using local queues; effectively, after these two game hops, the access pattern to the main memory will be distributed as in the CQoram scheme. Then, we can use a variant of the security argument for CQoram to perform one more game hop which changes the PRP's outputs to a random subset of the memory locations.

We now proceed more formally. Let $\mathcal{A}$ be a SnapObSec adversary. We will show that there exists adversaries $\mathcal{B}$, $\mathcal{C}$, and $\mathcal{D}$ such that

$$\mathbf{Adv}^{\text{snap}}_{\mathsf{PHQoram}}(\mathcal{A}) \leq 2\mathbf{Adv}^{\text{opo}}_{\mathsf{wOHQ}}(\mathcal{B}) + 2\mathbf{Adv}^{\text{opo}}_{\mathsf{rOHQ}}(\mathcal{C}) + 2\mathbf{Adv}^{\text{prp}}_{E}(\mathcal{D}) \ .$$

We do this via a sequence of games. Game G_0 is SnapObSec($\mathsf{PHQoram}, \mathcal{A}, c, 0$). Game G_1 is the same as G_0 except for two additional (local) queues, WQ and RQ, are added to PHQoram.exec that "mirror" (resp.) wOHQ and rOHQ: any modifications made to wOHQ or rOHQ are also made to their corresponding local queues, but the access pattern is otherwise unchanged. Clearly, this does not affect $\mathcal{A}$'s view, so $\Pr[G_0 = 1] = \Pr[G_1 = 1]$.

Next we define the game G_2. This game is identical to G_1, except the arguments to all wOHQ operations (except the state) are replaced with fixed values: all indices are replaced with zero. The local queue WQ is used in place of wOHQ. We can upper-bound the difference in advantage between G_1 and G_2 by building a reduction $\mathcal{B}_0$ to the public-operation obliviousness of wOHQ. The reduction $\mathcal{B}_0$ works as follows: first, it runs $\mathcal{A}_0$ to get $(DB_0, T_0), (DB_1, T_1)$. Then, it samples k_P and with its own simulated **MemR**, **MemW** oracles initializes rOHQ and executes PHQoram on (DB_0, T_0) as in G_1. However, $\mathcal{B}_0$ only uses WQ and does not perform wOHQ operations; instead, it marks the access patterns of these operations in AP with $\perp$ and records the operations that would have been executed against wOHQ. This is the "induced" transcript of operations on wOHQ in G_1. Call this transcript $\overrightarrow{op}_w^{G_1}$. Concretely, it consists of c Push operations made during init, then for each RAM operation, the transcript contains Find, Push, Access, Pop. (Note that the sequence of wOHQ operation *types* is fixed and does not depend on the RAM operation.) Then, $\mathcal{B}_0$ constructs the "dummy" transcript $\overrightarrow{op}_w^{G_2}$, containing the same operation types but with all-zero arguments; it then outputs $\overrightarrow{op}_w^{G_1}, \overrightarrow{op}_w^{G_2}$ as its chosen transcripts in its PublicOpOblivHashSec game. When $\mathcal{B}_0$ gets the array of access patterns in the second stage of the PublicOpOblivHashSec game, it uses them to fill in the entries of AP which were marked with $\perp$ previously.

At this point, $\mathcal{B}_0$ has an access pattern array AP which is distributed as in G_1 if $i = 0$ in PublicOpOblivHashSec, and distributed as in G_2 if $i = 1$. Thus, $\mathcal{B}_0$ can simply truncate AP to the last c operations, compute the state of Mem before these operations, run $\mathcal{A}_1$ as in SnapObSec, and return its output. By construction,

$$|\Pr[G_1 = 1] - \Pr[G_2 = 1]| \leq \mathbf{Adv}^{\text{opo}}_{\mathsf{wOHQ}}(\mathcal{B}_0) \ .$$

Next we define G_3, which is the same as G_2 except we also replace the arguments to rOHQ with "dummy" all-zeros strings. (Note that, like wOHQ, the operation types executed on rOHQ while PHQoram executes a RAM operation are fixed to Find, Push, Pop.) By an argument similar to the above, we can construct a reduction $\mathcal{C}_0$ to the public-operation obliviousness of rOHQ, giving us that $|\Pr[G_2 = 1] - \Pr[G_3 = 1]| \leq \mathbf{Adv}^{\text{opo}}_{\mathsf{rOHQ}}(\mathcal{C}_0)$.

In G_3, only the accesses to the "main" memory (i.e., the permuted array) depend on (DB_0, T_0). Dummy operations are made against wOHQ and rOHQ; the actual state of those queues is kept track of locally, as in the CQoram scheme in Section 5. Next, we construct game G_4, where the "main" memory consists of indices of the $2c$ accesses to the main memory (*secure-read*s and *secure-write*s) seen by $\mathcal{A}$ are chosen by sampling a

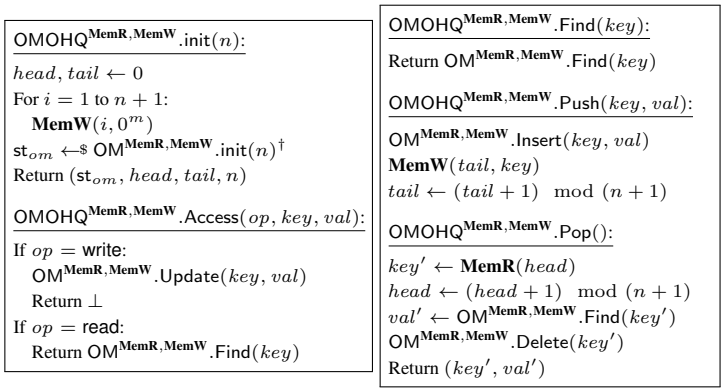

Fig. 9. The OMOHQ hash queue construction. All operations input and output the state returned from init; we leave this implicit for brevity. († We leave implicit the domain separation in these **MemR/MemW** oracles. See Section 6.)

subset of $[1, \ldots, |DB| + 2c]$ uniformly at random. By an argument very similar to the proof of Theorem 5, we can build $\mathcal{D}_0$ and $\mathcal{E}_0$ so that

$$|\Pr[G_3 = 1] - \Pr[G_4 = 1]| \leq \mathbf{Adv}^{\text{prp}}_E(\mathcal{D}_0) \ .$$

In game G_4, $\mathcal{A}$'s view does not depend on either (DB_0, T_0) or (DB_1, T_1). Thus, we can perform the previous game transitions in reverse to get to $\text{SnapObSec}(\mathsf{PHQoram}, \mathcal{A}, c, 1)$. A standard argument lets us build $\mathcal{B}, \mathcal{C}, \mathcal{D}$ whose advantages are at most twice the right-hand sides of the above terms; applying the triangle inequality yields the result. ∎

6.3 Constructing Public-operation Oblivious Hash Queues

Now that we have shown that c-snapshot ORAMs can be built from hash queues with public-operation obliviousness, we just need to construct a hash queue meeting this security notion. In this subsection we will give such a construction, which we call OMOHQ.

The OMOHQ **construction.** In Figure 9, we give the pseudocode of OMOHQ. It is built from an oblivious map which supports Insert, Find, Delete, Update, and an array which serves as a queue. The init function initializes OM, chooses a key k'_E, and writes an array of all-zeros to the memory. It also initializes two queue pointers $head, tail$ to zero. The Find and Access procedures are essentially pass-throughs to their corresponding oblivious map operations, where Access branches on the op input. The Push and Pop procedures use both the array and OM. Push inserts key, val in the end of the hash queue, by storing it at the $tail$ position and inserting the key/value pair in OM. Pop does the reverse—removing the key/value pair at the front of the hash queue. It does this by reading and decrypting the key stored at $head$ and using two OM operations to read its value val' and delete it.

Theorem 7. *If* OM *is an oblivious map, then* OMOHQ *in Figure 9 is a public operation oblivious hash queue.*

Proof. The high-level strategy is similar to the proof of Theorem 6: we will transition from PublicOpOblivHashSec with $i = 0$ to a game where all OM operations take fixed, dummy arguments, and use a local map to ensure the accesses to the array have the correct distribution. From there, we will transition to a game where the accesses in the array depend on the transcripts output by the adversary in PublicOpOblivHashSec. Reversing these transitions will get us to PublicOpOblivHashSec with $i = 0$.

We proceed via a sequence of game transitions. Let $\mathcal{A}$ be an adversary, and let game G_0 be PublicOpOblivHashSec($\mathsf{PHQoram}, \mathcal{A}, n, 0$). We transition to game G_1, where a local map data structure "mirrors" the oblivious map OM. Then, we transition to game G_2, where the arguments to OM operations are fixed to be all-zeros, and the array's contents are determined using the local map. We can upper-bound the difference in these two games outputting 1 by building a reduction $\mathcal{B}_0$ to the obliviousness of OM. The reduction $\mathcal{B}_0$ runs $\mathcal{A}$ to obtain T_0, T_1, then simulates OMOHQ on T_0 to determine the induced OM transcript. Then, $\mathcal{B}$ submits this along with the fixed all-zeros OM transcript as its chosen transcripts in the OblivMapSec game. It uses the access patterns it receives to simulate $\mathcal{A}$'s access pattern input. By construction, $|\Pr[G_1 = 1] - \Pr[G_2 = 1]| \leq \mathbf{Adv}^{\mathrm{om}}_{\mathsf{OM}}(\mathcal{B}_0)$.

We next move to game G_3, which is the same as G_2 except the array accesses depend only on the operation *type*, but not the arguments. The access pattern to the array is actually identically distributed in G_2 and G_3: observe that in OMOHQ, the way the array is accessed depends only on the operation type: init writes to it n times, Push writes to position $tail$, and Pop reads from $head$. Thus, for any pair of transcripts output by the adversary in PublicOpOblivHashSec, the access pattern to the array is fixed because the transcripts must have the same operation sequence. Thus, the game G_3 is identical to G_2, giving $\Pr[G_2 = 1] = \Pr[G_3 = 1]$.

In game G_3, the access patterns and the memory contents do not depend on either of $\mathcal{A}$'s output transcripts; thus, we can reverse these game transitions to get to PublicOpOblivHashSec with $i = 1$. By applying an argument similar to the one at the end of the proof of Theorem 6, the result follows. ∎

Asymptotic and concrete performance. The asymptotic performance of the c-snapshot ORAM PHQoram depends on how OM in OMOHQ is instantiated. A special-purpose oblivous map data structure (e.g. [52]) is likely to be the most efficient choice. The best-known oblivious maps achieve $\mathcal{O}(\log^2 n)$ bandwidth overhead for size-n memory. This implies that the bandwidth overhead of OMOHQ, and thus the PHQoram construction, is $\mathcal{O}(\log^2 c)$ for c-snapshot obliviousness.

The concrete performance of PHQoram is a more complex question, as it depends greatly on implementation specifics. The best-known oblivious map construction has good asymptotics, but its concrete bandwidth overhead is still quite large for small databases: for example, the evaluation of [46] shows that reading an eight-byte key/value pair requires communicating over 100 KBs to the client. Despite exponentially worse asymptotics, it may be the case that the CQoram scheme is more efficient than PHQoram for practical values of c, due to its small constants. It does not seem inherent that oblivious maps perform poorly for small memory sizes; we leave improving them in this parameter regime to future work.

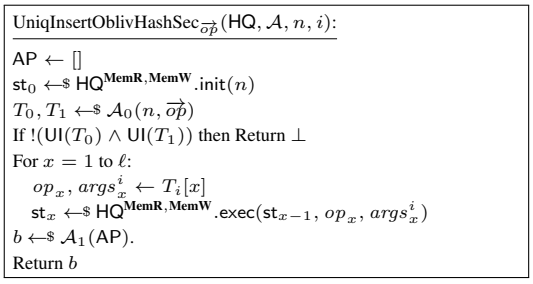

Fig. 10. Unique insertion oblivious hash queue security definition. The function $\mathsf{UI}(T)$ returns 1 if the keys given to Push operations are all distinct, and 0 otherwise.

7 Asymptotically-Optimal c-Snapshot ORAM

In this section, we give tight upper and lower bounds on the asymptotic performance of c-snapshot ORAMs. Beginning with the upper bound, we propose a new oblivious hash queue security definition different from Section 6 and show the UHQoram construction in Section 7.1 using an instance (CCOHQ) of our new oblivious hash queue variant. UHQoram is a modification of PHQoram which guarantees an important unique-insertion property for the queues: namely, that duplicate keys are never Pushed. Though a seemingly small change, we show that guaranteeing unique insertions is crucial because it allows weakening the security requirements on UHQoram's hash queues, admitting more efficient instantiations.

We show CCOHQ, a hash queue construction meeting this weakened security requirement with $\mathcal{O}(\log n)$ bandwidth overhead for n items. Instantiating UHQoram with CCOHQ gives a c-snapshot ORAM with $\mathcal{O}(\log c)$ bandwidth overhead. Finally, in Section 7.3, we extend the seminal $\Omega(\log n)$ lower bound of [38]. Our lower bound implies that any c-snapshot ORAM must have $\Omega(\log c)$ bandwidth overhead, implying UHQoram is asymptotically optimal in terms of bandwidth overhead.

We first define the weakened hash queue security notion that UHQoram will use.

Definition 8. ***(Unique-insertion oblivious hash queue)*** *Let HQ be a hash queue, and let $\overrightarrow{op}$ be a sequence of hash queue operation types. Let UniqInsertOblivHashSec be the game in Figure 10. We define the $\overrightarrow{op}$-unique insertion obliviousness advantage of an adversary $\mathcal{A}$ against HQ as*

$$\mathbf{Adv}^{uio}_{HQ,\overrightarrow{op}}(\mathcal{A}) = \big|\Pr[\mathit{UniqInsertOblivHashSec}_{\overrightarrow{op}}(HQ, \mathcal{A}, n, 0) = 1] - \Pr[\mathit{UniqInsertOblivHashSec}_{\overrightarrow{op}}(HQ, \mathcal{A}, n, 1) = 1]\big| .$$

We call HQ $\overrightarrow{op}$-unique-insertion oblivious if for all nuPPT adversaries $\mathcal{A}$, $\mathbf{Adv}^{uio}_{HQ}(\mathcal{A})$ is negligible. If HQ is $\overrightarrow{op}$-unique-insertion oblivious for all $\overrightarrow{op}$, we simply say it is unique-insertion oblivious.

Looking ahead, we will only analyze $\overrightarrow{op}$-unique-insertion oblivious for our CCOHQ hash queue construction for the fixed $\overrightarrow{op}$ induced by the UHQoram c-snapshot ORAM; thus, below we will always refer to $\overrightarrow{op}$-unique-insertion oblivious.

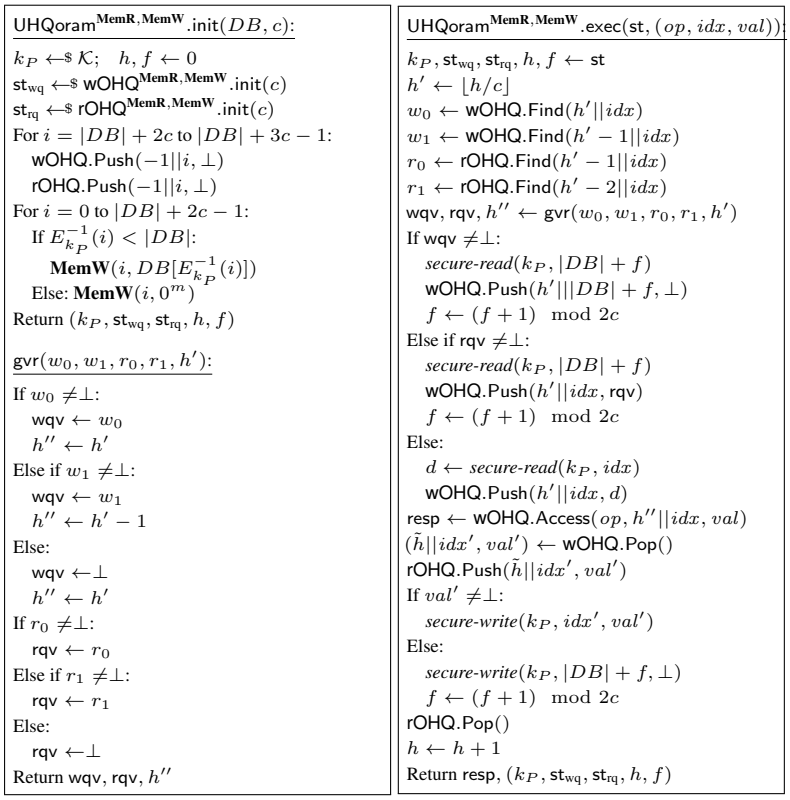

Fig. 11. Construction of UHQoram c-snapshot ORAM. The function gvr is a helper function used during exec. All hash queue operations in exec input and output a state. Oracles **MemR**, **MemW** are as defined in Figure 2.

7.1 The UHQoram Construction

The UHQoram construction is depicted in pseudocode in Figure 11. It is substantially similar to PHQoram above, with two important differences. First, in addition to the counter f, there is another counter h for the total number of operations executed. This counter is used to derive a "round" number, which is prepended to the index when it is written to either of the hash queues. This round number ensures all keys written to the hash queues are distinct (we will argue this more formally in Theorem 8). Another change from PHQoram is the addition of two calls to Find at the beginning of UHQoram.exec. Because each hash queue entry has a round number prepended, we need to check all possible round numbers to be sure to find an entry.

The final change is the use of a helper function gvr during exec. This helper function takes the result of the four Find operations, and outputs the correct value and the round number needed to modify the correct element in wOHQ.Access. The case logic in gvr looks complex, but it is just ensuring the newest copy of the element is always selected.

UHQoram has the same asymptotic overhead as the hash queue: if each hash queue operation takes $g(c)$ bandwidth, each UHQoram operation takes $\mathcal{O}(g(c))$ bandwidth.

Next we will state and prove a security theorem for UHQoram. This theorem will prove it is a c-snapshot ORAM by reduction to the PRP security, and the unique-insertion obliviousness of wOHQ, rOHQ.

We do not need unique-insertion obliviousness of wOHQ, rOHQ to hold for any operation sequences; for simplicity we instead focus on the two sequences induced by our UHQoram construction above. Specifically, define

$$\overrightarrow{op}_w = \mathsf{Push}, \ldots, \mathsf{Push}, \mathsf{Find}, \mathsf{Find}, \mathsf{Push}, \mathsf{Access}, \mathsf{Pop}, \ldots$$

where there are c Pushes, then copies of the Find, Find, Push, Access, Pop sequence. This is the sequence run on wOHQ by UHQoram above. Likewise, define

$$\overrightarrow{op}_r = \mathsf{Push}, \ldots, \mathsf{Push}, \mathsf{Find}, \mathsf{Find}, \mathsf{Push}, \mathsf{Pop}, \ldots \ .$$

This is the operation sequence for the rOHQ hash queue in UHQoram. The next theorem proves that as long as wOHQ and rOHQ are (resp.) $\overrightarrow{op}_w$ and $\overrightarrow{op}_r$-unique-insertion oblivious hash queues, UHQoram in Figure 11 is a c-snapshot oblivious RAM emulator.

Theorem 8. *Let E be a secure PRP and wOHQ be $\overrightarrow{op}_w$-, and rOHQ be $\overrightarrow{op}_r$-unique-insertion oblivious hash queues. Then* UHQoram *in Figure 11 is a c-snapshot oblivious RAM emulator.*

Proof. The proof is substantially similar to that of Theorem 6 above; the chief difference is that we reduce to a weaker security property of the hash queues (unique-insertion obliviousness for $\overrightarrow{op}_w$ and $\overrightarrow{op}_r$). Thus, we only need to extend our previous argument to explain why the unique-insertion property holds for wOHQ and rOHQ. First, define a "round" to be a group of c operations. We begin by proving there are no duplicate key insertions into wOHQ. An array entry idx, val can be inserted into wOHQ in only three places, namely the three branches of the first if-statement of UHQoram.exec. If it is inserted in the first branch, it is a dummy; since there are $2c$ dummies but the round counter h' increments every c operations, duplicate insertion is impossible there.

If it is inserted in the second branch, it is being re-added to wOHQ from rOHQ. In this case, the element had been in wOHQ previously; however, the round counter h' must be different from the one that was used in the previous insertion to wOHQ — this second branch can only happen c operations after the initial insertion.

If idx, val is inserted in the third branch, idx was neither in wOHQ nor rOHQ. Since the round counter for this insertion is always the current one, this insertion must be unique, since idx was last in wOHQ (with any round counter) at least c operations ago.

We've proven that wOHQ never sees a duplicate insertion, but still need to prove this holds for rOHQ. Observe that rOHQ contains exactly the same keys as wOHQ did c operations ago — essentially, rOHQ is an older replica of wOHQ. Thus, because wOHQ has the unique-insertion property, rOHQ does as well, and we are done. ∎

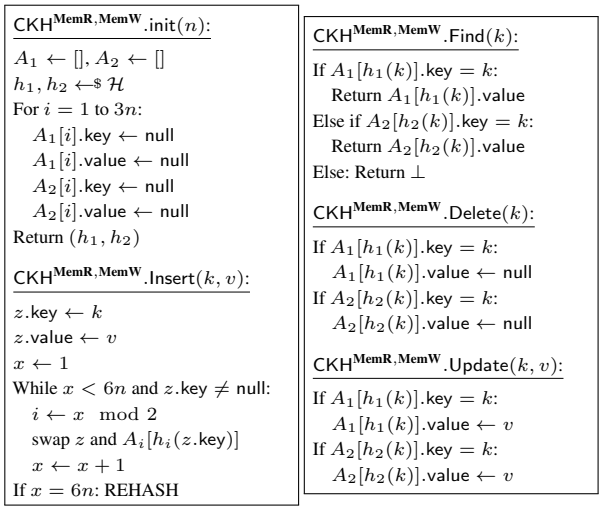

Fig. 12. Pseudocode for cuckoo hashing algorithms. For space reasons we leave the definition of the rehashing procedure implicit.

7.2 Constructing Unique-insertion Oblivious Hash Queues

Now, we give an oblivious hash queue called CCOHQ. Our pseudocode is in Figure 13. As with OMOHQ, our construction consists of two parts: an array to maintain first-in-first-out order and a dictionary data structure. In CCOHQ, though, we do not use a generic oblivious map: instead, we use a specific construction, namely cuckoo hashing running on top of a generic ORAM. We give pseudocode for cuckoo hashing in Figure 12. (Recall that cuckoo hashing supports $\mathcal{O}(1)$ time worst case lookup and delete, and expected $\mathcal{O}(1)$ time insert.) Note that to achieve bandwidth overhead $\mathcal{O}(\log c)$, we need to use an ORAM whose bandwidth overhead $\mathcal{O}(\log N)$, such as OptORAMa [4]. We depict this in the figure by having the cuckoo hash CKH use simulated memory read/write oracles built from ORAM, denoted **OMR** and **OMW**. We also apply a PRP to the keys before they are inserted into the cuckoo hash table. As we will see below, this is important to ensure security. We draw the reader's attention to the fact that this is different from oblivious cuckoo hashing in [10,4]. Their hash tables only support one-time lookups after being initialized but we need multiple time lookups and modifications.

Security of CCOHQ**.** Recall that UHQoram only needs hash queues that are unique-insertion oblivious for the two fixed operation sequences —$\overrightarrow{op}_w$ and $\overrightarrow{op}_r$—defined above. Thus, we only need to prove CCOHQ satisfies $\overrightarrow{op}_w$ and $\overrightarrow{op}_r$-unique-insertion obliviousness to conclude that UHQoram in Figure 11 is a c-snapshot oblivious RAM emulator when instantiated with CCOHQ.

Theorem 9. *Let E be a secure PRP and* ORAM *be an oblivious RAM. Then* CCOHQ *in Figure 13 is a $\overrightarrow{op}_w$- and $\overrightarrow{op}_r$-unique insertion oblivious hash queue.*

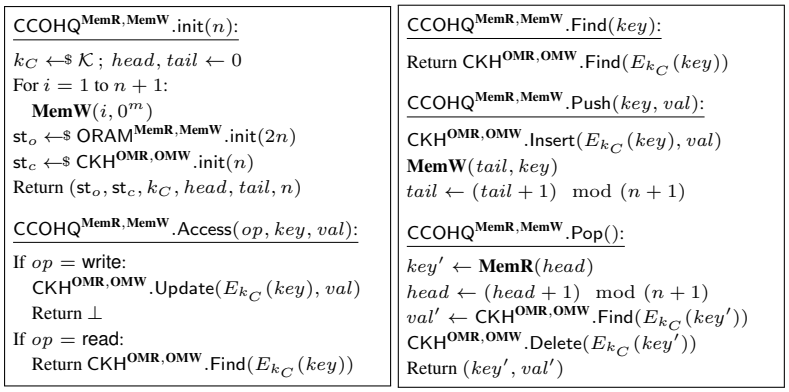

Fig. 13. The CCOHQ hash queue. All operations take a state as input and output. All operations executed by the cuckoo hash table CKH are executed with *simulated* memory read/write oracles built from ORAM.

Before the proof we give an idea of why a simple combination of cuckoo hashing and an ORAM does not give us an oblivious data structure that supports arbitrary insertion, even if we do not hide operation type. This is because the number of memory accesses made during insertion depends on the number of swaps. Take these two transcripts:

$$T_1 = \mathsf{Insert}(1), \ldots, \mathsf{Insert}(100), \mathsf{Insert}(0), \mathsf{Delete}(0), \mathsf{Insert}(0), \mathsf{Delete}(0), \ldots$$
$$T_2 = \mathsf{Insert}(1), \ldots, \mathsf{Insert}(100), \mathsf{Insert}(101), \mathsf{Delete}(1), \mathsf{Insert}(102), \mathsf{Delete}(2), \ldots.$$

Both transcripts insert 1 to 100 at the beginning. Then the first one repeatedly inserts 0 and deletes 0, while the second one inserts new keys and deletes old keys. Now let's analyze the transcripts starting the first Delete operation. In the first transcript, since 0 is always deleted before being inserted, inserting 0 takes only one ORAM access. However, in the second transcript, inserting new keys such as 101, 102, ... is very likely to incur swaps, and therefore makes the access pattern longer than the previous one.

Proof. At a high level, the proof has the following steps. We will begin in game $\text{UniqInsertOblivHashSec}_{\overrightarrow{op}_w}$ with $i = 0$. Then, we move to a game where the array is replaced by all zeros, and the queue is stored locally. Then, we use the obliviousness of ORAM to make a series of changes to the transcript of cuckoo hash operations: in one game transition, we change all Update operations to Finds. Then, we change the arguments of all Finds to all-zeros, and all second arguments of Insert to zeros (keeping the indices the same). At this point, we are in a game where only the indices passed to CKH.Insert and CKH.Delete depend on the adversary's chosen transcript. However, since we can guarantee duplicate indices are never passed to CKH.Insert, we can apply the PRP security to swap the set of indices for a random subset of $[|DB|]$.

We now proceed formally. Let $\mathcal{A}$ be an adversary, and let game G_0 be $\text{UniqInsertOblivHashSec}_{\overrightarrow{op}_w}(\mathsf{CCOHQ}, \mathcal{A}, n, 0)$. Let T_0 be $\mathcal{A}$'s left transcript.

We build the game G_1, which is just like G_0 except in all CCOHQAccess operations, CKH.Find is always executed instead of choosing between Find and Update based

on whether the operation is a read or a write. Again, the correctness of the values is maintained locally instead of by writing them to the cuckoo hash table. Note that this does not change the number of (oblivious) memory accesses made by CKH, since both Find and Update only access two locations. We can build a reduction $\mathcal{B}$ to the obliviousness of ORAM to get $|\Pr[G_1 = 1] - \Pr[G_0 = 1]| \leq \mathbf{Adv}^{\text{obl}}_{\mathsf{ORAM}}(\mathcal{B})$.

Next is game G_2, which is the same as G_1 except all CKH.Find operations have all-zeros arguments, and values written using CKH.Insert are replaced with zeros; correctness is ensured with local copies. Since this also does not change the number of operations executed, we can use a similar argument to build another reduction $\mathcal{C}$ to the obliviousness of ORAM, yielding $|\Pr[G_2 = 1] - \Pr[G_1 = 1]| \leq \mathbf{Adv}^{\text{obl}}_{\mathsf{ORAM}}(\mathcal{C})$.

In game G_2, only the indices passed to CKH.Insert and CKH.Delete depend on the adversary's transcript T_0. In game G_3, we replace the set of indices passed to CKH.Insert with a random subset of $[|DB|]$. This will change the number of memory accesses made by CKH.Insert, since a different number of swaps will be needed to insert the indices into the hash table. However, because of the unique-insertion property, in game G_2 the hash table contains the PRP evaluated on distinct keys; thus, by PRP security, the distribution of these inputs (and therefore of the swaps) is very similar in game G_3. We can build a reduction $\mathcal{D}$ to the PRP security of E to get $|\Pr[G_3 = 1] - \Pr[G_2 = 1]| \leq \mathbf{Adv}^{\text{prp}}_{E}(\mathcal{D})$.

Reversing these game transitions in a manner similar to the proofs above lets us transition to game $\text{UniqInsertOblivHashSec}_{\overrightarrow{op}_w}(\mathsf{CCOHQ}, \mathcal{A}, n, 1)$, and we are done. Finally, the proof for $\overrightarrow{op}_r$-unique-insertion obliviousness is similar, so we omit it. ∎

7.3 Lower bound

The lower bound for snapshot-oblivious RAM emulator follows from Larsen & Nielsen's lower bound [38]. In this subsection we first restate the main theorem of [38], then show that c-snapshot ORAM can simulate a normal ORAM in a parameter regime where the Larsen & Nielsen lower bound applies.

Theorem 10. *(Larsen & Nielsen lower bound [38]) Any online ORAM with n blocks of memory, consisting of $r \geq 1$ bits each, must have an expected amortized bandwidth overhead of $\Omega(\log(nr/m))$ on a sequence of $\Theta(n)$ operations. Here m denotes the client memory in bits.*

Applying this theorem to our setting where each block is only one address and the client memory is constant, which means r, m are constant, we obtain the following corollary.

Corollary 1. *Any RAM emulator defined in Section 2.2 initialized with a database of size N, executing on a sequence of N operations, with constant client storage, is secure only if it has an expected amortized bandwidth overhead of $\Omega(\log(N))$.*

The idea of our result is to use a snapshot-oblivious RAM emulator to simulate a full ORAM. Notice that if the transcript is of length c, and the memory is at least of size c, the c-snapshot-oblivious RAM emulator becomes a secure RAM emulator executing a sequence of c operations, and the corollary above applies. Thus, the lower bound of amortized bandwidth overhead is $\Omega(\log(c))$.

Theorem 11. *Let $c > 0$ be an integer. If RE is a c-snapshot oblivious RAM emulator, then RE must have $\Omega(\log c)$ expected amortized bandwidth overhead if the client has constant memory.*

Proof. Suppose for contradiction that RE is a c-snapshot oblivious RAM emulator, and it has $o(\log c)$ bandwidth overhead. We initialize RE on any database of size c. Given a transcript of c operations, RE can securely emulate the RAM by the definition of c-snapshot obliviousness, but its $o(\log c)$ bandwidth overhead contradicts Corollary 1. ∎

8 Conclusion

In this work, we initiated the study of snapshot-oblivious RAMs, a new oblivious memory primitive. There are many interesting open questions which we leave for future work.

First, while we prove that our UHQoram scheme is asymptotically optimal in terms of bandwidth overhead, its concrete performance is likely to be quite poor. Evaluating the concrete performance of c-snapshot ORAMs, and improving concretely upon the constructions of this paper, is a clear interesting question.

In this work we do not tackle the question of how system designers should choose c. This is a complex and highly contextual question; it is natural to imagine system designers choosing c by weighing the risks of different compromises in their systems. Which risks to consider, are questions we leave to future work.

For our security model to be an accurate characterization of real compromises, it should be the case in real systems that the amount of information about past operations is limited. If, for example, a system stored the history of every memory access on disk, the limited-time compromise model in this paper would be unrealistic.

Prior work found that existing systems do, in fact, store a great deal of information about past operations [23]. Realizing our security model in today's systems is indeed a challenge. We believe building systems with limited memories is ultimately tractable, and a fascinating research problem in its own right. In addition to being of theoretical interest today, our work builds a foundation for cryptography that can take advantage of these kinds of system-level guarantees in the future.

Finally, there are many interesting ways to extend and enrich our snapshot security model. One very clear open question is building schemes that remain secure even for multiple snapshot compromises that are separated in time. Real systems are sometimes compromised multiple times, so this extension is well-motivated practically. Another interesting enhancement is transcript-length-hiding: namely, requiring that the number of total operations executed is hidden by the snapshot-oblivious RAM.

Acknowledgments

The authors thank their shepherd Mark Simkin and the anonymous reviewers at CRYPTO 2022 for their helpful comments and suggestions. This work was partially supported by the Air Force Office of Scientific Research (AFOSR) under award number FA9550-20-1-0425; the National Science Foundation under grant CNS-1954712 and by a gift from Qualcomm.

References

1. Verizon Data Breach Incident Report (2021), https://www.verizon.com/business/resources/reports/2021-data-breach-investigations-report.pdfx

2. Abraham, I., Fletcher, C.W., Nayak, K., Pinkas, B., Ren, L.: Asymptotically tight bounds for composing ORAM with PIR. In: IACR PKC (2017)
3. Amjad, G., Kamara, S., Moataz, T.: Breach-resistant structured encryption. Proceedings on Privacy Enhancing Technologies (2019)
4. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: Optorama: Optimal oblivious RAM. In: IACR EUROCRYPT (2020)
5. Asharov, G., Komargodski, I., Lin, W.K., Shi, E.: Oblivious RAM with worst-case logarithmic overhead. In: IACR CRYPTO (2021)
6. Boyle, E., Naor, M.: Is there an oblivious RAM lower bound? In: ITCS (2016)
7. Cash, D.: A survey of Oblivious RAMs (2012), https://cseweb.ucsd.edu/~cdcash/oram-slides.pdf
8. Cash, D., Drucker, A., Hoover, A.: A lower bound for one-round oblivious RAM. In: IACR TCC (2020)
9. Cash, D., Grubbs, P., Perry, J., Ristenpart, T.: Leakage-abuse attacks against searchable encryption. In: ACM CCS (2015)
10. Chan, T.H.H., Guo, Y., Lin, W.K., Shi, E.: Oblivious hashing revisited, and applications to asymptotically efficient ORAM and OPRAM. In: IACR ASIACRYPT (2017)
11. Chase, M., Kamara, S.: Structured encryption and controlled disclosure. In: IACR ASIACRYPT (2010)
12. Chung, K.M., Liu, Z., Pass, R.: Statistically-secure ORAM with $\tilde{O}(\log^2 n)$ overhead. In: IACR ASIACRYPT (2014)
13. Curtmola, R., Garay, J., Kamara, S., Ostrovsky, R.: Searchable symmetric encryption: improved definitions and efficient constructions. Journal of Computer Security (2011)
14. Dautrich Jr, J.L., Ravishankar, C.V.: Compromising privacy in precise query protocols. In: EDBT (2013)
15. Demertzis, I., Papadopoulos, D., Papamanthou, C., Shintre, S.: SEAL: Attack mitigation for encrypted databases via adjustable leakage. In: Usenix Security (2020)
16. Devadas, S., Dijk, M.v., Fletcher, C.W., Ren, L., Shi, E., Wichs, D.: Onion ORAM: A constant bandwidth blowup oblivious RAM. In: IACR TCC (2016)
17. Gentry, C., Goldman, K.A., Halevi, S., Julta, C., Raykova, M., Wichs, D.: Optimizing ORAM and using it efficiently for secure computation. In: PETS (2013)
18. George, M., Kamara, S., Moataz, T.: Structured encryption and dynamic leakage suppression. In: IACR EUROCRYPT (2021)
19. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM (1996)
20. Goodrich, M.T., Mitzenmacher, M.: Privacy-preserving access of outsourced data via oblivious RAM simulation. In: ICALP (2011)
21. Grubbs, P., Khandelwal, A., Lacharité, M.S., Brown, L., Li, L., Agarwal, R., Ristenpart, T.: Pancake: Frequency smoothing for encrypted data stores. In: Usenix Security (2020)
22. Grubbs, P., McPherson, R., Naveed, M., Ristenpart, T., Shmatikov, V.: Breaking web applications built on top of encrypted data. In: ACM CCS (2016)
23. Grubbs, P., Ristenpart, T., Shmatikov, V.: Why your encrypted database is not secure. In: HotOS (2017)
24. Grubbs, P., Sekniqi, K., Bindschaedler, V., Naveed, M., Ristenpart, T.: Leakage-abuse attacks against order-revealing encryption. In: IEEE S&P (2017)
25. Hamlin, A., Varia, M.: Two-server distributed ORAM with sublinear computation and constant rounds. In: IACR PKC (2021)
26. Heath, D., Kolesnikov, V.: A 2.1 KHz zero-knowledge processor with BubbleRAM. In: ACM CCS (2020)
27. Heath, D., Kolesnikov, V.: PrORAM: Fast $O(\log n)$ private coin ZK ORAM. Cryptology ePrint Archive (2021), https://eprint.iacr.org/2021/587

28. Islam, M.S., Kuzu, M., Kantarcioglu, M.: Access pattern disclosure on searchable encryption: Ramification, attack and mitigation. In: NDSS (2012)
29. Jacob, R., Larsen, K.G., Nielsen, J.B.: Lower bounds for oblivious data structures. In: ACM SODA (2019)
30. Jafargholi, Z., Larsen, K.G., Simkin, M.: Optimal oblivious priority queues. In: ACM SODA (2021)
31. Kamara, S., Moataz, T., Ohrimenko, O.: Structured encryption and leakage suppression. In: IACR CRYPTO (2018)
32. Kellaris, G., Kollios, G., Nissim, K., O'Neill, A.: Generic attacks on secure outsourced databases. In: ACM CCS (2016)
33. Komargodski, I., Lin, W.K.: A logarithmic lower bound for oblivious RAM (for all parameters). In: IACR CRYPTO (2021)
34. Kornaropoulos, E.M., Papamanthou, C., Tamassia, R.: Data recovery on encrypted databases with k-nearest neighbor query leakage. In: IEEE S&P (2019)
35. Kushilevitz, E., Lu, S., Ostrovsky, R.: On the (in) security of hash-based oblivious RAM and a new balancing scheme. In: ACM SODA (2012)
36. Lacharité, M.S., Minaud, B., Paterson, K.G.: Improved reconstruction attacks on encrypted data using range query leakage. In: IEEE S&P (2018)
37. Lacharité, M.S., Paterson, K.G.: A note on the optimality of frequency analysis vs. ℓ_p-optimization. IACR ePrint (2015), http://eprint.iacr.org/2015/1158
38. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: IACR CRYPTO (2018)
39. Larsen, K.G., Simkin, M., Yeo, K.: Lower bounds for multi-server oblivious RAMs. In: IACR TCC (2020)
40. Moataz, T., Mayberry, T., Blass, E.O.: Constant communication ORAM with small blocksize. In: ACM CCS (2015)
41. Patel, S., Persiano, G., Raykova, M., Yeo, K.: PanORAMa: Oblivious RAM with logarithmic overhead. In: IEEE FOCS (2018)
42. Persiano, G., Yeo, K.: Lower bounds for differentially private RAMs. In: IACR EUROCRYPT (2019)
43. Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: IACR CRYPTO (2010)
44. Ren, L., Fletcher, C., Kwon, A., Stefanov, E., Shi, E., Van Dijk, M., Devadas, S.: Constants count: Practical improvements to oblivious RAM. In: Usenix Security (2015)
45. Ren, L., Fletcher, C.W., Yu, X., Van Dijk, M., Devadas, S.: Integrity verification for path oblivious-ram. In: IEEE HPEC (2013)
46. Roche, D.S., Aviv, A., Choi, S.G.: A practical oblivious map data structure with secure deletion and history independence. In: IEEE S&P (2016)
47. Shi, E., Chan, T.H.H., Stefanov, E., Li, M.: Oblivious RAM with $O(\log^3 N)$ worst-case cost. In: IACR ASIACRYPT (2011)
48. Song, D.X., Wagner, D., Perrig, A.: Practical techniques for searches on encrypted data. In: IEEE S&P (2000)
49. Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path ORAM: an extremely simple oblivious RAM protocol. In: ACM CCS (2013)
50. Wagh, S., Cuff, P., Mittal, P.: Differentially private oblivious RAM. Proceedings on Privacy Enhancing Technologies (2018)
51. Wang, X., Chan, H., Shi, E.: Circuit ORAM: On tightness of the goldreich-ostrovsky lower bound. In: ACM CCS (2015)
52. Wang, X., Nayak, K., Liu, C., Chan, T.H., Shi, E., Stefanov, E., Huang, Y.: Oblivious data structures. In: ACM CCS (2014)
53. Weiss, M., Wichs, D.: Is there an oblivious RAM lower bound for online reads? Journal of Cryptology (2021)