Multi-Client Oblivious RAM secure against Malicious Servers

Erik-Oliver Blass[1], Travis Mayberry[2], and Guevara Noubir[3]

[1] Airbus Group Innovation, Munich, Germany
erik-oliver.blass@airbus.com
[2] US Naval Academy, Annapolis MD,
mayberry@usna.edu
[3] Northeastern University, Boston MA,
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 clients. The challenge is that ORAMs are stateful, and clients would need to exchange updated state to maintain security. However, clients often do not have a way to directly communicate, and a malicious server can tamper with state information and thus break security. We answer the question of multi-client ORAM on malicious servers affirmatively by providing several new, efficient multi-client ORAM constructions. We first extend the classical square-root ORAM by Goldreich and the hierarchical one by Goldreich and Ostrovsky to become multi-client secure. We accomplish this by separating the *critical* parts of the access, which depend on the state of the ORAM, from the non-critical parts (cache access) that can be executed securely in any state. Our second and main contribution is a secure multi-client variant of Path ORAM. To enable secure meta-data update during evictions in Path ORAM, we employ our first result, small multi-client secure classical ORAMs, as a building block. Depending on the block size, the overhead of our multi-client secure construction reaches a low $O(\log n)$ communication complexity per client, similar to state-of-the-art single-client ORAMs.

1 Introduction

Practicality of Oblivious RAM has improved significantly in the last few years. The main metric of an ORAM's performance, communication overhead, has decreased by orders of magnitude. However, at least one significant hurdle to overcome for actual adoption remains: security of modern ORAMs relies on there being only a single client at all times. The problem stems from the fact that, in order to hide the access pattern, today's ORAMs must modify some of the data on the server after every access. If a malicious server "rewinds" the data and presents an old version to a client, further interactions may reveal details about the access pattern. In single client scenarios, this is typically solved by storing a small token on the client, such as the root of a hash tree [7]. This token authenticates and verifies freshness of all data retrieved from the server, ensuring that no such rewind attack is possible.

In this paper, we address some more complex scenarios where multiple clients share data stored in a single ORAM. With multiple clients, an authentication token is not

sufficient. Data may not pass one client's authentication, simply as it has been modified by one of the other clients. If clients 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 clients 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-client ORAM work only in the presence of an honest-but-curious adversary, which cannot perform rewind attacks on the clients. 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-client ORAM, recently proposed as an open question whether one could be secure for multiple clients against a malicious server.

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

- We start by focusing on two ORAM constructions that follow a "classical" approach, the *square-root* ORAM by Goldreich [2] and the *hierarchical* ORAM by Goldreich and Ostrovsky [1]. We adapt these ORAMs for multi-client security. Our approach is to separate client 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 integrity check which will reveal any malicious behavior, thereby allowing the client to terminate the protocol. Roughly speaking, we replicate the ORAM for each client, such that clients read only from their copy of the ORAM (integrity protected), but write into all ORAM copies (trivially secure).
- 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. [8] and Stefanov et al. [9] (Path ORAM) and many derivatives, provide better efficiency and worst-case guarantees. Consequently, we go on to demonstrate how a multi-client secure Path ORAM can be constructed. Among changing stash behavior and address mapping, we solve the key challenge of realizing a multi-client secure ReadAndRemove by storing Path ORAM's metadata using small "classical" ORAMs as building blocks. For reasonably small block sizes, this results in a multi-client ORAM which has the same per-client communication complexity as current single-client ORAMs, i.e., $O(\log n)$.

2 Multi-client ORAM

Instead of a single ORAM accessed by a single client, we can envision multiple clients securely exchanging or sharing data stored in a single ORAM. For example, imagine multiple employees of a company that want to read from and write into the same database stored at an untrusted server. Similar to standard ORAM security, sharing data and jointly working on the database should not leak the employees' access patterns

to the server. Alternatively, we can also envision a single person with multiple different devices (laptop, tablet, smartphone) accessing the same data hosted at an untrusted server. Again, working on the same data should not reveal access patterns. Throughout this paper we consider the terms "multi-client" and "multi-user" to be equivalent. As suggested by Goodrich et al. [4], we assume that clients all trust each other and leave expansion of our results for more fine-grained security as future work.

ORAM protocols provide security, because they are highly stateful. In order to hide the fact that a client 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 client when they make the second access. If the client 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 ORAM scheme. This is a straightforward attack. In the case of a single client, it is easily defeated by having the client store a token for authentication and freshness [7].

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

This paper tackles scenarios with fully malicious servers. Note that against an honest-but-curious adversary, multi-client security is trivial. The adversary is guaranteed not to change any data on the server, and so any ORAM protocol with small client memory can be used with multiple clients. The clients simply encrypt and upload all of their local memory (state) to the server after every access, and the next client downloads this memory to continue the protocol where the first client left off. In the face of a malicious server, this approach is not feasible, because the adversary can tamper with the uploaded client memory as described above. Therefore, the malicious adversary is crucial to the motivation for our work.

2.1 Security Definition

We briefly recall standard ORAM concepts. 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* here because the most likely application for a multi-client 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-client ORAM security definition which slightly augments the standard, single-client ORAM definition.

Definition 2 (Multi-client ORAM Π). *A multi-client ORAM $\Pi = (\mathsf{Init}, \mathsf{Access})$ comprises the following two algorithms.*

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 clients 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 client states $\{st_{u_1}, \ldots, st_{u_\psi}\}$ which are kept local to the individual clients.*
2. $\mathsf{Access}(\mathrm{OP}, \Sigma, st_{u_i})$ *performs operation OP on ORAM state Σ using client 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 client u_i.*

In contrast to single-client ORAM, a multi-client ORAM introduces the notion of clients. This is modeled by different per-client states, st_{u_i} for client u_i. Algorithm Init represents a "dealer" who outputs different initial states st_{u_i} after which, in practice, they would be distributed to the individual clients. Whenever client u_i executes Access on the multi-client ORAM, they can only update their own state st_{u_i}.

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

Definition 3 (Multi-client ORAM $\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda)$).

$b \xleftarrow{\$} \{0,1\}$
$(\Sigma_{\mathrm{init}}, st_{u_1}, \ldots, st_{u_\psi}) \leftarrow \mathsf{Init}(\lambda, n, B, \psi)$
$(\Sigma, \mathrm{OP}_0, \mathrm{OP}_1, i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(\lambda, n, B, \Sigma_{\mathrm{init}}, \psi)$
for $j = 1$ **to** *poly*(λ) **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-client 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 client u_i, determines two operations OP_0 and OP_1, and outputs an ORAM state Σ. The oracle performs operation OP_b as client 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 ORAM's standard 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 among themselves.

As discussed above, this work assumes that all clients trust each other and do not conspire. For ease of exposition, we assume that all client share a key κ used for encryptions, decryptions, and MAC computations that we will introduce later.

Consistency: An orthogonal concern to security for multi-client schemes is *consistency*, whether the clients each see the same version of the database when they access it. Because the clients in our model do not have any way of communicating except through the malicious adversary, it is possible for $\mathcal{A}$ to "desynchronize" the clients so that their updates are not propagated to each other. Our multi-client ORAM guarantees that in this case the clients still have complete security and access pattern privacy, but consistency cannot be guaranteed. This is a well known problem with the best solution being *fork* consistency [5], which we can achieve.

3 Multi-client Security for Classical ORAMs

We start by transforming two classical ORAM constructions, the original square-root solution by Goldreich [2] and the hierarchical one by Goldreich and Ostrovsky [1], into multi-client secure versions, retaining the same communication complexity per client.

Challenge When considering a multi-client scenario, it becomes easy for a malicious server to break security of the square-root ORAM. For example, client 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 client u_2 also attempts to access block x, the server will now observe that both clients read from the same location in main memory and know that u_1 and u_2 have accessed the same block (or not). Without the clients 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 client to leak information about their accesses.

Rationale Our approach for multi-client 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 storing it back with the server. Downloading and later uploading the cache implies

Input: Security parameter λ, number of blocks in each ORAM n, block size B, number of clients ϕ
Output: Initial ORAM state Σ_{init}, initial per client 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 client 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 multi-client square-root ORAM

always "touching" the same $\sqrt{n}$ blocks, independently of what the malicious server presents to a client as Σ, and also independent of the block being updated by the client. 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-client secure.

However, reading can leak information. Reading from the main ORAM is conditional on what the client 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 multiple clients for the square-root ORAM:

1. **Separate ORAMs:** Instead of a single ORAM, we use a sequence of ORAMs,

$$\mathsf{ORAM}_1, \mathsf{ORAM}_2, \ldots, \mathsf{ORAM}_\phi,$$

   one for each client. Client u_i will perform the critical part of their access only on ORAM_i's main memory and cache. Thus, each client 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-client secure and does not leak information.
2. **Authenticated Caches:** For each client 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 client 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 clients 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.

Input: Address x, new value v, client 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 output** Abort;
Decrypt c_i to get cache and counter χ'_i;
if $\chi'_i < \chi_i$ **then output** Abort;
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;
if $v = \perp$ **then** `// operation is a` Read
    $\nu \leftarrow v$;
else `// operation is a` Write
    $\nu \leftarrow$ existing value of block x;
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;
$\chi_i = \chi'_i + 1$; $mac_i = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i))$;
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;
    $mac_j = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache}||\chi_j||\gamma_j))$;
    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: $\mathsf{Access}(\mathrm{OP}, \Sigma, st_{u_i})$, Read, Write for multi-client square-root ORAM

3.1 Details

We detail the above ideas in two algorithms: Algorithm 1 shows the initialization procedure, and Algorithm 2 describes the way a client performs an access with our multi-client 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 client u_i in ORAM_i's epoch γ_i. For any client, 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 generate pseudo-random permutations π_{i,γ_i} on n elements in a deterministic fashion. For example, one can use $\mathrm{PRF}_\kappa(i||\gamma_i)$ as the seed in a PRG and therewith perform a Fisher-Yates shuffle.

In addition to the epoch counter, we also introduce a per client *cache counter* χ_i. Using χ_i, client 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 client u_i, χ_i is incremented. Each client 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-client ORAM security. Note in Algorithm 1 that a client u_j never increases χ_i of another client u_i. Only u_i ever updates χ_i.

In our algorithms, Enc_κ is an IND-CPA encryption such as AES-CBC. For convenience, we only write $\mathsf{Enc}_\kappa(\text{main memory})$, 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 client 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 client 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 clients' ORAMs. Note that for this, u_i does not read from the other ORAMs, but only completely downloads and re-encrypts their cache.

Note on complexity In addition to the communication complexity involved, there is also computation the client must perform in our scheme. Fortunately, the computation is exactly proportional to the communication and easily quantifiable. Every block of data retrieved from the server has a MAC that must be verified and a layer of encryption that must be removed. Since modern ciphers and hash functions are very efficient, and can even be done in hardware on many computers, communication is the clear bottleneck. For comparison, encryption and MACs are common on almost every secure network protocol, so we consider only the communication overhead in our analysis.

3.2 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 client 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 client 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 client 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 client u_i executes operation number j.

For all PPT adversaries $\mathcal{A}$ and security parameter λ, there exists a negligible function ϵ such that

$$\begin{aligned}&Pr\,[\textit{If } R(\Gamma_{i,j}, x_{i,j}), \textit{ then}\\ &\qquad (\forall k > j \textit{ with } E(i,j) = E(i,k) \wedge x_{i,j} = x_{i,k}) :\\ &\qquad R(\Gamma_{i,j}, x_{i,k}) \textit{ or client } u_i \textit{ outputs } \mathsf{Abort}] = 1 - \epsilon(\lambda)\end{aligned}$$

Proof. This follows from the security of MAC and the fact that no client will remove a block from the cache unless they are performing a shuffle. If during an access j client 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, $(\mathsf{Init}, \mathsf{Access})$ is multi-client secure during a single epoch.

Lemma 2. *For all PPT adversaries $\mathcal{A}$ and security parameter λ, there exists a negligible function ϵ such that*

$$\begin{aligned}&\textit{If } Pr\,[\forall i, j, k :\\ &\qquad x_{i,j} \neq x_{i,k} \vee R(\Gamma_{i,j}, x_{i,j}) \vee R(\Gamma_{i,k}, x_{i,k})\\ &\qquad\qquad \vee E(i,j) \neq E(i,k)] = 1 - \epsilon(\lambda)\end{aligned}$$

then $(\mathsf{Init}, \mathsf{Access})$ is a multi-client 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 client 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 client 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. *For all PPT adversaries $\mathcal{A}$ and security parameter λ, there exists a negligible function ϵ such that* $(\mathsf{Init}, \mathsf{Access})$ *is a multi-client Oblivious RAM secure against $\mathcal{A}$ with probability* $1 - \epsilon(\lambda)$.

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 client is trying to access (it is performed along with the client request simply to spread the work out), and as such can also be incorporated into our scheme to achieve sublinear worst-case overhead.

Multi-client security As this scheme is a generalization of the square-root one, our modifications extend naturally to provide multi-client security. Again, each client should have their own ORAM which they read from. Writing to other clients' 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 every level for each access would require downloading the whole data structure, we can instead use a Merkle tree [6]. This allows for efficient verification and updating of pieces of the cache without having access to the entire thing, 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 clients. 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 client 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 section, we will use Path ORAM [9] as the basis for our multi-client scheme, but the concepts apply similarly to other tree-based schemes.

Integrity Because of its tree structure, it is straightforward to ensure integrity in Path ORAM. The client 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 client 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-client 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, 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 client 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 client 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 every ORAM and 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. For every node in the tree, we include an encrypted array which indicates the ID of every block in that node. Removing a block from the tree can then be performed by simply changing the metadata to indicate that the slot is empty. It will be overwritten by the eviction routine with a real block if that slot is ever 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-client 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 also note that Path ORAM's stash concept cannot be used in a multi-client setting. Since the clients 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 affect the overall complexity.

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

4.2 Details

See Appendix A for full algorithms, Init, Access and Evict. To initialize the multi-client ORAM, ϕ separate ORAMs are created and the initial states (containing the shared key) are distributed to each client. 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 $M_{j,i}$.

To avoid confusion between different ORAM initialization functions, MInit is a reference to Algorithm 1, i.e., initialization of a multi-client secure classical ORAM.

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 $(\mathsf{ReadAndRemove}, \mathsf{Add})$ can be used to implement $(\mathsf{Read}, \mathsf{Write})$, 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 client starts with the root block and traverses the recursive map upwards to find the address of block x and finally retrieves 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 client 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 tree and ensure that it does not overflow. At the same time, the client also marks that block free in the metadata of each other client and inserts the new block value into the root of their trees. This is analogous to the previous scheme where a client reads from their own ORAM and writes back to the ORAMs of the other clients.

Again, we avoid confusion between different ORAM access operations by referring to the multi-client secure classical ORAM access operation of Algorithm 2 as MAccess.

Algorithm 5 illustrates the eviction procedure. Since eviction does not take as input any client access, it is non-critical. The client 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 client u_i will read the same tag t from a tree in their ORAM with probability negligible in λ.

Lemma 3. *Let u_i be a client i. For any two accesses to a map tree $T_{i,j}$, $1 \leq j \leq m$, by client 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. Client 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 client 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 $2^{-\lambda}$. client 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 client will have a collision in their t values with any non-negligible probability.

Theorem 2. *Our tree-based construction* $(\mathsf{Init}, \mathsf{Access})$ *is a multi-client Oblivious RAM secure against malicious adversaries.*

Proof. If h is a PRF, then assigning leaf nodes to blocks as $h(t, i)$ for client 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 but negligible probability no client will make two accesses that return the same value t_i. By induction, this means that the paths read in each tree when a client accesses their own ORAM will be distributed pseudorandomly, independent of the virtual block being accessed. Thus, a client reading from their own ORAM cannot leak any information that would allow an adversary to distinguish between two access patterns.

When clients write to other clients' ORAMs, they directly and deterministically access the stash. The clients additionally read and write with the sub-ORAM, which is in itself multi-client 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 information cannot give an advantage to the adversary in distinguishing access patterns.

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 client, 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 client 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 client 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 this is a substantial improvement over naive solutions which achieve the same level of security. The only straightforward way to have multi-client security against malicious servers is for each client to append their updates to a master list, and for clients to scan this list to find the most updated version of a block during reads. This is not only linear in the size of the database, but in the number of operations performed over the entire life of the ORAM.

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. [9] 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 client.

4.5 Conclusion

We have presented the first techniques that allow multi-client ORAM, secure even in the face of fully malicious servers. Our multi-client ORAMs are reasonably efficient with complexities between $O(\log N)$ to $O(\log^5 N)$ per client, 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-client setting as it is for a single client. 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 clients can be supported with a more fine-grained access control. For example, instead of every client have full permissions to the ORAM, can they have separate keys and somehow share only pieces of their individual databases.

References

[1] O. Goldreich and R. Ostrovsky. "Software protection and simulation on oblivious RAMs". In: *Journal of the ACM* 43.3 (1996). ISSN 0004-5411, pp. 431–473.

[2] Oded Goldreich. “Towards a Theory of Software Protection and Simulation by Oblivious RAMs”. In: *Proceedings of Symposium on Theory of Computing*. New York, USA, 1987, pp. 182–194.

[3] M.T. Goodrich et al. “Oblivious RAM simulation with efficient worst-case access overhead”. In: *Proceedings of Workshop on Cloud Computing Security Workshop*. Chicago, USA, 2011, pp. 95–100.

[4] M.T. Goodrich et al. “Privacy-preserving group data access via stateless oblivious RAM simulation”. In: *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*. 2012, pp. 157–167.

[5] J. Li et al. “Secure Untrusted Data Repository (SUNDR)”. In: *Proceedings of Operating System Design and Implementation*. San Francisco, USA, 2004, pp. 121–136.

[6] Ralph C Merkle. “A digital signature based on a conventional encryption function”. In: *Advances in Cryptology – CRYPTO*. Springer. 1988, pp. 369–378.

[7] L. Ren et al. “Integrity Verification for Path Oblivious-RAM”. In: *Proceedings of High Performance Extreme Computing Conference*. Waltham, USA, 2013, pp. 1–6.

[8] E. Shi et al. “Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost”. In: *Proceedings of Advances in Cryptology – ASIACRYPT*. Vol. 7073. Seoul, South Korea, 2011, pp. 197–214. ISBN: 978-3-642-25384-3.

[9] E. Stefanov et al. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: *Proceedings of Conference on Computer & Communications Security*. ISBN 978-1-4503-2477-9. Berlin, Germany, 2013, pp. 299–310.

A Tree-based Algorithms

Input: Security parameter λ, number of blocks in each ORAM N, block size B, number of clients ϕ, initialization sub-routine for multi-client secure classical ORAM MInit

Output: Initial ORAM state Σ_{init}, initial per client 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`
        $M_{j,i} = \mathsf{MInit}(\lambda, 2n_i \cdot Z, Z \cdot \log n_i, \phi)$;
        $N_{i+1} = N_i \cdot \lceil \log N_i / B \rceil$;
        $i = i + 1$;
    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 client 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 multi-client tree-based ORAM

Input: Address x, client u_i, $st_{u_i} = \{\kappa, \chi_i\}$, access sub-routine for multi-client secure classical ORAM $\mathsf{MAccess}$

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 client 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 $V \neq \mathsf{MACPath}(\Sigma, st_{u_i}, \mathcal{P}(\text{leaf}_j), S, \chi'_i) \vee \chi'_i \neq \chi_i$ **then** Abort ;

  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

  $\mathsf{MAccess}(M_j, (\mathsf{write}, a \cdot Z + b, \perp), u_i)$;

  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;

  Insert block x_j into the stash $S_{i,j}$;

  $\chi_i = \chi_i + 1$;

  Update MAC of stash to $\mathsf{MAC}_\kappa(S_{i,j}$, MAC of root bucket, χ_i, $e_{i,j})$;

  // Update the block in other client'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: $\mathsf{Access}(\text{OP}, \Sigma, st_{u_i})$, Read or Write on multi-client tree-based ORAM

Input: Address x, new value v, client 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 $V \neq \mathsf{MACPath}(\Sigma, st_{u_i}, \mathcal{P}(\text{leaf}_j), S_{j,r}, \chi_i', e_{j,r})$ **then** Abort ;
    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: Evict(Σ, st_{u_i}) – Perform Evict on multi-client tree-based ORAM

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
$V[0] = \mathsf{MAC}_\kappa(S$, MAC of root bucket, $\chi, e)$;
return V

Algorithm 6: MACPath(Σ, st_{u_i}, path $\mathcal{P}$, stash S, χ, eviction counter e)