Blurry-ORAM: A Multi-Client Oblivious Storage Architecture

Nikolaos P. Karvelas[1], Andreas Peter[2], and Stefan Katzenbeisser[1]

[1] TU Darmstadt, Germany
[2] University of Twente, The Netherlands

Abstract. Since the development of tree-based Oblivious RAM by Shi et al. (Asiacrypt '11) it has become apparent that privacy preserving outsourced storage can be practical. Although most current constructions follow a client-server model, in many applications it is desirable to share data between different clients, in a way that hides the access patterns, not only from the server, but also between the clients. In this work, we introduce Blurry-ORAM, an extension of Path-ORAM that allows for oblivious sharing of data in the multi-client setting, so that accesses can be hidden from the server and other clients. Our construction follows the design of Path-ORAM as closely as possible in order to benefit from its performance as well as security. We prove our construction secure in a setting where the clients are semi-honest, do not trust each other but try to learn the access patterns of each other.

Keywords: Path-ORAM, Oblivious storage, multiple clients

1 Introduction

As cloud applications continue to grow in popularity, outsourcing sensitive data to remote servers has become common practice, and with it, significant ramifications to users' privacy have been induced. While encryption plays a central role in data protection and privacy, it has become apparent that simply employing encryption is not enough to protect sensitive data, since significant information leakage can already occur when a remote server merely observes the clients' access patterns. Take for example, the case of genomic data: it is foreseeable, that in the future, genomic data will be stored as part of patients' electronic health records. This has the benefit that large sets of genomic data will also be available to research institutions, which will have the opportunity to carry out genomic tests, such as Genome Wide Association Studies (GWAS), in large datasets. In this setting, a third party, such as a medical research center (further called an investigator), could be granted access to specific parts of multiple patients' genomic data, which she could analyze in a study. Clearly, in such a setting, encryption is not enough to protect the interests of both the participants and the research center: even with the data encrypted, the server or an investigator can observe which portions of the genome are accessed by other investigators, and deduce vital information, such as what test has been run, or what disease a patient is suspected to suffer from. In [7] the authors proposed an architecture, which guarantees access pattern privacy for one single investigator, against honest but curious servers; the construction is based on the hierarchical ORAM of Williams et al. [15]. Besides heavy computational and communication costs, the solution suffers from the problem that it cannot guarantee access pattern privacy if multiple investigators are present, which is very realistic in the scenario of genomic databases. This is due to the fact that multiple investigators can observe and record the position of certain datablocks in the ORAM, which is an indication to fellow investigators whether a block has been accessed. Note that this does not contradict

the security requirements of an ORAM, as they only provide access pattern hiding against the server.

Inspired by the above scenario, in this work, we construct an ORAM that guarantees access pattern privacy in the presence of multiple clients (investigators). We construct Blurry-ORAM, an ORAM solution which is based on the highly efficient PathORAM [14], but allows multiple clients to store their private data on a server and share some parts of the data with each other. Abstractly, an investigator can be treated as a client, whose only data items are the ones to which he has been given access by other clients.

The problem of constructing a privacy preserving multi-client storage solution has been explored in a number of works, namely [17,3,6,9,10,1] and should not be confused with "Parallel ORAMs", where one dataset is accessed obliviously by multiple clients in *parallel*, while all the clients can read and write *all* the data in the dataset, a problem investigated in [16,8,2]. In a privacy preserving multi-client storage, each client has his own data (all stored in a single ORAM) and is free to share only parts of his data with other clients. In [3] the authors propose a solution, where a data owner can delegate rights to read or write some of his data items to other clients. This solution is based on the square-root ORAM [4] and thus suffers from heavy communication complexity. Equally important, it requires the data owner to be constantly online, thus restricting the applicability of the solution. The construction in [6] avoids this drawback, but suffers from high storage requirements on the client side. In a recent work of Maffei et al. [9], a multi-client ORAM is proposed, which is based on Path-ORAM [14] and achieves good security guarantees against a malicious server. Regarding the security against the clients, however, the main security focus lies on anonymity, i.e., unlinkability of a given operation on a datablock to a client, among the set of clients who have access to that specific data-block. The construction does not guarantee hiding access patterns between clients who share data, as privacy leakage can occur due to the stash or due to the position map. Facing a similar problem, Backes et. al. [1], examined the problem of access pattern anonymity in a multi-client ORAM, and proposed two constructions that achieve this goal. We stress here, that these solutions tackle a different problem, than the one we are dealing with here. They focus on anonymity of data accesses and do not consider data sharing between clients.

In this work, we focus on the development of an ORAM, that allows multiple clients to store their data on a server and share it with each other. At the same time, the system protects the access pattern privacy of the clients, not only against the server, but also against other clients as well. In our solution, we assume K semi-honest clients, with each of them storing up to N blocks of data. Every client encrypts his datablocks with his individual encryption key, and all encrypted datablocks are stored in a classical Path-ORAM of height $\log N$, with Z blocks per node. The resulting K Path-ORAM trees are merged on each node, resulting in a Path-ORAM of height $\log N$ with ZK blocks per node. At this point, in every node, blocks of all the clients can be found. After a number of data accesses, the blocks found in every node are eventually shuffled, thus ending up with a "blurred" version of the originally merged client trees, hence the name. Sharing a data block between two clients is done by having the original block owner change the private key under which the block is encrypted and handing over this new key to the client. Our ORAM construction is very efficient, and achieves full privacy, i.e., provides access pattern hiding against both the server and other clients. In Table 1 we sum up properties of the existing multi-client ORAM constructions, show their communication complexity and recall if they allow access pattern privacy against other clients, allow for anonymous accesses or sharing of data between the clients. To our knowledge, our construction is the first to allow data sharing between clients in an ORAM, and guarantee access pattern hiding not only against the server, but also against other clients.

Solution	Commun.Complexity	A.P. Hiding (Server)	A.P. Anonymity	A.P. Hiding (Clients)	Data sharing
[3]	$O(\sqrt{N}\log^2 N)$	✓	×	×	✓
[6]	$O(\log^2 N)$	✓	×	×	✓
[17]	$O(\log^3 N \log\log N)$	✓	×	×	×
[10]	$O(\log^2 N)$ to $O(\log^5 N)$	✓	×	×	✓
[9]	$O(G\log^2 N)$	✓	✓	×	✓
[1](linear)	$O(K\log^2 N)$	✓	✓	×	×
[1](polylog)	$O(\log^2(KN))$	✓	✓	×	×
This work	$O(KG\log^2 N)$	✓	×	✓	✓

Table 1: Comparison of multi-client ORAM solutions. N: Number of blocks per client, G: Number of groups, B: Block size, K: Number of clients.

2 System Model

In our ORAM construction, we consider one server and multiple clients. Both the server and the clients are assumed to be semi-honest, i.e., they try to extract as much information possible about the data belonging to other clients, but they never deviate from the protocol. Each client stores his data on the server, partitioned in N blocks of fixed size B, and each block is identified by a unique identifier id. Thus, we consider a block as a tuple $(\mathsf{id}, \mathsf{dat})$, with id being a unique identifier and dat the actual data. In the following we will abuse this notation, and, for ease of presentation, denote by id_i^j the block with identifier id_i, that can be accessed by client j, without referring to the actual data of the block, unless it is crucial for the description.

2.1 Functionality

Since we want to build a multi-client ORAM solution that allows block sharing between the clients, our architecture must support operations for reading, writing, sharing and revoking access to shares. The operations used in our construction are defined as follows:

Init(λ): The Init operation is run by every client and takes as input a security parameter λ. It outputs an encryption/decryption key pair, which the client uses to encrypt and decrypt his datablocks.
Read(id_i^j, $\mathsf{enc\_key}$, $\mathsf{dec\_key}$): The read operation is a protocol run between a client cli_j and the server. It returns the block identified by id_i^j or NULL, if the block was not found (for example, if a client queries for a block to which he does not have access rights).
Write(id_i^j, $\mathsf{enc\_key}$, $\mathsf{dec\_key}$, dat): The write operation is a protocol executed between a client cli_j and the server. It is similar to the read operation, as it returns the block with identifier id_i^j, but overwrites its contents with dat, in case the read operation was successful.
Share(cli_i, cli_j, id_u^i, $\mathsf{enc\_key}$, $\mathsf{dec\_key}$): The share operation is a protocol run between the server and the clients cli_i and cli_j. The goal is to make the block with identifier id_u^i, which is accessible by client cli_i, also accessible (for read, write, share and revoke) to client cli_j.
Revoke(cli_i, cli_j, id_u^S, $\mathsf{enc\_key}$, $\mathsf{dec\_key}$): The revoke operation is a protocol run between the server and a client cli_i. For a block with identifier id, that can be accessed by a set of clients

S, the purpose of this operation is to disallow cli_j (who is a member of S) from further being able to access (read, write, share or revoke) block id_u^S. Note, that the revoke operation is not recursive and disallows only one specific client (client cli_j) all further access to the block; thus, after a revoke operation, all other clients, to whom the revoked client granted access in the past, are still allowed to read, write, share and revoke the particular block.

2.2 Definitions

In order to be able to argue about the security of our scheme, we first need to define the notion of a view (or access pattern). We do this in the following definitions, that are tailored towards our setting of extended ORAM functionality, that allows block sharing between clients. To the best of our knowledge, we are the first to assume semi-honest clients that share data, without being constantly online.

Definition 1 *A data request is a tuple of the form* $(\mathsf{op}, \mathsf{id}_u^i, \mathsf{dat}, \mathsf{cli}_i, \mathsf{cli}_j)$. *The operation* op *is one of* $\{\mathsf{Read}, \mathsf{Write}, \mathsf{Share}, \mathsf{Revoke}\}$*;* id *is the block's identifier, and* dat *is the data to be written. If* $\mathsf{op} = \mathsf{Read}$*, then* $\mathsf{dat} = \mathsf{NULL}$ *and* $\mathsf{cli}_j = \mathsf{NULL}$*. If* $\mathsf{op} = \mathsf{Write}$*, then* $\mathsf{cli}_j = \mathsf{NULL}$*. If* $\mathsf{op} \in \{\mathsf{Share}, \mathsf{Revoke}\}$*, then* $\mathsf{dat} = \mathsf{NULL}$ *and* cli_j *is the client with which the block will be shared or from whom the sharing will be revoked.*

Definition 2 *A data request sequence is a tuple of the form* $(\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_l})$*, where each* $\boldsymbol{x_i}$ *is a data request. The number of data requests in a data request sequence is called the data request sequence's length.*

Definition 3 (View) *Let* X *be a data request sequence of length* l*. We call the sequence of all data read and written from the client and the server during the execution of* X*, the view (or access pattern) induced by* X*.*

Definition 4 (Shared Block) *In a multi-client ORAM construction that allows data sharing between clients, we call a block* id_i *shared if at least two clients have access to it.*

Definition 5 (Group of shared blocks) *In a multi-client ORAM construction that allows data sharing between clients, we call a set of shared blocks, which are accessible by the same set of clients, a group of shared blocks.*

2.3 Security Properties

We consider a semi-honest server and semi-honest clients that do not collude with each other, or with the server. Thus, all involved parties do not deviate from the protocol, but try to gain as much information possible (e.g., which data blocks were read or written by which client, how many blocks are shared with whom, etc.) by examining the views of the access transactions. As far as the security against the server is concerned, we follow the classical access pattern privacy definition, [14]:

Definition 6 (Client-to-Server Privacy) *We say that a multi-client ORAM protocol provides client-to-server privacy, if any two views of a client, that are of same length, are computationally indistinguishable by the server.*

Hiding the access patterns against the server, is already a non-trivial problem. When multiple clients are present, the situation becomes much more involved. Suppose that two clients share blocks with each other, and that one of them, the attacking client, acquires the views

of the other client's accesses (for example by eavesdropping the communication channel). The views potentially include datablocks, that the attacker shares with the client and he can thus decrypt. Therefore, traditional proof techniques used in the simple client-server model (showing, for example indistinguishability of the views) cannot be adapted directly to the multi-client case. This is due to the fact that the adversary can now read parts of the view. Thus, we must come up with a security model, that takes these observations into account.

In the following we attempt to model the behaviour of a semi-honest client, $\mathcal{A}$ acting as the attacker who wants to infer access patterns of another client. We assume that $\mathcal{A}$ will share some blocks with a client, whose access pattern privacy he intends to break. $\mathcal{A}$ will first perform a series of queries in an attempt to bring the ORAM in a state that allows him to execute his attacks. Then he will let the client under attack to perform his query. Subsequently, $\mathcal{A}$ will perform another series of queries before he finally guesses the access of the client under attack. This situation closely resembles the IND-CPA security game, played against an encryption scheme, where the adversary asks for a series of encryptions in a "learning phase", then sends two messages to his challenger, who randomly chooses one, encrypts and returns it. The adversary finally has to guess which message was chosen. It is therefore reasonable that we model the security of an ORAM against access pattern leakage between clients, in close analogy to the IND-CPA game, by a security game played between an adversary and a challenger, simulating both the server and the clients. As setup, the clients give their data to the server, and the adversary stores his blocks on the server, along with other clients. In a first "learning phase", the adversary is able to initiate any queries he wishes on the server, effectively bringing the server into a certain state. These queries cannot include accesses to blocks that the adversary has access to (after all he himself is also a client, and is thus allowed to read, write or share his blocks with any of the other clients he wishes) but also queries to blocks that belong to other clients and the adversary does not have access to. To do this, we give our adversary, "oracle" access to operations on blocks he does not have access to. In essence, the adversary asks other clients to perform specific queries of his choice, while he obtains the resulting views. At some point the adversary issues two data request sequences on blocks that he does not have access to (after all, we are interested in breaking the access pattern privacy for blocks that the adversary cannot see) and sends them to the challenger[3]. The challenger chooses one of these sequences at random, performs the query, and sends the resulting view to the adversary. The adversary then continues with a second learning phase, where he is again allowed to issue any data requests he wishes to the server. Intuitively, one would think that in this phase, the adversary should not be allowed to perform "share" or "revoke" operations on the blocks queried for during the challenge phase. However, if the share and revoke operations, involve changing of the encryption key, this restriction can be mitigated, since the adversary in this second learning phase, would not be in position to decrypt a block found in the view yielded during the challenge phase. At the end of the game, the adversary outputs a bit, indicating his guess for the data request sequence chosen by the challenger. The adversary wins the game if his guess was correct.

2.4 Security Definitions

The above observations can be summarized in a security game, which we call indistiguishability under chosen query attack (IND-CQA) game. The game is detailed in Table 2. We can then define access pattern privacy in a multi-client ORAM construction as follows:

[3] In this work we restrict these data request sequences to be on read and write operations, and leave the inclusion of share and revoke operations in the challenge phase, as future work

Definition 7 (Client-to-Client Privacy) *We say that a multi-client ORAM protocol which allows sharing of blocks between the clients provides client-to-client privacy (i.e., hides the access patterns of a client against semi-honest clients), if for every PPT adversary $\mathcal{A}$, the advantage of winning the IND-CQA game, described in Table 2 is negligible in the security parameter.*

Initialization: The challenger runs the Init algorighm and creates the public/private key pairs for all the clients except the adversary. The adversary runs the Init algorithm, creates his own private/public key pair and sends his public key to the challenger. The challenger uploads every client's encrypted data blocks to the ORAM and the adversary uploads his own data blocks.

Pre-challenge phase: For polynomially many (in the security parameter) times, the adversary runs any data request (may it be read, write, share or revoke) on any blocks of his choosing. For blocks he has access to, he executes the data request with the challenger. For blocks that the adversary does not have access to, he gets "oracle" access and receives by the challenger the view yielded by the corresponding data request of this operation.

Challenge: The challenge phase consists of two steps:

1. The adversary chooses two data requests (r_c^0, r_c^1) and sends them to the challenger. The two data requests must be read or write requests, performed on blocks that the adversary does not have access to.
2. The challenger chooses randomly a bit b, runs r_c^b and returns the view to the adversary.

Post-challenge phase: Identical to the pre-challenge phase.

Guess: The adversary guesses which data request was ran by the challenger during the challenge phase, and outputs a bit b'. He wins the game if $b' = b$.

Table 2: The IND-CQA game.

3 Blurry-ORAM Construction

3.1 Review of Path-ORAM

Our starting point is the Path-ORAM construction of Stefanov et al. [14], where N blocks of data are stored in a binary tree structure of N leaves, with each node holding Z blocks. If less than Z real blocks are stored in a node, then the node is filled up with fake blocks. Real and fake blocks are encrypted under a semantic secure encryption scheme, ensuring that encryptions of real and fake blocks are indistinguishable. Since each tree node can hold up to a constant amount of Z blocks, then for N real blocks, $Z(2N-1)-N$ fake blocks are stored in the structure. Each real block is mapped to a leaf of the tree, and every time the client wants to retrieve one of his elements, he downloads the whole path from the root node to the respective leaf; he is guaranteed to find the desired block in one of the nodes along this path. Once the path has been downloaded, the client chooses randomly a new leaf, re-maps the retrieved block to this leaf, and places it in the node closest to the leaf, which is the common ancestor of the retrieved element's previous and new mappings, if there is enough space in its buckets. Otherwise the

block is moved to higher and higher levels, until a node with enough space is found. However, it can happen (in fact, this event occurs with probability 1/2 during every remapping) that the only common ancestor of the two leaves is the root node. Thus, the root may quickly get filled up with elements. In this case, a small auxiliary storage, called a *stash*, is used to store the element instead, which is shown to grow only logarithmically (in the amount of the elements stored in the tree). Furthermore, for every real block replaced in the path's node, a fake block is put in its position and all blocks in every accessed node are rerandomized. The resulting path is then uploaded to the server, while the stash (due to its small size) is stored directly on the client. Note here, that in order for the client to know to which leaf an element is mapped in the tree, the client must store a table (called *position map*) that grows linear with the amount of elements he has outsourced to the server. In order to overcome this problem, the authors implement a recursive solution, first introduced in [11]. The key insight is that the position map is smaller than the original data and can thus be stored in another (smaller) Path-ORAM. This procedure can be repeated recursively, with the position map of the smaller ORAM stored in an even smaller one, until a Path-ORAM of constant size is achieved. Thus, in order to access one element, the client must first recursively retrieve this element's position from all smaller Path-ORAMs.

3.2 Construction of Blurry-ORAM

In our setting, we consider K clients, who store their data on a data structure residing on a remote server. Every client stores a maximum number of N real blocks and their corresponding fakes, just like in Path-ORAM. How the clients' data is laid on the remote data structure is of grave importance: The easiest way to do this, would be to construct a tree with KN leaves and assign every block to one of those leaves, as in a Path-ORAM. However, such a solution affects the stash size in a way that renders the underlying Path-ORAM inoperable, due to an exceedingly big stash size. The reason for this behavior is the following: Assume that each node of the tree can hold Z blocks and that all clients' blocks are uniformly distributed in the tree. Assume further, that every client can access only those blocks that belong to him or are shared with him. Then, in every path, a client can only find on average $Z/K \log(KN)$ blocks belonging to him, as opposed to $Z \log N$ blocks that he would find if he had stored his blocks in a single client Path-ORAM. This means that it will be more difficult for him to put the element he read back into the path, which will eventually force him to use his stash more often. We indeed observed this behavior experimentally.

In contrast, in Blurry-ORAM, we store the clients' blocks differently: We let each of the K clients store his N blocks in a separate binary tree with N leaves, where each node holds ZK blocks. Every block (real or fake) is encrypted using the client's public key, but in such a way that the block can be re-randomized without knowledge of the owner's public key (using for example the encryption scheme proposed in [5])[4]. As our construction is based on Path-ORAM, it inherits the need of using a stash, since there is a chance, that during an access, some blocks cannot be put back in the downloaded path. For blocks that belong solely to one client each client locally maintains a stash (called localstash in the rest of the paper). The bounds on Path-ORAM stash size apply here. However it might happen, that blocks shared between clients cannot be written back into the path. For this case we maintain a so-called "commonstash" which will contain all the encrypted shared blocks that could not fit into the tree. The size of this commonstash can be upper bounded, as we show in Section 4; this, we can thus fix its size,

[4] Note here that as long as one uses an IND$-CPA secure encryption scheme, the fake blocks that belong to every client are accessible only by their owner and are not shared, thus no client can distinguish between real and fake blocks that do not belong to him.

as a parameter of the system. The commonstash is initially filled with encrypted fake items, so that the server cannot observe if shared blocks have been added or removed, and stored on the server; each client retrieves this stash before he performs any operation. Furthermore, we use a dedicated private key for the fake blocks on the commonstash, so that the clients can distinguish between fake and real blocks in the commonstash. Unfortunately however, this ability of the clients can be a source of privacy leakage: clients who see shared datablocks on the commonstash, immediately know, that where previously the shared blocks resided, now real datablocks are to be found. Therefore, we must make sure that the commonstash remains as small as possible. We achieve this by changing the way the Read (and thus all other operations) operation is performed in Path-ORAM. In order for a client to read a block id, he first finds the leaf i, to which id is mapped. The client then downloads the commonstash and *two* paths [5]: One determined by the leaf i (which we will from then on call the "original" path), and the "symmetric" path, which is the path leading to the symmetric leaf of i, when considering as symmetry axis the line that cuts the leaf level into two parts of equal size. The client then identifies the blocks he has access to (real, fake and shared). This is done, with the client iterating on his keys, and checking for every block, if a specific condition is fulfilled (for example, by trying to decrypt while using an encryption scheme that returns $\perp$ when decrypting with the wrong key). By construction, one of them is guaranteed to be the block he is looking for. Consequently, the client copies the real and shared blocks of his in a local list, along with the blocks in his localstash, and replaces his real, shared and fakes in the paths, with empty placeholders. The client can now use all the empty placeholders in the paths for his eviction. First the client evicts all the shared blocks, trying to store them as deep down in either of the paths as possible. If a shared block cannot fit in any of the paths, it replaces a fake block in the commonstash. Subsequently, he evicts those blocks that are not shared, in a similar manner. If there is not enough space in the paths, then the localstash is used. The client then fills up the remaining placeholders with fake elements of his and finally re-randomizes all the blocks in the paths and the commonstash. The paths and the commonstash are then sent back to the server. Indeed, using these ideas, we observed also experimentally that the stash sizes were very small, with the commonstash being almost empty during all our experiments, even when we let Z as small as 2.

In our construction, sharing a block between clients is straightforward: Suppose client cli_i wants to share block id_u, with client cli_j; cli_i retrieves his block id_u, re-encrypts the block with the new key and uploads the block to the server. Finally, cli_i hands over to cli_j the new key and the index of leaf to which id_u is mapped. In a similar way, revocation of access rights is performed: If cli_i wants to revoke access rights of block id_u from cli_j, cli_i changes the key under which block id_u is encrypted and informs other clients about this change. Note here, that cli_i and cli_j can share multiple blocks under the same key, thus forming *groups* of shared datablocks. This way, more clients can share only one key for a whole set of blocks (for example client cli_i wishes to share all his blocks with clients cli_j and cli_u).

The algorithm in detail

In detail, an access to the Blurry-ORAM is described in Algorithm 1: First, the client finds the leaf to which the block he is looking for is currently mapped, and remaps the block randomly to a new leaf (lines 2 and 3). The client then downloads the original and its symmetric path, and stores them locally in the list $\mathsf{OriPATH}$ and $\mathsf{SymPATH}$ respectively (lines 4 and 5). The client

[5] Adopting directly the eviction from [11] which also involves reading two paths would unnecessarily degrade the protocol's performance, since we would have to store smaller ORAMs of size $\log(KN)$ in every node.

starts processing the paths, beginning with the original path, which he copies to PATH (line 6) and setting a flag, that indicates that the client is not processing the symmetric path (line 7). The client reads his localstash and the common-stash and copies them in a list $\mathcal{L}$ (lines 8 to 9). For every block in the common-stash, the client tries all his keys, in order to decrypt it. On success, the client adds the decrypted block in $\mathcal{L}$ (lines 10 to 14). For every block in every node in the path, the client tries to decrypt it, using all the keys that he has. If this succeeds, then the client adds the decrypted block to his list $\mathcal{L}$ and marks the block in PATH (lines 15 to 20). Marking the block in PATH is crucial, because this way, the client knows where in the PATH his blocks currently lie, he is therefore able to replace them with other blocks he has access, without moving blocks that belong to other clients.

Once the client has found all the blocks that belong to him, decrypted them and copied them to $\mathcal{L}$, he scans the list and reads the desired block, if the operation was a read or updates the block's contents is the operation was a write (lines 23 to 26). Now, if the operation is a share, then the client creates a new key with which he encrypts the block and sends this new key and the new position to which the block is mapped to client cli_i (lines 27 to 31). Revoking access rights to a block is done in a similar way, with client cli_i changing the encryption key, re-encrypting the block and putting it back, as if the operation was a normal write.

Now that the block has been found, the client continues with the eviction of the blocks present in his local list $\mathcal{L}$: As in classical Path-ORAM, the client tries to move as many blocks as he can closer to the leaf level of the tree. This is done with the function PushBlock, which takes as input a block, a path and a leaf and performs the classical Path-ORAM eviction algorithm. If the block fits in the input path, PushBlock returns 1, otherwise it returns 0. The client first evicts the shared blocks. If a block does not fit in the path, it is added in the common-stash. This procedure is described in lines 35 to 39. As mentioned earlier, the reason for evicting the shared blocks first, is that at this point there is more available space in the PATH and thus the probability that the shared blocks actually fit in the tree is higher. This way, the probability of using the common-stash is reduced. Once the shared blocks are evicted, the client continues with the eviction of all other blocks (lines 40 to 43).

The original path has now been processed and the OriPATH is updated with the blocks from PATH (line 46). Next the client must process the symmetric path (lines 44 to 50). Since the root node of the tree is the only common node between the original and the symmetric path, and during the processing of the original path it has changed, the root of the symmetric path is updated in line 45. The OriPATH is updated with PATH and PATH is emptied. The symmetric path is copied to PATH and is then processed. After this is done, the symmetric path is updated (line 51), both paths and the common-stash are re-randomized (line 52) and finally both paths and the encrypted common-stash are sent back to the server (line 53).

Storing the Position Map

In order to save space, in ORAM constructions that use a position map such as [14,11,12,13], the position map is stored recursively on the server, in smaller ORAMs, $\texttt{ORAM}_1, \cdots, \texttt{ORAM}_k$, where $\texttt{ORAM}_1$ stores the positions of the data and $\texttt{ORAM}_k$ is of constant size. In a multi-client ORAM, allowing data sharing, at least $\texttt{ORAM}_2, \cdots, \texttt{ORAM}_k$ must be accessible by all the clients. But in such a case, any client can infer that a position has been changed in $\texttt{ORAM}_1$, by noticing the changes in $\texttt{ORAM}_2$, thus trivially breaking the access pattern privacy of other clients, regardless if they share their data or not. In order to avoid this potential leakage, in Blurry-ORAM we store the clients' position maps in the following way: Every client stores a position map for his own blocks in a classic Path-ORAM on the server. Similarly, we store a position map in a separate ORAM for every group of shared datablocks to which all the members of the group

Algorithm 1: Blurry-ORAM(OP, id, dat, cli_i, cli_j)

Z: Number of blocks in bucket;
N: Number of client blocks;
λ: Security parameter;
$\text{KeyGen}(1^\lambda)$: Key Generation function;
Enc_{p_k}: Encryption under public key p_k;
DEC_k: Decryption Algorithm, using key k;
$\text{PushBlock}(B, \text{PATH}, x_1)$: Path-ORAM eviction algorithm for block B in path $PATH$ and new leaf position x_1;

1 $\mathcal{L} \leftarrow \emptyset$;
/* Find the leaf to which id is mapped in the position map */;
2 $x_0 = \text{PositionMap}(\text{id})$;
/* Choose new random leaf */;
3 $x_1 \xleftarrow{R} \mathbb{Z}_N$;
4 $\text{OriPATH}[] \leftarrow \text{GetPath}(x_0)$;
5 $\text{SymPATH}[] \leftarrow \text{GetSymmetricPath}(x_0)$;
6 $\text{PATH} \leftarrow \text{OriPATH}$;
7 $\text{ProcessSymmetricPath} = 0$;
8 $\mathcal{L} \leftarrow \text{ReadLocalStash}()$;
9 $\mathcal{C} \leftarrow \text{ReadCommonStash}()$;
10 **for** $i = 1$ **to** $\mathcal{C}.length()$ **do**
11 **for** $j = 1$ **to** KZ **do**
12 **for** k **in** $Keys$ **do**
13 **if** $DEC_k(i) \neq \perp$ **then**
14 $\mathcal{L} \leftarrow \mathcal{L} \cup DEC_k(i)$;
15 **for** $i = 1$ **to** $PATH.length()$ **do**
16 **for** $j = 1$ **to** KZ **do**
17 **for** k **in** $Keys$ **do**
18 **if** $DEC_k(PATH[i][j]) \neq \perp$ **then**
19 $\mathcal{L} \leftarrow \mathcal{L} \cup DEC_k(PATH[i][j])$;
20 $\text{Mark}(PATH[i][j])$;
21 **for** i **in** $\mathcal{L}$ **do**
22 **if** $i.\text{id} == \text{id}$ **then**
23 **if** $OP == "R"$ **then**
24 $\text{RetBlock} = i$;
25 **else if** $OP == "W"$ **then**
26 $i.data = \text{dat}$;
27 **else if** $OP == "Share"$ **then**
28 $k \leftarrow \text{KeyGen}(1^\lambda)$;
29 $i.data = \text{Enc}_k(\text{dat})$;
30 Send x_1 to cli_j;
31 Send k to cli_j;
32 **else if** $OP == "Revoke"$ **then**
33 $k \leftarrow \text{KeyGen}(1^\lambda)$;
34 $i.data = \text{Enc}_k(\text{dat})$;
/* First evict the shared blocks */;
35 **for** i **in** $\mathcal{L}$ **do**
36 **if** i **is** *shared* **then**
37 **if** $\text{PushBlock}(i, \text{PATH}, x_1) == 0$ **then**
/* Shared block did not fit into the tree */;
38 $\mathcal{C} \leftarrow i$;
39 Remove i from $\mathcal{L}$;
/* Now evict the remaining blocks */;
40 **for** i **in** $\mathcal{L}$ **do**
41 **if** $\text{PushBlock}(i, \text{PATH}, x_1) == 0$ **then**
/* Shared block did not fit into the tree */;
42 $\mathcal{C} \leftarrow i$;
43 Remove i from $\mathcal{L}$;
44 **if** $\text{ProcessSymmetricPath} == 0$ **then**
45 $\text{SymPATH}[0] \leftarrow \text{PATH}[0]$;
46 $\text{OriPATH} \leftarrow \text{PATH}$;
47 $\text{PATH} \leftarrow \emptyset$;
48 $\text{PATH} \leftarrow \text{SymPATH}$;
49 $\text{ProcessSymmetricPath} = 1$;
50 Repeat Steps 8 to 43;
51 $\text{SymPATH} \leftarrow \text{PATH}$;
52 $\text{Rerandomize}(\text{OriPATH}, \text{SymPATH}, \mathcal{C})$;
53 $\text{Upload}(\text{OriPATH}, \text{SymPATH}, \mathcal{C})$;
54 **return** *RetBlock*

have access. Of course, once such group position maps appear, in order to hide from the server whether a shared data-block or not shared is accessed, any client that accesses a block of his, must download one path in every position map ORAM that he has access to.

4 Analysis

4.1 Stash Size

As mentioned above, it is important that the commonstash remains very small. For this reason, the client first evicts all shared blocks found in the downloaded paths. Clearly, however, this does not guarantee that shared blocks never need to be stored outside the tree. Suppose that in a Blurry-ORAM with N leaves, serving K clients, each client shares at most m blocks. Observe that, since every client has a fixed amount of buckets that he can use in every node of the Blurry-ORAM, we could simulate every client's data as being stored in a single Path-ORAM, in which the client stores $N + m$ real blocks in a tree with N leaves. Since during eviction we let the shared items take the place of real items that belong to the client and we first push these shared items into the structure, in essence we treat these m blocks as the real blocks and all other blocks as fakes. Thus, in order to examine the commonstash size, we can simulate a Blurry-ORAM by a Path-ORAM, in which a client stores m real blocks in a tree with N leaves. Based on the proof of [14] we can estimate the probability of using a stash of size $O(\log \log N)$ for $m = \log^2 N$, by following the same argumentation as in the classical Path-ORAM, adjusting the number of leaves of the tree. These ideas are summarized in the following lemmata:

Lemma 1. *For a data request sequence α in a Blurry-ORAM with K clients, each having N blocks and sharing $O(\log^2 N)$ blocks, with $Z = 5$ buckets per client per node, that uses a commonstash of size R, the probability that the size of the commonstash during a data request sequence α exceeds R during one of the requests is bounded by*

$$\mathsf{Prob}[\mathsf{st}(\mathsf{Blurry} - \mathsf{ORAM}_L^Z)[\alpha] > R] \leq 1.002 \cdot (0.5006)^R$$

We can further use the above lemma in order to show that, in case $O(\log^2 N)$ blocks are shared, the commonstash does not grow bigger than $O(\log \log N)$. The proof follows the one given in [14] and is thus omitted.

Proposition 1 *For a Blurry-ORAM of height $L = \log N$, $Z = 5$ buckets per client, per node, $O(\log^2 N)$ blocks shared by every client, and a data request sequence α, of length $N + \log^2 N$, the probability that the commonstash st exceeds the size R, after a series of load/store operations that correspond to α, is at most*

$$\mathsf{Prob}[\mathsf{st}(\mathsf{Blurry} - \mathsf{ORAM}_L^Z)[\alpha] > R] = 14 \cdot (0.6002)^R$$

The above lemmata show that, as long as the amount of the shared elements is in $O(\log^2 N)$, the commonstash will remain small. We observed this behaviour also during our simulations, where the commonstash was very rarely used.

4.2 Time and Space requirements

Based on the observations made earlier in this section, we can now analyze the time and space requirements of our protocol. A client that participates in $O(\log N)$ groups, needs $O(\log N)$ space for the keys and $O(\log N)$ for the position maps (given the recursive position map storage). The client also needs to store his private stash, which follows the bounds provided in

Proposition 1, and is thus limited to $O(\log N)$. Now, each time the client performs a data request, the client downloads two paths of size $O(\log N)$, and the commonstash, which is in worst case of size $K \log\log N \in O(\log N)$. Thus, the amount of space needed during protocol execution for the client is $O(\log N)$.

As far as the computational complexity of the client is concerned, recall that the client has to iterate on his $O(\log N)$ keys for each of the downloaded blocks found in the paths, thus the computational complexity is $O(\log^2 N)$. For the recursive position map accesses, the client has to dedicate again $O(\log^2 N)$ time. Thus, the total computational complexity of the client does not deviate from classical Path-ORAM and is thus $O(\log^2 N)$.

5 Security

In this section we examine the security achieved by our construction.

Proposition 2 *Blurry-ORAM achieves Client-to-Server Privacy.*

Proof. (sketch) Recall that the position map and stash of every client are stored in exactly the same way as in the classical Path-ORAM. The commonstash is of fixed size and re-randomized every time a data request is performed by a client. Thus, it is easy to see, that the security of Blurry-ORAM against the server can be reduced to the security of Path-ORAM. □

Proving that our construction achieves access pattern privacy against clients is more involved and is done by showing that Blurry-ORAM satisfies Definition 7. To do this, however, we must first make sure, that the commonstash is empty. Indeed recalling Lemma 1 we see that the size of the commonstash is very small with high probability. We have also noticed this behaviour experimentally, as with setting the appropriate parameters, the commonstash was in all our experiments almost always empty. Thus we can state the following proposition:

Proposition 3 *If the commonstash is empty, Blurry-ORAM achieves Client-To-Client Privacy.*

Proof (of Proposition 3). Suppose that there exists a PPT adversary $\mathcal{A}$, that wins the IND-CQA game on a Blurry-ORAM with non-negligible advantage δ. We show how to construct a PPT algorithm $\mathcal{B}$, that breaks the semantic security of the encryption scheme used in Blurry-ORAM. To do this, suppose that $\mathcal{B}$ plays the IND-CPA game against a challenger $\mathcal{C}$. Let $\mathcal{E} = (\mathsf{KeyGen}(n), \mathsf{Enc}_{pk}(\cdot), \mathsf{Dec}_{sk}(\cdot))$ be a public key encryption scheme, used in Blurry-ORAM, with $(pk, sk) \leftarrow \mathsf{KeyGen}(n)$ the private/public key pair produced for the security parameter n. The challenger $\mathcal{C}$ runs $\mathsf{KeyGen}(n)$, gets (cpk, cs) and sends cpk to $\mathcal{B}$. $\mathcal{B}$ simulates the Blurry-ORAM server and runs the Init algorithm in order to create the private/public key pairs for all the clients present on the Blurry-ORAM, except for the adversary $\mathcal{A}$. For one of the clients that $\mathcal{B}$ simulates, say client cli_i, $\mathcal{B}$ uses the public key cpk that he got from his challenger $\mathcal{C}$, instead of creating a fresh one. $\mathcal{B}$ keeps the Blurry-ORAM in plaintext, except for the blocks that $\mathcal{A}$ has access to and are not shared by him, which $\mathcal{B}$ keeps encrypted on the Blurry-ORAM structure.

Whenever $\mathcal{B}$ gets a data request from $\mathcal{A}$, for a block that belongs to $\mathcal{A}$, $\mathcal{B}$ does the following:

1. $\mathcal{B}$ finds the path P that $\mathcal{A}$ asks for. The encrypted blocks found in P, $\mathcal{B}$ leaves as they are and every block in plain, $\mathcal{B}$ encrypts using each client's encryption key.
2. $\mathcal{B}$ sends the encrypted path to $\mathcal{A}$, who runs his eviction algorithm and sends the updated path back to $\mathcal{B}$.

3. $\mathcal{B}$ now tries to decrypt all the blocks in the path he received, and updates the corresponding blocks in the plain Blurry-ORAM version. This way any possible changes that $\mathcal{A}$ made on shared blocks, are considered. The blocks that could not be decrypted belong to either cli_i or $\mathcal{A}$. Since the blocks accessible only by cli_i cannot have been moved in the path during $\mathcal{A}$'s eviction, $\mathcal{B}$ replaces these encryptions with the plaintext blocks of cli_i from the path P.
4. $\mathcal{B}$ discards the version of the path on his Blurry-ORAM construction and sets in its place the newly updated path, which contains the blocks of all clients in plain (including the ones that $\mathcal{A}$ shares with other clients), except the blocks that only $\mathcal{A}$ has access to, which are encrypted.

Whenever $\mathcal{B}$ gets an "oracle" data request from $\mathcal{A}$ on data blocks that do not belong to $\mathcal{A}$, $\mathcal{B}$ runs the Blurry-ORAM protocol and produces an unencrypted view as follows:

1. $\mathcal{B}$ adds to the unencrypted view, the appropriate path (which we will call the downloaded path).
2. $\mathcal{B}$ updates the appropriate position map and runs the eviction algorihm.
3. $\mathcal{B}$ adds the updated path (which we will call the uploaded path) to the unencrypted view.

After the eviction is done, $\mathcal{B}$ creates the view, by encrypting it as follows: For every element found in the view, $\mathcal{B}$ (who knows to which client each block belongs) encrypts it, using the appropriate client's public key. Finally, $\mathcal{B}$ adds to the downloaded path, $\mathcal{A}$'s encrypted blocks that were found on the path, and to the uploaded path, a re-randomization of these encrypted blocks. The result of this operation is a view that $\mathcal{A}$ can process.

Note that in forming the view, only the downloaded and uploaded paths are used. This is because, we have assumed that no common-stash is used, that the position maps for all the blocks that are not shared or belong to $\mathcal{A}$ are kept encrypted on the server and that all other stashes are locally stored by every client.

In the challenge phase, $\mathcal{B}$ receives from $\mathcal{A}$ two data requests $(\mathsf{DR}_0, \mathsf{DR}_1)$. $\mathcal{B}$ copies the Blurry-ORAM, including all the position maps and stashes of the clients, in a Blurry-ORAM-COPY and runs both of the data requests, on each Blurry-ORAM (for example DR_0 on Blurry-ORAM and DR_1 on Blurry-ORAM-COPY) as earlier in the pre-challenge phase. $\mathcal{B}$ chooses a random bit b^* and selects the data request DR_{b^*} and the Blurry-ORAM yielded thereof (either the Blurry-ORAM or the Blurry-ORAM-COPY). In the two unencrypted views there will be at least one position in the data-blocks of client cli_i, where the unencrypted views will differ. $\mathcal{B}$ finds the first position where the two plain views differ, say position j of the views, and picks the two corresponding (plain) data, m_j^0 corresponding to DR_0 and m_j^1 corresponding to DR_1. Now $\mathcal{B}$ forms the challenge pair (m_j^0, m_j^1), that he sends to $\mathcal{C}$. $\mathcal{C}$ chooses a bit b and sends to $\mathcal{B}$ the encryption of m_j^b (under the public key cpk). $\mathcal{B}$ now encrypts the plain view corresponding to DR_{b^*}, in the same way he did in the pre-challenge phase, with the only difference, that he implants the encryption he was given from $\mathcal{C}$, at position j. $\mathcal{B}$ hands over to $\mathcal{A}$ the resulting view.

The post-challenge phase continues exactly as the pre-challenge phase with data requests issued from $\mathcal{A}$, which however cannot include share and revoke operations on the blocks requested during the challenge phase. At the end of this phase, $\mathcal{A}$ outputs a bit b' and $\mathcal{B}$ outputs b'.

If $b = b^*$, then $\mathcal{A}$ gets a well formed view and thus by assumption, $\mathcal{A}$ wins with non-negligible advantage δ. Suppose that if $b \neq b^*$, $\mathcal{A}$ outputs 0 with some probability α and outputs 1 with

probability $1-\alpha$. Then the probability of $\mathcal{B}$ winning the IND-CPA game is

$$\begin{aligned}&\mathsf{Prob}[\mathcal{B} \text{ wins IND-CPA}] = \\ &\mathsf{Prob}[\mathcal{A} \text{ wins IND-CQA} \,|\, b = b^*] + \mathsf{Prob}[\mathcal{A} \text{ wins IND-CQA} \,|\, b \neq b^*] = \\ &\frac{1}{4}\left(\left(\frac{1}{2}+\delta\right) + (1-\alpha) + \alpha + \left(\frac{1}{2}+\delta\right)\right) = \frac{1}{2} + \frac{\delta}{2}\end{aligned}$$

Thus $\mathcal{B}$ breaks the semantic security of the encryption scheme with advantage $\delta/2$, which is non-negligible and thus a contradiction.

6 Conclusion

In this work, we constructed Blurry-ORAM: an ORAM architecture, which allows multiple clients to store their encrypted data on a remote server and share any part of it with each other. To our knowledge, we are the first to propose an ORAM architecture that guarantees access pattern privacy not only against the server, but also against other semi-honest, non-colluding clients, without demanding from the clients to be constantly online. At the same time, since our construction is based on Path-ORAM, it inherits from it, not only its strong security guarantees, but also Path-ORAM's efficiency, thus we achieve $O(\log^2 N)$ computational complexity on the side of the client, while the client needs to dedicate $O(\log N)$ space. As future work, we want to examine how collusion between the clients, or between the clients and the server affects the security of our construction, and how potential leakage in such case, can be treated. Another interesting direction would be, to combine the ideas from [1] and [9], so that access pattern anonymity can be also achieved.

References

1. Michael Backes, Amir Herzberg, Aniket Kate, and Ivan Pryvalov. Anonymous RAM. In Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows, editors, *Computer Security - ESORICS 2016 - 21st European Symposium on Research in Computer Security, Heraklion, Greece, September 26-30, 2016, Proceedings, Part I*, volume 9878 of *Lecture Notes in Computer Science*, pages 344–362. Springer, 2016.
2. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC 2016-A, Part II*, pages 175–204.
3. Martin Franz, Peter Williams, Bogdan Carbunar, Stefan Katzenbeisser, Andreas Peter, Radu Sion, and Miroslava Sotáková. Oblivious outsourced storage with delegation. In George Danezis, editor, *Financial Cryptography and Data Security - 15th International Conference, FC 2011, Gros Islet, St. Lucia, February 28 - March 4, 2011, Revised Selected Papers*, volume 7035 of *Lecture Notes in Computer Science*, pages 127–140. Springer, 2011.
4. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
5. Philippe Golle, Markus Jakobsson, Ari Juels, and Paul F. Syverson. Universal re-encryption for mixnets. In Tatsuaki Okamoto, editor, *Topics in Cryptology - CT-RSA 2004, The Cryptographers' Track at the RSA Conference 2004, San Francisco, CA, USA, February 23-27, 2004, Proceedings*, volume 2964 of *Lecture Notes in Computer Science*, pages 163–178. Springer, 2004.
6. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious RAM simulation. In Yuval Rabani, editor, *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*, pages 157–167. SIAM, 2012.

7. Nikolaos P. Karvelas, Andreas Peter, Stefan Katzenbeisser, Erik Tews, and Kay Hamacher. Privacy-preserving whole genome sequence processing through proxy-aided ORAM. In Gail-Joon Ahn and Anupam Datta, editors, *Proceedings of the 13th Workshop on Privacy in the Electronic Society, WPES 2014, Scottsdale, AZ, USA, November 3, 2014*, pages 1–10. ACM, 2014.
8. Jacob R. Lorch, James W. Mickens, Bryan Parno, Mariana Raykova, and Joshua Schiffman. Toward practical private access to data centers via parallel ORAM. *IACR Cryptology ePrint Archive*, 2012:133, 2012.
9. Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 341–358. IEEE Computer Society, 2015.
10. Travis Mayberry, Erik-Oliver Blass, and Guevara Noubir. Multi-user oblivious RAM secure against malicious servers. *IACR Cryptology ePrint Archive*, 2015.
11. Elaine Shi, T.-H. Hubert Chan, Emil Stefanov, and Mingfei Li. Oblivious RAM with o((logn)3) worst-case cost. In Dong Hoon Lee and Xiaoyun Wang, editors, *Advances in Cryptology - ASIACRYPT 2011 - 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings*, volume 7073 of *Lecture Notes in Computer Science*, pages 197–214. Springer, 2011.
12. Emil Stefanov and Elaine Shi. Oblivistore: High performance oblivious distributed cloud data store. In *20th Annual Network and Distributed System Security Symposium, NDSS 2013, San Diego, California, USA, February 24-27, 2013*. The Internet Society, 2013.
13. Emil Stefanov, Elaine Shi, and Dawn Xiaodong Song. Towards practical oblivious RAM. In *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012.
14. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*, pages 299–310. ACM, 2013.
15. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 139–148. ACM, 2008.
16. Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: a parallel oblivious file system. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*, pages 977–988. ACM, 2012.
17. Jinsheng Zhang, Wensheng Zhang, and Qiao Daji. A multi-user oblivious ram for outsourced data. http://lib.dr.iastate.edu/cs_techreports/262/, 2014.