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 he 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 [9] the authors proposed an architecture, which guarantees access pattern privacy for one single investigator against honest but curious servers with a construction that is based on the hierarchical ORAM of Williams et al. [17]. Besides heavy computational and communication costs, the solution suffers from the problem that it does not deal with the potential privacy leakage that can occur in the presence of multiple investigators, which is very realistic in the scenario of genomic databases. Note that this does not contradict the security requirements of an ORAM, as that solution only provides 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 [16], 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 [19,5,8,11,12,1] 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 [18,10,3]. Here, 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 [5] 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 [6] and thus suffers from heavy communication complexity. Equally important is the fact that it requires the data owner to be constantly online, thus restricting the applicability of the solution. The construction in [8] avoids this drawback, but suffers from high storage requirements on the client side. In a recent work of Maffei et al. [11], a multi-client ORAM is proposed, which is based on Path-ORAM [16] 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: 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 other 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. To this end, our solution is directly applicable to the case of GWAS: multiple clients store their encrypted genomic data in a Blurry-ORAM and give partial access to multiple investigators. The investigators can then access only specific parts of the clients' sequenced genome, without leaking

any information about the parts they are interested, to any other investigators. Indeed, our experiments showed that our construction is efficient, as retrieving one block in a database where 100 clients store their whole genome in the form of Single Nucleotide Polymorphisms (SNPs) and they share parts of it with 10 investigators, can be done on average in 14.94 s seconds.

1.1 Organization of the paper

The rest of the paper is organized as follows: in Section 2, we describe the functionality we want to achieve, and define the security of the protocol. In Section 3, we recall the Path-ORAM protocol, and we describe our architecture. In Section 4, we examine the stash occupancy by our protocol, and analyze the time and space requirements of our solution. In Section 5, we analyze the security of our construction. In Section 6, we describe how our architecture can be used for GWAS. In Section 7, we present our experimental results, and we conclude in Section 8 with final remarks and future work.

Solution	Commun.Complexity	A.P.H against Server	A.P.H against Clients	Client Anonymity	Data Sharing
[5]	$O(\sqrt{N}\log^2 N)$ (amort.)	✓	×	×	✓
[8]	$O(\log^2 N)$	✓	×	×	✓
[19]	$O(\log^3 N \log\log N)$	✓	×	×	×
[12]	$O(\log^2 N)$ to $O(\log^5 N)$	✓	×	×	✓
[11]	$O(G\log^2 N)$	✓	×	✓	✓
[1](linear)	$O(K\log^2 N)$	✓	×	✓	×
[1](polylog)	$O(\log^2(KN))$	✓	×	✓	×
This work	$O(K\log^2 N\log(\log N))$	✓	×	✓	✓

Table 1: Comparison of multi-client ORAM solutions. A.P.H stands for Access Pattern Hiding. 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 do not 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 belongs to 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, takes as input a security parameter λ and 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, $\mathsf{enc_key}$, $\mathsf{dec_key}$): 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 clients cli_i and cli_j, the purpose of this operation is to disallow cli_j from further being able to access (read, write, share or revoke) block id. 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 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.

Definition 1 *A data request is a tuple of the form* $(\mathsf{op}, \mathsf{id}, \mathsf{dat}, \mathsf{cli}_j)$ *where* op *is the operation to be performed, i.e.,* $\mathsf{op} \in \{\mathsf{Read}, \mathsf{Write}, \mathsf{Share}, \mathsf{Revoke}\}$*,* 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* l *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 view (or access pattern) induced by* X*, everything that the server sees during the execution of* X *(ofter referred to also, as the protocol's transcript).*

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.*

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 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 [16]:

Definition 5 (Client-to-Server Privacy) *We say that a multi-client ORAM protocol provides client-to-server privacy, if any two views of a client induced by data request sequences of the same length, are computationally indistinguishable by the server.*

Hiding the access patterns against the server is already a non-trivial problem, so it is no surprise that 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 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 attacked client and he thus can decrypt. Therefore, traditional proof techniques used in the simple client-server model (showing, for example indistinguishability of these views) cannot be adapted directly to the multi-client case, due to the fact that the adversary can potentially read parts of the view. In our security model, which is based on the IND-CPA paradigm and detailed in Definition 6, we let the adversary have "oracle" access to any operations on blocks he does not have access to. In the challenge phase, the adversary issues two data request sequences on blocks that he does not have access to (ultimately, we are interested in breaking the access pattern privacy for blocks that the adversary cannot see), and Share and Revoke operations can also be included, as long as they do not result in the adversary gaining or losing access to blocks, after their execution. Observe that the notion described here, resembles exactly the adversary's ability to read parts of the views that do not belong to him, and can thus help him to (potentially) gain information about blocks he does not have access to. Further, remark that the problem we are dealing with has little to do with client anonymity as opposed to other works in the literature like [1,12,11], where the identity of the clients accessing the oram is protected. Here we model and want to minimize the privacy leakage potentially appearing between clients who *share parts of their data*, about the parts of the data they *do not* share. Having these in mind, we can now define access pattern privacy in a multi-client ORAM construction as follows:

Definition 6 (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 other 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.*

3 Blurry-ORAM Construction

3.1 Review of Path-ORAM

Our starting point is the Path-ORAM construction of Stefanov et al. [16], where a client stores N blocks of data 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 with fake blocks. Real and fake blocks are encrypted under a semantically 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, 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; the client is guaranteed to find the desired block in one of the nodes along this path. The client then 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

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 request sequences (drs_c^0, drs_c^1) and sends them to the challenger. The data request sequences may include read or write requests on blocks that the adversary does not have access to, and share or revoke operations that do not result in the adversary gaining or losing access to blocks.
2. The challenger chooses randomly a bit b, runs drs_c^b and returns the view to the adversary.

Post-challenge phase: Similar to the pre-challenge phase, with the restriction that no share or revoke operation on the blocks included in the challenge phase are allowed.

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.

and higher levels, until a node with enough space is found. 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 which case, a small auxiliary storage, called a *stash*, is used to store the element instead. 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. 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, but as shown in [13] can be recursively stored in smaller Path-ORAMs, until a Path-ORAM of constant size is reached.

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 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 the client 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, as can be seen in Figure 1. 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 [7]). Further, we employ a homomorphic encryption scheme, which is IND-CPA and IK-CPA secure. The latter notion is referred to as 'key privacy', introduced in [2] and is modeled similarly to the IND-CPA game, but in the challenge phase, the adversary sends a message and two keys to the challenger. The challenger then, chooses one of the two keys, and encrypts the message under this key and the adversary wins, if he can tell under which key the ciphertext presented to him was encrypted. This way, the property is achieved that all blocks (real and fake) belonging to a client, are indistinguishable for anyone but the client who can decrypt them. Note here, that the ElGamal encryption scheme (which we use in the implementation of our construction) has all the required properties, i.e. it is homomorphic, and is both IND-CPA and IK-CPA secure. 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. The commonstash is initially filled with encrypted fake items, so that the server cannot observe if shared blocks have been added or removed, and remains 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: once client cli_i notices that after a client's cli_j access, a shared datablock has been moved on the commonstash, cli_i immediately knows that in the last accessed path, all blocks of cli_j are real. Therefore, we must make sure that the commonstash remains as small as possible. We achieve this by changing the way the Read operation (for a block id) is performed: The client first finds the leaf i, to which id is mapped. The client then downloads the commonstash and *two* paths[3]: 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 by having the client iterate on his keys, and checking for every block, if he can decrypt it – as we detail however in Section 7.2, this can be done more efficiently, by using the properties of homomorphic schemes. By construction, one of these blocks is guaranteed to be the block that the client 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 shared blocks, trying to store them as deep down in either of the

[3] Adopting directly the eviction from [13], 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.

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 in a similar manner those blocks that are not shared. If there is not enough space in the paths, the localstash is used. The client then fills up the remaining placeholders with fake elements, 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 during our experiments (cf. Section 7) 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 (by means of Read operation, which changes also the block's path assignment), re-encrypts the block with a new key and uploads the block to the server. Finally, cli_i hands over to cli_j the *new key* and *the new index of the 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 (again by reading block id_u, and thus changing its path assignment) and informs other clients about the change of key and of path assignment. 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.

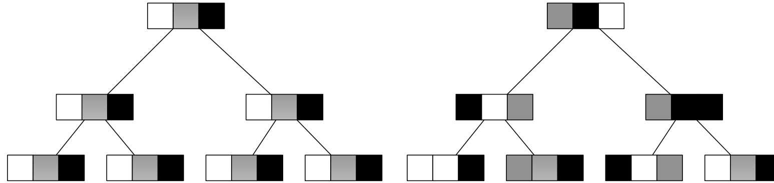

Fig. 1. Path-ORAMs of three clients, concatenated together, forming the initialization of Blurry-ORAM, on the left (blocks of different clients have different colors), and a Blurry-ORAM state after some accesses (the 'blurred' version of the multiple Path-ORAMs), on the right.

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 3 and 4). The client then downloads the original and its symmetric path, and stores them locally in the list $\mathsf{OriPATH}$ and $\mathsf{SymPATH}$ respectively (lines 5 and 6). The client starts processing the paths, beginning with the original path, which he copies to PATH (line 7) and setting a flag, that indicates that the client is not processing the symmetric path (line 8). The client reads his localstash and the commonstash and copies them in a list $\mathcal{L}$ (lines 9 to 10). For every block in the commonstash, the client tries all his keys, in order to decrypt it[4]. On success, the client adds the decrypted block in $\mathcal{L}$ (lines 11 to 14). For every block in every node

[4] We assume here the usage of an encryption scheme that returns a special symbol "⊥" if the encryption is done with the wrong key. However, using properties of homomorphic schemes one can avoid the expensive decryption step, and achieve simpler and more efficient ways of providing this functionality, as done in our implementation, cf. Section 7.2.

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 commonstash. 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 commonstash 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 commonstash are re-randomized (line 52) and finally both paths and the encrypted commonstash 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 [16,13,14,15], the position map is stored recursively on the server, in smaller ORAMs, $\mathtt{ORAM}_1, \ldots, \mathtt{ORAM}_k$, where $\mathtt{ORAM}_1$ stores the positions of the data and $\mathtt{ORAM}_k$ is of constant size. In a multi-client ORAM allowing data sharing, at least $\mathtt{ORAM}_2, \ldots, \mathtt{ORAM}_k$ must be accessible by all clients. But in such a case, any client can infer that a position has been changed in $\mathtt{ORAM}_1$, by noticing the changes in $\mathtt{ORAM}_2$, 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 have access. In order further to prevent the server from knowing whether a client is asking for a shared block (which the server could see by observing the position maps being accessed), all clients must access all the position maps for every access they make, even if they cannot decrypt blocks from certain position maps. In this case, the client simply downloads a path, re-encrypts it and uploads it.

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

```
   Z: Number of blocks in bucket;
   N: Number of client's blocks;
   λ: Security parameter;
   KeyGen(1^λ): Key Generation function;
   Enc_{p_k}: Encryption under public key p_k;
   Dec_k: Decryption Algorithm, using key k;
   PushBlock(B, PATH, x_1): Path-ORAM eviction
    algorithm for block
1  B in path PATH and new leaf position x_1;
2  L ← ∅;
   /* Find the leaf to which id is mapped in
   the position map                          */;
3  x_0 = PositionMap(id);
   /* Choose new random leaf                 */;
4  x_1 ←R Z_N ;
5  OriPATH[] ← GetPath(x_0);
6  SymPATH[] ← GetSymmetricPath(x_0);
7  PATH ← OriPATH;
8  ProcessSymmetricPath = 0;
9  L ← ReadLocalStash();
10 C ← ReadCommonStash();
11 for i = 1 to C.length() do
12     for k in Keys do
13         if Dec_k(i) ≠ ⊥ then
14             L ← L ∪ 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]) ≠ ⊥ then
19                 L ← L ∪ Dec_k(PATH[i][j]);
20                 Mark(PATH[i][j]);
21 for i in L do
22     if i.id == id then
23         if OP == "R" then
24             RetBlock = i;
25         else if OP == "W" then
26             i.data = dat;
27         else if OP == "Share" then
28             k ← KeyGen(1^λ) ;
29             i.data = Enc_k(dat);
30             Send x_1 to cli_j;
31             Send k to cli_j;
32         else if OP == "Revoke" then
33             k ← KeyGen(1^λ) ;
34             i.data = Enc_k(dat);
   /* First evict the shared blocks          */;
35 for i in L do
36     if i is shared then
37         if PushBlock(i, PATH, x_1) == 0 then
               /* Shared block did not fit into
               the tree                      */;
38             C ← i;
39         Remove i from L;
   /* Now evict the remaining blocks         */;
40 for i in L do
41     if PushBlock(i, PATH, x_1) == 0 then
           /* Shared block did not fit into
           the tree                          */;
42         C ← i;
43     Remove i from L;
44 if ProcessSymmetricPath == 0 then
45     SymPATH[0] ← PATH[0];
46     OriPATH ← PATH;
47     PATH ← ∅;
48     PATH ← SymPATH;
49     ProcessSymmetricPath = 1;
50     Repeat Steps 9 to 43;
51 SymPATH ← PATH;
52 Rerandomize(OriPATH, SymPATH, C);
53 Upload(OriPATH, SymPATH, C);
54 return RetBlock
```

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, and surely the greater the amount of shared blocks, the greater the probability that the commonstash will be used. We thus have to place a restriction on the amount of shared blocks. Suppose that in a Blurry-ORAM with N leaves, and K clients, each client shares at most m blocks. 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 [16] 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 larger than $O(\log\log N)$. The proof follows the one given in [16] 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 never used (cf. Section 7).

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 $O(\log N)$. Each time the client performs a data request, he downloads two paths of size $O(\log N)$, and the commonstash, which is in the worst case of size $O(\log\log N) \in O(\log N)$. 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 $O(\log^2 N)$. Note that since we have restricted the amount of shared items to $O(\log N)$ per client, there is a total $O(K \log N) \in O(\log N)$ position maps from each of which a client will read and rerandomize a path during a query.

Suppose that a client shares $N-1$ blocks of his with K clients. Instead of holding a different key for every group, the client can share all the common blocks with all the clients, using only one key, and create smaller position maps only for the blocks that are not shared with all of the clients. Thus, each smaller position map will not exceed a size of $O(\log N)$, which means that for every query, the client will have to dedicate $O(\log^2 N \log(\log N))$ time for the recursive position map accesses.

5 Security

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 6. 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 *Assuming that the commonstash is empty, Blurry-ORAM achieves Client-To-Client privacy, if the used encryption scheme $\mathcal{E}$ is both IND-CPA and IK-CPA secure.*

Proof. 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 either the semantic security or the key indistinguishability property of the encryption scheme used in Blurry-ORAM. To do this, suppose that $\mathcal{B}$ plays the IK-CPA and the IND-CPA game against a challenger $\mathcal{C}$, and wins if he wins in any one of them. Let $\mathcal{E} = (\mathsf{KeyGen}(n), \mathsf{Enc}_{pk}(\cdot)$,
$\mathsf{Dec}_{sk}(\cdot))$ be the public key encryption scheme used in Blurry-ORAM, with $(pk, sk) \leftarrow \mathsf{KeyGen}(n)$ a private/public key pair produced for the security parameter n. $\mathcal{C}$ 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.

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 algorithm.
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 $\mathcal{A}$, a tuple of two data request sequences (DRS_0, DRS_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 request sequences, on each Blurry-ORAM (for example DRS_0 on Blurry-ORAM and DRS_1 on Blurry-ORAM-COPY) as earlier in the pre-challenge phase. $\mathcal{B}$ first chooses a random bit $\tilde{b}$; if $\tilde{b} = 0$, $\mathcal{B}$ will play the IND-CPA game, else he plays the IK-CPA game. Now $\mathcal{B}$ chooses a random bit b^* and selects the data request sequence DRS_{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, picks the two corresponding (plain) data, m_j^0 corresponding to DRS_0 and m_j^1 corresponding to DRS_1, and forms two pairs of challenges: the IND-CPA challenge, (m_j^0, m_j^1) and the IK-CPA challenge, $(m_j^{b^*}, cpk, pk)$, where cpk is client's cli_i public key and pk is the public key of another client. $\mathcal{B}$ sends the two challenges to $\mathcal{C}$, who chooses a bit b. If $b = 0$, then $\mathcal{C}$ responds to the IND-CPA challenge with encrypting m_j^0 under cpk and to the IK-CPA by encrypting $m_j^{b^*}$ under pk. Otherwise, $\mathcal{C}$ encrypts m_j^1 (under cpk) and $m_j^{b^*}$, under cpk. $\mathcal{C}$ then sends back to $\mathcal{B}$, $(\mathcal{E}_0, \mathcal{E}_1)$ where $\mathcal{E}_0$ is $\mathcal{C}$'s response to the IND-CPA challenge, and $\mathcal{E}_1$ is $\mathcal{C}$'s response to the IK-CPA challenge.

$\mathcal{B}$ now encrypts the plain view corresponding to DRS_{b^*}, in the same way he did in the pre-challenge phase, with the only difference, that he implants one of the encryptions he was given from $\mathcal{C}$, at position j, based on his former choice of $\tilde{b}$: if $\mathcal{B}$ chose to play against the encryption scheme's IND-CPA security, he implants $\mathcal{E}_0$; otherwise he implants $\mathcal{E}_1$. $\mathcal{B}$ 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'. $\mathcal{B}$ then outputs (b', d) if he had chosen to play against IND-CPA, or $(d, \neg(b^* \oplus b'))$, if he had chosen to play against IK-CPA, where d is a random bit.

Observe now, that if $\mathcal{B}$ chooses to play against IND-CPA, and if $b = b^*$, then $\mathcal{A}$ gets a well formed view and thus by assumption, $\mathcal{A}$ wins with non-negligible advantage δ. In a similar way, if $\mathcal{B}$ plays against IK-CPA and $b = 1$, $\mathcal{A}$ gets a well formed view and thus by assumption, $\mathcal{A}$ wins with non-negligible advantage δ. Suppose now, that every time $\mathcal{A}$ gets a malformed view, he outputs 0 with some probability α and outpus 1 with probability $1 - \alpha$. Then $\mathcal{B}$'s total winning probability (i.e. breaking IND-CPA or IK-CPA) is

$$\mathsf{Prob}\left[{}^{\mathcal{B}\text{wins}}_{\text{IND-CPA}}\right] + \mathsf{Prob}\left[{}^{\mathcal{B}\text{wins}}_{\text{IK-CPA}}\right] =$$

$$\frac{1}{8}\left(\frac{1}{2} + \delta + \alpha + (1-\alpha) + \frac{1}{2} + \delta + (1-\alpha) + \frac{1}{2} + \delta + \alpha + \frac{1}{2} + \delta\right) = \frac{1}{2} + \frac{\delta}{2}$$

which means that with non-negligible advantage, $\mathcal{B}$ breaks either the semantic security or the key indistinguishability property of the encryprion scheme, which is a contradiction.

6 Application to Genomic Studies

We return to our motivating scenario of genomic studies, and we give a brief overview of the techniques used to store and query genomic data. A sequenced human genome is a set of approximately 3,3 billion characters, out of which only around 1% seem to be relevant. This specific part of the genome is typically stored in the form of Single Nucleotide Polymorphisms (SNPs), which are the positions in the sequenced genome of an individual which differ from what is known as the reference genome – and what is believed to be a representative example of a human genome.

Once a donor's SNPs have been determined, they are stored in biobanks, alongside those of other individuals, and can be used for various tests. These tests range from simple paternity and ancestry tests, to complex genome wide association studies. In the latter, the sequenced genome of multiple DNA donors is examined, and associations between specific parts of the human genome and various diseases are made. To do this, a biostatistician (in the following called an investigator) examines parts of the genome of multiple DNA samples that are suspected to be associated with a specific disease. Further, the investigator consults a table indicating whether if a DNA donor suffers from the disease or not, and thus can extract the probability that a particular DNA region (or SNP) is correlated with the disease. As pointed out in [9], estimating this probability in a privacy preserving way, can be done efficiently, using Secure Two Party Computation (STC) protocols. However, privacy leakage can occur, while the investigator retrieves the required SNPs (since the biobank can link parts of the genome to various diseases). Dealing with this leakage against the biobank is the main contribution of [9]. However, the authors in that solution, do not deal with the privacy leakage that may occur if multiple investigators are allowed to perform tests. To this end, our solution can be applied: By extending the highly efficient Path-ORAM construction, to a multi-client setting (where now a client might be a DNA donor or an investigator) so that information leakage between the participating entities is eliminated, we can substitute the ORAM used in [9] with Blurry-ORAM. This way, not only do we improve the computational complexity of the construction, we also improve the security guarantees. Observe here, that changing the ORAM construction of [9] does not affect either the security or the performance of the STC protocol used to evaluate the

correlation probability. Since retrieving the relevant genome parts in a privacy preserving way is the bottleneck of [9], we focus the evaluation on this part.

Note also that the honest-but-curious and non-colluding assumption in this setting is realistic, since the investigators might get permission from the DNA donors to share specific parts of the DNA between them. At the same time, the investigators and the biobank would not jeopardise their reputation, by colluding with each other.

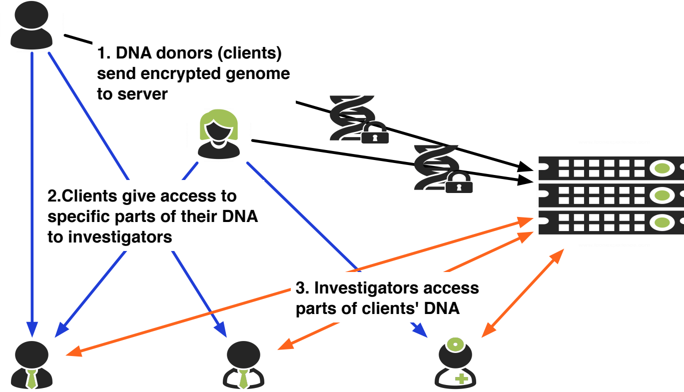

Fig. 2. Clients upload their encrypted genome on a Blurry-ORAM structure (step 1.), give to investigators access to parts of it (step 2.), which the latter access (step 3.)

7 Experimental Results

7.1 Experimental Setup

We implemented Blurry-ORAM on a Virtual Machine running Ubuntu Version 16.10 with 8 cores and 16GB of RAM, hosted on a 2x Xeon E5 2620v2 server with 12 Cores (24HT) and 64GB of RAM, using VMWare ESXi 6 for the virtualization. For the client we used a desktop PC equipped with an AMD FX(tm)-8350 Eight-Core Processor and 24GB of RAM, running on Ubuntu Desktop Version 16.04. As backend on the server, we used MySQL version 5.7.16. The code was compiled using *g++* version 6.2.0, for the cryptographic backend we used the OpenSSL library, version 1.0.2, and the experiments were ran on a 1Gbit LAN network. Note that using as many cores as possible is crucial in boosting the efficiency of our construction: After the client has identified the blocks he can decrypt, he can perform the eviction in parallel, with the rerandomization of all other blocks, using all available cores.

7.2 Instantiation

Throughout the description of our architecture (cf. Section 3.2), we have assumed a public key encryption scheme that provides indistinguishability of keys, indistinguishability of ciphertexts (so that fake blocks are indistinguishable from real ones), and allows re-randomization of ciphertexts without knowledge of the public key. We achieve these properties using a variant of the ElGamal encryption scheme on elliptic curves: For a given block B, we store as its encryption

the tuple (c_0, c_1, c_2, c_3), where $(c_0, c_1) = \mathsf{Enc}_k(B)$ and $(c_2, c_3) = \mathsf{Enc}_k(1)$. Thus for a client who has s_k as one of his secret keys, in order to check if a block belongs to him, he simply checks if $c_2^{s_k} = c_3$, instead of performing a costly decryption. Rerandomization without knowledge of the public key can also be easily done, due to the homomorphic properties, by using the ideas from [7].

7.3 Experiments

Using the techniques proposed in [4], one can efficiently store a patient's genome, by using roughly 2^{17} SNPs. Using an elliptic curve over a prime field $G(p)$ with p of size 256 bits, we can map a SNP to a single point of the elliptic curve. Typically for elliptic curve cryptography, mapping a plaintext to a point of the elliptic curve is not a trivial task. This is usually solved by concatenating random noise to the plaintext so that it can be mapped to a point of the elliptic curve. Indeed, doing this and using 16 bits of randomness for every block, we were able to map all the plaintexts we had, to points of the elliptic curve. Observe also, that for every SNP, we needed maximally 17 bits to represent the identifier and 16 bits of randomness to do the mapping to the elliptic curve. This left us with 223 bits, which provide adequate space to store the SNP, using the techniques described in [4].

In our simulations, therefore, we stored the SNPs of 100 clients, resulting in a database that stores a total of 2^{23} blocks (2^{17} SNPs per client, enough to hold a human genome stored in the form of SNPs as described in [4]). We then allowed multiple investigators access specific parts of the stored genomic data. We assume that every client distributed a different key to every investigator, so that neither other clients, nor investigators could learn anything about datablocks they would not have access to. The top left graph of Figure 3 shows the performance of our construction: we performed 10 rounds of 1 000 queries and measured the time needed for every query, depending on the number of keys a client has access to. The time needed for each query is affected by the number of investigators present (since decryption is attempted with all the client's keys), and varies from 6.48 s on average, when only one investigator is present to 63.01 s when 100 investigators are present.

In a similar setting, we varied the number of clients storing their data, (again with use 2^{17} SNPs per client), while 100 investigators were present, each one of them using a different key. We performed 10 rounds of 1 000 queries. The results are presented in the top right graph of Figure 3. We observe that, when 10 clients were present, the average time for a query was 6.9 s, when 50 clients were present, the average time was 31.40 s and when 100 clients were present, the average time for a query was 63.01 s.

In the lower left graph of Figure 3 we examined how the number of datablocks affects the performance of our construction. We instantiated our construction with 50 clients and 10 investigators (i.e., 10 keys per client) and stored 2^{15}, 2^{16} and 2^{17} datablocks per client. We performed again 10 rounds of 1 000 queries and measured an average query time of 6.92 s, 7.42 s and 7.71 s, when each client stored 2^{15}, 2^{16} and 2^{17} datablocks, respectively.

As we have seen in Section 5, the occupancy of the commonstash plays a very important role for client-to-client privacy. To experimentally assess the size of the stashes, we performed 10 rounds of 2^{17} queries (i.e., access of all blocks) in the database of 100 clients with 2^{17} blocks, 1 000 keys each, and set $Z = 2$, i.e. 2 blocks per client per node. We examined the sizes of the localstash and the commonstash. The results are shown in the bottom right graph of Figure 3, where we see that the commonstash was *never* used. On the other hand, the localstash attained once a maximum occupancy of 20 blocks.

The previous experiment focused on storing all SNPs, which store only of the positions where a human's genome differs from the reference genome. Now we examine the applicability of the

solution to databases storing full genomes. Using elliptic curves as described above, we can store the approximately $3 \cdot 10^9$ characters, using 2^{23} blocks per client, with each block storing approximately 128 characters[5]. We ran a series of 10 rounds of 1 000 queries in a database with 2, 4 and 8 clients that share 10 blocks. The results are shown in the left graph of Figure 4, where we see that the average query time was 50 s, 55 s, 60 s respectively. In a similar manner, we ran 10 rounds of 1 000 queries in a database storing the whole genome of 8 clients, that share 10, 50 and 100 blocks and the results. The right graph of Figure 4 shows that the average query time was 4,11, 6,98 and 11,26 respectively in this case, thus yielding a practical solution.

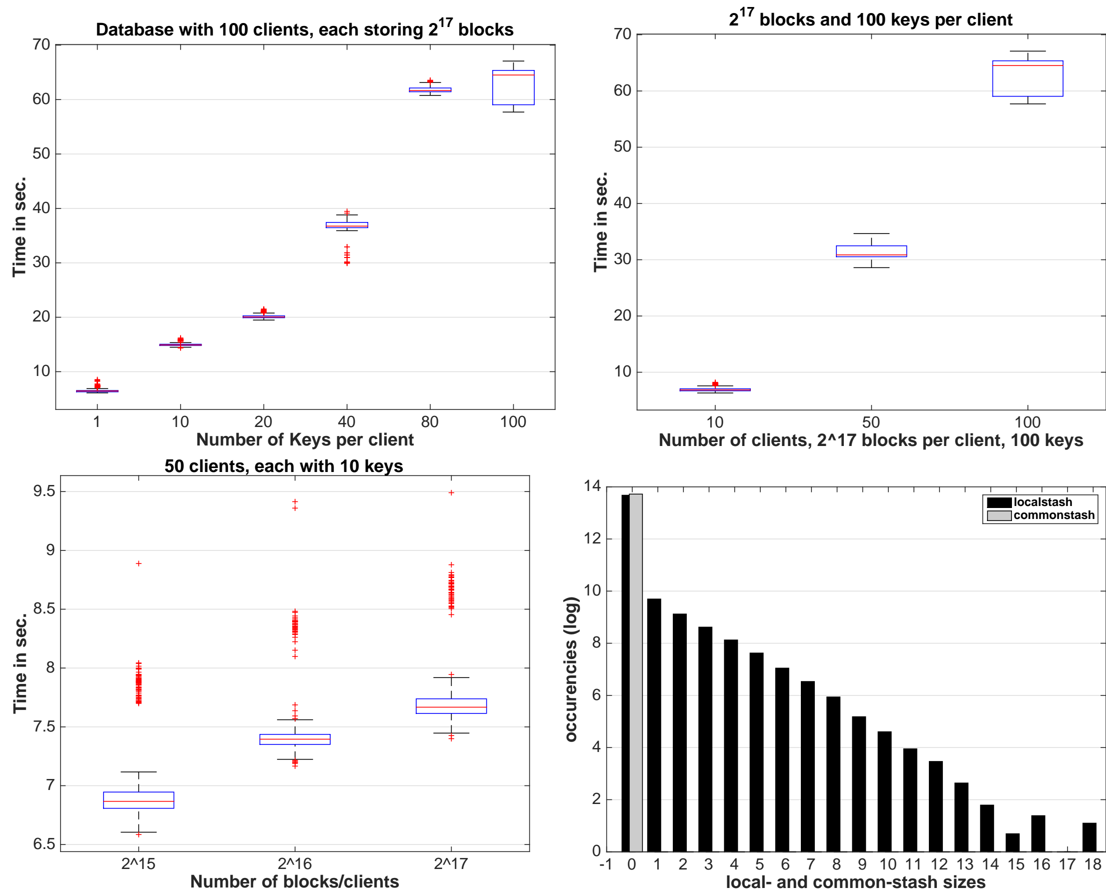

Fig. 3. Performance of Blurry-ORAM, when varying the number of keys (top left), and varying the number of clients (top right) in the scenario of storing the SNPs. Performance of Blurry-ORAM when varying the number of datablocks per client (bottom left). Sizes of the local- and commonstash for $Z = 2$ with each client sharing 1 000 blocks and storing 2^{17} blocks (bottom right).

[5] Note that since the genome's alphabet consists only of the four letters A,T,G,C, we only need 2 bits to represent each letter of the alphabet.

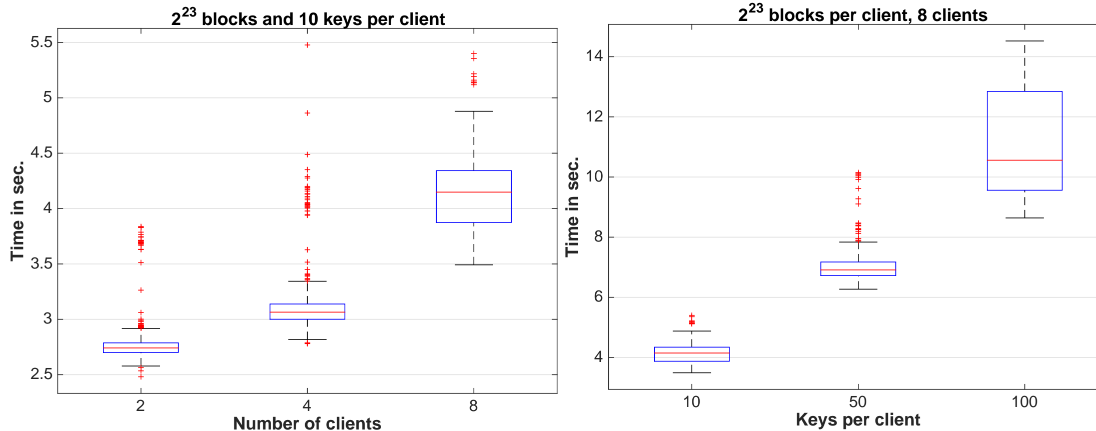

Fig. 4. Blurry-ORAM performance, when varying the number of clients (left), and the number of keys per client (right). The experiment used $Z = 2$, and allowed each client to store 2^{23} blocks.

8 Conclusion

In this work, we constructed Blurry-ORAM: an ORAM construction that allows multiple clients to store their encrypted data on a remote server, share parts of it with each other, and access it in a way that protects their privacy both against the server, and other clients. We formalized the notion of access pattern privacy in a multi-client setting, and inspired by the GWAS case, we showed that our construction is practical. 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. 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 [11], so that access pattern anonymity can be also achieved.

Acknowledgments This work has been funded by the DFG as part of project S5 within the CRC 1119 CROSSING, and by the Netherlands Organisation for Scientific Research (NWO) in the context of the CRIPTIM project.

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. Mihir Bellare, Alexandra Boldyreva, Anand Desai, and David Pointcheval. Key-privacy in public-key encryption. In *ASIACRYPT 2001*.
3. Elette Boyle, Kai-Min Chung, and Rafael Pass. Oblivious parallel RAM and applications. In *TCC 2016-A, Part II*, pages 175–204.
4. Marty C. Brandon, Douglas C. Wallace, and Pierre Baldi. Data structures and compression algorithms for genomic sequence data. *Bioinformatics*, 2009.

5. 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.
6. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Travis Mayberry, Erik-Oliver Blass, and Guevara Noubir. Multi-user oblivious RAM secure against malicious servers. *IACR Cryptology ePrint Archive*, 2015.
13. 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.
14. 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.
15. 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.
16. 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.
17. 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.
18. 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.
19. Jinsheng Zhang, Wensheng Zhang, and Qiao Daji. A multi-user oblivious ram for outsourced data. http://lib.dr.iastate.edu/cs_techreports/262/, 2014.