Multi-Client Oblivious RAM Secure Against Malicious Servers

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

[1] Airbus Group Innovations, Munich, Germany
erik-oliver.blass@airbus.com
[2] US Naval Academy, Annapolis MD, USA
mayberry@usna.edu
[3] Northeastern University, Boston MA, USA
noubir@ccs.neu.edu

Abstract. This paper tackles the open problem whether an Oblivious RAM can be shared among multiple clients in the presence of a fully malicious server. Current ORAM constructions rely on clients knowing the ORAM state to not reveal information about their access patter. With multiple clients, a straightforward approach requires clients exchanging updated state to maintain security. However, clients on the internet usually cannot directly communicate with each other due to NAT and firewall settings. Storing state on the server is the only option, but a malicious server can arbitrarily tamper with that information.
We first extend the classical square-root ORAM by Goldreich and the hierarchical one by Goldreich and Ostrovsky to add muti-client security. We accomplish this by separating the *critical* portions 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 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 communication complexity 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

The main metric of an ORAM's performance, communication overhead, has improved by orders of magnitude over the last few years. However, at least one significant hurdle to actual adoption remains: security of modern ORAMs relies on there being only a single client at all times. While there already exist multi-client ORAMs secure in the face of a *semi-honest* server [17], existence of an ORAM secure against a fully *malicious* server is still an open question. The main challenge here stems from the fact that today's ORAMs 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 [21]. This token authenticates and verifies freshness of all data retrieved from the server, ensuring that no such rewind attack is possible.

Table 1: Communication and storage worst-case complexity for existing single-client ORAMs and our new multi-client versions. ϕ is the number of different clients supported by the ORAM. $\hat{O}$ denotes amortized complexity.

	Communication		Storage	
	Single Client	**Multi Client**	Single Client	**Multi Client**
Square-Root [7]	$\hat{O}(\sqrt{N})$	$\hat{O}(\phi \cdot \sqrt{N})$	$O(N)$	$O(\phi \cdot N)$
(Deamortized) Hierarchical [8, 14]	$O(\log^3 N)$	$O(\phi \cdot \log^3 N)$	$O(N \cdot \log N)$	$O(\phi \cdot N \cdot \log N)$
Tree-based [24]	$O(\log N)$	$O(\phi \cdot \log N)$	$O(N \cdot \log N)$	$O(\phi \cdot N \cdot \log N)$

In this paper, we address the fundamental problem how multiple clients can 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 because 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 trivial to continually exchange and update each other with the most recent token. However, existence of secure out-of-band communication is often not a reasonable assumption for modern devices. As we will see, it is the absence of out-of-band-communication which makes multi-client ORAM technically challenging.

Current solutions for multi-client ORAM work only in the presence of semi-honest (honest-but-curious) adversaries, 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. Consequently, we only address fully malicious servers. Goodrich et al. [11], 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 We introduce the first construction for a multi-client ORAM and prove access pattern indistinguishability, 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 [7] and the *hierarchical* ORAM by Goldreich and Ostrovsky [8]. We adapt these ORAMs for multi-client security. Our approach is to separate client accesses into two parts: non-critical portions, which can be performed securely in the presence of a malicious server and critical portions, which cannot but contains efficient integrity checks which will reveal any malicious behavior and allow the client to terminate the protocol.
– The "classical" ORAM constructions have been largely overshadowed by more recent tree-based ORAMs [22, 24]. Consequently, we go on to demonstrate how a multi-client secure Path ORAM [24] can be constructed. We solve the key challenge of realizing a multi-client secure version of the *read* protocol by storing Path ORAM's metadata using small "classical" ORAMs as building blocks. For block sizes in $\Omega(\log^4 N)$, this results in a multi-client ORAM which has overall communication complexity of $O(\phi \cdot \log N)$.

Table 1 summarizes asymptotic behavior for our new multi-client ORAMs and compares them to their corresponding single-client ORAMs.

2 Motivation: Multi-Client ORAM

Instead of a single ORAM accessed by a single client, we envision multiple clients securely exchanging or sharing data stored in a single ORAM. For example, imagine multiple employees of a company that 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 (e.g., Dropbox). 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. [11], we assume that clients all trust each other and leave expansion of our results for more fine-grained access control as future work. In the multi-device scenarios above, it is reasonable that clients trust each other since they belong to a single user.

To provide security, ORAM protocols are *stateful*. Hiding client accesses to a certain data block is typically achieved by performing shuffling or reordering of blocks, such 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 (state) 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 that can be easily defeated in case of a single client: as an internal state, the client stores and updates a token for authentication and freshness, see Ren et al. [21].

However, with two or more clients sharing data in an ORAM, this attack becomes a new challenge. After watching one client retrieve some data, the adversary rewinds the ORAM's state and present the original view to the second client. If the second client accesses the same data (or not) 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.

2.1 Technical Challenges

A multi-client ORAM has to overcome a new technical challenge. Roughly speaking, the server is fully malicious and can present different ORAM states to different clients, i.e., different devices of the same user. As different clients do not have a direct communication channel to synchronize *their* state, it is difficult for them to synchronize on an ORAM state. We expand on this challenge below.

Adversary model: This paper tackles the scenario of ϕ trusted clients sharing storage on a fully malicious server (the adversary). Other works such as Maffei et al. [17] have addressed the problem against a semi-honest server, but in many scenarios that may not be sufficient. Real-world attacks that clients need to defend against include, e.g., insider

attacks from a cloud provider hosting the server and outside hackers compromising the server. Such attacks would allow for malicious adversarial behavior. In general, there is no strong line between the two adversarial models that suggests that one is more reasonable to defend against. To cope with all possible adversaries, it is therefore important to protect against malicious adversaries, too.

No out-of-band communication: We assume that beyond a single cryptographic key (possibly derived from a password) the clients do not share any long-term secrets and cannot communicate with each other except through the malicious server. This matches with existing cloud settings, since most consumer devices are behind NAT and cannot be directly contacted from the Internet. Major real-world cryptographic applications, for instance WhatsApp [25] and Semaphor [23], have all messages between clients relayed through the server for this reason.

We emphasize that in the malicious setting the server always has the option to simply stop responding or send purposefully wrong data as a denial-of-service attack. This cannot be avoided, but is also not a significant problem since it will be easily detected by clients. In contrast, the attacks we focus on in this paper are those where the server tries to compromise security without being detected.

2.2 Other Applications

A major application of this work is in supporting private cloud storage that is accessible from multiple devices. In reality, this is one of the most compelling use cases for cloud storage a la Dropbox, Google Drive, iCloud, etc. and is a good target for a privacy-preserving solution. Beyond simple storage, Oblivious RAM is used as a subroutine in other constructions, such as dynamic proofs of retrievability [2]. Any construction which uses ORAM and wishes to support multiple clients must rely on an ORAM that is secure for multiple clients.

Finally, an interesting application comes from the release of Intel's new SGX-enabled processors. SGX enables a trusted enclave to run protected code which cannot be examined from the outside, even by the operating system or hypervisor running on the machine. The major remaining channel for leakage in this system is in the pattern of accesses that the enclave code makes to the untrusted RAM lying outside the processor. It has already been noted that Oblivious RAM may be a valuable tool to eliminate that leakage [4]. Furthermore, it is expected that systems may have multiple enclaves running at the same time which wish to share information through the untrusted RAM. This scenario corresponds exactly with our multi-client ORAM setting, so the solution here could be used to securely offer that functionality.

2.3 Related Work

Most existing work on Oblivious RAM assumes only a single client. Franz et al. [6] proposed an solution for multiple clients using the original square-root ORAM, but relies on a semi-honest server. Goodrich et al. [11] extend that work to more modern tree-based ORAMs, still relying on a semi-honest server. Recent work by Maffei et al. [17] supports multiple clients with read/write access control, but again requires a semi-honest server. Other efficient solutions are possible with an even weaker security model by using trusted hardware on the server side [12, 16].

There is also a concurrent line of work in Parallel ORAMs which target ORAMs running on multi-core or multi-processor systems [1, 3, 19]. These schemes either do not target malicious adversaries or require constant and continuous communication between clients to synchronize their state. As stated above, this is not a viable solution for clients which are not always on or may be across different networks.

Currently, no solution exists that allows for multiple clients interacting with a malicious server and without direct client-to-client communication, or constant polling to create the same effect.

3 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 interacts with a malicious storage device. Below, we use Σ to represent the *interface* between a client and the actual storage device. A client with access to Σ issues Read and Write requests as needed. We stress that we make no assumptions about how the storage device responds to these requests, and allow for arbitrary malicious behavior. For instance, an adversary could respond to a Read request with old or corrupt data, refuse to actually perform a Write correctly etc. As related work, the (untrusted) storage device is part of a *server* as in envisioned applications for a (multi-client) ORAM such as 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. We emphasize that clients only interact with the server by read or write operations to a memory location; there are no other messages sent. Therefore the "protocol" is fully defined by these patterns of accesses.

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

1. $\mathsf{Init}(\lambda, N, B, \phi)$ *initializes Π. It takes as input security parameter λ, total number of blocks N, block size B, and number of clients ϕ.* Init *initializes the storage device represented by Σ and outputs a key κ.*
2. $\mathsf{ClientInit}(\lambda, N, B, j, \kappa)$ *uses security parameter λ, number of blocks N, block size B, and a secret key κ to initialize client u_j. It outputs a client state st_{u_j}.*
3. $\mathsf{Access}(\mathrm{OP}, \Sigma, st_{u_j})$ *performs operation OP on the ORAM using client u_j's state st_{u_j} and interface Σ.* Access *outputs a new state st_{u_j} for client u_j.*

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. After initializing the multi-client ORAM with Init, Algorithm ClientInit is run by each client (with no communication between them) separately and outputs their initial local states st_{u_i}. The ClientInit function only requires that each client have a shared key κ, which could

be derived from a password. Whenever client u_i executes Access on the multi-client ORAM, they can attempt to update the multi-client ORAM represented by Σ, and update their own local state st_{u_i}, but not the other clients' local states.

Finally, we define the security of a multi-client ORAM against malicious servers. Consider the game-based experiment $\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda)$ below. In this game, $\mathcal{A}$ has complete control over the storage device and how it responds to client requests. For ease of exposition, we model this as $\mathcal{A}$ outputting their own compromised version $\Sigma_{\mathcal{A}}$ of an interface to the storage device. It is this malicious interface $\Sigma_{\mathcal{A}}$ that clients will subsequently use for their Access operations. Interface $\Sigma_{\mathcal{A}}$ is controlled by the adversary and updates state $st_{\mathcal{A}}$. That is, $\mathcal{A}$ learns all clients' calls to the interface, therewith the clients' requested access pattern, and can adaptively react to clients' interface requests. To initialize the ORAM, ϕ time steps are used to do the setup for each client, then the game continues for poly(λ) additional steps where the adversary interactively specifies operations and maliciously modifies the storage.

Experiment 1 (Experiment $\mathsf{Sec}^{\mathsf{ORAM}}_{\mathcal{A},\Pi}(\lambda)$)

$b \xleftarrow{\$} \{0,1\}$
$(\kappa, \Sigma) \leftarrow \mathsf{Init}(\lambda, B, N, \phi)$
for $i = 1$ **to** ϕ **do**
    $st_{u_i} \leftarrow \mathsf{ClientInit}(\lambda, N, B, \kappa, i)$
end
$(\mathrm{OP}_{\phi+1,0}, \mathrm{OP}_{\phi+1,1}, i, st_{\mathcal{A}}, \Sigma_{\mathcal{A}}) \leftarrow \mathcal{A}(\lambda, N, B, \phi, \Sigma)$
for $j = \phi + 1$ **to** *poly*(λ) **do**
    $st_{u_i} \leftarrow \mathsf{Access}(\mathrm{OP}_{j,b}, \Sigma_{\mathcal{A}}, st_{u_i})$
    $(\mathrm{OP}_{j+1,0}, \mathrm{OP}_{j+1,1}, i, st_{\mathcal{A}}) \leftarrow \mathcal{A}(P, st_{\mathcal{A}})$
end
$b' \leftarrow \mathcal{A}(st_{\mathcal{A}})$
output *1* **iff** $b = b'$

In summary, $\mathcal{A}$ 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 and determines two operations $\mathrm{OP}_{j,0}$ and $\mathrm{OP}_{j,1}$. The oracle performs operation OP_b as client u_i with state st_{u_i}, interacting with the adversary-controlled $\Sigma_{\mathcal{A}}$ using protocol Π. Eventually, $\mathcal{A}$ guesses b.

Definition 3 (Multi-client ORAM security). *An ORAM $\Pi = (\mathsf{Init}, \mathsf{Access})$ is multi-client secure* iff *for all PPT adversaries $\mathcal{A}$, there exists a function $\epsilon(\lambda)$ negligible in security parameter λ such that*

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

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 ϕ 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 key κ used for encryption 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 only solution being *fork* consistency [15], which we achieve.

4 Multi-client Security for Classical ORAMs

We start by transforming two classical ORAM constructions, the original square-root solution by Goldreich [7] and the hierarchical one by Goldreich and Ostrovsky [8], into multi-client secure versions, retaining the same communication complexity per client. Our exposition focuses in the beginning on details for the multi-client square-root ORAM, as the hierarchical ORAM is borrowing from the same ideas.

Recall that the square-root ORAM algorithm works by dividing the server storage into two parts: the main memory and the cache, which is of size $O(\sqrt{N}))$. The main memory is shuffled by a pseudo-random permutation π. Every access reads the entire cache, plus one block in the main memory. If the block the client wants is found in the cache, a "dummy" location is read in the main memory, otherwise the actual location of the target block is read and it is inserted into the cache for later accesses. After $\sqrt{N}$ accesses, the cache is full and the client must download the entire ORAM and reshuffle it into a fresh state.

Specific 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 $\pi(x)$ 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 on the server. Downloading and later uploading the cache implies always "touching" the same $\sqrt{N}$ blocks. This is independent 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 ac-

Input: Storage interface Σ, Security parameter λ, number of blocks in each ORAM N, block size B, client number i, key κ

Output: Client state st_{u_i}

Generate permutation $\pi_{i,0}$ from key κ;

Initialize $\sqrt{N}+N$ main memory blocks (size B, shuffled with $\pi_{i,0}$) and $\sqrt{N}$ cache blocks;

Set cache counter $\chi_i = 0$; Set epoch counter $\gamma_i = 0$;

$data_i = \mathsf{Enc}_\kappa(\text{main memory})||\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i)$;

$mac_i = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i))$;

$\mathsf{ORAM}_i = data_i||mac_i$;

On Σ, replace the i^{th} ORAM by ORAM_i;

output $st_{u_i} = \{\kappa, \chi_i\}$;

Algorithm 1: $\mathsf{ClientInit}(\Sigma, \lambda, N, B, i, \kappa)$, initialize client square-root ORAM

cess, 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, $\Sigma = \mathsf{ORAM}_1, \mathsf{ORAM}_2, \ldots, \mathsf{ORAM}_\phi$, 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.

4.1 Details

We detail the above ideas in two algorithms: Algorithm 1 shows the per client initialization procedure ClientInit, and Algorithm 2 describes the way a client performs an Access with our multi-client secure square-root ORAM. The Init algorithm is trivial in our case, as it initializes Σ to ϕ empty arrays ORAM_j. Each array is of size $N + 2 \cdot \sqrt{N}$ blocks, each block has size B bits.

Before explaining ClientAccess, we first introduce the notion of an *epoch*. In general, after $\sqrt{N}$ accesses to a square-root 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

Input: Mult-client ORAM Σ, address x, new value v, client u_i, $st_{u_i} = \{\kappa, \chi_i\}$
Output: Value of block x, new state st_{u_i}

From ORAM_i in Σ: 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
    $\nu \leftarrow$ existing value of block x;
else `// operation is a` Write
    $\nu \leftarrow v$;
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 $\pi_{i,\gamma}$;
    Update ORAM_i with Enc_κ(main memory);
$\chi_i = \chi'_i + 1$; $mac_i = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i))$;
Update ORAM_i with $\mathsf{Enc}_\kappa(\text{cache}||\chi_i||\gamma_i)$ and mac_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};
        Update ORAM_j with Enc_κ(main memory);
    $mac_j = \mathsf{MAC}_\kappa(\mathsf{Enc}_\kappa(\text{cache}||\chi_j||\gamma_j))$;
    Update ORAM_j with $\mathsf{Enc}_\kappa(\text{cache}||\chi_j||\gamma_j)$ and mac_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

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, in the a cloud context, one can use $\mathrm{PRF}_\kappa(i||\gamma_i)$ as the seed in a PRG and therewith perform Knuth's Algorithm P (Fisher-Yates shuffle) [13]. Alternatively, one can use the idea of random tags followed by oblivious sorting by Goldreich and Ostrovsky [8].

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 2 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.

ClientInit: Each client runs the ClientInit algorithm to initialize their ORAM. The server stores the ORAMs (with MACs) computed with a single key κ. Each client receives their state from the ClientInit algorithm, comprising the cache counter. Note that although not captured in the security definition, our scheme also allows for dynamic adding and removing of clients. Removing is as simple as just asking the server to delete one of the ORAMs, and adding could be done by running ClientInit, but instead of initializing the blocks to be empty, the client first downloads a copy of another client's ORAM to get the most recent version of the database.

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.

Our scheme is effectively running ϕ traditional square-root ORAMs in parallel, making the overall complexity $O(\phi\sqrt{N})$. Due to limited space, see Appendix A for detailed complexity and security analysis.

4.2 Hierarchical Construction

In addition to the square-root ORAM, Goldreich and Ostrovsky [8] 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 $2 \cdot N$ buckets.

The reader is encouraged to refer to the original paper [8] 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 accessing 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 client 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 actually access a block, a client 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 the block was already 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-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 [18]. This allows for efficient verification and updating of pieces of the cache without having access to the entire thing, and it maintains poly-logarithmic communication complexity. The root of the Merkle tree will contain the counter that is incremented by the ORAM owner when they perform an access.

Deamortizing: Other authors have proposed deamortized versions of the hierarchical construction that achieve worst-case poly-logarithmic complexity, such as Kushilevitz et al. [14] and Ostrovsky and Shoup [20]. We will use as an example the "warm-up" construction from Kushilevitz et al. [14], Section 6.1. This is a direct deamortization of the original hierarchical scheme described above. They deamortize by using three separate hash tables at each level of the ORAM, labelled "active", "inactive", and "output". Instead of shuffling all at one time after 2^{j-1} accesses (which would lead to worst case $O(N)$ complexity), their approach is now different. When the cache fills up at level j, it is marked "inactive", and the old "inactive" buffer is cleared and marked "active". The idea will be that the "inactive" buffer is shuffled over time with each ORAM access, so that no worst-case $O(N)$ operations are required. As it is shuffled, the contents are copied into the "output" buffer. Accesses can continue while the "inactive" buffer is being shuffled, as long as a read operation searches both the "active" and "inactive" buffers (since a block could be in either one).

When the shuffle completes, the "output" buffer will contain the newly shuffled contents that go into level $j + 1$. This buffer is marked as "active" for level $j + 1$, the "active" buffer on level j is marked "inactive" and the "inactive" buffer is cleared and marked "active", restarting the whole process. Since the shuffle is spread out over 2^{j-1} accesses, and the shuffling was the only part that was worst-case $O(N)$, this makes a full construction that now has worst-case $O(\log^3 N)$ communication complexity.

In terms of multi-client security, the only important aspects of this process (analogous to Lemma 1) is that no elements be removed from "active" or "inactive" buffers that the owner of the ORAM has put there – until a shuffle is complete, starting a new epoch. The shuffling itself is automatically data oblivious and therewith "non-critical", in the terms we have established in this paper. Using a Merkle tree and counters, as described in the amortized version, will assure that the server cannot roll back the cache to any state prior to the last access by the owner, guaranteeing security.

Kushilevitz et al. [14] also propose an improved hierarchical scheme that achieves $O(\log^2 N/\log\log N)$ complexity, which is substantially more involving. As the deamortized hierarchical ORAM as described above is sufficient for our main contribution in Section 5, we leave it to future research to adapt Kushilevitz et al. [14]'s scheme for multi-client security.

5 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 and low constants in practice. 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.

5.1 Overview

For this section, we will use Path ORAM [24] as the basis for our multi-client scheme, but the concepts apply similarly to other tree-based schemes.

Although the interface exposed to the client by Path ORAM is the same as other ORAM protocols, it is easiest to understand the Access operation as being broken down into three parts: ReadAndRemove, Add, and Evict [22]. 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 used together form the basis of the Access operation, but it begins to illustrate the difficulty we have making this scheme multi-client secure: changing the value of a block implicitly requires reading it, meaning that both reading and writing are equally *critical* and not easily separated as our previous construction. 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 holds up to Z blocks, where Z is a small constant. Each block 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 client has a map storing for every block which leaf node the block is tagged for.

ReadAndRemove: To retrieve block x, the client 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 client 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 client 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 client keeps the block locally in a "stash" of size $Y = O(\log N)$, waiting for a later opportunity to insert the block into the tree.

Evict: So that the stash does not become too large, after every operation the client 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 order [5]) and moving blocks on that path as far down the tree as they can go, without violating the invariant. Additionally, the client inserts any matching block from the stash into the path.

Recursive Map: Typically, the client's map, which stores the tag for each block, has size $O(N \cdot \log N)$ bit and is often too large to store locally. Yet, if block size B is at least $2 \cdot \log N$ bit, the map can itself be stored *recursively* in an ORAM on the server, inducing a total communication complexity of $O(\log^2 N)$ blocks. Additionally, Stefanov et al. [24] show that if $B = \Omega(\log^2 N)$ bit, communication complexity can be reduced to $O(\log N)$ blocks.

Integrity: Because of its tree structure, it is straightforward to ensure integrity in Path ORAM. Similar to a Merkle tree, 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 additional challenges when trying to add multi-client capabilities with our previous strategy. First, recursively storing the map into ORAMs imposes a problem. To resolve a tag, each path accessed in the recursive ORAMs has to be different for each client. If we separate the map into ϕ separate ORAMs (which we will do), the standard recursive lookup results in a large blowup in communication costs. At the top level of the recursion, we would have ϕ ORAMs, one for each client. Yet, each of those will fan out to ϕ ORAMs to obliviously support the next level of recursion, each of which will have ϕ more, going down $\log n$ levels. The overall communication complexity for the tag lookup would be $\phi^{\log N} \in \Omega(N)$.

Second, an Add in Path ORAM cannot be performed without ReadAndRemove, so we cannot easily split the access into *critical* and *non-critical* parts like before.

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

1. **Unified Tagging:** Instead of separately tagging blocks in each of the ORAMs and storing the tags recursively, we will have a unified tagging system where the tag for a block can be computed for any of the separate ORAMs from a common "base tag." This is crucial to avoiding the $O(N)$ communication overhead that would otherwise be induced by the recursive map as described above. For a block x, the map will resolve to a base tag value t. This same tag value is stored in every client's recursive ORAM. Let h be a PRF mapping from $[0, 2^\lambda) \times [1, \phi]$ to $[0, N)$. The idea is now that the leaf that block x will be percolating to in the recursive ORAM tree differs for every ORAM of every client u_i and is pseudo-randomly determined by value $h(t, i)$. This way, (1) the paths accessed in all recursive map ORAMs for all clients differ for the same block x, and (2) only one lookup is necessary at each level of the recursive map to get the leaf node tag for all ϕ ORAMs.
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. [11], 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. As 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.

5.2 Details

To initialize the multi-client ORAM (Algorithm 3), ϕ separate ORAMs are created and the initial states (containing the shared key) are distributed to each client. For each client u_i, the ORAM takes the form of a series of trees $T_{j,i}$. 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 $\mathsf{ReadAndRemove}$ 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, finds the address of block x, and finally retrieves it from the main tree. For each recursive tree, it retrieves a tag value t allowing 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

Input: Security parameter λ, number of blocks in each ORAM N, block size B, number of clients ϕ, initialization sub-routine for multi-client 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
  `// Let` m `be number of recursive trees made in previous loop`
  $m = i$;
  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
Initialize Σ to hold $\{\mathsf{ORAM}_1, \ldots, \mathsf{ORAM}_\phi\}$;

Algorithm 3: $\mathsf{Init}(\lambda, N, B, \phi)$, initialize multi-client tree-based ORAM

own ORAM and writes back to the ORAMs of the other clients. We use a simple MAC technique for paths $\mathsf{MACPath}$, Algorithm 6, which we have moved to the appendix.

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

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.

The overhead from the additional metadata ORAMs that we have in our construction is fortunately not dependent on the block size B. Therefore, if B is large enough, we can achieve as low as the overhead of single-client Path ORAM, $O(\log N)$, or a total complexity of $O(\phi \log N)$ for ϕ users. However, this only applies if the block size B is sufficiently large, at least $\Omega(\log^4 N)$. Otherwise, for smaller B, the complexity can be up to $O(\phi \log^5 N)$. Due to limited space, see Appendix B for a detailed complexity and security analysis.

6 Conclusion

We have presented the first techniques that allow multi-client ORAM, specifically secure against *fully malicious* servers. Our multi-client ORAMs are reasonably efficient

Input: Address x, client u_i, $st_{u_i} = \{\kappa, \chi_i\}$, sub-routine for multi-client classical ORAM MAccess
Output: The value of block x

`// Let` m `be recursion depth,` n_j `the number of blocks in tree` j

Retrieve root block $\mathcal{R}$;
$pos = x/n$; $x_m = \lfloor pos \cdot (B/\lambda) \rfloor$; $t_m = \mathcal{R}[x_m]$`// Find tag` t_m `of address` x;
$t'_m \xleftarrow{\$} [0, 2^\lambda)$`// Compute new tag` t'_m `for` x;
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 $T_{p,j}$;
    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

with communication complexities as low as $O(\log N)$ per client. Future work will focus on efficiency improvements, including reducing worst-case complexity to 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 sub-ORAMs 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 as a subroutine. Finally, it may be interesting to investigate whether multiple clients can be supported with a more fine-grained access control, secure against fully malicious servers.

Input: Address x, new value v, client u_i, $st_{u_i} = \{k, \chi_i\}$, the number of recursive trees m
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: $\mathsf{Evict}(\Sigma, st_{u_i})$ – Perform Evict on multi-client tree-based ORAM

References

[1] Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel ram and applications. In *Theory of Cryptography Conference*, pages 175–204. Springer, 2016.

[2] D. Cash, A. Küpçü, and D. Wichs. Dynamic proofs of retrievability via oblivious ram. In *Conference on the Theory and Applications of Cryptographic Techniques*, pages 279–295. Springer, 2013.

[3] T-H. Hubert Chan and Elaine Shi. Circuit opram: A (somewhat) tight oblivious parallel ram. Cryptology ePrint Archive, Report 2016/1084, 2016. `http://eprint.iacr.org/2016/1084`.

[4] V. Costan and S. Devadas. Intel SGX explained. Technical report, Cryptology ePrint Archive, Report 2016/086, 2016. https://eprint. iacr. org/2016/086, 2016.

[5] C.W. Fletcher, L. Ren, A. Kwon, M. Van Dijk, E. Stefanov, and S. Devadas. Tiny ORAM: A Low-Latency, Low-Area Hardware ORAM Controller. Cryptology ePrint Archive, Report 2014/431, 2014. `http://eprint.iacr.org/`.

[6] M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, A. Peter, R. Sion, and M. Sotakova. Oblivious outsourced storage with delegation. In *International Conference on Financial Cryptography and Data Security*, pages 127–140. Springer, 2011.

[7] O. Goldreich. Towards a Theory of Software Protection and Simulation by Oblivious RAMs. In *Symposium on Theory of Computing*, pages 182–194, New York, USA, 1987.

[8] O. Goldreich and R. Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM*, 43(3):431–473, 1996. ISSN 0004-5411.

[9] Michael T. Goodrich. Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in o(n log n) time. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC'14, pages 684–693, 2014. ISBN 978-1-4503-2710-7.

[10] 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.

[11] M.T. Goodrich, M. Mitzenmacher, O. Ohrimenko, and R. Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 157–167, 2012.

[12] A. Iliev and S.W. Smith. Protecting Client Privacy with Trusted Computing at the Server. *IEEE Security & Privacy*, 3(2):20–28, 2005.

[13] D.E. Knuth. *The Art of Computer Programming, Seminumerical Algorithms*, volume 2, chapter 3.4.2, pages 139–140. Addison Wesley, 2nd edition, 1981. ISBN 978-0201896848.

[14] E. Kushilevitz, S. Lu, and R. Ostrovsky. On the (in)security of hash-based oblivious RAM and a new balancing scheme. In *Proceedings of Symposium on Discrete Algorithms*, pages 143–156, Kyoto, Japan, 2012.

[15] J. Li, M.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.
[16] J.R. Lorch, B. Parno, J. Mickens, M. Raykova, and J. Schiffman. Shroud: Ensuring private access to large-scale data in the data center. In *USENIX Conference on File and Storage Technologies*, pages 199–213, 2013.
[17] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder. Privacy and access control for outsourced personal records. In *IEEE Symposium on Security and Privacy*, pages 341–358. IEEE, 2015.
[18] R.C. Merkle. A digital signature based on a conventional encryption function. In *Advances in Cryptology*, pages 369–378. Springer, 1988.
[19] K. Nayak and J. Katz. An oblivious parallel ram with $o(\log^2 n)$ parallel runtime blowup. Cryptology ePrint Archive, Report 2016/1141, 2016. `http://eprint.iacr.org/2016/1141`.
[20] R. Ostrovsky and V. Shoup. Private Information Storage. In *Proceedings of Symposium on Theory of Computing*, pages 294–303. ACM, 1997.
[21] 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.
[22] E. Shi, T.-H.H. Hubert Chan, E. Stefanov, and M. Li. Oblivious RAM with $O(\log^3(N))$ Worst-Case Cost. In *Proceedings of Advances in Cryptology – ASIACRYPT*, volume 7073, pages 197–214, Seoul, South Korea, 2011. ISBN 978-3-642-25384-3.
[23] SpiderOak. Semaphor, 2016. URL `https://spideroak.com/solutions/semaphor`.
[24] E. Stefanov, M. van 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.
[25] WhatsApp. Whatsapp encryption overview, 2016. URL `https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf`.

A Security and Complexity Analysis of Square Root Construction

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 begins execution of 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

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

Proof. The claim immediately 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. Let

MAC be a message authentication code with an adversarial advantage in the existential forgery game that is negligible in λ. If during access j a client u_i sees a counter equal to the counter value at the end of access $j-1$, and the MAC verifies, then there exists a negligible function ϵ such that the client is sure with probability $1-\epsilon(\lambda)$ that every element in the cache during access $j-1$ is also in the cache during access j (unless there was a shuffle between). □

That is, if two accesses to the same block occur in the same epoch, at the start of the second access either the block will be in the client's cache or they will output Abort.

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

Proof. To prove security according to Experiment 1, we must show that the adversary cannot adaptively choose two access patterns, $< OP_{0,0}, \ldots, OP_{0,\ell} >$ and $< OP_{1,0}, \ldots, OP_{1,\ell} >$, such that the induced access pattern on the server by the clients executing one of those access patterns allows him to distinguish which pattern it is, with non-negligible advantage. For our construction, this reduces to showing indistinguishability of induced access patterns for the two parts of the access: the cache operation, and the access to main memory.

The first 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). For any two operations, this is an indistinguishable induced access pattern. If the MAC verifies, then the client continues, and if it does not then he outputs Abort. By security of the MAC (negligible adversarial advantage in λ against existential forgery), the client only outputs Abort if the server tampers with the data. Thus, an adversary seeing Abort does not learn anything besides what they already know: whether the adversary has tampered or not. The actual data read and written during this part will be indistinguishable for any two operations under the security of the IND-CPA encryption scheme used. Therefore, with secure encryption (negligible adversarial advantage in λ during IND-CPA game) and MAC, any pair of operations OP_0 and OP_1 results in computationally indistinguishable induced access patterns for this segment of the access.

The main memory segment 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. Lemma 1 ensures that with all except negligible probability in λ 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 (pseudo-)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.

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. Since each client only accesses their own main memory, and they keep a counter of what epoch

they were in during the last access, there is no way that the server can "roll back" to a previous epoch without the client noticing. If the client see's that their ORAM is in a higher epoch than the last time they accessed it, and all MACs verify, then they know that another client has performed a shuffle for them and they update their epoch counter to the new epoch. At that point, no accesses to the main memory can have been made since only the owner of that ORAM would read the main memory, and they have not performed an access since it rolled over to a new epoch.

In conclusion, given a secure MAC and IND-CPA encryption, we achieve multi-client ORAM security with probability $1 - \epsilon(\lambda)$ for some negligible function ϵ. □

Fork consistency: When a client makes an access, they add an element to the cache for all ϕ clients. Therefore, at any given timestep, if the server is not maliciously changing caches, all caches will have the same number of elements in them. Since each cache is verified by a MAC, the server cannot remove individual elements from a cache. The only viable attack is to present an old view of a cache which was at one point valid, but does not contain new updates that have been added by other clients. If the server chooses to do this, he creates a *fork* between the views of the clients which have seen the update and those that have not. Since the server can never "merge" caches together, but only present entire caches that have been verified with a MAC by a legitimate client, there is no way to reconcile two forks that were created without a client finding out. This achieves fork consistency for our scheme.

Complexity: Making the square-root solution multi-client secure does not induce any additional asymptotic complexity, per client. Each access requires downloading the cache of size $\sqrt{N}$ and accessing one block from the main memory. Every $\sqrt{N}$ accesses, the main memory and cache must be shuffled, requiring N communication if the client has enough storage to temporarily hold the database locally. If not, then Goldreich and Ostrovsky [8] noticed that one can use a Batcher sorting network to obliviously shuffle the database with complexity $O(N \log^2 N)$, or the AKS algorithm with complexity $O(N \log N)$. One can also reduce the hidden constant using a more efficient Zig-Zag sort [9]. In the first scenario, the amortized overall complexity is then $O(\phi\sqrt{N})$, while the second is $O(\phi\sqrt{N} \log N)$.

Goodrich et al. [10] also 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. Since the shuffling is completely oblivious (does not depend on any pattern of data accesses) and memoryless (the clients only need to know what step of the shuffle they are on in order to continue the shuffle), it can be considered a "non-critical" portion of the algorithm and no special protections need to be added for malicious security.

Note on computational complexity: While Algorithm 1 returns the whole updated state Σ, in practice a client only needs to update the other clients' caches (up to $\sqrt{N}$ times). 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 re-

Input: Σ, st_{u_i}, path $\mathcal{P}$, stash S, χ, eviction counter e
Output: Updated MAC values
1 **for** $j = \log n$ *to* 1 **do**
2 $V[j] = \mathsf{MAC}_\kappa$(contents of bucket $\mathcal{P}[j]$, MAC of left child, MAC of right child);
3 **end**
// Root MAC over the stash and tree parameters χ and e
4 $V[0] = \mathsf{MAC}_\kappa(S$, MAC of root bucket, $\chi, e)$;
5 **return** V

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

moved. 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.

Unified cache: A natural optimization to this scheme is to have one single shared cache instead of a separate one for each user. If the server behaves honestly, then all caches will contain the same blocks and be in the same state anyway, so a single cache can save some communication and storage. To still protect against a malicious server, one must be careful in this case to store ϕ different counters with the cache and have each client only increment their counter when they do an access. This ensures that if a client inserts a block into the cache it cannot be "rolled back" past the point that they inserted that block without them noticing. Since the cost to reshuffle the ORAMs dominates complexity of our scheme, this optimization does not change asymptotic performance. Resulting in only a small constant improvement and making the presentation and proof unnecessary difficult, we omit full discussion of this technique.

B Security and Complexity Analysis for Tree-based Construction

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 2. *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 tied to a (verifiable) counter, so against a malicious adversary the probability is 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* (Init, 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 2, 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. □

B.1 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 $O(\phi \cdot B \cdot \log^2 N)$ bits of 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 communication complexity is then $O(\phi \cdot \log^2 N \cdot [B + \mu(N, \log N)])$ bit. The deamortized hierarchical ORAM by Kushilevitz et al. [14] has $O(\log^3 N)$ blocks communication complexity, where each block is of size $\log N$ bit (the meta-data we need for our construction). Taking this hierarchical ORAM as a sub-ORAM, the total communication complexity computes to $O(\phi \cdot \log^2 N[B + \log^4 N])$ bits. If $B \in \Omega(\log^4 N)$ then the communication complexity, in terms of blocks, is $O(\phi \log^2 N)$, otherwise it is at most $O(\phi \log^5 N)$, i.e., with the assumption $B \in \Omega(\log N)$ (the minimal possible block size for Path ORAM to work).

Additionally, if we use the recursive optimization trick from Stefanov et al. [24] to reduce the overhead from the Path ORAM part of the construction from $O(\log^2 N)$ to $O(\log N)$, we can achieve a total complexity of $O(\log N)$ for blocks of size $\Omega(\log^4 N)$.

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. [24] 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.