Dog ORAM: A Distributed and Shared Oblivious RAM Model with Server Side Computation

Alexandre Pujol
University College Dublin
Belfield, Dublin 4, Ireland
Email: alexandre@pujol.io

Christina Thorpe
University College Dublin
Belfield, Dublin 4, Ireland
Email: christina.thorpe@ucd.ie

***Abstract*—Outsourcing to the Cloud is becoming an attractive option for many organisations dealing with large amounts of data. However, there is still a reluctance amongst companies dealing with highly regulated data because traditional Cloud storage does not support the level of privacy required to prevent access pattern leakage. Oblivious Random Access Machines (ORAM) have been a hot topic of research over the past number of years, proposing various cryptographic techniques to obtain the privacy levels required. We propose a new model, Dog ORAM — a distributed and shared oblivious RAM model with server side computation, that merges several models existing in the literature and includes a new method of access right management for multi-party data access. To achieve this, we use an additive homomorphic encryption scheme and a chameleon signature.**

***Keywords*—Oblivious RAM, Shared ORAM, Distributed ORAM, Access right management, AHE, Chameleon signature**

I. Introduction

In recent years the computing landscape has changed significantly; consumers are becoming increasingly interested in outsourcing their computing resources. The Cloud is becoming a hugely popular choice for organisations wishing to host products or store data, however, traditional Cloud environments do not provide the level of privacy necessary to store highly regulated data. Simply encrypting medical or financial data is not sufficient for compliance to data privacy laws. Encryption alone cannot prevent leakage of access patterns, which can be used to infer sensitive information.

For example, consider a Cloud solution that is used to host hospital patient records. Patient privacy is a legal obligation, thus records are encrypted to maintain the confidentiality of the data. Unfortunately, without accessing the actual plain text data, a careful adversary can still learn sensitive information from where, when and how often a client accesses their data. In this case, if the Cloud provider can see that an oncologist has accessed my records, they can infer that I have cancer, regardless of whether they can decrypt the actual records or not. The most dangerous aspect of these types of attacks is that they are cumulative. An adversary may learn only a small amount from any one access, but over time they can aggregate everything that they have discovered, with any side knowledge of the client they might have, to reveal a surprising amount of sensitive information. The dangers of access pattern leakage have been well documented in [1], where authors show that attacks are able to disclose sensitive information with a very high accuracy.

Oblivious Random Access Machine (ORAM) is a cryptographic construction that allows clients to access encrypted data residing on an untrusted storage server, while completely hiding the access patterns to storage. Particularly, the sequence of physical addresses accessed is independent of the actual data that the user is accessing. To achieve this, existing ORAM constructions continuously download, re-encrypt, reshuffle and re-upload data blocks on the storage server, to cryptographically conceal the logical access pattern [2].

In 1996 Goldreich and Ostrovsky [3] first introduced the notion of oblivious RAM. However, traditional ORAM schemes are limited in terms of complexity; in more recent years, the goal of most research contributions was to decrease the complexity of the model in order to make it practical in real life [4]. Thus in order to speed up and simplify the ORAM schemes, the structure of most server models changed to tree based models where the server structure is a binary tree [5], [6].

Given that executing all computation on the client side is not efficient and since the Cloud has the advantage of providing computation resources in addition to storage, ORAM with server side computation was proposed [7]. This model uses a fully homomorphic encryption scheme, resulting in the computation shrinking from logarithmic in client-side to constant in server-side computation model. However, server-side computation suffered from significant communication overhead. Onion ORAM [8] is a layered encryption approach that aims to reduce the communication overhead incurred in the oblivious storage process using bandwidth-efficient additive homomorphic encryption schemes. For instance, Onion ORAM [8] reduces the communication complexity to $O(1)$ breaking the lower bound established by [3]. One key benefit of remote access is the ability to share data amongst multiple users. The concept of group access in ORAM introduces some additional concerns; new parameters such as Access Rights (AR) management must be considered. Group ORAM [9] is a very recent proposal that is still in its infancy, one of its new concept is that both clients and servers could be malicious.

The main contribution in this paper is to propose a new model that integrates some concepts previously published in the literature. Distributed Onion Group ORAM (Dog ORAM) is the first ORAM scheme with server side computation, distributed in different servers and shared between a set of clients. It is a modified Onion ORAM [8] model that uses a Path ORAM [6] structure. The multi-user structure of Dog ORAM has been inspired by the Group ORAM model [9].

Finally, we added a distributed model to increase the security of the scheme.

The remainder of the paper is organised as follows: section II presents the security model defined to set the security objectives of the Dog ORAM scheme. Section III provides low-level technical details of algorithms used in the scheme. Section IV presents Dog ORAM and describes the critical points of our proposal. Finally, section V concludes the paper and presents some future work.

II. Security Model

A. Definitions

Honest but Curious Server (HbC) (Passive): The server is regarded as a passive adversary following the scheme specifications correctly but seeking to gather additional information about the client's data and access patterns.

Malicious Server/Client (Active): Server/client which can deviate arbitrarily from the scheme specifications.

B. Attacker Model

Our adversarial model draws from assumptions common in the ORAM literature, which are realistic:

- The data owner ($\mathcal{O}$) is honest.
- The clients $(C_1, ..., C_c)$ may be malicious.
- The authentication server S_{auth} is trusted for its purpose.
- The servers $(S_1, ..., S_s)$ are assumed to be honest-but-curious and do not collude with any clients.

C. Security Properties

The security objectives for this work are (inspired by [9]):

- **Secrecy:** A client can only read entries it holds read permissions on.
- **Integrity:** A client can only write entries it holds write permissions on.
- **Obliviousness:** A server cannot determine:
  1) What data is being accessed,
  2) How old the data is,
  3) If the same data has been accessed multiple times,
  4) The access pattern (sequential, random, etc.),
  5) Whether the access is a read or a write.
- **Anonymity:** A server and the data owner cannot determine who performed a given operation among the set of clients that are allowed to perform it.

III. Technical Background

A. Cryptographic Preliminaries

A probabilistic asymmetric additive homomorphic encryption scheme (see part III-B) is denoted by $AHE = (Gen_{AHE}, \mathcal{E}, \mathcal{D})$, where Gen_{AHE} is the key-generation algorithm and $\mathcal{E}$ (resp. $\mathcal{D}$) is the encryption (resp. decryption) algorithm. In this paper, the public key is shared with all the participants (clients and servers) and the secret key is shared with all the clients.

A chameleon hash scheme $CHF = (Gen_{CHF}, CH, Col)$ is used to generate a modifiable signature (see [10] and [11]). The setup algorithm Gen_{CHF} generates a key pair (cpk, csk), where cpk is the public key and csk is the secret key. The chameleon hash function (1) takes the public key cpk, a message m and a random number r_{CH} as input. It gives a hash tag t. The collision function (2) takes a secret key csk, a message m, a random number r_{CH} and another message m'. It gives a new random number r'_{CH} such that (3).

$$CH(cpk, m, r_{CH}) = t \tag{1}$$
$$Col(csk, m, r_{CH}, m') = r'_{CH} \tag{2}$$
$$CH(cpk, m, r_{CH}) = CH(cpk, m', r'_{CH}) \tag{3}$$

A signature scheme $\mathcal{S} = (Gen_{SIGN}, Sign, Verify)$ consist of a key-generation, a signature and a verification algorithm.

B. Additive Homomorphic Encryption

The Dog ORAM scheme uses an additive homomorphic encryption scheme (AHE). The following definitions are taken from [12]. We define:

- $\mathcal{E}$ Probabilistic asymmetric encryption operation.
- $\oplus$ Additive operation between two ciphertexts.
- $+$ Additive operation between two plaintexts.
- $\odot$ Scalar multiplication between a ciphertext and a plaintext.
- $\cdot$ Multiplication between two plaintexts.

Let x and y be two plaintexts. (4) is the additive homomorphic definition and (5) is a homomorphic scalar multiplication.

$$\mathcal{E}(x) \oplus \mathcal{E}(y) = \mathcal{E}(x + y) \tag{4}$$
$$\mathcal{E}(x) \odot y = \mathcal{E}(x \cdot y) \tag{5}$$

We define a sequence of additively homomorphic encryption schemes $\mathcal{E}_l$. Let $\mathbb{L}_l$ be the plaintext space and $\mathbb{L}_{l+1}$ the ciphertext space of $\mathcal{E}_l$. $\mathbb{L}_{l+1}$ is again the plaintext space of $\mathcal{E}_{l+1}$. Equation 6 denotes an l-layer encryption of a plaintext x:

$$\mathcal{E}^l(x) = \mathcal{E}_l(\mathcal{E}_{l-1}(...\mathcal{E}_1(x))) \tag{6}$$

A scalar multiplication between a plaintext in $\mathbb{L}_l$ and a ciphertext in $\mathbb{L}_{l+1}$ gives:

$$\mathcal{E}_{l+1}(x) \odot \mathcal{E}_l(y) = \mathcal{E}_{l+1}(x \odot \mathcal{E}_l(y)) \tag{7}$$

Applying (7) with $x = 0$ or $x = 1$ allows us to create a homomorphic selection:

$$\mathcal{E}_{l+1}(0) \odot \mathcal{E}_l(y) = \mathcal{E}_{l+1}(0)$$
$$\mathcal{E}_{l+1}(1) \odot \mathcal{E}_l(y) = \mathcal{E}_{l+1}(\mathcal{E}_l(y))$$

C. Double Homomorphic Select Operation

1) Preliminaries: In the Onion ORAM model [8], a homomorphic select operation is used in order to obliviously access a data block on a server. In this paper we build on [8], but we create a double selection. Let x, y and z be three plaintexts. Let the double homomorphic select operation (SO) be:

$$\begin{aligned} SO(x,y,z) &= \mathcal{E}_{l+2}(x) \odot (\mathcal{E}_{l+1}(y) \odot \mathcal{E}_l(z)) \\ &= \mathcal{E}_{l+2}(x) \odot \mathcal{E}_{l+1}(y \odot \mathcal{E}_l(z)) \\ &= \mathcal{E}_{l+2}(x \odot \mathcal{E}_{l+1}(y \odot \mathcal{E}_l(z))) \\ &= \begin{cases} 0 & \text{if x = 0} \\ \mathcal{E}_{l+2}(\mathcal{E}_{l+1}(0)) & \text{if x = 1 and y = 0} \\ \mathcal{E}_{l+2}(\mathcal{E}_{l+1}(\mathcal{E}_l(z))) & \text{if x = y = 1} \end{cases} \end{aligned}$$

x indicates whether the block z is selected or not. y gives the access right on z. Therefore if a block is selected ($x = 1$) and a client has the right to access it ($y = 1$), it will retrieve a triple encryption of this block. Otherwise it will get nothing or the double encryption of 0 if it has not the access right.

2) Server wide application: The double selection is applied on the server. The server stores n blocks of data. All these blocks have l-layers of encryption $\mathcal{E}^l(m_i)$. Where m_i, $1 \leq i \leq n$ is the plain part of a block. The client wants to retrieve a specific plain block $m_j, 1 \leq j \leq n$. The server receives:

- A *selection vector* $b = (\mathcal{E}_{l+2}(b_1), ..., \mathcal{E}_{l+2}(b_n))$. With $1 \leq i \leq n$:

$$b_i = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise} \end{cases}$$

- An *access right vector* $ar_k = (\mathcal{E}_{l+1}(p_1), ..., \mathcal{E}_{l+1}(p_n))$. Where k is the client id and p_i with $1 \leq i \leq n$ denotes the client's permission on the block i:

$$p_i = \begin{cases} 1 & \text{if the client } k \text{ can access the data,} \\ 0 & \text{otherwise} \end{cases}$$

The exact content of the vector ar_k is given in part IV-C.

The server performs the double selection operation SO on his n blocks with the selection vector b from the client k:

$$\begin{aligned} SO(b, ar_k) &= \bigoplus_{i=0}^{n} \mathcal{E}_{l+2}(b_i) \odot \mathcal{E}_{l+1}(p_i) \odot \mathcal{E}^l(m_i) \\ &= \bigoplus_{i=0}^{n} \mathcal{E}_{l+2}(b_i \odot \mathcal{E}_{l+1}(p_i \odot \mathcal{E}^l(m_i))) \\ &= \mathcal{E}_{l+2}\left(\sum_{i=0}^{n} b_i \odot \mathcal{E}_{l+1}(p_i \odot \mathcal{E}^l(m_i))\right) \\ &= \mathcal{E}_{l+2}(\mathcal{E}_{l+1}(p_j \odot \mathcal{E}^l(m_j))) \\ &= \begin{cases} \mathcal{E}_{l+2}(\mathcal{E}_{l+1}(0)) & \text{if } p_j = 0, \\ \mathcal{E}^{l+2}(m_j) & \text{otherwise} \end{cases} \end{aligned}$$

Therefore, if the client has the proper AR, it retrieves a $l+2$-layer encryption of the data they requested. Otherwise it gets a double encryption of its AR. But in every case, the server sends some data to the client, and has no way to know its permission.

In other words, the server is able to perform read/write operations on a data block for a client while controlling the client's AR without knowing neither the block id nor the clients rights on this very block.

3) Remarks: The following remarks are crucial for the efficiency and the security of the Dog ORAM scheme.

- This double homomorphic select operation is close to the simple select operation in the Onion ORAM scheme [8]. Therefore as in the Onion ORAM model, we have to ensure that the layers of onion encryption are bounded.
- If $b_i = 0$, the value of p_i does not matter. Usually, in order to quickly generate ar_k a dummy random value can be inserted in p_i for $i \neq j$, however, this is not possible within the scope of this paper (see part IV-C1).
- It is essential to ensure that the method used to get the AR vector does not leak any information to the server and that a client is not able to change this vector. This is detailed in section IV-C.

IV. Dog ORAM

Our goal is to create a secure and practical cloud storage model, therefore, we need to construct an ORAM scheme that respects the classic remote storage requirements. There are many challenges to overcome when attempting to merge classic remote storage requirements with the existing ORAM schemes:

1) **Fast:** All the existing ORAM schemes are slow, but we can optimise it by allowing server side computation.
2) **Shared:** There are only a few existing shared ORAM models, but none of them use server side computation. However, shared access is mandatory for practical Cloud storage.
3) **Distributed:** There is no existing distributed ORAM scheme. The two interests of an oblivious distributed scheme are both to speed up the ORAM scheme with server side computation and to increase the security of the scheme.

Table I summarizes our contributions and compares our scheme with some of the state-of-the-art ORAM constructions. Dog ORAM has several benefits over prior work, stemming from the management of access right in a server side computation model. Therefore, the Dog ORAM has the same big-O complexity with AR management as without. In table I, N, B and c represents the number of data blocks, the size of a block and the number of client respectively.

TABLE I. ORAM SCHEME COMPARISON

Scheme	Bandwidth Cost	Client Storage	Server Computation
Path ORAM [6]	$O(B \log N)$	$O(B\lambda)$	N/A
Group ORAM [9]	$O((B+c) \log N)$	$O(B \log N)$	N/A
Onion ORAM [8]	$O(B)$	$O(B)$	$O(B \log N)$
Dog ORAM	$O(B)$	$O(B)$	$O(B \log N)$

Figure 1 shows the general architecture of the model. The Dog ORAM scheme can be summarized as follows:

- A set of s independent servers $S_i, 1 \leq i \leq s$.
- An authentication server S_{auth}.
- A group of c clients $C_i, 1 \leq i \leq c$.
- The data owner $\mathcal{O}$.
- A set of algorithms (see part IV-D).

A server runs like an independent ORAM structure, thus a server S_i is not aware that it is part of a bigger set of servers.

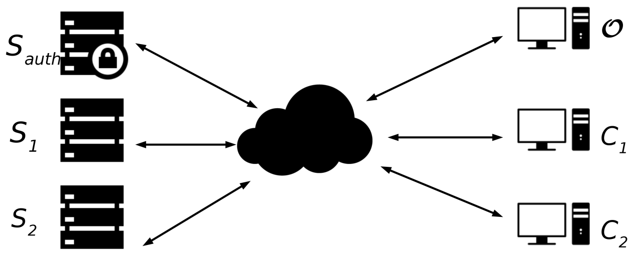

Fig. 1. Dog ORAM: General Setup

A. Server Structure

The server structure proposed is the same as in the Onion ORAM model. The data is stored in a binary tree of $L + 1$ levels (see figure 2). Each node is called a bucket. Each bucket contains Z blocks of data (real or dummy). The leaves are numbered from 0 to $2^L - 1$. Each block is associated with a random path from the root to a leaf l: $\mathcal{P}(l)$. All notations used throughout the rest of the paper are summarized in table II.

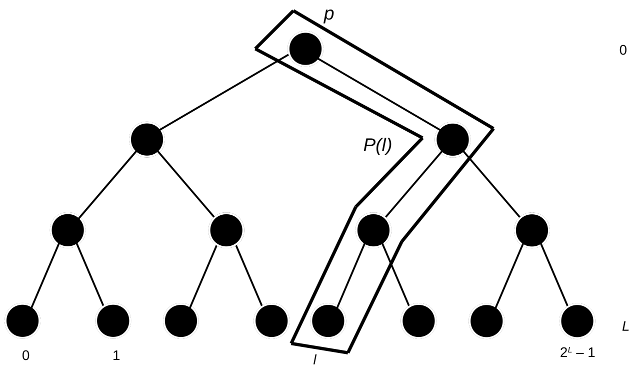

Fig. 2. Illustration of the tree ORAM

TABLE II. Dog ORAM parameters and notations.

Notation	Meaning
1^λ	Security parameter
n or N	Number of data blocks
c	Number of clients
s	Number of servers
b	Homomorphic selection vector
k	Secret client Id
ar_k	Homomorphic access right vector for the client k
p_i	Permission for the block i
L	Depth of the Dog ORAM tree
A	Eviction rate (larger means less frequent)
$EvictionCNT$	Counter for when a path needs to be evicted
$LeafCNT$	Counter for which path needs to be evicted
$\mathcal{P}(l)$	Path from the root to the leaf l
$\mathcal{P}(l, i)$	The i-th bucket (from the root) on $\mathcal{P}(l)$

B. Metadata

Metadata is a critical feature of all ORAM models; metadata allows the operations on the remote blocks and the management of the users' rights. The Dog ORAM scheme uses three matrices for metadata.

PositionMap: This is a matrix linking the virtual address a of a block to its assigned leaf on the server. $PositionMap[a]$ gives a leaf l. The block at the address a can be found in the server in the path $\mathcal{P}(l)$.

ServerMap: This is a matrix linking the virtual address a of a block to its assigned server. $ServerMap[a]$ gives a server number, thus it is used with the *PositionMap* matrix. There is a position map per server and a server map for all the system. In order to be shared, the server map has to be stored on a server.

AccessMap: This is a matrix linking a vector of virtual addresses $\vec{a}$, a client id k, a mode $mode$ and a layer of encryption l to an access right vector $ar_{k,\vec{a}}$: $AccessMap[\vec{a}, k, mode, l] = ar_{k,\vec{a}}$. IV-C describes its use.

To avoid incurring a large amount of client storage and in order to be able to share the metadata between the different users, all the metadata is stored in another smaller ORAM (except *AccessMap*). Thus, all the servers have a main ORAM structure and a smaller ORAM for the metadata. Similar to the Onion ORAM model, we assume that the (recursed) position map does not introduce extra asymptotic cost in terms of server storage or bandwidth.

C. Access Rights (AR) Management

1) Discussion: With the knowledge of the AR vector ar_k, the server is able to control the user's permissions without knowing anything about the user or its permissions. But, in order to maintain that security feature, we need to be able to generate, manage and send that vector to the server in a way that does not leak any information. It is a crucial problem because it means that:

- The client must not be able to change ar_k.
- The server must not know that ar_k is from the client k.
- The server must not know the access mode (read or write) paired to the vector ar_k.
- Only the data owner $\mathcal{O}$ is allowed to change the permissions.
- An access right vector cannot be used with an other list of virtual addresses than the one it is paired with.

Thus, even if only the permission of the selected block is important, we cannot put dummy values in the other permissions because the server must still be able to check the AR authenticity. Therefore all the permissions in ar_k must be valid.

2) Proposed Solution: All the permissions p_i for all the blocks $i, 1 \leq i \leq n$ are encrypted with the AHE scheme. An AR is denoted $\mathcal{E}(p_i)$. The data owner has sole management of the AR, and thus must sign them. The AR are stored in the *AccessMap* matrix, which is itself stored in the authentication server S_{auth}. S_{auth} is a trusted server for the authentication i.e., when request an AR vector for a list of addresses, it will return the correct list. In this way, a client can only use an AR vector with the list of virtual addresses that it is paired with.

Furthermore, in order to prevent permission replay from a client (if its permission changed), a unique random number r is appended to the AR: $\mathcal{E}_l(p_i)\|r$. Therefore, the AR used are updated (after each operation) with a new unique random number r' so the clients will not be able to reuse them. However, the owner's signature must be updated, but the owner may not always be available to do that. Since we consider the HbC server model, the server can be used to do this. Using a *chameleon signature*, the server has the ability to update the access permissions from $\mathcal{E}_l(p_i)\|r$ to $\mathcal{E}_l(p_i)\|r'$ while keeping a valid signature from the owner. As the permissions are encrypted, the server itself does not know the value it re-signs.

The server controls the permissions and sends the data to the allowed user. Since the data could be decrypted by any users, we apply a classic encryption scheme on the data in addition to our current strategy.

3) Chameleon Signature: The chameleon hash scheme is a hash scheme with a trapdoor. The knowledge of that trapdoor (a secret key csk) allows us to generate hash collisions (see (2) and (3)). Thus, in a signature scheme, a server with the chameleon secret key csk can modify the message without invalidating the signature. The complete chameleon signature scheme used in this paper is:

- **Generation:** The owner creates the AR for the mode $mode$ on the block i for the user k (8). It hashes the new AR with a random element r_{CH} (9). It signs the hash tag t with its private key $sk_{\mathcal{O}}$ (10) and writes the new AR in the Access Map (11).

$$\mathcal{E}_l(p_i)\|r \tag{8}$$
$$CH(cpk, \mathcal{E}_l(p_i)\|r, r_{CH}) = t \tag{9}$$
$$Sign(sk_{\mathcal{O}}, t) = \sigma \tag{10}$$
$$AccessMap[i, k, mode, l] = (\mathcal{E}_l(p_i)\|r, r, r_{CH}, t, \sigma) \tag{11}$$

- **Signature Update:** The server knows the trapdoor csk, it generates a new random number r', computes the collision function (12) and updates the AR (13).

$$Col(csk, \mathcal{E}_l(p_i)\|r, r_{CH}, \mathcal{E}_l(p_i)\|r') = r'_{CH} \tag{12}$$
$$AccessMap[i, k, mode, l] = (\mathcal{E}_l(p_i)\|r', r', r'_{CH}, t, \sigma) \tag{13}$$

- **Verification:** The server verifies the hash tag (14) and the signature (15).

$$CH(cpk, \mathcal{E}_l(p_i)\|r', r'_{CH}) == t \tag{14}$$
$$Verify(sk_{\mathcal{O}}, t, \sigma) == true \tag{15}$$

4) Authentication Server: The method for retrieving the AR vector for a list of virtual addresses is the following:

- A client k asks S_{auth} for an AR vector ($ar_{k,\vec{a}}$) for the addresses list $\vec{a}$.
- S_{auth} sends to the client: $(ar_{k,\vec{a}}\|\vec{a}, \sigma_{ar_{k,\vec{a}}})$ where $\sigma_{ar_{k,\vec{a}}}$ is the signature of $ar_{k,\vec{a}}\|\vec{a}$.

Then, an ORAM server can verify that the AR vector is signed by the authentication server.

D. Algorithms Description

In this section we describe the Dog ORAM algorithms. The *Access* and *Eviction* algorithms are both very similar to their counterparts in the Onion ORAM scheme. The changes are due to the multiuser support. The Dog ORAM scheme is distributed among s independent servers, each with its own keys, thus the algorithms 1 and 2 must be executed for all the servers.

Algorithm 1 $((sk_{\mathcal{O}}, pk_{\mathcal{O}}); (cpk, spk)) \leftarrow Init(1^\lambda, c)$

```
Input: Security parameter $1^\lambda$, number of clients $c$
Output: AHE, Signature and chameleon hash keys pair
1: $(hpk, hsk) \leftarrow Gen_{AHE}(1^\lambda)$
2: $(cpk, csk) \leftarrow Gen_{CHF}(1^\lambda)$
3: $(pk_{\mathcal{O}}, sk_{\mathcal{O}}) \leftarrow Gen_{SIGN}(1^\lambda)$
4: give $hpk$, $cpk$, $csk$ and $pk_{\mathcal{O}}$ to the server S
5: share $(hpk, hsk)$ with the users
6: initialize access right vector for $c$ users
7: return $((sk_{\mathcal{O}}, pk_{\mathcal{O}}); (cpk, spk))$
```

Algorithm 2 $k \leftarrow addClient(p_{write}, p_{read})$

```
Input: Permissions vectors for all the blocks in the servers
Output: New secret client id $k$
1: $k \leftarrow Random(1^\lambda)$
2: for all blocks in the server do
3:     generate AR
4:     append to $AccessMap$
5: return $k$
```

Initiation: The initiation algorithm 1 generates the homomorphic key pair and the chameleon signature public and private keys for a specific server.

Add a Client: In algorithm 2, the client id must be kept secret from the other participants. A client must not be able to find the id of another client. Otherwise, it could be able to use other client's access permissions. This is why the client id is a random number. However, it is not a perfect solution. See part V.

Add a block: Only the owner is able to add a new block (algorithm 3) and to generate AR for all the users.

Chmod: One of the advantages of our model is that we can change the access right very quickly because the owner does not have to access the block.

Homomorphic Selection: The selection algorithm 4 has been fully detailed in part III-C. The selection is executed on the block on the path l. The client needs to generate the selection vector b with the correct layer of encryption l. In the same way, it has to get an AR vector with the correct layer of encryption from the *AccessMap* matrix.

Eviction Algorithm: The eviction algorithm ensures that the root bucket is not full. Dog ORAM performs the eviction on a path l_{evict}. For every bucket on the path its blocks are

Algorithm 3 $addBlock(a, data, p)$

```
Input: virtual address $a$, block of data $data$, permission vector
    $p$ for all users in all modes
1: $l \leftarrow Random(0, 2^L - 1)$
2: $PositionMap[a] \leftarrow l$
3: for all clients do
4:     generate access right for the block $a$
5:     append to $AccessMap$
6: Create an empty selection vector $\vec{O}$
7: Get owner access rights $ar_{\mathcal{O},\vec{O}}$ and $\sigma_{ar_{\mathcal{O},\vec{O}}}$ from $S_{auth}$
8: $\perp \leftarrow Selection(\vec{O}, ar_{\mathcal{O},\vec{O}}, \sigma_{ar_{\mathcal{O},\vec{O}}})$  ▷ Simulate an access
9: $\mathcal{P}(l, 0) \leftarrow data$                ▷ Write $data$ on the root bucket.
10: Check if an eviction is needed               ▷ See algorithm 5
```

Algorithm 4 $\{data, \perp\} \leftarrow Selection(b, ar_{k,\vec{a}}, \sigma_{ar_{k,\vec{a}}})$

```
Input: selection vector $b$ on a path and access vector $ar_{k,\vec{a}}$, access vector signature $\sigma_{ar_{k,\vec{a}}}$
Output: data block in case of success, $\perp$ otherwise
1: if $Verify(sk_{S_{auth}}, ar_{k,\vec{a}} \| \vec{a}, \sigma_{ar_{k,\vec{a}}}) == \perp$ then
2:     return $\perp$
3: for all elements in $ar_{k,\vec{a}}$ do
4:     if Chameleon Signature not verified then
5:         return $\perp$
6: Update access permissions for all the blocks on the path and for all the users
7: return $SO(b, ar_{k,\vec{a}})$
```

Algorithm 5 $Access(a, k, mode, data')$

```
Input: Address of a block $a$, client secret id $k$, access mode $mode$, new data $data'$
1: $l \leftarrow PositionMap[a]$
2: Retrieve $\vec{a}$ addresses of all blocks in $\mathcal{P}(l)$ from the server
3: Find $a$ in these addresses
4: Create selection vector $b$        ▷ For the blocks of $\mathcal{P}(l)$
5: Retrieve $(ar_{k,\vec{a}} \| \vec{a}, \sigma_{ar_{k,\vec{a}}})$ from $S_{auth}$
6: $data \leftarrow Selection(b, ar_{k,\vec{a}}, \sigma_{ar_{k,\vec{a}}})$        ▷ algorithm 4
7: if $|data| == |\mathcal{E}_{l+2}(0)|$ then
8:     return $\perp$        ▷ the user has not the good permission, stop the algorithm
9: if $mode ==$ read then
10:     return $data$ to client
11: if $mode ==$ write then
12:     $data \leftarrow data'$
13: $PositionMap[a] \leftarrow Rand(1, 2^L - 1)$
14: $\mathcal{P}(l, 0) \leftarrow data$        ▷ Write $data$ on the root bucket.
15: $EvictionCNT \leftarrow EvictionCNT + 1 \mod A$
16: if $EvictionCNT == 0$ then
17:     $l_{evict} \leftarrow ReverseDigits(LeafCNT)$
18:     $LeafCNT \leftarrow LeafCNT + 1 \mod (2^L - 1)$
19:     $Eviction(l_{evict})$
```

moved into its two children nodes. It is the same eviction algorithm as Onion ORAM. The double selection is not needed, the simple selection is sufficient because the eviction does not depend on the client AR.

Access: The access algorithm 5 allows a user to read or write a data block. After each access to a block, this block is removed from its position and added to the root bucket. However if a user does not have the permission to access the block, we need to ensure that it is not overwritten. After A accesses the eviction algorithm is called in order to prevent an overflow of the root bucket.

V. Conclusion

This paper presents the Dog ORAM model for data privacy in the Cloud. The key differentiator from existing schemes is the novel method of controlling the AR: using a double homomorphic selection on a server to obliviously control the permissions. Additionally, a chameleon signature scheme is used to authenticate AR, even if they change after every access. The proposed access permission server control means that no other system is needed to detect an unauthorized access. The distributed part of the Dog ORAM model is still in its infancy, however the very simple distributed structure of the model already added an effective layer of security i.e., distributing across multiple servers means increasing the difficulty for an attacker.

The most critical feature of the Dog ORAM model is the method of retrieving the AR vector from a user. Future work will focus on improving and adapting it in order to apply it to a fully malicious attacker model. The current client id secret system is not fully secure; given sufficient time, an attacker could retrieve some client id, thus, a better solution is needed. Dog ORAM operates in the 'honest but curious' server attacker model. Similar to the Onion model [8], probabilistic checking and an error-correcting code could be used to work with a malicious attacker model. Additionally, more work is needed to develop the re-encryption step. Finally, in the first version of Dog ORAM, we achieved our goals of creating a distributed, shared model with server side computation. Our next steps will be to work on the complexity of the model and to create a verifiable eviction algorithm.

Acknowledgment

This work was supported, in part, by Science Foundation Ireland grant 10CEI1855 to Lero — the Irish Software Research Centre (www.lero.ie) and in part by Science Foundation Ireland grant 13RC2094 and co-funded under the European Regional Development Fund through the Southern & Eastern Regional Operational Programme to Lero — the Irish Software Research Centre (www.lero.ie).

References

[1] C. Liu, L. Zhu, M. Wang, and Y.-a. Tan, “Search Pattern Leakage In Searchable Encryption: Attacks And New Construction,” *Information Sciences*, vol. 265, pp. 176–188, 2014.

[2] E. Stefanov and E. Shi, “Multi-Cloud Oblivious Storage,” in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. ACM, 2013, pp. 247–258.

[3] O. Goldreich and R. Ostrovsky, “Software Protection And Simulation On Oblivious RAMs,” *Journal of the ACM (JACM)*, vol. 43, no. 3, pp. 431–473, 1996.

[4] D. Boneh, D. Mazieres, and R. A. Popa, “Remote Olivious Storage: Making Oblivious RAM Practical,” 2011.

[5] E. Shi, T.-H. H. Chan, E. Stefanov, and M. Li, “Oblivious RAM With O((logn) 3) Worst-Case Cost,” in *Advances in Cryptology–ASIACRYPT 2011*. Springer, 2011, pp. 197–214.

[6] E. Stefanov, M. Van Dijk, E. Shi, C. Fletcher, L. Ren, X. Yu, and S. Devadas, “Path ORAM: An Extremely Simple Oblivious RAM Protocol,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & communications security*. ACM, 2013, pp. 299–310.

[7] D. Apon, J. Katz, E. Shi, and A. Thiruvengadam, “Verifiable Oblivious Storage,” in *Public-Key Cryptography–PKC 2014*. Springer, 2014, pp. 131–148.

[8] S. Devadas, M. van Dijk, C. W. Fletcher, and L. Ren, “Onion ORAM: A Constant Bandwidth And Constant Client Storage ORAM (Without FHE or SWHE),” 2015.

[9] M. Maffei, G. Malavolta, M. Reinert, and D. Schröder, “Privacy And Access Control For Outsourced Personal Records,” in *36th IEEE Symposium on Security and Privacy*, 2015.

[10] G. Ateniese and B. de Medeiros, “On The Key Exposure Problem In Chameleon Hashes,” in *Security in Communication Networks*. Springer, 2005, pp. 165–179.

[11] G. Ateniese, D. H. Chou, B. De Medeiros, and G. Tsudik, “Sanitizable Signatures,” in *Computer Security–ESORICS 2005*. Springer, 2005, pp. 159–177.

[12] P. Teeuwen, “Evolution Of Oblivious RAM Schemes,” 2015.