Raphael R. Toledo and George Danezis

Mix-ORAM: Using delegated shuffles.

Abstract: Oblivious RAM is a key technology for securely storing data on untrusted storage but is commonly considered impractical due to its high overhead. To ensure full privacy, the database has to be periodically randomized, i.e. re-encrypted and shuffled, by the client. The computation and communication costs the client incurs are in reality super linear in the number of records, this very price deters the use of ORAM for most applications. We propose in this paper to increase ORAM's practicality by delegating the randomization process to semi-trusted third parties such that the clients do not sustain to the corresponding overhead. To do so, we present four different designs inspired by mix-net technologies and evaluate them. TODO: results

Keywords: Oblivious RAM, Mix-net, Private Queries

1 Introduction

Cloud technologies offer the ability to save impressive amounts of data safely and privately on remote servers. To do so, not only the data integrity must be preserved but also the confidentiality of the data content and meta data from both external adversaries and the cloud itself. Cryptographic measures are thereby taken such as secure communication channels, user authentication, data encryption and integrity checking. These actions, however, do not prevent the leakage of all meta data: the server owner can monitor user activities and watch which records were accessed.

Oblivious RAM (ORAM) [15], or Oblivious Storage (OS) [7], precisely prevents an adversary from observing the record access, and do so by introducing a pairing of virtual and real, or remote and local, indices. In these schemes, the records are encrypted and permuted before being uploaded to the untrusted storage and the information to decrypt the records, the encryption keys, and make the matching between the remote and local indices, a pseudo random generator seed for instance, is saved locally. When the user seeks a given record, the local client computes with the seed the corresponding remote index, fetches the encrypted data block obliviously and decrypts it. After a number of accesses, in order to render any leaked information obsolete, the database is fetched and before being uploaded back to the remote server, encrypted and shuffled once again during a so called eviction process.

This eviction is the main bottleneck of ORAM. Indeed, we assume in such design that the number of records stored remotely is orders of magnitude higher than the client storage. As every record must be re-encrypted and permuted locally to ensure nothing is leaked, the client has to download and process the database by chunks, and do so several times to undeniably hide from the adversary the records' ordering. Thus as the the database size grows, the randomization cost rises super linearly.

Mix networks [8] are anonymity systems whose goal is to obliviously shuffle packets. A batch of packets is processed by the mix-net in that each mix re-encrypts, or decrypts, and shuffles the packets before sending them to the next mix. For the adversary to link the mix-net's input and output, all of the mixes used in the shuffling have to be compromised. As the mixes perform all the operations needed in ORAM, mix-nets inspired designs could thus address the delegation of ORAM's randomization process. The total cost of the eviction would however increase as several mixes are needed to ensure the confidentiality of the randomization and that the computation and communication costs are linear in the number of mixes. If using these designs, methods to amortize the eviction must be taken into account to make the randomization practical.

In this work, we present several privacy friendly distributed systems inspired by mix-nets to safely delegate ORAM's randomization process to semi-trusted third parties. The advantages of such practices are, besides the reduction of the client computation, the possibility to delay the eviction to quieter times, the database availability during the eviction process regardless of the ORAM design and the independence from centralized parties.

Raphael R. Toledo: University College London (r.toledo@cs.ucl.ac.uk)
George Danezis: University College London (g.danezis@ucl.ac.uk)

Our contributions are listed as following:
- We present and motivate the use of mix-net to construct ORAM schemes.
- We present a number of designs, improve them with load balancing via parallel mixing and compare their costs and efficiency.
- evaluation
- implementation

We present the related work, the ORAM model, its associated threat model and explaining the different costs in Section 2 and 3, we use random transposition shuffles them in ORAM and together with a mixnet and discuss various optimizations in Section 4. We then present our implementation and compare the costs with several designs in Section 6. We finally evaluate our schemes and discuss about the advantages and drawbacks of using mix-nets.

2 Related Work

ORAM. ORAM was first presented by Goldreich and Ostrovsky in 1990 [29] to prevent reverse engineering and protect softwares run on tamper resistant CPU. In 2011, ORAM solutions, called Oblivious Storage (OS) [7], were introduce to protect data stored on untrusted remote servers. Since then, several types of enhancement have been proposed including *data structures* diversification [16, 32–34] with trees, partitions and hierarchichal solutions appearing, the use of more and more sophisticated *security definitions* with statistical security [1, 10] and differential privacy [35], and the revision of *item lookups* with cuckoo hashing [31] and bloom filters [37]. Most ORAM constructions are based on a single client-server model, but multi-user designs were gradually introduced as [4] in 2016 which also provides user anonymity, some relying on access control [14] or group access [20]. The eviction process is one of the principal problems of ORAM, the clients have to re-encrypt and process the whole database, a lengthy process during which record access is usually not possible, some designs permitting read while shuffling as [7].

Shuffling and Sorting. Shuffle and sorting algorithms are a thoroughly researched subject central to ORAM for the randomization process. However most of the existing methods are not useful for ORAM as they are not oblivious -in that the permutations done depends on the data itself-.

Examples of oblivious sorting algorithms include sorting networks such as Batcher's [5] and the ones based on AKS [2] which unfortunately were proved to be impractical because of the high number of I/Os, Batchers using $\mathcal{O}(n \log n)$ I/Os and AKS having a high constant factor, but also more recent and efficient ones [30]. The randomized Shellsort [17] is an elegant simple data-oblivious version of the Shellsort algorithm running in $\mathcal{O}(n \log n)$ time that sorts with high probability. The Zig Zag sort [18], presented in 2014, is the deterministic data-oblivious variant of the Shellsort with running time of $\mathcal{O}(n \log n)$. Lastly, the bucket sort Algorithm was studied in Melbourne shuffle [28] where a user rely on temporary arrays and dummies and in [19] where the authors assess the information leakage due the use of a partially compromised parallel mix-net.

Mix-nets. Mix-nets were first presented for anonymous e-mailing by David Chaum in 1981 [8]. As they became popular many improvements were made over the years [11–13, 27]. Mix-nets' main goal is to give users some anonymity by hiding the correspondence between the incoming users' packets and the mix-nets output. To do so, the users' packets go through several mixes which permute them and refresh their encryption. Either reencryption [36] and onion encryption can be used, proofs of shuffle [6, 21, 22] and Randomized Partial Checking [23] can help verify the shuffle correctness.

This work is inspired by the mix-net technology for its encryption and permutation functionalities, however, only the packet unlinkability property is of interest for ORAM. From now on, we refer traditional ORAM solutions as ORAM and our designs as Mix ORAM.

3 Preliminaries

3.1 Model

Oblivious RAM systems rely on two data arrays, one we call *database* and comprises the user's encrypted records and some dummies, and a temporary one we call *cache*, e.g. the shelter in [15], of lesser size so that it fits on the client and used to store the fetched records in order to hide the number of times they were accessed. We also assume the client has access to an index making the matching between records' addresses and names, and envisage an additional ORAM memory to store information about the eviction process.

Access method: To do a read or write operation, the client first downloads the cache and checks whether the desired record is present, if so a dummy is fetched from the database else the desired record. Finally, the fetched element is encrypted and if it is a read operation else overwritten with the provided data for a write operation. The element is then stored locally in the cache before the latter is sent back to the remote server.

Eviction method: When the cache is full, the client needs to start the eviction process to empty it and randomize the database. To do so, the client first *rebuilds* the database by merging the cache and the database obliviously and afterwards, starts the *oblivious shuffle*.
The *rebuild phase* consists in placing obliviously records from the cache back to the ORAM database. The client begins by downloading the cache, then re-encrypts the records and discards any dummies contained within it. It next fetches all the records previously accessed from the database, and overwrites them one by one with their cached version or with a dummy before updating them back in the database. The cache is now empty.
The client starts *the oblivious shuffle* by selecting a set of mixes from the mix-net and sending to them randomization instructions containing the list of participating mixes and the seeds used for shuffling and encrypting. When receiving the records, the first mixes fetch the database, encrypt the records, shuffle and transmit them to the next mixes according to these instructions.

3.2 Security definitions and Threat model

We presume here of the existence of motivated adversaries trying to subvert a target user's privacy and perhaps compromise his data integrity. We assume the target user utilizes an ORAM system, compliant with the Privacy Definition 1 introduced by Stefanov et al. [33], to protect his data. We futhermore assume that all communication between the client, ORAM server and mixes may be intercepted as in the *global passive adversary* assumption however only message timing, volume and size from honest parties can be known thanks to packet encryption. Finally, we suppose the adversaries have corrupted a number of machines to achieve their goal, unless said otherwise, the ORAM server and all but one mix are considered compromised. We will assume that the compromised machines behave in a *honest but curious* way in that every operation is correctly performed but passively recorded and shared with the adversaries.

Privacy Definition 1. *Let's denote a sequence of k queries $seq_k = \{(op_1, ad_1, data_1),\ ..., (op_k, ad_k, data_k)\}$, where op denotes a read or write operation, ad the address where to process the operation and data the block to write if needs be else $\perp$. We denote by $ORAM(seq_k)$ the resulting randomized data access from the ORAM process with input seq_k. The ORAM guarantees that $ORAM(seq_k)$ and $ORAM(seq'_{k'})$ are computationally indistinguishable if their lengths are equal ($k = k'$).*

This work focuses on the ORAM eviction process and more precisely on the oblivious shuffle phase where sequences of data-blocks are shuffled and encrypted in order to hide the records indices after access information has leaked. This problem refers to in the Square Root solution [29] the eviction of the shelter in the database and in the Hierarchical case [16] the eviction of upper partitions in a lower ones. We evaluate our designs with the following Security Game 3.2 and consider the adversaries have won the game when discovering the ordering of the remote records.

Security Game. *An adversary gives to the user two ORAM query sequences seq_k and seq'_k of the same size. The user chooses randomly one of the two, executes it and start the eviction. At the end of the eviction the adversary picks which ORAM query sequence was chosen. The adversary wins if the right sequence is chosen with probability higher than* $\Pr = \frac{1}{2} + \epsilon$, *with* $\epsilon \ll \frac{1}{2}$.

Rationales. We consider here an ORAM remote server consisting of a database with memory of n b-bit long data blocks and a cache with memory of s, $s \ll n$, b-bit long data blocks. We furthermore consider a mix-net composed of m mixes with memory of n/m data blocks, and a client with memory of s data blocks. The ORAM server, the mixes and the client additionally have a small memory of capacity $\mathcal{O}(m)$ to store extra information about permutation and encryption.

Costs. We denote the communication costs by ω_M for the number of *bits* sent by the ORAM, γ_M by the client, and μ_M sent per mix. We group aggregate all the computation costs (encryption, shuffle and secret generation), and denote them ω_P for the ORAM, γ_P for the client, and μ_P per mix.

3.3 Cryptographic Primitives

PRG & Seeds. ORAM systems make use of pseudo random generators (PRG) and seeds to make the matching between remote and real indices. A distribution $\mathcal{D}$ over strings of length l is said pseudo random if $\mathcal{D}$ is indistinguishable from the uniform distribution over strings of length l [24]. That means it is infeasible for any polynomial-time adversary to tell whether the string was sampled accordingly to D or was chosen uniformly at random. A PRG is a deterministic algorithm that receives as an input a short random key and stretches it into a long pseudo random stream.

Encryption. ORAM designs heavily rely on encryption mechanism to obfuscate the database records as the records must be re-encrypted during the eviction and access processes.

Advanced Encryption Standard (AES) [9] was conceived for high speed and low RAM requirements. It can present throughput over 700 MB/s per thread on recent CPUs such as the Intel Core i3 [26] which makes it the ideal choice for ORAM.

To minimize the size of the instructions stored and sent by the client, we make use of elements of a cyclic group of prime order satisfying the decisional Diffie-Hellman assumption. The mixes make use of these elements and derive the different permutation seeds and encryption keys with the a key derivation function, such as the HKDF [25], and refresh the them at each round by blinding them with the shared secrets as in [12].

4 Mix-ORAM

In this section, we first introduce two different methods we use to randomize the records during the eviction. We then present simple but expensive Mix-ORAM schemes before optimizing them with the use of distributed shuffle algorithms.

4.1 Mix and User encryption methods

We present here two ways to delegate the eviction process to a semi-trusted mix-net. For each method, we show how the mix-net encrypts and permutes the records and how the client recovers a record plain text. We make the assumption that all data has first been encrypted with the client private keys before using ORAM.

Layered method. In the Layered method, we use the data structure shown in Table 1 composed of an IV token, and a label appended to the record, e.g. the record local indices. The underlying principle of the layered method is to let the number of encryption and permutation layers grow. The records are encrypted and permuted by the mixes during the eviction, and only decrypted by the client during the access. We moreover make use of the record index to store the current remote indices. Before sending the records to the ORAM database, the client first needs to encrypt and permute them. Each record is thus appended with its local index making together the data in the data structure and a random IV token. The records are then encrypted with the client secret key as stated in Algorithm 4.1, and permuted with a newly generated random seed stored locally. Finally the record index is updated with the new record indices. The encryption keys, permutation seeds and the record index may be uploaded to the extra space of the ORAM database.

IV token	data = label | record
$8 \cdot \lceil \log(n)/8 \rceil$ bit	$8 \cdot \lceil \log(n)/8 \rceil + b$ bit

Table 1. Layered method data structure.

During the eviction, the mixes encrypt both the IV token and the data separately as in Algorithm 4.1. They first generate an initialisation array from the IV token and use it to encrypt the data with keys derived from the secrets shared with the client. The first bits of the data are then used to generate another IV to encrypt with the IV token. Before sending the records to the next mixes, the data block needs to be permuted. To do so, the mixes derive from the shared secrets permutation seeds from which the new orders are generated, the data blocks are then sorted according to them.

Algorithm 4.1: Layered encryption primitive

Input: Record rec;
Encryption key k;

1 $IV0 \leftarrow PRF(rec.token \cdot k, 128)$;
2 $rec.data \leftarrow enc\,(k,\ IV0,\ rec.data)$;
3 $IV1 \leftarrow PRF(rec.data \cdot k, 128)$;
4 $rec.token \leftarrow enc\,(k,\ IV1,\ rec.token)$;

Output: rec

When retrieving a record, the client needs to decipher all encryption layers. As the number of encryption layers per record varies, a timing attack can occur letting the adversary guess when the record was last fetched. To prevent the attack, we modify the access method so that the decryption happens offline (c.f. Algorithm 4.2). After retrieving a record, the client now directly encrypts it with its own key and updates it in the cache stored locally. The local cache is then uploaded to the remote server. After doing so, if it is a write operation, the client overwrite the record with its new version. If it is a read operation, the client locally decrypts the previous encryption and remove the mix encryption layers. And finally, the client save locally an encrypted version of the record.

Algorithm 4.2: Layered access method

Input: Local record index $index$;
Operation and data $op,\ towrite$;
Encryption and permutation keys $k,\ \sigma$;
Saved records $recs$;
Number of rounds r;

$j \leftarrow recover\_index(\sigma,\ index)$;
$cache \leftarrow fetch\_cache()$;
forall the $rec \in cache$ **do**
  $cache[rec] \leftarrow$ **decrypt**$(k,\ r,\ \sigma,\ cache[rec])$;
if $record \in cache$ **then**
  $tofetch \leftarrow choose\_dummy()$;
else
  $tofetch \leftarrow tofetch\{j\}$;
$record \leftarrow fetch\_record(tofetch, cache)$
$update\_cache(recs \cup$ **encrypt**$(record))$;
$send\_cache()$;
if $op! = read$ **then**
  $record \leftarrow \{towrite\}$;
else
  $record \leftarrow$ **decrypt**$(k,\ r,\ \sigma,\ record)$;
$save\_in\_recs($**encrypt**$(k,\ record))$;

Output: record

To decrypt a record, the client uses a trial error Algorithm 4.3 based on a decryption routine. The client first removes the client encryption, executes the routine r times with the shared secrets and decrypts the data block another time with its private key. If it reads the right label, the process stops, if not the data-block is re-encrypted and it starts again.

Algorithm 4.3: Layered Decryption algorithm

Input: Record and index $record,\ index$;
Shared encryption keys $k_{mix,eviction,round}$;
Private key prv;
Number of rounds r;
Permutation seeds σ;

$j, e = 0$;
$r \leftarrow$ **decrypt_rtn**(prv_e, rec);
while $rec.data.label! = i$ **do**
  if $e! = 0$ **then**
    $rec \leftarrow encrypt(prv_{e+1},\ rec)$;
  forall the $k \in [\![1 : r]\!]$ **do**
    $m \leftarrow retrieve\_mix(\sigma,\ e,\ j,\ index)$;
    $rec \leftarrow$ **decrypt_rtn**$(k_{m,\ e,\ j}, rec)$;
    $j \leftarrow j - 1$;
  $rec \leftarrow$ **decrypt_rtn**$(prv_e,\ rec)$;
  $e \leftarrow e - 1$;

Output: rec

The routine, written in Algorithm 4.4, consists in decrypting the record in the opposite way of the encryption. The IV token is first decrypted thanks to an IV generated from the data and the decryption key. It is then used to generate another IV used this time to decrypt the data.

Algorithm 4.4: Layered decryption routine

Input: Record rec;
Decryption key k;

$IV1 \leftarrow PRF(rec.data \cdot k, 128)$;
$rec.token \leftarrow dec\,(k,\ IV1,\ rec.token)$;
$IV0 \leftarrow PRF(rec.token \cdot k, 128)$;
$rec.data \leftarrow dec\,(k,\ IV0,\ rec.data)$;

Output: rec

Rebuild method. The rebuild method aims at replacing all the mix encryption and permutation layers with new ones. The difficulty of this method is to replace the layers in a manner such that the intermediaries never see the underlying client encryption in order to prevent an adversary from observing which record was accessed. In order to achieve this, the records are encrypted and decrypted in two phases : a simple encryption-decryption (E/D phase) phase and a encryption-permutation (E/Π phase) one. Contrary to the Layered method, we encrypt with AES in Counter mode (AES-CTR) and use the record current index as counter.

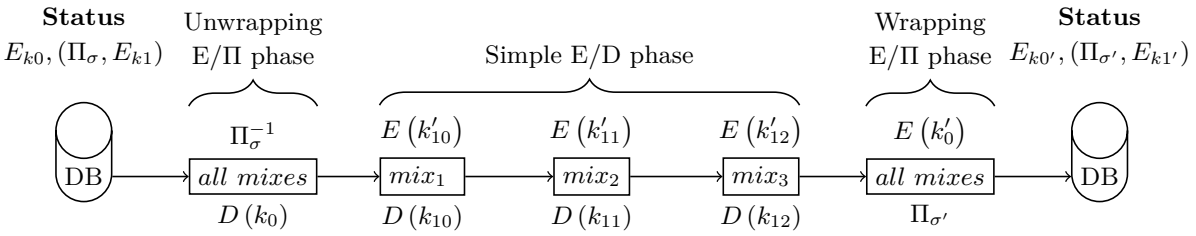

Fig. 1. Eviction under the Rebuild method, with encryptions (E) and decryptions (D) and permutations (Π) and three mixes.

Before sending the records to the untrusted storage, the client encrypts the plain-text three times. The records are first encrypted with the client own private keys and fixed counters, then with the keys generated from the secrets shared with the mixes and fixed counters. Finally the records are permuted and encrypted at the same time, with the permutation seeds and encryption keys generated from the shared secrets. For the sake of conciseness, we summarize the state of the database with the status $E_{k0}, (\Pi_\sigma, E_{k1})$ that must be understood as the database was first encrypted with the encryption keys $k0$ and then at the same time permuted and encrypted with the permutation seeds σ and the encryption keys $k1$.

During the eviction (c.f. Figure 1), the mixes first unravel the last layer (Π_σ, E_{k1}) by executing $(\Pi_\sigma^{-1}, D(k1))$ during an unwrapping E/Π phase. At the end of it, the database is then in the original sequence and encrypted under E_{k0} only. The mixes then start E/D phase where the records are encrypted with $E(k'_{0i})$ before being decrypting with $D(k_{0i})$ by each mix mix_i thanks to AES-CTR commutativity and the invariant counter. The database is now in the original sequence and encrypted under $E_{k0'}$ only. Finally, the wrapping E/Π phase starts with the mixes encrypting and permuting at the same time the records executing $(\Pi_{\sigma'}, E_{k1})$. The database is now permuted in the random order $\Pi_{\sigma'}$ and encrypted under $E_{k0'}, (\Pi_{\sigma'}, E_{k1'})$.

When retrieving a record, as shown in Alg 4.5, the client has to compute the record's remote index using the permutation seeds. The client saves all intermediary and final indices and use them as counters to decrypt the record sequentially r times as written in Alg 4.6. The client then decrypts the record with all the shared secrets and its own encryption key together with the original index as counter to reveal the plain-text. After updating the cache with the encryption of the record to read or write, the latter is sent back.

Algorithm 4.5: Rebuild access method

Input: Encryption keys $k_{mix,round}$;
Permutation seeds σ;
Record and index $record,\ index$;
Operation and data $op,\ towrite$;
Number of rounds and mixes $r,\ m$;

$indices \leftarrow \{\}$;
$round \leftarrow \{\}$;
forall the $i \in [\![1, r]\!]$ **do**
  $round,\ indices \leftarrow$
  $recover\_indices(index,\ r,\ \sigma)$;
$cache \leftarrow fetch\_cache()$;
if $record \in cache$ **then**
  $record \leftarrow find(record)$;
  $tofetch \leftarrow choose\_dummies(cache, 1)$;
else
  $tofetch \leftarrow indices[r]$;
$record, d \leftarrow fetch(tofetch)$;
$record \leftarrow$ **decrypt**$(record, k, \sigma)$;
if $op == read$ **then**
  $update\_cache(encrypt(record, k, \sigma))$;
else
  $update\_cache(encrypt(towrite, k, \sigma))$;
$send\_cache()$;

Output: record

Algorithm 4.6: Rebuild decryption algorithm

Input: Record and index $rec,\ index$;
Number of rounds and mixes $r,\ m$;
Encryption keys $k_{mix,round}$;
Mix and Indices $round,\ idx$;

forall the $i \in [\![1, r]\!]$ **do**
  $mix \leftarrow round[r - i]$
  $rec \leftarrow decrypt(rec,\ k_{mix,r-i}, idx[r - i])$;
forall the $i \in [\![1, m]\!]$ **do**
  $rec \leftarrow decrypt(rec, k_{i,0}, index)$;

Output: record

4.2 A simple Mix-ORAM

We consider here the mix-net in a classic cascade configuration and present two designs with either the use of the layered encryption method or the rebuild method. We also want to remind that we consider the database to always be permuted according to a number of seeds, denoted Π_σ, the eviction goal being to obviously sort the database to a new state $\Pi_{\sigma'}$.

Layered Cascade. The design sends the whole database through the mix-net where each mix adds a new permutation and a new encryption layers. Before the randomization starts, the database was permuted with the old seeds σ_i and encrypted with the keys k_i. After randomization, the database is encrypted with both k_i and k'_i and permuted with both seeds $\sigma'_i,\ \sigma_i$, as shown in Fig 2.

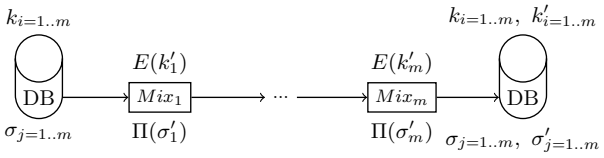

Fig. 2. Layered simple Mix-ORAM eviction.

Mix instructions. The client sends to every mix one element of a cyclic group of prime order satisfying the decisional Diffie-Hellman Assumption as in Sphinx [12] α to perform the permutations and to encrypt the records. Alongside this elements, are sent the signed list of mixes $(ports,\ ips)$ and the database access information db consisting of the IP addresses and access token. The client thus send to each mix mix_i:

$$db,\ \alpha_i,\ (ports,\ ips)$$

Let g be a generator of the prime-order cyclic group $\mathcal{G}$ satisfying the Diffie-Hillman Assumption and q the prime order of $\mathcal{G}$. We assume that each mix mix_i has a public key $y_i = g^{x_i} \in \mathcal{G}^*$ with $x_i \in_{\mathbb{R}} \mathbb{Z}_{||}$ alongside the presence of a Public Key Infrastructure to distribute an authenticated list of all (mix_i, y_i). To generate the αs, the client pick at random in $\mathbb{Z}_q$ for each mix mix_i the element z_i. The group element α and the shared secret ss are generated and the encryption keys and permutation seeds are derived as follows:

$$\alpha_i = g^{z_i}\ ;\ ss_i = y_i^{z_i}\ ;\ k_i,\ \sigma_i = hkdf(ss_i)$$

Mix operations. In this simple scheme, the mix mix_i receives a list of encrypted records from the mix mix_j (or fetch the database if $i = 0$). The mix mix_i encrypts the records with the k_i and permutes the records with the seed σ_i and sent to mix_{i+1} (or the database if $i = m$).

Algorithm 4.7: Layered Cascade mix operation for mix mix_i.

Input: Database info db;
Group elements α_i;
List of mixes $list = (ips,\ ports)$;
Private key x_i;
Data $data$;

$k_i,\ \sigma_i = hkdf(\alpha_i^{x_i})$;
$index \leftarrow list.index(mix_i)$;
if $index == 0$ **then**
  $data \leftarrow fetch\_DB(db)$;
forall the $d \in data$ **do**
  $d \leftarrow encrypt\_cbc(k_i, d)$;
$data \leftarrow \Pi_{\sigma_i}(data)$ **if** $index! = m$ **then**
  $data \leftarrow send\_mix(list[index + 1], data)$;
else
  $data \leftarrow send\_DB(db, data)$;

Client Operations. To find a record position in the database, the client just needs to look at the current index in the record index.

Costs. As the whole database is sent through the mix-net, the mix communication cost is $\cdot m \cdot n \cdot b$. The client communication is $m \cdot (32 + m \cdot (ip + ports) + token)$. The computation cost for the client is ..., and ... per mix TODO: .

Rebuild Cascade. The design refreshes both permutation and AES-CTR encryption. Before the randomization starts, the database was permuted with the old seeds σ_i and the records encrypted with the keys k_i and their index as counter. After randomization, the database is permuted with the new seeds σ'_i and encrypted with the new keys k'_i. The mixes operates as as depicted in Fig 3: they first decrypt and unwrap the records, then change the inner encryption by encrypting with the new keys and decrypting with the old, and finally wrap and encrypt again with the new keys.

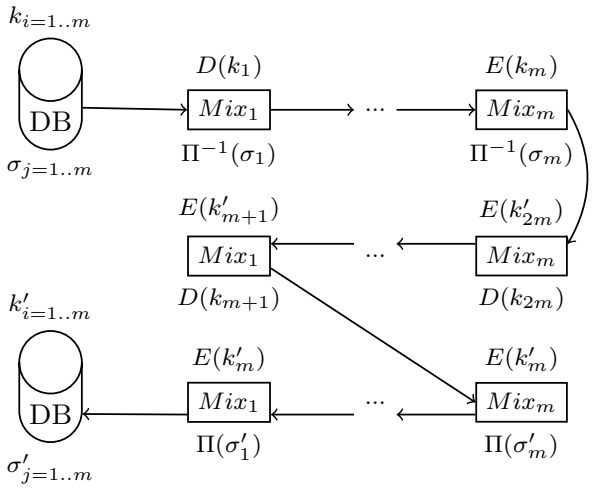

Fig. 3. Rebuilt simple Mix-ORAM.

Mix instructions. The client sends to every mix two elements of a cyclic group of prime order satisfying the decisional Diffie-Hellman Assumption α_{old} to undo the old permutations and decrypt the old layers, and α_{new} to perform the new ones. Alongside these elements, are sent the signed list of mixes $(ports,\ ips)$ and the database access information db consisting of the IP addresses and access token. The client thus send to each mix mix_i:

$$db,\ \alpha_{i,\ old},\ \alpha_{i,\ new}\ (ports,\ ips)$$

Let g be a generator of the prime-order cyclic group $\mathcal{G}$ satisfying the Diffie-Hillman Assumption and q the prime order of $\mathcal{G}$. We assume that each mix mix_i has a public key $y_i = g^{x_i} \in \mathcal{G}^*$ with $x_i \in_{\mathbb{R}} \mathbb{Z}_{\text{ll}}$ alongside the presence of a Public Key Infrastructure to distribute an authenticated list of all (mix_i, y_i). To generate the αs, the client pick at random in $\mathbb{Z}_q$ for each mix mix_i the element z_i. The group elements α, shared secretsss are generated and the encryption keys and permutation seeds are derived as follows:

$$\alpha_i = g^{z_i},\ ss_i = y_i^{z_i},\ k_i, \sigma_i = hkdf(ss_i)$$

Mix operations. In this scheme, the mix mix_i receives a list of encrypted records from the mix mix_j (or fetch the database if $i = 0$). At first, the records are permuted with the old seed σ_i and encrypted with the key k_i and sent to mix_{i+1} (or itself). The mixes then encrypt the record with both the new and old keys and send to the previous mix (or the last if $i = 0$). Finally, the records are permuted with σ'_i, encrypted with k'_i and sent to mix_{i-1} (or the database if $i = 0$).

Algorithm 4.8: Rebuild Cascade mix operation for mix_i.

Input: Database info db;
Group elements $\alpha_i,\ \alpha'_i$;
List of mixes $list = (ips,\ ports)$;
Private key x_i;
Data and sender $data,\ sender$;
Time $time = 0$;

$\sigma_i,\ k_i = hkdf(\alpha_i^{x_i})$;
$\sigma'_i,\ k'_i = hkdf(\alpha_i'^{x_i})$;
$index \leftarrow list.index(mix_i)$;
if $time == 0$ **then**
  if $index == 0$ **then**
    $data \leftarrow fetch\_DB(db)$;
  $data \leftarrow \Pi^{-1}_{\sigma_i}(data)$;
  forall the $d \in [\![1 : data]\!]$ **do**
    $data[d] \leftarrow encrypt\_ctr(k_i, d, data[d])$;
  $time \leftarrow 1$;
  $receiver \leftarrow list[index]$;
  if $index! = m$ **then**
    $receiver \leftarrow list[index + 1]$;
  $send\_mix(receiver, data)$;
else if $time == 1$ **then**
  forall the $d \in [\![1 : data]\!]$ **do**
    $data[d] \leftarrow encrypt\_ctr(k_i, d, data[d])$;
    $data[d] \leftarrow encrypt\_ctr(k'_i, d, data[d])$;
  $time \leftarrow 2$;
  $receiver \leftarrow list[m]$;
  if $index! = 1$ **then**
    $receiver \leftarrow list[index - 1]$;
  $send\_mix(receiver, data)$;
else
  forall the $d \in [\![1 : data]\!]$ **do**
    $data[d] \leftarrow encrypt\_ctr(k'_i, d, data[d])$;
  $data \leftarrow \Pi_{\sigma'_i}(data)$;
  $receiver \leftarrow db$;
  if $index! = 1$ **then**
    $receiver \leftarrow list[index - 1]$;
  $send\_mix(receiver, data)$;

Client operations. When retrieving a record, the user recovers the remote index thanks to the last seeds as depicted in Algorithm 4.9 where the last index being the one sought.

Algorithm 4.9: Cascade Rebuild Index Lookup

Input: Seeds σ;
Number of records and mixes $n,\ m$;
Record index $index$;

```
mixes ← {,};
indices ← {,};
forall the i ∈ [[1, m]] do
    index ← Π_{σ_i}(n, index);
    mixes ← mixes ∪ {i};
    indices ← ∪{index}
```

Output: $mixes,\ indices$

Costs. As the whole database is sent through the mix-net three times, the mix communication cost is $3 \cdot m \cdot n \cdot b$. The client communication is $m \cdot (2 \cdot 32 + (m+1) \cdot (ip + ports) + token)$. The computation cost ... TODO: .

Partial Conclusion. These two designs are not efficient as only a mix work at a time. To increase the mix-net efficiency, we study in the following section parallelization to distributing the workload among mixes.

4.3 Parallelizing the Eviction process.

In this section, we replace the cascade configuration of the mix-net with a parallel one and simulate random transposition shuffles (RTS) thanks to the use of private and public permutations as shown in Figure **??**. We also calculate the number of rounds needed to reach perfect security by presenting firstly the mixing time of k-RTS before introducing ORAM assumptions to reduce the expected time to achieve randomness.

During the eviction process, we assign chunks of the database to each mix: when the eviction starts, each mix fetches its assigned chunk, for instance mix_i fetches the indices in $[\![i \cdot n/m : (i+1)n/m[\![$. The mixes, after receiving the records, permute them with private permutation seeds and finally send them to the mix-net according to public permutation seeds. At the end of the eviction, the mixes uploads the records back to the database on their assigned indices.

4.3.1 k-Random Transposition Shuffle.

Random Transposition Shuffles (RTS) are widely used models in the study of card shuffling. It consists in a player picking randomly a couple of cards from a same deck, permuting them according to a coin toss and putting them back at the same location. These steps, usually called a round, are then repeated until the deck of cards has been properly shuffled, i.e. until every card arranging is possible.
We can already see why RTS are natural candidates for amortized ORAMs : if they can be broke down in independent rounds which can be spread over several entities and time, so can a randomization process based on them. Furthermore, having the client (player) permuting the data blocks (cards) locally is enough to make RTS oblivious to the eyes of an adversary. Diaconis in 1986 [3] has proved that the RTS mixing time of a deck of n cards is of the order $O(n \log(n))$. We thus first look at oblivious k-RTS, an RTS where the client picks and transposes locally k distinct cards to make the scheme more efficient. We stress the difference between doing successively $k/2$ transpositions and what we call k-RTS: in the first case, an element can be transposed several times in a row which leads to a different probability distribution. The result we present affirms that k-RTS converges to the uniform distribution more rapidly than repeating normal RTS.

Security Theorem 1. ***Mixing time of k-RTS.*** *A k-random permutation shuffle of a n card game reaches the uniform distribution in τ rounds, such that*

$$E(\tau) < \frac{2}{k} \cdot \frac{n^2}{n+1} \cdot (\log(n) + \mathcal{O}(1))$$

Proof. See Appendix 10.1. □

This theorem gives an upper bound of the number of rounds for $k/2$ disjoint transpositions. However, we use in practice PRG keys which make a permutation of the k elements. This permutation can be decomposed as a sequence of transpositions which may not be disjoint or of size k. Hence when using PRG seeds, we converge even faster to the uniform distribution. We thus consider that in practice an oblivious k-RTS implies computation and communication cost of the order of $\mathcal{O}\left(\frac{n}{k} \cdot \log(n)\right)$.

When different parties, the m mixes in our case, perform in parallel the k-RTS, we can improve by another factor m the eviction computation time. However, as some mixes can have been compromised by the adversary we ask each mix to perform $2m \log n$ rounds to guarantee privacy.

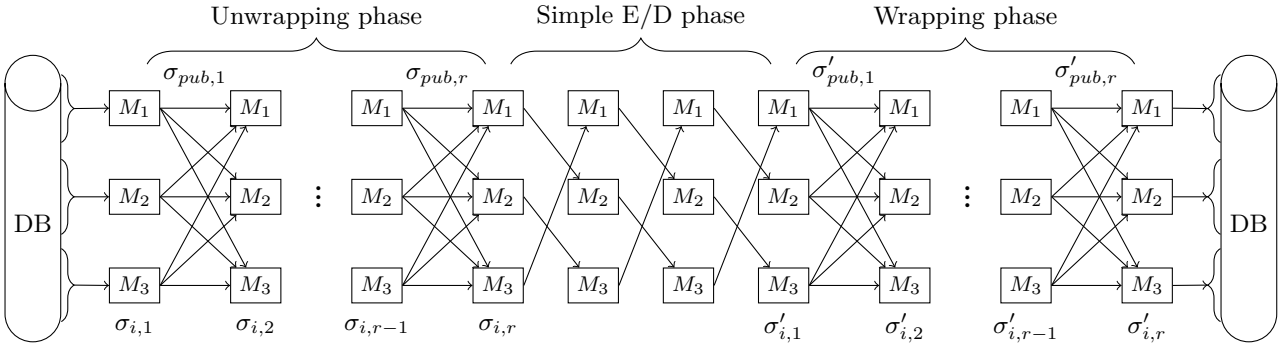

Fig. 4. Parallel mix-net: Rebuild method (all phase) and Layered method (only the Wrapping phase)

4.3.2 Oblivious Merge

Before the eviction algorithm is run, the database can be divided in two sets of records depending on whether or not they were retrieved by the user. As such, the database can be represented as a simple binary array of n bits out of which s are 1s, the accessed ones, and $n-s$ are 0s, the others. We argue that in this representation, elements of the same sets are indistinguishable to the adversary thanks to prior encryptions and permutations and thus, less rounds are necessary to obliviously shuffle the database from this state. Indeed, this assumption significantly reduces the number of possible orders in the adversarial view, there are $\binom{n}{s}$ orders instead of $n!$ (using the Bars and Stripes theorem).

We now consider the RTS process in that scenario and suppose the records (the bits) are re-encrypted before being permuting such that the merge of the two sets is oblivious to the adversary.

Security Theorem 2. *An oblivious merge (OM) of 2 indistinguishable sets of respective size n and s elements requires τ rounds of 2-RTS such that any arranging is possible, with*

$$\tau(\epsilon) \leq \frac{n}{2} \cdot \left[\left(\log\left(\frac{n}{s} - 0.5\right) + \mathcal{O}(1)\right) - \frac{2}{s}\log(4 \cdot \epsilon)\right]$$

Proof. See Appendix 10.2. □

As we could see that k-RTS decreased the mixing time by at least a factor k, and does so independently of the items to shuffle, we make the following conjecture.

Security Conjecture 1. *A k-oblivious Merge (k-OM) of 2 indistinguishable sets of n and s element requires τ rounds such that any order is equally possible, with*

$$\tau(\epsilon) \leq \frac{n}{2 \cdot k} \cdot \left[\left(\log\left(\frac{n}{s} - 0.5\right) + \mathcal{O}(1)\right) - \frac{2}{s}\log(4 \cdot \epsilon)\right]$$

4.4 Parallel Mix-ORAM

We now consider the mix-net as a collection of mixes communicating to each other. The eviction algorithm is composed of the unwrapping (for the rebuild method) and the wrapping phases which consist each of r rounds.

The mixes have each been allocated a chunk of the database (mix_{idx} having $[idx \cdot n/m : (idx+1) \cdot n/m]$) and use the public permutation seed to compute which record to send to each mix as written in Algorithm 4.10.

Algorithm 4.10: Public Record Allocation for mix_{idx} at round rnd

Input: Public seeds $\sigma_{pub,round}$;
Number of records and mixes $n,\ m$;

$records \leftarrow \Pi_{\sigma_{pub,rnd}}([\![1:n]\!])$;
$mixes \leftarrow []$;
forall the $i \in [\![1,m]\!]$ **do**
  $mixes \leftarrow mixes \cup records[i \cdot n/m : (i+1) \cdot n/m]$;
  $mixes[i] \leftarrow [mixes[i][j]$ for $j \in [1 : n/m]$ if $mixes[i][j] \in [idx \cdot n/m : (idx+1) \cdot n/m]]$;
Output: $mixes$

Layered method. In this design, chunks of the database are attributed to each mix which keep encrypting and permuting the records. Before the eviction, the database is permuted with the old seeds σ_i and encrypted with the old encryption keys k_i. Afterwards, the records are encrypted with both k_i and k'_i, permuted with both σ_i and σ'_i, and the indices are saved in the record index. As no permutation layer is removed, the record indistinguishability assumption holds, the eviction can then consist of $r = m/2\ \log(n/s)$ rounds.

Mix Instructions. The client need to send to each mix the session keys to access the database ($id,\ token$), the total number of records n, the elements used to compute the encryption keys and permutation seeds ($\alpha_i, \alpha_{pub,i}$), the number of rounds r, and the ordered $list = (ports,\ ips)$ of the mixes participating in the eviction. The client thus send :

$$db,\ \alpha_i,\ \alpha_{pub,i},\ n,\ r,\ list$$

As for the previous schemes, we have g a generator of the prime-order q cyclic group $\mathcal{G}$ satisfying the Diffie-Hillman Assumption. We assume that each mix mix_i has a private key x_i and a public key y_i, and the client the private and public key $x_c,\ y_c$; all public keys being distributed thanks to a PKI in a authenticated manner. We also assume the mixes have exchanged with the client private random elements $m_i \in_{\mathbb{R}} \mathbb{Z}_{||}$. The client sends to the mix $mix_{i\in 1..m}$ α_i generated with a random element of $\mathbb{Z}_q$, β_i computed from the m_i to compute the shared secrets $ss_i,\ sk$ and the encryption keys k_i and the permutation seeds σ as follows :

$$\alpha_{i,0} = g^{z_i}, \qquad ss_{i,0} = y_{i,0}^{z_i}, \quad k_{i,0},\ \sigma_{i,0} = hkdf(ss_{i,0})$$
$$\beta_{i,0} = g^{\Pi_{j\neq i} m_j}, \quad sk_0 = \beta_{i,0}^{m_i}, \qquad \sigma_{pub,0} = hkdf(sk_0)$$

We furthermore refresh the permutation seeds and encryption keys at each round thanks to blinding the group elements. Let $h_b : \mathcal{G}^* \to \mathbb{Z}_q^*$ the hash function we use for computing blinding factors, we can then compute the α and β for the round $j+1$ as follows:

$$b_{i,j+1} = h_b(\alpha_{i,j}, ss_{i,j}), \quad \alpha_{i,j+1} = g^{z_i \Pi_{k\leq j} b_{i,k}}$$
$$b_{pub,j+1} = h_b(y_c, sk_j), \qquad \beta_{i,j+1} = g^{\Pi_{k\leq j} b_{pub,k} \Pi_{l\neq i} m_l}$$

Mix operations. In this scheme the mix mix_i receive a list of encrypted record from all mixes (or the database). It first merges the record lists and sorts them to the natural order thanks to the public seed. It then encrypts each record and permute them according to the private key and seed. It finally computes the record allocation arrays (c.f. Algorithm 4.10) before sending the records accordingly to them.

Client operations. To find a record position in the database, the client simply needs to look at the current index in the record index.

Costs. The communication costs are ... and the computation costs ... .

Rebuild method. When performing an eviction, the client assigns to each mix $k = \frac{n}{m}$ distinct indexes together with a list of shared secrets from which will be derive public and private permutation seeds and encryption keys.
At first, the mixes fetch their allocated records from the database. They then unwrap the previous permutation: the last wrapping permutations are undone in reverse order thanks to the old private seeds and the records are decrypted with the old encryption keys before being distributed to all mixes according to the inverse permutation of the old public seeds. Then starts the wrapping, where the records are permuted according to the new seeds, encrypted with the new encryption keys and distributed to the mixes according to the new public seeds. Between the wrapping and unwrapping phases, the mixes exchanged in a cascade fashion their records and encrypt them with both the new and old key.

Mix Instructions. The client sends the same instructions as in the Parallel Layered design but with double the amount of group elements α, the old and the new ones. The message format is then :

$$db,\ \alpha_i,\ \alpha'_i,\ \alpha_{pub},\ \alpha'_{pub},\ n,\ r,\ list$$

To derive the permutation seeds σ and encryption keys k, we make use of the random private elements z, i and m_i, the public and private keys y and x.

$$\alpha_{i,0} = g^{z_i}, \qquad ss_{i,0} = y_{i,0}^{z_i}, \quad k_{i,0},\ \sigma_{i,0} = hkdf(ss_{i,0})$$
$$\beta_{i,0} = g^{\Pi_{j\neq i} m_j}, \quad sk_0 = \beta_{i,0}^{m_i}, \qquad \sigma_{pub,0} = hkdf(sk_0)$$

Contrary to the Layered method, we derive the α r more times for the simple the Encryption/Decryption phase.

$$b_{i,j+1} = h_b(\alpha_{i,j}, ss_{i,j}), \quad \alpha_{i,j+1} = g^{z_i \Pi_{k\leq j} b_{i,k}}$$
$$b_{pub,j+1} = h_b(y_c, sk_j), \qquad \beta_{i,j+1} = g^{\Pi_{k\leq j} b_{pub,k} \Pi_{l\neq i} m_l}$$

Mix Operations. During the first r rounds, the records are first sorted, then unwrapped (permuted and decrypted with the old keys and seeds) and sent to the mix-net according to the public record allocation. Then the groups of n/m records are encrypted and decrypted in m parallel cascades. Finally, the records are sorted, wrapped (encrypted and permuted with the new ones) and sent to the mix-net according to the public allocation during the last r rounds.

Client Operations. To recover a record remote position, the client derive from the $m+1$ alphas the permutation seeds used during the r rounds of operations as depicted in Algorithm 4.11 with the last index in the indices array being the one sought.

Algorithm 4.11: Parallel Index Lookup

Input: Private and public seeds $\sigma_{i,round},\ \sigma_{round}$;
Number of records, mixes and rounds
$n,\ m,\ r$;
Record index $index$;

$mixes \leftarrow \{,\}$;
$indices \leftarrow \{,\}$;
forall the $i \in [\![1, r]\!]$ **do**
  $mix \leftarrow \lfloor index/m \rfloor$;
  $mixes \leftarrow mixes \cup \{mix\}$;
  $shuffle \leftarrow \Pi_{\sigma_{mix,i}}(i \cdot n/m,\ )i+1) \cdot n/m)$;
  $index \leftarrow i \cdot n/m + shuffle.index(index)$;
  $indices \leftarrow \cup\{index\}$;
  $shuffle \leftarrow \Pi_{\sigma_i(1,n)}$;
  $index \leftarrow shuffle.index(index)$;

Output: $mixes,\ indices$

Costs. The communication costs are ... and the computation costs ... .

5 Security Proof

We first remark that all of the eviction meta data is independent of data content, as it is entirely determined by the sole parameter n. The mix instructions are never shared between mixes and never shared with the database, the encryption keys and permutation seeds thus remain secret and are refreshed at every round. The data content is also never revealed to the adversary and is refreshed upon reception by every mix.

As stated before, the security game consists in the adversary giving to the client two query sequences to process before triggering the eviction. To win the game the adversary must after the eviction choose with probability higher than $1/2+\epsilon$ which sequence was processed.

We argue that the adversary can win the game in only two ways: decrypting all the records or finding out the records' position before the eviction.

5.1 In Search of the Lost Position.

Cascade mix-net. In this architecture, the whole database is sent from a mix to another. To know the index of any block, the adversary needs to retrieve the group elements sent to the honest mixes. George:

Parallel mix-net. In this design, chunks of the database are exchanged between mixes during $r = 2m\log(n)$ rounds. The adversary can benefit of the fact that some records may never go to the uncompromised mixes (with probability $p = (1-1/m)^r << 1$) to win the game.

*Rebuild method.*Goodrich in 2012 [19], proved the security of such scheme using the sum of squares metric. We derived from their theorem a more precise condition for all but one compomised mixes (see Appendix 10.3). In that case, we need the mix-net to shuffle the records during $t = bm\log(n)$ rounds so that the expected sum of square error between the card assignment probabilities and the uniform distribution is at most $1/n^b$.

The value we chose $r = 2m\log(n)$, give us a closeness better than $1/n^2 = o(1/n)$, we can consider the scheme secure.

Layered method. In this scheme, we propose to maintain the randomness of the database after it was first shuffled by the client with evictions of $r = \frac{m}{2} \cdot log\left(\frac{n}{s}\right)$ rounds (c.f. Theorem 1). The theorem's indistinguishability assumption holds as we do not remove any permutation layer, thus the design is secure.

5.2 In Search of the Lost Key.

The Layered method. In this method, we prevent any timing attack with offline decryption, the adversary thereby cannot distinguish whether a record was accessed recently or not. The encryptions keys being never shared, a polynomial adversary cannot break the AES CBC encryption.

The Rebuild method. In this method, the adversary just need to break one layer of encryption after the wrapping phase. However as the encryptions keys are not shared between mixes, a polynomial adversary cannot break the AES CTR encryption.

6 Evaluation

Layered method. We look here at the average number of encryption layers e a record has before being decrypted. Making the assumption that the record access distribution is uniform, we can represent the problem of accessing all records at least once as a coupon collector problem. In that case, we expect $E[e_{all}] \leq (n/s) \cdot H_n$ evictions before all records have been fetched once with H_n the n^{th} harmonic number. The expected number of encryption layers per record before decryption is however $E[e] \leq r/s \cdot \left(\frac{n+1}{2} \cdot (H_n - 1/2) + 1/2\right)$. For $n = 10^6$ and $s = \sqrt{n}$, we have $E[e] \approx 15 \cdot 10^3$ and $E[r] \approx 7 \cdot 10^3 \cdot r$.

Proof. Lets τ_n be the random number of coupons collected when the first set contains every n types. We have, $E[\tau_n] = n \sum_{i=1}^{n} \frac{1}{i} = n \cdot H_n$. Since we fetch s unique records per eviction (we cannot fetch a record already in the stash), the previous result is an upper bound of the number of requests needed and so the expected number of eviction is $E[e_{all}] \leq n/s\ H_n$.

We now want to find the average number of encryption layers per record before decryption, this is equivalent to finding the average number of evictions before a record is deciphered. Hence we have, $E[e] \leq r/s \cdot \sum_{i=1}^{n} E[\tau_i] = r/s \sum_{i=1}^{n} \left(\frac{(n+1-i)(n+i)}{2} \cdot \frac{1}{i}\right)$ from which can be calculated the result presented earlier. □

To reduce these numbers, we modify the access method written in Algorithm 4.2. When the client requests a record from the database, d other records are chosen uniformly at random from the set of unaccessed records. These records are then fetched, their encryption is refreshed as written previously and the client overwrites with these records their older version on the database. Doing so, with d high enough, yields a better approximation of the uniform distribution assumption and we would obtain $E[e_{all}] \leq n/(sd) \cdot H_n$ and $E[e] \leq r/(sd) \cdot \left[\frac{n+1}{2} \cdot (H_n - 1/2) + 1/2\right]\ H_n$. With $d = \sqrt{n}$, we now have $E[e_{all}] \leq 15r$ and $E[e] \leq 7r$.

6.1 Comparison

Comparison of our schemes. We did not include the record allocation cost in the tables as it can be done while waiting for the records.

- For the cascade architecture, all the costs are linear in the number of mixes.
- For the parallel architecture, no computation cost depends on m (considering that $C_E \gg C_\Pi$), we could "nearly" say it for the communication cost George:
- The parallel architecture can be faster than the cascade one. For most database depending on the the underlying ORAM system (on s) (if $m \geq 1/2 \log(\frac{n}{s})$), for the Layered method for small to medium sized databases for the Rebuild method (if $m \geq log(n) + 1/2$).
- The Layered method needs more additional space and decryption takes longer than for the Rebuild method, however the eviction is faster Raphael: at least if the cost induced by the additional bytes for the Layered method is insignificant.

6.2 Implementation, Benchmark and Performances

xxx line of Python ...

Amazon EC2 with CPU ...

	3 mixes	5 mixes	7 mixes
Cascade Layered	0	0	0
Cascade Rebuild	0	0	0
Parallel Layered	0	0	0
Parallel Rebuild	0	0	0

Table 3. Performance (s) of the designs for the eviction of xxx records of yykbit.

performance per enc, performance per design, performance with implementation

7 Discussion

7.1 Active Adversary

As misbehaving mixes are already discarded in the eviction setup, we could integrate in the client a light-weight local mix reputation system based on Randomized Partial Checking.

8 Acknowledgement

Danezis was supported by H2020 PANORAMIX Grant (ref. 653497) and EPSRC Grant EP/M013286/1; and Toledo by Microsoft Research.

	Cascade - Layered	Cascade - Rebuild	Parallel - Layered	Parallel - Rebuild
#Rounds (r)	m	$3m$	$\frac{m}{2}\log\left(\frac{n}{s}\right)$	$4m\log(n)+m$
Mix Encryption cost	$m\cdot n$	$4m\cdot n$	$\frac{n}{2}\log\left(\frac{n}{s}\right)$	$n(4\log(n)+2)$
Mix Permutation cost	$mn\cdot C_\Pi(n)$	$4mnC_\Pi(n)$	$\frac{m}{2}\log\left(\frac{n}{s}\right)\cdot C_\Pi\left(\frac{n}{m}\right)$	$4m\log(n)\cdot C_\Pi\left(\frac{n}{m}\right)$
Mix Communication cost	$(r+1)\cdot C_{com}(n)$	$(r+1)\cdot C_{com}(n)$	$(r+1)\cdot C_{com}\left(\frac{n}{m}\right)$	$(r+1)\cdot C_{com}\left(\frac{n}{m}\right)$
Client Lookup cost	$O(1)$	$m\cdot C_\Pi(n)$	$O(1)$	$m\cdot[C_\Pi\left(\frac{n}{m}\right)h+2C_\Pi(n)]$
Client Decryption cost	$\sim\frac{nr}{2s}H_n$	$2m$	$\sim\frac{nr}{2s}H_n$	$2m$
Client Additional storage	$n\log(n)+km$	km	$n\log(n)+k(m+1)$	$k(m+1)$

Table 2. Cost comparison of the designs,
with C_E the cost of 1 encryption, $C_\Pi(x)$ the permutation cost and $C_{com}(x)$ the communication cost of x records in the scheme.

9 Conclusion

- New ORAM with mix-net
- amortizable eviction
- can fetch while eviction running
- multi-user friendly as all the secrets are on the server

References

[1] Ajtai, M.: Oblivious RAMs without cryptographic assumptions. In: Proceedings of the forty-second ACM symposium on Theory of computing. pp. 181–190. ACM (2010)

[2] Ajtai, M., Komlós, J., Szemerédi, E.: An 0(n log n) sorting network. In: Proceedings of the fifteenth annual ACM symposium on Theory of computing. pp. 1–9. ACM (1983)

[3] Aldous, D., Diaconis, P.: Shuffling cards and stopping times. The American Mathematical Monthly 93(5), 333–348 (1986)

[4] Backes, M., Herzberg, A., Kate, A., Pryvalov, I.: Anonymous RAM

[5] Batcher, K.E.: Sorting networks and their applications. In: Proceedings of the April 30–May 2, 1968, spring joint computer conference. pp. 307–314. ACM (1968)

[6] Bayer, S., Groth, J.: Efficient zero-knowledge argument for correctness of a shuffle. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques. pp. 263–280. Springer (2012)

[7] Boneh, D., Mazieres, D., Popa, R.A.: Remote oblivious storage: Making oblivious RAM practical (2011)

[8] Chaum, D.L.: Untraceable electronic mail, return addresses, and digital pseudonyms. Communications of the ACM 24(2), 84–90 (1981)

[9] Daemen, J., Rijmen, V.: The design of Rijndael: AES-the advanced encryption standard. Springer Science & Business Media (2013)

[10] Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious RAM without random oracles. In: Theory of Cryptography Conference. pp. 144–163. Springer (2011)

[11] Danezis, G., Dingledine, R., Mathewson, N.: Mixminion: Design of a type iii anonymous remailer protocol. In: Security and Privacy, 2003. Proceedings. 2003 Symposium on. pp. 2–15. IEEE (2003)

[12] Danezis, G., Goldberg, I.: Sphinx: A compact and provably secure mix format. In: 2009 30th IEEE Symposium on Security and Privacy. pp. 269–282. IEEE (2009)

[13] Danezis, G., Laurie, B.: Minx: A simple and efficient anonymous packet format. In: Proceedings of the 2004 ACM workshop on Privacy in the electronic society. pp. 59–65. ACM (2004)

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

[15] Goldreich, O.: Towards a theory of software protection and simulation by oblivious RAMs. In: Proceedings of the nineteenth annual ACM symposium on Theory of computing. pp. 182–194. ACM (1987)

[16] Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. Journal of the ACM (JACM) 43(3), 431–473 (1996)

[17] Goodrich, M.T.: Randomized shellsort: A simple oblivious sorting algorithm. In: Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms. pp. 1262–1277. Society for Industrial and Applied Mathematics (2010)

[18] Goodrich, M.T.: Zig-zag sort: A simple deterministic data-oblivious sorting algorithm running in O(n log n) time. In: Proceedings of the 46th Annual ACM Symposium on Theory of Computing. pp. 684–693. ACM (2014)

[19] Goodrich, M.T., Mitzenmacher, M.: Anonymous card shuffling and its applications to parallel mixnets. In: International Colloquium on Automata, Languages, and Programming. pp. 549–560. Springer (2012)

[20] Goodrich, M.T., Mitzenmacher, M., Ohrimenko, O., Tamassia, R.: Privacy-preserving group data access via stateless oblivious ram simulation. In: Proceedings of the twenty-third annual ACM-SIAM symposium on Discrete Algorithms. pp. 157–167. SIAM (2012)

[21] Groth, J., Lu, S.: A non-interactive shuffle with pairing based verifiability. In: International Conference on the Theory and Application of Cryptology and Information Security. pp. 51–67. Springer (2007)

[22] Groth, J., Lu, S.: Verifiable shuffle of large size ciphertexts. In: International Workshop on Public Key Cryptography. pp. 377–392. Springer (2007)

[23] Jakobsson, M., Juels, A., Rivest, R.L.: Making mix nets robust for electronic voting by randomized partial checking. In: USENIX security symposium. pp. 339–353. San Francisco, USA (2002)
[24] Katz, J., Lindell, Y.: Introduction to modern cryptography. CRC press (2014)
[25] Krawczyk, H.: Cryptographic extraction and key derivation: The hkdf scheme. In: Annual Cryptology Conference. pp. 631–648. Springer (2010)
[26] McWilliams, G.: Hardware aes showdown-via padlock vs intel aes-ni vs amd hexacore (2014)
[27] Möller, U., Cottrell, L., Palfrader, P., Sassaman, L.: Mixmaster protocolâĂŤversion 2. Draft, July 154 (2003)
[28] Ohrimenko, O., Goodrich, M.T., Tamassia, R., Upfal, E.: The melbourne shuffle: Improving oblivious storage in the cloud. In: International Colloquium on Automata, Languages, and Programming. pp. 556–567. Springer (2014)
[29] Ostrovsky, R.: Efficient computation on oblivious RAMs. In: Proceedings of the twenty-second annual ACM symposium on Theory of computing. pp. 514–523. ACM (1990)
[30] Paterson, M.S.: Improved sorting networks with O(log N) depth. Algorithmica 5(1-4), 75–92 (1990)
[31] Pinkas, B., Reinman, T.: Oblivious RAM revisited. In: Annual Cryptology Conference. pp. 502–519. Springer (2010)
[32] Ren, L., Fletcher, C.W., Kwon, A., Stefanov, E., Shi, E., van Dijk, M., Devadas, S.: Ring oram: Closing the gap between small and large client storage oblivious RAM. IACR Cryptology ePrint Archive 2014, 997 (2014)
[33] Stefanov, E., Shi, E., Song, D.: Towards practical oblivious RAM. arXiv preprint arXiv:1106.3652 (2011)
[34] Stefanov, E., Van Dijk, M., Shi, E., Fletcher, C., Ren, L., Yu, X., Devadas, S.: Path oram: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security. pp. 299–310. ACM (2013)
[35] Wagh, S., Cuff, P., Mittal, P.: Root oram: A tunable differentially private oblivious RAM. arXiv preprint arXiv:1601.03378 (2016)
[36] Wikström, D., Groth, J.: An adaptively secure mix-net without erasures. In: International Colloquium on Automata, Languages, and Programming. pp. 276–287. Springer (2006)
[37] Williams, P., Sion, R., Carbunar, B.: Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In: Proceedings of the 15th ACM conference on Computer and communications security. pp. 139–148. ACM (2008)

10 Appendix

10.1 Proof k-RTS

Proof. To first prove the upper bound and variance, we use Diaconis et al. results [3] which states that τ defined in the following game is a strong stationary time. In a random transposition shuffle, the cards chosen by the right and left hands at time t are respectively called R_t and L_t. Assuming that when $t = 0$, no card is marked, we mark R_t if R_t was unmarked before and either L_t is marked or $L_t = R_t$. The variable τ represents the time when every card has been marked, we call it the stopping time.

Let be τ_t the number of transpositions after the t^{th} card is marked, up to and including when the $(t+1)^{th}$ is marked.

$$\tau = \sum_{i=0}^{n-1} \tau_i$$

The τ_t are independent geometric variables with probability of success p_t as implied by the game rules. The probability of success corresponds to the probability of marking at least one card, one to t cards exactly. To do so, the right cards must be chosen from the unmarked set, comprising $n-t$ cards at time t, and the left cards from the union of the marked set and the right cards.

$$\begin{aligned} p_t &= \sum_{i=1}^{min(k,n-t)} \binom{k}{i} \cdot \binom{t+1}{i} \cdot \binom{n-t}{i} \cdot \binom{n}{i}^{-2} \\ &= \frac{1}{n^2} \cdot \left(k \cdot (t+1) \cdot (n-t) + \alpha_{n,t,k}\right),\ 0 < \alpha_{n,t,k} = \mathcal{O}\left(n^{-k}\right) \end{aligned}$$

We can thus rewrite τ's expectation as following.

$$\begin{aligned} E(\tau) &= \sum_{t=0}^{n-1} \frac{1}{p_t} = \sum_{t=0}^{n-1} \frac{n^2}{k \cdot (t+1) \cdot (n-t) + \alpha_{n,t,k}} \\ &< \sum_{t=0}^{n-1} \frac{n^2}{k \cdot (t+1) \cdot (n-t)} \\ &< \frac{1}{k} \cdot \frac{n^2}{n+1} \cdot \sum_{t=0}^{n-1} \left(\frac{1}{t+1} + \frac{1}{n-t}\right) \\ &< \frac{2}{k} \cdot \frac{n^2}{n+1} \cdot \left(\ln(n) + \gamma + \mathcal{O}(\frac{1}{n})\right), \gamma = \lim_{n\to\infty} H_n - \ln(n) \end{aligned}$$

$$\begin{aligned} var(\tau) &= \sum_{t=0}^{n-1} \frac{1-p_t}{{p_t}^2} \ < \ \sum_{t=0}^{n-1} \frac{1}{{p_t}^2} \\ &< \sum_{t=0}^{n-1} \frac{1}{\left(k\frac{(t+1)\cdot(n-t)}{n^2} + \alpha_{n,t,k}\right)^2} \\ &< \sum_{t=0}^{n-1} \frac{1}{\left(k\frac{(t+1)\cdot(n-t)}{n^2}\right)^2} \\ &< 2 \cdot \left(\frac{n}{k}\right)^2 \cdot \left(\frac{n}{n/2}\right)^2 \cdot \sum_{t=0}^{n/2-1} \frac{1}{(t+1)^2} \\ &< \frac{4}{3}\pi^2 \cdot \left(\frac{n}{k}\right)^2 \end{aligned}$$

To now prove the lower bound of τ, we will compare the number of fixed points of a permutation σ, $F(\sigma)$, for our shuffle, the permutation obtained from the identity by applying kt random transpositions $P^{kt}(id,\cdot)$, and the uniform distribution π, or more precisely compare the corresponding probabilities over the set $A = \{\sigma : F(\sigma) \geq \frac{\mu}{2}\}$. We can say that after t shuffles, the number of untouched cards of our shuffle has the same distribution as the number R_{2kt} of uncollected coupon types after $2kt$ steps of a coupon collector chain and that about $P^{kt}(id,\cdot)$ that the associate $F(\sigma)$ is at least as large as the number of cards that were touched by none of the transpositions, i.e. $P^{kt}(id, A) \geq P(R_{kt} \geq A)$.

We know that the R_{2kt} has expectation $\mu = np$ with $p = \left(1 - \frac{1}{n}\right)^{2kt}$, variance $var = np(1-p) < \mu$ and by Chebyshev, we know that $\Pr(R_{2kt} \leq \frac{\mu}{2}) \leq \frac{4}{\mu}$ as $\Pr(|R_{2kt} - \mu| \geq \frac{\mu}{2}) = \Pr(R_{2kt} \geq \frac{3\mu}{2}) + \Pr(R_{2kt} \leq \frac{\mu}{2}) > \Pr(R_{2kt} \leq \frac{\mu}{2})$.

By Markov's inequality we know that $\pi(A) \leq \frac{2}{\mu}$.

As $P^{kt}(id, A) \geq P(R_{kt} \geq A)$, we also have $P^{kt}(id, A^c) \leq P(R_{2kt} \leq A) \leq \frac{4}{\mu}$ which leads to $P^{kt}(id, A) \geq 1 - \frac{4}{\mu}$.

Thus we have $d(t) = ||P^{kt}(id,) - \pi||_{TV} \geq |1 - \frac{4}{\mu} - \frac{2}{\mu}| \geq 1 - \frac{6}{\mu}$.

We want to find the minimum t such that $1 - \frac{6}{\mu} \geq \epsilon$, which is equivalent to $n \cdot \left(1 - \frac{1}{n}\right)^{2kt} \geq \frac{6}{1-\epsilon}$ and to

$$\log\left(\frac{n \cdot (1-\epsilon)}{6}\right) \geq 2 \cdot k \cdot t \cdot \log\left(\frac{n}{n-1}\right)$$

As $\log(1+x) \leq x$, the previous inequality holds if $\log\left(\frac{n\cdot(1-\epsilon)}{6}\right) \geq \frac{2kt}{n-1}$ which means that if $t \leq \frac{n-1}{2k} \cdot \log\left(\frac{n(1-\epsilon)}{6}\right)$ then $d(t) \geq \epsilon$. Thus,

$$\tau(\epsilon) \geq \frac{n-1}{2k} \ln(n \cdot \frac{1-\epsilon}{6})$$

. □

10.2 Proof of Oblivious Merge

Proof. We want to find the mixing time $\tau(\epsilon)$ of our oblivious merge of two sets of indistinguishable elements. To do so, we use the bound of the mixing time of an irreducible ergodic Markov Chain, where $p = \frac{1}{|V|}$ and $1 - \lambda^*$ is the spectral gap,

$$\frac{\lambda^*}{1-\lambda^*} \cdot \log\left(\frac{1}{2\epsilon}\right) \leq \tau(\epsilon) \leq \frac{1}{1-\lambda^*} \cdot \log\left(\frac{1}{2\epsilon \cdot \sqrt{p}}\right)$$

We now want to find a bound for λ^*. We represent the arranging of merge of the 2 distinct sets by the graph $\mathcal{G}$, a k-regular graph with v vertices corresponding to the different orderings and the undirected edges to transpositions of two elements. By definition, the eigenvalues of the transition matrix of the $\mathcal{G}$ are $k = \lambda'_0 > \lambda'_1 \geq ... \geq \lambda'_{n-1}$, and we have,

$$\text{diam}(\mathcal{G}) \leq \frac{log(v-1)}{log(\frac{k}{\lambda'^*})} + 1 \text{ with } \lambda'^* = max_{i \neq 0}(\lambda'_i) = k \cdot \lambda^*$$

with $diam(\mathcal{G}) = s$ the diameter of the graph, $v = \binom{n}{s}$ the number of vertices and $k = s \cdot (n-s)$.

We can thus find a first relation:

$$log(\frac{k}{\lambda'^*}) = log(\frac{1}{\lambda^*}) \leq \frac{\log(v-1)}{\text{diam}(\mathcal{G}) - 1}$$

$$log(\lambda^*) \geq \frac{log(v-1)}{1 - \text{diam}(\mathcal{G})}$$

$$\lambda^* \geq (v-1)^{\frac{-1}{\text{diam}(\mathcal{G})-1}}$$

$$\lambda^* \geq \left(\binom{n}{s} - 1\right)^{\frac{1}{1-s}} \geq \left(\frac{n \cdot e}{s}\right)^{\frac{s}{1-s}}$$

And can derive the minimum value of $\Delta = \frac{\lambda^*}{1-\lambda^*}$,

$$\Delta = \frac{1}{(\lambda^*)^{-1} - 1}$$

$$\geq \frac{1}{\left(\frac{n\cdot e}{s}\right)^{\frac{s}{s-1}} - 1}$$

To find an upper-bound of λ^*, we will focus on the spectral gap bounding. Let's $\mathcal{G}_{0,1} = \{0,1\}^n$ be the group of elements with the XOR operation and $\mathcal{S} = \{x \in \mathcal{G},\ weight(x) = s\}$ the symmetric subset of $\mathcal{G}$ of n-binary array with s 1s and $n-s$ 0s. We call $Cay_{n,s} = Graph(\mathcal{G}_{0,1}, \mathcal{S})$ the Cayley graph generated from these structures.

Lemma 1. *Let $\mathcal{G}$ be a finite Abelian group, $\chi : \mathcal{G} \to \mathbb{C}$ be a character of $\mathcal{G}$, $\mathcal{S} \subseteq \mathcal{G}$ be a symmetric set. Let M be the normalized adjacency matrix of the Cayley graph $G = Cay(\mathcal{G}, \mathcal{S})$. Consider the vector $x \in \mathbb{C}^{\mathcal{G}}$ such that $x_a = \chi(a)$. Then x is an eigenvector of G, with eigenvalue*

$$\frac{1}{\mathcal{S}} \sum_{s \in \mathcal{S}} \chi(s)$$

Theorem 1. *The Cayley graph $Cay_{n,s}$ has for eigenvalues $\mu_0 = 1 > \mu_1 \geq ... \geq \mu_n$ with,*

$$\mu_r = \frac{1}{|\mathcal{S}|} \sum_{i=1}^{min(r,n-r)} (-1)^i \binom{r}{i}\binom{n-r}{s-i}$$

Proof. $\forall r \in \{0,1\}^n$, with $\chi_r(x) = (-1)^{\sum r_i \cdot x_i}$, we have,

$$\begin{aligned}
\mu_r &= \frac{1}{\mathcal{S}} \sum_{s\in\mathcal{S}} \chi(s) \\
&= \frac{1}{|\mathcal{S}|} \sum_{s\in\mathcal{S}} (-1)^{\sum r_i \cdot s_i} \\
&= \frac{1}{|\mathcal{S}|} \left(|\text{“}1\text{”}| - |\text{“}-1\text{”}|\right) \\
&= \frac{1}{|\mathcal{S}|} \sum_{i=1}^{min(r,s)} (-1)^i \binom{r}{i}\binom{n-r}{s-i} \\
&= 1 - \frac{2}{\binom{n}{s}} \cdot \sum_{i=0}^{min(\frac{r-1}{2},\frac{s-1}{2})} \binom{r}{1+2i}\binom{n-r}{s-2i-1} \\
&= \frac{\binom{n-r}{s}}{\binom{n}{s}}\ {}_2F_1(-r,-s,n-2r+1,-1)
\end{aligned}$$

■

Remark. We recognize here the Vandermonde identity with alternating numbers. We argue that the eigenvalues of the Cayley graph $Cay_{n,s}$ are all positive as the smallest eigenvalue is null. For $r = n-r$, the expression simplifies to $\mu_r = \binom{r}{\frac{n}{2}}$ if n even, 0 otherwise. For $r = 1$, the expression simplifies to $\mu_1 = 1 - 2 \cdot \frac{s}{n}$, the spectral gap of $Cay_{n,s}$ is thus equal to $2 \cdot \frac{s}{n}$.

We notice that the first graph $\mathcal{G}$ actually is a subgraph of $Cay_{n,s}$ and as such the the adjacent matrix of the first graph is included in the second's. For $s > 1$, $Cay_{n,s}$ is divided in two sub-graphs representing the cosets of $\{0,1\}^n$ as $\mathcal{S}$ is not a generating group of $\mathcal{G}_{0,1}$, $\mathcal{G}$ is only contained in one of the sub-graphs. We use the Cauchy's Interlace Theorem to bound the eigenvalues of $\mathcal{G}$ with the ones of $Cay_{n,s}$,.

Theorem 2. *Let M be a Hermitian $n \times n$ matrix with eigenvalues $\mu'_0 \geq ... \geq \mu'_{n-1}$ and N a $m \times m$ sub-matrix of M with eigenvalues $\lambda'_0 \geq ... \geq \lambda'_{m-1}$, we have*

$$\mu'_i \geq \lambda'_i \geq \mu'_{n-m+i+1}$$

We are here only interested in an upper-bound of $\lambda*$, as we have $\mu_{2^n+2-\binom{n}{s}} \leq \lambda_1 \leq 1 - 2\frac{s}{n}$ and $0 \leq \lambda_n \leq \mu_2$, $\lambda* \leq 1 - 2\frac{s}{n}$. We thus have $\frac{1}{1-\lambda*} \leq \frac{n}{2\cdot s}$ □

10.3 Proof of Parallel mix-net

Proof. This proof is derived from Goodrich and Mitzenmacher [20] who bounded the closeness of a shuffle to the uniform distribution using a compromised parallel mix-net.

Let $\Phi(t) = \Sigma_{i=1}^n (w_i(t) - 1/n)^2$ the sum of square metric with $w_i(t)$ the probability that the i^{th} card after t rounds is the first card from time 0 from the point of view of the adversary.

We have by recurrence that the potential $\Delta\Phi^*$ changes when a group of K card is shuffled during a round as following : $\Delta\Phi^* = \Sigma_{1\leq i\leq n}(w_i - w_j)^2$. Thereby,

$$\begin{aligned}
E[\Delta\Phi] &= \frac{m}{n}\Sigma_{1\leq i\leq n} \Pr((i,j) \textit{ in the same honest mix})(w_i - w_j)^2 \\
&= \frac{k-1}{k(n-1)} \cdot \frac{m-m_a}{m}\Sigma_{i<j}(w_i - w_j)^2 \\
&= \frac{(m-m_a)(k-1)}{2n(n-1)}\Sigma_{1\leq i\leq n}(w_i - w_j)^2
\end{aligned}$$

$$\begin{aligned}
E[\Delta\Phi/\Phi] &= \frac{(m-m_a)(k-1)}{2n(n-1)} \frac{\Sigma_{i,j}((w_i - 1/n) - (w_j - 1/n))^2}{\Sigma_k (w_k - 1/n)^2} \\
&= \frac{(m-m_a)(k-1)}{2n(n-1)} \frac{\Sigma_{i,j}(x_i - x_j)^2}{\Sigma_k x_k^2} \text{ with } x_i = w_i - 1 \\
&= \frac{(m-m_a)(k-1)}{n-1}
\end{aligned}$$

Since $\Sigma_k x_k = 0$, we have :

$$\Sigma_{i,j}(x_i - x_j)^2 = \Sigma_{i,j}(x_i - x_j)^2 + 2\left(\Sigma_k x_k^2\right)^2 = 2n\Sigma_k x_k^2$$

We have,

$$E[\Phi(t+1)] = (1 - \frac{(m-m_a)(k-1)}{n-1})E[\Phi(t)]$$
$$E[\Phi(t)] = (1 - \frac{(m-m_a)(k-1)}{n-1})^t$$

We want to find the conditions on c such that the corrupted parallel mix-net can mix in $t = bc\log(n)$ such that $n^{-2b} < E[\Phi(t)] < n^{-b}$.

$$\begin{aligned}
E[\Phi(t)] &< n^{-b} \\
(1 - \frac{(m-m_a)(k-1)}{n-1})^t &< n^{-b} \\
(1 - \frac{(m-m_a)(k-1)}{n-1})^{bc\log(n)} &< n^{-b} \\
bc\log(n) \cdot \log(1 - \frac{(m-m_a)(k-1)}{n-1}) &< -b\log(n) \\
c \cdot \log(1 - \frac{(m-m_a)(k-1)}{n-1}) &< -1 \\
c \cdot \log(1 + \frac{1}{\frac{n-1}{(m-m_a)(k-1)} - 1}) &> 1
\end{aligned}$$

Using Taylor series, assuming that $n-1 >> (m-m_a)(k-1)$, let $y = \frac{n-1}{(m-m_a)(k-1)} - 1$:

$$
\begin{aligned}
c &> 1/\log(1+\frac{1}{y}) \\
c &> \left(\frac{1}{y} + o\left(\frac{1}{y}\right)\right)^{-1} \\
c &> y - o\left(1\right) \simeq \frac{m}{m-m_a} - o(1)
\end{aligned}
$$

Thus, when shuffling n cards with n a parallel mix-net composed of m mixes out of which m_a were compromised, we need $t > b \cdot \frac{m}{m-m_a}\log(n)$ rounds before the expected sum of squares error $E[\Phi(t)]$ between the card assignment probabilities and the uniform distribution is at most $1/n^b$ for any fixed $b > 1$. □

10.4 Proof of Fake access

Proof. We want to prove that the average number of fake access is 0 in case of a uniform distribution. To do so, we consider the Markov chain and its Transition Matrix. The transition matrix P represents the s transient state, in which the stash is not completely filled, and the absorption state in which the stash is full. Thus, P can be decomposed in 4 sub-matrices: the square sub-matrix Q_s representing all the transient state, the column matrix R with the probabilities of transitioning to the absorbing state, the null row matrix and the absorption matrix.

$$\begin{bmatrix} Q_s & R \\ 0_{1\times s} & I_1 \end{bmatrix}$$

To find the average number of steps from one state to the absorbing one, we have solve the following equation, each row corresponding to the average number of steps from the corresponding state (the stashed filled with some records) to the state where the stash is full.

$$
\begin{aligned}
t &= \left(\sum_{k=0}^{\infty} {Q_s}^k\right) 1 \\
&= (I_s - Q_s)^{-1} 1
\end{aligned}
$$

This equation has a solution since $M = I_s - Q_s$ have independent rows and thus an inverse that we call N. By calculus we find that,

$$
\begin{aligned}
n_{i,j} &= 0 && \text{if } i > j, \\
n_{i,j} &= \frac{1}{m_{i,i}} && \text{if } i = j, \\
n_{i,j} &= -n_{i+1,j} \cdot \frac{m_{i,i+1}}{m_{i,i}} && \text{if } i < j
\end{aligned}
$$

which can be simplified by

$$
\begin{aligned}
n_{i,j} &= 0 && \text{if } i > j, \\
n_{i,j} &= \frac{1}{m_{j,j}} \prod_{k=1}^{j-1} \left(-\frac{m_{i,k+1}}{m_{k,k}}\right) && \text{if } i \leq j
\end{aligned}
$$

We only want to calculate the first solution S_1 from the equation.

$$
\begin{aligned}
S_1 &= \sum_{j=0}^{s-1} \frac{1}{m_{j,j}} \prod_{k=1}^{j-1} \left(-\frac{m_{i,k+1}}{m_{k,k}}\right) \\
&= \sum_{j=1}^{s-1} \frac{1}{m_{j,j}} \text{ as } m_{i,k+1} = -m_{k,k} \\
&= \sum_{j=0}^{s-1} \frac{1}{1-\frac{j}{n}} = \sum_{j=0}^{s-1} \left(1 + \frac{j}{n-j}\right) \\
&= s + \sum_{j=0}^{s-1} \frac{j}{n-j}
\end{aligned}
$$

As s steps are required to fill the stash, we thus find the following inequality for the number of fake access f:

$$\frac{s\cdot(s+1)}{2\cdot n} < f < \frac{s\cdot(s+1)}{2\cdot(n+1-s)}$$

□