Efficient Privacy-Preserving Big Data Processing through Proxy-Assisted ORAM

(Extended Abstract)

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

[1] CASED & TU-Darmstadt, Germany
[2] University of Twente, the Netherlands

Abstract. We present a novel mechanism that allows a client to securely outsource his private data to the cloud while at the same time to delegate to a third party the right to run certain algorithms on his data. The mechanism is privacy-preserving, meaning that the third party only learns the result of his algorithm on the client's data, while at the same time the access pattern on the client's data is hidden from the cloud. To achieve this we combine recent advances in the field of Oblivious RAM and Secure Two-Party Computation: We develop an Oblivious RAM which is ran between the cloud and a proxy server, and which does not need the data to be decrypted at any point. The evaluation on the data is done by employing Yao's garbled circuit solution for Secure Two-Party Computation.

Keywords: Oblivious RAM, Secure Two-Party Computation, Garbled Circuits, Privacy.

1 Introduction

Outsourcing data to remote servers has in the recent years evolved from an optional feature to a practical need. Personal data grows fast by the day and it seems reasonable to store it remotely, where the user can access it from any part of the world and with any computational device that can connect to the Internet. Furthermore, it is preferable for the user to outsource demanding operations to the cloud, since the computations are done faster and without adding extra local load. In this setting it is reasonable to assume that in the near future applications will arise, in which the user will delegate to a third party the rights to access and process private information stored in the cloud, so that the latter will evaluate on the data, while the data owner is offline.

The above mentioned applications however raise some serious privacy concerns. Imagine for example the case, where the client uploads to a remote server his encrypted fully sequenced DNA and wants to delegate a doctor (i.e. the delegated third party which in the sequel we will call the investigator) to perform a query on his private data, such as checking if a specific mutation is present. In this case the client wants to maintain data privacy as far as both the storage server and the investigator are concerned: Encrypting the data and using secure two-party computation (STC) techniques achieves data privacy as far as the investigator is concerned. On the server side however the situation is quite different. Although the server never sees the underlying plaintexts,

he can extract important information about the query itself by carefully examining the parts of the data that have been accessed. Knowing for example that a client has been checked for a certain mutation is already an important privacy leakage, even if the result of the test is never revealed. It is then clear that hiding the access patterns should be dealt with in any of the above mentioned applications.

In this work we propose a scheme, which employs two servers (the cloud and the proxy) and solves the above described problem as efficiently as current secure two-party computation protocols allow. We extend the solution to a setting where the remote server stores the data of multiple users and allows the investigator to compute on different users' encrypted data stored on the server's database.

After the data is uploaded to the cloud, we split the overall problem of obliviously computing on it into two distinct subproblems as indicated in Figure 1: We first retrieve the required encrypted data and subsequently evaluate the data using STC. In the first step the investigator retrieves the encrypted data from the cloud in a way that maintains access pattern privacy. In the second step he obtains the result of the computation by running an STC protocol between the cloud and the proxy. We deal with the first subproblem by adjusting current ORAM schemes to our needs. We thus end up with what we call a proxy-ORAM, which provides the same functionalities as traditional ORAMs, without however demanding the data owner to be online. For the second subproblem we employ tools developed recently for STC and which allow a plethora of functions to be evaluated.

In our solution, we successfully combine ORAM techniques for the data retrieval with the abilities offered by the compiler from [8] and we divise a protocol that contributes in the area of Privacy Enhancing Technologies in the following ways:

1. Our scheme provides a tight integration between ORAM and secure two-party computation. Doing so it offers delegated outsourced computations that maintain access pattern hiding.
2. With slight modifications, the proposed scheme can be turned into a typical ORAM: It offers read/write indistinguishability and hides the access patters while requiring logarithmic storage on the client side (who in our case is the investigator). That way our ORAM offers one new important feature: the data owner is not needed to be constantly online.

1.1 Organization of the Paper

The rest of the paper is organized as follows: We give a brief overview of the related work on Oblivious RAM and on secure two-party computation in Section 2. In Section 3 we describe the necessary cryptographic principles, which we use in our solution. Our solution is described in detail in Section 4. We finish with some concluding remarks and future work in Section 5.

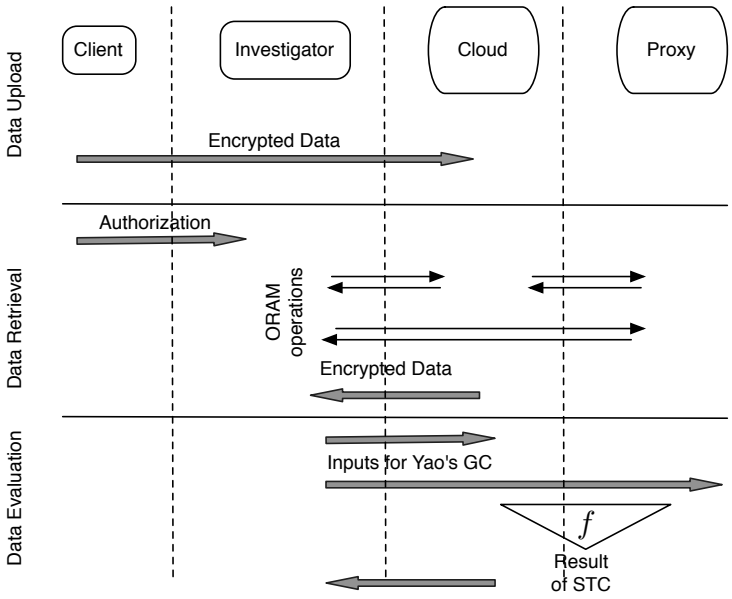

Fig. 1. After the initial upload of the encrypted data, oblivious computing on it using proxy-ORAM is done by splitting the problem into two phases, the Data Retrieval and the Data Evaluation.

2 Related Work

2.1 Oblivious RAM

In [6], Goldreich and Ostrovsky give the first non-trivial solution to the problem of access pattern hiding, called Oblivious RAM (ORAM). They consider a machine whose memory consists of N blocks arranged pyramid-like in $\log N$ levels. Each level is represented as a hash table, consisting of a level dependent number of fixed sized buckets: Level i holds 4^i buckets. Each block is assigned to one of the 4^i buckets at this level, distributed by a hash function H_i^j, with i being the level and j the so-called "epoch", which is associated to the amount of times that this level has been emptied into the next. Emptying one level to the next has to take place in regular intervals and must maintain pattern obliviousness. This procedure is called a "reshuffle" and is the most expensive operation with respect to communication and computational complexity. For a detailed description we refer to [6,18,19,14,7].

In a high level overview the ORAM works as follows: Every block of `data` is identified by an identifier `id` and for each one of them, the client uploads to the server a pair $(H_i^j(\texttt{id}), \mathsf{Enc}(\texttt{data}))$, where Enc is a semantically secure encryption function and H_i^j belongs to a family of hash functions $H : \{0,1\}^* \to \{0,1\}^{4^i}$. Due to hash collisions, every bucket contains up to $\log N$ blocks. Specifically in level i and at epoch j the 4^i positions given by the hash function are split into up to 2^{i-1} real elements which have been entered in the table during the previous reshuffle, 2^{i-1} "fake" elements (whose purpose will become clear in the following) and 2^i empty positions.

In a query for the block with identifier `id` and at epoch j, the client goes through every level of the construction and retrieves a bucket from each, after consulting the hash function H_i^j at every level i. From each bucket the client decrypts every block until he finds the block he was looking for. After the block has been found, the client continues by requesting fake blocks. More precisely, the client performs the following operations:

- All the buckets from the first level are downloaded and checked.
- Until the element is found, the client downloads from level i with $i \in 1, \ldots, \log N$ the bucket indicated by $H_i^j(\texttt{id})$
- After the block has been found, the client continues downloading buckets from all the remaining levels, but this time asking for "fake" buckets[3], indicated by H_i^j.

Once the client retrieves the block he wanted, he re-encrypts `data` and uploads it to the first level of the database, where the server stores it in $H_1^k(\texttt{id})$, with k being the appropriate epoch at level 1. After a certain amount of queries, some levels will become full. In order to avoid overflows, a reshuffle is triggered.

It is then clear from the above discussion, that as long as an encryption scheme is used, that produces unique ciphertexts of the same message, distinguishable only with a negligible advantage from a computationally bounded adversary, the server will not be able to retrieve any information about the queries just by looking at the access patterns.

Goldreich and Ostrovsky's ORAM, which required a total of $O((\log N)^2)$ items to be downloaded per query, was further improved by Williams et. al. in [19] where another approach is followed: Although the database is structured in the same way as in [6], instead of using hashed buckets, an encrypted Bloom Filter per level is used. A Bloom Filter is a randomized data structure that, given a set of elements $S = (s_1, \ldots, s_n)$ and a query for an element s, it returns true with probability 1 if $s \in S$; if $s \notin S$ it returns false with probability p and true with probability $1 - p$. It is implemented as an array B of m bits, initialized to the 0-string together with a total number of ξ hash functions, $\{h_j\}_{j=1}^{\xi} : \{0,1\}^* \to [m]$, such that for every element $s \in S$ it holds that $B[h_j(s)] = 1$ for every $j = 1, \ldots, \xi$. In [19], the hash functions h_j are defined as $\{h_j\}_{j=1}^{\xi} : \{0,1\}^{2^i} \to [m]$, where i denotes the database level; the hash functions are keyed and each bit of the BF is encrypted using a semantically secure encryption scheme. The query runs in a similar way to the one in the Goldreich and Ostrovsky approach: The client goes through every level of the database, first consulting the Bloom Filter for the current level to check whether the element he queries for is on that level. If it is then he retrieves it directly; otherwise he retrieves a fake element. Using this protocol, the client does not need to scan a bucket at every level and therefore avoids one $\log N$ factor of the communication overhead. Recently in [10], the authors revealed a flaw in most of the ORAM constructions, that under certain circumstances can lead to access pattern distinguishability. This problem is mitigated in the construction of [19] as long as the Bloom Filters are of superlogarithmic size.

The authors in [17] construct a novel and intuitive ORAM which is easy to implement and therefore made the construction of a secure processor a reality in [12]. However the ORAM from [17]

[3] These buckets have been placed on the level during the previous reshuffle.

maintains a stash with elements retrieved that cannot fit to the storage. This stash is updated from one access to another and therefore in our multi-client environment can result in information leakage.

2.2 Secure Two-Party Computation (STC)

In his seminal paper [20], Andrew Yao describes generic solutions to the problem of "secure function evaluation". Later he suggests a concrete example of the methods he proposes, with his idea of "garbled circuits". Since then many attempts have been made, in order to make "secure computation" practical, as well as rigorously proving the "garbled circuits" solution secure. In Section 3, we only give a brief overview of Yao's solution and we refer to [11] for a detailed description as well as a rigorous security proof assuming passive adversaries.

Although Yao's solution is straighforward and easy to understand, implementing it is not an easy task. The reason for this is that creating a Boolean circuit for a random functionality f is anything but trivial. In the past years many frameworks like [9,1,2,13] have been developed in order to provide efficient implementations of Yao's solution. However these solutions are either very limited in the functionalities that they can support, or need the circuits to be described in languages such as VHDL, hindering that way their usage in mass scale applications. Under this light the authors in [8] construct a compiler that given the description of a functionality in C, produces a Boolean circuit, which then passes to the STC engine of [9], to handle the execution of the STC protocol.

3 Building Blocks

In this section we briefly describe the cryptographic tools that we have employed in building our solution.

Garbled Circuits. Based on the ideas described in [20], Yao's garbled circuit solution proceeds as follows: For two parties A and B to evaluate a function $f(x, y)$ on their private inputs x_A and x_B respectively, without revealing their inputs to the other party, B creates a Boolean circuit C, which represents the function f and whose input wires represent the inputs of the respective parties. For every gate he then assigns two random keys to each wire, encrypts the gate's operation table and permutes all the entries of the table. The resulting garbled circuit he sends to A, along with the keys that correspond to B's input values. A obtains the keys that correspond to his input values using Oblivious Transfer and evaluates the garbled circuit iteratively going through every one of its' gates and decrypting all the entries of its operation table until he finds the only one that can be correctly decrypted. After evaluating the whole circuit, A announces the result.

ElGamal key transformation. An important property of the ElGamal encryption scheme is that for a given encryption of a message under a public key p_k^1 from a cyclic group G of order q

and with generator g, one can change p_k^1 to another key $p_k^2 \neq p_k^1$ (from the same group G and with the same generator g) without the need of decrypting and then reencrypting with the new key. This property is used in critical parts of our proxy-ORAM and we describe it here as "ElGamal key transformation": Let (c_1, c_2) with $c_1 = g^r$ and $c_2 = mg^{ra}$ be the ElGamal encryption of a message $m \in \mathbb{Z}_p^*$ under the public key $g^a \in \mathbb{Z}_p^*$ with $g \in \mathbb{Z}_p^*$ a group generator and a private key $a \in \mathbb{Z}_p^*$ for a random $r \in \mathbb{Z}_p^*$. Then by picking a random $b \in \mathbb{Z}_p^*$ and multiplying c_2 with g^{rb} we end up with $(c_1, c_2') = (g^r, g^{r(a+b)}m)$ which is the ElGamal encryption of m under the public key g^{a+b}. Inverting this procedure (and thus turning back to the original (c_1, c_2)) knowing b is simply done by multiplying c_2' with $(g^{rb})^{-1}$.

Bresson-Catalano-Pointcheval Encryption scheme. For a detailed description of the scheme we refer to [4]. Here we describe in short the cryptosystem: For a security parameter κ, $\mathsf{Setup}(\kappa)$ chooses a κ-bit safe-prime RSA modulus $N = pq$ (i.e. $p = 2p'+1$, $q = 2q'+1$ for two distinct primes p', q') and picks a random element $g \in \mathbb{Z}_{N^2}^*$ of order $pp'qq'$, such that $g^{p'q'} \mod N^2 = 1 + kN$, for $k \in [1, N-1]$. The plaintext space is $\mathbb{Z}_N$ and the algorithm outputs the public parameters PP=(N, k, g).

The key generation algorithm, $\mathsf{KeyGen}(\text{PP})$ picks a random element $a \in \mathbb{Z}_{N^2}^*$, computes $h = g^a \mod N^2$ and outputs h as the public key and a as the secret key. The encryption algorithm $\mathcal{B}.\mathsf{Enc}_{pk}(m)$, picks a random pad $r \in \mathbb{Z}_{N^2}$ and outputs the ciphertext $(A, B) = (g^r \mod N^2, h^r(1+mN) \mod N^2$. The decryption algorithm $\mathcal{B}.\mathsf{Dec}_{(\text{PP},sk)}(A, B)$ outputs the plaintext as

$$m = \frac{B/A^a - 1 \mod N^2}{N}.$$

CvHP-Hashing. An important role in the way we represent the data in our construction is played by the Chaum-van Heijst-Pfitzmann (CvHP) family of hash functions which was introduced in [5]: Given two primes p and q such that $p = 2p + 1$, two elements α and β, $\alpha \neq \beta$ of order q (i.e. $\alpha^q \equiv 1 \mod p$) and such that the discrete log problem in $\langle\alpha\rangle$ is difficult, the message $m \in \mathbb{Z}_p^*$ is "split" into m_1 and m_2 ($m_1, m_2 \in \mathbb{Z}_q^*$) and the hash function $h : \mathbb{Z}_q^* \times \mathbb{Z}_q^* \mapsto \mathbb{Z}_p^*$ is computed on m as $h(m_1, m_2) = \alpha^{m_1}\beta^{m_2}$. It is easy to see that this hash function is collision resistant, since otherwise, there would exist (m_1, m_2) and (m_3, m_4) such that $\alpha^{m_1}\beta^{m_2} = \alpha^{m_3}\beta^{m_4} \mod p$. Then picking $t = (m_4 - m_2)^{-1}$ we would have that $\alpha^{(m_1-m_3)t} = \beta \mod p$. But this would mean that one could easily compute $\log_\alpha \beta$, which we have assumed to be hard.

4 Architecture

4.1 Overview and Initialization

In our scenario we assume that the client's private data consists of a total number of N blocks. For every block the client also creates a fake block which is sent to the cloud during the original upload and is indistinguishable from a real block. To each block (real or fake) we assign an identifier `id` which we use to form a tuple (c_1, c_2, c_3, c_4) called a *packet* which is the stored representation of a

block on the cloud. The packets are stored on the cloud in a pyramid-like structure consisting of $\log N$ levels, with level i holding up to 2^{i+1} packets (real and fake). After the initial upload, the client delegates to a third party (here called the investigator) the rights to perform computations on his encrypted blocks. Every evaluation on the blocks is performed in two steps: the packet retrieval and the actual evaluation on the retrieved blocks. Each step is performed between the investigator and the cloud with the assistance of a second server, which we call the proxy. The cloud is essentially an ORAM server that unlike classic ORAM constructions (like [6,19]) has the additional property that the ORAM client does not need to be able to decrypt. Our ORAM is built in the way proposed in [19], i.e. maintaining a Bloom Filter in every level, which indicates whether a block is on that level or not. The Bloom Filter is held encrypted by the proxy server and is used by the investigator in order to check if a given element is on a level or not. Building the Bloom Filter is originally done in the upload phase and then in every reshuffle (detailed in 4.3) using a CvHP hash function, that for an identifier `id` is calculated as $BF(id) = g_3^{k_i} g_4^{id} \mod b(l)$ where g_3 and g_4 are group generators of order q, k_i is the $i-$th from a total of ξ keys for the CvHP hash function and $b(i)$ is the variable size of the Bloom Filter depending on the level i. The reshuffle is run between the cloud and the proxy on encrypted data, whose underlying plaintexts are never seen in the clear.

Before we go into more details about the data retrieval and evaluation, we first describe the packet structure.

Packet Description Each packet is identified by its first element c_1, which we call the packet's `index`. The elements c_2 and c_3 are ElGamal encrypted metadata, which are used to build the Bloom Filter and the `index` (both described in section 4.3) in a manner that hides the access patterns.

The data of the block itself is Bresson-Catalano-Pointcheval (BCP) encrypted (described in Section 3) and is stored as the element c_4 of the above defined tuple. The elements c_2 and c_3 are built with the help of the unique identifier `id`, which is associated to every block: For two group generators g_2 and g_4 of order q, the element c_3 is the ElGamal encryption of $g_4^{\mathtt{id}}$ and c_2 is the ElGamal encryption of $g_2^{\mathtt{id}}$. The `index` c_1 at level l is the output of a CvHP hash function (as described in Section 3)

$$\mathtt{index} : \mathbb{Z}_q \times \mathbb{Z}_q \to \mathbb{Z}_p^*$$

such that

$$\mathtt{index}(l, \mathtt{id}) = g_1^{\mathcal{K}(r(l)||l||c)} g_2^{\mathtt{id}}$$

where $\mathcal{K} : \{0,1\}^* \to \mathbb{Z}_q$ is a cryptographic hash function, g_1 is a group generator of order q, r is a function that for the level l calculates its "epoch", i.e. how many times it has been reshuffled and c is a counter kept by the proxy and increased at every reshuffle done in the system. Practically, in order to compute $\mathtt{index}(l, \mathtt{id})$, one needs to know g_1, compute $\mathcal{K}(r(l)||l||c)$ and decrypt c_2.

Summing up the above description, for a block of data $\texttt{data}$ and identifier $\texttt{id}$, its corresponding $\mathsf{packet}(\texttt{index}(l, \texttt{id}))$ is the following tuple:

$$(c_1, c_2, c_3, c_4) =$$

$$(g_1^{\mathcal{K}(r(l)||l||c)} g_2^{\texttt{id}}, \mathcal{E}.\mathsf{Enc}_{pk}(g_2^{\texttt{id}}), \mathcal{E}.\mathsf{Enc}_{pk}(g_4^{\texttt{id}}), \mathcal{B}.\mathsf{Enc}_{ppk}(\texttt{data}))$$

where $\mathcal{E}.\mathsf{Enc}_{pk}$ is ElGamal encryption under the public key pk, $\mathcal{B}.\mathsf{Enc}_{ppk}$ is the BCP encryption under the public key ppk and $\mathcal{K}$, r, g_1, g_2 and g_4 are the variables and functions described earlier (cf. Table 1).

ORAM Initialization In order for the client to initialize the system he does the following:

1. He generates (p_1, q_1, g_1, g_2) for the first CvHP hash (used for the packets' index).
2. He generates (p_2, q_2, g_3, g_4) for the second CvHP hash (used for the Bloom Filter).
3. He generates an ElGamal encryption key pair (sk, pk), used for encrypting the metadata.
4. He Generates an ElGamal encryption key pair ($bfsk$, $bfpk$), used for encrypting the Bloom Filter bits.
5. He generates a BCP encryption key pair (ssk, ppk), used for encrypting the blocks.
6. The client then sends g_1, g_3, sk, $bfsk$ and ssk to the proxy server.
7. He sends g_2, g_4 to the investigator and
8. he sends $bfpk$ to the cloud.

A complete listing of the parameters that each party knows can be found in Table 1.

Parameters	Description	client	investigator	cloud	proxy
g_1	group generator of order q_1 used for `index`	•	•		•
g_2	group generator of order q_1 used for `index`	•	•		
g_3	group generator of order q_2 used for the Bloom Filter	•			•
g_4	group generator of order q_2 used for the Bloom Filter	•	•		
sk	ElGamal secret key for encrypted metadata	•			•
$bfsk$	ElGamal secret key for encrypted bits	•		•	
ssk	BCP private key for encrypted blocks				•

Table 1. Overview of the values known to the various parties

The client then encrypts the blocks and for each one of them creates the corresponding tuple (as described earlier). He also creates as many fake ids as the number of blocks he has, along with their corresponding packets. All the packets are uploaded and stored on the last level of the cloud's ORAM by the client, one by one and after this original upload, all levels are empty except for the last one. A fake packet is uploaded randomly before or after a real packet. This procedure ensures that the cloud cannot distinguish between a real and a fake packet and is described in algorithm (1). The client also creates the Bloom Filter for the last level (only in this first upload) and sends it encrypted to the proxy along with the fake indices.

Algorithm 1 Initialize ORAM(N, real $\mathtt{id}$s, data)

Initialize a list L of size N
for $j = 1$ **to** N **do**
  $\mathtt{id}_{\text{fake}} \xleftarrow{R} \mathbb{Z}_p^*$ {Select the fake $\mathtt{id}$s not allowing duplicates}
  $L \leftarrow \mathtt{id}_{\text{fake}}$
  $b \xleftarrow{R} \{0, 1\}$
  if $b == 0$ **then**
    ORAM-add $\mathtt{packet}(\mathtt{index}(1, \mathtt{id}_j))$
    ORAM-add $\mathtt{packet}(\mathtt{index}(1, \mathtt{id}_{\text{fake}}))$
  end if
  ORAM-add $\mathtt{packet}(\mathtt{index}(1, \mathtt{id}_{\text{fake}}))$
  ORAM-add $\mathtt{packet}(\mathtt{index}(1, \mathtt{id}_j))$
end for
Send L to the proxy

Having given an overview of our construction, the way the blocks are represented and stored in our architecture as well as a description of the initialization phase of our scheme, we can now go into details of the packet's retrieval and evaluation.

4.2 Step 0: Authorization

The authorization of the investigator to access certain data of the client is done through standard access control techniques that are enforced by the cloud. To simplify our exposition we will from now on assume that these techniques are employed and will not go into details in this respect.

4.3 Step 1: Data Retrieval

Query The investigator's query runs through all the levels and in each one he does the following:

1. The investigator sends $g_4^{\mathtt{id}+v}$ to the proxy, for the packet corresponding to index $\mathtt{id}$, that he is interested in, blinded with a blinding value v.
2. The proxy calculates $\{g_3^{k_i} g_4^{\mathtt{id}+v}\}_{i=1}^{\xi}$ and sends them back to the investigator, along with $\mathcal{K}(r(i)||i||c)$.
3. The investigator unblinds the received values and calculates them $\bmod b(i)$, which are the Bloom Filter positions that he should check. The (encrypted) Bloom Filter bits corresponding to these positions he receives from the proxy, along with an index for a fake element $\mathtt{id}_{\text{fake}}$ at that level.
4. Running a similar protocol like the one described above, the investigator decrypts the Bloom Filter bits received with the help of the cloud. If the value of every decrypted bit is 1, then he asks from the cloud for $\mathtt{packet}(\mathtt{index}(l, \mathtt{id}))$, whose index he can form, since earlier he was given $\mathcal{K}(r(i)||i||c)$. Otherwise he asks for $\mathtt{packet}(\mathtt{index}(l, \mathtt{id}_{\text{fake}}))$

After the investigator has received the packet he was asking for, he selects randomly one of the fake packets he retrieved during the query and reencrypts c_2, c_3, c_4 of both packets (the real and the fake). He sends the real and the selected fake packet to the cloud, who stores them in the first level of the ORAM and sends the fake id, $\mathtt{id}_{\text{fake}}$ to the proxy. A graphical overview of the query phase can be found in Figures 2 and 3.

We note here the following:

1. In the case that a level is empty, the proxy informs the investigator and nothing is retrieved from that level.
2. The proxy server keeps track of the fake ids and he sends one to the investigator from every level. This fake id he then discards. However given the fact that during the original upload of the elements, an equal amount of fake ids are set and that at the end of each query the investigator puts back one fake id, the proxy server has always enough fake ids to provide to the investigator, a fact that can be easily shown by induction; note here that the fake ids are only marked as "usable" and "not usable". Every time a fake from a specific level is used, it is marked as "not usable" but once this level is pushed into the next, the fake ids are again marked as "usable", thus meaning that the total amount of the fake ids (and their corresponding fake packets) only increases.
3. Given the above observations and the description of the query, one sees that at every given moment there are at least two levels that contain elements (real and fake). Then the reencryption of the elements that are put back under a rerandomizable encryption scheme guarantees hiding of the access patterns.

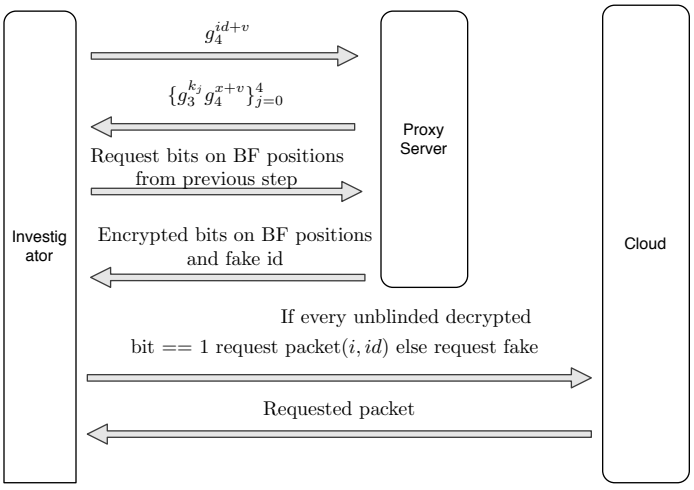

Fig. 2. Overview of the query for id at level i.

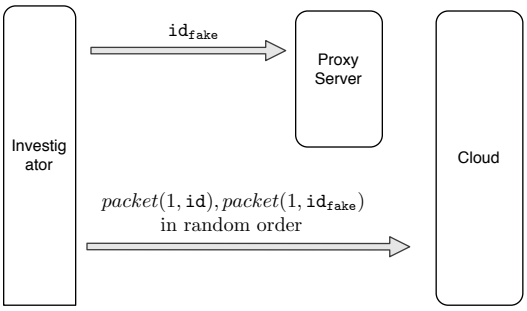

Fig. 3. Putting back the real and a fake packet, after retrieving the real packet.

Reshuffle Once a level is filled with elements, it has to be emptied to the next one. However in order to maintain pattern obliviousness, the elements have to be obliviously reshuffled while they are entered into the new level. Our construction is based on the idea of using Bloom Filters, first proposed in [19], adjusted to the latest developments made by [10,3] and the role of the client from [19] is now played by the proxy, who however must not see the elements that are being reshuffled.

The reshuffle consists of two different phases which are done in the following order: First a Bloom Filter creation phase followed by an update phase, during which the the packets of the two levels merge, leaving one of them empty. A graphical overview of the two phases can be found in Figures 4 and 5 respectively.

Bloom Filter Creation:

1. The cloud merges the two levels and creates a new pair of ElGamal keys (sk', pk'). For every element in the new level, he performs an ElGamal key transformation as described in 3. The resulting encrypted metadata $\mathsf{Enc}_{pk \cdot pk'}(g_4^{\mathtt{index}})$ he puts in a list L_1 which he sends to the proxy server.
2. The proxy forms the underlying Bloom Filter positions by multiplying the received elements with $\mathsf{Enc}_{pk \cdot pk'}(g_3^{k_i})$ for $i = 1 \dots \xi$ and then removes the pk part from every one. The resulting list he sends to the cloud.
3. The cloud initiates a Bloom Filter of size $b(i)$ with values set to $\mathsf{Enc}(0)$. For every element of the received list, the cloud decrypts the metadata (since now they are only encrypted under the public key pk') and sets the positions in the Bloom Filter modulo $b(i)$ to $\mathsf{Enc}(1)$.
4. The resulting encrypted Bloom Filter, the cloud sends to the proxy.

Update:

1. The cloud creates a new pair of ElGamal keys (sk', pk') and for every packet in the new level, he performs an ElGamal key transformation as described in 3. The resulting packets he puts in a list L_2 which he sends to the proxy server, along with the key $pk \cdot pk'$.

2. The proxy forms the underlying new index by multiplying the received elements with $\mathsf{Enc}_{pk \cdot pk'}(g_1^{\mathcal{K}(r(i)||i||c+1)})$, increasing the counter, removes the pk part from the index of every packet and permutes the list. The permuted list he sends to the cloud.
3. The cloud decrypts the index of every packet and removes the pk' from the other coordinates of every packet.

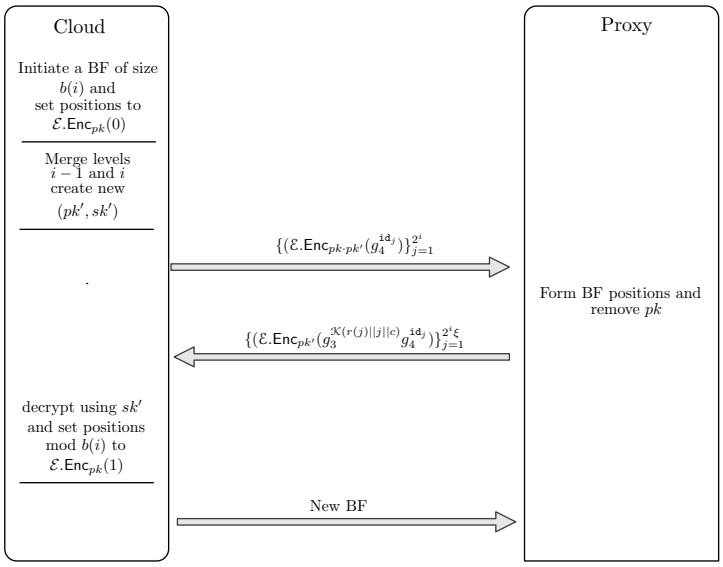

Fig. 4. Overview of the Bloom Filter creation as described in 4.3.

4.4 Step 2: Computation on Encrypted Data

Let $\{\mathcal{B}.\mathsf{Enc}_{ppk}(\mathtt{data}_i)\}_{i=1}^k$ be the packets that the investigator retrieved from the cloud, during the first step. In order to evaluate a functionality f on these inputs, the investigator blinds the encrypted data. He sends the blinded encrypted data and f's description to the proxy and sends the blinding values to the cloud. The proxy decrypts the data and is left with blinded versions of them. Using f's description the proxy creates the circuit. The latter and the cloud can now run Yao's "garbled circuits" using as inputs the values that they were given by the investigator: The proxy uses the blinded decrypted values and the cloud uses the blinding values. The result is then announced to the investigator.

In more detail the second step of our scheme runs as follows:

1. The investigator picks random values $\{r_i\}_{i=1}^k$ and blinds the retrieved packets, thus sending $\{\mathcal{B}.\mathsf{Enc}_{ppk}(\mathtt{data}_i + r_i)\}_{i=1}^k$ along with the functionality's description to the proxy.
2. The blinding values $\{r_i\}_{i=1}^k$ are sent by the investigator to the cloud.

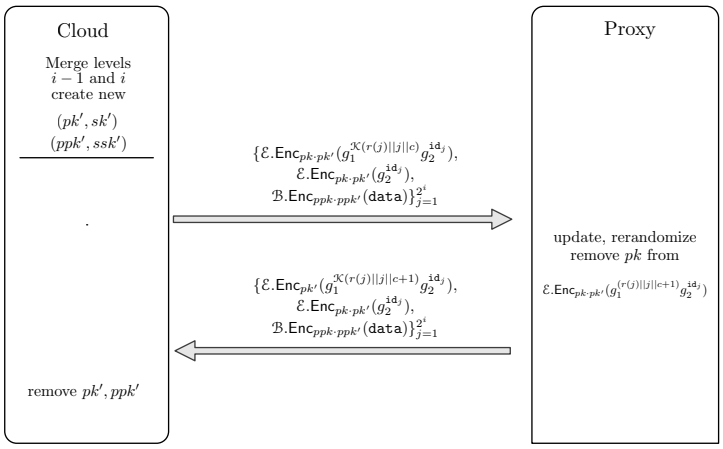

Fig. 5. Overview of the level update in 4.3.

3. The proxy decrypts the blinded data and prepares the "garbled circuit" with the blinded decrypted values, $\{\mathtt{data}_i + r_i\}_{i=1}^{k}$ as his input. In order to prepare the circuit, the proxy uses a compiler such as the one in [8].
4. The proxy and the cloud run Yao's protocol and send the blinded result to the investigator.

4.5 Security

Since the evaluation is done using Yao's "garbled circuit" solution, our security is based on the semi honest adversarial model. We also need to assume that any two parties do not collude; then in this setting our mechanism guarantees access pattern privacy: Any two access patterns are indistinguishable, since the cloud retrieves one block from every level and at the end of the query the investigator puts back on the cloud's first level a fake block and the real block both rerandomized. At the same time the proxy cannot distinguish between two different requests for Bloom Filter bits since any one of those is done on blinded (with different blinding values) data.

5 Conclusion

In this paper we proposed a solution to the problem of delegated, privacy preserving computations on outsourced, encrypted data. Our scheme combines fruitfully ORAM techniques with secure two-party computation, thus allowing a wide applicability that can range from private genome processing to cloud computing. In order to improve the scheme's efficiency, one needs to think of ways to lower the (in the worst case) high communication complexity between the cloud and the proxy server.

References

1. Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: A System for Secure Multi-Party Computation. In *Proceedings of the 15th ACM Conference on Computer and Communications Security*, CCS '08, pages 17–21. ACM, 2008.
2. Dan Bogdanov, Sven Laur, and Jan Willemson. Sharemind: A Framework for Fast Privacy-Preserving Computations. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security*, ESORICS '08, pages 192–206. Springer, 2008.
3. Prosenjit Bose, Hua Guo, Evangelos Kranakis, Anil Maheshwari, Pat Morin, Jason Morrison, Michiel H. M. Smid, and Yihui Tang. On the false-positive rate of bloom filters. *Inf. Process. Lett.*, 108(4):210–213, 2008.
4. Emmanuel Bresson, Dario Catalano, and David Pointcheval. A simple public-key cryptosystem with a double trapdoor decryption mechanism and its applications. In Chi-Sung Laih, editor, *ASIACRYPT*, volume 2894 of *Lecture Notes in Computer Science*, pages 37–54. Springer, 2003.
5. David Chaum, Eugène van Heijst, and Birgit Pfitzmann. Cryptographically strong undeniable signatures, unconditionally secure for the signer. In Joan Feigenbaum, editor, *CRYPTO*, volume 576 of *Lecture Notes in Computer Science*, pages 470–484. Springer, 1991.
6. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious rams. *J. ACM*, 43(3):431–473, 1996.
7. Michael T. Goodrich, Michael Mitzenmacher, Olga Ohrimenko, and Roberto Tamassia. Privacy-preserving group data access via stateless oblivious ram simulation. In Rabani [15], pages 157–167.
8. Andreas Holzer, Martin Franz, Stefan Katzenbeisser, and Helmut Veith. Secure two-party computations in ansi c. In Yu et al. [21], pages 772–783.
9. Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.
10. Eyal Kushilevitz, Steve Lu, and Rafail Ostrovsky. On the (in)security of hash-based oblivious ram and a new balancing scheme. In Rabani [15], pages 143–156.
11. Yehuda Lindell and Benny Pinkas. A proof of security of yao's protocol for two-party computation. *J. Cryptology*, 22(2):161–188, 2009.
12. Martin Maas, Eric Love, Emil Stefanov, Mohit Tiwari, Elaine Shi, Krste Asanovic, John Kubiatowicz, and Dawn Song. Phantom: practical oblivious computation in a secure processor. In Sadeghi et al. [16], pages 311–324.
13. Lior Malka. Vmcrypt: modular software architecture for scalable secure computation. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM Conference on Computer and Communications Security*, pages 715–724. ACM, 2011.
14. Benny Pinkas and Tzachy Reinman. Oblivious ram revisited. In Tal Rabin, editor, *CRYPTO*, volume 6223 of *Lecture Notes in Computer Science*, pages 502–519. Springer, 2010.
15. Yuval Rabani, editor. *Proceedings of the Twenty-Third Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2012, Kyoto, Japan, January 17-19, 2012*. SIAM, 2012.
16. Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors. *2013 ACM SIGSAC Conference on Computer and Communications Security, CCS'13, Berlin, Germany, November 4-8, 2013*. ACM, 2013.
17. Emil Stefanov, Marten van Dijk, Elaine Shi, Christopher W. Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path oram: an extremely simple oblivious ram protocol. In Sadeghi et al. [16], pages 299–310.

18. Peter Williams, Radu Sion, and Bogdan Carbunar. Building castles out of mud: practical access pattern privacy and correctness on untrusted storage. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 139–148. ACM, 2008.
19. Peter Williams, Radu Sion, and Alin Tomescu. Privatefs: a parallel oblivious file system. In Yu et al. [21], pages 977–988.
20. Andrew Chi-Chih Yao. How to generate and exchange secrets (extended abstract). In *FOCS*, pages 162–167. IEEE Computer Society, 1986.
21. Ting Yu, George Danezis, and Virgil D. Gligor, editors. *the ACM Conference on Computer and Communications Security, CCS'12, Raleigh, NC, USA, October 16-18, 2012*. ACM, 2012.