A Study of Privacy Preserving Queries with Bloom Filters

Sara Ramezanian

Master of Science Thesis
January 2016

Department of Mathematics and Statistics
University of Turku

University of Turku
Department of Mathematics and Statistics

RAMEZANIAN, SARA: A Study of Privacy Preserving Queries with Bloom Filters
Master of Science Thesis, 59 p., 16 app. p.
Information Technology - Cryptography and Data Security Track
January 2016

This thesis focuses on the private membership test (PMT) problem and presents three single-server protocols to resolve this problem. In the presented solutions, a client can perform an inclusion test for some record x in a server's database, without revealing his record. Moreover, after executing the protocols, the contents of server's database remain secret.

In each of these solutions, a different cryptographic protocol is utilized to construct a privacy preserving variant of Bloom filter. The three suggested solutions are slightly different from each other, from privacy perspective and also from complexity point of view. Therefore, their use cases are different and it is impossible to choose one that is clearly the best between all three.

We present the software developments of the three protocols by utilizing various pseudocodes. The performance of our implementation is measured based on a real case scenario.

This thesis is a spin-off from the Academy of Finland research project "Cloud Security Services".

Keywords: Private information retrieval, Bloom filter, Private membership test, Homomorphic encryption, Blind signature, Oblivious pseudorandom functions.

Acknowledgements

Foremost, I would like to express my sincere gratitude to my supervisors Valtteri Niemi and Tommi Meskanen for giving me the opportunity to work in their group and for introducing me to the fascinating world of research. I am thankful for their kind supervision, patience, immense knowledge, and all their explanations and helpful comments on my work. It's been indeed a special privilege and a pleasure to work with them and learn from them, and I am looking forward to continuing our work together.

My sincere thanks also goes to Arto Lepistö for code-reviewing of the software development in this thesis and his valuable comments on how to write a professional code. Thanks as well to my dear friend Masoud Naderpour for providing technical guidance on my implementation and for all the fun we have had in the last three years.

Last but not least, I owe my gratitudes to my mother and my uncle Nosrat, for supporting me spiritually throughout my life. I would not be where I am today, if it wasn't for them.

List of Figures

List of Tables

List of Acronyms

AES	Advanced Encryption Standard
B	Bloom Filter
CB	Counting Bloom Filter
CPA	Chosen Plaintext Attack
CPIR	Computationally Private Information Retrieval
EB	Encrypted Bloom Filter
FP	False Positive
OPRF	Oblivious Pseudorandom Function
OT	Oblivious Transfer
PIR	Private Information Retrieval
PKS	Private Keyword Search
PMT	Private Membership Test
PRF	Pseudorandom Function
QNR	Quadratic Non-Residue
QR	Quadratic Residue
QNR$_n$	The Set of all Quadratic Non-Residues Modulo n
QR$_n$	The Set of all Quadratic Residues Modulo n
TH	Trusted Hardware
TTP	Trusted Third Party
VI-CB	Variable-Increment Counting Bloom Filter
1-2 OT	1 out of 2 Oblivious Transfer

Contents

1 Introduction

Development of the Internet and constantly improving electronic devices provide more services for the users. Every day more and more people are utilizing publicly accessible databases to gain information. However, while users query search engines, curious service providers can collect their users' personal information. By studying these collections, users' interests, lifestyles and other confidential information can be inferred. These leakages of personal data assist advertisement companies to perform better and therefore obtain more profit [1].

However, further exploitation is possible from the collected information. Seneviratne et al. have shown that a single snapshot of installed applications on users' smart phones can leak a lot of personal information about them with 90% precision. These information consist of relation status, religion, spoken languages, countries of interest, and whether the user is a parent of small children [2]. Another study by Kosinski et al. demonstrates that digital records of behavior can predict accurately many sensitive personal attributes such as ethnicity, religious and political views, personality traits and parental separation [3].

These sort of studies show that it is necessary to protect users' privacy on the Internet and motivate many researchers to look for possible solutions to the issues of privacy. There are plenty of well-known concepts in cryptography, which are proposed to preserve the privacy of Internet users such as *Oblivious Transfer* [4] and *Private Information Retrieval* [5].

The aim of this chapter is to present the background of our research and help the reader to have an overview on our work.

1.1 Background

A protocol to transfer a piece of information from a sender S to a receiver R is called *oblivious transfer* (OT) if after executing the protocol, sender remains unaware which part of information (if any) has been received by R. In Rabin's oblivious transfer, R gets S's secret message with probability of 1/2 [4]. Another form of OT is 1-*out-of-n oblivious transfer*. It can be described as a game between two players: S who owns n secrets $x_1, x_2, ..., x_n$ and R who holds an index $i \in \{1, 2, ..., n\}$. At the end of the game, S has no clue about index i and the only information R receives is x_i. While in OT protocols secrecy is required for both parties (database provider and user), the main purpose of all *Private Information Retrieval* (PIR) schemes is just to protect user's privacy. In this regard, PIR is a weaker version of 1-out-of-n oblivious transfer.

In general, PIR techniques enable users to query through databases without revealing which piece of information they are interested in. The trivial approach to PIR problem is to deliver a copy of the database to each user, therefore one can easily retrieve data privately. However this solution is impractical due to the high usage of bandwidth.

Before going any further, with the purpose of making the problem more concrete, we assume that the database is a string of n bits: $x_1x_2...x_n$ and the user is interested in ith bit x_i where $i \in \{1, 2, ..., n\}$. With this assumption the communication complexity (the minimum number of bits exchanged between database and user) of the above naive solution is $O(n)$.

The first non-trivial PIR scheme was introduced in 1995 by Chor et al. [6]. In their proposed solution, the database should be replicated k times ($k \geq 2$) and held by different servers which are not allowed to communicate to each other. The communication complexity of this solution is $O(n^{1/k})$. Although the scheme of [6] is information theoretically secure (i.e. an adversary cannot break it even if he has unlimited computing power) it is communicatively expensive. Also, the replicated databases increase the security risk.

In 1997, the first single-server computationally-private information retrieval (CPIR) is presented by Kushilevitz and Ostrovsky [7]. The communication complexity of their CPIR scheme is less than $O(n^{\epsilon})$ for any $\epsilon > 0$. Afterwards, many models of single-server CPIR are proposed in academia with different communication complexity [8].

PIR schemes assume that user knows the index i of his element in the database. This assumption makes PIR protocols sometimes unrealistic. In many cases an Internet user holds a search-word (for instance, the user is interested in gathering information about a specific university), instead of the physical address of the sought item in the database (index i). In this scenario, the database is a set of n pairs $\{(k_1, v_1), ..., (k_n, v_n)\}$ where k_i is a keyword and v_i is a value. A private query through this database is called *Private Keyword Search* (PKS) by Chor et al. [9].

Let's assume that the database is a set of records and the aim of user's query is to check whether a certain record belongs to this database. *Private Membership Test* (PMT) protocol enables to decide about this set inclusion in a private form. Despite of many efforts that have been done on OT, PIR and PKS, researchers have been less interested in PMT. To explain the necessity of PMT protocols in the field of secure computation, consider the following situation: A security company has a collection of known malware. One client of this company wants to install an application and he wants to know whether this application is malicious or not. As mentioned before, if the client discloses to the database holder which application he is interested in, then unwanted information about the client may be given away.

1.2 Contributions

In order to preserve client's privacy in membership tests, we want to design a single-server protocol which allows the interaction between client and server's database in a private manner. As in most scenarios, service providers' desire is to keep the total content of their databases secret and thus our protocol should not reveal information about other members in the database, except the one item that client is seeking for. For this purpose, one of the fundamentals of our work is a semi-honest adversary model. It means we have the assumption that a participant is following the protocol as expected, but may attempt to acquire further information than what is intended during protocol execution time. This model is also known as 'honest but curious' model.

We introduce three protocols for the PMT problem. The first protocol is based on Goldwasser-Micali homomorphic encryption and was presented for the first time in our paper in 2015 [10]. Protocols two and three utilize blind RSA signature and oblivious pseudorandom function, respectively. The idea behind the last two protocols is adapted from previous publication by Nojima and Kadobayashi at 2009 [11]. But the protocols are slightly modified to be fully functional and more secure.

From privacy perspective, each of these three protocols is slightly different from the others, so that their use cases are different. This makes it impossible to choose one that is clearly best between all three.

In all three protocols, we assume that the party who hosts the server, stores its database X in a space-efficient probabilistic data structure which is called *Bloom filter* [1].

We also assume that in these protocols, there are two parties who engage in an interactive communication:

- A **Server** S who possesses a database set X.
- A **Client** C who wants to test if an element x belongs to X.

The concept of our PMT protocols is summarized below:

Private Membership Test Algorithm

```
1: Prerequirements: $S$ stores its database $X$ in a Bloom filter (B) with $l$
   independent hash functions $H_i$ where $i = 1, 2, ..., l$.
2: Prerequirements in some protocols: $S$ picks a cryptosystem and encrypts
   the B to get EB.
3: Server part: $S$ sends B or EB, its $l$ hash functions, a hash function $H$,
   and possible other parameters to $C$.
4: Client part: $C$ queries his $x$ in B or EB (instead of the database $X$) and
   finds the corresponding indexes in the filter.
5: The result: $C$ and $S$ engage in an interactive communication and as a
   result; $C$ with the help of $S$ verifies: $[B(H_1(x)) = 1] \wedge ... \wedge [B(H_l(x)) = 1]$
   ($H_i(x)$ is an index in the Bloom filter and $B(H_i(x))$ is the value of the
   bit located in that index.)
     • If True then probably $x \in X$.
     • If False then surely $x \notin X$.
```

The encrypted Bloom filter itself would not reveal any information about server's database to the client or any third party. This guarantees S's secrecy. In all three protocols, the amount of information transferred between S and C is minimal in the sense that the transfered data would not reveal any extra information above what is intended. Also based on our implementations all three protocols are reasonably fast.

The three protocols, their implementations and comparison between them are the main contributions of this thesis.

[1]The notion of Bloom filter is fully explained in chapter two.

1.3 Outline

Chapter 2 familiarizes the reader with the notations and theoretical background and introduces the cryptographic protocols used in this work. In Chapter 3, we explain a privacy preserving variant of Bloom filters in the form of three different protocols. Chapter 4 gives the design ideas behind software development for the three protocols in details and shows the results of measurements in our implementations. Then, on Chapter 5, we compare the security and efficiency of the given solutions for the PMT problem. In Chapter 6, we describe alternative approaches to solve the PMT and PIR problems. On Chapter 7, we conclude this thesis with an outlook to further work.

2 Fundamentals

This chapter serves as a brief introduction to fundamental notions used in this thesis. It describes common notations, definitions and cryptographic protocols utilized later in the concepts of privacy preserving queries.

2.1 Notation

Let us begin with some basic notations of number theory. The set, consisting of equivalence classes modulo n, is denoted as $\mathbb{Z}_n = \{0, 1, ..., n-1\}$. Moreover, $\mathbb{Z}_n^* = \{a \in \mathbb{Z}_n \mid gcd(a, n) = 1\}$. This implies if n is a prime, then $\mathbb{Z}_n^* = \{1, 2, \cdots, n-1\}$.

If an integer $q \neq 0$ is congruent to a square modulo n, that means there exists an integer x such that: $x^2 \equiv q \pmod{n}$. Such an integer q is called a quadratic residue (QR) modulo n. If q is not a QR modulo n then it is called a quadratic non-residue (QNR) modulo n. The set of all quadratic residues modulo n is denoted by QR_n and consequently the set of all QNR modulo n is denoted by QNR_n.

Let a be an integer and p be an odd prime number. Then the *Legendre symbol* is a function of a and p, with value of $-1, 0$ or 1 defined as:

$$\left(\frac{a}{p}\right) = \begin{cases} -1 & \text{if } a \text{ is a quadratic non-residue modulo } p, \\ 1 & \text{if } a \text{ is a quadratic residue modulo } p \text{ and } a \not\equiv 0 \pmod{p}, \\ 0 & \text{if } a \equiv 0 \pmod{p}. \end{cases}$$

The Legendre symbol is *homomorphic* with respect to multiplication, i.e.

$$\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right).$$

Let a be an integer and n be an odd integer such that $n = p_1^{k_1} p_2^{k_2} \cdots p_m^{k_m}$ where the integers p_i are different primes. The *Jacobi symbol* of a and n is the generalization of Legendre symbol and defined as:

$$\text{Jacobi}(a, n) = \left(\frac{a}{p_1}\right)^{k_1} \left(\frac{a}{p_2}\right)^{k_2} \cdots \left(\frac{a}{p_m}\right)^{k_m}.$$

In formal language theory, *string concatenation* is the operation of joining two strings end-to-end together and is denoted as $||$. For instance, if $x = 1001$ and $y = 00$ then $x||y = 100100$.

The *encryption* of a message m with a key k to generate a ciphertext c is denoted as $c = Enc_k(m)$. The corresponding *decryption* is denoted as $m = Dec_k(c)$.

2.2 Basic Concepts

In this work, some basics of cryptography, computation and complexity, like boolean circuits and asymptotic complexity (denoted as $O(n)$) are assumed to be known. The readers who are interested in more details can use *Handbook of Applied Cryptography* [12] and *Encyclopedia of Cryptography and Security* [13] as the references.

In cryptography, *digital signature* is used to demonstrate authentication, integrity and nonrepudiation for digital messages. In other words, it gives a receiver proof that the message was sent by a particular known sender.

Any efficiently computable function with arbitrary input lengths and fixed output length is called a *hash function.* For instance, SHA-1 maps any message to a 160-bit hash value. Cryptographically secure hash functions have also the properties of preimage resistance, 2nd preimage resistance and collision resistance.

Homomorphic Encryption is a form of encryption that allows computations to be done on ciphertexts, without decrypting them first. For instance, if $(G_1, \star)$ and $(G_2, \cdot)$ are two groups, x_1 and x_2 belongs to G_2, and $Enc_k(x_1)$ and $Enc_k(x_2)$ are from G_1 then

$$Enc_k(x_1) \star Enc_k(x_2) = Enc_k(x_1 \cdot x_2).$$

Bloom Filters are probabilistic space-efficient data structures that are used to do a membership test. They were introduced by Burton Bloom at 1970 [14]. Bloom filters are suitable to store enormous databases. They have strong space and time advantages compare to other data structures to represent a set. More specifically a Bloom filter is an array of m bits which are initially all set to 0 (figure 1 picture a). There are l different hash functions defined for the Bloom filter to map set element to array positions randomly with a uniform distribution. To add an element to the filter, one should feed the element to l hashes and get l positions, then change those positions to 1 (figure 1 picture b).

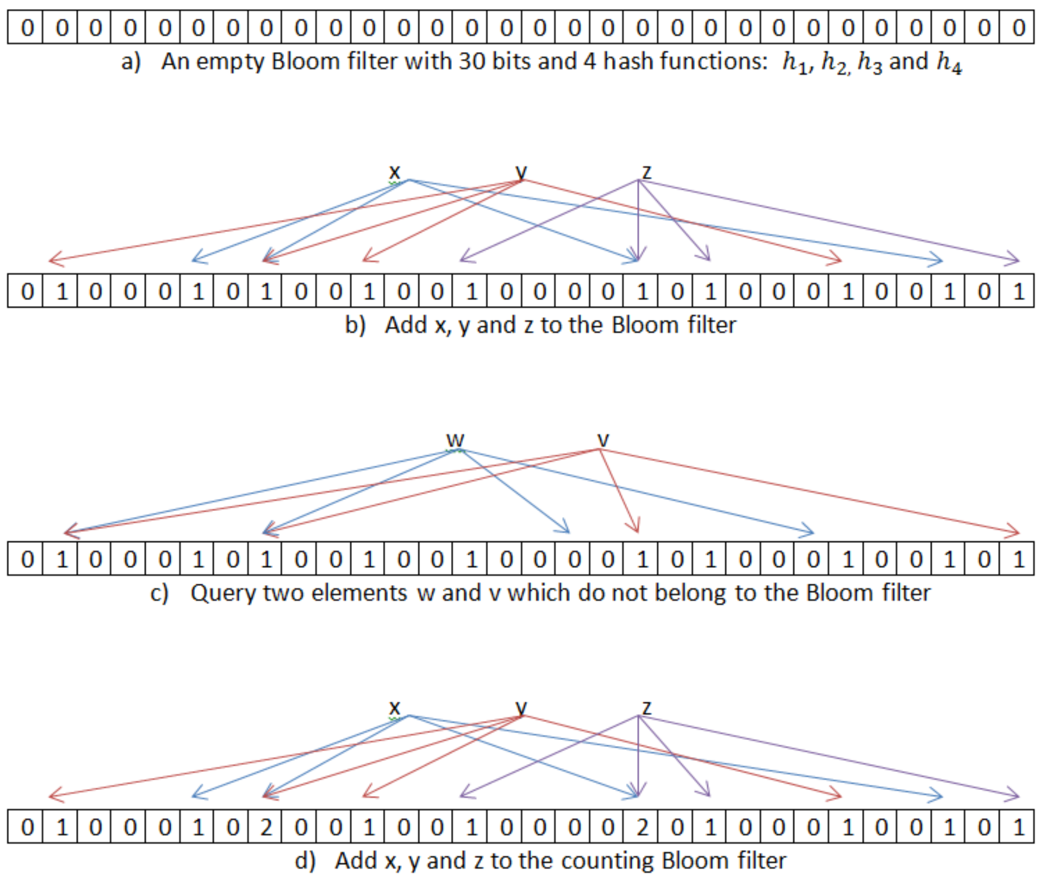


Figure 1: An example of Bloom filter and counting Bloom filter

To query an element from a Bloom filter (B), one should calculate all the hash values of that element to obtain l array positions of the filter. If every one of those l positions have a value 1, then this query results in positive.

But sometimes a query through this filter results in false positive (FP). For instance, in figure 1(c), we query v and w through a Bloom filter which represents the set $X = \{x, y, z\}$. Despite the fact that v does not belong to this set, the array positions corresponding to hash values of v, were all set to 1 because of some members of the set X. Because of this possibility a query from a Bloom filter, returns either "possibly in the set" or "definitely not in the set".

Let m be the number of bits in the filter and n denote the size of set which is inserted to B. To compute the probability of the false positive, we assume that a hash function selects each entry of B with the same probability of $1/m$. Consequently, after applying one hash function to one member of X, a specific bit of B is not set to one with the probability of $1 - 1/m$. After inserting n elements to B, the probability that none of l hash functions set a certain bit to one is

$$\left(1 - \frac{1}{m}\right)^{nl}$$

and therefore a given bit of the Bloom filter is set to one with the probability of

$$1 - \left(1 - \frac{1}{m}\right)^{nl}.$$

This helps us to calculate the false positive error rate of the Bloom filter. If all the l array positions of an element that is not part of the set, are set to one then FP happens and the probability of this event, with the approximate equality of $(1 - 1/m) \approx e^{-1/m}$ is

$$\left(1 - \left(1 - \frac{1}{m}\right)^{nl}\right)^{l} \approx (1 - e^{-nl/m})^{l}.$$

The false positive error rate of a Bloom filter, depends on m, n, and l. To find the minimal false positive rate, $(1 - e^{-nl/m})^l$ should be minimized with respect to l [15]. Let $f = (1 - e^{-nl/m})^l$ then

$$\frac{\partial f}{\partial l} = (1 - e^{-nl/m})^{l} \left(\ln(1 - e^{-nl/m}) + \frac{\frac{nl}{m} e^{-nl/m}}{1 - e^{-nl/m}}\right) = 0$$

and it follows that

$$(1-e^{-nl/m})^l = 0 \text{ or } \left(\ln(1-e^{-nl/m}) + \frac{\frac{nl}{m}e^{-nl/m}}{1-e^{-nl/m}}\right) = 0$$

If $(1-e^{-nl/m})^l = 0$ then $l = 0$ and this is not an acceptable solution. Therefore,

$$\left(\ln(1-e^{-nl/m}) + \frac{\frac{nl}{m}e^{-nl/m}}{1-e^{-nl/m}}\right) = 0$$

and it follows that

$$(1-e^{-nl/m})\ln(1-e^{-nl/m}) + \frac{nl}{m}e^{-nl/m} = 0\cdot$$

Let $U = e^{-nl/m}$. Then $\ln(U) = -nl/m$ and

$$(1-U)\ln(1-U) + (\ln(\frac{1}{U}))U = 0$$

hence

$$\ln\left((1-U)^{(1-U)} \cdot U^{-U}\right) = 0$$

and it follows that

$$(1-U)^{(1-U)} \cdot (U^{-U}) = 1$$

hence

$$(1-U)^{(1-U)} = U^U$$

and one solution of this is $U = 1/2\cdot$

Finally, the optimal number of l is accomplished which is

$$l_{opt} = \frac{m}{n}ln2 \approx \frac{9m}{13n}\cdot$$

Once the set has been inserted to the Bloom filter, adding more elements is possible but deletion is not. It is possible to solve this issue by using a counting Bloom filter .

In a *Counting Bloom Filter* (CB) the arrays consist of n-bit counters (see figure 1 picture d). This data structure performs in a similar manner as a normal Bloom filter; except that CB is able to keep track of the number of element insertions by increasing the counters. This enables us to delete elements by decreasing the counters [16].

The analysis from Fan et al. in [17] shows that in typical use cases that appear in practice, 4 bits per counter in a counting Bloom filter, which uses the optimal number of hash functions, should be sufficient to prevent counters overflow. This analysis is described in the following.

Let $c(i)$ be the count corresponding the ith counter. We assume that the counting Bloom filter has m counters. The probability that the ith counter is increased j times can be estimated as follows:

$$P(c(i) = j) = \binom{nl}{j}(\frac{1}{m})^j(1 - \frac{1}{m})^{nl-j} \leq \binom{nl}{j}(\frac{1}{m})^j \leq \left(\frac{enl}{jm}\right)^j.$$

The last step is obtained from the well-known upper bound of the binomial coefficient:

$$\binom{n}{k} \leq \left(\frac{en}{k}\right)^k.$$

By substituting the optimal number of hash functions ($l = m/n \ln 2$), we obtain that

$$P(c(i) = j) \leq \left(\frac{e \ln 2}{j}\right)^j.$$

If there are 4 bits per counter, then $j = 16$ is a critical value for the overflow. We can now continue our estimations as follows:

$$P(c(i) \geq 16) \leq \sum_{j=16}^{\infty} \left(\frac{e \ln 2}{j}\right)^j \leq \sum_{j=16}^{\infty} \left(\frac{e \ln 2}{16}\right)^j \leq \sum_{j=16}^{\infty} \left(\frac{1}{8}\right)^j$$
$$= \frac{1}{1 - \frac{1}{8}} \cdot \left(\frac{1}{8}\right)^{16} = \frac{8}{7} \times 2^{-48}.$$

Now the probability that any counter is incremented at least j times can be estimated as:

$$P(max(c) \geq j) \leq \frac{8}{7} \times 2^{-48} \times m$$

which is extremely small for practical values of m.

2.3 Cryptographic Protocols

The following protocols are essential in this thesis and therefore briefly explained.

RSA public key cryptosystem is proposed by Rivest, Shamir, Adleman in 1977 and consists of four steps [18]:

RSA Encryption Algorithm

```
1: Key Generation : Generate two different large odd prime numbers $p$ and $q$ and let $N = pq$. Find positive integer $e$ such that $gcd(e, (p-1)(q-1)) = 1$ then compute $d$ to be inverse of $e$ modulo $(p-1)(q-1)$.
2: Key Distribution : Public key is the pair $(N, e)$ and private key is $d$.
3: Encryption : To encrypt a message $m$ and obtain a ciphertext $c$, one should use public keys and get $c \equiv m^e \pmod{N}$.
4: Decryption: To decrypt the ciphertext $c$, private key should be utilized: $c^d \equiv (m^e)^d \equiv m \pmod{N}$.
```

Goldwasser-Micali Homomorphic Encryption is the first probabilistic encryption protocol and has been proposed in 1982 by Goldwasser and Micali. This encryption scheme is inspired by RSA public key cryptosystem and Rabin's scheme [19] and it is detailed below [20]:

Goldwasser-Micali Homomorphic Encryption Scheme

```
1: Key Generator Algorithm : Generate two different large prime numbers $p$ and $q$.
2: Public Key and Private Key : Let $N = pq$ and find $y \in \mathrm{QNR}_N$ in such a way that $Jacobi(y, N) = 1$. Now $(N, y)$ is the public key and $(p, q)$ is the private key.
3: To Encrypt a bit $x$ : Generate a random $r \in \mathbb{Z}_N^*$, then $c = Enc(x)$ is as follows:
       $$Enc(x) = \begin{cases} r^2 \pmod{N} & \text{if } x \text{ is zero} \\ yr^2 \pmod{N} & \text{if } x \text{ is one} \end{cases}$$
4: To Decrypt a ciphertext $c$:
       $$Dec(c) = \begin{cases} 0 & \text{if } c \in \mathrm{QR}_N \\ 1 & \text{if } c \in \mathrm{QNR}_N \end{cases}$$
```

Note that step 4 can be done efficiently if p and q are known. If that is the case then $Dec(c) = \left(\frac{c}{p}\right)$ and the Legendre symbol can be efficiently computed. The Goldwasser-Micali scheme is *homomorphic* because :

$$Enc(x_1) \cdot Enc(x_2) = (y^{x_1} r_1^2) \cdot (y^{x_2} r_2^2) =$$
$$y^{(x_1+x_2)} \cdot (r_1 r_2)^2 = Enc(x_1 + x_2) \pmod{N}.$$

Advanced Encryption Standard (AES), originally known as Rijndael, is a symmetric key encryption scheme based on the substitution-permutation network. AES is developed by Joan Daemen and Vincent Rijmen, and in 2001 established as a standard by the National Institute of Standards and Technology. The key length in AES is 128, 192 or 256 and the plaintext and ciphertext size is 128. AES consists of four steps; 1)key expansions, 2)initial round, 3)rounds and 4)final round. A reader who is interested in more details about AES can consult [21].

In 1982 David Chaum proposed a type of digital signature that is called *blind signature*. Blind signature is usually used in protocols, in which there is a third party (other than sender and receiver) to authorize (sign) a message without having any knowledge about the content of the message [22]. Chaum's blind signature protocol is as follows:

Chaum 's Blind Signature Protocol

```
1: The signer picks a signing function $s'$ with the inverse of $s$.
2: The user, who wants to obtain the signature of his message $x$, knows a
   commuting function $c$ and it's inverse $c'$ in a way that $c'(s'(c(x))) = s'(x)$.
3: The signer receives $c(x)$ from the user. He then signs $c(x)$ and sends
   $s'(c(x))$ to the user.
4: The user applies $c'$ to $s'(c(x))$ and will obtain the signed message $s'(x)$
   because $c'(s'(c(x))) = s'(x)$.
```

The signing function s' is just known for the signer but s is a publicly known function that doesn't reveal any information of s'. In step 2, the user picks the commuting function c in a way that $c(x)$ and s' doesn't reveal any information about x. Anyone can apply s to the signed message $s'(x)$ to see whether it gives the message x and confirms that $s'(x)$ is indeed formed by the signer.

In Chaum's protocol, one can substitute any public key cryptosystem for the functions s and s'. The blind signature protocol based on RSA is proposed by Bellare et al. [23]. It involves two parties: a signer and a user. The signer generates keys d, e, N for RSA encryption scheme and sends the public key (N, e) to the user, who holds a message x. The protocol is explained below:

Blind Signature Protocol based on RSA

```
1: User chooses a random number $r \in \mathbb{Z}^*{}_N$ and blinds her message $x' = r^e \cdot x \pmod{N}$. She then sends $x'$ to the signer.
2: Signer signs $x'$ and sends $x'' = (x')^d \pmod{N}$ to the user.
3: User retrieves signed message $x^d$ by
       $(x'') \cdot r^{-1} \equiv (x')^d \cdot r^{-1} \equiv (r^e \cdot x)^d \cdot r^{-1} \equiv r^{(ed-1)} x^d \equiv x^d \pmod{N}.$
```

1 out of 2 Oblivious transfer $(1-2$ OT$)$ is a game between a sender who holds two messages m_0 and m_1, and a receiver who holds a bit b . At the end of the game, receiver will just get m_b while the sender remains unaware of b [24]. With utilizing RSA assumption, $1-2$ OT is as follows:

$1-2$ OT Protocol based on RSA

```
1: Sender (S) generates RSA public and private keys; $(N, e)$ and $d$. She also generates two random values $x_0$ and $x_1$.
2: S sends $x_0$, $x_1$ and $(N, e)$ to the receiver (R).
3: R chooses $x_b$ according to his bit $b$ and generates a random value $k$ to blind $x_b$ as $v = (x_b + k^e) \mod N$. He sends $v$ to S.
4: S computes $k_0 = (v - x_0)^d \mod N$ and $k_1 = (v - x_1)^d \mod N$.
5: S calculates $m'_0 = m_0 + k_0$ and $m'_1 = m_1 + k_1$ and sends them to R.
6: R can calculate exactly one of the messages by $m_b = m'_b - k$.
```

Pseudorandom Functions (PRF) are efficiently-computable deterministic functions that always output a pseudo-random value based on a given key k and an argument x. We denote this as $f_k(x)$. We call a PRF as an *Oblivious Pseudorandom Function* (OPRF), if it is computed by a secure protocol between two parties: Alice who holds a value x and Bob who has a key k. They aim to evaluate a pseudorandom function $f_k(x)$ jointly, without revealing their inputs to each other. Freedman et al. construct an OPRF protocol based on $1-2$ OT [25]:

OPRF Protocol based on $1-2$ OT

```
1: Assume that $g$ is a generator of a group $G$ and $\mathrm{ord}(G) = p$.
2: Alice holds an $n$-bit $x = x_1x_2...x_n$ and Bob has a vector $\bar{k} = (k_1, ..., k_n)$.
3: Bob chooses a random vector $\bar{a} = (a_1, ..., a_n) \in (\mathbb{Z}_p^*)^n$.
4: Alice and Bob interact in an 1 out of 2 OT for each $x_i$, in such a way that
   Alice retrieves only one of the items $v_{ix_i}$ where $v_{i0} = a_i$ and $v_{i1} = a_i \cdot k_i$.
5: Bob sends $\hat{g} = g^{1/\prod_{i=1}^{n} a_i}$ to Alice.
6: Alice computes $f_{\bar{k}}(x) = \hat{g}^{\prod_{i=1}^{n} v_{ix_i}}$.
```

A *secure multiparty computation* is a protocol in cryptography with the goal of computing a public function over participants' private inputs. More specifically, the protocol consists of n parties $p_1, p_2, \cdots, p_n$ where each have private data, respectively $d_1, d_2, \cdots, d_n$. At the end of the protocol, participants get the value of a function $F(d_1, d_2, \cdots, d_n)$ without revealing their own inputs to other participants [26].

Yao's Garbled Circuits is the first two-party protocol for secure multiparty computation [27]. It takes place between a circuit creator and an evaluator. The creator picks two secret keys (garbled values) k_0 and k_1 for every wire to encrypt its value. For any gate the creator encrypts it in such a way that given one key for each input wire, it is possible to calculate the key of the corresponding gate output. Also the encryption makes it impossible to obtain any further information. The creator also computes a permuted garbled table of all possible encrypted inputs and with a $1-2$ OT protocol, sends evaluator's garbled inputs to the evaluator. Finally the evaluator decrypts all the gates' garbled outputs repeatedly and gains the circuit's garbled output.

JustGarble is a compact garbling system (i.e. the whole implementation is under 800 K byte) for boolean circuits designed by Bellare et al. in 2013 with the goal of optimized garbling [28]. In a boolean circuit the wire values are 0 and 1. JustGarble system utilizes a fixed-key block cipher, in particular AES, to optimize the *Garble* and *Evaluate* modules of the system. The main advantage of using AES comes from the fact that recent processors have exclusive instruction set, AES-NI [29], for performing key expansion and doing the rounds of AES in hardware which is much faster than software implementation.

3 Private Membership Test with Bloom Filters

Probabilistic techniques such as Bloom filters, are heavily used by many service providers in numerous distributed systems. The space-efficiency of Bloom filters make them convenient instruments to store databases by many servers to reduce networking costs [16]. Utilizing hash functions to construct this filter makes it secure. In other words, one-wayness of hash functions makes the filter itself an array of zeros and ones that cannot disclose the actual database. On the other hand, Bloom filters are suitable data structures to perform a membership test, especially in the cases that the problems caused by false positive are small. This motivates us to find possible solutions to create a privacy preserving variant of Bloom filter to fulfill private membership tests.

We motivate the problem of private membership test in the context of a realistic scenario; a server hosted by a security company stores its malware signature database in a Bloom filter and a client of this company wants to know if a file in his possession is malicious or not. If the test results in positive, we encourage the client to reveal his file to the security company to get further help. In this scenario, if a query from Bloom filter returns false positive, the error happened on the safe side. All malicious files will be disclosed to the security company together with a few randomly chosen clean files.

We recall from chapter 1 that if the client simply reveals the actual file to the server, it may lead to violation of his privacy. Moreover, it may be already too much to reveal the information that an organization possesses a particular document. A trivial solution to the PMT problem is that the server ships all of its database to the clients. But the need of a high bandwidth makes this solution infeasible. Also, most probably service providers do not want to completely reveal their databases. Therefore implementing a fully functional protocol to address the problem of PMT is an essential task.

In order to design a private membership test, we introduce three protocols which are detailed in this chapter. All protocols involve two parties: a server S and a client C. We assume that the server possesses a set of n elements $X = \{x_1, x_2, \cdot\cdot\cdot, x_n\}$ as its database and the client wants to search for x through X. After executing these protocols, S doesn't learn anything from client's x and C only learns whether x is a member of server's database. We design the protocols in a way that it would be possible for the server to prove that he cannot retrieve any information about x unless C directly releases it to him.

3.1 Protocol 1, utilizing Goldwasser-Micali homomorphic encryption

The idea behind our first solution is to generate an encrypted Bloom filter which can be transferred to the client, without jeopardizing the secrecy of server's database. The algorithm to encrypt the Bloom filter is utilizing Goldwasser-Micali homomorphic encryption scheme as a building block and is explained bellow:

Encrypting the Bloom Filter in Protocol 1

```
1: $S$ stores $X$ in a Bloom filter B with $l$ hash functions $H_1, H_2, ..., H_l$.
2: $S$ picks $p$ and $q$ as two distinct large prime numbers and computes
   $N = pq$.
3: $S$ chooses another hash function $H : \{0,1\}^* \to \mathbb{Z}_N$.
4: $S$ finds some $y \in QNR \pmod{N}$ in such a way that Jacobi$(y, N) = 1$.
5: The public keys are $N$ and $y$, the private ones are $p$ and $q$.
6: For every index $i \in$ B, with trial-and-error method, $S$ finds the smallest
   integer $j$ such that Jacobi$(H(j||i), N) = 1$.
7: $S$ encrypts each entry $i$ of B (denote as $B(i)$) and gets $EB(i)$ in such a
   way that:
```

$$EB(i) = \begin{cases} B(i) & \text{if } H(j||i) \text{ is in } QR_N \\ 1 - B(i) & \text{if } H(j||i) \text{ is in } QNR_N \end{cases}$$

Now the encrypted Bloom filter (EB) itself would not reveal any information about the actual database. In other words, a query from this encrypted Bloom filter will get some array positions but wouldn't reveal if the value of bits on each of these positions is zero or one. Server ships EB, hash functions H_i, $i = 1, 2, \cdots, l$, hash function H and public keys (N, y) to the client. It is now possible for C to calculate Jacobi$(H(j||i), N)$, but he doesn't know whether $H(j||i)$ is a quadratic residue modulo N. This is due to the fact that N in Goldwasser-Micali encryption scheme is chosen to be impossible to factor in polynomial time.

Client C who wants to do a membership test for an element x, queries EB and finds the corresponding indices $EB(H_i(x))$ where $i \in \{1, \cdots, l\}$. He then encrypts those indexes utilizing Goldwasser-Micali homomorphic encryption schemes. The algorithm to query EB is as follows:

Querying the Encrypted Bloom Filter in Protocol 1

```
1: $C$ calculates $H_i(x)$ $(i = 1, \cdots, l)$ which are indices in the EB that he
   wants to decrypt.
2: For each i, $C$ finds with trial-and-error method the smallest integer $j$
   such that
       $\text{Jacobi}(H(j||H_i(x)), N) = 1.$
3: $C$ chooses $r^2$, a random square modulo $N$, and computes
       $H(j||H_i(x))r^2.$
4: $C$ multiplies the result of step 3 by $y$ with the chance of 50% and obtains
   $z$ such that
       $z = H(j||H_i(x))r^2 y^{mask}$
   where $mask$ is 0 or 1, each with the probability of 1/2. He then sends
   the result $z$ to $S$.
5: $S$ computes Jacobi$(z, p)$ and Jacobi$(z, q)$ and tells $C$ if $z$ is in $QR_N$ or in
   $QNR_N$.
6: $C$ decrypts $EB(H_i(x))$ as follows:
       $\begin{cases} B(H_i(x)) = EB(H_i(x)) & \text{if } z \text{ is in } QR_N \text{ and } mask = 0 \\ B(H_i(x)) = 1 - EB(H_i(x)) & \text{if } z \text{ is in } QNR_N \text{ and } mask = 0 \\ B(H_i(x)) = 1 - EB(H_i(x)) & \text{if } z \text{ is in } QR_N \text{ and } mask = 1 \\ B(H_i(x)) = EB(H_i(x)) & \text{if } z \text{ is in } QNR_N \text{ and } mask = 1 \end{cases}$
7: If the result of step 6 is $B(H_i(x)) = 1$ for every hash function $H_i$, then the
   output of this algorithm is positive and this means $x$ probably belongs
   to the database $X$. Otherwise, as soon as the first zero appeared in step
   6, algorithm can be stopped. In this case $x$ definitely does not belong to
   the database $X$.
```

If a client wants to decrypt all the $B(H_i(x))$, he should repeat the above algorithm l times.

C blinds $H_i(x)$ for every i in step 3. Because of random square r^2, the server wouldn't learn client's actual indices and therefore S cannot guess the value of client's element x. We highlight that Jacobi$(H(j||H_i(x))r^2, N) = 1$ because the Jacobi symbol of any random square is one. Moreover, in step 4 client masks the quadratic residues characteristic of his l positions of the filter. The result z that is computed in step 3 and 4 guarantees the secrecy for the client. At step 6, for each i, client does the following reasoning to decrypt $EB(H_i(x))$:

- If z is a quadratic residue modulo N and $mask = 0$ (C did not multiply z by y), then $z = H(j||H_i(x))r^2$. We conclude that $H(j||H_i(x)) \in QR_N$ because:

$$\begin{aligned} z \in QR_N &\Rightarrow \left(\frac{z}{p}\right) = \left(\frac{z}{q}\right) = 1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot \left(\frac{r^2}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot \left(\frac{r^2}{q}\right) = 1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot (1) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot (1) = 1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) = 1 \\ &\Rightarrow \ H(j||H_i(x)) \text{ is a quadratic residue modulo } N. \end{aligned}$$

  and $B(H_i(x)) = EB(H_i(x))$.

- If z is a quadratic non-residue modulo N and C did not multiply z by y, then $z = H(j||H_i(x))r^2$. We conclude that $H(j||H_i(x)) \in QNR_N$ because:

$$\begin{aligned} z \in QNR_N &\Rightarrow \left(\frac{z}{p}\right) = \left(\frac{z}{q}\right) = -1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot \left(\frac{r^2}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot \left(\frac{r^2}{q}\right) = -1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot (1) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot (1) = -1 \\ &\Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) = -1 \\ &\Rightarrow \ H(j||H_i(x)) \text{ is a quadratic non-residue modulo } N. \end{aligned}$$

  Therefore $B(H_i(x)) = 1 - EB(H_i(x))$.

- If z is a quadratic residue modulo N and $mask = 1$ (C multiplied z by y) then $z = H(j||H_i(x)) \cdot r^2 \cdot y$. We conclude that $H(j||H_i(x)) \in QNR_N$ because:

  $$\begin{aligned}
  & z \in QR_N \Rightarrow \left(\frac{z}{p}\right) = \left(\frac{z}{q}\right) = 1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot \left(\frac{r^2}{p}\right)\left(\frac{y}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot \left(\frac{r^2}{q}\right)\left(\frac{y}{q}\right) = 1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot (1)(-1) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot (1)(-1) = 1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) = -1 \\
  & \Rightarrow \ H(j||H_i(x)) \text{ is a quadratic non residue modulo } N.
  \end{aligned}$$

  and $B(H_i(x)) = 1 - EB(H_i(x))$.

- If z is a quadratic non-residue modulo N and C multiplied z by y then $z = H(j||H_i(x)) \cdot r^2 \cdot y$. We conclude that $H(j||H_i(x)) \in QR_N$ because:

  $$\begin{aligned}
  & z \in QNR_N \text{ and Jacobi}(z, N) = 1 \Rightarrow \left(\frac{z}{p}\right) = \left(\frac{z}{q}\right) = -1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot \left(\frac{r^2}{p}\right)\left(\frac{y}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot \left(\frac{r^2}{q}\right)\left(\frac{y}{q}\right) = -1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) \cdot (1)(-1) = \left(\frac{H(j||H_i(x))}{q}\right) \cdot (1)(-1) = -1 \\
  & \Rightarrow \left(\frac{H(j||H_i(x))}{p}\right) = \left(\frac{H(j||H_i(x))}{q}\right) = 1 \\
  & \Rightarrow \ H(j||H_i(x)) \text{ is a quadratic residue modulo } N.
  \end{aligned}$$

  and $B(H_i(x)) = EB(H_i(x))$.

If it is needed, S is prepared to show that N is chosen as in Goldwasser-Micali and indeed y is a quadratic non-residue modulo N. Therefore S can prove that he does not have the capabilities to guess x by obtaining z. This first protocol is the novelty of our work and can be also find in [10]. Protocol 1 is summarized in figure 2 [30].

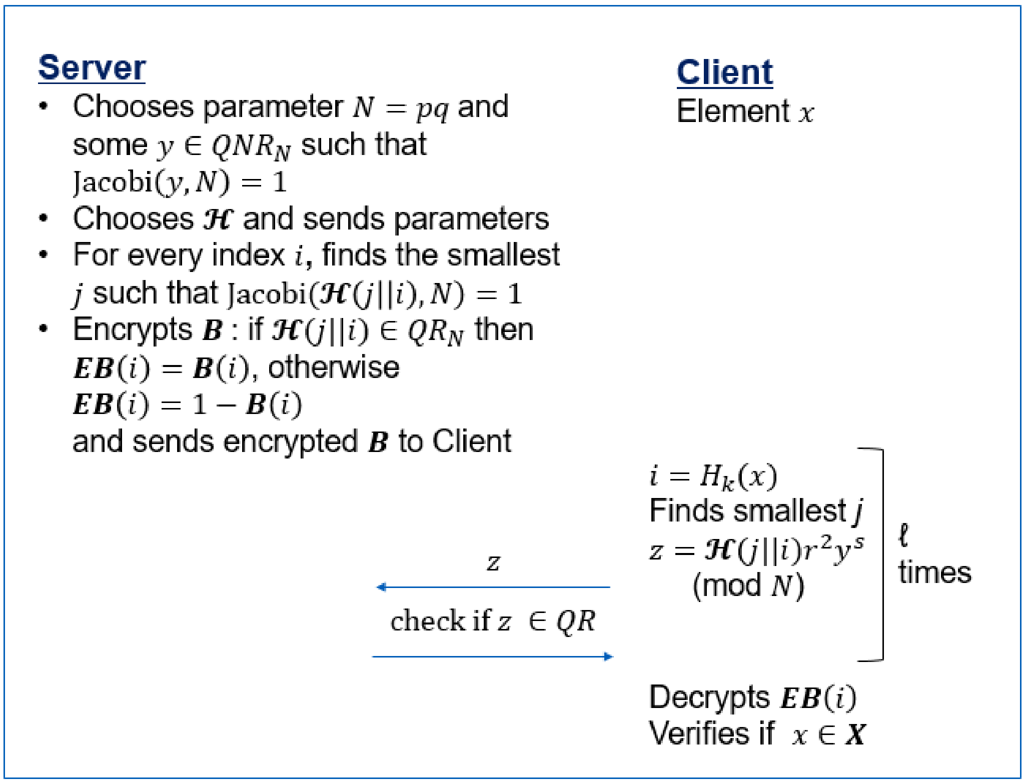

Server
Chooses parameter $N = pq$ and some $y \in QNR_N$ such that $\text{Jacobi}(y, N) = 1$
Chooses $\mathcal{H}$ and sends parameters
For every index i, finds the smallest j such that $\text{Jacobi}(\mathcal{H}(j||i), N) = 1$
Encrypts $\boldsymbol{B}$: if $\mathcal{H}(j||i) \in QR_N$ then $\boldsymbol{EB}(i) = \boldsymbol{B}(i)$, otherwise $\boldsymbol{EB}(i) = 1 - \boldsymbol{B}(i)$ and sends encrypted $\boldsymbol{B}$ to Client
Client
Element x
$i = H_k(x)$
Finds smallest j
$z = \mathcal{H}(j||i)r^2y^s \pmod N$
ℓ times
z
check if $z \in QR$
Decrypts $\boldsymbol{EB}(i)$
Verifies if $x \in \boldsymbol{X}$

Figure 2: Protocol 1

3.2 Protocol 2, utilizing blind RSA signature

The second suggested solution is adopted from the paper by Nojima et al. [11]. Although the construction of their protocol is based on any blind signature scheme, we use blind RSA signature as a building block to implement this protocol [10].

On the first solution, in order to preserve the secrecy of the database we encrypted the Bloom filter. On the second solution instead of encrypting the Bloom filter, we generate a Bloom filter for signed record utilizing the following algorithm:

Generating the Bloom Filter for Signed Records

```
1: $S$ picks $e, d, N$ based on RSA signature protocol. He also picks a hash function $H$.
2: $S$ defines $Sig(x) = H(x)^d$ to be the signature for a record $x$. He then computes one RSA signature for every $x \in X$.
3: $S$ chooses $l$ hash functions $H_1, H_2, ..., H_l$ for the Bloom filter.
4: For all $x \in X$ the server inserts $x||Sig(x)$ to a Bloom filter B.
5: $S$ sends $C$ the Bloom filter B and its hash functions, his public RSA key $(e, N)$ and the hash function $H$.
```

We emphasize that the Bloom filter B is representing the combination of records in the database, with their RSA signature. It means that a bit $B(h) = 1$ iff $h = H_i(x||Sig(x))$ for some $i \in \{1, \cdots, l\}$ and for some element $x \in X$. So this filter does not represent the database X directly, and therefore the client C can not obtain X from B. Also S is ready to prove that he chose N as in RSA encryption scheme.

The client C can follow the next algorithm to query the Bloom filter for his record x.

Querying the Bloom Filter

```
1: $C$ chooses a random $r \in \mathbb{Z}_N$ and computes $y = H(x)r^e$ modulo $N$. He then sends $y$ to $S$.
2: $S$ calculates the signature of $y$ and obtains $z = y^d \pmod N$. He then sends $z$ to $C$.
3: $C$ is now able to compute the signature of $x$ because
       $$\begin{aligned} z = y^d = H(x)^d r^{ed} = H(x)^d r^1 &\pmod N \\ \text{and } Sig(x) = z/r &\pmod N \end{aligned}$$
4: $C$ queries the Bloom filter for $x||Sig(x)$.
```

Client blinds his record x in step 1 by utilizing a random number r. This guarantees to C that S can not retrieve x from y. In step 4, if for every hash function H_i the client learns that $B(H_i(x||Sig(x))) = 1$ then probably x is in the database X.

Unlike in protocol 1, in the second protocol the client only sends one query to the server. This solution is summarized in figure 3 [30].

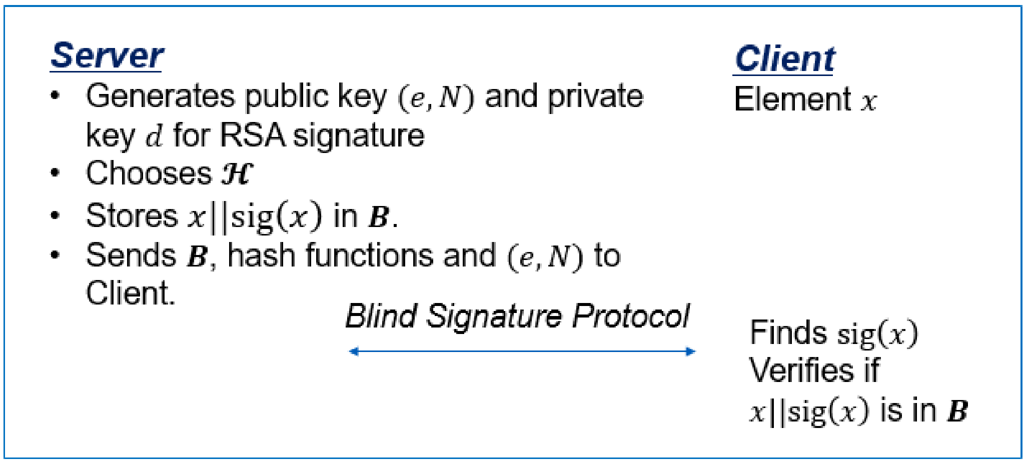

Server
Generates public key (e, N) and private key d for RSA signature
Chooses $\mathcal{H}$
Stores $x||\mathrm{sig}(x)$ in $\boldsymbol{B}$.
Sends $\boldsymbol{B}$, hash functions and (e, N) to Client.
Blind Signature Protocol
Client
Element x
Finds $\mathrm{sig}(x)$
Verifies if $x||\mathrm{sig}(x)$ is in $\boldsymbol{B}$

Figure 3: Protocol 2

3.3 Protocol 3, utilizing oblivious pseudorandom function

Our last protocol is partly based on a protocol that is proposed by Nojima et al. [11]. Our protocol, unlike the protocol in [11], has a function K to encrypt the Bloom filter and this consequently requires additional steps in both server and client sides [10]. Similar to the previous two solutions, the first step in the third protocol is to generate the Bloom filter.

Generating the Bloom Filter

```
1: $S$ chooses $l$ hash functions $H_1, H_2, ..., H_l$ for the Bloom filter.
2: $S$ picks an OPRF $K$ with a secret key $k'$, to generate one bit of output
   that can be utilized as one-time pad key.
3: $S$ selects another OPRF $F$ with a secret key $k$, such that
       $F_k(H(x)) = (H_1(x), H_2(x), \cdots, H_l(x))$.
4: $S$ inserts the database $X$ in the Bloom filter B.
5: $S$ encrypts B and generates EB in such a way that
       $EB(i) = B(i) \oplus K_{k'}(i)$ for every $i$.
6: $S$ sends the hash function $H$ and the encrypted Bloom filter EB to the
   client $C$.
```

The oblivious pseudorandom functions that is utilized in step 2 of this algorithm can be evaluated using multi-party computation between S and C. Its output is a one time pad key to encrypt the Bloom filter. Another multi-party computation between the server and the client can be utilized to evaluate $F_k(x)$, the server and client's secret inputs are respectively k and x, and the output is $F_k(H(x)) = (H_1(x), H_2(x), \cdots, H_l(x))$. The client C is unaware of OPRF K and its output, therefore he can not retrieve X from encrypted Bloom filter.

We highlight that unlike in the previous two protocols, in this solution S does not send hash functions of the Bloom filter to C. Let $h_i = H_i(x)$ for $i \in \{1, \cdots, l\}$. In order to find the right Bloom filter entries, S and C together evaluate $F_k(H(x))$. After this step C only learns $(h_1, h_2, \cdots, h_l)$ and the value of $H(x)$ will remain secret to S.

By following the next algorithm, the client C can decrypt the entries of EB which correspond to his record x.

Decrypting the Bloom Filter Entries

```
1: $S$ and $C$ together calculate $K_{k'}(h_i)$ for every $h_i$, $i \in \{1, \cdots, l\}$ in such a
   way that only the client $C$ obtains the value of bits $b_i = K_{k'}(h_i)$.
2: $C$ can decrypt $l$ bits of encrypted Bloom filter by computing $EB(h_i) \oplus b_i$
   for all $i \in \{1, \cdots, l\}$.
```

After the first step, the server will not learn the indexes h_i and C will not discover the secret keys k and k'. This fact proves that the third protocol is also preserving secrecy for both parties. In the second step, if for every i, $EB(h_i) \oplus b_i = 1$ then the client C learns that his record x probably belongs to the database X.

To simulate the implementation of the third protocol, we substitute oblivious pseudorandom function by garbled circuits. We utilize JustGarble algorithm to evaluate one block of AES [28]. This function of AES with a fixed key of length 128 bits, is used as a replacement of OPRF. The server is ready to prove to a trusted third party that he has garbled a correct function.

In order to evaluate $F_k(H(x))$, the server needs to evaluate AES128 different number of times. For instance, if $l \leq 5$ and the number of bits in the Bloom filter is $m \leq 2^{25}$, then we can interpret the output of 128 bits as five hash functions. In this case each hash function has an output of 25 bits. Consequently, if $l \leq 10$ then two evaluations of AES128 are sufficient. To compute one bit of OPRF K, only one evaluation of AES128 is required. Protocol 3, based on our implementation is pictured in figure 4 [30].

Figures 2, 3 and 4 are obtained from the poster which was presented at the CloSe project workshop [2].

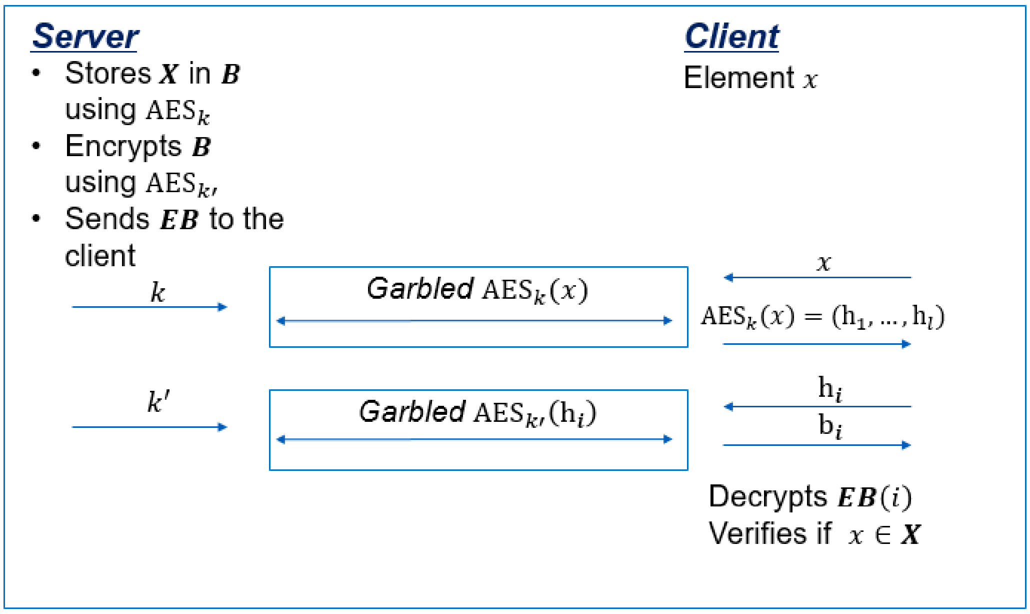

Server
Stores X in B using AES_k
Encrypts B using $\mathrm{AES}_{k'}$
Sends EB to the client
Client
Element x
k
Garbled $\mathrm{AES}_k(x)$
x
$\mathrm{AES}_k(x) = (\mathrm{h}_1, \ldots, \mathrm{h}_l)$
k'
Garbled $\mathrm{AES}_{k'}(\mathrm{h}_i)$
h_i
b_i
Decrypts $EB(i)$
Verifies if $x \in X$

Figure 4: Protocol 3

[2]The CloSe workshop was held on 15th and 16th of April 2015 at Otaniementie 17, Espoo. Website: https://wiki.aalto.fi/display/CloSeProject/CloSe+Project+Workshop

4 Implementation

This chapter presents a detailed insight into the implementation of the three suggested protocols from the previous chapter. We utilize Python programming language version 2.6 to implement our work. We base our software developments on certain existing Python packages. To install these packages in Windows and Linux platforms, use the following command:
python install setup.py <package address>.

To present the software development of this work, we utilize various pseudocode. Thus it is easier for readers to follow the idea behind our implementations. First we explain the Python packages that are utilized in our implementations. We show some of their limitations and then we explain how to improve them. After that, the required Python codes to create Python libraries for chapter 3 protocols are explained in details. Finally the performance of our implementation is measured based on a real case scenario.

4.1 Software Development

Bloom filter is the fundamental part for all of the three protocols and therefore we start our software developing with the task of creating a Bloom filter. We recall that each Bloom filter has three parameters m, n and l which are respectively number of bits in the filter, size of the set and number of filter's hash functions. The first step to create a Bloom filter is to build a bit array of m-bits and set all bits to zero. Second, we should define l hash functions in such a way that these hash functions map each given item to a random number, chosen uniformly over the range of m. In order to develop such a hash function, one can use this paper [31] as one ideal solution. To insert n elements of the set to the Bloom filter B one can utilize the next pseudocode.

Inserting the Set of n Elements to the Bloom Filter B

```
for $i : 1, \cdots, n$ do
  for $j : 1, \cdots, l$ do
    $b \leftarrow h_j(x_i)$
    if $B_b == 0$ then
      $B_b \leftarrow 1$
    end if
  end for
end for
```

A query for x from the Bloom filter B, can be done utilizing the following pseudocode.

Membership Test for an Element x in the Bloom Filter B

```
$result \leftarrow True$
for $j : 1, \cdots, l$ do
  $b \leftarrow h_j(x)$
  if $B_b == 0$ then
    $result \leftarrow False$
  end if
end for
return $result$
```

As mentioned before, Bloom filters are popular data structures and therefore many libraries are written to construct them, with different programming languages. To implement the initial Bloom filter we use an open source package of Python, called *pybloom* and authored by Jay Baird and Bob Ippolito [32]. The first step to create a Bloom filter f with this package is to determine the maximum number of elements that can be inserted to this filter, denoted as the *capacity*, before exceeding the FP probability that is defined for f and denoted as the *error rate*. The following syntaxes create an empty Bloom filter f with the capacity of α and the error rate of β:

```
from pybloom import BloomFilter

f = BloomFilter(α, β)
```

The following snippet of code inserts the set X to the Bloom filter f, which has been created before:

```
for element in X:
    f.add(element)
```

In order to perform a membership test for x in the Bloom filter f, one can use the following syntaxes. If the size of X exceeds the capacity that is defined for f, then the program throws an error.

```
print x in f
```

If the above program prints $False$, it means x does not belong to the X and if the program prints $True$ it means x probably belongs to the X.

One of the limitations with a normal Bloom filter, and also with this package, is deletion. In other words, after generating a Bloom filter it is possible to add more elements to the filter until reaching the capacity, but deleting an existing record from this filter is not possible. To solve this issue we also utilize another variant of a Bloom filter: counting Bloom filter. We recall from the second chapter of this thesis that the counting Bloom filter has the ability of deletion. Before going any further, let us explain how to construct a counting Bloom filter. The pseudocode for counting Bloom filter insertion is as follows.

Inserting the Set of n Elements to the counting Bloom Filter C

```
  for $i : 1, \cdots, n$ do
    for $j : 1, \cdots, l$ do
      $b \leftarrow h_j(x_i)$
      $C_b \leftarrow C_b + 1$
    end for
  end for
```

Consequently, to delete an element from a counting filter one should decrease the corresponding counters.

Deleting an Element x from the Counting Bloom Filter C

```
  for $j : 1, \cdots, l$ do
    $b \leftarrow h_j(x)$
    $C_b \leftarrow C_b - 1$
  end for
```

We modify the pybloom library to create a new class 'BloomFilterEx', which stores database records in a counting Bloom filter as well as a normal Bloom filter. In order to do this improvement, another Python open source library is used which is called *numpy* [33]. This package contains an N-dimensional array object that can be utilized to design a counting Bloom filter. The modification of pybloom package can be done with the following snippet of code [3].

[3]To install this package on Windows platform, one might need to install Visual Studio program as well.

```
import numpy as np
from pybloom import BloomFilter
class BloomFilterEx(BloomFilter):
  def__init__(self,capacity,error_rate=0.001,
                 prime_sz=1024):
    super(BloomFilterEx,self).__init__(capacity,
                 error_rate)
    self.__data = np.zeros(self.num_bits,dtype=np.int)
    self.__prime_sz = prime_sz
    self.p = getPrime(prime_sz, randfunc=None)
    self.q = getPrime(prime_sz, randfunc=None)
```

To remove an element from the Bloom filter, one should first delete it from the counting Bloom filter. If any of the counters corresponding to that element went to zero, then the value of corresponding bits in B should be changed to zero. On the other hand if after deletion, the counters are more than one, then the Bloom filter does not change. The function to remove a key from the Bloom filter can be defined with Python programming language as:

```
def remove(self, key):
  for k in hashes:
    j = h_k(key)
    data[j] -= 1
    if data[j] == 0:
      bitarray[j] = not (bitarray[j])
```

To construct the cryptographic primitives that are needed in the implementation of the first protocol, another package of Python called *pycrypto* [34] is used. The *pkcs1* Python library, which is authored by Benjamin Dauvergne, is used to compute the Jacobi symbols [35].

The implementation of encrypted Bloom filter can be done with the following function. The hash function Sha-1 is chosen for the hash function H. Moreover, p and q are two distinct prime numbers and $N = pq$. We later explain why the letter 'x' is needed to implement the encryption function.

Pseudocode to Define an Encryption Function to create EB

```
DEFINE : encrypt()
for $i : 1, \cdots, \text{len (bitarray)}$ do
    $j \leftarrow 0$
    newindex ← ('x' + string(i))
    indexhash ← Sha1(string(j)+newindex) (mod $N$)
    while jacobi(indexhash, $n$) $\neq 1$ do
        $j \leftarrow j + 1$
        indexhash ← Sha1(string(j)+newindex) (mod $N$)
    end while
    if jacobi(indexhash, $p$) $= -1$ then
        $(\text{ebitarray})_i \leftarrow \text{NOT}(\text{bitarray})_i$
    else
        $(\text{ebitarray})_i \leftarrow (\text{bitarray})_i$
    end if
end for
return The encrypted bitarray
```

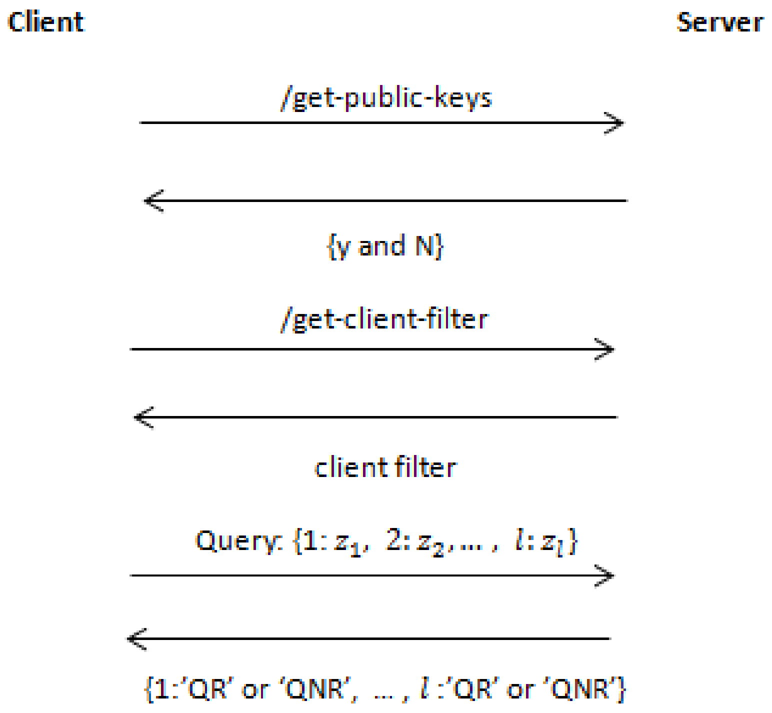

Client
Server
/get-public-keys
{y and N}
/get-client-filter
client filter
Query: {1: z_1, 2: z_2, ... , l: z_l}
{1:'QR' or 'QNR', ... , l :'QR' or 'QNR'}

Figure 5: Request-Response between Two Parties in Protocol 1

During the development of an encrypted function for B, we noticed that concatenating j and i may lead to reveal more than one entry of the Bloom filter to the client. For instance, combination of $j = 10$ and $i = 111$, is the same as concatenation of $j = 101$ and $i = 11$. To address this problem we combine the English letter 'x' with i and compute $H(j||x||i)$ instead of $H(j||i)$. For the sake of clarity, we note that this letter 'x' has nothing to do with the client's item x.

The next important part of this implementation is to define a proper function for the server, to create the response for the parameter z that is received from the client. If this result z is valid i.e. Jacobi$(z, N) = 1$, then the server determines whether z is a QR or a QNR modulo N.

Pseudocode to Define a Response Function for S

```
DEFINE : response($z$)
if jacobi($z, n$) $\neq 1$ then
  return $z$ is not valid!
end if
answer $\leftarrow$ " "
if jacobi($z, p$) $= 1$ and jacobi($z, q$) $= 1$ then
  answer $\leftarrow$ "QR"
else
  answer $\leftarrow$ "QNR"
end if
return answer
```

In the first protocol, lots of communication take place between the server and the client (see figure 5), therefore the next step of this software development is to design a web server. To fulfill this request-response pattern, a Python web framework called *Bottle* is used [36].

We define command line tools for the server which enable it to create its Bloom filter with any capacity and error rate and later insert any set to it. To build these command-line interfaces, the Python *Argparse* built-in module can be utilized[4] as shown below:

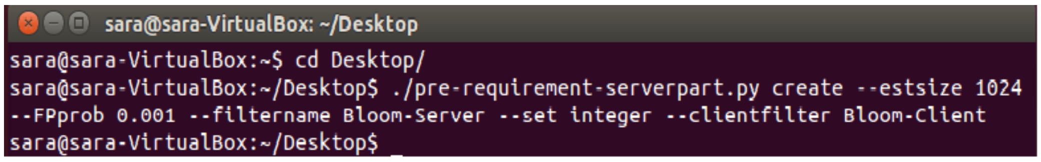
```
import argparse
p = argparse.ArgumentParser()
```

[4]https://docs.python.org/3/library/argparse.html

```
p.add_argument("--estsize",type=int,required=False)
p.add_argument("--set",type=str,required=False)
p.add_argument("--FPprob",type=float,required=False)
p.add_argument("--filtername",type=str,required=False)
```

After creating the normal Bloom filter, its encrypted version and the counting Bloom filter, the server needs to save them for later serializing and de-serializing. The Python built-in module; *Pickle*[5] converts an object (e.g. a Bloom filter) into a stream of bytes. Pickle makes it possible to transmit the object into the memory and recreate it when it is needed. This procedure is called serialization and the reverse operation is de-serialization (e.g. a byte stream is converted back into the Bloom filter). The syntaxes below, first serialize (pickle) a Bloom filter f to a file named server and then de-serialize (unpickle) it to a new file called Bloom-filter-f.

```
import pickle
f = Bloomfilter(capacity,error-rate)
filename = open ('server','w')
pickle.dump(f,filename)
filename.close()
f = open ('server','r')
Bloom-filter-f = pickle.load(f)
f.close()
```



```
sara@sara-VirtualBox: ~/Desktop
sara@sara-VirtualBox:~$ cd Desktop/
sara@sara-VirtualBox:~/Desktop$ ./pre-requirement-serverpart.py create --estsize 1024
--FPprob 0.001 --filtername Bloom-Server --set integer --clientfilter Bloom-Client
sara@sara-VirtualBox:~/Desktop$ _
```

Figure 6: A Snapshot from an Ubuntu Terminal to Create B

The final step of this software development is to construct a Python library that makes it easier for potential users to install and utilize it. Therefore, creating a setup.py[6] file is the last task in implementation of the first protocol. A simple yet efficient example of setup.py file is as follows.

```
import sys
from cx_Freeze import setup, Executable
```

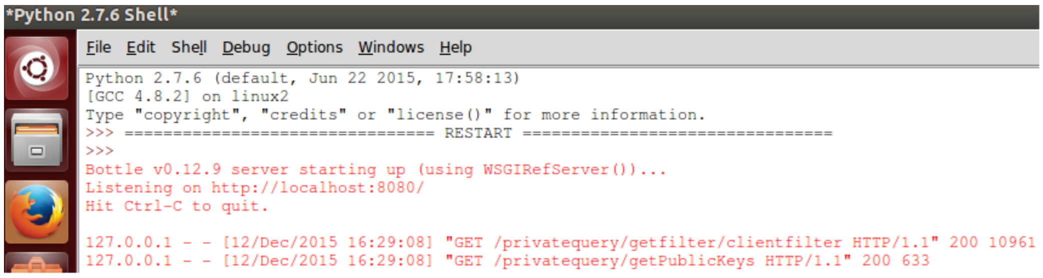
[5]https://docs.python.org/3/library/pickle.html
[6]cx_Freeze package can be retrieved from http://cx-freeze.sourceforge.net/

```
setup(
    name = "package-name",
    version = "1",
    description = "Python-Library-for ...",
    executables = [Executable("package-name.py",
                   base="Win32GUI")])
```

The following commands create a Python virtual environment and install the required packages defined in requirements.txt.

```
$ virtualenv ve
$ . ./ve/bin/activate
$ pip install -r requirements.txt
```

Now we have all the prerequirements to run the first protocol. To test our prototype, we assume that the server S creates a Bloom filter that represents a set of size 2^{10} containing integer numbers $\{1, 2, \cdots, 1024\}$ with false positive error rate of 0.001. S then encrypts the Bloom filter and serializes this object to a file named Bloom-Server (see figure 6). This way the server can modify the Bloom filter whenever it is needed. Moreover S generates an object Bloom-Client that is the encrypted Bloom filter for the client. This object does not contain p and q, server's private parameters, and therefore the client can not decrypt it unless he seeks help from S.



```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
Bottle v0.12.9 server starting up (using WSGIRefServer())...
Listening on http://localhost:8080/
Hit Ctrl-C to quit.

127.0.0.1 - - [12/Dec/2015 16:29:08] "GET /privatequery/getfilter/clientfilter HTTP/1.1" 200 10961
127.0.0.1 - - [12/Dec/2015 16:29:08] "GET /privatequery/getPublicKeys HTTP/1.1" 200 633
```

Figure 7: A Snapshot from Server that Listens to the Client's Requests

A client C who wants to initiate a privacy preserving query, sends a request to server and asks for the encrypted Bloom filter and the public keys y and N.

The server responds to the client and C receives two files in his folder (see figure 8). C sends more requests for his results z and finally he is able to obtain the result of his PMT. The complete Python codes for the first protocol are available in the appendices A, B and C.

Figure 8: A Snapshot from the Client's folder

The implementation of the first protocol was presented in the 'Secure Systems Annual Demo Day' [7].

To implement the second protocol, Python package pybloom is used to create a Bloom filter. From Python library pycrypto, we utilize RSA to compute the blind signature of the items in the database. One can create the public and private keys for the RSA signature, with the following snippet of code. We assume that the size of modulus N is 2048 bits.

```
from Crypto.PublicKey import RSA
from Crypto.Util.number import getRandomRange
loadedPrivate = RSA.generate(2048)
loadedPublic = loadedPrivate.publickey()
```

We utilize the hash function md5 for H. After obtaining the signatures of the records (Sign) in the database X, the server inserts them to its Bloom filter f. This procedure can be done by the following code.

```
for record in X:
    messageHash = md5.new(str(record)).digest()
    Sign = loadedPrivate.sign(messageHash,
                    loadedPrivate.n)[0]
    f.add(Sign)
```

The client in the second protocol should blind his message by utilizing the following snippet of code.

[7]The 'Secure Systems Annual Demo Day was held at Aalto University in Konemiehentie 2, Espoo on first of June 2015. Website: https://wiki.aalto.fi/display/sesy/Secure+Systems+Annual+Demo+Day

```
import random
def generateBlindFactor(RSAobj):
  return (getRandomRange(1, RSAobj.key.n-1,
  randfunc=RSAobj._randfunc))
  # generates r
  r = generateBlindFactor(loadedPrivate)
  blind-message = loadedPublic.blind(messagehash,r)
```

The client sends his blind-message to the server. Then, the server signs this blind-message by

```
blindSigned = loadedPrivate.sign(blindmessage,
              loadedPrivate.n)[0]
```

Readers can retrieve more information about the software developing of the second protocol from appendix D.

In the third protocol, it is easier to understand the complexity of implementation, if we substitute the OPRF with garbled circuits. We simulate this implementation based on the JustGarble algorithm that is described in [28].

4.2 Results

We recall from the previous chapter that motivation to implement the protocols comes from the following scenario: a server S holds a database of malicious samples and a Client C aims to make sure that certain files are not malicious. These queries are considered to be leaking information about the clients who possess some files that are supposed to be private, for example, document files. In this section, we assume that there are 2^{21} malicious samples[8] in S's database and test our solutions to show the results that are obtained from each of the protocols. Currently document exploits are a significant attack vector so we assume that S has a collection of known exploit documents.

We implement the first two protocols, using an x86-64 Intel Core i5-2450 processor clocked at 2.5 GHz with a 3MB L3 cache. To measure the performance of our implementation more accurately, we make experiments in two popular operating systems; Windows 7 and Ubuntu 14.04.3 LTS. We

[8]This number is realistic and obtained from the industry.

run Ubuntu in Oracle VirtualBox[9] which is installed on Windows 7 platform.

Let us start with the first protocol. To protect others from security threats (e.g. to not spread the malware samples accidentally), S stores the SHA-1 values of its malware signatures in a database X. Moreover, we implement a Bloom filter B of length 2^{25} with the false positive rate of 0.001 and use 10 hash functions to insert X in B. One can simulate such a database, utilizing the following syntaxes. Additionally, we pick two prime numbers p and q with the size of 1024 bits to compute the modulus N in Goldwasser-Micali encryption scheme.

```
from Crypto.Hash import SHA as sha1
dummysha1 = open('sha1input','a')
for x in range(1, (2^21)+1):
    dummysha1.write(sha1.new(str(x)).hexdigest()+"\n")
dummysha1.close()
```

To calculate the required time for the server to generate an encrypted Bloom filter and also to observe the accuracy of the implementation, we test it for several databases with different sizes. We use one of the Python built-in modules, *time*, for measuring the execution time of a program. More specifically, the Python function *time.clock()* is utilized to return the execution time is microseconds. Table 1 gives the time executions for some selective database sizes.

Size of X	Size of the Filter	Error Rate	Required Time in Seconds
2^{10}	2^{14}	0.001	13.181999
2^{11}	2^{15}	0.001	25.449000
2^{12}	2^{16}	0.001	58.358999
2^{13}	2^{17}	0.001	100.294999
2^{14}	2^{18}	0.001	193.168999

Table 1: Encrypting B in Protocol 1 on Windows 7

Table 1 can also be used to estimate the time for generating an encrypted Bloom filter for a database with any size. In our scenario, it would take 6.7 hours to generate EB for a database with the size of 2^{21}. It is much more efficient for the server to perform the pickling since the preprocessing is a

[9]https://www.virtualbox.org/

very time consuming task and repeating it to apply the new changes to the filter drastically decreases the efficiency.

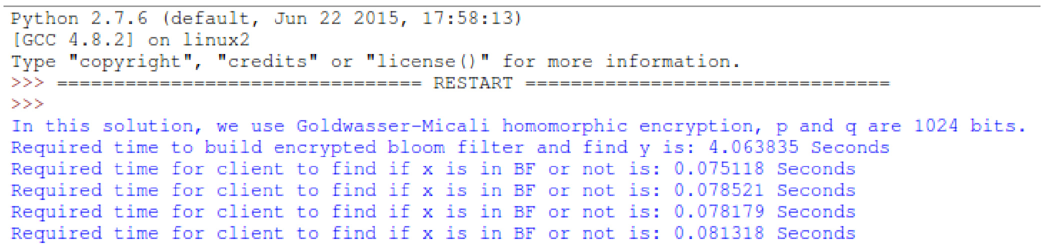
Size of X	Size of the Filter	Error Rate	Required Time in Seconds
2^{10}	2^{14}	0.001	3.975636
2^{11}	2^{15}	0.001	7.553784
2^{12}	2^{16}	0.001	15.151596
2^{13}	2^{17}	0.001	29.917604
2^{14}	2^{18}	0.001	60.001534

Table 2: Encrypting B in Protocol 1 on Ubuntu 14.04.3

We repeat the same measurements to encrypt B in Linux operating system. The results are shown in the table 2.

Extrapolating from the table in Linux, the estimation of the required time to create the encrypted Bloom filter in our case (size of X is 2^{21}, filter size is 2^{25}) is 2 hours.

A client C who wants to perform a PMT for a document x should decrypt 10 bits of EB at the worst case, hence he should send at the most 10 requests to the server to acknowledge the quadratic residue characteristics of his results z. In average, one such a query takes 0.0079 seconds and thus ten queries demand 0.08 seconds. Figure 9 shows few examples of these queries in Linux.



```
Python 2.7.6 (default, Jun 22 2015, 17:58:13)
[GCC 4.8.2] on linux2
Type "copyright", "credits" or "license()" for more information.
>>> ================================ RESTART ================================
>>>
In this solution, we use Goldwasser-Micali homomorphic encryption, p and q are 1024 bits.
Required time to build encrypted bloom filter and find y is: 4.063835 Seconds
Required time for client to find if x is in BF or not is: 0.075118 Seconds
Required time for client to find if x is in BF or not is: 0.078521 Seconds
Required time for client to find if x is in BF or not is: 0.078179 Seconds
Required time for client to find if x is in BF or not is: 0.081318 Seconds
```

Figure 9: The Required Time for C to Get Responds from S

We consider the same scenario as before to evaluate the performance of the implementations that have been done for the second protocol. Moreover the modulus N in RSA is 2048 bits long.

Size of X	Size of the Filter	Error Rate	Required Time in Seconds
2^{10}	2^{14}	0.001	50.792000
2^{11}	2^{15}	0.001	97.828000
2^{12}	2^{16}	0.001	190.720999
2^{13}	2^{17}	0.001	380.545000
2^{14}	2^{18}	0.001	757.915000

Table 3: Protocol 2 - Insert an Encrypted Database to B in Windows 7

We replicate the same procedure to check if it scales well with the increasing of database size. Thus we choose a number of databases that vary in size to compute the required time for the server to generate an encrypted database and insert it to the Bloom filter. Table 3 gives an extraction of results.

According to table 3 and considering our case scenario, the time estimation for generating a Bloom filter for the encrypted database in protocol 2 is 27 hours. Once more, we repeat the same measurements in Linux and the results are shown in the table 4.

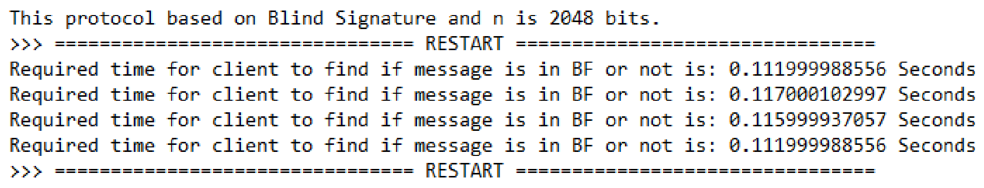
Size of X	Size of the Filter	Error Rate	Required Time in Seconds
2^{10}	2^{14}	0.001	4.616233
2^{11}	2^{15}	0.001	8.345523
2^{12}	2^{16}	0.001	17.149652
2^{13}	2^{17}	0.001	35.326805
2^{14}	2^{18}	0.001	68.880740

Table 4: Protocol 2 - Insert an Encrypted Database to B in Ubuntu 14.04.3

Based on the results in table 4, the approximate required preprocessing time for the server who runs his protocol in Linux is 2.5 hours.

Here, unlike the previous protocol, the client C who desires to accomplish a PMT for a file x sends only one request to the server. In average one such a query takes 0.11 seconds, as it is shown in Figure 10.

As explained in the end of chapter 3, we simulate the implementation of the third protocol by utilizing garbled circuits instead of an OPRF. The implementation takes place based on JustGarble algorithm using an x86-64



```
This protocol based on Blind Signature and n is 2048 bits.
>>> =============================== RESTART ===============================
Required time for client to find if message is in BF or not is: 0.111999988556 Seconds
Required time for client to find if message is in BF or not is: 0.117000102997 Seconds
Required time for client to find if message is in BF or not is: 0.115999937057 Seconds
Required time for client to find if message is in BF or not is: 0.111999988556 Seconds
>>> =============================== RESTART ===============================
```

Figure 10: The Required Time for C to Get one Respond from S

Intel Core i7-970 processor clocked at 3.201 GHz with a 12MB L3 cache [28]. In this machine, the garbling of one block of AES128 takes 637 μs and its evaluation takes 264 μs.

For the third protocol, in order to indirectly estimate the speed of one PMT query in the client side, we assume that $K = 0$ for all inputs and therefore the Bloom filter and its encryption are identical. By assuming that $K = 0$, we have the advantage of utilizing the results in the paper [37] by Kreuter et al. They generate and evaluate an AES128 circuit utilizing an Intel Core i5 processor clocked at 2.53 GHz with a 4GB 1067 MHz DDR3 memory. As we mentioned before, for a Bloom filter with 10 hash functions, two evaluations of AES128 is required and according to Kreuter et al. this takes 5.4 seconds.

We finalize this chapter with a discussion about false positive error rate of the Bloom filters. The different possibilities for a query from a Bloom filter is summarized in table 5.

	Real Positive	Real Negative
Observed Positive	True Positive	False Positive
Observed Negative	False Negative = 0	True Negative

Table 5: A Query from a Bloom Filter

We remark that the false positive rate of the Bloom filter, obtained from $(1 - e^{-nl/m})^l$ is equal to

$$\frac{\text{number of false positive results}}{\text{number of false positive} + \text{true negative results}}$$

and thus to compute a total of false positive accrues in a real tryout of B, one should compute

$$\frac{\text{number of false positive results}}{\text{number of true positive + false positive + true negative results}}.$$

In other words, if a bits of a Bloom filter with m bits and l hash functions, are set to one then the probability of total false positive in practice can be computed with

$$\frac{\binom{a}{l}}{\binom{m}{l}}\,.$$

Measuring from many experiments with a Bloom filter that has reached full capacity, we notice that roughly 30% of the bits are set one.

Considering the values of parameters in our scenario, the probability of false positive in a real tryout of B is as follows

$$\frac{\binom{0.3\times(2^{25})}{10}}{\binom{2^{25}}{10}} \approx 5\times 10^{-6}\,.$$

5 Comparison

We devote this part of our work to compare the three suggested protocols in chapter 3 with each other. In our comparison, we consider security and efficiency of each one of the protocols for the both parties; the server and the client.

5.1 Security

In this section we take a closer look at the security of our three protocols. It is shown that after executing these protocols, the server's and the client's privacy are preserved. Moreover, we show that the protocols are secure against malicious client and also malicious server. Finally, we present some use cases for the protocols, based on their security differences.

The first protocol uses Goldwasser-Micali homomorphic encryption to compute result $z = H(j||H_i(x))r^2y^{mask}$. In order to encrypt $H(j||H_i(x))$, there are lots of possibilities (because of the variety to choose r and $mask$), but it is shown in [20] that the decryption is always unique. As explained before, random square r^2 and $mask$ respectively hide the index i and its quadratic residue characteristic. Therefore, at the end of the protocol, the server doesn't learn anything about the client's query and client's privacy is preserved.

On the other hand, it is computationally difficult for the malicious server or an adversary who obtains such z to decode it, because this decryption is identical to deciding quadratic residuary modulo composite numbers [20]. This means that the first protocol is secure against the malicious server. Of course, a malicious server could put non-malicious files into its database and leave malicious ones intentionally out. In this way, the server could fool the client but this kind of malicious server is beyond the scope of this thesis.

We note that if $N = pq$ where p and q are distinct odd prime numbers, then half of the numbers in $\mathbb{Z}_N^*$ are quadratic non-residues modulo N. A reader who is interested in more details regarding this matter can consult [38]. We recall from the third chapter of this thesis that in order to encrypt the Bloom filter in the first protocol, for all index $i \in \mathrm{B}$, if $H(j||i)$ is a QNR modulo N then B_i should be flipped. Thus, distribution of 1 and 0 in the encrypted Bloom filter are random and the number of bits which are set to one and zero, are equal. This means EB does not reveal any clue about the size of the server's database and therefore, after executing protocol 1, the semi-honest client would only learn whether his item x belongs to the server's database and server's privacy is preserved.

Finally, if the Bloom filter has l hash functions, after each PMT, at the most l bits of the filter would be revealed to the client. Thus, a malicious client who is interested to decrypt the filter, can not obtain more than l bits per query. Considering our scenario in chapter 4, where the Bloom filter has 2^{25} bits and 10 hash functions, then after each query 10 bits of 33.5 million bits will be decrypted. The server can change H or N, and hence the encrypted Bloom filter at any time. Therefore it can guarantee that nobody has time to learn all the values of B and consequently this protocol is secure against malicious client.

Let us assume that a client knows beforehand that for some j, $\mathrm{B}_j = 0$ and for some x and some i, $H_i(x) = j$. Therefore, this client concludes that $x \notin X$ without the help of the server. If the client knows the value of u bits in the encrypted Bloom filter, the probability that at least one of $H_i(x)$ is among those u bits is:

$$1 - \left(1 - \frac{u}{m}\right)^l .$$

If $u = 5000$, $l = 10$ and $m = 2^{25}$ then this probability is 0.15% which is very small in practice.

In the second protocol, the client blinds his record x with a random $r \in \mathbb{Z}_N$ and sends $y = H(x)r^e$ modulo N to the server. This guarantees that the server S can not learn the client's record x and therefore client's privacy is preserved. On the other hand, a malicious server which has a collection of $H(x_i)$ where x_i are known records to this server could perform a brute-force attack to recover the client record x from $y = H(x)r^e$. Considering that r is randomly chosen from $\mathbb{Z}_N$, this action is computationally infeasible and therefore the protocol is secure against malicious server.

After each instance of PMT, the client C learns which l bits of the Bloom filter are representing his item x. However, for some x', if C does not know $Sig(x')$ then he does not know which bits of B are relevant to x'. Therefore, protocol 2 is secure against malicious client. Unlike the first protocol, there is no need for the server to change its Bloom filter after certain number of queries.

The server in the second protocol, encrypts the database to preserve its secrecy but S does not encrypt the Bloom filter. Therefore, the number of bits set to one in the filter can be turned to a good estimate of the size of the server's database X. This implies that the second protocol might not be a good choice for a server who wants to keep the size of its database as a secret.

In the third protocol, the Bloom filter is encrypted and its hash functions remain secret to preserve server's secrecy. To perform a membership test, even after several queries from EB, there is always need to seek help from the server. This means that protocol 3 is secure against malicious client. Moreover, utilizing an OPRF F_k to find the right Bloom filter entries brings secrecy for the client because after evaluating this OPRF, the malicious server can not recover $H(x)$.

Protocol	Cryptographic primitive	Security issues for the Server
1	Goldwasser-Micali of [20]	Some queries possible without S
2	Blind Signature of [22]	Reveals the size of database
3	OPRF of [25]	

Table 6: Comparison of the security in three protocols

To sum up, if the size of the server's database is important to remain secret, the first and third protocols should be utilized. If the server wants to prevent the client from independent queries (i.e. without seeking help from the server), protocols two and three must be used.

5.2 Efficiency

In this section, we compare the efficiency of the three protocols in detail. First, the amount of memory that is used to run the protocols in the server side, is presented. Then, we compare the communication and computation complexity of the protocols. Finally, we compare the result of our measurements, which are obtained in the fourth chapter, to present the situations that each one of the protocols fits in.

In the first and third protocols, the size of the encrypted Bloom filter and the Bloom filter are equal. In the second protocol, the number of elements in the encrypted database is equal to the number of items in the server's database. Furthermore, as shown in table 7, all the three protocols use the same space to store the Bloom filter.

The server which utilizes protocol 1 or 3, should store both B and EB for further updates. On the other hand, if the server uses protocol 2, it should store both database and database of signed records for later changes. Therefore, for very big databases, protocol 1 and 3 are recommended.

Protocol	Versions of database	Versions of Bloom filter	Size of B
1	X	B and EB	m
2	X and its signed	B	m
3	X	B and EB	m

Table 7: Comparison of the space complexity in three protocols

As mentioned earlier about the distribution of quadratic residue numbers over $\mathbb{Z}_N^*$, if r is a random number then the probability that Jacobi$(r, N) = 1$ is 50%. Thus, to encrypt the Bloom filter in protocol 1, for each index of B in average two evaluations of Jacobi symbol is required. The time complexity (i.e. the required time for an algorithm to run) to evaluate one Jacobi symbol is $O(\log(N)^2)$.

If the client C wants to query for x, in average, he needs to evaluate $2l$ Jacobi symbols. He also requires to perform $2.5l$ multiplications modulo N. This is because of the fact that the value of $mask$ in $z = H(j||H_i(x))r^2y^{mask}$ is equal to one, with the chance of 1/2.

S transfers EB (m bits) to C. For each H_i, C transfers one element of $\mathbb{Z}_N$ to S, and S responds with one bit. Therefore for each query, C sends l elements of $\mathbb{Z}_N$ to S, and S's response is l bits.

In the second protocol, in order to encrypt the database, the server is required to produce one RSA signature for every element in X. To perform a query, the client and the server both need to compute l exponentiations modulo N.

In order to encrypt the Bloom filter in the third protocol, the server should evaluates one OPRF F for every element in the database X, and m times OPRF K to generate the encryption key. To perform a query S and C together must evaluate F one time and the evaluation of K should be done l times.

To compare the complexities of the three protocols, we consider the values that are presented in our scenario in chapter four. Therefore the Bloom filter has 2^{25} bits, 10 hash functions and the size of database is 2^{21}. This comparison is shown in table 8.

Protocol	Preprocessing by S	Query for S	Query for C
1	2^{26} Jacobi symbols	20 Jacobi symbols	20 Jacobi symbols
2	2^{21} signature	1 signature	1 modular exp.
3	2^{21} evaluations of OPRF F and 2^{25} evaluations of OPRF K	1 evaluation of OPRF F and 10 evaluations of OPRF K	1 evaluation of OPRF F and 10 evaluations of OPRF K

Table 8: Comparison of the complexities using different protocols

In order to perform a PMT for the client's element x, in each protocol different numbers of communication between C and S are required as follows:

- In protocol 1, at most l times.
- In protocol 2, one time.
- In protocol 3, one time for F and at the most l times for K's.

We recall that to estimate the performance of protocol 3, the garbled circuits are used instead of an oblivious pseudorandom function. Therefore, the client and the server should utilize an oblivious transfer protocol to transfer the garbling key for x. This makes one query utilizing protocol 3 to be much slower than protocols 1 and 2. Therefore, in the use cases that the speed of the communication between S and C is a key factor, protocols 1 and 2 are more suitable.

Protocol	Preprocessing time in Windows	Preprocessing time in Linux	Time for one query
1	6.7 hours	2 hours	0.08 seconds
2	27 hours	2.5 hours	0.11 seconds

Table 9: Comparison of the performance in protocol 1 and 2

We compare the performance of the protocols 1 and 2 in the table 9.

From the table 8, we can conclude that the preprocessing time can be reduced in the both protocols 1 and 2, with utilizing Linux operating system instead of the Windows platform. Furthermore, the preprocessing time can be reduced significantly with parallelizing the encryption of the server's Bloom filter and database, respectively, in the first protocol and the second one.

We emphasize that to utilize JustGarble a particular hardware crypto accelerator is needed [28]. Thus, protocols 1 and 2 are more practical for general use cases.

If the Bloom filter has 2^{25} bits, then in all the three protocols, the amount of data that needs to be transfered from S to C is 4 MB. For a frequently changing database, it might not be feasible solution to transfer the Bloom filter upon each query.

In protocol 3, if the server picks the function F in such a way that the client can compute it without S's help, then the resulting protocol, like protocol 1, reveals the hash functions of the encrypted Bloom filter. On the other hand, assume that the server, instead of utilizing an OPRF, picks the value of function K to be always zero. Then, similar to protocol 2, in this modified protocol S is only required to decide which entries of the filter are relevant to x. Summarizing, the third protocol is, in a certain sense, more general than protocols 1 and 2.

6 Alternative Approaches to PMT and PIR

This chapter elaborates a few alternative approaches to solve the problem of constructing a privacy preserving queries and PIR. First we present a variant of Bloom filters that is demonstrated to achieve a lower false positive error rate than a normal Bloom filter. Then we discuss about other possible methods to utilize the Bloom filters to perform a PMT, than the proposed protocols in chapter 3. Finally we introduce another approach to PIR with the support of a trusted hardware.

6.1 Variable-Increment Counting Bloom Filter

While Bloom filters are reportedly used in well-known products, unfortunately the problem of false positive makes them not suitable for the situations where more accurate results from a membership test are required. Attempts to generate a more efficient variant of Bloom filter are of interests to many researchers [16]. A counting Bloom filter, a variant that supports deletion, is not usable for many networking devices because it needs large amounts of memory space. Rottenstreich et al. proposed a variant of Bloom filters called *Variable-Increment Counting Bloom Filter* (VI-CB) and analytically show that utilizing the VI-CB in practical systems always achieve a lower FP rate than CB [39].

Informally, the difference between a CB and a VI-CB is in the way we update their counters. The counters increment in a VI-CB take place by variable numbers rather than by one. Therefore to implement a VI-CB one should first define a set of possible variable increments D. In order to increase an entry of the VI-CB we hash it to an element of D and increment the counter by its hash value in D. Furthermore, to determine whether an element belongs to this filter, one should check in each of its counters and discover if its hashed value in D could be found in that counter as part of the sum.

Figure 11 illustrates a toy example of a VI-CB with $D = \{4, 7, 8, 13\}$. First we insert the set $S = \{x, y\}$ in the filter and then we perform a membership test for an element z. As its shown in this figure, the third hashed entry of z has the value of 13, while its corresponding counter has the value of 16. We can conclude that $z \notin S$ because $16 - 13 = 3$ and $3 \notin D$. Clearly, if we utilize CB instead of VI-CB in a situation similar to figure 11, the query results in false positive.

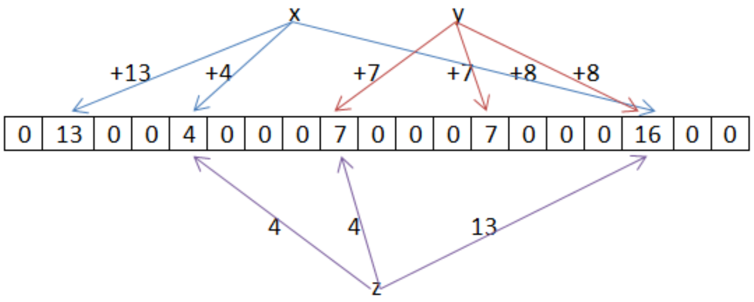


Figure 11: An Example of a VI-CB Represents $S = \{x, y\}$ and a Query for z

In order to implement a VI-CB, one should first define a set of possible variable increments D. Compare to the three suggested protocols, the server which utilizes the VI-CB needs to store D as well. On the other hand, the size of the counters are larger than one bit and consequently VI-CB occupies more space in the memory in comparison with the normal Bloom filter. Moreover, the counters are not always incrementing by one and therefore VI-CB occupies more space in the memory than CB. Thus, if the false positive error is the more important issue than the space complexity, the VI-CB should be utilized.

6.2 Alternative Methods to Utilize Bloom Filters for PMT

All the three proposed protocols of chapter 3 require to transfer a version of server's Bloom filter to the client. Although all the protocols are feasible for many use cases, in case of a frequently updating database, it might not be practical for the server to ship entire filter upon each new query. In this regard, we try to find alternative methods to perform PMT utilizing Bloom filters than the three suggested protocols.

One possible solution for the server is to insert the database in a Bloom filter with l hash functions and send all the hash functions to the client. Thus, the client can feed his item to those hash functions and get corresponding indexes of the filter. He then needs to somehow obtain the value of those array positions of the Bloom filter, if all of them are one, most probably client's item belongs to the server's database. In the following, we discuss about four different methods for the client to retrieve the value of his item's corresponding entries in the filter.

- One trivial way for the client is to send all the l indexes to the server. While one-wayness of hash functions make it impossible for the server to retrieve client's item, the server can have a collection of particular items (in our case, special documents). Therefore, it might be possible for the server to guess client's query. Moreover, the server can guess which clients have common items by analysing the intersection between their queries. In this solution if the server frequently changes the Bloom filter, then there would be no collision between two queries of two clients with an identical item but the other issue remains.

- The client can conceal the l indexes of the filter, which he is interested in, by hiding them between some other indexes. For instance, the client can add to the set of his item's indexes, l more randomly chosen entries. A potential problem is that the server can guess common items of different clients by analyzing the intersection of their queries. If dummy indexes are chosen too close to the real ones, after several queries of same item, 'median' of intersections would reveal the real indexes. In order to reduce the probability of collision detection, the client can increase the number of the dummy indexes. One mathematical solution to find the efficient number of the dummy indexes is utilizing birthday paradox [40]. If the Bloom filter has m bits then according to birthday paradox $\sqrt{m}$ dummy indexes are needed to increase the number of collision detections and therefore to better hide the client's query.

- The client can choose randomly between his indexes and send some of them at a time to the server. For instance, if there are ten indexes corresponding to each item, the client would first choose three of them and ask from the server whether they are zero or one. If even one of them is zero then query is done and the client's item doesn't belong to the server's database. In this case the client will not reveal all the indexes corresponding to his item and therefore the server is not able to guess the real item. The possibility of collision detection between queries of different clients is very low, at least when the selection of the indexes is randomized. In this scenario, if client's item belongs to the server's database, then all the indexes would be revealed to the server. But this problem is not a concern in our case because typically the malicious item will be revealed to the server.

- The server can store its database in two Bloom filters simultaneously and send two sets of hash functions with the size of l, to the client. In this way the client chooses between the sets of hash functions and get the corresponding indexes of one of the filters. He can then add some dummy indexes to his query and sends all the entries to the server. This makes the possibility of collision detection even smaller than before.

In all the mentioned methods, the client would reveal some information about his item. Although this information can not disclose the client's item, they are open for further analyses. For instance, the server or a third party can search in the client's set of entries to check for the indexes corresponding a particular item. If the size of client's set of dummy indexes is t, then the probability that given index is in dummy set is t/m. The probability that all the indexes of a special item are in the dummy set is $(t/m)^l$.

In conclusion, the above suggested methods should not be used if the secrecy of the client is the main purpose of the PMT.

6.3 An Approach to PIR Utilizing Trusted Hardware

Another approach to solve the problem of private information retrieval and also private membership test, is utilizing a *trusted hardware* (TH). The trusted hardware executes the PIR and PMT protocols honestly, in the sense that neither any adversary nor the server can tamper its execution.

Wang et al. suggested a hardware-based PIR model to reduce the communication complexity compared to the PIR models that do not utilize any TH [41]. Their scheme achieved the communication complexity of $O(\log n)$ where n is the size of the server's database.

Their TH is assumed to be secure under chosen plaintext attack (CPA-secure). They also consider no trust on the server. Outside attackers and the server are allowed to monitor all the queries and replies of TH, they are also able to perform a query to TH.

The proposed PIR model by Wang et al. involves a trusted third party (TTP) to initiate the system setup. TTP secretly picks a random permutation[10] π_0 and permutes the server's database X to X_{π_0}. He then selects a secret key sk_0 and encrypts X_{π_0} under this key, and delivers the encrypted X_{π_0} to the server. Finally, TTP sends π_0 and sk_0 to TH through a secret

[10]The reader who is interested in random permutation may consult [42] as one reference.

channel.

A client's query for ith element of the database is done as follows. TH who knows π_0, sends a request to the server to get the encrypted item X_{π_0}. Then TH decrypts the retrieved item with sk_0 and sends x_i to the client.

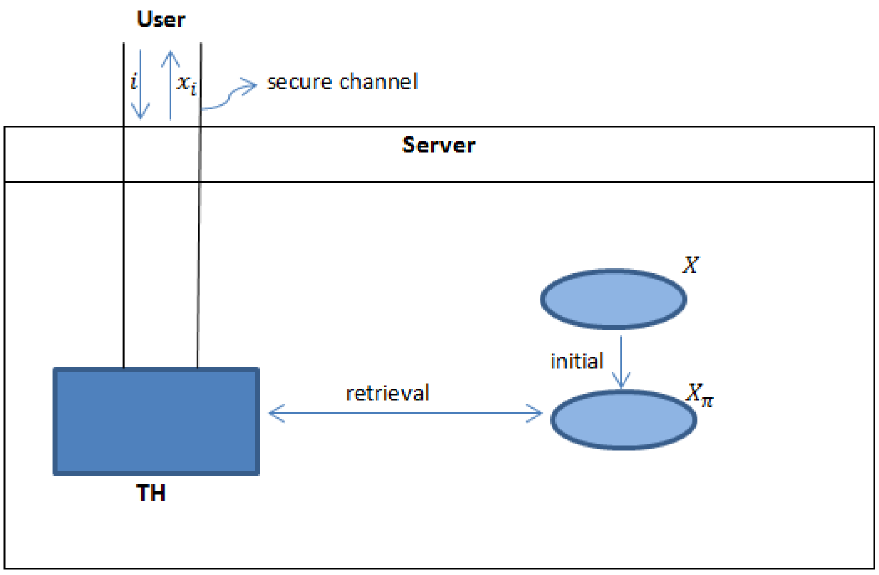


Figure 12: A model of utilizing TH for PIR

7 Conclusion

The contents of this thesis is summarized in this chapter. We discuss the presented solutions for the problem of PMT and analyze the results that are obtained from our implementations. We conclude this thesis with an outlook to possible further work in the field of PIR and PMT.

7.1 Summary

Demand of privacy grows in parallel with development of the Internet and electronic devices. Many researches are devoted to discover possible solutions to preserve privacy of the users on the Internet. For instance, in 1981 Michael O. Rabin introduced a protocol to exchange secrets that is called *oblivious transfer* [4]. In 1995, a weaker version of 1 out of n oblivious transfer scheme was introduced by Chor et al. that is called *private information retrieval* [6]. The aim of the PIR protocols is to provide methods for the users to query through databases without reveling which items they are interested in, to the database holders.

The main focus of this work is to study the problem of *private membership test*, in order to protect users from profiling. We attempt to design a protocol which allows the interaction between the clients and the server in a private manner. In this regard, we presented three different protocols in chapter 3, that are utilizing Bloom filters to perform PMT. For each protocol, we have used a different cryptographic primitive. In order to check the feasibility and efficiency of the suggested protocols, we have implemented them, as reported in the fourth chapter.

7.2 Discussion

We considered a realistic scenario to measure the performance of our solutions. In this scenario, a security company which hosts a server S, possesses a collection of malicious samples (e.g. document exploits) with the size of 2^{21}. This company stores the SHA-1 values of its malware signatures in a database X.

Each hash value that is produced by SHA-1, consists of 160 bits and therefore, the size of the database X is $2^{21} \times 160 \approx 3.3 \times 10^8$ bits (approximately 40 Megabytes). Although the size of the database is not huge, it is more space efficient for S to store X in a Bloom filter made up of 2^{25} bits (4 Megabytes). We assume that S utilizes 10 hash functions for its Bloom filter and modulus N in Goldwasser-Micali encryption and RSA signature is 2048 bits.

The server S can use any of the three suggested protocols to perform PMT with its Bloom filter. Depending on the privacy requirements, the choice of a protocol can be done. Moreover, each protocol has a different time complexity and based on the efficiency requirements, each can be suitable for a different use case.

If it is not acceptable for the server to reveal the hash functions of its Bloom filter to the client, then the third protocol should be used. On the other hand, protocol 3 needs a fast implementation for OPRF. If such an implementation is not available, then utilizing the third protocol is a trade off between security and efficiency.

If it is not acceptable for the server to reveal any information about the size of its database, then protocol 2 should not be used. On the other hand, the second protocol requires just one query in the client side to fulfill the membership test. Therefore, in the situation that the minimal of communication between the server and the client is required, protocol 2 is the best choice.

If it is not acceptable for the server that a client who collects many entries of the filter, perform a membership test independent from S, then the first protocol is not suitable. However, based on our implementation, protocol 1 is the fastest one.

In all the protocols, the size of the Bloom filter that S sends to the client is 4 Megabytes. Moreover, if there is an update in the database then the server needs to ship an updated filter to C. Therefore, utilizing either of the three presented solutions might not be practical for frequently changing databases.

It is possible to send the Bloom filter to each client with a unique identifier. Therefore, the server can keep track of clients' Bloom filter by utilizing the unique identifiers, and whenever the Bloom filter is updated, server can just ship the updated bits to clients. However, for the server who doesn't want to transfer the entire filter to the client, we presented four alternative solutions in chapter 6.

In all of these suggested solutions, further exploitation is possible from the clients queries. More specifically, the server can always *guess* whether a certain client possesses an item of special interest. Hence, in the use cases that the secrecy of the clients is the main goal of PMT, the four alternative solutions should not be used.

If the server has the advantage of being able to utilize a trusted hardware, then the membership tests can be done in a private manner, without any of the suggested cryptographic solutions in this work.

7.3 Further Work

We measured the performance of our implementation in Windows 7 and Ubuntu 14.04.3 operating systems. The comparison of these two platforms shows that Ubuntu 14.04.3 is a better choice to execute protocols 1 and 2. One can run these protocols in the different operating systems than Windows 7 and Ubuntu 14.04.3 (e.g. Mac and Android OS), as one possibility to extend our work in this thesis.

There is room to improve our implementation by parallelizing the encryption of server's Bloom filter or its database. This can significantly reduce the preprocessing time in the server side.

Seeking for other possible use cases for the three suggested protocols than an identifier to the clients' possibly malicious samples, can also be part of the further work.

One weakness of the counting Bloom filters is deleting a duplicate record. In other words, if by accident, a record is inserted in this filter more than once, then the corresponding counters are incremented more than one time. Therefore, in case of deletion, that record would not be removed from the filter. Moreover, if mistakenly the deletion occurs for an item that doesn't belong to CB, this may lead to get negative values in some counters. Thus, searching for a variant of the Bloom filters that keeps the track of records, is another possible extension.

We introduced the variable increment-counting Bloom filter in the sixth chapter. Although this variant of the Bloom filters can reduce the false positive error rate, as a result of utilizing bigger counters than a counting Bloom filter, using them might not be space efficient. The implementation of VI-CB can show the trade off between space complexity and false positive error rate.

Finally, the time complexities of the three protocols can be improved by finding a faster evaluation for a Jacobi symbol, a modular exponentiation and an OPRF.

This thesis is a spin-off from the Academy of Finland research project "Cloud Security Services".

References

[1] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, (8):30–37, 2009.

[2] Suranga Seneviratne, Aruna Seneviratne, Prasant Mohapatra, and Anirban Mahanti. Predicting user traits from a snapshot of apps installed on a smartphone. *ACM SIGMOBILE Mobile Computing and Communications Review*, 18(2):1–8, 2014.

[3] Michal Kosinski, David Stillwell, and Thore Graepel. Private traits and attributes are predictable from digital records of human behavior. *Proceedings of the National Academy of Sciences*, 110(15):5802–5805, 2013.

[4] Michael O Rabin. How to exchange secrets with oblivious transfer. *IACR Cryptology ePrint Archive*, 2005:187, 2005.

[5] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[6] Benny Chor, Eyal Kushilevitz, Oded Goldreich, and Madhu Sudan. Private information retrieval. *In 36th IEEE Conference on the Foundations of Computer Science*, pages 41–50, 1995.

[7] Eyal Kushilevitz and Rafail Ostrovsky. Replication is not needed: Single database, computationally-private information retrieval. In *focs*, page 364. IEEE, 1997.

[8] Rafail Ostrovsky and William E Skeith III. A survey of single-database private information retrieval: Techniques and applications. In *Public Key Cryptography–PKC 2007*, pages 393–411. Springer, 2007.

[9] Benny Chor, Niv Gilboa, and Moni Naor. *Private information retrieval by keywords.* Citeseer, 1997.

[10] Tommi Meskanen, Jian Liu, Sara Ramezanian, and Valtteri Niemi. Private membership test for bloom filters. In *The 14th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (IEEE TrustCom-15)*, August 2015.

[11] Ryo Nojima and Youki Kadobayashi. Cryptographically secure bloomfilters. *Transactions on Data Privacy*, 2(2):131–139, 2009.

[12] Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography.* CRC press, 1996.

[13] Henk CA Van Tilborg and Sushil Jajodia. *Encyclopedia of cryptography and security.* Springer Science & Business Media, 2011.

[14] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[15] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[16] Sasu Tarkoma, Christian Esteve Rothenberg, and Eemil Lagerspetz. Theory and practice of bloom filters for distributed systems. *Communications Surveys & Tutorials, IEEE*, 14(1):131–155, 2012.

[17] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z Broder. Summary cache: a scalable wide-area web cache sharing protocol. *IEEE/ACM Transactions on Networking (TON)*, 8(3):281–293, 2000.

[18] Ronald L Rivest, Adi Shamir, and Len Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[19] Michael O Rabin. Digitalized signatures and public-key functions as intractable as factorization. Technical report, DTIC Document, 1979.

[20] Shafi Goldwasser and Silvio Micali. Probabilistic encryption & how to play mental poker keeping secret all partial information. In *Proceedings of the fourteenth annual ACM symposium on Theory of computing*, pages 365–377. ACM, 1982.

[21] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard.* Springer Science & Business Media, 2013.

[22] David Chaum. Blind signatures for untraceable payments. In *Advances in cryptology*, pages 199–203. Springer, 1983.

[23] Mihir Bellare, Chanathip Namprempre, David Pointcheval, Michael Semanko, et al. The one-more-rsa-inversion problems and the security of chaum's blind signature scheme. *Journal of Cryptology*, 16(3):185–215, 2003.

[24] Shimon Even, Oded Goldreich, and Abraham Lempel. A randomized protocol for signing contracts. *Communications of the ACM*, 28(6):637–647, 1985.

[25] Michael J Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. Keyword search and oblivious pseudorandom functions. In *Theory of Cryptography*, pages 303–324. Springer, 2005.

[26] Oded Goldreich. Secure multi-party computation. *Manuscript. Preliminary version*, 1998.

[27] Andrew Chi-Chih Yao. Protocols for secure computations. In *FOCS*, volume 82, pages 160–164, 1982.

[28] Mihir Bellare, Viet Tung Hoang, Sriram Keelveedhi, and Phillip Rogaway. Efficient garbling from a fixed-key blockcipher. In *Security and Privacy (SP), 2013 IEEE Symposium on*, pages 478–492. IEEE, 2013.

[29] Jeffrey Rott. Intel advanced encryption standard instructions (aes-ni). Technical report, Technical report, Intel, 2010.

[30] Private Membership Test with Bloom Filters. Close project workshop: Posters and demos, https://wiki.aalto.fi/ display/CloSeProject/-CloSe+Project+Workshop+Posters+and+Demos/. Accessed : November 2015.

[31] Anna Ostlin and Rasmus Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*, pages 622–628. ACM, 2003.

[32] Jay Baird. 'python-bloomfilter', https://github.com/jaybaird/python-bloomfilter/. Accessed : February 2015.

[33] 'numpy', http://www.numpy.org/. License: BSD , Accessed : February 2015.

[34] 'the python cryptography toolkit', https://www.dlitz.net/software /pycrypto/. Accessed : February 2015.

[35] 'python-pkcs1', https://github.com/ bdauvergne/python-pkcs1/. Accessed : February 2015.

[36] 'bottle: Python web framework', http://bottlepy.org/docs/0.12/. Accessed :April 2015.

[37] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, volume 12, pages 285–300, 2012.

[38] David A Burgess. The distribution of quadratic residues and non-residues. *Mathematika*, 4(02):106–112, 1957.

[39] Ori Rottenstreich, Yossi Kanizo, and Isaac Keslassy. The variable-increment counting bloom filter. *IEEE/ACM Transactions on Networking (TON)*, 22(4):1092–1105, 2014.

[40] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[41] Shuhong Wang, Xuhua Ding, Robert H Deng, and Feng Bao. Private information retrieval using trusted hardware. In *Computer Security–ESORICS 2006*, pages 49–64. Springer, 2006.

[42] Lincoln E Moses. *Tables of random permutations*. Stanford University Press, 1978.

Appendices

A Protocol 1 - Preprocessing

```
#!/usr/bin/python
import random
import numpy as np
from pkcs1.primes import jacobi
from Crypto.Hash import SHA as sha1
from Crypto.Util.number import getPrime
from pybloom import BloomFilter

# Server Part
class BloomFilterEx(BloomFilter):
  def __init__(self,capacity,error_rate=0.001,
               prime_sz=1024):
    super(BloomFilterEx, self).__init__(capacity,
                                        error_rate)
    self.__data = np.zeros(self.num_bits,dtype=np.int)
    self.__prime_sz = prime_sz
    self.p = getPrime(prime_sz, randfunc=None)
    self.q = getPrime(prime_sz, randfunc=None)
    self.n = self.p * self.q
    y=2
    while jacobi(y,self.q)!= -1 or jacobi(y,self.p)!= -1 :
      y +=1
    self.y = y
    self.__encrypted = False
    return

  def __contains__(self, key):
    bits_per_slice = self.bits_per_slice
    bitarray = self.bitarray
    data = self.__data
    if not isinstance(key, list):
      hashes = self.make_hashes(key)
    else:
      hashes = key
    offset = 0
    for k in hashes:
      j = offset + k
      if data[j] == 0:
        return False
      offset += bits_per_slice
    return True

  def __change_pq(self):
    encrypted = self.__encrypted
    if encrypted:
      self.__decrypt()
    self.__p = getPrime(prime_sz, randfunc=None)
    self.__q = getPrime(prime_sz, randfunc=None)
    self.n = self.__p * self.__q
    y = 2
    while jacobi(y,self.__q)!=-1 or
              jacobi(y,self.__p)!=-1:
      y +=1
    self.y = y
    if encrypted:
      self.__encrypt()

  def response(self,z):
    #----- Server checks to see whether jacobi(z,n)=1
    if jacobi(z,self.n)!=1:
      print "z is not valid"
    #--------------------------------------------------
    #----server calculates jacobi(z,p) and jacobi(z,q)
    #----to see if it's a QR or not
    answer=""
    if jacobi(z,self.__p)==1 and jacobi(z,self.__q)==1:
      answer="QR"
    else:
      answer="QNR"
    return answer

  def encrypt(self):
    if self.__encrypted:
      return
    bitarray = self.bitarray
    n = self.n
  p = self.p
  for i in range(len(bitarray)):
    j=0
    stri = "x"+str(i)
    indexhash=int((sha1.new(str(j)+stri).
                 hexdigest()[:12]),16) % n
    while jacobi(indexhash,n)!= 1:
      j+=1
      indexhash=int((sha1.new(str(j)+stri).
                 hexdigest()[:12]),16) % n
    if jacobi(indexhash,p)==-1:
      bitarray[i] = not (bitarray[i])
  self.__encrypted = True
  return

def decrypt(self):
  if self.__encrypted:
    bitarray = self.bitarray
    data = self.__data
    for k in range(len(bitarray)):
      bitarray[k] = (data[k] != 0)
    self.__encrypted = False
  return

def indexes(self, key):
  bits_per_slice = self.bits_per_slice
  if not isinstance(key, list):
    hashes = self.make_hashes(key)
  else:
    hashes = key
  offset = 0
  indexes = []
  for k in hashes:
    indexes.append(offset+k)
    offset += bits_per_slice
  return indexes

def add(self, key, skip_check=False):
  bits_per_slice = self.bits_per_slice
  bitarray = self.bitarray
  data = self.__data
```

```
    hashes = self.make_hashes(key)
    if (not skip_check) and (hashes in self):
      return True
    if self.count > self.capacity:
      raise IndexError("BloomFilterEx is at capacity")
    offset = 0
    for k in hashes:
      j = offset + k
      if data[j] == 0:
        bitarray[j] = not (bitarray[j])
      data[j] += 1
      offset += bits_per_slice
    self.count += 1
    return False

  def remove(self, key):
    bits_per_slice = self.bits_per_slice
    bitarray = self.bitarray
    data = self.__data
    hashes = self.make_hashes(key)
    if not (hashes in self):
      return True
    offset = 0
    changed = False
    for k in hashes:
      j = offset + k
      data[j] -= 1
      if data[j] == 0:
        bitarray[j] = not (bitarray[j])
        changed = True
      offset += bits_per_slice
    if not changed :
      print "After deleting, filter didn't change!"
    return False

  def addnew(self, key, skip_check=False):
    bits_per_slice = self.bits_per_slice
    bitarray = self.bitarray
    data = self.__data
    hashes = self.make_hashes(key)
    if (not skip_check) and (hashes in self):
      return True
    if self.count > self.capacity:
      raise IndexError("BloomFilterEx is at capacity")
    offset = 0
    for k in hashes:
      j = offset + k
      if data[j] == 0:
        bitarray[j] = not (bitarray[j])
      data[j] += 1
      offset += bits_per_slice
    self.count += 1
    return False

  def copy(self):
    new_filter = BloomFilterEx(self.capacity,
                               self.error_rate, self.prime_sz)
    new_filter.bitarray = self.bitarray.copy()
    new_filter.data = self.data.copy()
    new_filter.__p = self.__p
    new_filter.__q = self.__q
    new_filter.n = self.n
    new_filter.encrypted = self.encrypted
    return

  def union(self, other):
    raise NotImplementedError("Union not implemented!")
    return

  def __or__(self, other):
    return self.union(other)

  def intersection(self, other):
    raise NotImplementedError("Intersect not implemented!")
    return

  def __and__(self, other):
    return self.intersection(other)

    return

  @classmethod
  def fromfile(cls, f, n=-1):
    raise NotImplementedError("not-implemented!")
    return

# Server object
class BloomServer:
  def __init__(self,filter_sz,filter_err_rate,
                               prime_sz):
    # __f is an enhanced Bloom filter(capacity,
    #                          error-rate,prime-size)
    self.__f=BloomFilterEx(filter_sz,filter_err_rate,
                               prime_sz)
    return

  def encrypt(self):
    self.__f.encrypt()
    return

  def indexes(self, key):
    return self.__f.indexes(key)

  def add(self, key):
    return self.__f.add(key)

  def remove(self, key):
    return self.__f.remove(key)

  def addnew(self, key):
    return self.__f.addnew(key)

  @property
  def bitarray(self):
    return self.__f.bitarray

  @property
  def n(self):
    return self.__f.n

  @property
  def y(self):
    return self.__f.y
  @property
  def p(self):
    return self.__f.p

  @property
  def q(self):
    return self.__f.q

  def response(self,z):
    return self.__f.response(z)

# Client object
class BloomClient:
  def __init__(self, server):
    self.bitarray = server.bitarray
    self.y = server.y
    self.n = server.n
    self.server_response = server.response
    self.indexes = server.indexes
    return

# Client execution part

def client_part(client, x):
  bits = client.bitarray
  n = client.n
  y = client.y
  indexes = client.indexes(x)
  finalresult = []
#-----------------------------------------------------------
  for i in indexes:
    j=0
    stri = "x"+str(i)
    indexhash=int((sha1.new(str(j)+stri).
                              hexdigest()[:12]),16) % n
    while jacobi(indexhash,n)!= 1:
      j=j+1
      indexhash=int((sha1.new(str(j)+stri).
                              hexdigest()[:12]),16) % n
    r = random.randint(1,n)
```

```
        randomsquare = r**2 % n
        mask = random.randint(0,1)
        resultz = (indexhash * (randomsquare) *
                                        (y ** mask)) % n

        # getting response from the server
        serverresult = client.server_response(resultz)

        #------------------------------------------------
        #--Client can decrypt the i'th bit of Bloom filter
        if (serverresult=="QR") ^ (mask==0):
          finalresult.append(not (bits[i]))
        else:
          finalresult.append(bits[i])
      return all(finalresult)
    def main():
    import pickle
    import argparse
    parser = argparse.ArgumentParser()
    subparsers = parser.add_subparsers(help='Operational
    modes.Create, add or remove items.',dest="subcommand")
    parser_create = subparsers.add_parser('create',
    help='Create a new filter.')
    parser_create.add_argument("--filtername",type=str,
    required=True, help="File name of the bloom filter to
    operate on.")
    parser_create.add_argument("--clientfilter",type=str,
    required=True,
    help="File name of the client bloom filter
    to operate on.")
    parser_create.add_argument("--estsize",type=int,
    required=True,
    help="Estimated size of the filter.")
    parser_create.add_argument("--FPprob",type=float,
    required=True,
    help="Propability of filter returning a false
    positive.")
    parser_create.add_argument("--set",type=str,
    required=True)
    parser_add = subparsers.add_parser('add',
    help='Add items to an existing filter.')
```

```
parser_add.add_argument("--filtername",type=str,
required=True,
help="File name of the bloom filter to operate on.")
parser_add.add_argument("--clientfilter",type=str,
required=True,help="File name of the client bloom
filter to operate on.")
parser_add.add_argument("--sha1input",type=str,
required=True)
parser_remove = subparsers.add_parser('remove',
help='Remove items from an existing filter.')
parser_remove.add_argument("--filtername",type=str,
required=True,
help="File name of the bloom filter to operate on.")
parser_remove.add_argument("--clientfilter",type=str,
required=True,
help="File name of the client bloom filter to
operate on.")
parser_remove.add_argument("--sha1input",type=str,
required=True)
arg = vars(parser.parse_args())
args = parser.parse_args()
if args.subcommand == "create":
  bf_server = BloomServer(arg['estsize'],
                          arg['FPprob'], 1024)
  with open(arg['set'],'r') as ins:
  for line in ins:
    bf_server.add(line)
  bf_server.encrypt()
  filtername = open (arg['filtername'] ,'w')
  pickle.dump(bf_server,filtername)
  filtername.close()
  clientfilter = open (arg['clientfilter'] ,'w')
  a = BloomFilter(arg['estsize'], arg['FPprob'])
  a.bitarray = bf_server.bitarray.copy()
  pickle.dump(a,clientfilter)
  ##          pickle.dump(bf_server.n, clientfilter)
  ##          pickle.dump(bf_server.y, clientfilter)
  clientfilter.close()
  f = open('publickeys','w')
  pickle.dump(bf_server.n,f)
  pickle.dump(bf_server.y,f)
  f.close()
  #print 'p is', bf_server.p
  #print 'q is', bf_server.q
elif args.subcommand == "add":
  f = open (arg['filtername'],'r')
  bf_server = pickle.load(f)
  f.close()
  with open(arg['sha1input'],'r') as ins:
    for line in ins:
      bf_server.addnew(line)
  f = open (arg['filtername'],'w')
  pickle.dump(bf_server,f)
  f.close()
  a = open (arg['clientfilter'] ,'r')
  bf_client = pickle.load(a)
  a.close()
  bf_client.bitarray = bf_server.bitarray.copy()
  a = open (arg['clientfilter'] ,'w')
  pickle.dump(bf_client,a)
  a.close()
elif args.subcommand == "remove":
  f = open (arg['filtername'],'r')
  bf_server = pickle.load(f)
  f.close()
  with open(arg['sha1input'],'r') as ins:
    for line in ins:
    bf_server.remove(line)
  f = open (arg['filtername'],'w')
  pickle.dump(bf_server,f)
  f.close()
  clientfilter = open (arg['clientfilter'] ,'w')
  a = BloomFilter(arg['estsize'], arg['FPprob'])
  a.bitarray = bf_server.bitarray.copy()
  pickle.dump(a,clientfilter)
  pickle.dump(bf_server.n,clientfilter)
  pickle.dump(bf_server.y,clientfilter)
  clientfilter.close()
  else:
  print 'error:one-of-subcommands-must-be-given'
if __name__ == "__main__":
main()
```

B Protocol 1 - Server Part

```
#File: server.py
from bottle import Bottle, run, static_file, request
import pickle
from appendixA import *

server = Bottle()
server.route('/privatequery')
def hello():
  return "Please proceed to '/privatequery/getfilter
  /filter.dat' to get the Filter"

@server.route('/privatequery/getfilter/clientfilter')
def getFilter():
  f = open ('clientfilter','r')
  clientfilter = pickle.load(f)
  f.close()

return static_file('clientfilter', root='./')

@server.route('/privatequery/getPublicKeys')
def get_public_keys():
  f = open ('publickey','r')
  keys = {}
  keys["n"]=pickle.load(f)
  keys["y"]=pickle.load(f)
  f.close()
  return keys

@server.route('/privatequery/query', method='POST' )
def queryResponce():
  f = open ('serverfilter','r')
  serverfilter=pickle.load(f)
  f.close()
  qres = {}
  indexes = request.json
  for (key,value) in indexes.items():
    if jacobi(value,serverfilter.p)==1 and
          jacobi(value,serverfilter.q)==1:
            qres[key]= 'QR'
        else:
            qres[key]= 'QNR'
    print qres
    return qres


port = 8080
host = 'localhost'
server.run(host=host, port=port, debug=True)
```

C Protocol 1 - Client Part

```
#File: client.py
import requests
import json
import shutil
import pickle
from pybloom import BloomFilter
import random
from pkcs1.primes import jacobi
from Crypto.Hash import SHA as sha1
import bitarray

class publickeys():
  def __init__(self):
    self.n = 0
    self.y = 0

def getFilterFile():
  filterUrl = 'http://localhost:8080/privatequery/
␣␣getfilter/clientfilter'
  responce = requests.get(filterUrl, stream=True)
  if responce.status_code == 200:
    fname = 'clientfilter'
    with open(fname, 'w') as f:
      shutil.copyfileobj(responce.raw, f)
  return True

def getPublicKeys():

  clientkeys = publickeys()
  url='http://localhost:8080/privatequery/getPublicKeys'
  responce = requests.get(url)
  keys = responce.json()
  clientkeys.n = keys['n']
  clientkeys.y = keys['y']
  with open('clientkeys','w') as f:
    pickle.dump(clientkeys,f)

  return True
```

```
def query():

  f = open ('clientfilter','r')
  clientfilter = pickle.load(f)
  f.close()
  x = input ('write-the-item-you-want-to-
␣␣check-in-Bloom-Filter:')
  checkitem = sha1.new(str(x)).hexdigest()+"\n"
  print 'Sha1(x)␣=',checkitem
  #——— indexes is a list which gives the
  #result of hash functions
  bits_per_slice = clientfilter.bits_per_slice
  hashes = clientfilter.make_hashes(checkitem)
  with open('clientkeys','r') as f:
    clientkeys = pickle.load(f)
  n = clientkeys.n
  y = clientkeys.y
  offset = 0
  indexes = []
  for k in hashes:
    indexes.append(offset+k)
    offset += bits_per_slice
    #——————————————————————————————————
  stop = raw_input('')
  print 'indexes␣of␣the␣bloom␣filter␣are:',indexes
  clientencrypt = []
  indexesList = []
  for i in indexes:
    j=0
    stri = "x"+str(i)
    indexhash=int((sha1.new(str(j)+stri).
    hexdigest()[:12]),16) % n
    while jacobi(indexhash,n)!= 1:
      j=j+1
      indexhash=int((sha1.new(str(j)+stri).
      hexdigest()[:12]),16) % n
    r = random.randint(1,n)
    randomsquare = r**2 % n
    mask = random.randint(0,1)
    resultz = (indexhash * (randomsquare) *
        (y ** mask)) % n
        clientencrypt.append(mask)
        indexesList.append(resultz)
    jsonData = dict(zip(range(len(indexesList)),
                        indexesList))
    stop = raw_input('')
    print 'Client␣encrypts␣the␣indexes:',jsonData
    url = 'http://localhost:8080/privatequery/query'
    responce = requests.post(url, json=jsonData)
    responce =  responce.json()
##      print "query",indexesList
##      print "responce", responce
    finalresult = []
    for i in range(10) :
        if responce[str(i)]=="QR":
            if clientencrypt[i] == 0:
                finalresult.append(clientfilter.
                        bitarray[indexes[i]])
            else:
                finalresult.append(not (clientfilter.
                        bitarray[indexes[i]]))
        else:
            if clientencrypt[i] == 0:
                finalresult.append(not (clientfilter.
                         bitarray[indexes[i]]))
            else:
                finalresult.append(clientfilter.
                           bitarray[indexes[i]])
#
    stop = raw_input('')
    if finalresult.count(False)==0:
        print "item␣is␣in␣bloom␣filter."
    else:
        print "item␣is␣not␣in␣bloom␣filter."



if __name__ == "__main__":
getFilterFile()
getPublicKeys()
query()
```

D Protocol 2

```
Protocol2
## Server Part
import time
start = time.time()
from pybloom import BloomFilter
f = BloomFilter(capacity,error rate)
from Crypto.Hash import MD5 as md5
from Crypto.PublicKey import RSA
from Crypto.Util.number import getRandomRange
loadedPrivate = RSA.generate(2048)
loadedPublic = loadedPrivate.publickey()
for x in range(capacity):
  messageHash = md5.new(str(x)).digest()
  Sign = loadedPrivate.sign(messageHash,
                     loadedPrivate.n)[0]
  f.add(Sign)
end = time.time()
print "Required-time-to-build-BF-is:",
                        end - start,"Seconds"
## Client Part
import time
start = time.time()
import random
message=random.randint(1,capacity)
def generateBlindFactor(RSAobj):
  return (getRandomRange(1, RSAobj.key.n-1,
                 randfunc=RSAobj._randfunc))
# generates r
r = generateBlindFactor(loadedPrivate)
messagehash = md5.new(str(message)).digest()
blindmessage = loadedPublic.blind(messagehash,r)
blindSigned = loadedPrivate.sign(blindmessage,
                              loadedPrivate.n)[0]
unblind = loadedPublic.unblind(blindSigned,r)
print unblind in f
end = time.time()
print "Required-time-for-client-to-find-if
message-is-in-BF-or-not-is:",end - start,"Seconds"
```