Random-Index Oblivious RAM

Shai Halevi
Algorand Foundation, USA

Eyal Kushilevitz
Technion, Israel

July 31, 2022

Abstract

We study the notion of *Random-index ORAM* (RORAM), which is a weak form of ORAM where the Client is limited to asking for (and possibly modifying) random elements of the N-items memory, rather than specific ones. That is, whenever the client issues a request, it gets in return a pair (r, x_r) where $r \in_R [N]$ is a random index and x_r is the content of the r-th memory item. Then, the client can also modify the content to some new value x'_r.

We first argue that the limited functionality of RORAM still suffices for certain applications. These include various applications of sampling (or sub-sampling), and in particular the very-large-scale MPC application in the setting of Benhamouda et al. [2]. Clearly, RORAM can be implemented using any ORAM scheme (by the Client selecting the random r's by himself), but the hope is that the limited functionality of RORAM can make it faster and easier to implement than ORAM. Indeed, our main contributions are several RORAM schemes (both of the hierarchical-type and the tree-type) of lighter complexity than that of ORAM.

1 Introduction

Oblivious RAM (ORAM), introduced by Goldreich and Ostrovsky [12, 19, 13], is a method to compile RAM programs into corresponding "oblivious" programs that keep the same functionality but hide the original access pattern to the memory. This is aimed at preventing leakage of secret information about the data that is revealed by such access patterns and cannot be hidden by merely encrypting the data. ORAM started with software protection motivation in mind, but since then found many applications. In particular, ORAM is used for storing data in cloud applications, where a Client stores in a cloud Server N data items that he owns and it accesses them via a sequence of read/write operations, where the sequence of physical blocks that are read and written does not leak to the Server information about which virtual items the client accesses.

ORAM was the subject of a lot of research, resulting in many schemes (e.g., [12, 19, 16, 22, 9, 23, 5, 20, 1] and *many* more), with the goal of minimizing the overhead incurred by the RAM to ORAM transformation. This beautiful line of work recently culminated with the OptORAMa scheme [1], whose $O(\log N)$ overhead meets the known lower bounds [13, 17].[1] ORAM found many uses beyond cloud storage, for example for efficient MPC protocols, enabling simulating computations that are represented by RAM programs rather than the less efficient circuit representation. Modern ORAM constructions are also sufficiently efficient to be useful in practical systems. Other variants of ORAM

[1] For now, we concentrate only on the overhead in terms of the number of accesses to the memory and ignore other relevant parameters, such as the overhead in Server storage, amount of local memory at the Client, size of each memory item, etc.

(not addressed by the current work) were also considered in the literature, motivated by other models and by applications. Examples include Parallel ORAM [3], Distributed ORAM [18, 7, 14] and others.

To further reduce the overhead below the $\Omega(\log N)$ lower bound (or, more generally, to reduce the practical cost of ORAM schemes), one may consider restricted variants of ORAM that do not provide the full ORAM functionality. One examples is *Offline ORAM* (where the entire sequence is known to the client in advance), for which Boyle and Naor [4] show an improved construction under some assumption related to sorting circuits (in contrast to the lower bound of [13] for offline ORAM in the "balls-and-bins" setting). Another example is *Read only ORAM*, where Weiss and Wichs [24] show an improved construction, based on assumption related to the existence of small sorting circuits and assuming very good locally decodable codes (LDCs),[2] while, again, in the "balls-and-bins" setting, the lower bound of [13] still holds.

1.1 Random-Index ORAM

In this work we introduce another variant, motivated by applications, which we term Random-index-ORAM (RORAM), that only supports random selection of elements from the memory. As in the ORAM setting, the Server stores for the Client a memory that contains N items. Differently from standard ORAM, however, access requests by the Client do not ask for a specific memory location. Instead, whenever the client issues a request, it receives a random sample from the memory. Namely, the client receives a pair (r, x_r), with r a uniformly random location and x_r the data stored in that location. Crucially, the location r is kept hidden from Server/Adversary. (We also allow for Write operations, where the Client can replace the selected value x_r by another value x'_r.) This is clearly solvable by using a standard ORAM scheme; namely, by the client picking the location r at random and using the standard ORAM to read its content. The goal is to improve efficiency beyond that of standard ORAM schemes. Note that the Goldreich-Ostrovsky "balls-and-bins" lower bound [13] applies to RORAM just as for full ORAM, hence we cannot expect to get asymptotic improvement, but we can get significant practical speed-ups, as we show in the sequel.

RORAM variants. We consider two main security notions for RORAM schemes, depending on the application. The natural extension of standard ORAM security demands that the index r remains secret, and the adversary should learn no information about the sequence of locations $r_1, r_2, r_3, \ldots$. (A weaker version of this condition allows the adversary to learn information, so long as the sequence retains enough min-entropy.)

In some applications however (see below), the client's use of the returned entries may reveal to the server the indexes that were received in the past. In those cases, we still need future r_i's look random to the server (till they are used), but we do not care about hiding past indexes. We can therefore use a weaker condition, requiring only that the next r_{i+1} should be pseudorandom/unpredictable given the entire history so far. This weakening enables RORAM protocols that intentionally leak information about past indexes to the server, which may improve performance.

All these notions also have batch variants, where in each step the client receives a batch of items, rather than just one.

[2] Such LDCs are not known to exist but are also beyond current lower bound techniques.

1.2 Applications

Clearly, RORAM is a weaker primitive than ORAM, making it possible to achieve meaningful efficiency gains in applications where the RORAM functionality is sufficient. Indeed, we show in this work how to simplify existing ORAM constructions in this case, e.g. by avoiding the need to build and maintain certain hash tables or recursive structures. We sketch here a few applications where the RORAM functionality is sufficient.

Oblivious statistics. Consider an information-provider with a huge dataset, stored in the cloud, where customers/users may want to make *statistical queries* on this data. (Such queries are very common in data analysis and in machine learning, see e.g. [15, 21].) Concretely, when a query q comes from a user, the provider samples a *random* set S of items from its dataset (of an appropriate size, depending on the desired accuracy) and estimates the answer to q based on this sample. Using RORAM to sample the set S, ensures that S remains hidden, even if the user and the Cloud collude (i.e., it makes it difficult to link the answer to q, known to the user, with specific items).

Sub-Sampling. RORAM is also useful for simulating randomized algorithms that are based on sub-sampling; namely, where the algorithm samples a set S of $N' \ll N$ of the items in a large database, and compute on S. A client with very small space — not even enough to keep the sub-sample — can use RORAM to sample the N' items in S, build an *ORAM* data-set of size N' with the items in S, and execute the computation on S using ORAM. The gain is that this solution pays the ORAM complexity on only N' items, and the smaller RORAM complexity on the larger data-set of N items.

Large-scale secure computation. Our original motivation for studying RORAM comes from the work of Gentry et al. [10], where they studied a similar notion of random-index PIR (RPIR). That work, like ours, was motivated by an application of these notions to very-large-scale secure MPC, specifically for the secrets-on-blockchain architecture of Benhamouda et al. [2]. The architecture from [2] requires periodic random selection of small committees out of a huge population, without the adversary learning who was selected to each committee until after the fact.

A solution to the sampling problem, proposed by Gentry et al. [10], is to assign to the previous committee the task of choosing the next one, as follows:

- The list of parties (or their public keys) is viewed as a public database, from which we seek to sample random entries obliviously.
- The server state is public, making it possible for members of the previous committee to simulate the server actions in their head.
- The client state is shared among the committee, hence client queries and client output are generated via a secure-MPC protocol.

At the conclusion of the selection protocol, the identity (or keys) of the chosen parties for the next committee are known to the client, i.e., they are shared among the previous committee. The adversary, controlling only a minority of the previous committee, does not know who was chosen. The previous committee can then use anonymous public-key encryption over broadcast to transfer

its state to the next committee, thereby "activating" it while keeping the adversary in the dark about who they are (until after they start sending messages).

Using this approach, the communication of the protocol depends only on the client complexity. If the selection protocol features small client circuits, even for large databases, then the overall solution could be sub-linear in the total population size.

Gentry et al. [10] mentioned that RORAM can be used in this fashion instead of RPIR, but did not develop this observation much. In particular, they did not present RORAM constructions (and, moreover, it seems that the RORAM definition from [10] would require implementing a full-fledged ORAM, see more discussion in section 5.1).

A desirable property: bounded history. When using RORAM to implement the above approach, all parties must be able to fully reconstruct the server state when they are called to serve on the committee. It is therefore desirable that this state can be recovered by looking only at the transcript of the last T queries, for some predefined (preferably small) parameter T. This will make it easier for parties to exercise a "lazy" strategy, ignoring the server state when they are not on the committee and reconstructing it only when they are chosen to serve.

1.3 Our Contributions

In this work we study random-index ORAM (RORAM), introduce a few security notions for it, and describe constructions that achieve meaningful speed-ups over full-fledged ORAM.

Our constructions follow the two main types of constructions in the ORAM literature: the *hierarchical*-type ORAM constructions (started from [19, 13]) and the *tree*-type ORAM constructions (started from [22]). In each case, we can forgo some ingredients of the constructions, whose purpose is to locate specific items of the dataset, replacing them with lighter mechanisms that still allow random selection.

- In section 3 we describe two simple hierarchical-RORAM protocols, achieving two different notions of security. In both protocols, we dispose of the hashing steps that are needed in standard hierarchical-ORAM protocols in order to find specific elements in the various levels of the hierarchy. We show that for RORAM it is enough to use a much lighter element-fetching mechanism. We still use the reshuffling procedures from [20, 1, 6] (that already improves over the heavier oblivious sort).

- In section 4 we describe a tree-RORAM protocol, where we can eliminate the recursive construction which is needed for full-fledged ORAM, yielding a $O(\log N)$-factor improvement.

While the constructions that we describe are all quite simple, in some of them the probabilistic analysis of the scheme is a major challenge. We analyze some variants in this work, and leave the analysis of others to future work.

Finally, in section 5 we discuss some questions and directions for future work. One such direction are various possibilities for hybrid ORAM/RORAM schemes that can support the full functionality of ORAM but enjoy the efficiency of RORAM schemes in random selection steps. Other directions include the possibility of constructing ORAM from RORAM (the reverse direction is trivial), issues related to data updates in the context of the application to very-large-scale MPC, and possible directions for improving the analyses.

2 Definitions

Random-index Oblivious-RAM (RORAM) was sketched by Gentry et al. in [10], but our definitions are somewhat different than theirs. A RORAM is a two party protocol between a client and a server, where the server holds the state corresponding to the database, but it does not learn the access pattern of the client. Differently from a full-fledged ORAM, in RORAM the client can only read and write random entries, not specific ones.

Similarly to ORAM, we have procedures for Init, Read, and Write, except that the index to be accessed is not an input to the protocol but an output of it. To allow increasing the database size, we also use $\mathsf{Concatenate}$ operation.[3]

Definition 1 (RORAM Syntax)**.** *A Random-Index ORAM protocol (RORAM) consists of the following components:*

- $\mathsf{Init}(1^\lambda, \mathsf{Db}) \to (\mathsf{cst}; \mathsf{SST})$*: The initialization algorithm takes as input the security parameter and initial database* $\mathsf{Db} \in \{0,1\}^*$ *(that could be empty), and generates an initial secret client state* cst *and a public server state* SST*.*

- $\mathsf{Read}(\mathsf{cst}; \mathsf{SST}) \to ((r, x, \mathsf{cst}'); \mathsf{SST}')$*: The client fetches* (r, x)*, with* x *the element in position* r *in the database (presumably for a uniformly random index* $r \in_R [|\mathsf{Db}|]$*). The client and server states are updated to* $\mathsf{cst}', \mathsf{SST}'$*, respectively.*

- $\mathsf{Write}((\mathsf{cst}, x'(\cdot)); \mathsf{SST}) \to ((r, x, \mathsf{cst}'); \mathsf{SST}')$*: Similar to* Read*, except that in addition to returning the index* r *and previous content* x*, the content of position* r *in the database is replaced by* $x' = x'(r, x)$*. (Note that we let the new value* x' *depend on the old value* x *and its location* r*.*[4]*)*

- $\mathsf{Concatenate}((\mathsf{cst}, x); \mathsf{SST}) \to (\mathsf{cst}'; \mathsf{SST}')$*: The database size is increased by one, and* x *is inserted in the new entry.*

A RORAM protocol is nontrivial if the client and server work in each of these operations is $o(|\mathsf{Db}|)$*.*

2.1 RORAM Security

We consider two notions of RORAM security in this work.

- The weaker notion, motivated by the application to large-scale secure MPC, is *future-randomness*. It asserts that the next index to be returned to the client is random from the server's point of view, even conditioned on all the indexes and elements that were returned in the past.

- The stronger notion, that we call just *randomness*, requires that both future and past indexes are random from the server's point of view.

[3] We could also have a Drop operation to remove elements from the database, but since we cannot target specific elements then dropping will create holes in the database.

[4] Note that x' only depends on (r, x) and not on internals of the RORAM protocol (such as its transcript), since the "higher level" client should not be exposed to these internals.

To see the difference, consider a RORAM protocol in which the client's next query includes the index that it received previously.[5] Such protocol cannot offer randomness, but it can still offer future-randomness.

Similarly to RPIR [10], for RORAM too we can weaken these two notions to only require "high entropy" (or unpredictability) rather than pseudorandomness. We can also look at batch versions, where multiple indexes are returned at once. Below are the formal definitions.

2.1.1 Future Randomness

Here we consider a game in which together with each client query q_j, the server also gets the index r_{j-1} that the client received in the previous step. The (semi honest) server answers all queries as prescribed by the protocol, until it decides to end the game. It then outputs both the answer to the last query q_j (from which the client can deduce (r_j, x_{r_j})), as well as a guess r'_j for the index r_j. We call this the future-randomness game.

Definition 2 (Future randomness)**.** *A RORAM protocol offers future-randomness if for any sequence of queries and any semi-honest PPT server in the future-randomness game, it holds that* $\Pr[r_j = r'_j] \leq 1/N + \mathsf{negl}(\lambda)$*. Here N is the number of elements in the database after the last step, and the probability is taken over all the randomness used by the parties throughout the game.*

2.1.2 Randomness

In the randomness game, we require that the server cannot distinguish the indexes that the client receive from a uniformly random sequence of indexes. Specifically, at the onset of the randomness game, a bit is chosen at random $b \leftarrow \{0,1\}$ and kept secret from the server. If $b = 1$ then the game proceeds similarly to the future-randomness game, where after answering each query q_j the server is shown the index r_j that the client received. If $b = 0$, then instead of r_j, the server is given r'_j which is chosen uniformly at random from $[N_j]$ (where N_j is the number of elements in the database after step j). The game proceeds in this manner until the server decides to end it, outputting a guess b' for b.

Definition 3 (Randomness)**.** *A RORAM protocol offers randomness if for any sequence of queries and semi-honest PPT server in the future-randomness game, it holds that* $|\Pr[b = b'] - \frac{1}{2}| \leq \mathsf{negl}(\lambda)$*. The probability is taken over all the randomness used by the parties throughout the game.*

Both these notions can be relaxed by replacing the uniform distribution by other distributions with sufficient min-entropy.

2.2 Batch RORAM

Many RORAM applications need to draw not just one but many random samples in each step, so it makes sense to try and amortize the lookup cost. It is straightforward to extend the definitions above to the case where each access returns exactly k elements, where k is a parameter. We just replace the single index r_j in step j by a vector of indexes $\vec{r}_j \in [N_j]^k$ (and the corresponding vector of elements $\vec{x}_j$ that are stored in these positions).

[5] Indeed, the most efficient hierarchical-RORAM protocol that we describe in section 3 has exactly that structure, each client query must include the index that the client received in the previous step.

Another useful variant, which we employ in section 4, is where the batch size itself could vary. The syntax is exactly the same as above, except that the batch size k is a random variable, determined by the protocol randomness. We are interested in this new notion in the context of not-quite-random-but-high-entropy distributions, but measuring (min-)entropy in this case is a little awkward. Hence we use a more special-purpose notion of *guessing resilience*, which directly measures what we need in the application to large-scale secure MPC.

This notion, which is a variant of future-randomness, has two parameters $\epsilon \leq \delta$. We consider a server with a "budget" of ϵN elements that it can guess, and bound the probability that this server is able to guess more than a δ-fraction of the samples drawn in the next step.

This security game proceeds similarly to the future-randomness process, where the server gets with each query q_j also all the indexes in R_{j-1} that were received by the client in the previous query. When the server decides to end the game, it outputs a set of indexes $R' \subset [N]$. The server is considered successful if $|R'| \leq \epsilon N$ but $|R \cap R'| \geq \delta |R|$, where R is the set of indexes returned in the last step.

Definition 4. *For parameters $\epsilon \leq \delta$, a RORAM protocol offers (ϵ, δ) guessing-resilience if for any sequence of queries and any semi-honest PPT server in the guessing game above, it holds that* $\Pr[|R'| \leq \epsilon N \text{ but } |R \cap R'| \geq \delta |R|] \leq \mathsf{negl}(\lambda)$.

3 Hierarchical RORAM

In this section, we describe two hierarchical RORAM protocols which are more efficient that full-blown ORAM. Specifically, we try to reduce the use of heavy oblivious sorts and to eliminate the use and maintenance of hash tables in each level of the hierarchy (whose goal, in regular hierarchical ORAM, is to locate the concrete item we search for). We assume that the reader is familiar with the common features of all hierarchical ORAM schemes such as:

- The memory is organized in $O(\log N)$ levels of growing sizes, the i'th level of size 2^i.
- In each step, some items are accessed, and then they are removed from their level and re-inserted into the smallest level.
- Every $O(2^i)$ steps the contents of the first i levels are randomly reshuffled into level $i+1$.

The description below is focused on the new aspects of our protocols.

3.1 Protocol 1 - Future Randomness

We start with a scheme for the future-randomness variant; i.e., where the adversary gets to see the chosen locations *after* they are selected and before the next selection (see definition 2). Since the adversary sees after the fact the actual sequence of locations, then it can also compute which items are contained in what levels of the hierarchy. The only information that is hidden from the adversary is *the order* of items within each level. Whenever a request for a next item comes, the following is executed:

- Server (and Client) know, for each level i, the number of items n_i that reside in this level. But the items in each level are randomly ordered, and this ordering is secret. Client selects the winning level i_0 for this round in proportion to the level size. I.e., each level i is the winner with probability $n_i / \sum_j n_j$.

- Client reads one items from each level (to hide the identity of i_0). For simplicity, say that it reads the last item in the level (i.e., the one at position n_i). It discards all those items except the one read from level i_0.[6]

- After the output is revealed, the Server omits this item from level i_0 (simply by decreasing n_{i_0} by one) and adds it to the top (smallest) level.[7] More concretely, this is done by the client reading the entire top level (which is small), randomly permuting its elements, and rewriting it (and updating its size).

- Finally, every 2^i steps, all levels i and above are merged into level $i+1$ (with the client's help), to avoid overflow. As observed in [20, 1], this operation (termed "intersperse") does *not* require oblivious sort, since it merges randomly permuted arrays. It just need to hide, for each item in the resulting array, from what level it came. Moreover, in our case we also do not need to bring each item to a specific location, determined by the hash, and/or to construct a corresponding hash table; we only need the result to maintain a random order.

  The complexity of intersperse was bounded in PanORAMa [20] by $n \log\log n$ (vs. $n \log n$ for Oblivious Sort), and was further reduced to the asymptotically optimal $O(n)$ in OptORAMa [1].

For correctness, the ides is that selecting each level in proportion to its size, together with the random order within each level, ensures uniform probability for each output element.

The security argument. Recall that we are in the future-randomness model, where the adversary anyway sees each r_i before r_{i+1} is selected. Given the sequence so far $(r_1, \ldots, r_i)$ (which implies the knowledge of what level contains each item), the actions that occur *before* giving r_{i+1} to the Client (namely, obliviously reading the "last" item from each level) give the adversary no information and so each element has probability $1/N$ to be the next item.

Theorem 1. *Protocol 1 is a RORAM protocol satisfying future-randomness, and overhead of* $O(\log N)$.

3.2 Protocol 2 - Randomness

In the protocol above, the server learns after the fact the level from which the item r_i was selected (as this information can be computed by knowing r_i). In the case of the randomness game (definition 3), where the adversary does not get the r_i's, we must in particular hide the winning label i_0 from the server. Hence, the server no longer knows exactly how many elements are in each level, so it cannot just read "the last item" from each level as it did above.

A naive attempt to fix the protocol is for the server to read "the next item" from each level in each step. Namely, in the first step after levels $i-1$ and above are merged into level i, the server will read the first item from level i, then the second item, the the third, and so on. With this

[6] We can further reduce the time in this step at the cost of a (small) penalty in space. That is, if Client has a little extra memory, it could keep the values of all read items (one from each level) from one invocation to the other, instead of reading them again in the next invocation and only update in the next invocation the single item that is currently used (which by then will anyway be known to the adversary).

[7] Note that in this way we refrain from the need to handle so-called "dummy" items, in standard ORAM constructions.

protocol, one could hope that level i will contain enough items whp, so the server will be able to keep reading them for 2^i steps, until that level is merged to the level below. Unfortunately, this is not the case. In fact, it can be shown that no matter the ratio between the level sizes, and no matter how often levels are merged into the levels before, level i will always run out of items before the next time that it is merged into level $i+1$.

Rather than reading one element at a time, we therefore switch to reading a "window of elements" from each level. The idea is that (1) most read elements are not really used, since we read from all buffers at each step and only select one (according to buffer sizes). (2) if the buffer is of size X, we expect to use it with probability X/n, i.e. once every n/X steps (which overcomes the previous problems when reading a new element in each step). Since this is only in expectation, if we start by reading λ elements from the buffer and then, every n/X steps read one more element (and maintain the "window" of last λ elements), we will argue that whp we will always have enough elements to read. (The client rewrites everything that is read, and the selected item is replaced by "dummy", so the Server does not know which one was actually used; if the client reads from some buffer it takes the first non-dummy element.)

In more detail, each level i has size 2^i, where we start from level $i \approx \log \lambda$ (of size λ, security parameter) and end in level $L = \log n$ of size n (that contains all elements).

- Every 2^{i+1} steps we empty all levels i and above into level $i+1$. This means that each level i is empty for 2^i steps, and then level $i-1$ and above are emptied into level i. The only exception here is level L which is never emptied and every n steps all elements are coming back to it.

- At each step, a level j is selected with probabilities proportional to level sizes (which are known to the client). One item from winning level is then accessed and moved to the root, see below for details on which item is read. (Reading from level L is simple: just one-by-one for n steps and then when it is re-filled we start over.)

For each level we maintain a public window of size λ (the security parameter) which are the items that the server will read and send to the client. For each level i we have a public parameter ρ_i which is the rate of advancing the window. (We set ρ_i slightly above $2^i/n$; the exact value to be determined below.) As mentioned, level i is filled in step 2^i and emptied in step 2^{i+1}. Thereafter in each step $2^i + t$ (for $t < 2^i$) we read the window $[s_i, s_i + \lambda - 1]$ from this level, where $s_i = \lfloor \rho_i \cdot t \rfloor$.

We are also keeping for each level i a secret pointer p_i to the next element to read, this pointer is known to the client but not the server. When the level is filled in step 2^i the pointer is set to $p_i = 1$, and thereafter it is advanced whenever we actually access an item from that level (i.e., that level was selected), or when p_i lags behind the rear of the window. (This ensures that we always have $p_i \geq s_i$.)

We note that security of this protocol is completely straightforward, as the server never sees any non-encrypted content and its access pattern is deterministic: In each step it reads from each level i all the items in the window $[s_i, s_i + \lambda - 1]$. The hard part is proving correctness, i.e. that every step indeed returns some item to the client. The rest of this section is devoted to proving it, yielding the following theorem:

Theorem 2. *Protocol 2 is a RORAM protocol satisfying the randomness property, and overhead of $O(\log N)$.*

Invariants. Recall that for each level i we have at any point a public window $[s_i, s_i + \lambda - 1]$, and a secret pointer p_i (all indexes into the i'th level). What we need to prove is that in every step, some element in indeed returned to the client. A sufficient condition for this method to work, is maintaining these two conditions:

1. The front of the window never exceeds the number of elements in the level, until the level is emptied to the next one. Denoting by S_i the number of items in level i when it is filled in step 2^i, we show that whp we have $S_i \geq \lfloor \rho_i \cdot 2^i \rfloor + \lambda$, which ensures that we have enough elements for 2^i steps.

2. The pointer p_i is always included in the current window $[s_i, s_i + \lambda - 1]$. Since $p_i \geq s_i$ by definition, this is reduced to showing that whp we always have $p_i < s_i + \lambda$.

Analysis of size behavior.

- Level L is of size n: it starts with all n elements, and every n steps all elements are back and we start another such phase.

- Level $i < L$ has capacity 2^i. Every 2^{i+1} steps it is being emptied to the level below (together with all levels above it), and every 2^i steps it is being filled from the levels above. This means that, if we look at 2^{i+1} consecutive steps, during the first 2^i steps this level is empty, and then we insert into it everything from above. This can be at most 2^i elements, if in all 2^i steps the selected elements are taken from levels below i. After another 2^i steps, this level is emptied again.

Let's consider an interval that begins when level i is emptied, so all the elements are at lower levels. Let us compute the expected number of elements that *remain below levels* i during the 2^i steps before that level is filled. (These are the elements that *will not be* in level i when it is filled.)

For each element $j \in [n]$, let X_j be a characteristic random variable of the event that j was never chosen in any of the 2^i steps (so it remained below level i). Then the expected value of each X_j is $E[X_j] = \Pr[X_j = 1] = (1 - 1/n)^{2^i}$. The size of level i when it is filled is exactly $n - \sum_j X_j$, so the expected size of level i is $n - \sum_j E[X_j] = n(1 - (1 - 1/n)^{2^i})$.

Recalling that $(1 - 1/n)^{2^i} = \big((1 - 1/n)^n\big)^{2^i/n} \leq (1/e)^{2^i/n}$, the expected size of level $L - i$ when it is filled is therefore bounded from below by (and very close to) $n(1 - (1/e)^{2^{-i}})$. To give a few example, denote by S_i the size of level i when it is filled, and its expected value by $\mu_i = E[S_i]$, then we have

- $\mu_{L-1} \geq n\big(1 - \sqrt{1/e}\big) \approx 0.393n \approx n/2.5$
- $\mu_{L-2} \geq n\big(1 - \sqrt[4]{1/e}\big) \approx 0.221n \approx n/4.5$
- $\mu_{L-3} \geq n\big(1 - \sqrt[8]{1/e}\big) \approx 0.118n \approx n/8.5$
- $\mu_{L-4} \geq n\big(1 - \sqrt[16]{1/e}\big) \approx 0.061n \approx n/16.5$
- ...

It is not hard to see that $\mu_{L-i} < 2^{-i}n$: By symmetry, all elements $j \in [n]$ have the same probability of being chosen at least once, so we might as well look at the probability that a *random element* is chosen at least once. We have 2^{L-i} steps until we fill level $L-i$, so we can choose at most 2^{L-i} distinct elements. Hence for a random element, the probability of it being chosen (at least once) cannot be more than $2^{L-i}/n = 2^{L-i}/2^L = 2^{-i}$.

By a similar argument, μ_{L-i} gets closer to $2^{-i}n$ as i grows: Let $\#\mathsf{col}_i$ be the expected number of collisions when choosing 2^{L-i} elements at random with repetitions (i.e., the number of times we've chosen an element that was already chosen before). Considering again a random element, we have $\mu_{L-i} = 2^{-i}n - \#\mathsf{col}_i$. Increasing i by one (so halving the number of elements chosen) decreases the expected number of collisions by more than a factor of two (roughly by a factor of four). Hence $\mu_{L-i} = 2^{-i}n(1-\epsilon_i)$ with ϵ_i monotonically decreasing in i. We saw above that already for $i=2$ we have $\mu_{L-2} \geq 2^{-2}n \cdot 0.88$, and it gets closer to $2^{-i}n$ as i increases.

Beyond computing the expected value of the size S_i of level i, we need to also get high-probability bounds on S_i. To that end, we would like to use the fact that $S_i = \sum_j X_j$ (with X_j the characteristic random variables from above) and apply Chernoff bound, but the X_j's are not quite independent.

Luckily, we can use the results from [8] to argue that they are negatively associated, and therefore the Chernoff bound apply to them as well. Specifically, [8, Thm 13] shows that the random variables N_j =#-of-times-element-i-was-chosen are negatively associated. Then Proposition 7 implies that also the indicators X_j are negatively associated (since each X_j is a monotonic function of the corresponding N_j). Finally, Proposition 5 says that Chernoff/Hoeffding bounds therefore apply to the sum of the X_j's. We conclude that for all i and every $0 < \delta \leq 1$:

$$\Pr[S_i > \mu_i(1+\delta)] < \exp\big(-\delta^2\mu_i/3\big). \tag{1}$$

$$\Pr[S_i < \mu_i(1-\delta)] < \exp\big(-\delta^2\mu_i/2\big) \tag{2}$$

We will use the first equation to prove Invariant 1 and the second equation to prove Invariant 2. Let us set the rate of advancing the window at level i to:

$$\rho_i = 3/2 \cdot 2^i/n \text{ for } i \leq L-2, \text{ and } r_{L-1} = 0.54.$$

With these rates, and assuming that the levels are large enough so for all i we have $2^i > 4\lambda$ and also $n > 25\lambda$, we can conclude the following:

- For $i \leq L-2$, we have $2^i \leq n/4$ and $2^i > \mu_i > 2^i \cdot 0.88$. Together with $2^i \geq 4\lambda$ we get

$$\begin{aligned} \lfloor \rho_i \cdot 2^i \rfloor + \lambda &\leq 3/2 \cdot 2^{2i}/n + \lambda \leq 2^i \cdot (3/2 \cdot 2^i/n + \lambda/2^i) \\ &\leq 2^i \cdot (3/8 + 1/4) < \mu_i/0.88 \cdot 0.625 < \mu_i(1-0.289). \end{aligned}$$

  Hence

$$\begin{aligned} \Pr[S_i < \lfloor \rho_i 2^i \rfloor + \lambda] &< \exp(-0.289^2\mu_i/2) < \exp(-0.289^2 \cdot 0.88 \cdot 2^i/2) \\ &< \exp(-0.289^2 \cdot 0.88 \cdot 2\lambda) < \exp(-\lambda/7). \end{aligned}$$

- For $i = L-1$ we have $2^i = n/2$, $\rho_i = 0.54$ and $\mu_i > 0.39n$. Assuming $n > 25\lambda$ we get

$$\lfloor \rho_i 2^i \rfloor + \lambda \leq 0.27n + n/25 = 0.31n < \mu_i(1/4 + 1/24)/0.39 < \mu_i(1-0.2).$$

Hence

$$\begin{aligned}\Pr[S_i < \lfloor \rho_i 2^i \rfloor + \lambda] &< \exp(-0.2^2\mu_i/2) < \exp(-0.2^2 \cdot 0.39n/2) < \exp(-0.2^2\mu_i/2) \\ &< \exp(-0.2^2 \cdot 0.39 \cdot 25\lambda/2) < \exp(-\lambda/6).\end{aligned}$$

To analyze the 2nd invariant, fix some level i and consider the case where this invariant is violated at some step between 2^i and 2^{i+1}. Let $t_0 < 2^i$ be such that step $2^i + t_0$ was the last time that we advanced the next-element-to-read pointer p_i due to lagging behind the rear of the window ($t_0 = 0$ if it never lagged). Let t_1 be such that step $2^i + t_1$ is the first time that p_i exceeded the front of the window, and note that between steps $2^i + t_0$ and $2^i + t_1$ we only advanced the pointer due to reading elements from level i. Denote $\Delta = t_1 - t_0$.

Hence the number of elements that we read from that level during the interval $[2^i + t_0, 2^i + t_1]$ was $\lambda + \lfloor \rho_i \cdot t_1 \rfloor - \lfloor \rho_i \cdot t_0 \rfloor \geq \lambda + \rho_i\Delta - 1$. Since in each step we read an element with probability at most S_i/n, then the number of elements read is bounded below a binomial random variable with Δ trials and success probability S_i/n. We can bound the probability of violating the 2nd invariant by

$$\begin{aligned}&\Pr[\text{Invariant 2 is violated between steps } 2^i \text{ and } 2^{i+1}] \\ &\leq \Pr[\exists t_0, \Delta, \text{ s.t. } t_0 + \Delta < 2^i \text{ and } \mathsf{Bin}(S_i/n, \Delta) > \lambda + \rho_i\Delta] < 2^{2i} \Pr[\mathsf{Bin}(S_i/n, \Delta) > \lambda + \rho_i\Delta].\end{aligned}$$

Obviously for any p, x we have $\mathsf{Bin}(p, x) \leq x$ (with probability one). So $\mathsf{Bin}(S_i/n, \Delta) > \lambda + \rho_i\Delta$ has positive probability only when $\Delta \geq \lambda/(1 - \rho_i)$. Using Inequality (1) and the choice of ρ_i values from above:

- For $i \leq L - 2$, assuming $2^i > 4\lambda$ we have $2^i \geq \mu_i \geq 0.88 \cdot 2^i > 3.52\lambda$, so

  $$\Pr[S_i > 5/4 \cdot 2^i] < \Pr[S_i > \mu_i(1 + 1/4)] < \exp(-\mu_i/48) < \exp(-\lambda/14).$$

  Moreover, with $\rho_i = 3/2 \cdot 2^i/n$, if $\gamma := S_i/n \leq 5/4 \cdot 2^i/n = \rho_i/1.2$ then we get

  $$\Pr[\mathsf{Bin}(\gamma, \Delta) > \lambda + \rho_i\Delta] < \Pr[\mathsf{Bin}(\gamma, \Delta) > \gamma(\lambda/\gamma\Delta + 1.2)\Delta] \leq \Pr[\mathsf{Bin}(\gamma, \Delta) > \gamma(16\lambda/5\Delta + 1.2)\Delta],$$

  where the last inequality follows since, for $i \leq L - 2$, we have $\gamma = 5/4 \cdot 2^i/n \leq 5/16$. Recalling that $\Delta > \lambda$,we consider three cases:

  - $\Delta > \lambda > \Delta/4$: In this case $16\lambda/5\Delta + 1.2 > 2$ and therefore

    $$\Pr[\mathsf{Bin}(\gamma, \Delta) > \lambda + \rho_i\Delta] < \Pr[Bin(\gamma, \Delta) > 2\gamma\Delta] < \exp(-\Delta/3) < \exp(-\lambda/3).$$

  - $\Delta/4 \geq \lambda > \Delta/16$: In this case $16\lambda/5\Delta + 1.2 > 1.4$ and therefore

    $$\Pr[\mathsf{Bin}(\gamma, \Delta) > \lambda + \rho_i\Delta] < \Pr[Bin(\gamma, \Delta) > 1.4\gamma\Delta] < \exp(-0.16\Delta/3) \leq \exp(-\lambda/5).$$

  - $\Delta/16 \geq \lambda$: In this case, we trivially have $16\lambda/5\Delta + 1.2 > 1.2$ and therefore

    $$\Pr[\mathsf{Bin}(\gamma, \Delta) > \lambda + \rho_i\Delta] < \Pr[Bin(\gamma, \Delta) > 1.2\gamma\Delta] < \exp(-0.04\Delta/3) \leq \exp(-\lambda/5).$$

  In any case, we get

  $$\begin{aligned}&\Pr[\text{Invariant 2 is violated between steps } 2^i \text{ and } 2^{i+1}] \\ &\qquad < 2^{2i}\big(\exp(-\lambda/14) + \exp(-\lambda/5)\big) < n^2/8 \cdot \exp(-\lambda/14).\end{aligned}$$

- For $i = L-1$ we have $0.39n < \mu_i < 0.4n$ and hence (assuming $n > 25\lambda$) we get

$$\begin{aligned}\Pr[S_i > 0.46n] &< \Pr[S_i > 0.46 \cdot \mu_i/0.40] < \Pr[S_i > \mu_i(1+0.15)] \\ &< \exp(0.15^2\mu_i/3) < \exp(0.15^2 \cdot 0.39n/3) \\ &< \exp(0.15^2 \cdot 0.39 \cdot 25\lambda/3) < \exp(-0.002n) < \exp(-\lambda/14).\end{aligned}$$

  Moreover, with $\rho_i = 0.54$, if $\gamma := S_i/n \leq 0.46 < \rho_i/1.17$ then we get

$$\Pr[Bin(\gamma,\Delta) > \lambda + \rho_i\Delta] < \Pr[Bin(\gamma,\Delta) > \gamma(\lambda/\Delta\gamma + 1.17)\Delta].$$

  The same case analysis as above (but using 1.17 instead of 1.2) implies that here we have in all cases $\Pr[Bin(\gamma,\Delta) > \lambda + \rho_i\Delta] < \exp(-\lambda/7)$. Hence for $i = L-1$ we have

$$\begin{aligned}&\Pr[\text{Invariant 2 is violated between steps } 2^i \text{ and } 2^{i+1}] \\ &\quad < 2^{2i}\big(\exp(-\lambda/14) + \exp(-\lambda/7)\big) < n^2/2 \cdot \exp(-\lambda/14).\end{aligned}$$

This concludes the proof of theorem 2. □

improving the concrete efficiency by improved parameters. The analysis above yields failure probability exponentially small in λ, but the constants are not great (i.e., $\exp(-\lambda/14)$). This can be improved a lot in various ways. Some potential approaches include:

- Making the one-but-last level smaller: The parameters for higher-up levels are much better than for level $L-1$, if instead we make the level above the leaves of size only $n/4$ rather than $n/2$ then the parameters will improve. (The cost is that merging into the last level will happen twice as often.)
- For the same reason, we can make do with smaller windows at higher levels than we do at lower ones.
- The procedure for advancing the windows could be changed, so that we move it faster at first (when the level has more elements in it) and slower later on.
- Violating the first invariant is not necessarily a failure, what we really care about is the next-element-to-read pointer p_i exceeding S_i, not necessarily the front of the window.

3.2.1 Bounded History

We note that for both protocols in this section, the entire state of the server can be reconstructed by looking at the recent n operations (or less). Specifically, any history that includes the last intersperse operation from levels $L-1$ and above to level L, has enough information to reconstruct all the ciphertexts in all the levels.

4 Tree-Based RORAM

Below we introduce a class of simple tree-base RORAM schemes, then describe in detail and analyze one specific scheme in this class.

Recall that in tree-ORAM schemes ([22] and follow-up schemes), data items are held in the nodes of a binary tree. At any point, each data item is assigned to one leaf in the tree, and it can be found somewhere on the path from the root to its assigned leaf. A data-access operation consists of looking up a single root-leaf path in the tree, extracting the relevant data item from it, then assigning that element to a new random leaf and pushing it back at the root of the tree. After each data access, a maintenance process is invoked to push elements down the tree towards their assigned leaves, so as to prevent overflow at the top levels.

Full-fledged tree-ORAM must be able to determine *which root-leaf path to read* in order to find specific data items. This is solved via a recursive structure with smaller and smaller trees, where each tree contains information about where to find data items in the next larger tree. This solution, however, adds a $O(\log N)$ factor to the ORAM overhead.

4.1 A Class of Tree-RORAM Schemes

We note that since RORAM schemes do not need to look up specific elements, there is no real need for the recursive construction. Instead, here we consider schemes that employ just one tree with all the data items, and where each step just looks up an arbitrary leaf (either a random one or according to some deterministic order). Each operation therefore consists of the following steps:

1. Determine the leaf to read, and look up the path in the tree from the root to that element;
2. Extract one or more data items from this path and return (their encryption) to the client;
3. Assign a fresh random leaf to each extracted element (keep this information within the element), optionally change the data stored in the element, and then push the element back at the root of the tree;
4. Invoke the maintenance process to push elements down towards their assigned leaves.

This class of schemes has several different variations:

- The next leaf to read can be chosen at random in every step, or the scheme can use a round-robin ordering of leaves;
- When reading a root-leaf path, it can return to the client either one data item from the ones assigned to the target leaf, or a batch of some fixed number of them (a parameter k), or all the data items that are assigned to that leaf;
- The elements that were not returned to the client (if any), could either be left in their place, or extracted and assigned new leaves as well;
- Finally, another source of variation is the maintenance process.

Not a perfect RORAM. It is important to note that RORAM schemes from the class above *do not result in a completely uniformly random choice* of data items from the server's point of view. To see that, notice that choosing a random data item would imply a non-uniform distribution on the leaf which is read next (since some leaves will have more items assigned to them than others). Conversely, choosing a uniform leaf (or using a deterministic ordering) implies a non-uniform probability distribution over the chosen data item.

4.2 The Scheme that we Analyze

In this work, we only present one scheme from the class above that arguably features the easiest analysis. This is a "worst case scheme", where the server is provided with as much information as possible (and hence unfortunately it also has the worse parameters).

Specifically, we consider the variant where the leaves are accessed in a round-robin fashion, where all the data-items that are assigned to the target leaf are extracted and returned to the client, and we consider the forward-randomness game where the server gets to see all these items after they are given to the client.

In terms of the maintenance process, we adopt the process of Gentry et al. from [11], which also use a deterministic leaf traversal, specifically bit-reverse ordering. Namely, with a binary tree on L leaves, we name each leaf by a $\log L$ bit string, describing the path from the root (MSB) to that leaf (LSB). For example with $L = 2^8$, leaf 100 is obtained by the path right-left-left from the root, and leaf 001 is obtained by the path left-left-right. With this representation, the leaves are accessed in reverse bit ordering, namely $\mathsf{bitReverse}(0), \mathsf{bitReverse}(1), \ldots, \mathsf{bitReverse}(L-1)$. In the example with $L = 2^3$, the order will be $(000, 100, 010, 110, 001, 101, 011, 111)$, or in numbers $(0, 4, 2, 6, 1, 5, 3, 7)$.

Hence the scheme that we consider for the rest of this section is as follows. It maintain an L-leaf binary tree, holding N items in total, where each node can hold upto m data items. m and L are parameters, TBD later as a function of N and the security parameter, with L a power of two. The i'th data-access operation ($i = 0, 1, 2, \ldots$) is implemented as follows:

1. Let $j = \mathsf{bitReverse}(i \bmod L)$. Read the path from the root to leaf j, extract from it all the data items that are assigned to leaf j, and return then to the client;
2. Assign to each of these data items a fresh uniform leaf in $[L]$, and place them all back at the root of the tree (after possibly updating the data in them);
3. Push each data item on the root-to-j path as far down toward its assigned leaf as it can go along that path (i.e., an item assigned to j' is pushed as far down as $\mathsf{commonPrefix}(j, j')$);
4. Write the updated path back to the tree.

Note that as opposed to the future-randomness hierarchical-RORAM protocol from section 3.1, the protocol here does not rely on the server learning the past indexes. Nonetheless, below we analyze it only in the forward-randomness setting where all these indexes are revealed to the server after the fact.

4.2.1 A Technical Lemma

Underlying most of our analysis in this section is the following simple technical lemma, bounding the probability of finding one specific element in one specific leaf. This lemma is independent of the server's view, or the number of elements that are returned to the client. It depends only on the fact that all the elements are evicted from the leaves that we examine, and are then assigned independently to fresh random leaves.

Lemma 1. *Fix N, L, and an arbitrary starting assignment of elements in leaves. Assume that the first leaf to be examined is leaf zero. Then for every element, the probability that it will be assigned to leaf zero the next time that it is examined, is between $1/L$ and e/L.*

Note that the lemma talks about the next time that leaf zero is examined, not this time. That is, we consider some initial configuration at step 0 where leaf zero is examined, and assert that in step L, when leaf zero will be examined again, each item has probability between $1/L$ and e/L of being there.

Proof. For $i = 0, 1, ..., L-1$, let p_i be the probability that an element which is currently assigned to leaf i will be assigned to leaf 0 the next time that we look at it (not this time). We can define p_i inductively as follows:

- $p_{L-1} = 1/L$;
- $p_i = 1/L \cdot \sum_{j=i+1}^{L-1} 1/L \cdot p_j$.

Solving this recurrence, we get $p_i = \frac{(1+1/L)^{L-i}}{L}$, hence $1/L \leq p_i \leq e/L$ holds for all i. □

4.3 Bounding the Prediction Probability

Recalling that the scheme above has a variable batch size, we next analyze its security in terms of definition 4. Namely, motivated by the application to large-scale secure-MPC, we focus below on a setting where the adversary has a "budget" of upto ϵN guesses, and we bound the probability that it guesses more than a δ-fraction of the elements that are returned in the next data access. The rest of this subsection is devoted for proving the following:

Theorem 3. *For every constant $\delta > 0$, there is another constant $\epsilon = \Omega(\delta^2)$, such that the RORAM scheme above offers (ϵ, δ)-guessing resilience as per definition 4.* □

To simplify the analysis, we assume at first that the nodes have infinite capacity, so we can ignore issues of overflow. As argued below, standard analysis shows that for appropriate setting of the node-size m, overflow would only happens with negligible probability. Assuming no overflow, we can ignore the tree altogether and consecrate only on the leaves. That is, we consider a process in which an element which is assigned to some leaf is immediately placed in that leaf. For simplicity below we also ignore the reverse-bit-ordering of leaves (which is only needed to argue about overflow probability). Instead we consider the natural ordering of leaves, $0, 1, 2, \ldots, L-1$.

Recall again that below we only analyze the "pessimistic" case where the adversary sees all the elements that are extracted from the previous leaves. Other cases where it only sees a few of these elements clearly reduces the server's ability to guess, but we were not able to analyze this improvement. We leave it as an interesting open question.

4.3.1 The Optimal Adversary

When the adversary sees all the extracted elements, it is easy to describe the distribution over the locations of all the elements, conditioned on the adversary's view: Those elements that were last seen $L-1$ steps ago must all be in the next leaf, elements that were last seen $L-2$ steps ago are uniformly distributed between the next leaf and the one after that, elements that were last seen $L-3$ steps ago are uniform over the next three leaves, etc. In general, the location of an element that was last seen $(L-j)$ steps ago is uniformly distributed among the next j leaves. Moreover,

the location of the different elements are all independent of each other, even conditioned on the adversary's view.[8]

Hence, the optimal strategy for an adversary that tries to guess the elements placed in the next leaf is as follows: The adversary first guesses all the elements that were seen $L-1$ steps ago, followed by those that were seen $L-2$ steps ago, then $L-3$, etc., until the entire budget of ϵN guesses is exhausted.

4.4 Analysis of the optimal strategy

The number of correct guesses that the optimal strategy yields is a sum of $B = \epsilon N$ Bernoulli random variables, with the ones corresponding to elements seen $(L-j)$ steps ago having success probability of $1/j$. To bound the success probability of this optimal strategy, it remains just to bound the number of elements of each type.

Fix some arbitrary initial configuration from at least $2L$ steps ago, and consider the process starting from that configuration, until the current step. To fix the indexing, assume that the leaf to be examined next is leaf 0. Below we first devise a high-probability upper bound on the number of leaves that the strategy above considers before running out of budget (call that upper-bound J). Then we show an upper bound on the number of elements from these J leaves that will end up in leaf 0, and lower-bound the number of elements from the other $L-J$ leaves, hence getting an upper bound on the fraction of correct guesses.

For $i \in [1, N]$ and $j \in [1, L]$, denote by χ_{ij} the indicator random variable which is one if element i was last seen $L-j$ steps ago (i.e., the last time that we examined leaf j mod L), and zero otherwise. The same analysis as in Lemma 1 implies that starting from any arbitrary initial configuration, the probability that element i was found in leaf j mod L the last time that we examined it, is bounded between $1/L$ and e/L. If it happens to be there, then the probability of not seeing it again until the current step is exactly j/L. Hence we have

- For any i, j, $\Pr[\chi_{ij} = 1] \in [j/L^2, ej/L^2]$.
- Any set of χ_{ij}'s with distinct indexes i are independent, since the locations of different elements are independent.
- For any fixed i the variables $\chi_{i0}, \ldots, \chi_{i,L-1}$ are negatively associated (as they sum up to one).

Given the properties above, we can use the Chernoff bound to reason about sums of these variables. For all the lemmas below, let $\epsilon < 1/4$ be a constant, and let λ be the security parameter.

Fix N, L such that $N > L$, and denote $B = \lfloor \epsilon N \rfloor$. We start by devising an upper-bound on the number of leaves that the adversary considers before running out of guessing budget. That is, we denote by X_j the number of elements that were last seen $L-j$ steps ago (namely $X_j = \sum_{i=1}^{n} \chi_{ij}$), and establish a number J such that whp $\sum_{j=1}^{J} X_i > B$.

Lemma 2. *Assume that $N > 4\lambda \ln(2)/\epsilon$. Using the notations above and setting $J = \lceil 2L\sqrt{\epsilon} \rceil$ (and noting that $J < L$ since $\epsilon < 1/4$), we have $\Pr[\sum_{j=1}^{J} X_j < B] < 2^{-\lambda}$.*

[8] When the adversary only sees a few of the extracted elements, the location of the different elements may not be independent when conditioned on the view. One example is provided in Fig. 1 in the appendix.

Proof. The sum $S = \sum_{j=1}^{J} X_j = \sum_{j=1}^{J} \sum_{i=1}^{N} \chi_{ij}$ is a sum of $N \cdot J$ negatively associated Bernoulli random variables, with the success probability of each χ_{ij} between j/L^2 and ej/L^2. Denoting the expected value of the sum by $\mu = E[S]$, we therefore have

$$\mu \geq \sum_{j=1}^{J} \frac{j}{L^2} \cdot N = N \binom{J+1}{2} / L^2 > N/2 \cdot (J/L)^2.$$

Using the Chernoff bound with $\delta = 0.5$, we have $\Pr[S < \mu/2] \leq \exp(-\mu/8)$. The proof now follows just by plugging the values of λ and J: Recall that $J = \lceil 2L\sqrt{\epsilon} \rceil$ and therefore $(J/L)^2 \geq (2L\sqrt{\epsilon}/L)^2 = 4\epsilon$. This implies that $\mu \geq N/2 \cdot (J/L)^2 \geq N/2 \cdot 4\epsilon = 2\epsilon N$.

On the one hand, since $N > 4\lambda \ln(2)/\epsilon$, then $\mu/8 \geq \epsilon N/4 > \lambda \ln(2)$ and hence $\exp(-\mu/8) \leq 2^{-\lambda}$. On the other hand, we have $B \leq \epsilon N \leq \mu/2$. Putting them together, we get

$$\Pr[S < B] \leq \Pr[S < \mu/2] \leq \exp(-\mu/8) \leq 2^{-\lambda},$$

as needed. □

For the next two lemmas, let $\gamma_{i,j}$ be the indicator random variable which is one if element i was last seen $L - j$ steps ago *and is currently found in leaf 0*, and zero otherwise. As we explain above, if element i was last seen $L - j$ steps ago then the probability of finding it now in leaf 0 is exactly $1/j$. Therefore γ_{ij} is an AND of χ_{ij} and an independent Bernoulli variable with success probability $1/j$, which means that $\Pr[\gamma_{ij} = 1] \in [1/L^2, e/L^2]$. Also, just like for the χ_{ij}'s, γ_{ij} is independent of all the $\gamma_{i'j'}$'s with $i' \neq i$, and negatively associated with the $\gamma_{ij'}$'s for $j' \neq j$. Hence we can use the Chernoff bound on their sums, as we do in the next two lemmas. It will be convenient to denote by Y_j the number of elements that were last seen $L - j$ steps ago and are found in leaf 0, namely $Y_j = \sum_{i=1}^{N} \gamma_{ij}$.

Lemma 3. *With the setting above, assuming that* $N/L > \frac{3\ln(2)}{2\sqrt{\epsilon}} \cdot \lambda$ *and letting* $\alpha = 12\sqrt{\epsilon} \cdot N/L$, *then* $\Pr[\sum_{j=1}^{J} Y_j > \alpha] < 2^{-\lambda}$.

Proof. The sum $S = \sum_{j=1}^{J} Y_j = \sum_{j=1}^{J} \sum_{i=1}^{N} \gamma_{ij}$ is a sum of $N \cdot J$ negatively associated Bernoulli random variables, each with the success probability between $1/L^2$ and e/L^2. Hence the expected value of the sum is $\mu = E[S] \in [JN/L^2, JNe/L^2]$, and using the Chernoff bound with $\delta = 1$ we have $\Pr[S > 2\mu] \leq \exp(-\mu/3)$.

Substituting $J = \lceil 2L\sqrt{\epsilon} \rceil$ we have $\mu \geq JN/L^2 \geq 2\sqrt{\epsilon}N/L$, and also $\mu \leq JNe/L^2 \leq 6\sqrt{\epsilon}N/L$. On the one hand, since $N/L > \frac{3\ln(2)}{2\sqrt{\epsilon}} \cdot \lambda$, then $\mu/3 \geq 2\sqrt{\epsilon}N/3L \geq \lambda \ln(2)$ so $\exp(-\mu/3) \leq 2^{-\lambda}$. On the other hand, we have $\alpha = 12\sqrt{\epsilon} \cdot N/L \geq 2\mu$. Putting them together, we get

$$\Pr[S > \alpha] \leq \Pr[S > 2\mu] \leq \exp(-\mu/3) \leq 2^{-\lambda},$$

as needed. □

Lemma 4. *With the setting above, assuming that* $N/L \geq \frac{8\ln(2)}{(1-3\sqrt{\epsilon})} \cdot \lambda$ *and letting* $\beta = \frac{(1-3\sqrt{\epsilon})}{2} \cdot N/L$, *then* $\Pr[\sum_{j=J+1}^{L} Y_j < \beta] < 2^{-\lambda}$.

Proof. The sum $S = \sum_{j=J+1}^{L} Y_j = \sum_{j=J+1}^{L} \sum_{i=1}^{N} \gamma_{ij}$ is a sum of $N \cdot (L-J)$ negatively associated Bernoulli random variables, each with success probability between $1/L^2$ and e/L^2. Hence the expected value of the sum is $\mu = E[S] \in [(L-J)N/L^2, (L-J)Ne/L^2]$, and using the Chernoff bound with $\delta = 0.5$ we have $\Pr[S < \mu/2] \leq \exp(-\mu/8)$.

Substituting $J = \lceil 2L\sqrt{\epsilon} \rceil$ we have $\mu \geq (L-J)N/L^2 \geq (1-3\sqrt{\epsilon})N/L$. On the one hand, since $N/L \geq \frac{8\ln(2)}{(1-3\sqrt{\epsilon})} \cdot \lambda$, then $\mu/8 \geq (1-3\sqrt{\epsilon})N/8L \geq \lambda \ln(2)$ so $\exp(-\mu/8) \leq 2^{-\lambda}$. On the other hand, we have $\beta = \frac{(1-3\sqrt{\epsilon})}{2} \cdot N/L \leq \mu/2$. Putting them together, we get

$$\Pr[S < \beta] \leq \Pr[S < \mu/2] \leq \exp(-\mu/8) \leq 2^{-\lambda},$$

as needed. □

The next lemma concludes the proof of theorem 3.

Lemma 5. *Let $\epsilon \leq 1/9$ be a constant and λ the security parameter. Fix N, L such that $L > \frac{3}{8\sqrt{\epsilon}}$ and $N/L \geq \max\{\frac{3\ln(2)}{2\sqrt{\epsilon}}, \frac{8\ln(2)}{(1-3\sqrt{\epsilon})}\} \cdot \lambda$. Also, fix some arbitrary initial configuration from at least $2L$ steps ago, and consider the process starting from that configuration, until the current step. Denote $\delta = \frac{24\sqrt{\epsilon}}{1+21\sqrt{\epsilon}} < 1$.*

Then, an adversary that sees all the elements that were extracted in previous steps and can guess upto ϵN elements, has probability at most $3 \cdot 2^{-\lambda}$ of guessing more than an δ-fraction of the elements that will be extracted in the current step.

Proof. The conditions on the quantities N, L, ϵ, λ ensure that all the conditions in Lemmas 2, 3, and 4 are satisfied. Hence the conclusions in all these lemmas hold, except perhaps with probability of $3 \cdot 2^{-\lambda}$.

By Lemma 2 for the optimal adversary strategy above, the adversary exhausts all the guessing budget on elements that were last seen $L-j$ steps ago, for $j = 1, 2, \ldots, J$. By Lemma 3, at most α of the elements that were last seen in those steps will be found in the next leaf, and by Lemma 4 at least β other elements will be fond there. Hence the fraction of elements that the adversary guesses is at most

$$\frac{\alpha}{\alpha+\beta} = \frac{12\sqrt{\epsilon} \cdot N/L}{12\sqrt{\epsilon} \cdot N/L + \frac{(1-3\sqrt{\epsilon})}{2} \cdot N/L} = \frac{24\sqrt{\epsilon}}{1+21\sqrt{\epsilon}} = \delta.$$

□

A remark about constants. The analysis above is very loose, giving up on many constants (on top of using the rather loose Chernoff bound). For example, to ensure that the adversary cannot guess more than $\delta = 1/2$ the elements in the leaf, the theorem above requires that the adversary's budget be limited to only $N/729$. In any real application, the constants will of course be determined by simulation rather than by the above. Some initial simulations that we ran indicate that the real number is something like $\epsilon \approx 2\delta^2$ (so to get $\delta = 1/2$ we need $\epsilon \approx 1/8$).

4.4.1 Bounded History

Since the protocol above examines the leaves in a round-robin fashion, then the entire state of the server can be reconstructed by looking only at the last L steps of the protocol.

5 Discussion and Future Directions

Before concluding, we discuss below a few other related topics and point out possible future directions and open problems.

5.1 Hybrid ORAM/RORAM Schemes

An interesting direction is constructing a full-fledged ORAM scheme, but such that accessing a random element costs a lot less than a specific element. This would be used to speed up access-oblivious randomized algorithms (e.g. quicksort) that interleave random selections with accessing specific elements.

If the underlying dataset is fixed and need not be updated, then a simple solution is to just keep two copies of the dataset, one in RORAM for the random selections and the other in a full-fledged ORAM for fetching specific elements.

If updates are needed, however, then keeping such separate structures no longer works, since we will not be able to update specific elements in the RORAM structure. In that case, we can still offer some savings by keeping in each structure enough extra information with each element to find that element in the other structure. For example, if the other structure is a hierarchical ORAM then we can keep information about which level it is found at and where in that level, thus avoiding the need to consult the hash tables.

Similarly, if the other structure is a (recursive) tree ORAM, then we can keep information about which leaf that element in assigned to (in all the recursive levels). This way, once we locate an element in the non-recursive RORAM tree, we can update all the recursive trees in a full ORAM in one round, rather than having to interact for $O(\log N)$ communication rounds to find it.

Note, however, that keeping the two structures synchronized in this way means that we have to update the heavier ORAM structure every time we read from the lighter RORAM structure, thereby negating much of the RORAM savings.

5.2 Refreshing Keys in Large-Scale MPC

Using RORAM for large-scale MPC as sketched in the introduction brings up the question of how to handle key-refresh by parties. Recall that in that application, the RORAM structure is holding the public keys of all the parties. Since RORAM cannot fetch/update specific elements, it is not clear how can parties refresh their keys and get the new key into the RORAM.[9]

One solution is to only let parties refresh their keys when they are selected to the committee, using the RORAM write operation. This may be sensible in setting where parties serve on committees often enough.[10]

Another option is to handle periodic key-refresh by keeping two structures at any point, the current RORAM that we use and the next one that we are building. The next-RORAM will be called on the entire new database of keys, running the Init operation (and it can do this operation "in the background", a little bit at a time). Once the new RORAM is fully built, we switch to it as the one in use, discard the old one, and begin building the next RORAM after that.

[9] Most likely, this is the reason why the RORAM definition from [10] features random reads but writes of specific elements, which seem to require the hybrid RORAM/ORAM solution.

[10] For example, in the realistic threat model where fail-stop attacks are easy to mount but real compromise rarely happens, it may be okay to have parties refresh their keys very rarely, when they are selected to the committee.

5.3 Improved Schemes and Analysis

Regarding the tree constructions, it is very likely that giving the adversary less information will result in much better parameters (in terms of the dependence of ϵ, δ). For example, instead of revealing all the elements that were extracted from the last leaf, we can reveal (say) only half of them, but still extract and re-assign all the elements from the current leaf. Or maybe even, only extract and re-assign half the elements from the current leaf, leaving the rest of them in place. In all of these options, it is not longer easy to describe the probability distribution over the location of elements, conditioned on the server's view, or to figure out the optimal server guessing strategy. Moreover, the locations of different elements are no longer independent of each other conditioned on this view, complicating the analysis further. (See one example in fig. 1 in the appendix.) Getting a good handle over the whole class of protocols of this form is an interesting open problem.

5.4 From RORAM to ORAM

One natural question is whether we can construct full-fledged ORAM scheme from RORAM in a black-box manner. For RPIR, Gentry et al. described a RPIR-to-PIR black box transformation, but it requires $O(N)$ work for the server so we cannot use it in the RORAM/ORAM world. Finding such black-box constructions (or an argument why they are unlikely) is still an open problem.

References

[1] G. Asharov, I. Komargodski, W. Lin, K. Nayak, E. Peserico, and E. Shi. Optorama: Optimal oblivious RAM. In A. Canteaut and Y. Ishai, editors, *Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part II*, volume 12106 of *Lecture Notes in Computer Science*, pages 403–432. Springer, 2020.

[2] F. Benhamouda, C. Gentry, S. Gorbunov, S. Halevi, H. Krawczyk, C. Lin, T. Rabin, and L. Reyzin. Can a public blockchain keep a secret? In R. Pass and K. Pietrzak, editors, *Theory of Cryptography - 18th International Conference, TCC 2020, Durham, NC, USA, November 16-19, 2020, Proceedings, Part I*, volume 12550 of *Lecture Notes in Computer Science*, pages 260–290. Springer, 2020.

[3] E. Boyle, K. Chung, and R. Pass. Oblivious parallel RAM and applications. In E. Kushilevitz and T. Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 175–204. Springer, 2016.

[4] E. Boyle and M. Naor. Is there an oblivious RAM lower bound? In M. Sudan, editor, *Proceedings of the 2016 ACM Conference on Innovations in Theoretical Computer Science, Cambridge, MA, USA, January 14-16, 2016*, pages 357–368. ACM, 2016.

[5] S. Devadas, M. van Dijk, C. W. Fletcher, L. Ren, E. Shi, and D. Wichs. Onion ORAM: A constant bandwidth blowup oblivious RAM. In E. Kushilevitz and T. Malkin, editors, *Theory of Cryptography - 13th International Conference, TCC 2016-A, Tel Aviv, Israel, January 10-13, 2016, Proceedings, Part II*, volume 9563 of *Lecture Notes in Computer Science*, pages 145–174. Springer, 2016.

[6] S. Dittmer and R. Ostrovsky. Oblivious tight compaction in o(n) time with smaller constant. In C. Galdi and V. Kolesnikov, editors, *Security and Cryptography for Networks - 12th International Conference, SCN 2020, Amalfi, Italy, September 14-16, 2020, Proceedings*, volume 12238 of *Lecture Notes in Computer Science*, pages 253–274. Springer, 2020.

[7] J. Doerner and A. shelat. Scaling ORAM for secure computation. In B. M. Thuraisingham, D. Evans, T. Malkin, and D. Xu, editors, *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 523–535. ACM, 2017.

[8] D. P. Dubhashi and D. Ranjan. Balls and bins: A study in negative dependence. *Random Struct. Algorithms*, 13(2):99–124, 1998. Available from https://www.brics.dk/RS/96/25/BRICS-RS-96-25.pdf.

[9] C. Gentry, K. A. Goldman, S. Halevi, C. S. Jutla, M. Raykova, and D. Wichs. Optimizing ORAM and using it efficiently for secure computation. In E. D. Cristofaro and M. K. Wright, editors, *Privacy Enhancing Technologies - 13th International Symposium, PETS 2013, Bloomington, IN, USA, July 10-12, 2013. Proceedings*, volume 7981 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2013.

[10] C. Gentry, S. Halevi, B. Magri, J. B. Nielsen, and S. Yakoubov. Random-index PIR and applications. In K. Nissim and B. Waters, editors, *Theory of Cryptography - 19th International Conference, TCC 2021, Raleigh, NC, USA, November 8-11, 2021, Proceedings, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 32–61. Springer, 2021.

[11] C. Gentry, S. Halevi, M. Raykova, and D. Wichs. Outsourcing private RAM computation. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 404–413. IEEE Computer Society, 2014.

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

[13] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *J. ACM*, 43(3):431–473, 1996.

[14] S. D. Gordon, J. Katz, and X. Wang. Simple and efficient two-server ORAM. In T. Peyrin and S. D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part III*, volume 11274 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2018.

[15] M. J. Kearns. Efficient noise-tolerant learning from statistical queries. *J. ACM*, 45(6):983–1006, 1998.

[16] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In Y. Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 143–156. SIAM, 2012.

[17] K. G. Larsen and J. B. Nielsen. Yes, there is an oblivious RAM lower bound! In H. Shacham and A. Boldyreva, editors, *Advances in Cryptology - CRYPTO 2018 - 38th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2018, Proceedings, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 523–542. Springer, 2018.

[18] S. Lu and R. Ostrovsky. Distributed oblivious RAM for secure two-party computation. In A. Sahai, editor, *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, volume 7785 of *Lecture Notes in Computer Science*, pages 377–396. Springer, 2013.

[19] R. Ostrovsky. Efficient computation on oblivious RAMs. In H. Ortiz, editor, *Proceedings of the 22nd Annual ACM Symposium on Theory of Computing, May 13-17, 1990, Baltimore, Maryland, USA*, pages 514–523. ACM, 1990.

[20] S. Patel, G. Persiano, M. Raykova, and K. Yeo. Panorama: Oblivious RAM with logarithmic overhead. In M. Thorup, editor, *59th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2018, Paris, France, October 7-9, 2018*, pages 871–882. IEEE Computer Society, 2018.

[21] L. Reyzin. Statistical queries and statistical algorithms: Foundations and applications. *CoRR*, abs/2004.00557, 2020.

[22] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In A. Sadeghi, V. D. Gligor, and M. Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013.

[23] X. Wang, T. H. Chan, and E. Shi. Circuit ORAM: on tightness of the goldreich-ostrovsky lower bound. In I. Ray, N. Li, and C. Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, pages 850–861. ACM, 2015.

[24] M. Weiss and D. Wichs. Is there an oblivious RAM lower bound for online reads? In A. Beimel and S. Dziembowski, editors, *Theory of Cryptography - 16th International Conference, TCC 2018, Panaji, India, November 11-14, 2018, Proceedings, Part II*, volume 11240 of *Lecture Notes in Computer Science*, pages 603–635. Springer, 2018.

A Non-Independence for a Tree-RORAM Construction

As mentioned in section 5.3, we show here an example of a tree-RORAM scheme where the location of the different elements is *not independent* when conditioned on the server's view. Specifically, the variant here shows the server one element out of each leaf, but still evict and re-assigns all the elements from the current leaf.

First step (sees #1)	Second step (sees #2)	Pr[this history]	Pr[#3 in leaf1 | this history]	Pr[#1, #2 in leaf2 | this history]
[1,2,3][·]	[1,3] [2]	$\frac{1}{8}\cdot\frac{1}{3}\cdot\frac{1}{8}\cdot 1 = \frac{1}{192}$	1	0
[1,2,3][·]	[1] [2,3]	$\frac{1}{8}\cdot\frac{1}{3}\cdot\frac{1}{8}\cdot\frac{1}{2} = \frac{1}{384}$	1/2	0
[1,2,3][·]	[3] [1,2]	$\frac{1}{8}\cdot\frac{1}{3}\cdot\frac{1}{8}\cdot\frac{1}{2} = \frac{1}{384}$	1	1/4
[1,2,3][·]	[·][1,2,3]	$\frac{1}{8}\cdot\frac{1}{3}\cdot\frac{1}{8}\cdot\frac{1}{3} = \frac{1}{576}$	1/2	1/4
[1,2] [3]	[1] [2,3]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{1}{128}$	1/2	0
[1,2] [3]	[·][1,2,3]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{3} = \frac{1}{192}$	1/2	1/4
[1,3] [2]	[1,3] [2]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot 1 = \frac{1}{64}$	1	0
[1,3] [2]	[1] [2,3]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{1}{128}$	1/2	0
[1,3] [2]	[3] [1,2]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{2} = \frac{1}{128}$	1	1/4
[1,3] [2]	[·][1,2,3]	$\frac{1}{8}\cdot\frac{1}{2}\cdot\frac{1}{4}\cdot\frac{1}{3} = \frac{1}{192}$	1/2	1/4
[1] [2,3]	[1] [2,3]	$\frac{1}{8}\cdot 1\cdot\frac{1}{2}\cdot\frac{1}{2} = \frac{1}{32}$	1/2	0
[1] [2,3]	[·][1,2,3]	$\frac{1}{8}\cdot 1\cdot\frac{1}{2}\cdot\frac{1}{3} = \frac{1}{48}$	1/2	1/4
Pr[#3 in leaf1 | view] = 0.6374			Pr[#3 in leaf1 | #1, #2 in leaf2 & view] = 0.62	

A 2-step process with 3 elements and 2 leaves, starting from a random placement, where the adversary sees element #1 coming out of leaf1 and element #2 coming out of leaf2.

Figure 1: An example of dependence between different elements.