Privacy-Preserving Whole Genome Sequence Processing through Proxy-Aided ORAM

Nikolaos P. Karvelas
TU Darmstadt & CASED
Germany

Andreas Peter
University of Twente
The Netherlands

Stefan Katzenbeisser
TU Darmstadt & CASED
Germany

Erik Tews
TU Darmstadt & CASED
Germany

Kay Hamacher
TU Darmstadt
Germany

ABSTRACT

Widespread use and low prices of genomic sequencing bring us into the area of personalized medicine and biostatistics of large cohorts. As the processed genomic data is highly sensitive, Privacy-Enhancing Technologies for genomic data need to be developed. In this work, we present a novel and flexible mechanism for the private processing of *whole* genomic sequences which is flexible enough to support *any* query. The basic underlying idea is to store DNA in several small encrypted blocks, use ORAM mechanisms to access the desired blocks in an oblivious manner, and finally run secure two-party protocols to privately compute the desired functionality on the retrieved encrypted blocks. Our construction keeps all sensitive information hidden and reveals only the end result to the legitimate party. Our main technical contribution is the design of a new ORAM that allows for access rights delegation while not requiring the data owner to be online to reshuffle the database. We validate the practicability of our approach through experimental studies.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and protection*; K.4.1 [**Computer and Society**]: Public Policy Issues—*Privacy*; J.3 [**Life and Medical Sciences**]: Biology and genetics

Keywords

Oblivious RAM, Secure Computation, Genome Privacy

1. INTRODUCTION

Personalized medicine based on genomic data will result in a need to store genetic data as part of a patient's electronic health record. Furthermore, the great benefits of biostatistical analysis and large collections of genomic data for public health will put pressure on a traditional understanding of privacy. Technically, the major roadblock for this development is already removed: the costs to sequence an individual's genome dropped well below $400. Great benefits and ever sinking costs lead us to expect a sharp increase in the volume of stored genomic data in the coming years.

Unfortunately, the availability of a large set of genetic data incurs huge privacy problems; genetic data can arguably be seen as one of the most sensitive forms of medical data. Furthermore, our knowledge on the human genome increases over time; at present, it is impossible to estimate the future consequences in case a breach of genomic data occurs. Thus, genomic databases should be protected with strong Privacy-Enhancing Technologies (PETs) early on.

The most promising approach to protect genomic data is the use of cryptographic techniques from secure computation, since they provide strong cryptographic security, both during storage and processing. In this approach, genomic data is stored in encrypted form and the evaluation is performed directly on encrypted data. This both protects the raw genomic data and allows to control the types of queries that can be performed.

The design of such PETs faces many, synergistically occurring challenges: For one, the entire genome of a patient requires some 700MB of storage, which results in a huge overhead when this data is processed "under encryption". Additional data of uttermost importance, such as methylation patterns, require additional space. In contrast to these demands, current cryptographic methods for secure computation are tailored towards small and mid-size computational problems. At the same time, it is at the moment unclear how genomic data will be used in the future and which queries will be performed on a genomic database. Currently, the predominant tests look for single-nucleotide polymorphisms (SNPs) – which amounts to checking whether a certain symbol occurred in the genome at a given position. It is foreseeable and desirable that more complex statistical tests will enter the arena [1, 2, 3]. Any practical PET for genetic data should thus support *any* query in order to adapt to future advances in genetic research. Since a cryptographically protected sequenced genome will likely be stored only once during the lifetime of a patient, the form of protection should not be tailored to a specific query type but should be as flexible as possible to be able to cope with future queries. Furthermore association studies where genetic and physiological data are combined pose a great challenge in biostatistics and therefore PETs should support these applications as well. Analysis *across* several genomes must

WPES'14, November 03, 2014, Scottsdale, AZ, USA.

http://dx.doi.org/10.1145/2665943.2665962.

be possible to investigate inheritable traits. Again, autism is an example, where genomic heterogeneity and so-called copy-number variation are known to be disease-related [4]. Finally, an ideal PET solution should be able to hide the type of the query from the involved parties. For example, if a human genetics laboratory performs a preimplantation genetic diagnosis for, e.g., autism and the correlated presence of a specific mutation or combination thereof, the mere fact that certain parts of the genetic data are accessed in the parental genomes already leaks privacy-sensitive information (namely that an implantation is to occur, who the parents are, and that they wonder about their own susceptibility for a disease). Ideally, this fact should be hidden from the (commercial) entity that stores the genomic data.

Related work on DNA protection. All existing solutions have drawbacks in at least one of the above listed aspects. Early works focused on processing short DNA fragments and were tailored towards simple queries: for example, [5] showed how to run queries in the form of finite state machines on DNA sequences, and was subsequently improved in [6, 7, 8, 9]. Unfortunately due to their computational complexity, these approaches are only applicable to small fragments of DNA strings and not to the entire genome. Further, they are tailored towards very specific queries, which can be represented as finite state machines.

Similarly solutions tailored to perform specific forensic tests efficiently have been proposed as well [10, 11]. Only a couple of works targeted efficient private queries on fully sequenced genomes: [12] focused on targeted tests for paternity, personalized medicine applications and genetic compatibility; technically, they employ secure set intersection protocols, which are extremely efficient but only work for simple queries. Finally, [13] proposed an architecture based on homomorphic encryption which is flexible with respect to the query, but leaks the type of test performed to the involved parties. Still, they lack the capacity to learn and answer important "fuzzy" (statistical) questions – namely to act in concert with biostatistics. Note, that all of these approaches apply to digitally represented and sequenced genomic data; they can be amended by privacy-preserving genomic sequencing techniques [14].

Contributions. In this work, we propose a solution that is *flexible to support future query types*, is applicable to *fully sequenced genomes* and allows to *hide the nature of the performed test*. To this end, we store the sequenced genome at a server in encrypted form in a special randomized data structure that achieves access pattern privacy, called Oblivious RAM (ORAM) [15]. In a subsequent query phase the stored data can be accessed and utilized in a secure computation. By separating secure storage from the query phase, we achieve full flexibility to adapt to future queries. We achieve efficiency by only requesting as input to the secure computation phase those parts of the genomic string that are actually necessary to perform the computation and not the entire genomic sequence – while hiding the queried positions themselves. This enables query times that are sublinear in the length of the genomic sequence. Finally, due to the use of an ORAM, the party storing the genome is fully oblivious on the data accessed during a query; he only sees the *amount* of data accessed, but not its location in the genome.

In more detail, our construction employs two separate servers (called cloud and proxy in the sequel) which jointly operate an ORAM to which encrypted genetic data can be uploaded. A client can subsequently authorize a different party, called "investigator" to perform a query on the data in two steps: In the first step the investigator retrieves the required encrypted genetic data from the ORAM. We do this by developing an ORAM which does not require a "data owner" (such as the patient whose genome was sequenced) to be constantly online. In the second step the investigator obtains the result of the computation by running any secure computation protocol between the cloud and the proxy, which provides the flexibility to perform any genetic test securely. We also extend the solution to a setting where the remote server stores data of multiple users in one or more different ORAMs and allows the investigator to compute on different users' encrypted data stored on the server's database. Finally, we demonstrate the practical feasibility of the approach by implementing three analysis techniques operating on (simulated) fully sequenced genomes.

2. BUILDING BLOCKS

Bresson-Catalano-Pointcheval Encryption [16]. For a security parameter κ, $\mathcal{B}.\mathsf{Setup}(\kappa)$ 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 $\mathrm{PP} = (N, k, g)$. The key generation algorithm $\mathcal{B}.\mathsf{KeyGen}(\mathrm{PP})$ outputs a random element $a \in \mathbb{Z}^*_{N^2}$ as *secret* key and the element $h = g^a \mod N^2$ as *public* 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 $\mathcal{B}.\mathsf{Dec}_{(\mathrm{PP},sk)}(A, B)$ outputs the plaintext as $m = (B/A^a - 1 \mod N^2)/N$. The Bresson-Catalano-Pointcheval (BCP) cryptosystem is semantically secure under the DDH assumption and is additively homomorphic, i.e.,

$$\mathcal{B}.\mathsf{Dec}_{(\mathrm{PP},sk)}(\mathcal{B}.\mathsf{Enc}_{pk}(m) \cdot \mathcal{B}.\mathsf{Enc}_{pk}(m')) = m + m'.$$

ElGamal Proxy Re-Encryption [17]. Proxy Reencryption allows a semi-honest proxy to transform a ciphertext, generated by a party A with her public key, into an encryption under the public key of another party B. The proxy can do so by means of a *re-encryption key for B* given by A. Ivan and Dodis [18] showed that ElGamal-like schemes (such as the BCP scheme) are in fact proxy re-encryption schemes: Let $\mathcal{E} = (\mathcal{E}.\mathsf{KeyGen}, \mathcal{E}.\mathsf{Enc}, \mathcal{E}.\mathsf{Dec})$ denote the ElGamal cryptosystem and let (c_1, c_2) be an ElGamal ciphertext with $c_1 = g^r$ and $c_2 = mg^{ra}$ of a message m under the public key g^a for some cyclic group generator g, secret key a, and random exponent r. Consider a "secret sharing" of the key $a = x_1 + x_2$ into two random exponents x_1 and x_2. Then, x_1 becomes the "re-encryption key" and x_2 the new decryption key: Given an encryption (c_1, c_2) under the public key g^a, $(c_1, c_2 \cdot c_1^{-x_1})$ yields an encryption under g^{x_2}.

Bloom Filters [19]. 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, 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

all $j = 1, \ldots, \xi$. Testing if an element s is in the Bloom filter amounts to verifying that $B[h_j(s)] = 1$ for every $j = 1 \ldots, \xi$.

Chaum-van Heijst-Pfitzmann (CvHP) Hash [20]. Given two primes p and q such that $p = 2q+1$, two elements α and β, $\alpha \neq \beta$ of order q and such that the DL problem in the group $\langle\alpha\rangle$ generated by α 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 $h(m_1, m_2) = \alpha^{m_1}\beta^{m_2}$. Collision resistance follows from the DL problem.

Secure Two-Party Computation (STC) using Garbled Circuits [21]. Suppose two parties A and B want to securely evaluate a certain function on their respective private inputs. This can be done by letting party A create a Boolean circuit of this function and assign two random keys (representing 0 or 1) to each wire of every gate; further, A encryps and permutes the gate's operation table. A sends the resulting "garbled circuit" to party B, along with the corresponding keys for A's input values. Using Oblivious Transfer to obtain the keys corresponding to B's input, and iterating through every gate, B eventually evaluates the whole circuit on their inputs and obtain the final result. We refer to [22] for a detailed description as well as a rigorous security proof assuming passive adversaries. To automatically turn an arbitrary function into an STC protocol, we will use the compiler of [23]. This compiler takes the description of a function in ANSI C and produces a Boolean circuit, which can be passed to an STC framework such as [24] to obliviously compute the desired function.

Oblivious RAM [15]. Suppose a client wants to outsource her private data to a semi-honest server in an encrypted form such that she can later on still read and write her encrypted data blocks, while the server should not learn anything other than the number of blocks it stores and the total number of accesses. In particular, the server should not see the access pattern of the queries (i.e., which element was accessed at which time). Goldreich and Ostrovsky [15] proposed the first solution to this problem which considers the encrypted data blocks, say N in total, along with an equal amount of "fake" blocks, arranged in a pyramid-like structure of $\log N$ levels. Each level is represented as a hash table, consisting of a level dependent number of fixed sized buckets, to which the blocks are assigned. In order to retrieve one block from the ORAM, the client downloads and decrypts one bucket from every level, as indicated by the level's hash table. In one of those buckets the client is guaranteed to find the block she was querying for, while for the server the accesses look totally random. After the client has finished her query, she writes the block back into the first level's bucket: If the query was a read operation, she writes back the block re-encrypted; in case of a write operation, she writes back an encryption of the updated block. Once the buckets of a level are filled, they are emptied to the next level; this process (called a "reshuffle") achieves a complete destruction of any correlation between the blocks accessed on the two levels.

Subsequent works [25, 26] replaced the hash table originally proposed in [15] with a Bloom filter. In every level encypted real and fake items are stored. A query for an element runs in a similar way as before, but instead of looking at the hash table in order to see if an element is on the level and downloading the corresponding bucket, the client now consults the corresponding Bloom filter and downloads either the real element (if it was found on that level) or a fake one, thus saving an $O(\log N)$ communication factor per query. The reshuffle is also simplified: after two levels are merged, one only needs to re-randomize the elements and build a new Bloom filter. This Bloom filter-based construction will be the starting point for our new ORAM described in Section 4. We stress that the recent attack by Kushilevitz et al. [27] does not apply to such Bloom filter-based ORAMs as long as the Bloom filter sizes are of adequate size as proposed by [28].

Today, there is a large body of works on ORAM. Most notably, Stefanov et al. [29] construct a highly efficient ORAM which constitutes the current state-of-the-art. However, it maintains a client-side storage (stash) with elements previously retrieved that cannot fit into the ORAM. This stash is updated from one access to another. In the multi-client environment that we look at, this can result in serious information leakage. Another related work is that of Franz et al. [30] who extend the standard ORAM to the multi-client setting by allowing the data owner to delegate access to her data. This solution cannot directly be applied in our DNA setting either, as it requires a data owner (such as the DNA donor) to be online for performing the reshuffle which is undesirable in our scenario. We strive for a multi-client ORAM solution that requires no client-side stateful storage and runs autonomously from the DNA donors.

3. ARCHITECTURE

Our privacy-preserving computation architecture utilizes three major semi-honest and non-colluding entities, that communicate over secure channels: a *client* (e.g., a patient), an *investigator* (e.g., a physician), and a DNA storage service (e.g., a hospital or biobank). The storage service is considered as a specialized service which consists of the actual data store, called the *cloud*, and a separate non-colluding *proxy* that assists in certain computational tasks. Splitting the databases into two non-colluding servers is a wide-spread approach in current research, proved to be very promising (cf. Boneh et al. [31] and Stefanov et al. [32]).

Upon acquisition of genetic data, a client uploads her DNA in small encrypted blocks to the cloud. Subsequent access to the data is illustrated in Figure 1. Whenever the investigator wants to access certain DNA blocks of a client he first asks the client for authorization (steps 1–2) and retrieves the required blocks from the cloud in a private manner (step 3) through a special ORAM protocol. Once all data is retrieved, the investigator can command the cloud and the proxy to jointly compute a certain function (e.g., a medical test or study) on the retrieved encrypted blocks, again in a secure and private way (steps 4–5). To do this, the cloud and the proxy run Yao's garbled circuit protocol, with the inputs and circuit created by the investigator. Throughout these steps, all private data from the client remains encrypted and only the result of the computation is revealed to the investigator (step 6). Neither the cloud nor the proxy will learn anything about the stored data, including the investigator's access patterns (except for the number of blocks accessed). This architecture can be generalized in a straightforward way to tests that operate on multiple genomes: in the first step, all required encrypted data blocks are fetched from the cloud given the consent of all involved users. Note that in such a multi-client setting, DNA data of multiple clients can be stored in one ORAM database; alternatively one ORAM can be used per client.

We implement the client authorization through proxy reencryption: The client sends a secret key to the investigator as an access token and the reecryption key for the investigator's token to the proxy. This way the encryption of the stored blocks can be transformed to an encryption that is accessible by the investigator. Hiding the access pattern of the investigator is done through a new ORAM mechanism that we will explain in Section 4. The challenge here is to design an ORAM that allows the delegation of accesses while, in contrast to [30], not requiring clients to be online in order to perform the ORAM reshuffle. This way, we achieve a completely autonomous DNA storage service that provides data confidentiality while hiding the access patterns.

The retrieved DNA blocks are homomorphically encrypted and subsequently converted to shares, which are given to the cloud and the proxy, along with a Boolean circuit representation of the functionality the investigator wants to evaluate. This transformation allows full flexibility of queries as the cloud and the proxy can run Yao's "garbled circuit" for any desired functionality. This step is detailed in Section 6.

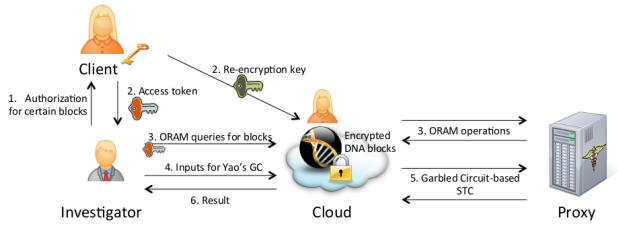

Figure 1: Overview of our architecture.

4. A NOVEL ORAM MECHANISM

We assume that the client's fully sequenced DNA is stored encrypted in small blocks of fixed size. Every block is identified uniquely by a randomly assigned identifier $\texttt{id}$ (details can be found in Section 7). Typical for an ORAM construction, fake blocks must also reside in the ORAM and both real and fake blocks must be encrypted under a rerandomizable encryption scheme so that one is not able to distinguish whether a real or a fake block has been accessed. Assuming that the client stores 2^{N-1} blocks of data, we add another 2^{N-1} fake blocks. Each fake block contains random data and is indistinguishable from a real block, when encrypted. To each fake block we assign a random identifier $\texttt{id}_{\text{fake}}$ (making sure that there are no collisions between the ids of fake and real elements). The fake and real blocks are stored encrypted on the cloud in an N level pyramid like structure. Contrary to typical ORAM constructions, in our architecture the encrypted blocks are initially uploaded to the last pyramidal level (i.e., a level that can hold all initially uploaded data). After a block is accessed, it is always placed on the first pyramid level. If the first level does not have enough space, then it is recursively emptied to lower ones, until enough space has been created on the first level.

In the following we describe how blocks in the ORAM are structured and how the block access works. In order for the client to find the block he wants, he must first find the level on which the block currently resides and then the block's exact position within that level. Finding the right level is done with the aid of Bloom Filters: Each pyramid level has one Bloom filter, whose contents (i.e., the individual filter bits) are kept encrypted and which is built interactively between the cloud and the proxy, whenever a level is reshuffled (see Section 4.2). In contrast to classic Bloom Filters, we use the CvHP hash function (cf. Section 2) as basis due to its homomorphic property, which allows to construct an efficient reshuffle procedure. If an element with identifier $\texttt{id}$ is present on level l, we set all Bloom filter positions $h(k_i, \texttt{id}) = g_3^{k_i} g_4^{\texttt{id}} \mod b(l)$ of level l to one, where k_i with $i \in \{1, \ldots, \xi\}$ are used as keys for the CvHP hash function, created by the proxy during the Bloom filter creation and stored on the proxy (cf. Table 1). The two generators of the CvHP are g_3 and g_4, where g_3 is known to the client and the proxy while g_4 is known to the client and the investigator. Note that the output of the CvHP hash function is further hashed modulo a publicly known, level dependent function $b(l)$ so that we can maintain a Bloom filter whose size is proportional to that of the level.

Once the block has been identified to reside on a specific level, the investigator must retrieve it from that level. This needs to be done in a way that does not leak the block identifier. In particular access patterns need to change upon every access. Again we use a CvHP hash function, this time on a "salt" and the block identifier. The salt is computed as the output of a publicly known hash function $\mathcal{K}$ on an input composed of the level l, a counter $r(l)$ of the reshuffles done at this level and a counter c which is initiated to a random value chosen by the proxy and subsequently incremented every time a reshuffle is performed at any level of the ORAM: $salt(l, c) = \mathcal{K}(l||r(l)||c)$. The two generators g_1 and g_2 of the CvHP hash function are known to the client and the investigator, while the proxy knows only g_1. Thus the index of a block at a certain level is given as $index(l, \texttt{id}) = g_1^{salt(l,c)} g_2^{\texttt{id}} = g_1^{\mathcal{K}(l||r(l)||c)} g_2^{\texttt{id}}$.

Within the ORAM we store the encrypted data block together with its index. In order to compute the Bloom filter positions and be able to change the index once one block was accessed, we also have to store metadata, namely the elements $g_2^{\texttt{id}}$ and $g_4^{\texttt{id}}$, encrypted under a rerandomizable encryption scheme. This block represenation will from now on be called a *packet* of the block with identifier $\texttt{id}$ on level l; $\texttt{packet}(\texttt{index}(l, \texttt{id})) = (c_1, c_2, c_3, c_4)$, with

$$\begin{aligned} c_1 = g_1^{\mathcal{K}(r(l)||l||c)} g_2^{\texttt{id}},\ c_2 = \mathcal{E}.\mathsf{Enc}_{pk}(g_2^{\texttt{id}}), \\ c_3 = \mathcal{E}.\mathsf{Enc}_{pk}(g_4^{\texttt{id}}),\ c_4 = \mathcal{B}.\mathsf{Enc}_{ppk}(\texttt{data}). \end{aligned} \tag{1}$$

From what will become clear in the following, the elements c_2 and c_3 have to be encrypted under a rerandomizable multiplicatively homomorphic encryption scheme, for which we choose ElGamal encryption $\mathcal{E}.\mathsf{Enc}_{pk}$ under the public key pk (the keys for c_2 and c_3 have to be different but for ease of presentation, we will not detail this here). For the element c_4 we need a rerandomizable encryption scheme, which has to be additively homomorphic, a property imposed by the application scenarios that we examine (cf. Section 6). Thus we choose the BCP encryption scheme $\mathcal{B}.\mathsf{Enc}_{ppk}$ under the public key ppk.

4.1 ORAM Initialization and Data Upload

Initialization and Upload. Before the client uploads her encrypted blocks to the cloud, she creates an equal amount of fake blocks and assigns a unique identifier (avoiding col-

lisions) to every block (real and fake). She initializes the secret generators g_1, g_2, g_3, g_4 and creates the ElGamal keys (sk, pk), used for encrypting every packet's metadata as well as the BCP keys (ssk, ppk) in order to encrypt the actual blocks. She also generates another pair of ElGamal keys, $(bfsk, bfpk)$, with which the Bloom filter contents will be encrypted. For every block (real and fake) she then creates its corresponding packet, see equation (1). The client distributes the keys and the elements g_1, g_2, g_3, g_4 to the parties as follows: The cloud receives the secret key $bfsk$, while the proxy receives g_1, g_3 and the secret keys sk and ssk (cf. Table 1). Observe here that the cloud has the keys to decrypt the Bloom filter contents but only stores the packets, while the proxy has the keys to decrypt every packet's c_4 element (i.e., the encrypted block), but only stores the Bloom Filter. This is secure as along as the two servers do not collude.

The client then uploads the packets (fake and real) in random order to the cloud, who stores them on the last level of the pyramid, while all levels above are initialized and left empty. Finally she sends the list of the fake ids to the proxy (the reason for this will become apparent in the detailed description of the ORAM operations in Section 4.2). Once the packets have been uploaded, a Bloom filter creation (see Section 4.2) is initiated between the cloud and the proxy, resulting in the Bloom filter for the last level, which is stored in encrypted form on the proxy.

Authorization. At the beginning of a session in which the investigator wants to read encrypted DNA blocks from the cloud, he receives from the client the authentication token. This consists of the group generators g_1, g_2, g_4 and a temporary secret key tsk. The temporary key is also sent to the cloud and the proxy and is used so that the investigator can decrypt Bloom filter bits and the c_2 elements of all packets (only in the current session).

	g_1	g_2	g_3	g_4	sk	tsk	$bfsk$	ssk	k_i
Client	•	•	•	•	•	•	•	•	
Inv.	•	•		•		•			
Cloud						•	•		
Proxy	•		•		•	•		•	•

Table 1: Overview of the values known to all parties.

4.2 ORAM Operations

Like classical ORAMs, our ORAM supports three operations, namely `read`, `write` and `reshuffle`. After a `read` or `write` is performed, the investigator puts *two* packets back on the top level of the cloud's pyramid: the real one and a fake one. In case of a `read`, the investigator re-randomizes the elements c_2, c_3, c_4 of the real and the fake packet, while in case of a `write` he additionally replaces c_4 of the real packet with an encryption of the new data. Since c_4 is a BCP encryption of the block's data and a re-randomized ciphertext is indistinguishable from a fresh encryption, we can discuss from here on only the case of a `read`, knowing that the same algorithm applies for the case of a `write`. Note that, due to this property, neither the proxy nor the cloud can distinguish a `read` from a `write`. Regardless of the operation performed, the packet's index c_1 is also updated, as described below.

Read Operation. The investigator's query for a block of identifier `id` runs through all the ORAM levels and works as follows: If the level is empty then there is nothing to be done. If the level is non-empty then the investigator first receives the id of a fake packet to be found on that level from the proxy (step 1). Subsequently he interactively with the proxy computes the Bloom filter positions corresponding to the block he is querying for, assuming that it resides on that level (steps 2 to 4). Using ElGamal proxy re-encryption the investigator learns if the block is on that level or not (steps 5 and 6) by decrypting the Bloom filter bits. Subsequently he either retrieves the real block (if it is on that level) or the fake block (which is certainly there), by computing its `index` and requesting the corresponding packet from the cloud (step 7). In step 8 the investigator makes sure that in case he asked for a real packet, he has retrieved the correct one. In more details the steps are described below:

1. The investigator retrieves the $\texttt{id}_{\text{fake}}$ for a fake block guaranteed to reside on level l from the proxy[1].
2. The investigator sends $g_4^{\texttt{id}+v}$ to the proxy, for the packet corresponding to identifier `id`, that he is interested in, blinded with a randomly chosen blinding value v.
3. The proxy calculates $\{g_3^{k_i} g_4^{\texttt{id}+v}\}_{i=1}^{\xi}$ and sends these values back to the investigator, along with $\mathcal{K}(r(l)||l||c)$, where $\{k_i\}_{i=1}^{\xi}$ are the ξ Bloom filter keys held by him.
4. The investigator unblinds the received values and takes them modulo $b(l)$, which results in the Bloom filter positions that he should check. He then requests the encrypted Bloom filter contents of these positions from the proxy.
5. The investigator blinds the encrypted Bloom filter contents (each with a different blinding value) and sends them to the cloud.
6. The cloud (knowing the secret key $bfsk$ for the Bloom filter bits) performs an ElGamal proxy re-encryption for the key tsk on the requested Bloom filter bits and sends them to the investigator.
7. Using the authentication token tsk, the investigator decrypts and unblinds the Bloom filter bits, thus learning if the desired block with identifier `id` is on the level. If the block was already found on a previous level or the block is not on that level, then the investigator computes the index of the fake packet and requests from the cloud the $\texttt{packet}(\texttt{index}(l, \texttt{id}_{\text{fake}}))$. Otherwise he computes the index of the desired block id and requests from the cloud the $\texttt{packet}(\texttt{index}(l, \texttt{id}))$, whose element c_2, the cloud proxy re-encrypts for the key tsk and sends to the investigator.
8. If the investigator asked for a real packet, then using his authentication token, he decrypts the element c_2 of the packet and compares it to $g_2^{\texttt{id}}$. If they are not the same, then a false positive has occured within the Bloom filter, meaning that the packet retrieved was not the real one. Then the investigator continues as if he had found a fake element during this query.

[1]Note here that the proxy knows *only* the fake ids on the levels, but does not see if the investigator retrieved a fake or a real packet.

After the investigator has received the packet he was asking for, he generates a fake packet by computing the elements c_2, c_3, c_4 for the fake id retrieved from the last ORAM level. He then rerandomizes the elements c_2, c_3, c_4 of the real packet he retrieved and sends the fake id ($\mathtt{id}_{\text{fake}}$) he got from the last level to the proxy. The proxy stores $\mathtt{id}_{\text{fake}}$ on the list of fake ids that are available on the first level and marks all the fake ids that he gave throughout the query as "used", so that they are not re-used in subsequent queries. Finally the investigator sends the real and the fake packet in random order to the cloud. The cloud creates the indices for the two packets with the help of the proxy in the following way: If the cloud does not have enough space to hold the two elements in the first level, then a `reshuffle` is triggered, during which the freshly added elements and the ones that were already in the first level are shuffled, merged with the first level and their indices are updated. Otherwise if there is enough space for the two elements to be stored in the cloud's first level, the two packets to be added are considered as a "level 0" and are reshuffled with the elements of the first level. Thus the indices of the two packets are generated, their order is randomly permuted and a new Bloom filter for level 1 is created. Details are given below:

Reshuffle. Once a level is entirely filled with packets, it is emptied recursively to the next one, until a level is reached, which does not overflow after its previous level is emptied into it. In order to maintain pattern obliviousness, the elements have to be obliviously reshuffled while they are entered into the new level. The `reshuffle` consists of two distinct phases: First, a Bloom filter creation phase, where the Bloom filter for the new level is created and then an update phase during which the packets of the two levels are merged, leaving the smaller of the levels empty. In details reshuffling levels $l-1$ and l works as follows:

Phase 1: Bloom Filter Creation. In the first reshuffle phase the current Bloom filters of levels $l-1$ and l are destroyed and a new Bloom filter for level l is created. The operation is interactively performed between the proxy and the cloud, who compute the positions of the Bloom filter that need to be set to one, without being able to identify the blocks in the two levels. In more detail the steps performed are as follows:

1. The current Bloom filters and the keys $\{k_i\}_{i=1}^{\xi}$ of levels $l-1$ and l are discarded and new keys for level l are selected by the proxy.
2. The proxy marks the ids of all the fake elements from level $l-1$ as "not used" and merges them with the list of fake ids maintained for level l.
3. The cloud creates a list that contains all the packets from levels $l-1$ and l. Furthermore, he creates a temporary pair of random ElGamal keys (sk', pk'), which will be discarded at the end of this phase. For every packet (c_1, c_2, c_3, c_4) present on the list, the cloud uses the proxy re-encryption property of ElGamal to transform $c_3 = \mathcal{E}.\mathsf{Enc}_{pk}(g_4^{\mathtt{id}})$ into an encryption under the public key $pk \cdot pk'$. This is done by using $-sk'$ as the re-encryption key so that $sk+sk'$ is the new decryption key. The cloud sends the resulting encrypted metadata $\mathcal{E}.\mathsf{Enc}_{pk\cdot pk'}(g_4^{\mathtt{id}})$ and the key pk' to the proxy.
4. For every packet to be inserted in level l, the proxy computes the ξ Bloom filter positions that will have to be set to one. He does this by first multiplying the element $\mathcal{E}.\mathsf{Enc}_{pk\cdot pk'}(g_4^{\mathtt{id}})$ with $\mathcal{E}.\mathsf{Enc}_{pk\cdot pk'}(g_3^{k_i})$ for $i = 1 \ldots \xi$. Then he uses the proxy re-encryption property to transform the encryption into an encryption under the public key pk'. This is done by using sk as the re-encryption key. He sends the resulting list of elements $\mathcal{E}.\mathsf{Enc}_{pk'}(g_3^{k_i} g_4^{\mathtt{id}})$ back to the cloud.
5. The cloud initializes a Bloom filter of size $b(l)$ with all values set to $\mathcal{E}.\mathsf{Enc}_{bfpk}(0)$. For every element of the received list, the cloud decrypts the elements received in the previous step (since now this is only encrypted under the public key pk') and sets the Bloom filter positions $g_3^{k_i} g_4^{\mathtt{id}} \bmod b(l)$ to $\mathcal{E}.\mathsf{Enc}_{bfpk}(1)$.
6. The cloud sends the resulting encrypted Bloom Filter to the proxy and discards the key pair (sk', pk').

Phase 2: Update. In this second phase the packets of levels $l-1$ and l are merged into level l, re-randomized and randomly permuted. The global counter c and the level's reshuffle counter are updated on the proxy side and the packets' new indices are computed. In detail, this is done as follows:

1. All packets currently stored on level $l-1$ are moved to level l.
2. The cloud creates a new pair of temporary random ElGamal keys (sk', pk') and a pair of random BCP keys (ssk', ppk'). For every packet (c_1, c_2, c_3, c_4) stored on this new level, he first removes c_1 and transforms the parts (c_2, c_3) into encryptions under the public key $pk \cdot pk'$ by using $-sk'$ as the re-encryption key. Further, he transforms c_4 into an encryption under the public key $ppk \cdot ppk'$. He sends the transformed parts (c_2, c_3, c_4) of each packet to the server, along with the public keys pk' and ppk'.
3. The proxy increases the counters c and $r(l)$ and computes the level's salt $\mathcal{K}(r(l)||l||c)$. For each received element, he then multiplies the packet's c_2 part by $\mathcal{E}.\mathsf{Enc}_{pk\cdot pk'}(g_1^{\mathcal{K}(r(l)||l||c)})$. This results in the new correct index c_1, but now encrypted under $pk \cdot pk'$. The proxy then removes the dependency on the pk from these encrypted indices by using sk as the re-encryption key. Finally, he randomly permutes and re-randomizes all obtained elements (c_1, c_2, c_3, c_4) and sends them back to the cloud.
4. The cloud first decrypts the index c_1 of every received packet using the secret key sk'. He then removes the dependency on pk' from the parts (c_2, c_3) and the dependency on ppk' from part c_4 of all packets by using sk' and ssk', respectively, as the re-encryption keys. The result of this process is a randomly permuted list of packets with fresh indices, where (c_2, c_3) are encrypted metadata under pk and c_4 is the actual encrypted block under ppk.

Note that since after each access, the investigator puts back on the ORAM's top level a fake and a real packet, duplicate real packets will occur. However this does not affect the correctness of the protocol, since the most recently updated duplicate will be found on a higher ORAM level. Similarily the security of the protocol is not affected since every duplicate is indistinguishable from any other packet.

5. SECURITY ANALYSIS

In this section we analyze the security of the ORAM construction proposed in Section 4. We start by proving that the architecture is correct: for every block request, our construction returns the correct block back to the investigator. Then we show that our construction is secure, meaning that the access patterns remain hidden from the ORAM server (the cloud) and the proxy, while the investigator learns only the encrypted block. For the proofs of the propositions presented here, we refer to the paper's full version.

In our security analysis we assume that all the parties are semi-honest (i.e., they read and try to extract information from the messages they exchanged throughout the protocol, but do not tamper with the messages) and non-colluding. Clearly the non-collusion assumption cannot be lifted, since any collaboration between the proxy (who has the keys for the encrypted blocks) and the cloud would immediately render the protocol insecure.

PROPOSITION 5.1. *The ORAM protocol described in Section 4 is correct, i.e. for every query, the ORAM server (cloud) returns only the correct (queried) block except with negligible probability.*

Before we can prove our protocol secure we must formally define the notion of an access pattern and what it means for a participating member of the protocol to be oblivious. We do this with the help of the following definitions.

DEFINITION 1 (VIEW). *For a member $\mathcal{P}$ participating in our ORAM protocol we call the transcript of the protocol for an operation $\mathcal{R}$, that $\mathcal{P}$ sees, with $\mathcal{R}$ being a* `read`*, a* `write` *or a* `reshuffle`*, the* view $V^{\mathcal{P}}(\mathcal{R})$ *of the participant $\mathcal{P}$ on operation $\mathcal{R}$.*

Note that the transcript includes all messages that a participating member of our protocol sees during an execution of the protocol. This includes not only the messages that this party sends and receives, but also all the data that this party stores and has stored during previous executions of the protocol. Under this light we can now introduce the following definition:

DEFINITION 2 (ACCESS PATTERN). *Suppose that a sequence of operations $\vec{\mathcal{R}} = (\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_k)$, is issued by a party $\mathcal{P}$. We call the tuple $A^{\mathcal{P}}(\vec{\mathcal{R}}) = (V^{\mathcal{P}}(\mathcal{R}_1), V^{\mathcal{P}}(\mathcal{R}_2), \ldots, V^{\mathcal{P}}(\mathcal{R}_k))$, the* access pattern *of $\mathcal{P}$ for the operations $\vec{\mathcal{R}}$.*

DEFINITION 3 (OBLIVIOUS PARTY). *We call a participating member $\mathcal{P}$ of our ORAM protocol* oblivious*, if any two access patterns $A^{\mathcal{P}}(\vec{\mathcal{X}})$ and $A^{\mathcal{P}}(\vec{\mathcal{Y}})$ of the same length starting from the same configuration of the ORAM, are computationally indistinguishable.*

PROPOSITION 5.2. *The ORAM protocol described in Section 4 is secure (in the random oracle model), i.e. the cloud and the proxy are oblivious parties.*

6. DATA ANALYSIS

General Framework. Recall that in our scenario we assume that a client uploaded her fully sequenced encrypted DNA in blocks of fixed size to the cloud. She then allows a third party (such as a medical facility or a private physician), here called an investigator, to access some of these blocks. Suppose then that the investigator wants to perform a certain query on DNA strings for single or multiple clients. He first retrieves all blocks on which the query depends using the methods from the previous section, thus hiding the access pattern. These retrieved blocks are taken as input for secure two-party computation that is run between the cloud and the proxy. For simplicity, we follow the garbled circuit-based approach, but a strategy based on homomorphic encryption could be employed as well.

Let $\{\mathcal{B}.\mathsf{Enc}_{ppk}(\mathtt{data}_i)\}_{i=1}^{k}$ be the blocks that the investigator retrieved from the cloud during the first step. In order to evaluate a specific functionality f on these blocks, the investigator first converts the encryption into shares as follows: he blinds the retrieved encrypted data with a random value and sends it to the proxy, while the blinding values are sent to the cloud. The proxy now decrypts the obtained blocks and is left with blinded versions of the required data items. Since the cloud has the blindings, this means that the proxy and the cloud have a secret sharing of the data contained in the blocks. Subsequently the investigator creates the circuit corresponding to the functionality f he wants to compute (which needs to include the necessary data unblinding step) and sends it to the proxy who performs the circuit garbling. Therefore the cloud and the proxy can now evaluate the desired function on the secret shared blocks together using any garbled circuit framework. Finally, the result of the computation is returned to the investigator.

We illustrate this general procedure with three specific queries, often performed in the area of genetic analysis.

Pattern Matching in SNPs. In our architecture it is straightforward to search for a specific mutation in a block of sequenced DNA; this conforms to the present practice of finding markers via the SNPs discussed earlier in the introduction. Here, the investigator first retrieves the encrypted block that contains the SNP that is to be analyzed. Subsequently he runs a simple comparison protocol that compares the SNP part of the block to the desired mutation. Technically, the circuit for this operation needs to unblind the retrieved block, extract some nucleotides and perform the comparison operation. We used the compiler from [23] and representing each nucleotide by two bits in a DNA block, we produced the corresponding circuit, which consists of $20,490$ gates. Using a framework like [33], which evaluates a gate per 7.25 nsec, this circuit can be evaluated in $1\mu s$.

DNA Fingerprints in Forensics. Usage of genomic data in criminal forensics has become a wide-spread tool [34]. It is based on the identification of short tandem repeats (STRs) which are highly polymorphic regions of short repeated sequences of DNA – each around four nucleotides long. Typically, investigators focus on several loci[2] in the genome, looking for several STRs. Then, the number of repeated copies is extracted and used as a "description vector" for an individual or a probe found at a crime scene. We stress that there is conclusive evidence that STRs reveal information about family members and kinship [35].

Note that this procedure is related but substantially different from the above SNP pattern matching for two reasons: the output is a vector of integers representing the amount

[2]For instance, the US Combined DNA Index System (CODIS) maintained by the FBI uses 13 such loci.

of copies of individual STRs and the loci are not necessarily in vicinity but scattered through the *entire* genome.

Initializing the parameters of our system as will be discussed in Section 7 and thus storing 128 character long DNA blocks, we implemented the above approach using [23]. The resulting circuit that outputs the "description vector" is of size $1,697,280$ gates, which again using the framework from [33] can be evaluated in roughly 1 second.

Statistical Queries. Our framework also supports computation tasks on data coming from multiple patients: Assume that n clients have their fully sequenced, encrypted DNA stored on the cloud (each in an individual ORAM) for example along with an encrypted table for each client, containing information on whether the client suffers from a certain disease or not. The investigator now wants to evaluate the probability that specific DNA mutation patterns – potentially distributed throughout the entire genome – are associated with this disease.

To this end, one can perform a statistical analysis: Consider the events $A \in \{0,1\}$ stating that the patient suffers $(A = 1)$ or does not suffer $(A = 0)$ from a particular disease, and $B_i^p \in \{0,1\}$ stating that a distinctive pattern p occurs in DNA block indexed by i. Note, that p and i do not necessarily contain all possible and thus combinatorially many patterns of nucleotides or DNA fragments, but rather a small and well understood subset.

A biostatistician, as the investigator, knows the disease classifications A and wants to train a statistical model for future prediction of disease prevalence. He can thus compute the prior probabilities $\mathsf{Prob}(B_i^p|A)$ for a given set of stored genomes – taken as a training set. Furthermore, he can compute the prior probability distribution $\mathsf{Prob}(A)$ on the prevalence of the disease in the sample. Using our architecture he can obtain all probabilities $\mathsf{Prob}(B_i^p)$ for $B_i^p \in \{0,1\}$ – without revealing the individual genomes. Now, using Bayesian estimation, a medical practitioner who was provided with the biostatistician's model can diagnose a patient by computing the probability $\mathsf{Prob}(A|B_i^p) = \frac{\mathsf{Prob}(B_i^p|A)\cdot\mathsf{Prob}(A)}{\mathsf{Prob}(B_i^p)}$ given a genome of a patient. Note, that updates on the model can be performed as well: whenever new genomes enter the pool of genomic data in the cloud (preferably many), the biostatistician can use Bayesian updating to accommodate the new data and modify the effective model $\mathsf{Prob}(A|B_i^p)$.

In order to estimate the complexity of this approach, we implemented the computation of the above-mentioned probabilities on $1,000$ retrieved encrypted DNA data blocks using fixed point arithmetic with 10 bit precision. The resulting circuit consists of $21,989$ gates for one block, out of which $21,590$ gates were used to perform the unblinding of the $2,048$ bit long blinded block. Since the circuit creation scales linearly, the resulting circuit for $1,000$ DNA blocks has a size of about 21 million gates, which can be evaluated in 1.2 seconds using the framework from [33]. Note that this number amounts for the time required to perform computations on the retrieved blocks; timings for the data retrieval part are given in the next section.

7. EXPERIMENTAL RESULTS

We experimentally evaluate the performance of out ORAM construction, initiated with a fully sequenced genome.

Parameters. We follow the recommendations suggested by EcryptII [36], which are slightly more conservative than those of NIST. Thus we choose for the ElGamal group a safe prime $p = 2\cdot q\cdot r+1$ of length $1,024$ bits with q and r primes and set the discrete logarithm key of size q, to 160 bits. In order then for the CvHP hash function to be compatible with the ElGamal encryption we choose the element α (as mentioned in Section 2) to be a generator of the subgroup $\mathbb{Z}_q$ and the generator β to be a multiple of α. The block ids are drawn uniformly and at random from $\mathbb{Z}_q^*$ making sure that every block (fake and real) is assigned a different id. We choose the BCP keys to be of size $4,096$ bits, thus having the underlying prime factors of size $1,024$ bits. Regarding our Bloom filters and in order to minimize the disk seeks during the queries, we set the number of Bloom filter keys to 5, randomly chosen from $\mathbb{Z}_q^*$. The Bloom filter for every level is of size chosen using the results from [28] so that the security concerns raised from [27] can be mitigated and the false positive ratio of the Bloom filter[3] is smaller than 2^{-12}.

Experimental Setup. We simulated our mechanism on an *IBM System x3550 M3* server equipped with *32GB of RAM* and two *Intel Xeon X5650 2.67GHz* processors running *Ubuntu Server 11.10*. Our system was implemented in C++ and compiled using *g++* from the *GNU Compiler Collection* version 4.6.1. For the cryptographic operations, we used the *Crypto++* library in version *5.6.1*. We used MySQL version 5.1.69 as a storage backend.

Results. A full human genome can be encoded using approximately 3 billion characters. Needing only two bits to encode a nucleotide, storing the entire genome would need approximately 2^{32} bits. The BCP parameters described above support plaintext blocks of size at most $2,048$ bits, meaning that we could store a fully sequenced human genome in less than $2^{25} = 33,554,432$ blocks of 128 characters each. We simulated the client's DNA, by storing random numbers to represent the various DNA blocks. We note that our implementation does not initially put a Bloom filter into the last level N. Since all elements (potentially old versions) reside there anyway due to the ORAM's initialization, a Bloom filter is only needed after 2^N queries when a reshuffle of the last level is performed which automatically creates the Bloom filter (see the paper's full version for details).

Our system's bottleneck is its initialization: Populating databases of various sizes scales linearly: a $2^{17} = 131,072$ blocks database needed approximately 2 hours and 36 minutes, a $2^{24} = 16,777,216$ blocks database needed approximately 7 days and the one holding all 2^{25} real blocks (representing a fully sequenced genome) needed approximately 14 days to be populated. Nevertheless, as this is an operation performed once during a client's lifetime, these durations seem justified.

We set up two databases of different size in order to test the block retrieval efficiency: a database that contains 2^{17} blocks and the large database containing 2^{25} blocks. We performed 10 rounds of $2,100$ queries in both databases. In the upper graph of Figure 2, we show the average timings for the 10 rounds of queries on the large database: We see that the majority of queries run in less than 100 seconds. Note that this scenario resembles the case of storing a full genome. The peaks in this graph occur whenever a reshuffle is performed. Observe however the periodicity of the data:

[3]Note here that the probability of a false positive does not affect the correct retrieval of a block, as pointed out in Sections 4.2 and 5

The time needed for a reshuffle of a higher level amounts to twice that of the previous level. However since the elapsed time between two reshuffles increases exponentially, a low amortized time of approximately 12.39 seconds per query is achieved. In the lower graph of Figure 2 we compare the average time needed for 10 repetitions of the $2,100$ queries in the two databases. From the results we see that the query time is roughly independent of the size of the database and depends only on the number of the queries performed so far. This is expected, as at any given time, not all the ORAM levels are full.

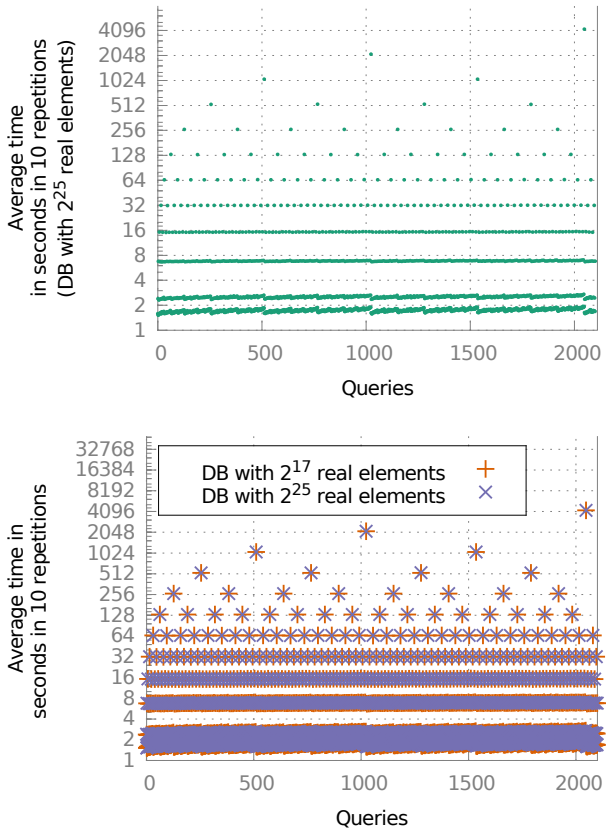

Figure 2: Average time of $2,100$ queries on a database with 2^{25} real packets (top); average time for $2,100$ queries on the the two databases holding 2^{17} and 2^{25} real packets respectively (bottom).

To measure the traffic exchanged between the parties we performed $2,100$ queries locally on the loopback interface using *iptables* for traffic accounting, thus the numbers also include all TCP and IPv4 headers and not just the application layer payload. The top graph of Figure 3 depicts the summed traffic between the cloud and the proxy, showing certain "jumps". These occur whenever a reshuffle is performed and as in the time measurement results, each jump's height is twice that of the previous one, with an exponential amount of queries being performed from one jump to the next. The cloud's communication overhead is due to sending of the Bloom filter in step 6 of the Bloom filter creation phase. During the $2,048$th query, the proxy sends 318MB to the cloud with an average of only 1.202MB transferred per query between the proxy and the cloud. The lower graph

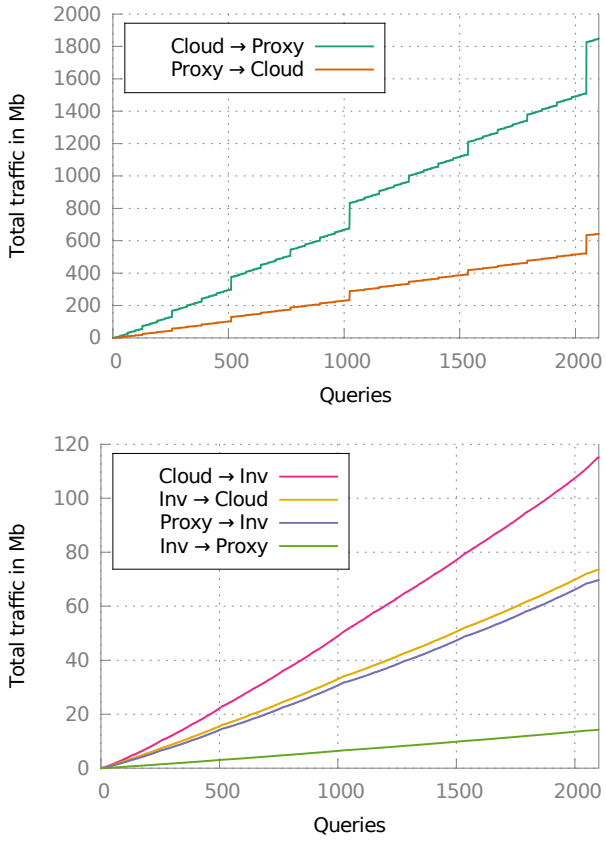

Figure 3: Traffic exchanged between cloud and proxy (top) and cloud, proxy and investigator (bottom) in $2,100$ queries.

of Figure 3 shows the traffic between the investigator and the other parties, with an average of 133KB per query. Note that the traffic is only affected by the number of queries performed and not by the number of items in the database. Our results show that the investigator is almost not burdened by the query, as required by the problem domain.

8. CONCLUSION

In this paper we presented a solution which synergistically combines ORAM techniques with secure two-party computation solutions in order to offer privacy preserving computations on outsourced, fully sequenced DNA providing full query flexibility. Our starting point was storing the fully sequenced DNA in small blocks and outsource it to a remote server, hiding at the same time the access patterns. Our main building block was a new ORAM construction that allows the data owner to be offline even when a reshuffle is performed. Using secure computation on the retrieved DNA blocks, the computation can be performed in an oblivious manner. Decoupling the data retrieval and computation process, we obtain full flexibility to adapt to future queries.

9. ACKNOWLEDGMENTS

This work is partly supported by CASED, EC SPRIDE and by the THeCS project as part of the Dutch national program COMMIT.

10. REFERENCES

[1] A. Philippe, Martinez *et al.*, "Genome-wide scan for autism susceptibility genes," *Human Molecular Genetics*, vol. 8, no. 5, pp. 805–812, 1999.

[2] M. Franz, S. Katzenbeisser, B. Deiseroth, K. Hamacher, S. Jha, and H. Schröder, "Towards secure bioinformatics services," in *FC*, ser. LNCS 7035. Springer, 2011, pp. 276–283.

[3] G. Lauss, C. Schröder, P. Dabrock, J. Eder, K. Hamacher, K. Kuhn, and H. Gottweis, "Towards biobank privacy regimes in responsible innovation societies," *Biopreservation and Biobanking*, vol. 11, pp. 319–323, 2013.

[4] D. H. Abrahams, Brett S.and Geschwind, "Advances in autism genetics: on the threshold of a new neurobiology," *Nat Rev Genet*, vol. 9, no. 5, pp. 341–355, May 2008.

[5] J. R. Troncoso-Pastoriza, S. Katzenbeisser, and M. U. Celik, "Privacy preserving error resilient DNA searching through oblivious automata," in *CCS*, 2007, pp. 519–528.

[6] M. Blanton and M. Aliasgari, "Secure outsourcing of DNA searching via finite automata," in *DBSec*, 2010, pp. 49–64.

[7] K. B. Frikken, "Practical private DNA string searching and matching through efficient oblivious automata evaluation," in *DBSec*, 2009, pp. 81–94.

[8] L. Wei and M. K. Reiter, "Third-party private DFA evaluation on encrypted files in the cloud," in *ESORICS*, 2012, pp. 523–540.

[9] ——, "Ensuring file authenticity in private DFA evaluation on encrypted files in the cloud," in *ESORICS*, 2013, pp. 147–163.

[10] F. Bruekers, S. Katzenbeisser, K. Kursawe, and P. Tuyls, "Privacy-preserving matching of DNA profiles," *IACR ePrint*, vol. 2008, 2008.

[11] J. Katz and L. Malka, "Secure text processing with applications to private DNA matching," in *ACM CCS*, 2010, pp. 485–492.

[12] P. Baldi, R. Baronio, E. D. Cristofaro, P. Gasti, and G. Tsudik, "Countering GATTACA: efficient and secure testing of fully-sequenced human genomes," in *ACM CCS*, 2011, pp. 691–702.

[13] E. Ayday, J. L. Raisaro, J.-P. Hubaux, and J. Rougemont, "Protecting and evaluating genomic privacy in medical tests and personalized medicine," in *WPES*, 2013, pp. 95–106.

[14] Y. Chen, B. Peng, X. Wang, and H. Tang, "Large-scale privacy-preserving mapping of human genomic sequences on hybrid clouds," in *NDSS*, 2012.

[15] O. Goldreich and R. Ostrovsky, "Software protection and simulation on oblivious RAMs," *J. ACM*, vol. 43, no. 3, pp. 431–473, 1996.

[16] E. Bresson, D. Catalano, and D. Pointcheval, "A simple public-key cryptosystem with a double trapdoor decryption mechanism and its applications," in *ASIACRYPT*, 2003, pp. 37–54.

[17] M. Blaze, G. Bleumer, and M. Strauss, "Divertible protocols and atomic proxy cryptography," in *EUROCRYPT*, 1998, pp. 127–144.

[18] A.-A. Ivan and Y. Dodis, "Proxy cryptography revisited," in *NDSS*. The Internet Society, 2003.

[19] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[20] D. Chaum, E. van Heijst, and B. Pfitzmann, "Cryptographically strong undeniable signatures, unconditionally secure for the signer," in *CRYPTO*, 1991, pp. 470–484.

[21] A. C.-C. Yao, "How to generate and exchange secrets (extended abstract)," in *FOCS*, 1986, pp. 162–167.

[22] Y. Lindell and B. Pinkas, "A proof of security of Yao's protocol for two-party computation," *J. Cryptology*, vol. 22, no. 2, pp. 161–188, 2009.

[23] A. Holzer, M. Franz, S. Katzenbeisser, and H. Veith, "Secure two-party computations in ANSI C," in *ACM CCS*, 2012, pp. 772–783.

[24] Y. Huang, D. Evans, J. Katz, and L. Malka, "Faster secure two-party computation using garbled circuits," in *USENIX Security*, 2011.

[25] P. Williams, R. Sion, and B. Carbunar, "Building castles out of mud: practical access pattern privacy and correctness on untrusted storage," in *ACM CCS*, 2008, pp. 139–148.

[26] P. Williams, R. Sion, and A. Tomescu, "PrivateFS: a parallel oblivious file system," in *ACM CCS*, 2012, pp. 977–988.

[27] E. Kushilevitz, S. Lu, and R. Ostrovsky, "On the (in)security of hash-based oblivious RAM and a new balancing scheme," in *SODA*, 2012, pp. 143–156.

[28] K. J. Christensen, A. Roginsky, and M. Jimeno, "A new analysis of the false positive rate of a bloom filter," *Inf. Process. Lett.*, vol. 110, no. 21, pp. 944–949, 2010.

[29] E. Stefanov, M. van Dijk, E. Shi, C. W. Fletcher, L. Ren, X. Yu, and S. Devadas, "Path ORAM: an extremely simple oblivious RAM protocol," in *ACM CCS*, 2013, pp. 299–310.

[30] M. Franz, P. Williams, B. Carbunar, S. Katzenbeisser, A. Peter, R. Sion, and M. Sotáková, "Oblivious outsourced storage with delegation," in *FC*, ser. LNCS 7035. Springer, 2011, pp. 127–140.

[31] D. Boneh, C. Gentry, S. Halevi, F. Wang, and D. J. Wu, "Private database queries using somewhat homomorphic encryption," in *ACNS*, ser. LNCS 7954. Springer, 2013, pp. 102–118.

[32] E. Stefanov and E. Shi, "Multi-cloud oblivious storage," in *ACM CCS*, 2013, pp. 247–258.

[33] M. Bellare, V. T. Hoang, S. Keelveedhi, and P. Rogaway, "Efficient garbling from a fixed-key blockcipher," in *IEEE S&P*, 2013, pp. 478–492.

[34] L. Roewer, "DNA fingerprinting in forensics: past, present, future," *Investigative Genetics*, vol. 4, no. 1, p. 22, 2013.

[35] F. R. Bieber, C. H. Brenner, and D. Lazer, "Finding criminals through DNA of their relatives," *Science*, vol. 312, no. 5778, pp. 1315–1316, 2006.

[36] "European network of excellence in cryptology II," http://www.keylength.com/en/3/.