Multi-User Oblivious RAM Secure Against Malicious Servers

Erik-Oliver Blass
Airbus Innovation
erik-oliver.blass@airbus.com
2nd. author

Travis Mayberry
Northeastern University
travism@ccs.neu.edu 3rd. author

Guevara Noubir
Northeastern University
noubir@ccs.neu.edu

ABSTRACT

It has been an open question whether Oblivious RAM stored on a malicious server can be securely shared among multiple users. ORAMs are stateful, and users need to exchange updated state to maintain security. This is a challenge, as the motivation for using ORAM is that the users may not have a way to directly communicate. A malicious server can potentially tamper with state information and thus break security. We answer the question of multi-user ORAM on malicious servers affirmatively by providing several new, efficient multi-user ORAM constructions. We first show how to make the original square-root solution by Goldreich and the hierarchical one by Goldreich and Ostrovsky multi-user secure. We accomplish this by separating the *critical* parts of the access, which depends on the state of the ORAM, from the non-critical parts that can be safely executed in any state. Our second and main contribution is a multi-user variant of Path ORAM. To enable secure meta-data update during evictions, we employ our first result, small multi-user secure classical ORAMs, as a building block. Depending on the block size, the overhead of our construction reaches a low $O(\log n)$ communication complexity per user, similar to state-of-the-art single-user ORAMs.

1. INTRODUCTION

Oblivious RAM (ORAM) has been a popular research topic for a number of years. Yet, only recently it has come close to being practical by offering sublinear worst-case communication complexity. Recent papers have tackled further efficiency improvements, reducing burdensome memory restrictions and adding the ability to seamlessly scale database sizes up and down. However, at least one significant hurdle to adoption remains: security of modern ORAMs relies on there being only a single user at all times. The problem stems from the fact that, in order to hide the access pattern, ORAM algorithms must modify some of the data on the server after every access. If the server is permitted to, e.g., "rewind" the data and present an old version to a user, further interactions may reveal something about their access pattern. Fortunately, with a single user this is easily solved by storing a small token, such as the root of a hash tree [9]. This token authenticates and verifies freshness of all data retrieved from the server, ensuring that no such rewind attack is possible.

However, with multiple users, an authentication token is not enough. Data may not pass authentication for a valid reason: it has been modified by one of the other users. If users could communicate with each other using a secure out-of-band channel, then it becomes possible to continually exchange and update each other with the most recent token. However, existence of secure out-of-band-communication is not always a reasonable assumption. If users already have a secure method of continuously communicating with each other, one may argue that ORAM may not even be needed in the first place. Current solutions for multi-user ORAM work only in the presence of an honest-but-curious adversary, which cannot perform rewind attacks on the users. Often, this is not a very satisfying model, since rewind attacks are very easy to execute for real-world adversaries and would be difficult to detect. Goodrich et al. [4], in their paper examining multi-user ORAM, recently proposed as an open question whether one could be secure for multiple users against a malicious server.

Technical Highlights: In this paper, we introduce the first construction for a multi-user ORAM. We prove security even if the server is fully malicious. Our contribution is twofold, specifically:

- First, we focus on two ORAM constructions that follow a "classical" approach, the *square-root* ORAM by Goldreich [1] and the *hierarchical* ORAM by Goldreich and Ostrovsky [2]. We adapt these ORAMs for multi-user security. The intuition is to separate user accesses into two parts. One part can be performed securely in the presence of a malicious server. The other part cannot be performed securely, but contains an efficient check which will reveal any malicious behavior, thereby allowing the user to terminate the protocol.

- The "classical" ORAM constructions have been largely overshadowed by more recent tree-based ORAMs. Tree-based ORAMs such as the one by Shi et al. [10] and Stefanov et al. [11] and many derivatives, provide better efficiency and worst-case guarantees. Consequently, we go on to demonstrate how a multi-user secure tree-

based ORAM can be constructed using one of the "classical" ORAMs as a building block. For reasonably small block sizes, this results in a multi-user ORAM which has similar per-user complexity to state of the art single-user ORAM constructions, namely $O(\log n)$ overhead.

2. MULTI-USER ORAM

ORAM protocols provide security because they are highly stateful. In order to hide the fact that a user accesses a certain data block, ORAMs typically perform shuffling or reordering of blocks so that two accesses are not recognizable as being the same. An obvious attack that a malicious server can do is to undo or "rewind" that shuffling after the first access and present the same, original view of the data to the user when they make the second access. If the user was to blindly execute their access, and it was the same block of data as the first access, it would result in the same pattern of interactions with the server that the first access did. The server would immediately have broken the security of the scheme. This is a straightforward attack, and it is easily defeated by having the user store a token for authentication and freshness [9].

However, with two (or more) users, the server can execute the same attack, but against the two users separately. After watching one user retrieve some data, he can rewind the ORAM's state and present the original view to the second user. If the second user accesses the same data that the first user did, the server will recognize it and break security. Without having some secure side-channel to exchange authentication tokens after every access, it is difficult for users to detect such an attack.

2.1 Security Definition

We start by briefly recalling the standard Oblivious RAM concept. An ORAM provides an interface to read from and write to blocks of a RAM (an array of storage blocks). It supports $\mathsf{Read}(x)$, to read from the block at address x and $\mathsf{Write}(x, v)$ to write value v to block x. The ORAM allows storage of n blocks, each of size B. To securely realize this functionality, an ORAM outsources a state Σ to an untrusted storage. For convenience, state Σ can be represented as a sequence of fixed-length strings. We will call the untrusted storage provider a *server* in this paper because the most likely application for a multi-user ORAM would be outsourced cloud storage.

DEFINITION 1 (ORAM OPERATION OP). *An operation* OP *is defined as* $\mathrm{OP} = (o, x, v)$*, where* $o = \{\mathsf{Read}, \mathsf{Write}\}$*,* x *is the virtual address of the block to be accessed and* v *is the value to write to that block.* $v = \perp$ *when* $o = \mathsf{Read}$*.*

We now present our multi-user ORAM security definition which slightly augments the standard, single-user ORAM definition.

DEFINITION 2 (MULTI-USER ORAM Π). *A multi-user ORAM* Π *is defined by tuple* $\Pi = (\mathsf{Init}, \mathsf{Access})$*.*

1. $\mathsf{Init}(\lambda, n, B, \psi)$ *initializes* Π*. It takes as input security parameter* λ*, total number of blocks* n*, block size* B*, and number of users* ψ*.* Init *outputs an initial ORAM state* Σ_{init}*, which encompasses the entirety of the ORAM that is stored on the server, and a list of per user states* $\{st_{u_1}, \ldots, st_{u_\psi}\}$ *which are kept local to the individual users.*
2. $\mathsf{Access}(\mathrm{OP}, \Sigma, st_{u_i})$ *performs operation* OP *on ORAM state* Σ *using user* u_i*'s state* st_{u_i}*.* Access *outputs (1) an access pattern* $< (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >$*, where* (α_j, ν_j) *denotes that the string at position* α_j *in state* Σ *is read from or replaced by string* ν_j*, and (2) a new state* st_{u_i} *for user* u_i*.*

In contrast to single-user ORAM, a multi-user ORAM introduces the notion of users. This is modeled by different per-user states, st_{u_i} for user u_i. Algorithm Init outputs different initial states st_{u_i} and, in practice, would distribute these initial states to users. One can assume this distribution during initialization taking place over a secure out-of-band communication channel. However after initialization, users cannot use an out-of-band channel anymore. Whenever user u_i executes Access on the multi-user ORAM, they can only update their own state st_{u_i}.

Finally, we define the security of a multi-user ORAM against malicious servers. Consider the following experiment $\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda)$.

DEFINITION 3 (MULTI-USER ORAM $\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda)$).

$b \xleftarrow{\$} \{0,1\}$
$(\Sigma_{\text{init}}, st_{u_1}, \ldots, st_{u_\psi}) \leftarrow \mathsf{Init}(\lambda, n, B, \psi)$
$(\Sigma, \mathrm{OP}_0, \mathrm{OP}_1, i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(\lambda, n, B, \Sigma_{\text{init}}, \psi)$
for $j = 1$ **to** $poly(\lambda)$ **do**
  $(st_{u_i}, < (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >) \leftarrow \mathsf{Access}(\mathrm{OP}_b, \Sigma, st_{u_i})$
  $(\Sigma, \mathrm{OP}_0, \mathrm{OP}_1, i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(< (\alpha_1, \nu_1), \ldots, (\alpha_m, \nu_m) >, st_{\mathcal{A}})$
end
$b' \leftarrow \mathcal{A}(st_{\mathcal{A}})$
output 1 iff $b = b'$

An ORAM $\Pi = (\mathsf{Init}, \mathsf{Access})$ *is multi-user secure* iff *for all PPT adversaries* $\mathcal{A}$

$$Pr[\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda) = 1] < \frac{1}{2} + \epsilon(\lambda),$$

where ϵ *is a negligible function in security parameter* λ*.*

First, a random bit b is chosen, and both the ORAM and adversary $\mathcal{A}$ are initialized. Then, $\mathcal{A}$ gets oracle access to the ORAM and can adaptively query it during $poly(\lambda)$ rounds. In each round, $\mathcal{A}$ selects a user u_i, determines two operations OP_0 and OP_1, and outputs an ORAM state Σ. The oracle performs operation OP_b as user u_i with state st_{u_i} and ORAM state Σ using protocol Π. The oracle returns access pattern (α_i, ν_i) induced by Π back to $\mathcal{A}$. Each tuple (α_i, ν_i) tells the adversary which part of Σ was read or overwritten (with value ν). Eventually, $\mathcal{A}$ guesses b.

Our game-based definition is equivalent to the standard ORAM security definition with two exceptions: we allow the adversary to arbitrarily change the state of the on-server storage

Σ, and we split the ORAM algorithm into ψ different pieces which cannot share state between themselves.

Note that, in this work, we assume that all users trust each other and do not conspire. For ease of exposition, we assume that all user share a key κ used for encryptions, decryptions, and MAC computations that we will introduce later.

Non-Goal Consistency: While certainly important, we stress that consistency issues are a non-goal in this paper. Informally, $\mathcal{A}$ could present different versions of ORAM state Σ to different users, leading to desynchronization and inconsistent views of the ORAM. Not surprisingly, it is impossible to protect against desynchronization in the absence of out-of-band communication. The strongest consistency achievable would be *fork* consistency. For more information, see Li et al. [6]. In this paper, we allow $\mathcal{A}$ to desynchronize users, as long as security following Definition 3 is not violated: users' access patterns must never be revealed. Along the same lines, "reaction-attacks" [5] (and variants) are out of scope. In a reaction attack, $\mathcal{A}$ would send an old state Σ to a user, observing the user's reaction. For example, the old state could force higher applications layers into accessing the ORAM more (or less) often. Again, in this work, we only aim at accesses pattern protection, but cannot enforce a specific number of accesses.

3. MULTI-USER SECURITY FOR CLASSICAL ORAMS

We start by demonstrating how two existing ORAM constructions, the original square-root solution by Goldreich [1] and the hierarchical one by Goldreich and Ostrovsky [2], can be transformed into multi-user secure versions with the same complexity per user.

3.1 Overview

We briefly review Goldreich [1]'s square-root ORAM. In this ORAM, storage is divided into two partitions. One holds $n+\sqrt{n}$ blocks, and we will call this *main memory*. The other partition holds $\sqrt{n}$ blocks, the *cache*. When the ORAM is initialized, a random permutation π on n elements is sampled, and each block x is stored encrypted in the main memory at location $\pi(x)$. In addition, we store $\sqrt{n}$ encrypted *dummy* elements at positions $\pi(n+1), \ldots, \pi(n+\sqrt{n})$. The cache is also encrypted, initially holding $\sqrt{n}$ empty blocks. Every access $\mathsf{Read}(x)$ that the (single) user u makes is then broken down into two steps. First, u retrieves the cache in its entirety, decrypts it, and searches to see whether the cache contains block x. If it is not, u retrieves block $\pi(x)$ from the main memory and inserts it into the local cache. If x is already in the cache, u simply reads its value from the local cache and performs an additional read of the next dummy element $\pi(x'), x' \in \{n+1, \ldots, n+\sqrt{n}\}$, from main memory. This hides the fact that u found the block they wanted in the cache. The cache is then finally re-encrypted and stored back in the server.

A $\mathsf{Write}(x, v)$ to the ORAM is implemented similarly. User u again downloads the cache, decrypts it, and stores block x with its new value v in it. Before re-encrypting and uploading the cache back to the server, u also reads the next dummy element from main memory. To always know which dummy element is the next dummy element, the user can store an (encrypted) dummy counter inside the cache. To ease exposition, we will omit this detail in our protocol description below. For integrity verification, recall that u typically needs to compute and store an authentication token (e.g., a MAC) of the latest version of the cache in their local memory [9].

After $\sqrt{n}$ operations, the cache will become full. At that point, the user downloads the cache and the entirety of the main memory. The user merges main memory and cache together by emptying the cache and reshuffling main memory with a new permutation π. For blocks which are in both the cache and main memory, the cache version will be newer and so is taken to replace the old version.

For blocks of size B bit, the amortized complexity of this scheme is $\tilde{O}(\sqrt{n} \cdot B)$, and worst-case complexity is $O(n \cdot B)$. Each access requires $O(\sqrt{n})$ blocks, but every $O(\sqrt{n})$ operations require an $O(n)$ access. Security is straightforward, because the user never reads the same block twice from main memory without an intervening complete shuffle. The cache is read and re-encrypted in its entirety for each operation. In effect, the cache acts like a "trivial" ORAM.

Challenge. When considering a multi-user scenario, it becomes very easy for a malicious server to break security of the square-root ORAM. For example, user u_1 can access a block x that is not in the cache, requiring u_1 to read $\pi(x)$ from main memory and insert it into the cache. The malicious server now restores the cache to the state it was in before u_1's access added block x. If a second user u_2 also attempts to access block x, the server will now observe that both users read from the same location in main memory and know that u_1 and u_2 have accessed the same block. Without the users having a way to communicate directly with each other and pass information that allows them to verify the changes to the cache, the server can always "rewind" the cache back to a previous state. This will eventually force one user to leak information about their accesses.

Rationale. Our approach for multi-user security is based on the observation that the *cache update* part of the square-root solution is secure by itself. Updating the cache only involves downloading the cache, changing one element in it, re-encrypting, and finally strong it back with the server. Downloading and later uploading the cache implies always "touching" the same $\sqrt{n}$ blocks, independently of what the malicious server presents to a user as Σ, and also independent of the block being updated by the user. Changing values inside the cache cannot leak any information to the server, as its content is always newly IND-CPA encrypted. Succinctly, being similar to a trivial ORAM, updating a cache is automatically multi-user secure.

However, any reading can leak information. Reading from main memory is conditional on what the user finds in the cache. We call this part the *critical* part of the access, and the cache update correspondingly *non-critical*. To counteract this leakage, we implement the following changes to enable multi-user support for the square-root ORAM:

Input: Security parameter λ, number of blocks in each ORAM n, block size B, number of users ϕ
Output: Initial ORAM state Σ_{init}, initial per user states $\{st_{u_1}, \ldots, st_{u_\phi}\}$

$\kappa \xleftarrow{\$} \{0,1\}^\lambda$;
for $i := 1$ *to* ϕ **do**
  Generate permutation $\pi_{i,0}$ from key κ;
  Initialize $\sqrt{n}+n$ main memory blocks, shuffled with $\pi_{i,1}$, and $\sqrt{n}$ cache blocks;
  Set cache counter $\chi_i = 0$;
  Set epoch counter $\gamma_i = 0$;
  $\mathsf{ORAM}_i = \mathsf{Enc}_\kappa(\text{main memory}) \,||\, \mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_i \,||\, \gamma_i)$;
  $mac_i = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_i \,||\, \gamma_i))$;
  Send $st_{u_i} = \{\kappa, \chi_i\}$ to user u_i;
end
Send $\Sigma_{\text{init}} = \{(\mathsf{ORAM}_1, mac_1), \ldots, (\mathsf{ORAM}_\phi, mac_\phi)\}$ to server;

Algorithm 1: $\mathsf{Init}(\lambda, n, B, \phi)$ – Initialize ORAM

1. **Separate ORAMs:** Instead of a single ORAM, we use ϕ separate ORAMs, $\mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_\phi$, one for each user. Each user u_i will perform the critical part of the access only on their own ORAM, that is, ORAM_i's main memory and cache. Thus, each user can guarantee they will not read the same address from their ORAM's main memory twice. However, any change to the cache as part of ORAM $\mathsf{Read}(x)$ or $\mathsf{Write}(x, v)$ operations will be written to every ORAM's cache. Updating the cache on any ORAM is already guaranteed to be multi-user secure and does not leak information.

2. **Authenticated Caches:** For each user u_i to guarantee that they will not repeat access to the main memory of ORAM_i, the cache is stored together with an encrypted *access counter* χ on the server. Each user stores locally a MAC over both the cache and the encrypted access counter χ of their own ORAM. Every access to their own cache increments the counter and updates the MAC. Since users read only from their own ORAMs, and they can always verify the counter value for the last time that they performed a read, the server cannot roll back beyond that point. Two reads will never be performed with the cache in the same state.

3.2 Details

We detail the above ideas in two algorithms: Algorithm 1 shows the initialization procedure, and Algorithm 2 describes the way a user performs an access with our multi-user secure square-root ORAM.

First, we introduce the notion of an *epoch*. After $\sqrt{n}$ accesses to an ORAM, its cache is "full", and the whole ORAM needs to be re-shuffled. Re-shuffling requires computing a new permutation π. Per ORAM, a permutation can be used for $\sqrt{n}$ operations, i.e., one *epoch*. The next $\sqrt{n}$ operations, i.e., the next epoch, will use another permutation and so on. In the two algorithms, we use an epoch counter γ_i. Therewith, π_{i,γ_i} denotes the permutation of user u_i in ORAM_i's epoch γ_i. For any user, to be able to know the current epoch of ORAM_i, we store γ_i together with the ORAM's cache on the server.

On a side note, we point out that there are various ways to

Input: Address x, new value v, user u_i, $st_{u_i} = \{k, \chi_i\}$
Output: The value of block x, new state st_{u_i}

From ORAM_i: read $c_i = \mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_i \,||\, \gamma_i)$ and mac_i;
$mac'_i = \mathsf{MAC}_\kappa(c_i)$;
if $mac'_i \neq mac_i$ **then**
  Abort ;
end
Decrypt c_i to get cache and counter χ'_i;
if $\chi'_i < \chi_i$ **then**
  Abort ;
end
if *block* $x \notin$ *cache* **then**
  Read and decrypt block $\pi_{i,\gamma_i}(x)$ from ORAM_i's main memory ;
else
  Read next dummy block from ORAM_i's main memory;
end
if $v = \perp$ **then** `// operation is a Read`
  $\nu \leftarrow v$;
else `// operation is a Write`
  $\nu \leftarrow$ existing value of block x;
end
Append block (x, ν) to cache;
if *cache is full* **then**
  $\gamma_i = \gamma_i + 1$;
  Compute new permutation π_{i,γ_i};
  Read and decrypt ORAM_i's main memory;
  Shuffle cache and main memory using $\pi_{i,\gamma}$;
  Send Enc_κ(main memory) to server to update ORAM_i;
end
$\chi_i = \chi'_i + 1$;
$mac_i = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_i \,||\, \gamma_i))$;
Send new $\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i)$ and mac_i to server to update ORAM_i;
for $j \neq i$ **do** `// for all` $\mathsf{ORAM}_j \neq \mathsf{ORAM}_i$
  Read and decrypt cache and χ_j from ORAM_j;
  Read and verify mac_i from ORAM_j ;
  Append block (x, ν) to cache;
  if *cache is full* **then**
    $\gamma_j = \gamma_j + 1$;
    Compute new permutation π_{j,γ_j};
    Read and decrypt ORAM_j's main memory;
    Shuffle cache and main memory using π_{j,γ_j};
    Send Enc_κ(main memory) to server to update ORAM_j;
  end
  $mac_j = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_j \,||\, \gamma_j))$;
  Send new $\mathsf{Enc}_\kappa(\text{cache} \,||\, \chi_j \,||\, \gamma_j)$ and mac_j to server to update ORAM_j;
end
output $(\nu, st_{u_i} = \{\kappa, \chi_i\})$;

Algorithm 2: $\mathsf{Access}(\text{OP}, \Sigma, st_{u_i})$ – Perform Read or Write

generate pseudo-random permutations π_{i,γ_i} on n elements in a deterministic fashion. For example, one can use $\text{PRF}_\kappa(i||\gamma_i)$ as the seed in a PRG and therewith perform a Fisher-Yates shuffle.

In addition to the epoch counter, we also introduce a per user *cache counter* χ_i. Using χ_i, user u_i counts the number of accesses of u_i to the main memory and cache of their own ORAM_i. After each access to ORAM_i by user u_i, χ_i is incremented. Each user u_i keeps a local copy of χ_i and therewith verifies freshness of data presented by the server. As we will see below, this method ensures multi-user ORAM security. Note in Algorithm 1 that a user u_j never increases χ_i of another user u_i. Only u_i updates χ_i.

In our algorithms, Enc_κ is an IND-CPA encryption such

as AES-CBC. For convenience, we only write Enc_κ(main memory), although the main memory needs to be encrypted block by block to allow for the retrieval of specific blocks. Also, for the encryption of main memory blocks, Enc_κ offers authenticated encryption such as encrypt-then-MAC.

A user can determine whether a *cache is full* in Algorithm 2 by the convention that empty blocks in the cache decrypt to $\perp$. As long as there are blocks in the cache remaining with value $\perp$, the cache is not full.

Init: All ϕ ORAMs together with their cache and epoch counters are initialized. The server stores the ORAMs and MACs computed with a single key κ. Each user receives their state, comprising κ and cache counter.

Access: After verifying the MAC for ORAM_i and whether its cache is not from before u_i's last access, u_i performs a standard Read or Write operation for block x on ORAM_i. If the cache is full, u_i re-shuffles ORAM_i updating π. In addition, u_i also adds block x to all other users' ORAMs. Note that for this u_i does not read from the other ORAMs, but only completely downloads and re-encrypts their cache.

3.3 Security Analysis

First, we ensure with Lemma 1 that once a block $x_{i,j}$ enters the cache of ORAM_i, it can never be removed without user u_i noticing or the end of an epoch (and a new shuffle) occurring.

LEMMA 1. *Let $\Gamma_{i,j}$ be the state of the cache of ORAM_i when user u_i executes their j^{th} access. Let $R(\Gamma, x)$ be the predicate $x \in \Gamma$ which indicates if block x is already resident in the cache Γ. Let $x_{i,j}$ be the virtual block that user u_i accesses during operation j. $E(i,j)$ is the epoch that ORAM_i is in (as represented by the data returned from the adversary) when user u_i executes operation number j.*

If $(\Gamma_{i,j}, x_{i,j})$, *then*

$$\forall k > j \text{ with } E(i,j) = E(i,k) \wedge x_{i,j} = x_{i,k} : R(\Gamma_{i,j}, x_{i,k})$$

or user u_i executes Abort.

PROOF. This follows from the security of MAC and the fact that no user will remove a block from the cache unless they are performing a shuffle. If during an access j user u_i sees a counter value greater than or equal to the counter value from access $j-1$, and the MAC verifies, then they can be sure that every element in the cache during access $j-1$ is also in the cache during access j (unless there was a shuffle between). □

Now, Lemma 2 shows that if Lemma 1 holds, (Init, Access) is multi-user secure during a single epoch.

LEMMA 2.

If $\forall i, j, k$:

$$x_{i,j} \neq x_{i,k} \vee R(\Gamma_{i,j}, x_{i,j}) \vee R(\Gamma_{i,k}, x_{i,k}) \vee E(i,j) \neq E(i,k)$$

then (Init, Access) *is a multi-user ORAM secure against malicious adversaries.*

PROOF. The writing part of a Read or Write operation always reads and writes the same strings $\alpha \in \Sigma$, namely the cache and its authentication data (counters and *mac*). Therefore, with IND-CPA encryption, any pair of operations OP_0 and OP_1 will be indistinguishable for the writing part.

The read part on the other hand contains a conditional access to main memory. The goal is to show that this access does not leak any information that would allow an adversary to distinguish between two accesses. Our condition above ensures that there will not be two operations in the same epoch where the user requests a block and it is not in the cache. Since a block in main memory is only accessed if it does not already exist in the cache, this guarantees that each user i will never access the same block in main memory twice in the same epoch. Recall that every block is mapped to a random location, following a permutation π. If π is a random permutation, then the access pattern to main memory will be indistinguishable from random accesses, and the adversary's view will be indistinguishable for all pair of operations OP_0 and OP_1. □

THEOREM 1. (Init, Access) *is a multi-user Oblivious RAM secure against malicious adversaries.*

PROOF. In the same epoch, security follows from Lemma 2 and Lemma 1. Between epochs, main memory is re-shuffled and the ORAM is effectively reinitialized, with security of this new epoch being ensured as was the previous. □

Deamortizing. Goodrich et al. [3] propose a way to deamortize the classical square-root ORAM such that it obtains a worst-case overhead factor of $\sqrt{n} \cdot \log^2(n)$. Their method involves dividing the work of shuffling over the $\sqrt{n}$ operations during an epoch such that when the cache is full there is a newly shuffled main memory to swap in right away. This shuffling induces an access pattern in the RAM which is independent of the block a user is trying to access (it is performed along with the user request simply to spread the work out), and as such can also be incorporated into our scheme to achieve sublinear worst-case overhead.

3.4 Hierarchical Construction

In addition to the square-root ORAM, Goldreich and Ostrovsky [2] also propose a generalization which achieves poly-log overhead. In order to do this, it has a hierarchical series of caches instead of a single cache. Each cache has 2^j slots in it, for j from 1 to $\log n$, where each slot is a bucket holding $O(\log n)$ blocks. At the bottom of the hierarchy is the main memory which has $2 \cdot n$ blocks.

The reader is encouraged to refer to the original paper [2] for full details, but the main idea is that each level of the cache is structured as a hash table. Up to 2^{j-1} blocks can be stored in cache level j, half the space is reserved for dummies like

in the previous construction. After 2^{j-1} blocks, the entire level is retrieved and shuffled into the next level. Shuffling involves generating a new hash function and rehashing all the blocks into their new locations in level $j+1$, until the shuffling percolates all the way to the bottom and the user must shuffle main memory to start again. Level j must be shuffled after 2^{j-1} accesses, resulting in an amortized poly-logarithmic cost.

To access a block, a user queries the caches in order using the unique hash function at each level. When the block is found, the remainder of the queries will be on dummy blocks to hide that it was found. After reading, and potentially changing the value of the block, it is added back into the first level of the cache and the cache is shuffled as necessary.

Multi-user security. As this scheme is a generalization of the square-root one, our modifications extend naturally to provide multi-user security. Again, each user should have their own ORAM which they read from. Writing to other users' ORAMs is done by inserting the block into the top level of their cache and then shuffling as necessary. The only difference this time is that each level of the cache must be independently authenticated. Since the cache levels are now hash tables, and computing a MAC over them for each access would be prohibitively expensive, we can instead use a Merkle tree [8]. This allows for efficient verification and updating of pieces of the cache, and it maintains poly-log communication overhead.

4. TREE-BASED CONSTRUCTION

While pioneering the research, classical ORAMs have been outperformed by newer tree-based ORAMs which achieve better average and worst-case complexity. We now proceed to show how these constructions can be modified to also support multiple users. Our strategy will be similar to before, but with one major twist: in order to avoid linear worst case complexity, tree-based ORAMs do only small local "shuffling," which turns out to make separating a user access into critical and non-critical parts much more difficult. When writing, one must not only add a new version of the block to the ORAM, but also explicitly mark the old version as obsolete, requiring a conditional access. This is in contrast with our previous construction where old versions of a block would simply be discarded during the shuffle.

4.1 Overview

For this paper, we will use Path ORAM [11] as the basis for our multi-user scheme, but the concepts apply similarly to other tree-based schemes. To start, we briefly describe the original Path ORAM construction with emphasis on the features that will be important in our modified version. See [11] for complete details of the scheme.

Unlike classical ORAMs, instead of supporting Read and Write operations, Path ORAM supports ReadAndRemove, Add, and Evict. ReadAndRemove, as the name suggests, reads a block from the ORAM and removes it, while Add adds it back to the ORAM, potentially with a different value. These two operations can be used to emulate classical Read and Write operations, but it begins to illustrate the difficulty we have making this scheme multi-user secure: changing the value of a block implicitly requires reading it, meaning that both reading and writing are equally *critical* and not easily separated. The third operation, Evict, is a partial shuffling that is done after each access in order to maintain the integrity of the tree.

The RAM in Path ORAM is structured as a tree with n leaf nodes, each node in the tree holding up to Z blocks where Z is a small constant. Each block in the ORAM is tagged with a value uniform in the range $[0, n)$. As an invariant, blocks will always be located on the path from the root of the tree to the leaf node corresponding to their tag. Over the lifecycle of the tree, blocks will enter at the root and filter their way down toward the leaves, making room for new blocks to in turn enter at the root. The user has a map which stores, for every block, which leaf node it is tagged for.

ReadAndRemove: To retrieve block x, the user looks up in the map which leaf node it is tagged for and retrieves all nodes from the root to that leaf node, denoted $\mathcal{P}(x)$. By the tree invariant, block x will be found somewhere on the path $\mathcal{P}(x)$. The user then removes block x from the node it was found in, reencrypts all the nodes and puts them back in the RAM.

Add: To put a block back in the ORAM, the user simply retrieves the root node and inserts the block into one of its free slots, reencrypting and writing the node back afterwards. The map is updated with a new random tag for this block in the interval $[0, n)$. If there is not enough room in the root node, the user keeps the block locally in a "stash", waiting for a later opportunity to insert it into the tree.

Evict: So that the stash does not become too large, after every operation the user also performs an eviction which moves blocks down the tree to free up space. Eviction consists of picking a path in the tree (using reverse lexicographic ordering [**?**]) and moving all blocks on that path as far down the tree as they can go, without violating the invariant. Additionally, the user inserts any matching block from the stash into the path.

Typically, the user's map, which indicates for each block which path it will be found on, is too large to store locally. Fortunately, if the block size is at least $2 \cdot \log n$, the map can itself be stored recursively in another ORAM, and so on, inducing a total communication complexity of $O(\log^2 n)$. Additionally, Stefanov et al. [11] show that if $B = \Omega(\log^2 n)$, the complexity can be reduced to $O(\log n)$.

Integrity. Because of its tree structure, it is straightforward to ensure integrity in Path ORAM. The user can store a MAC in every node of the tree that is computed over the contents of that node and the respective MACs of its two children. Since the client accesses entire paths in the tree at once, verifying and updating the MAC values when an access is done incurs minimal overhead. This is a common strategy with tree-based ORAMs, which we will make integral use of in our scheme. We will also include user u_i's counter χ_u in the root MAC as before, to prevent rollback attacks (see below).

Challenge. Looking at Path ORAM, there exist several challenges when trying to add multi-user capabilities with our previous strategy. First, if we separate it into ϕ separate ORAMs (which we will do), we actually end up with a very large blowup because of the recursion. At the top level, we will have ϕ ORAMs, but each of those will have to have ϕ ORAMs in turn to support the map, each of which will have ϕ more, going down $\log n$ levels. The overall complexity would be $\phi^{\log n} \in \Omega(n)$. Additionally, as alluded to above, the fact that Add cannot be performed without ReadAndRemove means that we cannot easily split the access into *critical* and *non-critical* parts like before.

Rationale. To remedy these problems, we institute the following major changes to Path ORAM:

1. **Unified Tagging:** Instead of separately tagging every block in each of the ORAMs, we will have a unified tagging system where a block x has the "same" tag in each of the separate user ORAMs. This allows us to avoid a branching factor for the recursive map. For a block x, the map will resolve to a tag value t which describes its path in each one of the user ORAMs. Let h be a PRF mapping from $[0, 2^\lambda) \times [1, \phi]$ to $[0, n)$. The leaf that block x is percolating to differs for any ORAM. For ORAM_i of user u_i, it is pseudo-randomly determined by value $h(t, i)$.
2. **Secure Block Removal:** The central problem with ReadAndRemove is that it is required before every Add so that the tree will not fill up with old, obsolete blocks which cannot be removed. Unlike the square-root ORAM, the shuffling process (eviction) happens locally and cannot know about other versions of a block which exist on different paths. We solve this problem by including metadata on each bucket, as in Mayberry et al. [7]. For every node in the tree, we include an encrypted array which indicates the ID of every block in that node, or a special value $\perp$ for slots which are empty. Removing a block from the tree can now be performed by simply changing the metadata to indicate that the slot is empty. It will then be overwritten by the eviction routine with a real block if that slot is every needed. If B is large, this metadata is substantially smaller than the real blocks. We can then store it in a less efficient classical ORAM described above which is itself multi-user secure. This allows us to take advantage of the better complexity provided by tree-based ORAMs for the majority of the data, while falling back on a simpler ORAM for the metadata which is independent of B. We will show that, for modest block sizes, this leads to significantly improved performance compared to storing the data entirely in a classical ORAM.

We also note that Path ORAM's stash concept cannot be used in a multi-user setting. Since the users do not have a way of communicating with each other out of band, all shared state (which includes the stash) must be stored in the RAM. This has already been noted by Goodrich et al. [4], and since the size of the stash does not exceed $\log n$, storing it in the RAM (encrypted and integrity protected) does not effect the overall complexity.

Input: Security parameter λ, number of blocks in each ORAM n, block size B, number of users ϕ, sub-routine ORAM $(\mathsf{M-Init}, \mathsf{M-Access})$
Output: Initial ORAM state Σ_{init}, initial per user states $\{st_{u_1}, \ldots, st_{u_\phi}\}$

$\kappa \xleftarrow{\$} \{0,1\}^\lambda$;
for $j = 1$ *to* ϕ **do**
  $i = 0$;
  $n_0 = n$;
  while $n_i > 1$ **do**
    Initialize a tree $T_{j,i}$ with n_i leaf nodes ;
    Set eviction counter $e_{j,i} = 0$;
    `// The stash must also be stored on the server`
    Create array $S_{j,i}$ with Y blocks ;
    `// Use a sub-ORAM to hold block metadata`
    $M_{j,i} = \mathsf{M-Init}(\lambda, 2n_i \cdot Z, Z \cdot \log n_i, \phi)$;
    $n_{i+1} = n_i \cdot \lceil \log n_i / B \rceil$;
    $i = i + 1$;
  end
  Create a root block $\mathcal{R}_j$;
  Set ORAM counter $\chi_j = 0$;
  $\mathsf{ORAM}_j = \mathsf{Enc}_\kappa((T_{j,0}, M_{j,0}, S_{j,0}, e_{j,0}) || \ldots || (T_{j,m}, M_{j,m}, S_{j,m}, e_{j,m}) || \chi_j || \mathcal{R}_j)$;
  Send $st_{u_j} = \{\kappa, \chi_j, e_{j,0}, \ldots, e_{j_m}\}$ to user u_j ;
end
Send $\Sigma_{\text{init}} = \{\mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_\phi\}$ to server;

Algorithm 3: $\mathsf{Init}(\lambda, n, B, \phi)$ – Initialize ORAM

Similar to before, we also introduce an eviction counter e for each ORAM. User u_i will verify whether, for each of their recursive ORAMs, this eviction counter is fresh.

4.2 Details

Algorithm 3 initializes ϕ separate ORAMs and distributes the initial states (containing the shared key) to each user. These ORAMs $T_{j,i}$ each take the form of a series of trees. The first tree stores the data blocks, while the remaining trees recursively store the map which relates block addresses to leaf nodes. In addition to this, as described above, each tree has its own sub-ORAM to keep track of block metadata. The stash of each (sub-)ORAM is called $S_{0,i}$, and the metadata (classical) ORAM $M_{j,i}$.

For simplicity, we assume that Enc_κ encrypts each node of a tree separately, therewith allowing individual node access. Also, we assume authenticated encryption, using the per node integrity protection previously mentioned.

As noted above, the functions (ReadAndRemove, Add) can be used to implement (Read, Write), which in turn can implement a simple interface (Access). Because our construction introduces dependencies between ReadAndRemove and Add, in Algorithm 4 we illustrate a unified Access function for our scheme. The user starts with the root block and traverses the recursive map upwards to find the address of block x and finally retrieve it from the main tree. For each recursive tree, it retrieves a value t which allows it to locate the correct block in the next tree. After retrieving a block in each tree, the user marks that block as free in the metadata ORAM so that it can be overwritten during a future eviction. This is necessary to maintain the integrity of the

Input: Address x, user u_i, $st_{u_i} = \{\kappa, \chi_i\}$
Output: The value of block x
// Let m be the depth of recursion, n_j be the number of blocks in tree j
Retrieve root block $\mathcal{R}$;
// Find tag t_m where x is mapped to
$pos = x/n$; $x_m = \lfloor pos \cdot (B/\lambda) \rfloor$; $t_m = \mathcal{R}[x_m]$;
// Compute new tag t'_m for x
$t'_m \xleftarrow{\$} [0, 2^\lambda)$;
for $j = m$ *to* 0 **do**
  $\text{leaf}_j = h(t_j, u_i)$// Compute leaf of user u_i's ORAM_i;
  Read path $\mathcal{P}(\text{leaf}_j)$ and $S_{i,j}$ from $T_{i,j}$, locating block x_j;
  Retrieve MAC values for $\mathcal{P}(\text{leaf}_j)$ as V and the stored counter as χ'_i;
  if $V \neq \mathsf{MAC} - \mathsf{Path}(\Sigma, st_{u_i}, \mathcal{P}(\text{leaf}_j), S, \chi'_i) \vee \chi'_i \neq \chi_i$ **then**
    Abort;
  end
  Re-encrypt and write back $\mathcal{P}(\text{leaf}_j)$ and $S_{i,j}$ to $T_{i,j}$;
  // Let (a, b) be the node and slot that x_j was found at
  $\mathsf{M} - \mathsf{Access}(M_j, (\mathsf{write}, a \cdot Z + b, \perp), u_i)$;
  if $j \neq 0$ **then**
    $t'_j \xleftarrow{\$} [0, 2^\lambda)$ // Sample a new value for t;
    // Block x_j contains multiple t values
    Extract t_{j-1} from block x_j;
    Update block x_j with new value t'_{j-1} and new leaf tag t'_j;
  else
    Set v to the value of block x_j;
    If OP is a write, update x_j with new value;
  end
  Insert block x_j into the stash $S_{i,j}$;
  $\chi_i = \chi_i + 1$;
  Update MAC of stash to $\mathsf{MAC}_\kappa(S_{i,j} \,||\, \mathsf{MAC}$ of root bucket $||$ $\chi_i \,||\, e_{i,j})$;
  // Update the block in other user's ORAMs
  for $p \neq i$ **do**
    Retrieve path $\mathcal{P}(h(t_j, u_p))$ from $T_{p,j}$ and update metadata so block x_j is removed;
    Insert block x_j into the stash $S_{p,j}$ of $T_{p,j}$;
    Update MAC of root bucket in $T_{i,j}$;
  end
  output $(v, st_{u_i} = \{\kappa, \chi_i, e_{i,0}, \ldots, e_{i,m}\})$;
end

Algorithm 4: $\mathsf{Access}(\text{OP}, \Sigma, st_{u_i})$ – Perform Read or Write

tree and ensure that it does not overflow. At the same time, the user also marks that block free in the metadata of each other user and inserts the new block value into the root of their trees. This is analogous to the previous scheme where a user reads from their own ORAM and writes back to the ORAMs of the other users.

Algorithm 5 illustrates the eviction procedure. Since eviction does not take as input any user access, it is non-critical. The user simply downloads a path in the tree which is specified by eviction counter e and retrieves it in its entirety. The only modification that we make from the original Path ORAM scheme is that we read block metadata from the sub-ORAM that indicates which blocks in the path are free and can be overwritten by new blocks being pushed down the tree.

4.3 Security Analysis

We start the security analysis by showing that, due to the MACs authenticating each data structure, a specific user u_i

Input: Address x, new value v, user u_i, $st_{u_i} = \{k, \chi_i\}$
Output: The value of block x
for $j = 1$ *to* ϕ **do**
  for $r = 1$ *to* m **do**
    Retrieve eviction counter $e_{j,r}$ for $T_{j,r}$;
    Retrieve path $\mathcal{P}(e_{j,r})$, $S_{j,r}$ and MAC chain V;
    // Verify integrity of the path and eviction counter
    if $V \neq \mathsf{MAC} - \mathsf{Path}(\Sigma, st_{u_i}, \mathcal{P}(\text{leaf}_j), S_{j,r}, \chi'_i, e_{j,r})$ **then**
      Abort;
    end
    Read metadata for path from $M_{j,r}$;
    Move blocks out of the stash and down the path as far as possible;
    Reencrypt $\mathcal{P}(e_{j,r})$ and $S_{j,r}$ and write back to server;
    Update metadata for path $M_{j,r}$;
    $e_{j,r} = e_{j,r} + 1$;
  end
end

Algorithm 5: $\mathsf{Evict}(\Sigma, st_{u_i})$ – Perform Evict

Input: Σ, st_{u_i}, path $\mathcal{P}$, stash S, χ, eviction counter e
Output: Updated MAC values
for $j = \log n$ *to* 1 **do**
  $V[j] = \mathsf{MAC}_\kappa$(contents of bucket $\mathcal{P}[j]$ $||$ MAC of left child $||$ MAC of right child) ;
end
// Root MAC over the stash and tree parameters χ and e
$V[0] = \mathsf{MAC}_\kappa(S \,||\, \mathsf{MAC}$ of root bucket $||\, \chi \,||\, e)$;
return V

Algorithm 6: $\mathsf{MAC} - \mathsf{Path}(\Sigma,\ st_{u_i},$ path $\mathcal{P}$, stash S, χ, eviction counter $e)$

will read the same tag t from a tree in their ORAM with probability negligible in λ.

LEMMA 3. *For any two accesses to a map tree $T_{i,j}$, $1 \leq j \leq m$, by arbitrary but fixed user u_i, which do not result in u_i aborting, the probability that they both return the same value t is negligible in λ.*

PROOF. We start with the root block. User u_i replaces each value with a fresh t_0 in the range $[0, 2^\lambda)$ after each access. So, if the server is honest, u_i will read the same value in two separate accesses only with probability $2^{-\lambda}$. For the case of a malicious server, u_i also keeps a counter χ_i which is incremented after every access. The root block on the server additionally stores this counter along with a MAC that authenticates the block-counter combination. As long as the MAC is unforgeable with chance $1 - 2^{-\lambda}$, the probability that u_i does not abort on a bad block-counter combination is negligible.

After the root block, we continue with the map trees. The user will read a path in each tree which contains the target block, and next value t_j. If the server is honest, u_i would have changed t_j since the last time it was accessed and the probability would again be $2^{-\lambda}$. User u_i also has a MAC chain here tied to a counter which can be verified, so against a malicious adversary the probability is still negligible in λ. □

With that lemma, we can prove that our construction is secure based on the fact that the t values induce a uniform distribution of blocks across the leaf nodes and that no user will have a collision in their t values with any non-negligible probability.

THEOREM 2. *Our tree-based construction* (Init, Access) *is a multi-user Oblivious RAM secure against malicious adversaries.*

PROOF. If h is a PRF, then assigning leaf nodes to blocks as $h(t, i)$ for user u_i will result in a (pseudo-)random distribution over the leaf nodes for every block in every tree. By Lemma 3, even against a malicious adversary, with all by negligible probability no user will make two accesses that return the same value t_i. By induction, this means that the paths read in each tree when a user accesses their own ORAM will be distributed pseudorandomly, independent of the virtual block being accessed. Thus, a user reading from their own ORAM cannot leak any information that would allow an adversary to distinguish between two access patterns.

When users write to other users' ORAMs, they directly and deterministically access the stash. The users additionally read an write with the sub-ORAM, which is in itself multi-user secure. Since they always execute the same number of accesses ($\log n$ per tree) on this ORAM, and the number of accesses is the only thing leaked to the adversary with a secure ORAM, this part cannot give advantage to a distinguisher.

The last algorithm is eviction. Since the path chosen during eviction is deterministic (based on the counter) and independent of any accesses done by any user, it is straightforward to see that it also will induce a pattern on the server which is indistinguishable. □

4.4 Complexity

The complexity of our scheme is dominated by the cost of an eviction. For a user to read a path in each of $O(\log n)$ recursive trees, for each of the ϕ different ORAMs, it takes $O(\phi \cdot B \cdot \log^2 n)$ communication. Additionally, the user must make $O(\phi \cdot \log^2 n)$ accesses to a metadata ORAM. If $\mu(n, B)$ denotes the cost of a single access in such a sub-ORAM, the overall complexity is then $O(\phi \cdot \log^2 n \cdot [B + \mu(n, \log n)])$. Taking the hierarchical ORAM as a sub-ORAM, the total worst-case communication complexity computes to $O(\phi \cdot \log^2 n[B + \log^4 n])$. If $B \in \Omega(\log^4 n)$ then the communication complexity, in terms of blocks, is $O(\log^2 n)$, otherwise it is at most $O(\log^5 n)$, i.e., with the assumption $B \in \Omega(\log n)$ (minimal possible block size). Although a complexity linear in ϕ may seem at first to be expensive, we stress that we are the first to achieve Oblivious RAM against malicious adversaries.

One notable difference in parameters from basic Path ORAM is that we require a block size of at least $c \cdot \lambda$, where $c \geq 2$. Path ORAM only needs $c \cdot \log n$, and for security parameter λ, $\lambda > \log n$ holds. In our scheme, the map trees do not directly hold addresses, but t values which are of size λ. In order for the map recursion to terminate in $O(\log n)$ steps, blocks must be big enough to hold at least two t values of size λ. If the block size is $\Omega(\lambda^2)$, we can also take advantage of the asymmetric block optimization from Stefanov et al. [11] to reduce the complexity to $O(\phi \cdot (\log^6 n + B \cdot \log n)$. Then, if additionally $B \in \Omega(\log^5 n)$, the total complexity is reduced to $O(\log n)$ per user.

4.5 Conclusion and Future Work

We have presented the first techniques that allow multi-user ORAM, secure even in the face of fully malicious servers. Our multi-user ORAMs are reasonably efficient with complexities between $O(\log n)$ to $O(\log^5 n)$ per user, depending on the underlying block size. Future work will focus on efficiency improvements, e.g., reducing worst-case complexity to being sublinear in ϕ. Additionally, the question of whether tree-based constructions are more efficient than classical ones is not as clear in the multi-user setting as it is for a single user. Although tree ORAMs are more efficient for a number of parameter choices, they incur substantial overhead from using a sub-ORAM to hold tree metadata. This is not required for the classical constructions. Future research may focus on achieving a "pure" tree-based construction which does not depend on another ORAM. Finally, it may be interesting to investigate whether multiple users can be supported with a more fine-grained access control. Instead of every user have full permissions to the ORAM, can they have separate keys and somehow share only pieces of their individual databases.

References

[1] O. Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In *Proceedings of Symposium on Theory of Computing*, pages 182–194, New York, USA, 1987. ISBN 0-89791-221-7.

[2] O. Goldreich and R. Ostrovsky. Software Protection and Simulation on Oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. ISSN 0004-5411.

[3] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Oblivious RAM simulation with efficient worst-case access overhead. In *Proceedings of Workshop on Cloud Computing Security Workshop*, pages 95–100, Chicago, USA, 2011.

[4] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In *Proceedings of Symposium on Discrete Algorithms*, pages 157–167, Kyoto, Japan, 2012. ISBN 978-1-61197-210-8.

[5] C. Hall, I. Goldberg, and B. Schneier. Reaction attacks against several public-key cryptosystems. In *Proceedings of International Conference on Information and Communication Security*, pages 2–12, Sydney, Australia, 1999. ISBN 3-540-66682-6.

[6] J. Li, Maxwell N. Krohn, D. Mazières, and D. Shasha. Secure Untrusted Data Repository (SUNDR). In *Proceedings of Operating System Design and Implementation*, pages 121–136, San Francisco, USA, 2004.

[7] T. Mayberry, E.-O. Blass, and A.H. Chan. Efficient private file retrieval by combining ORAM and PIR. In

Proceedings ofNetwork & Distributed System Security Symposium, pages 1–11, San Diego, USA, 2014.

[8] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Advances in CryptologyÑCRYPTOÕ87*, pages 369–378. Springer, 1988.

[9] L. Ren, C.W. Fletcher, X. Yu, M. v. Dijk, and S. Devadas. Integrity Verification for Path Oblivious-RAM. In *Proceedings of High Performance Extreme Computing Conference*, pages 1–6, Waltham, USA, 2013.

[10] E. Shi, T.-H.H. Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.

[11] E. Stefanov, M. v. Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas. Path ORAM: An extremely simple oblivious ram protocol. In *Proceedings of Conference on Computer & Communications Security*, pages 299–310, Berlin, Germany, 2013. ISBN 978-1-4503-2477-9.