Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

RANGE QUERIES ON AN ENCRYPTED OUTSOURCED DATABASE

Relatore:
Prof. Ozalp Babaoglu
Co-relatore:
Prof. Erkay Savas

Presentata da:
Stefano Bernagozzi

I Sessione
Anno Accademico 2015/2016

Abstract

A seguito della larga informatizzazione che ha ormai interessato ogni settore, oggi la maggior parte dei nostri dati (sensibili o meno) è memorizzata in qualche server in rete, del quale il più volte non conosciamo nè l'ubicazione fisica nè il proprietario. Con lo sviluppo di questa tendenza quindi, è sempre più a rischio la sicurezza di tali informazioni. Sono infatti sempre più frequenti le notizie di attacchi informatici contro server per il recupero di dati, con scopi anche criminosi. Famosi, per citare qualche caso, lo scandalo che ha provocato il furto di dati subito da Ashley Madison nel luglio 2015 [1] ed il non meno grave furto alla Sony nel dicembre 2014 [2].
Per evitare che ciò accada si sta sempre più diffondendo l'uso della crittografia a protezione dei dati, tramite la quale viene garantito all'utente l'uso esclusivo (o circoscritto ad un ristretto gruppo di persone) delle informazioni messe a disposizione dal server. Tale metodo però non è sufficiente a garantirne l'assoluta riservatezza in quanto nella maggior parte di casi la crittografia viene utilizzata solo nella trasmissione dei dati, che vengono poi salvati in chiaro sui server, rendendone possibile la lettura e l utilizzo da parte del proprietario che potrebbe anche venderli o comunque mostrarli in pubblico. Inoltre se un criminale informatico riuscisse a ottenere l'accesso al server, verrebbe in possesso delle informazioni senza troppe difficoltà. Proprio per questi motivi, il metodo migliore per assicurare agli utenti l'inviolabilita delle loro informazioni, consiste nel crittografare i dati prima di inviarli al server, di modo che anche in caso di accesso forzato non sarà possibile recuperare alcun dato.
Di seguito analizzeremo e compareremo due sistemi che consentono di garantire tale sicurezza; quindi illustreremo il modo per implementarli successivamente su di un elaboratore.

Il primo metodo analizzato (al quale successivamente faremo riferimento come FSSI, Fake Searches and Shuffle Indexes)utilizza la struttura b+tree per l'indicizzazione dei dati, e successivamente la cifratura dei vari settori, l'utilizzo della cache, il rimescolamento dei nodi dell'albero e alcune false ricerche per nascondere al server le informazioni cruciali per recuperare i dati. Con questa soluzione, i dati, che inizialmente sono salvati su di un file o inseriti manualmente dall'utente, vengono indicizzati, criptati e spediti al server. A seguito dell'esecuzione di una query da parte del client sul database, i dati vengono recuperati a partire dalla radice dell'albero e vengono tenuti in cache per effettuare le query più velocemente. Quando il client ha terminato di utilizzare il database tutti i nodi dell'albero in memoria vengono scambiati di settore e rispediti al server, crittografandoli anche con un salting diverso. Inoltre per garantire che il server non intuisca quali settori vengono recuperati, vengono effettuate delle "fake searches", cioè al momento di recuperare ogni nodo si guardano i figli e se ne ha piu' di uno vengono recuperati alcuni settori che possono non essere utili. Questo algoritmo e' stato studiato dalle università degli studi di Milano e Bergamo e dal Politecnico di Milano congiunte in [18]

Il secondo metodo utilizza gli alberi binari di ricerca e si avvale del protocollo Path ORAM, ovvero un algoritmo che permette di nascondere quasi completamente ad un utente esterno il pattern di accesso alla memoria. A tale scopo la memoria interna del server viene suddivisa in blocchi di uguale unità. Tramite questa suddivisione il client impedisce al server di carpire informazioni sul contenuto della memoria. Questo metodo e' stato studiato approfonditamente dall'università di Berkeley in [16] [17].

Contents

Abstract **i**

1 Introduction **1**

1.1 Problem formalisation 1

1.2 Background knowledge 2

1.2.1 B+Tree and Binary Tree 2

1.2.2 AES algorithm 3

1.2.3 Path ORAM 10

2 State of the art **11**

2.1 Data encryption at rest 11

2.2 Data encryption in transit 13

2.3 Data encryption in use 14

2.4 Our Approaches 14

3 FSSI Implementation **15**

3.1 Server 15

3.2 Client 15

3.2.1 B+tree implementation 16

3.2.2 Cryptography 23

3.2.3 User Interface 24

4 Path ORAM Implementation **27**

4.0.1 Server 27

4.0.2 Client 28

5 Comparison and test 31

5.1 128 records 32

5.2 1024 records 39

5.3 16384 records 46

6 Conclusions 55

7 Ringraziamenti 57

Bibliography 59

Chapter 1

Introduction

1.1 Problem formalisation

This project is about retrieving data in range without allowing the server to read it, when the database is stored in the server. Basically, our goal is to build a database that allows the client to maintain the confidentiality of the data stored, despite all the data is stored in a different location from the client's hard disk. This means that all the information written on the hard disk can be easily read by another person who can do anything with it. Given that, we need to encrypt that data from eavesdroppers or other people. This is because they could sell it or log into accounts and use them for stealing money or identities. In order to achieve this, we need to encrypt the data stored in the hard drive, so that only the possessor of the key can easily read the information stored, while all the others are going to read only encrypted data. Obviously, according to that, all the data management must be done by the client, otherwise any malicious person can easily retrieve it and use it for any malicious intention. The first chapter introduces the structures used for the realisation of the project and some technical terms used in the second part.
The second chapter examines various encrypted databases and analyses them thoroughly. The third chapter concerns about the implementation of the first project and its details, while the fourth explain deeply the second project. Fourth chapter is about ORAM algorithm and its implementation, as well as some changes made from the original algorithm.

In the fifth chapter there is the testing of the project on some databases.
Last chapters explains the results obtained and the future work based on this project, with some possible uses of it.

1.2 Background knowledge

1.2.1 B+Tree and Binary Tree

A B+tree is a data structure used for ordering the keys in a database. Each node has a large number of children. They are used in many contexts, from filesystems, like NTFS, to databases. It is indexed on a single key and, differently from a standard binary tree, all the data is stored under the leaves, as it is shown in figure 1. This means that, if we want to retrieve a record from this database, we have a complexity of $o(h)$ where h is the height of the tree. The root and the internal node have more than $\frac{n}{2}$ children, where n is the maximum number of children that a node can have. All the data under the leaves is linked by a pointer that points to the record. Searching the specified key on this data structure can be easily made by following the path denoted by the keys in the internal nodes. It has a complexity of $O(\log_a n)$, where a is the branching factor of the tree and n is the height of the tree (defined as the number of levels in the tree). In our implementation we don't have any link between the leaves, that means we have to start from the parent for retrieve any sibling of the node, and also we have to start from the root to do range queries. We used this implementation because in order to to preserve the tree we need to keep in memory all the nodes between the root and the current, so that each node we retrieve, including the ones recovered from fake searches, must be attainable from the root.
A binary tree is a b+tree with a branching factor of 2.

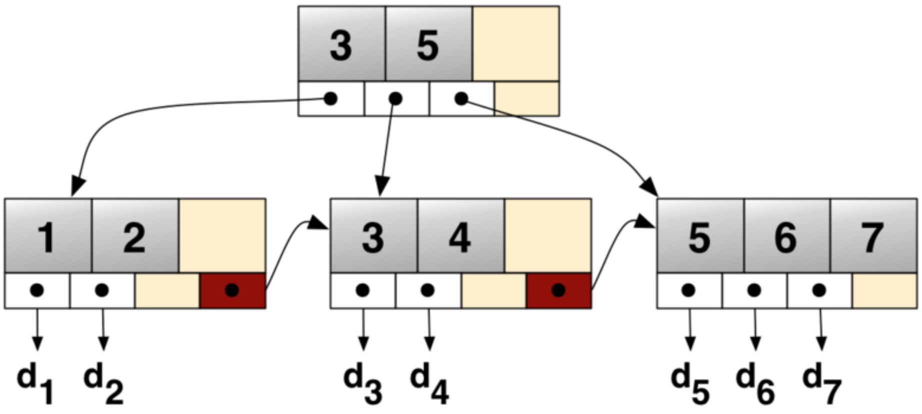


Figure 1: A graphical representation of a b+tree (source www.wikipedia.org)

1.2.2 AES algorithm

The AES algorithm (also called Rijndael from its creators Vincent Rijmen and Joan Daemon), is a symmetric encryption algorithm. Symmetric means that the same key is used both for encrypting and decrypting. It became effective in 2002, after the approval of NIST. This algorithm is a block cipher, i.e. it processes blocks of fixed length (128 bit) arranged on a matrix (called **State Array**) where the first word (each word is four bytes) is in the first column and so on.

The algorithm is composed of k rounds (as it is shown in figure 2) where k depends on the key length. In our case we have a 128 bit key, that means AES standard, therefore we have k equal to 10 plus the first add round key (in case of 196 bits k is 12 and 14 in case of 256 bits key).

$$StateArray = \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix}$$

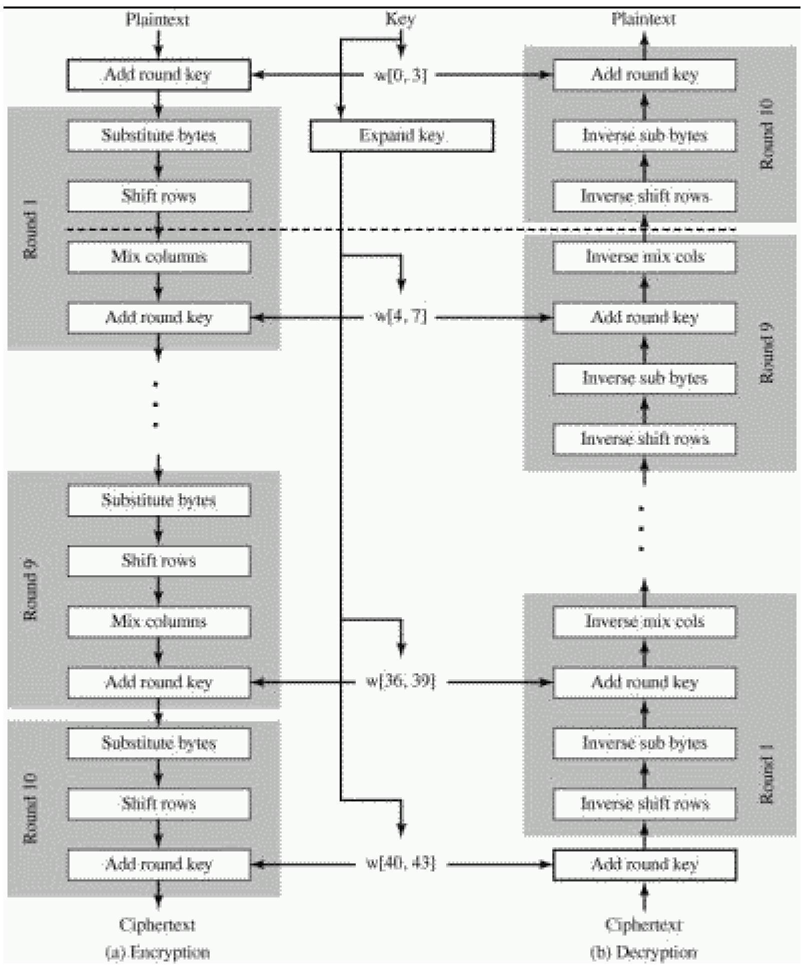


Figure 2: AES structure (source http://www.codeproject.com)

1.2.2.1 Key expansion algorithm

Before applying the algorithm the key must be expanded, because at each step the add round key function needs a portion of it. It is designed to ensure that a minimum change in the key (just only one bit is enough) should affect the next round keys for several times. A 128 bit key is expanded in the following manner: At first it is divided in 4 words as below:

$$\begin{bmatrix} k_0 & k_1 & k_2 & k_3 \\ k_4 & k_5 & k_6 & k_7 \\ k_8 & k_9 & k_1 0 & k_1 1 \\ k_1 2 & k_1 3 & k_1 4 & k_1 5 \end{bmatrix}$$
$$\Downarrow$$
$$\begin{bmatrix} w_0 & w_1 & w_2 & w_3 \end{bmatrix}$$

After this step the algorithm proceed four word at a time, as it is shown below in figure 4.

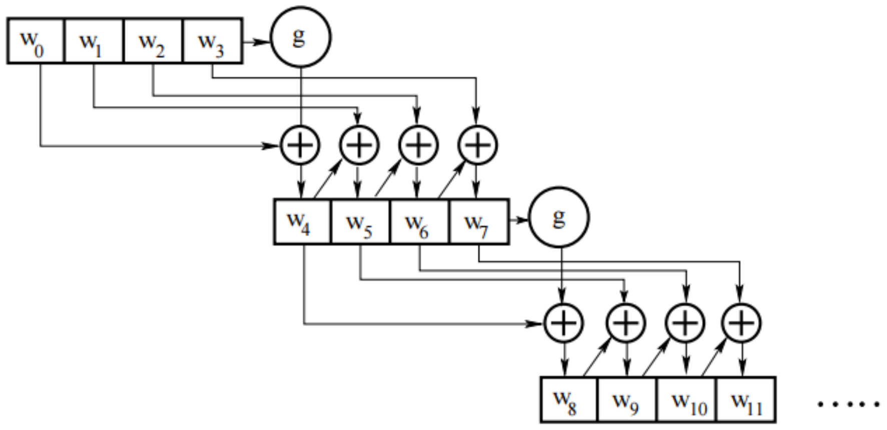


Figure 4: *The key expansion takes place on a four-word to four-word basis as shown here.* (*This figure is from Lecture 8 of "Computer and Network Security" by Avi Kak*)

Where $\oplus$ is the addition modulus 2 between the bits of the words and g is a mathematical function as below. $g()$ is composed by 3 steps:

1. One-byte left circular rotation on the 4 byte word.

2. Byte substitution for each byte of the word returned using the look-up table taken from the Substitute Bytes step.

3. XOR the bytes obtained from the previous step with a round constant. This step is important to destroy any symmetry that can exists in the previous steps.
   The round constant is derived at each round with the following formula (i starting from 1 is the number of the round):
   $Rcon[i] = (RC[i], 0x00, 0x00, 0x00)$
   Where:
   $RC[1] = 0x01$
   $RC[j] = 0x02 \times RC[j-1]$

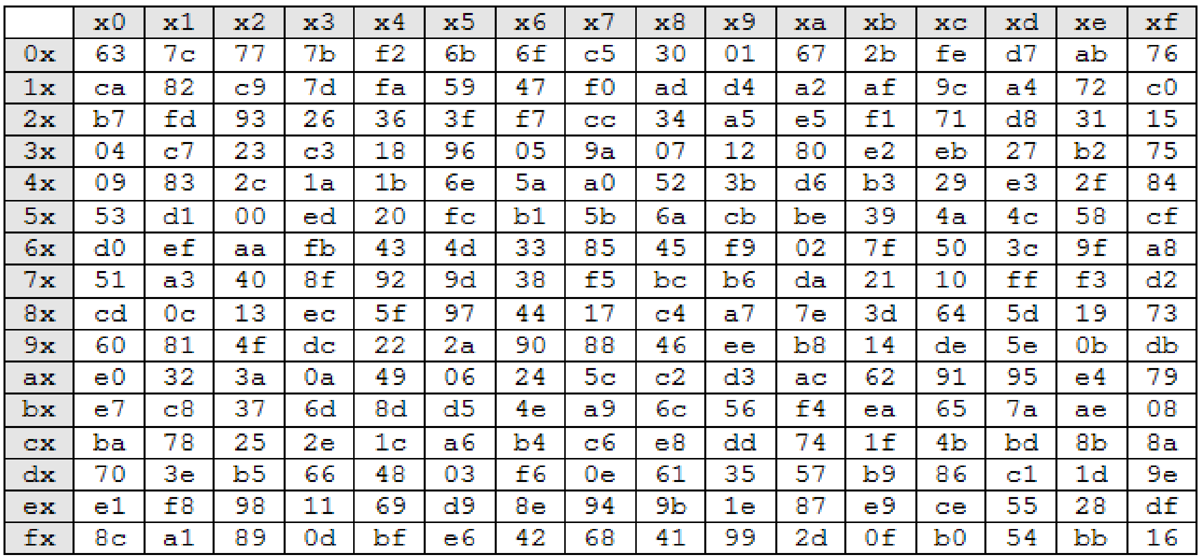
1.2.2.2 Encryption

Each round, except for the first (that is only an add round key) and the last, is composed by 4 steps:

1. substitute bytes
2. shift rows
3. mix columns
4. add round key

The last round doesn't have the mix columns step in order to make the algorithm reversible.

Substitute Bytes: The bytes substitution in the encryption works by replacing each byte with the corresponding value in the S-BOX, that is a 16×16 matrix (as it is shown below in figure 3). The look-up table is shown in hexadecimal value.



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	63	7c	77	7b	f2	6b	6f	c5	30	01	67	2b	fe	d7	ab	76
1x	ca	82	c9	7d	fa	59	47	f0	ad	d4	a2	af	9c	a4	72	c0
2x	b7	fd	93	26	36	3f	f7	cc	34	a5	e5	f1	71	d8	31	15
3x	04	c7	23	c3	18	96	05	9a	07	12	80	e2	eb	27	b2	75
4x	09	83	2c	1a	1b	6e	5a	a0	52	3b	d6	b3	29	e3	2f	84
5x	53	d1	00	ed	20	fc	b1	5b	6a	cb	be	39	4a	4c	58	cf
6x	d0	ef	aa	fb	43	4d	33	85	45	f9	02	7f	50	3c	9f	a8
7x	51	a3	40	8f	92	9d	38	f5	bc	b6	da	21	10	ff	f3	d2
8x	cd	0c	13	ec	5f	97	44	17	c4	a7	7e	3d	64	5d	19	73
9x	60	81	4f	dc	22	2a	90	88	46	ee	b8	14	de	5e	0b	db
ax	e0	32	3a	0a	49	06	24	5c	c2	d3	ac	62	91	95	e4	79
bx	e7	c8	37	6d	8d	d5	4e	a9	6c	56	f4	ea	65	7a	ae	08
cx	ba	78	25	2e	1c	a6	b4	c6	e8	dd	74	1f	4b	bd	8b	8a
dx	70	3e	b5	66	48	03	f6	0e	61	35	57	b9	86	c1	1d	9e
ex	e1	f8	98	11	69	d9	8e	94	9b	1e	87	e9	ce	55	28	df
fx	8c	a1	89	0d	bf	e6	42	68	41	99	2d	0f	b0	54	bb	16

Figure 3: S-BOX (source https://edipermadi.wordpress.com)

In this step the look-up table cannot be described as a mathematical function so the algorithm has to access the table every time.

Shift Rows: During the shift rows step each row of the state array is shifted as following:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \Longrightarrow \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,1} & s_{1,2} & s_{1,3} & s_{1,0} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,3} & s_{3,0} & s_{3,1} & s_{3,2} \end{bmatrix}$$

Mix Columns The mix columns step is composed by four operations, one for each row of the state array, as it is shown below:

- For the first row the operation can be stated as: $s'_{0,j} = (0x02 \times s_{0,j}) \otimes (0x03 \times s_{1,j}) \otimes s_{2,j} \otimes s_{3,j})$
- For the second row the operation can be stated as: $s'_{1,j} = s_{0,j} \otimes (0x02 \times s_{1,j}) \otimes (0x03 \times s_{2,j}) \otimes s_{3,j})$

- For the third row the operation can be stated as: $s'_{2,j} = s_{0,j} \otimes s_{1,j} \otimes (0x02 \times s_{2,j}) \otimes (0x03 \times s_{3,j}))$
- For the fourth row the operation can be stated as: $s'_{3,j} = (0x03 \times s_{0,j}) \otimes s_{1,j} \otimes s_{2,j} \otimes (0x02 \times s_{3,j}))$

Or more compactly: $$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \times \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

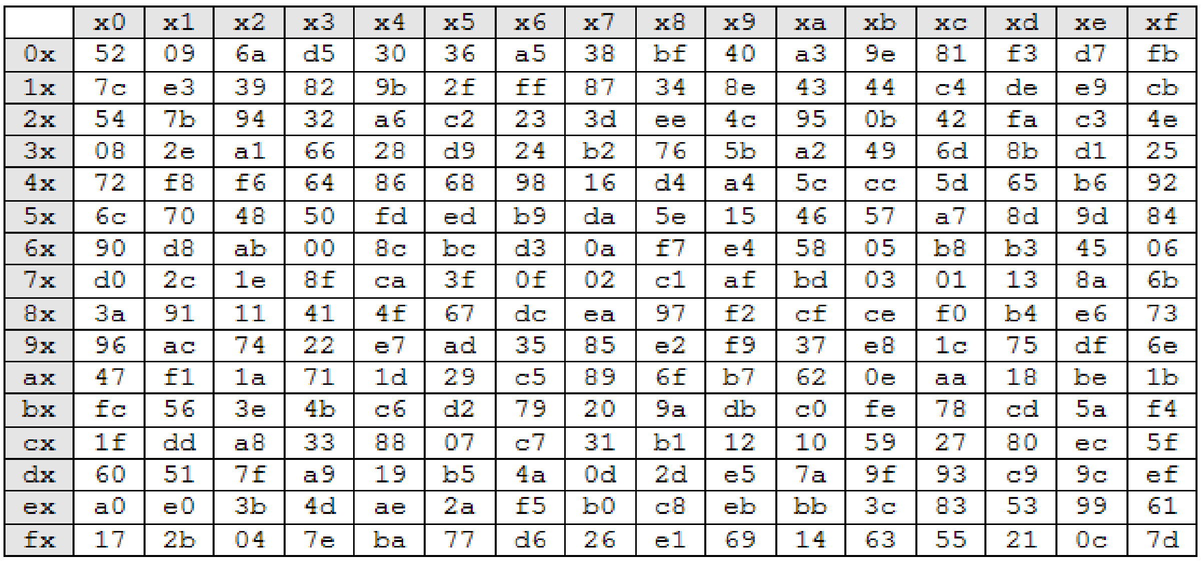
Add round key The add round key step is done by XORing each byte of the state with the relative byte of the key expanded. In the first round the input block is XORed with the first 4 words of the expanded key, and then at each of the following rounds the block is XORed with the next 4 words at a time.

1.2.2.3 Decryption

Each round, as it works in the encryption, is composed by 4 steps (first add round key and last round excluded):

1. substitute bytes
2. shift rows
3. mix columns
4. add round key

Inverse Substitute Bytes: Like in the encryption, the bytes substitution is made by replacing every byte with the corresponding value. Differently from the encryption the value is searched in the inverse S-BOX (as it is shown below in figure 4).



	x0	x1	x2	x3	x4	x5	x6	x7	x8	x9	xa	xb	xc	xd	xe	xf
0x	52	09	6a	d5	30	36	a5	38	bf	40	a3	9e	81	f3	d7	fb
1x	7c	e3	39	82	9b	2f	ff	87	34	8e	43	44	c4	de	e9	cb
2x	54	7b	94	32	a6	c2	23	3d	ee	4c	95	0b	42	fa	c3	4e
3x	08	2e	a1	66	28	d9	24	b2	76	5b	a2	49	6d	8b	d1	25
4x	72	f8	f6	64	86	68	98	16	d4	a4	5c	cc	5d	65	b6	92
5x	6c	70	48	50	fd	ed	b9	da	5e	15	46	57	a7	8d	9d	84
6x	90	d8	ab	00	8c	bc	d3	0a	f7	e4	58	05	b8	b3	45	06
7x	d0	2c	1e	8f	ca	3f	0f	02	c1	af	bd	03	01	13	8a	6b
8x	3a	91	11	41	4f	67	dc	ea	97	f2	cf	ce	f0	b4	e6	73
9x	96	ac	74	22	e7	ad	35	85	e2	f9	37	e8	1c	75	df	6e
ax	47	f1	1a	71	1d	29	c5	89	6f	b7	62	0e	aa	18	be	1b
bx	fc	56	3e	4b	c6	d2	79	20	9a	db	c0	fe	78	cd	5a	f4
cx	1f	dd	a8	33	88	07	c7	31	b1	12	10	59	27	80	ec	5f
dx	60	51	7f	a9	19	b5	4a	0d	2d	e5	7a	9f	93	c9	9c	ef
ex	a0	e0	3b	4d	ae	2a	f5	b0	c8	eb	bb	3c	83	53	99	61
fx	17	2b	04	7e	ba	77	d6	26	e1	69	14	63	55	21	0c	7d

Figure 4: Inverse S-BOX (source https://edipermadi.wordpress.com)

Shift rows During the shift rows step each row of the state array is shifted as following:

$$\begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} \Longrightarrow \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,3} & s_{1,0} & s_{1,1} & s_{1,2} \\ s_{2,2} & s_{2,3} & s_{2,0} & s_{2,1} \\ s_{3,1} & s_{3,2} & s_{3,3} & s_{3,0} \end{bmatrix}$$

Mix columns The mix columns step is composed by four operations, one for each row of the state array, as it is shown below:

$$\begin{bmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{bmatrix} \times \begin{bmatrix} s_{0,0} & s_{0,1} & s_{0,2} & s_{0,3} \\ s_{1,0} & s_{1,1} & s_{1,2} & s_{1,3} \\ s_{2,0} & s_{2,1} & s_{2,2} & s_{2,3} \\ s_{3,0} & s_{3,1} & s_{3,2} & s_{3,3} \end{bmatrix} = \begin{bmatrix} s'_{0,0} & s'_{0,1} & s'_{0,2} & s'_{0,3} \\ s'_{1,0} & s'_{1,1} & s'_{1,2} & s'_{1,3} \\ s'_{2,0} & s'_{2,1} & s'_{2,2} & s'_{2,3} \\ s'_{3,0} & s'_{3,1} & s'_{3,2} & s'_{3,3} \end{bmatrix}$$

Add round key The add round key step in the decryption is the same as the encryption.

1.2.3 Path ORAM

An Oblivious RAM algorithm is an interface between the client and the server whose purpose is to obfuscate data access pattern in ram or in other outsourced resources. From the server's perspective data access patterns from two sequences of read or write operations with equal length must be equivalent. Originally it was studied for prevent reverse engineering on software, but today with the large use of cloud computers it has gained many applications in privacy preserving storage outsourcing applications. This algorithm:

- Guarantees the indistinguishability between reading or writing;
- Allows the encryption of the storage;
- Prevents the recovery of the time of last access;
- Carries out the obfuscation of access pattern;
- Guarantees the indistinguishability of the same data after the rewrite

At the beginning the memory is divided into $Z \cdot N$ blocks of size B, which are atomic units, this means if the client wants to access a section of the block it must access the entire block. Since data on the server is treated as a binary tree, where each node is a bucket that contains a fixed number of blocks (up to Z real blocks, if there are less blocks is padded with dummy content). At any time each block is mapped into a leaf of the tree. Since block are re-encrypted at each time the server cannot retrieve any useful information about them because each time it changes the salting, so the server is not able to retrieve if it was changed or not.

Chapter 2

State of the art

At the date of writing, there are several implementations of encrypted databases. Since the server is a part that cannot be fully trusted, most of the implementations are supposed to encrypt the data before it reaches the server. There are many levels of encryption, depending on how the data is encrypted and if it is all encrypted or only a part of it. Another important aspect is how data is managed: in many cases data is only stored encrypted while all the process is done in clear-text, meanwhile some databases have implemented various algorithms for processing data in their encrypted form, so they don't have to manage clear-text. All of the methods described below are not related to any particular encryption method. According to these various methods we can divide them into 3 main categories:

2.1 Data encryption at rest

This type of encryption is also called TDE (Transparent Data Encryption) because it is transparent to the user. The algorithm works by encrypting the database after the client ends the connection, and stores it to the disk ciphered, to prevent any malicious attacker from reading the database while it is not used. Of course, unless they recover the key. After that, it restores the database in plain-text only when the client needs to update it, so only in that moment the attacker can read it. A graphical representation of the algorithm is shown in figure 5. Encrypting data at rest has the advantage that one doesn't have to change the application level. In particular, all the applications that execute queries over the database remain the same. This means that all the data is in clear-text while the computation is done and, most importantly, while it is being sent through the network. This implies that any malicious eavesdropper can read any query or data retrieved and use it for his purposes. The first TDE implementation was done by Microsoft in SQL Server 2008 [5]. Following Bill Gates's company, IBM [7] and Oracle [6] have implemented their own encrypted databases of this type. The company with headquarters in New York has implemented a software co-related with DB2, while Oracle has upgraded his own database. Due to the passage of clear-text over the network this implementation is not related to our work, despite this type of database is widely used, since is the most transparent and quick encryption for a database. The interesting thing about this type of encryption is that it can be integrated to record/column encryption, so it can add more security to the database (but increases the query time because of the double encryption). An implementation like the last one was done by Google[8]. Such algorithm can also be used for encrypting file-systems, like has done Windows with EFS[9].

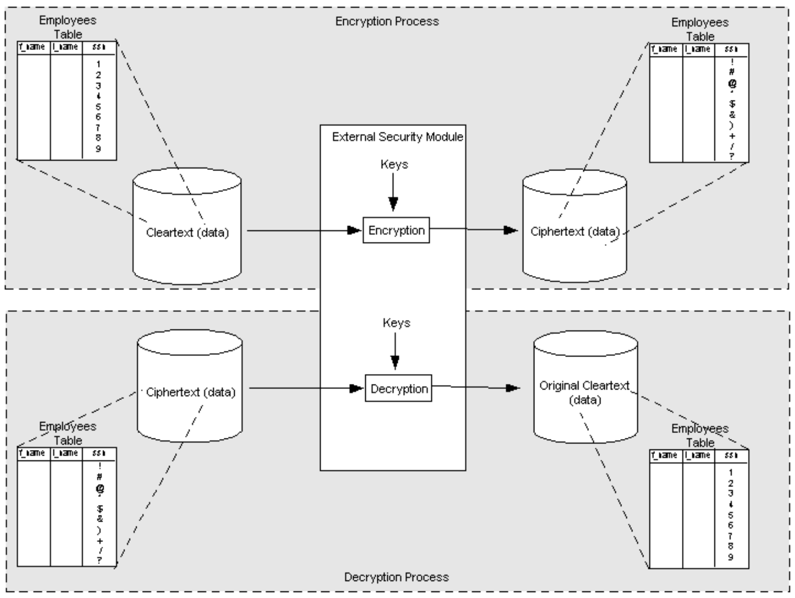


Figure 1: A graphical representation Transparent Data Encryption (source www.oracle.com)

2.2 Data encryption in transit

The second method for encrypting data is to encrypt them ”in transit”, that means all the computation is done in plain-text. The advantage is that since the computation occurs client side, the data is encrypted before it is sent to the server, so all the packages that go through the network are ciphered, so if an eavesdropper recovers them, they cannot use them for malicious purposes. This implementation relies on ciphering the single column of the database or, as it is in our cases, single cells/records of it. Another interesting fact about this method is that the key is stored client side and never goes to the server, so the client has the security of it. Since the server only stores the data its part is very minimal and most of the computation is done by the client, then we cannot do very high usages since the client normally is not a powerful computer. This method is widely used for file-systems, but today many companies are implementing it in their databases, since this ensures the client more security. This method has been studied in the USA by [10].

2.3 Data encryption in use

The last method relies on managing encrypted data, this means as soon as the data is inserted by the user it is encrypted in a particular manner. This allows the server to retrieve the data although it is encrypted. The main technique for the implementation of this databases relies on homomorphic encryption. This type of encryption is particular because it allows anyone (not just the key-holder) to output a cipher-text that encrypts f(1, . . . , t) for any desired function f, with given cipher-texts that encrypts 1, . . . , t, as long as that function can be efficiently computed. No information about 1, . . . , t or f(1, . . . , t) should leak; the inputs, output and intermediate values are always encrypted. It has been studied in various articles, like [12], [13], [11] and [14].

2.4 Our Approaches

Our approaches are based on data encryption in transit. In fact we encrypt all the packets before sending them to the server. In the first project the only information that the server has in plain-text are: the length of the data received and the initialization vector. Since the last one is different for each encryption it must be added to the data. Since we are using an AES-256 encryption those information are not useful for decrypting any message, so the server hasn't got any useful information. The other information that the server can retrieve are the level of the sector (and also if the sector is a node or a record) in the tree, but it doesn't have any useful information about the links between the sectors. Those things guarantee the security of our algorithm. In the second project the only information that the server can retrieve is that the client is retrieving and sending a block. Since blocks can have many sectors inside the server cannot retrieve any useful information about it.

Chapter 3

FSSI Implementation

Our implementation is based on C++ programming language. Most of the computation is done by the client, since the server cannot know which data is manipulating. This method allows the client to maintain the control over the clear text data. In particular, the server cannot read it unless it recovers the key from the client. This maintains very strongly security of the database, since no one that has access to the server can obtain the true information from the data.

3.1 Server

The server side is very small in this program because of the confidentiality of the data. Due to this, the only task of the server is to interact between the client and the hard disk. It's implemented in C++ but most of its parts are written in C.
Since we want a fast program the server part is a class of the program and the only thing that it does is retrieving the encrypted sectors from the client, copy them into its local ram and adding them into a map between the number of the sector and the data.

3.2 Client

Since most of the computation is done by the client, this part is the most important of the project. This section is divided into 4 parts: the first one is related to the implementation of the b+tree and how to serialize it, the second part regards the encryption of all the sections of the tree, the third one is related to bandwidth usage and the fourth one is about the user interface. Obviously if the server has no data stored, the client begins locally a new tree and after it has created all the tree with local data, it can do search operations on them and after that it sends all the data to the server for storing it. For the cryptography is used the openssl library, that implements an AES 256 bit. For more information see [15].

3.2.1 B+tree implementation

The code is based on a b+tree written in C (for more specification see [3]) with some changes: our implementation has no link between the leafs, is encrypted and stores the nodes in an external hard drive. The structures used for the realization of the tree are:

- *node:* A node is the internal part of the tree and is composed by:
  - a key that is a 32 bit integer;
  - an array of keys, that is associated with the pointers array and the sectors array
  - an array of sectors;
  - an array of pointers;
  - a pointer to the parent node;
  - an integer that counts the number of the keys;
  - an integer that mantain the sector in which it was stored.

  it is important to distinguish between leafs and internal nodes because in the leafs the pointers corresponds to the pointers of the record with that key, while in the internal nodes the first pointer points to a node that contains the keys lower than the first key, the second points to a node with keys higher or equal to the first key and lower than the second and so on until the last node that points to a node with keys higher or equal to the last-1 key.

- *record:* A record is the part of the tree where all the data is stored. It is collocated on the leafs of the tree and is composed by:
  - a 32 bit integer that is the key;
  - a pointer to an array of fields, in which all the fields of the database are stored. It isn't allocated before for allowing the client to choose the number of fields.
  - a pointer to the parent node
  - an integer that is the number of the sector.
- *info:* The info is stored in the first sector of the hard drive and is composed by:
  - an integer that is the sector of the root;
  - an integer that is the last sector used;
  - an integer that is the number of fields.

3.2.1.1 Auxiliary functions

The client has some auxiliary functions that are used during the program. 3 functions are made for the creation of records and nodes: make_node() and make_record() are made for allocating a new node and a new record. Each of these functions assign at the node/record a sector that is equal to the last sector used + 1 and updates the info− >last_sector_used global variable, the map and the set of sector used. this is necessary for later storing the tree. The second also take as input the key and an array with the fields and initialize the record with them. Another useful function is make_leaf(), that takes a node as input and sets the boolean variable is_leaf as TRUE.

3.2.1.2 Search

The search has 2 main functions:

- *Single record queries:* This search is done with the function find that takes as input a pointer to the root and an integer that contains the key. After that is called the function find_leaf(), that explores the tree and searches for a leaf that can contain the specified key. Since it is a b+tree the search has a logarithmic complexity in the size of the database, but it is slowed by fake searches, because they retrieve more nodes. In case such leaf exists, the find() function scrolls the keys of it and searches if the key is found. Since all the keys are ordered it is very easy to look if a key exists. In the affirmative case the function returns the record, otherwise it returns NULL.

- *Range queries:* This type of search, that is very important in the project, begins with the function find_and_print_range(): The function itself is not very important because it declares a list of records and calls the core of the search: find_range(). This function is built by doing a Breadth first search in the tree, this means the search is done level by level. The BFS implementation begins with declaring a queue, that contains the nodes to be explored. After that it begins a loop where each node is explored. At this point we can have 2 conditions:
    - *The node is internal:* in this case the algorithm looks the keys of the node. Then it is divided in 3 cases:
        - *The first pointer:* For the first pointer the only comparison needed is if the first key is less or equal than the lower extreme of the range. In this case the algorithm retrieves the node and puts it into the queue of nodes. Otherwise it doesn't do anything.
        - *The internal pointers:* In this case for each pointer it is checked if the previous key is lower or equal than the end of the range or if the key with the same index as the pointer is higher than the key with the same index of the pointer. In case one of the two conditions is valid the algorithm retrieves the node and puts it into the queue of nodes. Otherwise it doesn't do anything.
        - *The last pointer:* For the last pointer is checked if the last - 1 key (that is denoted by num_keys) is lower than the higher extreme of the range. If it is the algorithm retrieves the node and puts it into the queue of nodes. Otherwise it doesn't do anything.
    - *The node is a leaf:* In this case the algorithm checks every key and if they are in the range it inserts the records into the list. If the records are not in the

cache they are retrieved by a fake search.

At every time of the step if a node or record isn't cached, it is retrieved with a fake search.

3.2.1.3 Insertion

The insertion of a record into a tree starts by calling the insert() function. It takes as input 3 arguments: a pointer to the root,an integer that contains the key and an array with the fields of the record; and returns a pointer to the root of the tree. It starts by checking if the root isn't NULL, if it isn't the algorithm calls the function find() and searches for the record. In case a record is found it returns the root.

If it isn't found the function creates a new record with the data passed by the caller. at this point it checks if the root is NULL, in this case creates a new node, it is marked as leaf and the record is inserted in the first pointer, with the corresponding sector and key. After that the node just created is returned.

In case a tree already exists but no record with the specified key is found, it calls the function find_leaf() and checks if there's enough space in the leaf returned. In the affirmative case, the record is inserted in the right position in the leaf (all the keys must be in ascending order) and the root is returned. In this case the root remains the same. In the negative case, the leaf must be splitted, and so on the function insert_into_leaf_after_splitting() is called. This function takes as input a pointer to the root of the tree, a pointer to the leaf previously found, an integer that contains the key that must be inserted and a pointer to the record previously created.

This function calculate the point where the leaf must be splitted, then creates a new node and puts in it pointer and keys higher and equal to the middle key, leaving the space for the key that must be inserted free. After that it inserts the key and calls the function insert_into_parent(). This last call is necessary because the first key of the new node must be inserted in the parent node. It takes as input a pointer to the root of the tree, the pointers to the previous leaf and to the next leaf, and an integer that contains the key that must be inserted in the parent. This function begins by checking the parent of the first leaf. If the parent is NULL this means we have the root and we need to insert the 2 leafs into a new root. So the algorithm creates a new root and puts the two leafs as child of it, with the function insert_into_new_root(). In case the parent exists, then the index of the left leaf (the old one) is retrieved in the parent's sectors. After that it is checked if the parent has enough space to contain the new leaf, in this case with the function insert_into_node() the new leaf is inserted into the parent by putting the key after the old node's key and shifting all the following ones by 1. In case there isn't enough space, then the function insert_into_node_after_splitting() is called. This function works more or less the same as insert_into_leaf_after_splitting(), with the only difference that the latter works on leafs, so key indexes are equal to pointer and sector indexes, while in the nodes the key indexes that must be checked are 2. After creating the new node the function insert_into_node() is called again until it doesn't have any node to split, after that the root pointer is returned.

During all the insertion operations if the node is not in cache(that means the client hasn't retrieved it yet) then it is retrieved by doing a fake search as below.

3.2.1.4 Storing The Tree

Storing the tree and sending them to the server can happen in 2 methods: the first is when the client wants to send it and calls the command q, the second is when the client wants to close the connection with the server. This part is done by calling the function send_tree(), that shuffle the sectors of cached nodes and records, serialize them into a string, encrypt them, sends it to the server and frees the client's storage. Since send_tree() is a standalone function we have divided our program into 2 different programs:

- $FSSI1$ re-sends the tree after doing all the queries;
- $FSSI2$ re-sends the tree after each query.

We have opted for these 2 solutions because with the first there is less data usage but more client's storage is used, while with the second there is a high data usage but less client's memory is used. This is the only difference between the 2 programs.

Shuffling The Sectors Shuffling the sectors is useful so the server cannot establish a connection between the nodes and their position in the tree. For example, if we don't shuffle the sectors then the server can easily discover which is the root node and the nodes mostly retrieved by looking at the history of the requests. This part Is implemented by storing each sector retrieved from the server in a set, and then when we have to resend them to the server, at each node/record is assigned a sector taken randomly from the set, meanwhile the parent is updated with the right sector. This redistribution occurs staring from the root and, doing a BFS as above, assigns at each node/record a sector. When a sector is assigned it is removed from the set. During this part when all the sectors of the children of a node are reassigned, that node is serialized, encrypted and sent to the server, that stores it in the new sector. When all the internal nodes are sent to the server, the algorithm continues to the leafs. For each leaf if a record is cached in the client, that record's sector is changed and then sent to the server. When there is no child of the leaf cached, then it is sent the leaf. After the serialization of the tree the program serialize the struct with basics information for retrieving the tree in the first sector, so when it retrieves again the tree those information are easily retrievable.

Serialization

1. *Serialize Nodes:* The serialization of nodes begins by converting each integer from host size to network size (for the endianess of the processor)and by setting each pointer to NULL. After that it allocates an array with a size that is composed by a fixed part (that is the length of the node + 10) and a variable part to make the length of the array divisible by 16 (this is needed for AES algorithm). Following all the node is copied into the string and the remaining bytes are taken randomly (this is considered as salting.). The string obtained by this process is passed to the AES algorithm as explained below. Then it creates another string that contains:

    - in the first 4 bytes the size of the data encrypted in network size (this is used for the decryption),
    - the node encrypted,
    - 16 bytes that contains the iv (Initialization Vector, this also is used for adding randomness).

    This last string is sent to the server alongside the number of the sector.

2. *Serialize Sectors:* The serialization of sectors begins by converting each integer from host size to network size (for the endianess of the processor)and by setting each pointer to NULL. After that it allocates an array with a size that is composed by a fixed part (that is the length of the record + the length of the array with the fields + 10) and a variable part to make the length of the array divisible by 16 (this is needed for AES algorithm). After that all the record is copied into the string , following all the fields of the record and the remaining bytes are taken randomly (this is considered as salting.). The string obtained by this process is passed to the AES algorithm as explained below. Then it creates another string that contains:

   - in the first 4 bytes the size of the data encrypted in network size (this is used for the decryption),
   - the sector encrypted,
   - and in the end 16 bytes that contains the iv (this also is used for adding randomness).

   This string is sent to the server alongside the number of the sector.

3. *Serialize Info:* The serialization of the information is the same as the serialization of nodes changing only the size of the first array, that is composed by the size of the struct info instead of the size of the node.

3.2.1.5 Retrieving The Tree

For retrieving the tree (we suppose that we already have one stored in the server) we start by retrieving the information about the tree stored in the first sector of the hard drive (sector 0). Here can be found the necessary information for retrieving the tree:

- the sector where the root is stored,
- the last sector used (useful for inserting a new node/record in the tree),
- the number of fields in the records.

De-serialization When a sector is retrieved the first thing to do is to de-serialize it. This happens by reversing the algorithm applied in the encryption. In particular the first thing done is to retrieve the right length of the data, by taking the first 4 bytes and converting them into host endianess. After that the iv is retrieved, that is stored in the last 16 bytes. The last step is to decrypt the last part, that is done with the openssl library. Following the record/node is copied into a new node and all the numbers and pointers are set with the correct values (the numbers are restored in host endianess and the pointers are set to the right node/sector when it exists). In the end parent's pointers are set to the correct address. For the record is also allocated a new array of fields and all the values previously serialized are restored.

Fake Searches Fake searches are useful for confuse the server on which sector we are retrieving. They are used when we have to retrieve a node, both from the server and none. The algorithm takes a node and the index of the child to retrieve, then it takes n-1/2 other random indexes (where n is the number of keys in the node) and searches for them. If a sector is already cached then it passes to the next sector, otherwise if a sector is not cached it retrieves it from the server and updates the map and the set of the sector used. With this last step the next time we won't have to retrieve the sector from the server.

Cache Searches In our implementation we suppose to have a cache that stores all the nodes retrieved since that moment and not already sent again to the server. Here all the nodes retrieved from the tree are maintained, until the user decides to send them to the server. It is all stored into the ram and it is indexed with a map: this is a pair node/pointer.

3.2.2 Cryptography

The encryption is done node by node. An AES 256 bit for encrypting is used, and randomness is guaranteed by adding 10 or more random bytes at the end of the string that contains the node/record. Also a random iv is taken and put it alongside the data sent for the decryption. In client's program nothing is encrypted unless it has to be sent to the server, while in server's program all the data managed, except the sector and the iv, is cipher text. Since the most important thing is the key, this implementation stores the key into a file key.aes, so with this file the database can be retrieved from any computer. At the beginning the algorithm generates a random key, that is used for the rest of the computation. If the client wants to use another key, it should replace the file key.aes with another that contains the key, or delete it in order to create a new key. The algorithm used is the AES 256 EVP CBC, that is an interface for the encryption adopted by Openssl. CBC stands for cipher block chaining, that means the block encrypted are related to the previous block in the encryption.

3.2.3 User Interface

The user interface is made entirely via terminal. It is composed by 2 parts:

3.2.3.1 Set the database

The setting of the database is done step by step: At the beginning the user decides if he wants to create a new database or use an existing one: in the first case nothing is done because the database already exists and it is retrieved only when needed. In the second case the user needs to perform 2 steps:

- insert the number of fields he wants in each record,
- another choice: to insert them manually or take the fields from a file. If he wants to insert them manually he passes directly to the second part, otherwise he has to insert the name of the file and the index of the key.

3.2.3.2 Queries and insertion

This section is composed by 6 commands:

- **q** With this command the user stores all the data in the server, included the information in the first sector. After that he can decide to retrieve again the root and do other searches or close the connection.
- **c** This command closes the connection with the server and exits from the program.

- **r int1 int2** This command retrieves the root in case it is not already on the server and prints all the records that have the key between int1 and int2. If no record is found nothing is printed.
- **p int** This command prints on the terminal the record with key int1. If no record is found nothing is printed.
- **i int1 ... intN** This command inserts a new record in the tree. N must be equal to the number of fields.
- **a filename indexKey** This commands add to the tree all the rows contained in filename and takes the key stored in the position indexKey.
- **f filename** execute all the queries contained in filename.

Chapter 4

Path ORAM Implementation

This project was done by mer Mert Candan and it is about the use of Path ORAM protocol in range queries. It is based on [17] and adopted to be a database. Also this program is divided into client and server. Both client and server are a class in the program, but the client server cannot access to client's private variables.

4.0.1 Server

As it happened with the previous program the server side is very small. It is composed by 3 functions, the first is the constructor that allocates the necessary memory in the heap and initializes it with dummy contents. The amount of memory needed can be adjusted by the parameters given when the server is constructed in the main function. The server side is implemented as a binary tree where each node is a bucket, that contains up to Z blocks. The tree is constructed with a single-dimensional array, where the nodes of the tree are numbered from 1 to $2^{height-1}$. With this method we can reach child nodes of n by simply doubling it (2n) and reach the other child node by decrementing the first child by one (2n - 1). Since we are interested in leaf nodes rather than internal nodes, they can be accessed with indexes $2^L ... 2^{L+1} - 1$ where L is the last level of the tree (root node lies at L = 0). The server reaches the bucket at level l by first finding the leaf node and dividing its index value by 2 enough times that it reaches the required level.

4.0.2 Client

Since most of the computation is done by the client we're focusing on it. We start by illustrating the tree and therefore we will explain how the tree is managed, how the encryption works and how search works under it.

4.0.2.1 Storage

Client's storage is composed by 2 parts:

- The stash
- A position map of the blocks

Stash The stash is a sort of depot where during the course of the algorithm the overflowed blocks are stored. It has worst case size of $O(logN)$, but it is normally empty after each ORAM read/write complete operation.

Position map The position map is a structure that maps each block to a leaf node of the tree. This means that, if block a is mapped to a position $P(x)$, this block relies in a bucket along the path to leaf $P(x)$ or in the stash. Position map changes at each time a block is accessed.

4.0.2.2 Access

Access operation is done as pointed out in Berkeley's papers [16] and [17]. In fact this function is the core of the algorithm, since it encloses most of the obfuscation techniques of the program. Each entry in the position map contains a number between 0 and 2L-1. When I send this position to the server, it adds 2L to this position to find the true index of the leaf node on the binary tree. The buckets are returned to the client as a whole and at each access operation the client requests all the buckets that lies along the path of the leaf node. Here path implies the shortest sequence of nodes to follow in order to reach the root node. After all the buckets are read on the path of the specified leaf node, they are stored in the stash on the client side. If the operation is write then the specified block is changed with the given data. The buckets always contain Z blocks. If there aren't enough real blocks the remaining blocks are placed with dummy blocks. Since there may be dummy blocks returned from the server, the stash is cleared from these dummy blocks before the next step of the algorithm. At this point, even if the operation is read, the blocks on the stash are to be written back to the server via WriteBucket function. To write a block from stash to a bucket on the server there is a simple requirement: the path of the block (where it is mapped in the position map) should intersect with the bucket that is being written on.

4.0.2.3 Encryption

The encryption is performed on each block, that is composed by an id and the content. Both are encrypted together with AES in CBC mode. In order to distinguish the re-encryption of the blocks, they are padded with random characters so they cannot be related. Encryption re-occurs every time a block is read from the server.

4.0.2.4 Indexing

Indexing and search are the 2 differences between Path ORAM 1 and 2. This part is very important for the final results. The basic solution is that for each block a range is assigned, so they are easier to reach.

Path ORAM 1 In the first implementation we consider only the attribute that we want to index and we create the tree only on that index. This speeds up the algorithm because it doesn't need to consider all the other keys, but it takes more time to create it. This is due to less blocks used since most of the data is in the same range (the database is very dense). With scattered indexes is better because we can have more blocks and they have few records inside.

Path ORAM 2 The second one was the original one. It is indexed on all the 4 attributes. This speeds up the creation of the database but, since every query must search also on the other attributes, the query time slows down. This is very good for dense indexes because we can have more blocks and split the database into many parts.

4.0.2.5 Search

Since the aim of this project is to do range queries, the search is very important. Search works initially by mapping the query to all needed blocks, and afterwards by accessing each block with the access function. After that it searches for all the blocks and requests them to the server with the access function and at the end it returns the found records. Since all the blocks are indexed as previously the search for these blocks is very simple.

Chapter 5

Comparison and test

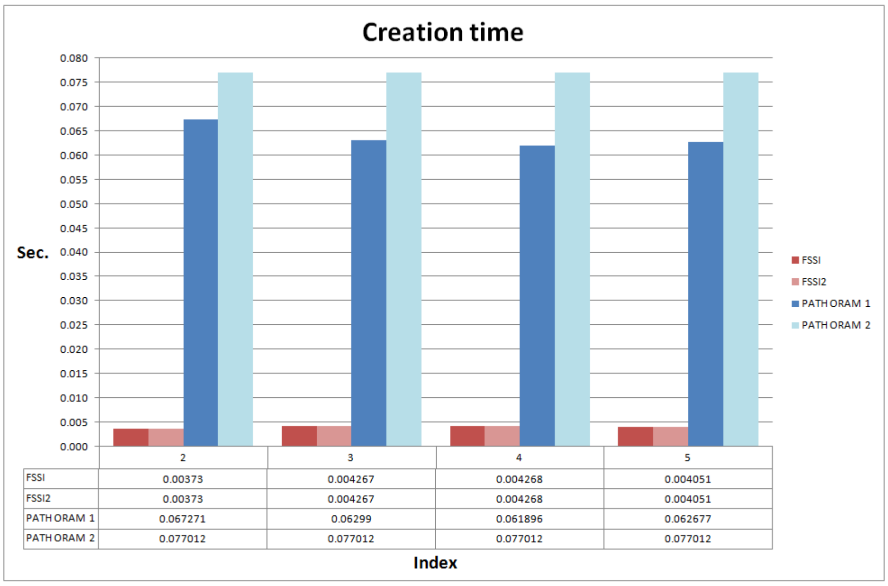
We tested the database on simple datasets, not very large (they can be found at [4], and each of them is less than 3 Megabytes).
The testing is done on a desktop computer with an Intel core i5-4460 quad core processor @ 3.20GHz, 4 GB of ram at 1600 MHz, 128 GB SSD hard drive. The results that we aim to show are mainly related to the comparison of network, memory occupation and execution time of the programs. Since both FSSI programs requires the same during the database creation we will have the same creation time. The average queries time includes the creation time, the queries time and the closing time until the database reach a quiet state where all the data is on the server. Indexes ranges are as follows:

- Second index: from 1 to 7
- Third Index: from 1 to 50
- Fourth index: from 1908 to 93340
- Fifth index: from 0 to 8

5.1 128 records

The first dataset used is composed by 128 records.

Creating database 128 records

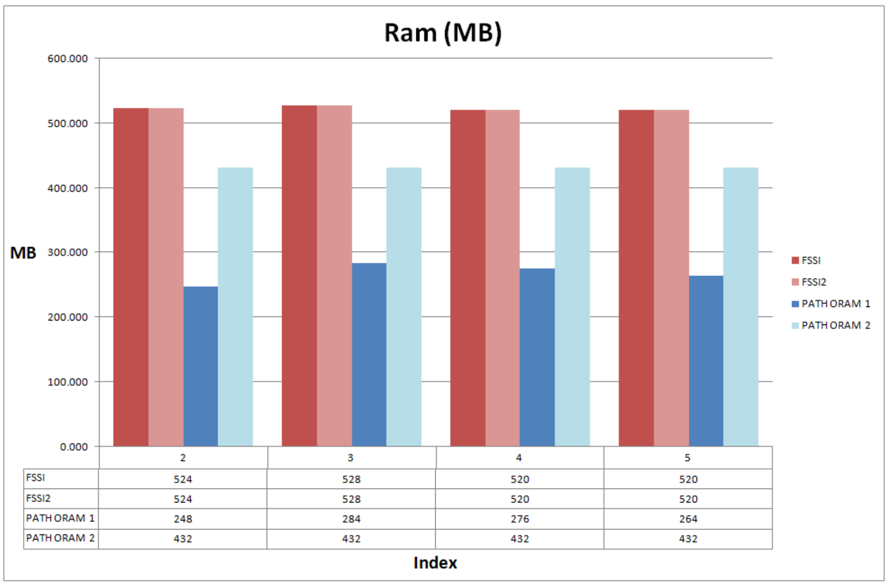


	2	3	4	5
FSSI	0.00373	0.004267	0.004268	0.004051
FSSI2	0.00373	0.004267	0.004268	0.004051
PATH ORAM 1	0.067271	0.06299	0.061896	0.062677
PATH ORAM 2	0.077012	0.077012	0.077012	0.077012

Figure 5.1: This graph shows the time required for creating the index of the database, encrypting and inserting all records into it with a dataset of 128 lines.

Creation Time In Fig. 5.1 we are comparing the time used by each program to create the database. That time includes reading all the records from the file, indexing, encrypting and sending them to the server. We can notice that FSSI is more or less 20 times faster than both ORAM algorithms, due to its local creation of the database which doesn't need to pass through the network and the simplicity of the algorithm that doesn't need to remap the blocks after each insertion. Another important fact is that ORAM 1 needs more or less the same time as ORAM 2. This fact is due to their similar insertion method and due to few records inserted, since at this point the size of the buckets is still small. The last thing that we can evince is how ORAM 1 time is decreasing when the keys become scattered, while in FSSI the opposite occurs. This particular fact is due to the better mapping of the ORAM algorithm with scattered indexes, because it has smaller buckets to retrieve every time. For FSSI the increasing of the time needed to create the database is due to the size of the tree that with scattered indexes has more internal nodes.

Memory database 128 records

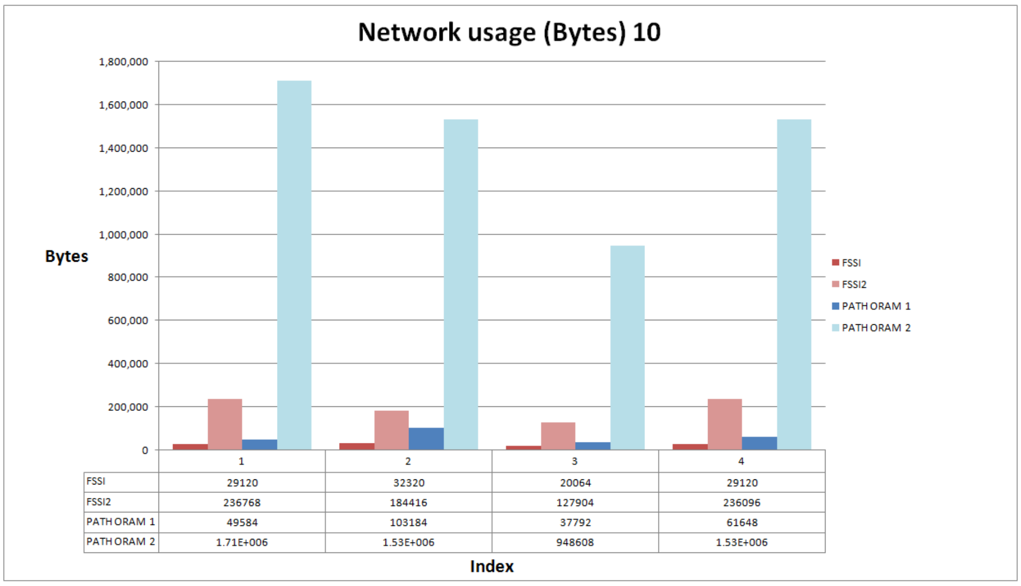


	2	3	4	5
FSSI	524	528	520	520
FSSI2	524	528	520	520
PATH ORAM 1	248	284	276	264
PATH ORAM 2	432	432	432	432

Figure 5.2: This graph shows the amount of memory required by every program after the creation on each index with a dataset of 128 records.

Memory As we can infer from Fig. 5.2, which shows the amount of memory that each program needs to store the data in the database, both FSSI and ORAM 2 are quite similar with respect to memory usage. This is due to the small size of the dataset, because in FSSI we still have few internal nodes, while in ORAM we have small buckets that don't require much memory. The interesting thing is that ORAM 1 needs half the memory of FSSI, since it doesn't need to store a tree for indexing the database. It needs less memory than ORAM 2 since ORAM 2 requires more buckets to store the database, that increases the memory required. Another important result is that with few records is not so relevant the density of the indexes for most of the programs, but ORAM 1 reflects the change of index. This is due to the small size of the tree in FSSI that doesn't require many internal nodes, while for ORAM 1 we have a significant change in the number of buckets. ORAM 2 requires the same time because the number of buckets remains the same and it indexes the records on all 4 attributes.

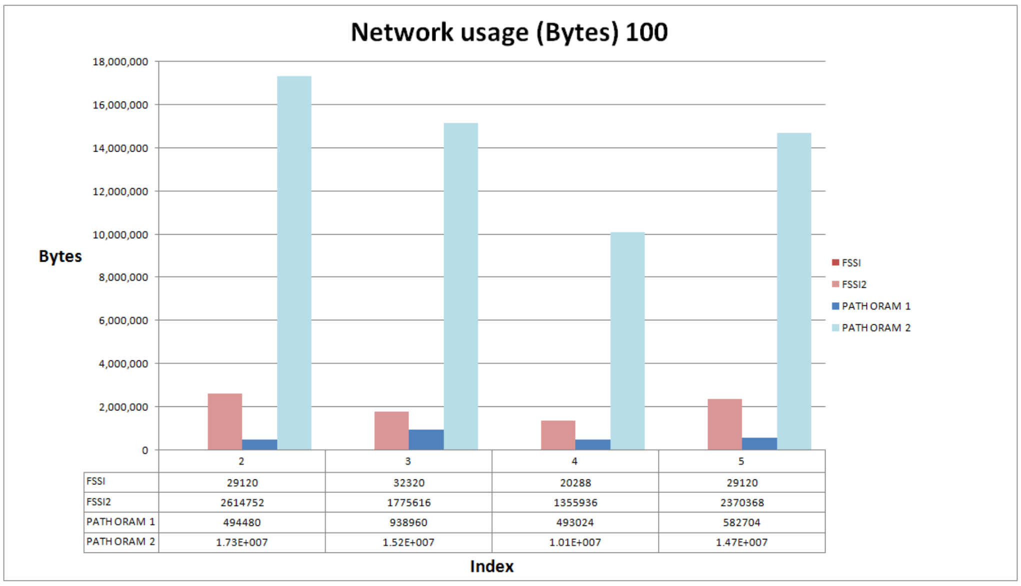
Network usage for 10 queries on 128 records



	1	2	3	4
FSSI	29120	32320	20064	29120
FSSI2	236768	184416	127904	236096
PATH ORAM 1	49584	103184	37792	61648
PATH ORAM 2	1.71E+006	1.53E+006	948608	1.53E+006

Figure 5.3: This graph shows the amount of data sent over the network by each program while executing 10 queries and send back the database to the server with 128 records.

Network usage for 100 queries on 128 records

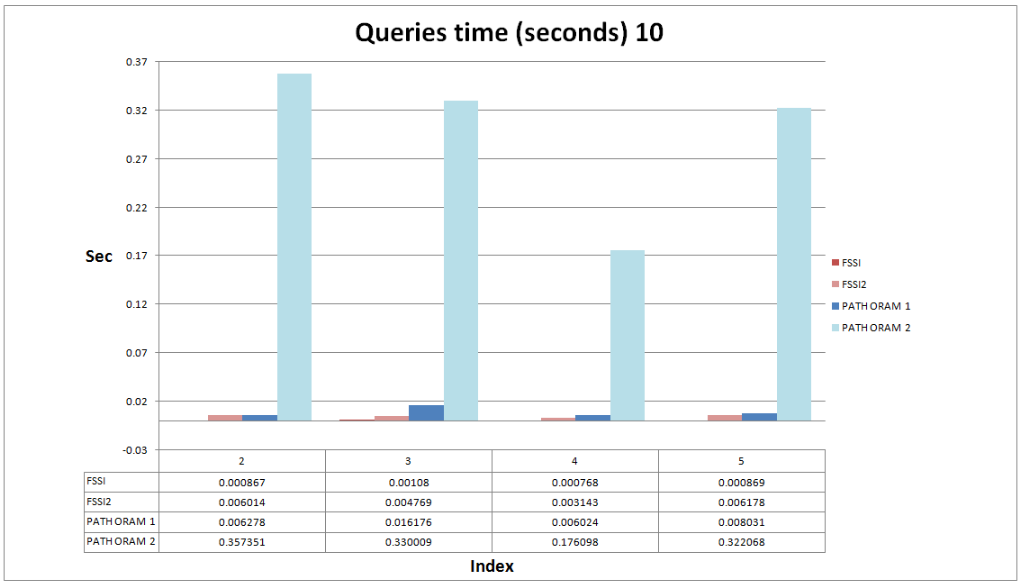


	2	3	4	5
FSSI	29120	32320	20288	29120
FSSI2	2614752	1775616	1355936	2370368
PATH ORAM 1	494480	938960	493024	582704
PATH ORAM 2	1.73E+007	1.52E+007	1.01E+007	1.47E+007

Figure 5.4: This graph shows the amount of data sent over the network by each program when executing 100 queries and sending the database back to the server.

Network From Fig. 5.3 and Fig. 5.4, that display the data passing from the client to the server and back, we can deduce that FSSI1 has a very low network usage, due to its cache system and the small size of the database. ORAM 2 has low performance, because each time it requires a lot of data to pass through the network. This is due to its large number of buckets, which can have many unnecessary blocks that at each time are retrieved and sent back to the server. Also, FSSI2 has quite large data usage due to its fake searches, which retrieve more data than needed and sends it again back to the server without even analysing it. The most important thing is that ORAM 1 has more or less the same data usage of FSSI1 on 10 queries. This is interesting since it shows that the cache system of FSSI 1 on a small number of queries is not so relevant. With 100 queries, it can be seen that the cache has a high influence over the network because it doesn't need to retrieve at each time all the records and the internal nodes.

Query times for 10 queries on 128 records

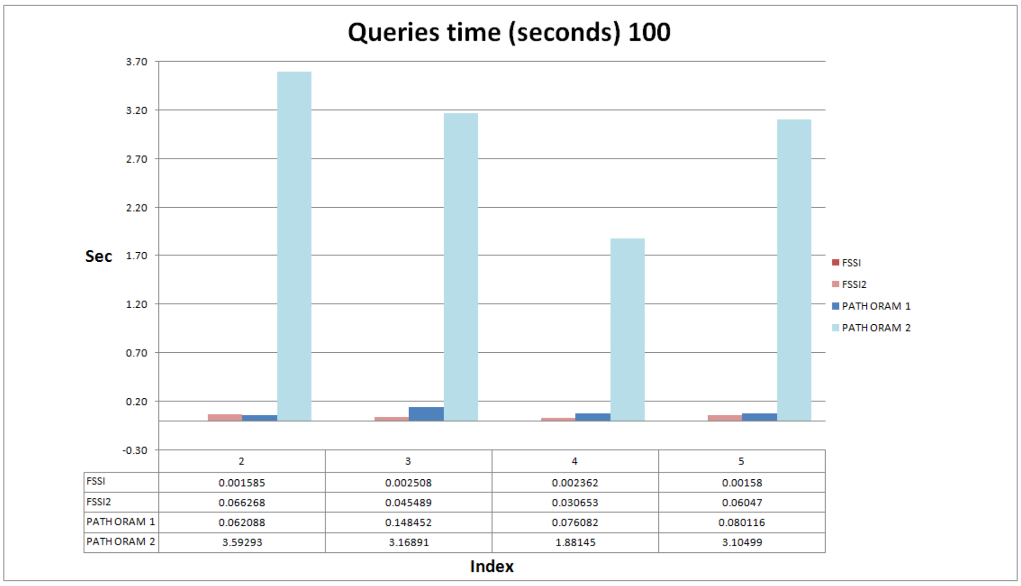


	2	3	4	5
FSSI	0.000867	0.00108	0.000768	0.000869
FSSI2	0.006014	0.004769	0.003143	0.006178
PATH ORAM 1	0.006278	0.016176	0.006024	0.008031
PATH ORAM 2	0.357351	0.330009	0.176098	0.322068

Figure 5.5: This graph displays the total time needed for executing 10 queries and send back the database to the server.

Query total times Graphs in Fig. 5.5 and Fig. 5.6, that show the total time for executing 10 or 100 queries by each program with 128 records, show us that FSSI 1 is very fast compared to all the other algorithms. This is due to its particular cache system

Query times for 100 queries on 128 records



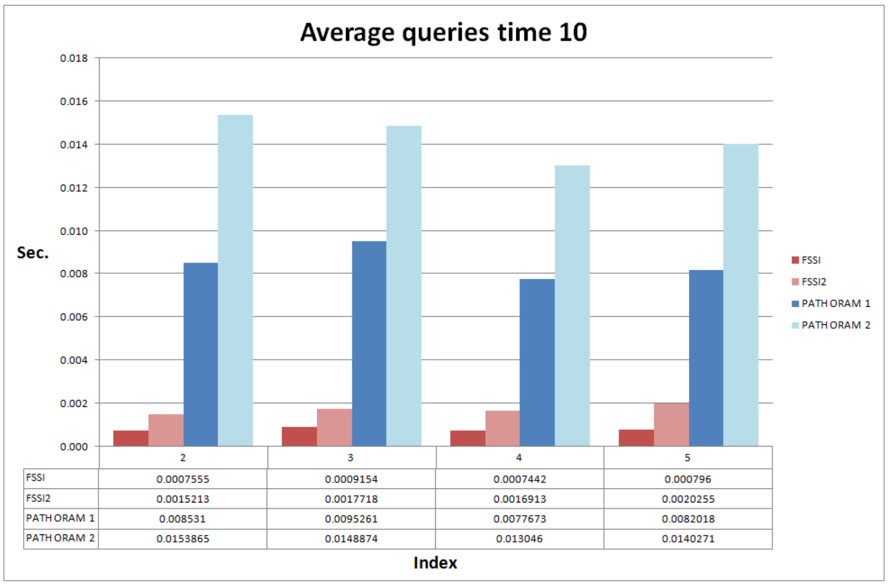
	2	3	4	5
FSSI	0.001585	0.002508	0.002362	0.00158
FSSI2	0.066268	0.045489	0.030653	0.06047
PATH ORAM 1	0.062088	0.148452	0.076082	0.080116
PATH ORAM 2	3.59293	3.16891	1.88145	3.10499

Figure 5.6: This graph displays the total time needed for executing 100 queries and send back the database to the server.

that splits the time needed to retrieve and resend the sectors to the database over all the queries. This allows them to perform the computation faster. About FSSI 2 we can notice that is more or less similar to ORAM 1, because they both have many sectors to retrieve and resend to the server. The interesting thing is that with large ranges FSSI 2 is better, because it has to retrieve less data due to its scattered feature. Another thing that influences this difference is that ORAM 1 with scattered indexes has more buckets to retrieve. ORAM 2 is very bad performing since the indexing requires a lot of time, since it's on many blocks and there is a large number of buckets. We can also note how the key on which databases are indexed affects the execution time, since if we have scattered indexes all the algorithms are faster (except ORAM 1 that has a higher execution time due to the higher number of buckets).

Query average times From graphs in Fig. 5.7 and Fig. 5.8, that compare the average time for executing 1 query, we can deduce that both FSSI are very fast. This is also due to their creation time (already fast) that, spread on several queries, doesn't affect the average time. Another important fact that can be obtained is that FSSI 1 remains the

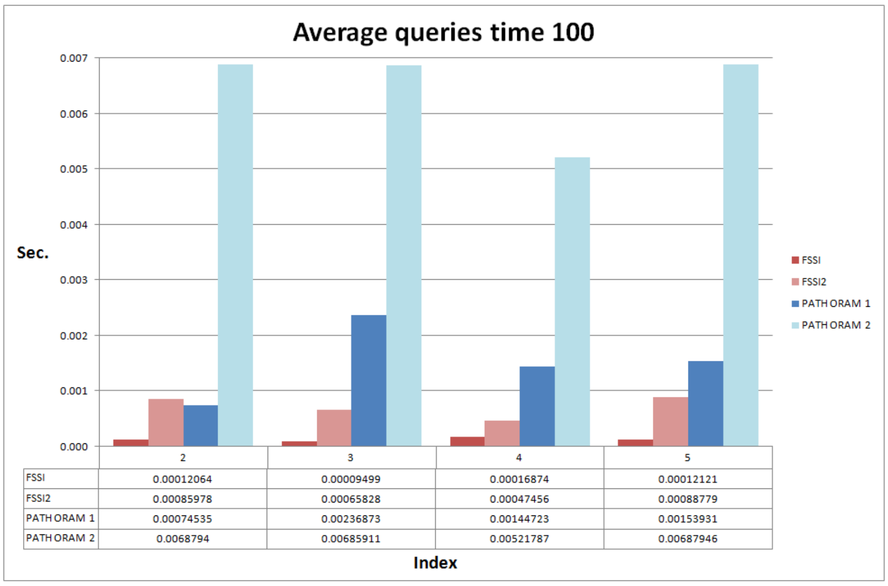
Average query times for 10 queries on 128 records



	2	3	4	5
FSSI	0.0007555	0.0009154	0.0007442	0.000796
FSSI2	0.0015213	0.0017718	0.0016913	0.0020255
PATH ORAM 1	0.008531	0.0095261	0.0077673	0.0082018
PATH ORAM 2	0.0153865	0.0148874	0.013046	0.0140271

Figure 5.7: This graph displays the average time needed by 1 query to be executed, while we are executing 10 queries and sending the database back to the server. It includes also the creation time.

Average query times for 100 queries on 128 records



	2	3	4	5
FSSI	0.00012064	0.00009499	0.00016874	0.00012121
FSSI2	0.00085978	0.00065828	0.00047456	0.00088779
PATH ORAM 1	0.00074535	0.00236873	0.00144723	0.00153931
PATH ORAM 2	0.0068794	0.00685911	0.00521787	0.00687946

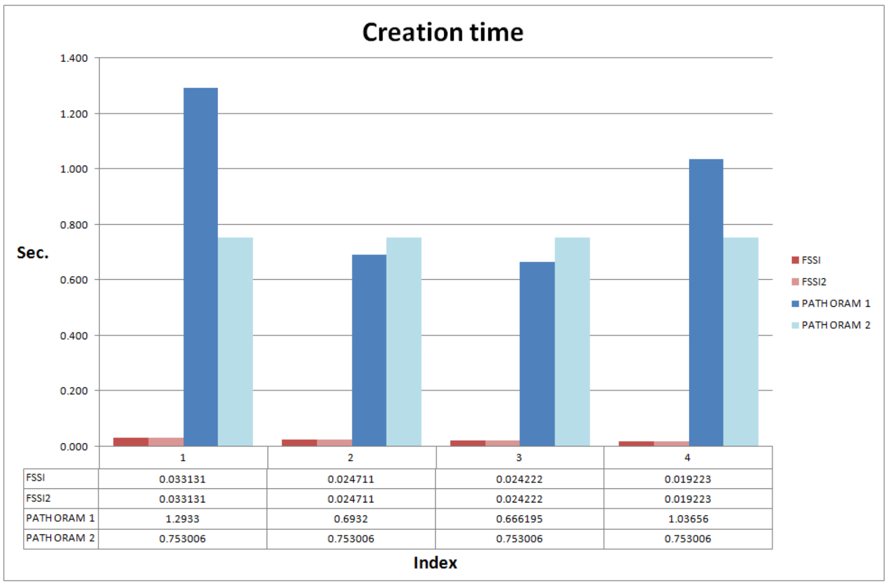
Figure 5.8: This graph displays the average time needed by 1 query to be executed, while we are executing 100 queries and send back the database to the server. It includes also the creation time.

best. For last, both ORAM are very slow since they have a high creation and execution time (as explained before). The main information that we can infer from this graph is that creation time affects a lot the average time, but as we increase the number of queries the creation time becomes less relevant.

5.2 1024 records

The second dataset used is composed by 1024 records.

Creating database 1024 records

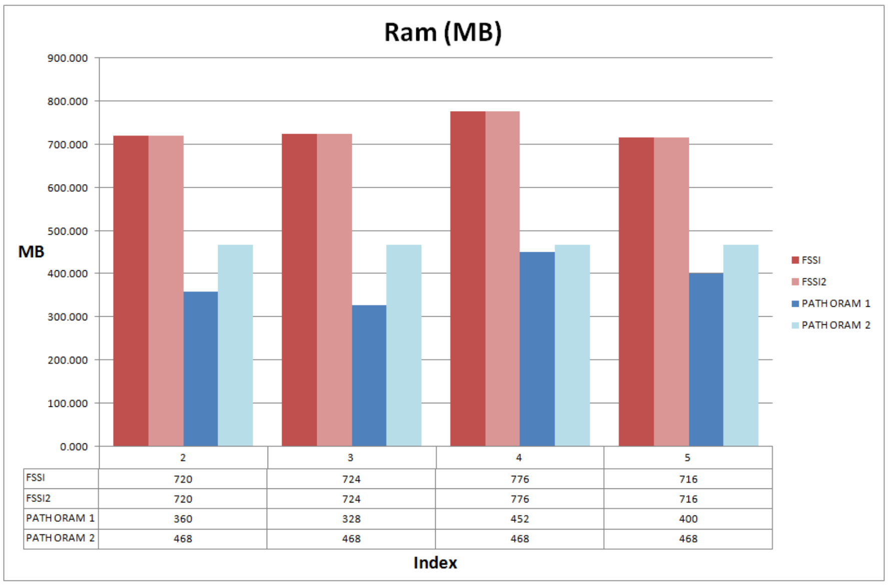


	1	2	3	4
FSSI	0.033131	0.024711	0.024222	0.019223
FSSI2	0.033131	0.024711	0.024222	0.019223
PATH ORAM 1	1.2933	0.6932	0.666195	1.03656
PATH ORAM 2	0.753006	0.753006	0.753006	0.753006

Figure 5.9: This graph shows the time required for creating the index of the database, encrypting and inserting all records into it with a dataset of 1024 lines.

Creation Time In Fig. 5.9 is shown the time that each algorithm needs to create the database (which includes reading all the 1024 records from the file, indexing, encrypting and sending them to the server). We can notice that, as it occurs with 128 records, also in this case both creating the database is really quick for the FSSI algorithms. ORAM require instead much more time in that phase. For both ORAM algorithms the time needed to create the database still depends mainly on the number of buckets. ORAM 1, that for 128 records was faster than ORAM 2, still remains faster than the one on scattered indexes. As one could have expected, ORAM 1 on dense indexes is not so performing in time. This is due to the type of indexing, given that with dense indexes we have to reduce the number of buckets. Since the size of the buckets become larger, the program then, at each record insertion, has to retrieve, decrypt, re-encrypt and send back to the server more data.

Memory database 1024 records



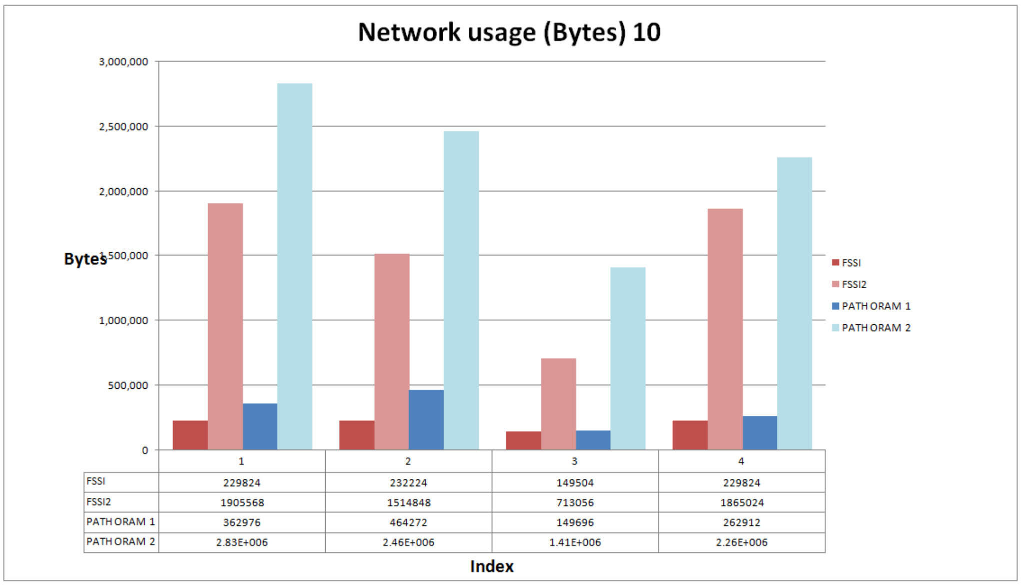
	2	3	4	5
FSSI	720	724	776	716
FSSI2	720	724	776	716
PATH ORAM 1	360	328	452	400
PATH ORAM 2	468	468	468	468

Figure 5.10: This graph shows the amount of memory required by every program after the creation on each index with a dataset of 1024 records.

Memory In Fig. 5.10 we have a graph that illustrates the memory needed to store all the data during the creation of the database (for each algorithm). In this figure we can notice that the trend of the graph, as it happens for the creation time, is similar to the 128 records one. As this outline shows, both FSSI algorithms need more RAM than the ORAM ones and the memory needed increases as the range becomes larger. The reason why this happens is that we have a larger tree in FSSI, and in ORAM we have larger buckets. Only ORAM 2 requires the same amount of memory for all indexes, due to its indexing on all of them. Most importantly, we can see that as the number of records increase, the amount of memory increase in a linear way for ORAM, and more than linearly for FSSI.

Network Graphs in Fig. 5.11, Fig. 5.12 and Fig. 5.13 focus on the network usage by all programs with a 1024 records dataset. In these graphs we can notice that 4th index requires generally about half of the network usage with respect to the previous indexes. In FSSI 2, ORAM 1 and ORAM 2 the network usage increases in a linear way as the number of queries increase, but in FSSI 1 the network usage is constant, due to its cache

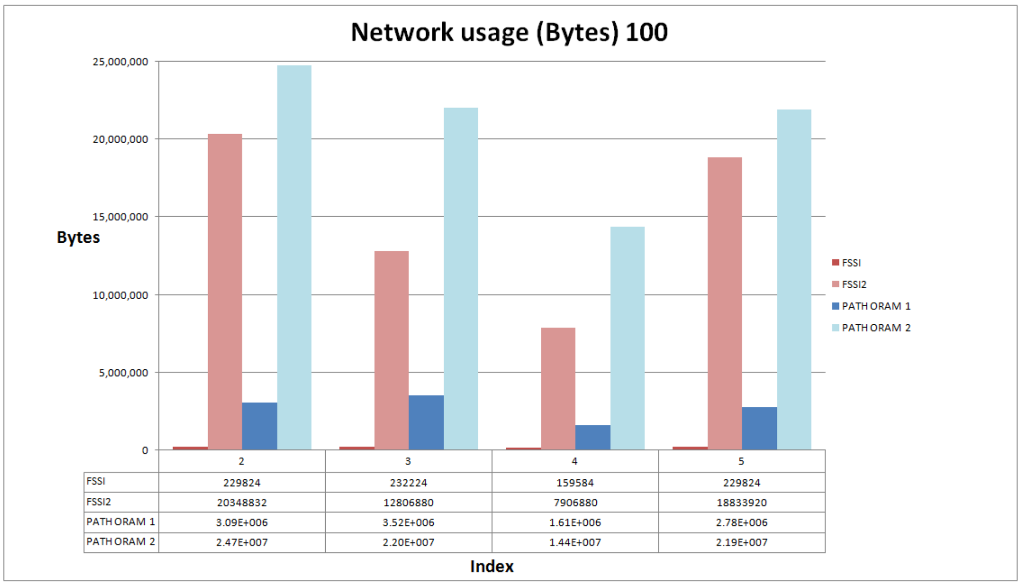
Network usage for 10 queries on 1024 records



	1	2	3	4
FSSI	229824	232224	149504	229824
FSSI2	1905568	1514848	713056	1865024
PATH ORAM 1	362976	464272	149696	262912
PATH ORAM 2	2.83E+006	2.46E+006	1.41E+006	2.26E+006

Figure 5.11: This graph shows the amount of data sent over the network by each program while executing 10 queries and send back the database to the server with 1024 records.

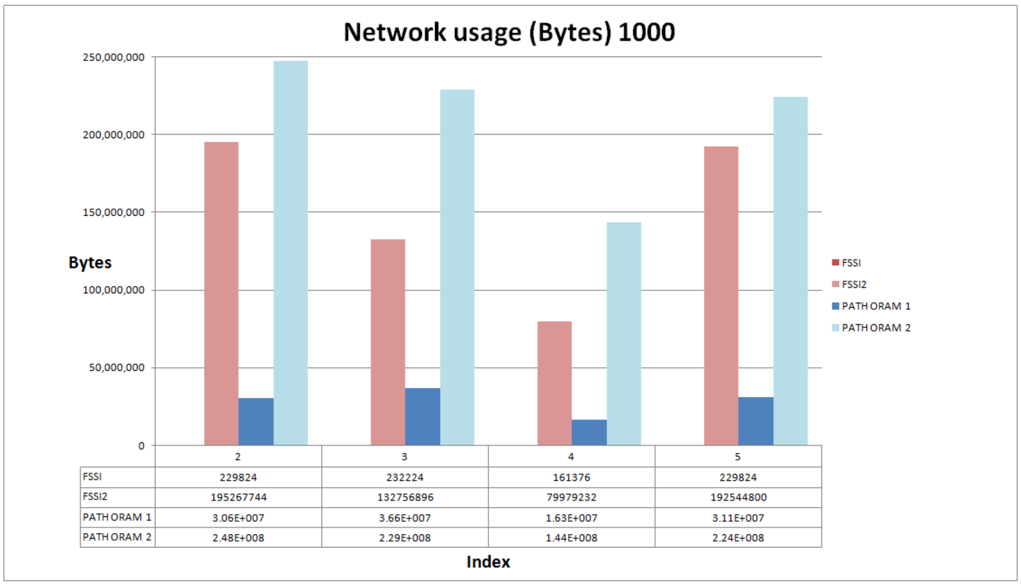
Network usage for 100 queries on 1024 records



	2	3	4	5
FSSI	229824	232224	159584	229824
FSSI2	20348832	12806880	7906880	18833920
PATH ORAM 1	3.09E+006	3.52E+006	1.61E+006	2.78E+006
PATH ORAM 2	2.47E+007	2.20E+007	1.44E+007	2.19E+007

Figure 5.12: This graph shows the amount of data sent over the network by each program while executing 100 queries and send back the database to the server with 1024 records.

Network usage for 1000 queries on 1024 records



	2	3	4	5
FSSI	229824	232224	161376	229824
FSSI2	195267744	132756896	79979232	192544800
PATH ORAM 1	3.06E+007	3.66E+007	1.63E+007	3.11E+007
PATH ORAM 2	2.48E+008	2.29E+008	1.44E+008	2.24E+008

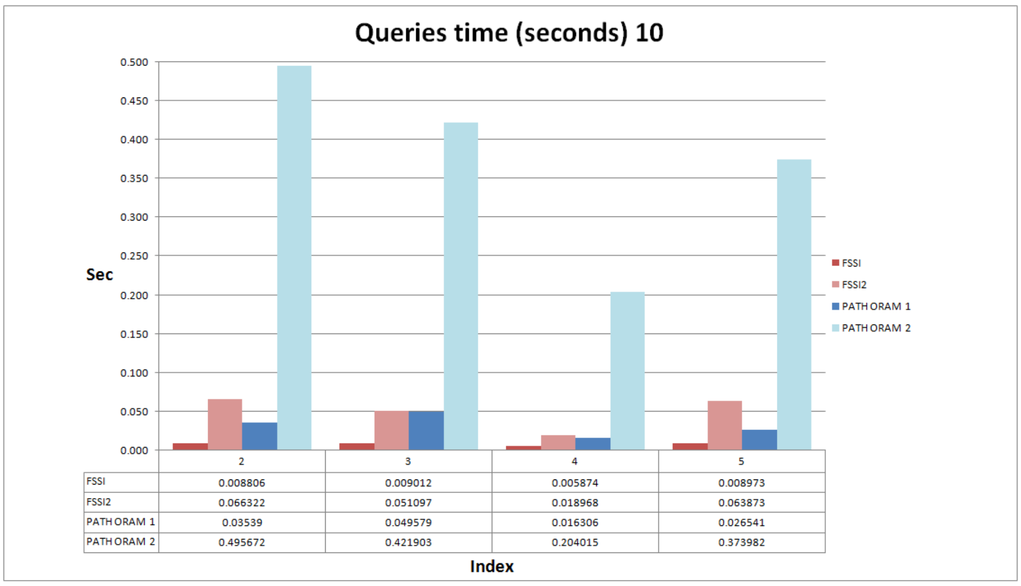
Figure 5.13: This graph shows the amount of data sent over the network by each program while executing 1000 queries and send back the database to the server with 1024 records.

system. As it also occurs for 128 records, ORAM 2 uses a lot of network due to many dummy blocks, but differently from the 128 records dataset, here also FSSI 2 requires a lot of network. This is due to the large number of records that, with scattered indexes, requires a retrieval of almost the whole database at each query.

Query total times Graphs in Fig. 5.14, Fig. 5.15 and Fig. 5.16 show the total time for executing 10 or 100 queries with 1024 records. We can notice that the graphs are following the 128 records outline. There is shown that FSSI 1 is better with respect to all the other algorithms since its cache system allows a faster computation. Focusing on the other algorithms, FSSI 2 and ORAM 1 require more or less the same execution time on scattered indexes, while on dense indexes ORAM 1 is considerably better. ORAM 2 is once more the worst due to its multiple keys indexing that was made to have multi-dimensional queries.

Query average times Graphs in Fig. 5.17, Fig. 5.17 and Fig. 5.19 compare the average time to execute 1 query. It can be drawn that with a small number of queries

Query times for 10 queries on 1024 records

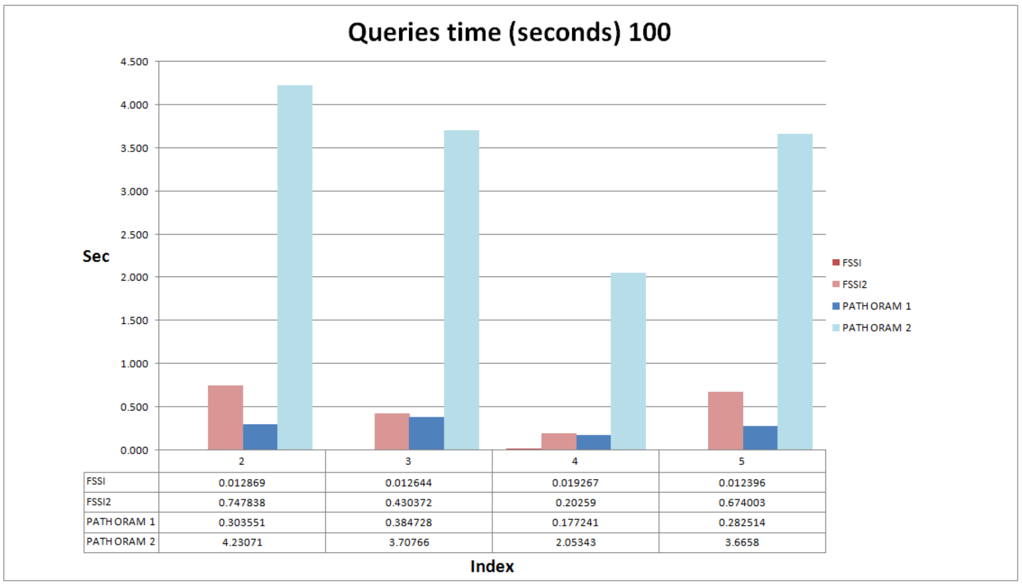


	2	3	4	5
FSSI	0.008806	0.009012	0.005874	0.008973
FSSI2	0.066322	0.051097	0.018968	0.063873
PATH ORAM 1	0.03539	0.049579	0.016306	0.026541
PATH ORAM 2	0.495672	0.421903	0.204015	0.373982

Figure 5.14: This graph displays the total time needed for executing 10 queries and sending back the database to the server.

both FSSI algorithms are better. On scattered indexes all the four algorithms have more or less the same behaviour as with 128 records. On dense indexes ORAM 1 increase its average time drastically, due to its high creation time. Analysing more queries we have that ORAM 1 become faster since the creation time is amortised, until it becomes more performing than FSSI 2 on dense indexes. ORAM 2 is still quite slow.

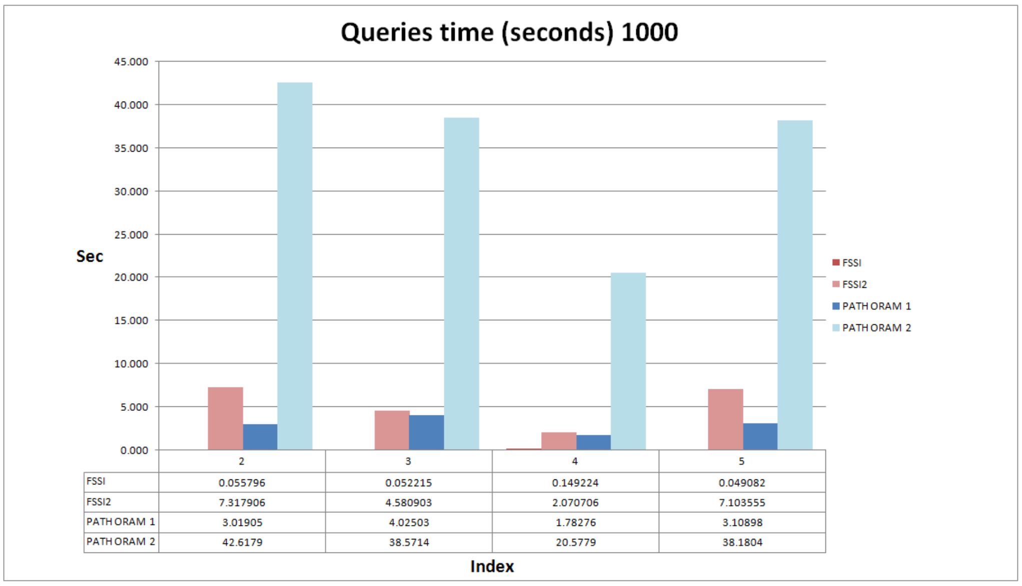
Query times for 100 queries on 1024 records



	2	3	4	5
FSSI	0.012869	0.012644	0.019267	0.012396
FSSI2	0.747838	0.430372	0.20259	0.674003
PATH ORAM 1	0.303551	0.384728	0.177241	0.282514
PATH ORAM 2	4.23071	3.70766	2.05343	3.6658

Figure 5.15: This graph displays the total time needed for executing 100 queries and sending back the database to the server.

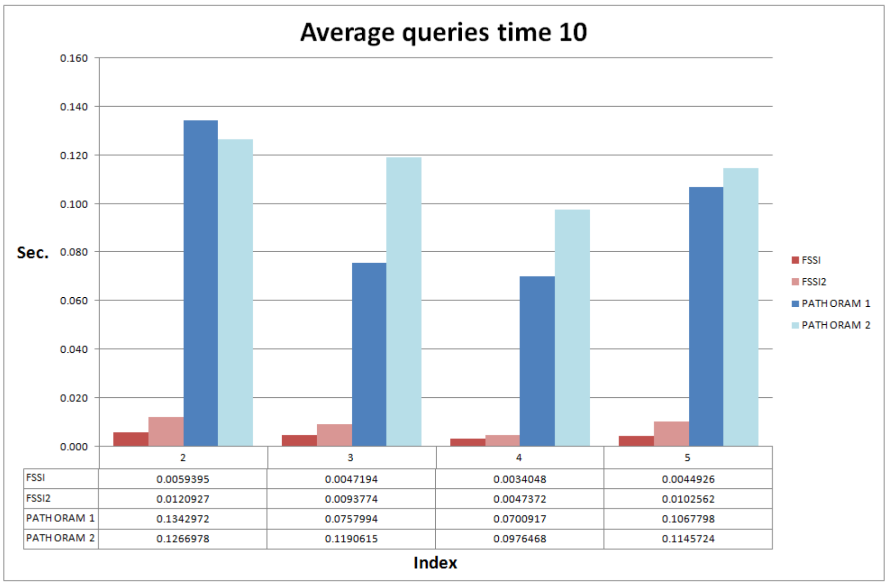
Query times for 1000 queries on 1024 records



	2	3	4	5
FSSI	0.055796	0.052215	0.149224	0.049082
FSSI2	7.317906	4.580903	2.070706	7.103555
PATH ORAM 1	3.01905	4.02503	1.78276	3.10898
PATH ORAM 2	42.6179	38.5714	20.5779	38.1804

Figure 5.16: This graph displays the total time needed for executing 1000 queries and sending back the database to the server.

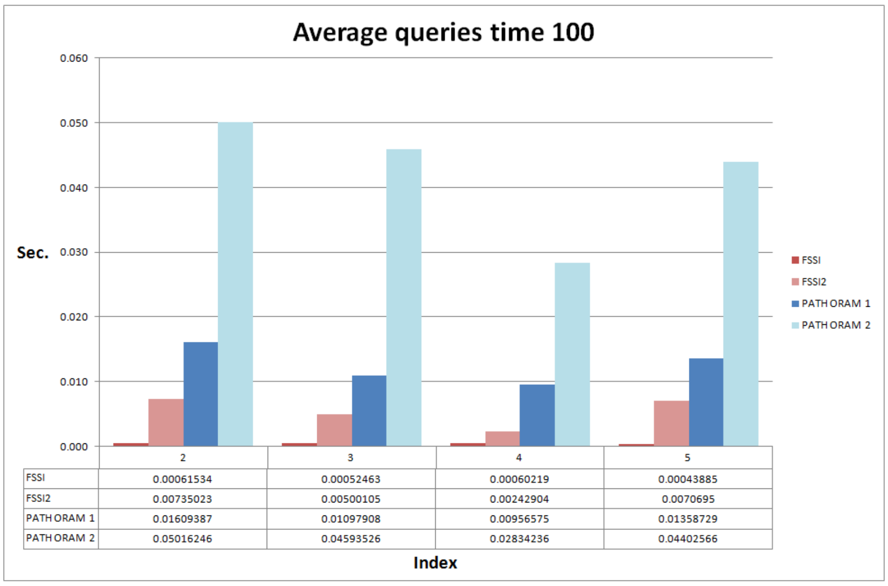
Average query times for 10 queries on 1024 records



	2	3	4	5
🟥 FSSI	0.0059395	0.0047194	0.0034048	0.0044926
🟥 FSSI2	0.0120927	0.0093774	0.0047372	0.0102562
🟦 PATH ORAM 1	0.1342972	0.0757994	0.0700917	0.1067798
🟦 PATH ORAM 2	0.1266978	0.1190615	0.0976468	0.1145724

Figure 5.17: This graph displays the average time needed to execute 1 query, while we are executing 10 queries and sending the database back to the server. It also includes the creation time.

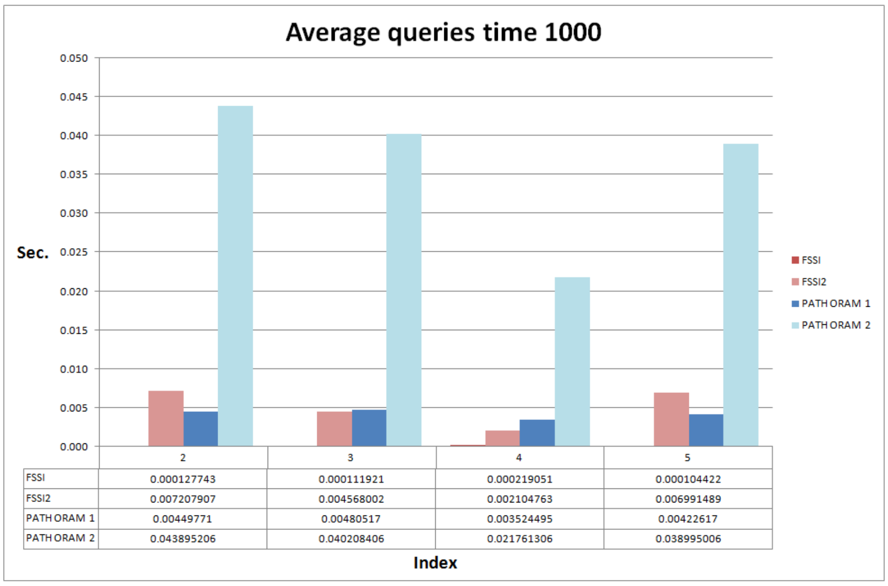
Average query times for 100 queries on 1024 records



	2	3	4	5
🟥 FSSI	0.00061534	0.00052463	0.00060219	0.00043885
🟥 FSSI2	0.00735023	0.00500105	0.00242904	0.0070695
🟦 PATH ORAM 1	0.01609387	0.01097908	0.00956575	0.01358729
🟦 PATH ORAM 2	0.05016246	0.04593526	0.02834236	0.04402566

Figure 5.18: This graph displays the average time needed to execute 1 query, while we are executing 100 queries and sending the database back to the server. It also includes the creation time.

Average query times for 1000 queries on 1024 records



	2	3	4	5
FSSI	0.000127743	0.000111921	0.000219051	0.000104422
FSSI2	0.007207907	0.004568002	0.002104763	0.006991489
PATH ORAM 1	0.00449771	0.00480517	0.003524495	0.00422617
PATH ORAM 2	0.043895206	0.040208406	0.021761306	0.038995006

Figure 5.19: This graph displays the average time needed to execute 1 query, while we are executing 1000 queries and sending the database back to the server. It also includes the creation time.

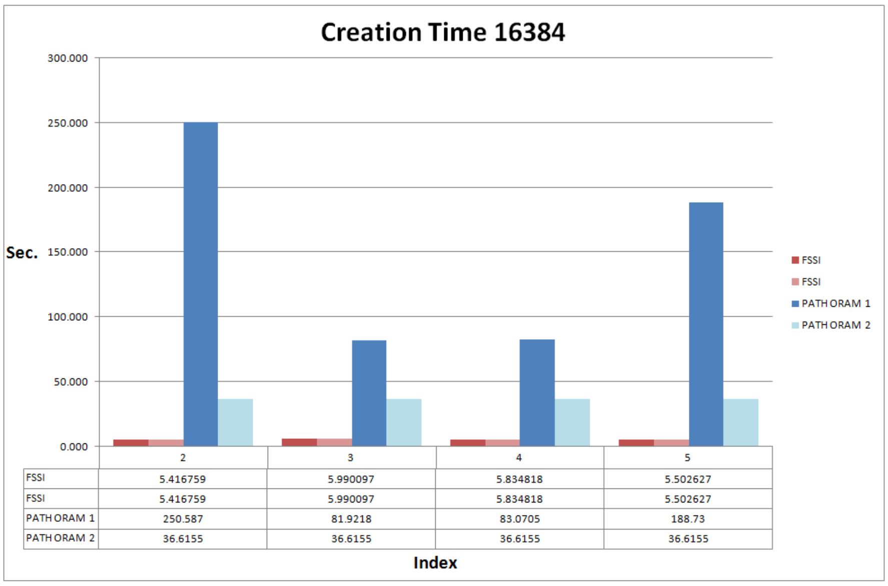
5.3 16384 records

The third dataset used is composed by 16384 records.

Creation Time In Fig. 5.20 is shown the time that each program needs to create the database. We can notice that on the 2nd and 5th indexes it follows the outline of the previous datasets, while for the 2nd and the 3rd indexes ORAM 1 becomes the worst algorithm. The interesting fact is that ORAM 2 becomes better on large indexes, but still worse than FSSI. This particular fact is due to the number of buckets that here are higher than before, which allows the algorithm to reduce the size of the buckets and to have a better distribution of the blocks. The reason why FSSI is the best is still the same.

Memory Graph in Fig. 5.21 shows the memory needed to store the database after the creation. It can be seen that both ORAM algorithm need one third of the memory needed by FSSI. This is due to the good distribution of the records into the buckets in

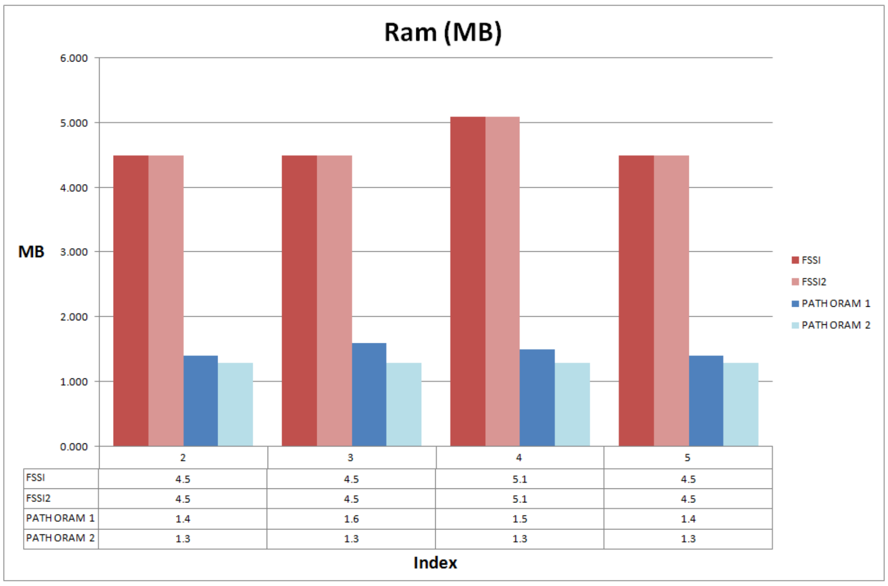
Creating database 16384 records



	2	3	4	5
FSSI	5.416759	5.990097	5.834818	5.502627
FSSI	5.416759	5.990097	5.834818	5.502627
PATH ORAM 1	250.587	81.9218	83.0705	188.73
PATH ORAM 2	36.6155	36.6155	36.6155	36.6155

Figure 5.20: This graph shows the time required for creating the index of the database, encrypting and inserting all records into it with a dataset of 16384 lines.

Memory database 16384 records

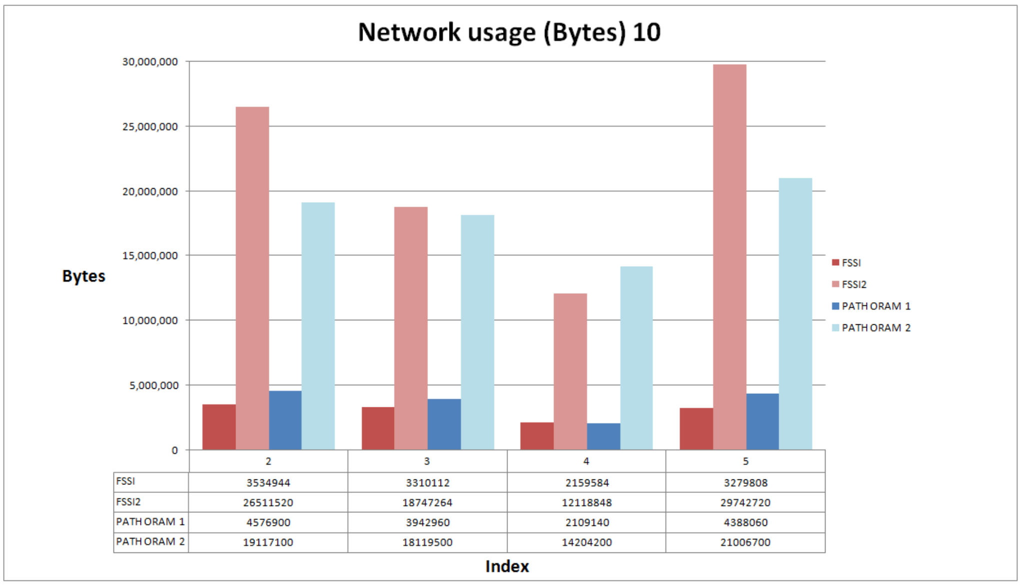


	2	3	4	5
FSSI	4.5	4.5	5.1	4.5
FSSI2	4.5	4.5	5.1	4.5
PATH ORAM 1	1.4	1.6	1.5	1.4
PATH ORAM 2	1.3	1.3	1.3	1.3

Figure 5.21: This graph shows the amount of memory required by every program after the creation on each index with a dataset of 16384 records.

ORAM and to the size of the data structures in FSSI. Another important thing that we can notice is that ORAM 2 becomes the best. This because we have a number of buckets that compared with ORAM 1 on many records becomes better. Also it helps that ORAM 2 decreases the size of the buckets by reducing the number of dummy blocks.

Network usage for 10 queries on 16384 records



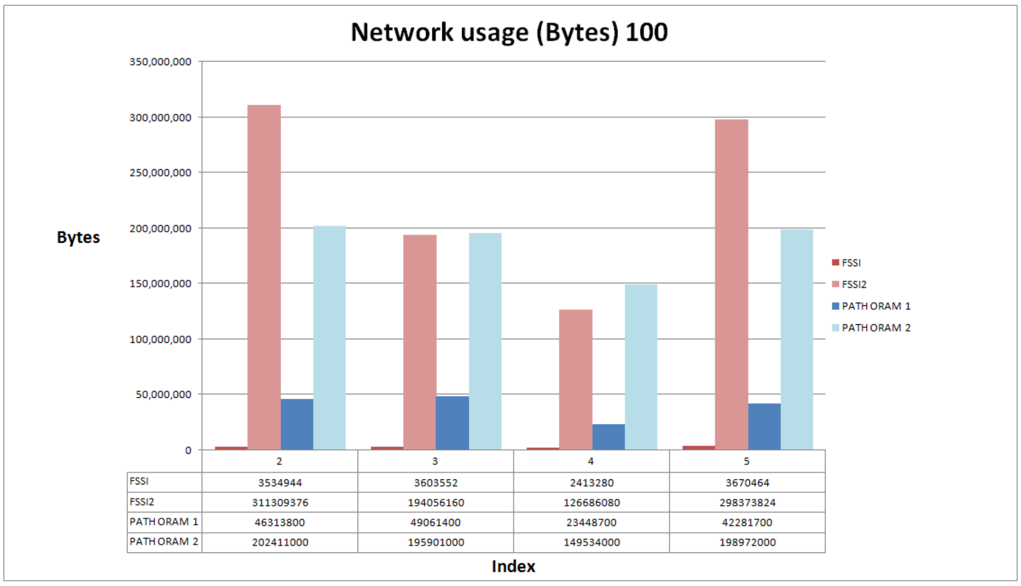
	2	3	4	5
FSSI	3534944	3310112	2159584	3279808
FSSI2	26511520	18747264	12118848	29742720
PATH ORAM 1	4576900	3942960	2109140	4388060
PATH ORAM 2	19117100	18119500	14204200	21006700

Figure 5.22: This graph shows the amount of data sent over the network by each program while executing 10 queries and sending back the database to the server with 16384 records.

Network In Graphs in Fig. 5.22, Fig. 5.22 and Fig5.24 we have the network used to execute a certain number of queries. FSSI 2 and ORAM 2 use the same amount of network on scattered indexes, while on dense indexes FSSI 2 becomes the worst. Focusing on the other programs we can notice how ORAM 1 improves its data usage as the number of records increase, till becoming as performant as FSSI 1. This shows us that the bucket scheme tends to become better on large indexes.

Query total times In Fig5.25, Fig. 5.26 and Fig. 5.27 is shown the total execution time for 10, 100 and 1000 queries on every program. It can be seen that as the number of records increase FSSI 2 time becomes higher. Since ORAM 2 time doesn't increas as much, it becomes better performing on a high number of records than FSSI 2. For

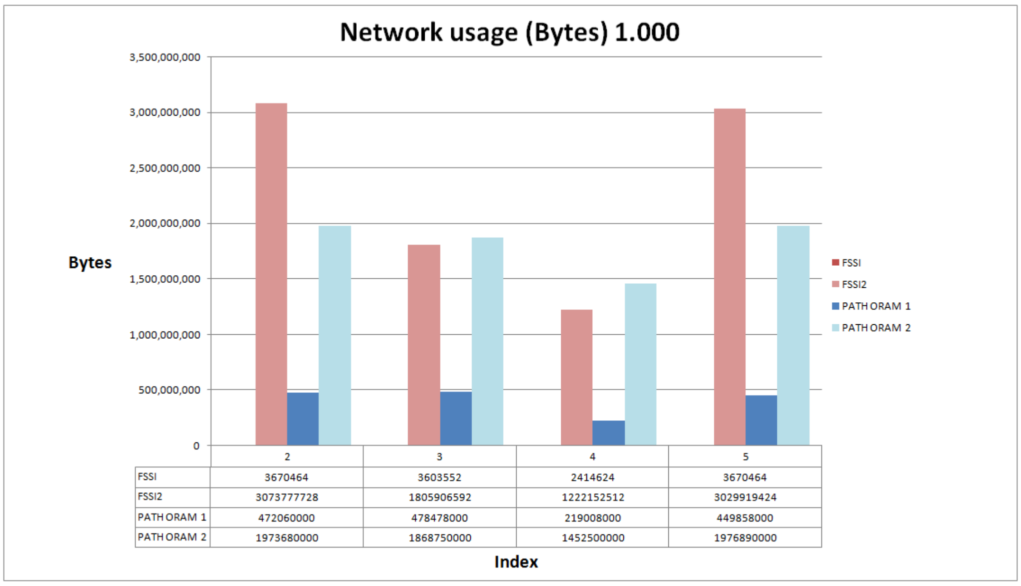
Network usage for 100 queries on 16384 records



	2	3	4	5
FSSI	3534944	3603552	2413280	3670464
FSSI2	311309376	194056160	126686080	298373824
PATH ORAM 1	46313800	49061400	23448700	42281700
PATH ORAM 2	202411000	195901000	149534000	198972000

Figure 5.23: This graph shows the amount of data sent over the network by each program while executing 100 queries and sending back the database to the server with 16384 records.

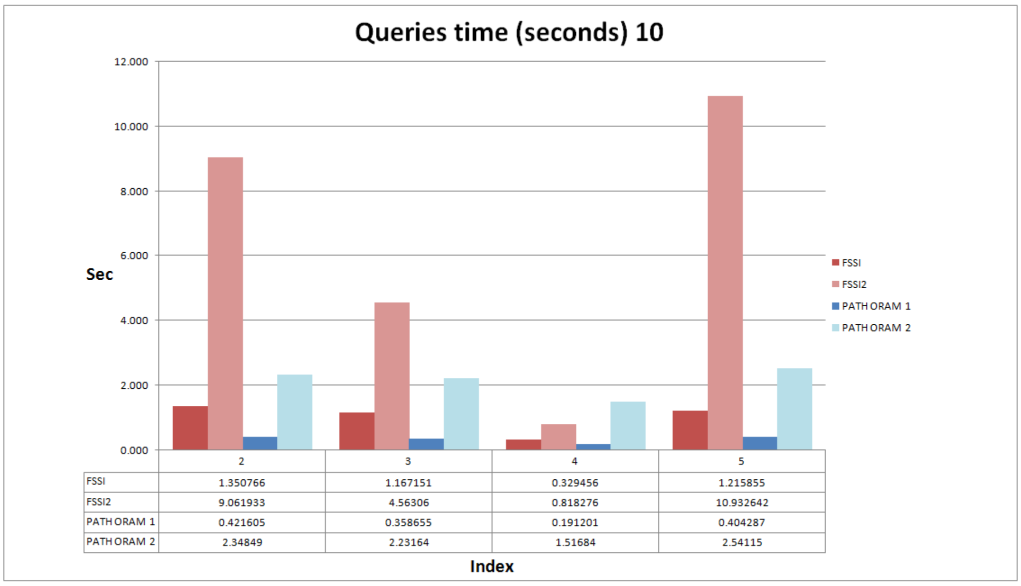
Network usage for 1000 queries on 16384 records



	2	3	4	5
FSSI	3670464	3603552	2414624	3670464
FSSI2	3073777728	1805906592	1222152512	3029919424
PATH ORAM 1	472060000	478478000	219008000	449858000
PATH ORAM 2	1973680000	1868750000	1452500000	1976890000

Figure 5.24: This graph shows the amount of data sent over the network by each program while executing 1000 queries and sending back the database to the server with 16384 records.

Query times for 10 queries on 16384 records



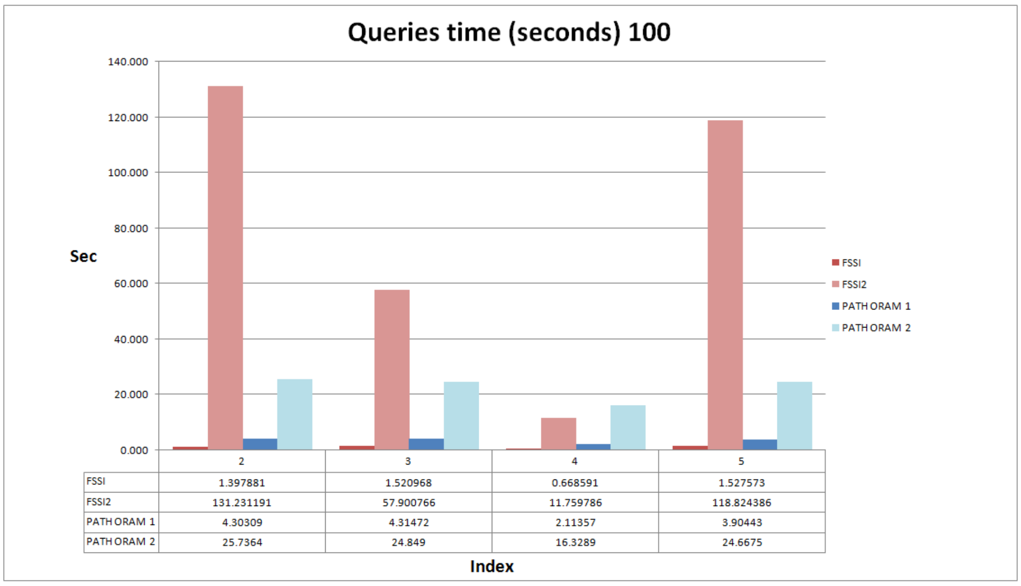
	2	3	4	5
FSSI	1.350766	1.167151	0.329456	1.215855
FSSI2	9.061933	4.56306	0.818276	10.932642
PATH ORAM 1	0.421605	0.358655	0.191201	0.404287
PATH ORAM 2	2.34849	2.23164	1.51684	2.54115

Figure 5.25: This graph displays the total time needed for executing 10 queries and send back the database to the server.

a small number of queries the more performing is ORAM 1, while for a large number of queries the cache of FSSI 1 speeds the program up and reduces the execution time, making it the best performing. Another important fact is that on scattered indexes FSSI 2 time is more or less 10 times better than on dense indexes. Thus, the best indexing strategy for FSSI 2 is scattered indexes.

Query average times Graphs in Fig. 5.28, Fig. 5.29 and Fig. 5.30 show the average queries execution time for all programs. Comparing 128 and 1024 ones we can see that as the number of records increases ORAM 2 tends to become better on a small number of queries, while ORAM 1 execution time becomes higher due to its high creation time. As the number of queries increases FSSI 2 has a lot of data to exchange and therefore its execution time becomes higher. ORAM 1 and ORAM 2 can amortise the creation time until it becomes better than FSSI 2. We can also notice how creation time has a high influence on ORAM 1, that for 1000 queries becomes the second best algorithm. FSSI 1 still is the best.

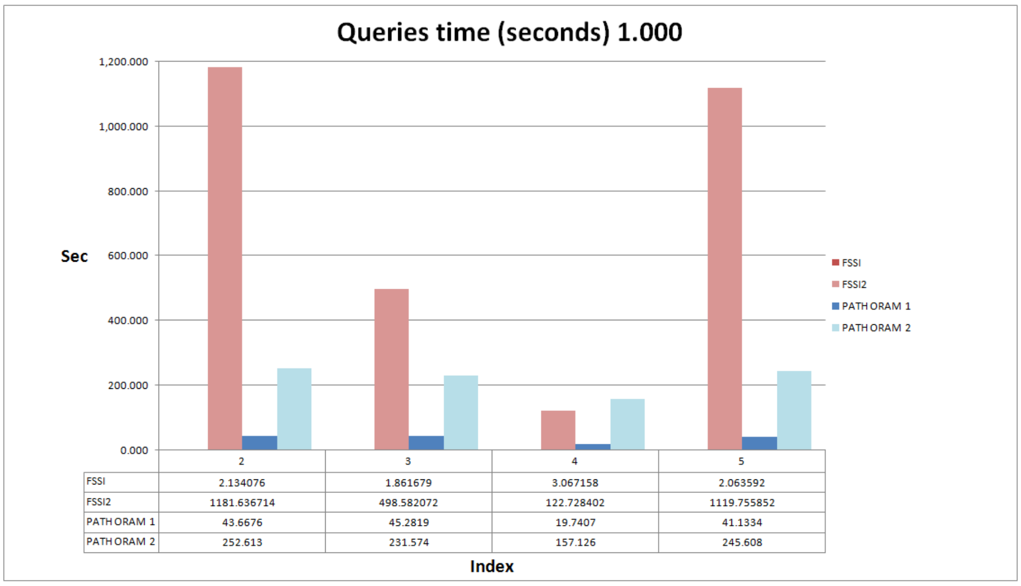
Query times for 100 queries on 16384 records



	2	3	4	5
FSSI	1.397881	1.520968	0.668591	1.527573
FSSI2	131.231191	57.900766	11.759786	118.824386
PATH ORAM 1	4.30309	4.31472	2.11357	3.90443
PATH ORAM 2	25.7364	24.849	16.3289	24.6675

Figure 5.26: This graph displays the total time needed for executing 100 queries and send back the database to the server.

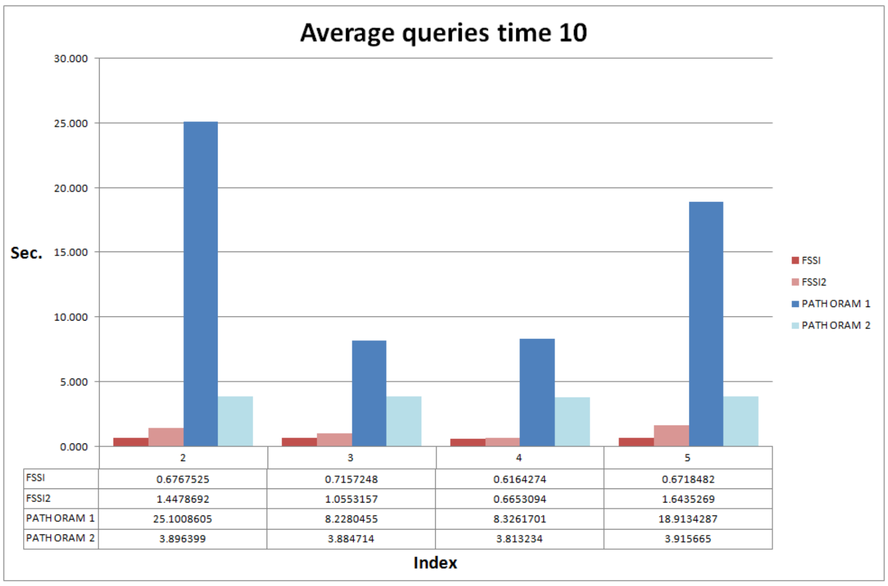
Query times for 1000 queries on 16384 records



	2	3	4	5
FSSI	2.134076	1.861679	3.067158	2.063592
FSSI2	1181.636714	498.582072	122.728402	1119.755852
PATH ORAM 1	43.6676	45.2819	19.7407	41.1334
PATH ORAM 2	252.613	231.574	157.126	245.608

Figure 5.27: This graph displays the total time needed for executing 1000 queries and send back the database to the server.

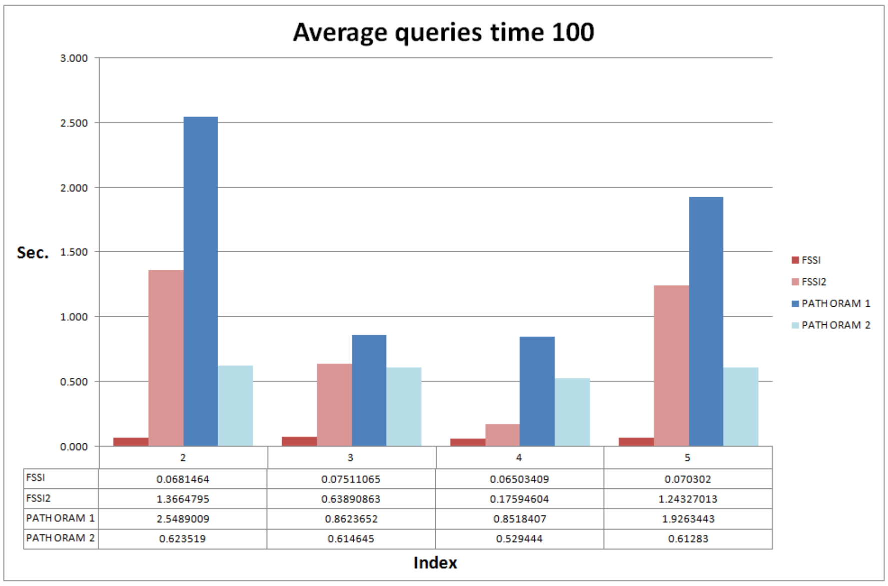
Average query times for 10 queries on 16384 records



	2	3	4	5
FSSI	0.6767525	0.7157248	0.6164274	0.6718482
FSSI2	1.4478692	1.0553157	0.6653094	1.6435269
PATH ORAM 1	25.1008605	8.2280455	8.3261701	18.9134287
PATH ORAM 2	3.896399	3.884714	3.813234	3.915665

Figure 5.28: This graph displays the average time needed to execute 1 query, while we are executing 10 queries and sending the database back to the server. It includes also the creation time.

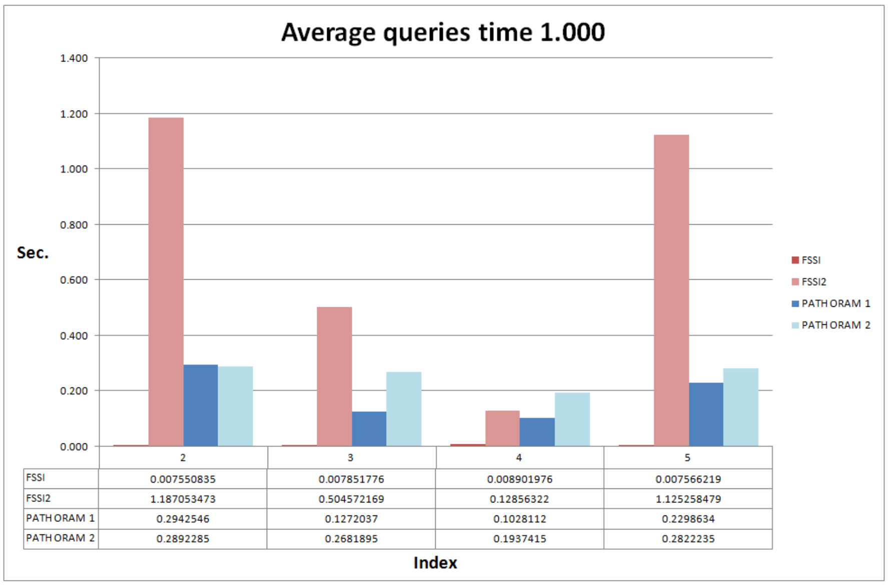
Average query times for 100 queries on 16384 records



	2	3	4	5
FSSI	0.0681464	0.07511065	0.06503409	0.070302
FSSI2	1.3664795	0.63890863	0.17594604	1.24327013
PATH ORAM 1	2.5489009	0.8623652	0.8518407	1.9263443
PATH ORAM 2	0.623519	0.614645	0.529444	0.61283

Figure 5.29: This graph displays the average time needed to execute 1 query, while we are executing 100 queries and sending the database back to the server. It includes also the creation time.

Average query times for 1000 queries on 16384 records



	2	3	4	5
FSSI	0.007550835	0.007851776	0.008901976	0.007566219
FSSI2	1.187053473	0.504572169	0.12856322	1.125258479
PATH ORAM 1	0.2942546	0.1272037	0.1028112	0.2298634
PATH ORAM 2	0.2892285	0.2681895	0.1937415	0.2822235

Figure 5.30: This graph displays the average time needed to execute 1 query, while we are executing 1000 queries and sending the database back to the server. It includes also the creation time.

Chapter 6

Conclusions

The conclusions that can be drawn from our testing are:

Creation The creation of the database is very important since it put the base for the execution of the queries. Since this occurs normally on the beginning and it won't change during the execution of the queries, this is important only if we need to change the database many times. From the testing we can evince that both FSSI are very fast due to their local creation and the simplicity of their algorithms, while both ORAM are slower because at each insertion they need to re-encrypt each bucket. We can notice that as the number of records increase ORAM 2 tends to become the worst.

Memory For the amount of memory the best algorithm is ORAM, since it requires in every case less memory than FSSI. Both oram are very good since they needs lower storage, but with a large dataset ORAM 2 tends to become better. The only difference between them is due to the number of buckets. this is due to the number of buckets used. The "failure" of FSSI in this field is due to the data structure used, since they are larger than the ones used in both ORAM algorithm. Also we can notice that the size of FSSI grows in more than linear compared to ORAM.

Network As we are working with range queries, we can notice that the result in terms of network usage is highly influenced by the ratio between the range of the keys and the number of records. If we have a limited range then at each query we will receive much data, otherwise in case of a large range we will receive less data due to the distribution. This means that every algorithm will reduce its network usage on scattered indexes. From our test we can retrieve that FSSI 1 has little usage due to its cache, while FSSI 2 and oram are not very applicable because with many queries they need a lot of data to go through the network, while ORAM 2 remains applicable. This is due to the number of buckets in ORAM 2 and the serialization of every sector retrieved in FSSI 2.

Queries time The most important comparison is related to the time of execution for the queries. FSSI 1 is very fast with a large number of queries due to its cache system, the main problem remains that in case of many queries sometimes its need to save the database locally on client's ram, that can be very bad if we want to have large datasets. FSSI 2 solves this problems but increases irremediably the execution time, for this reason it is not very good to be used practically, unless we want to do only few queries. ORAM 1 differently become very good on a large number of queries because they amortize the creation time. We have also noticed that ORAM 1 creation and queries execution time are related inversely proportional, so with few buckets the creation time increase while the execution time decrease and so on. Also from average time we can notice that the creation time has a high influence on queries execution time in both FSSI while in both ORAM is not relevant. We can also notice that for a large number of queries, the average time and the execution time (divided by the number of queries) become close to each other. This is important since if we want to extrapolate the average time for a large number of queries, we can use an extrapolation over the execution time.

Concluding we can assert that ORAM 2 is not applicable in practice, while both FSSI and ORAM 1 are applicable, depending on what the client needs. If he needs a large dataset, ORAM 1 is preferable if he needs to execute many queries, while if he has few queries we can suggest FSSI 2 if he needs more obfuscation and security, while if he needs less security but more performances we suggest FSSI 1.

Chapter 7

Ringraziamenti

Desidero ringraziare il prof. Ozalp Babaoglu, il prof Erkay Savas per la loro disponibilità nell'aiutarmi nell'ideazione e nella stesura della tesi.
Mio babbo, mia mamma e mio fratello senza i quali non sarei riuscito a portare avanti questa tesi.
Dani per tutte le correzioni che mi ha fatto e soprattutto per certe S.
Monica che mi ha dato una grossa mano nel crederci.
Tutti i miei amici della pallavolo che mi hanno supportato (e sopportato) durante la stesura e che mi hanno aiutato a crederci.
I miei zii e cugini che mi sono sempre stati vicino, i miei nonni e tutti quelli che hanno contribuito anche solo in minima parte ad aiutarmi in questa tesi e durante tutta l'università (vi evito la lista di nomi se no non finirei più e mi scorderei sicuramente qualcuno).

Bibliography

[1] Krebs On Security [2015] *"Online Cheating Site AshleyMadison Hacked"*. Available on http://krebsonsecurity.com.

[2] James Cook [December 16, 2014] *"Sony Hackers Have Over 100 Terabytes Of Documents. Only Released 200 Gigabytes So Far"* Available on http://uk.businessinsider.com/.

[3] Amittai Aviram [2010] *"B+Tree Implementation"* Available on http://www.amittai.com/prose/bpt.c.

[4] Bernagozzi Stefano [2015] *"Source Code Of The Project"* Available on https://github.com/ste93/thesis_code.

[5] Microsoft SQL server [2008] *Detailed Description.* https://technet.microsoft.com/en-us/library/cc278098(v=sql.100).aspx.

[6] Oracle encrypted database *Description of the database.* https://docs.oracle.com/cd /B19306_01/network.102/b14268/asotrans.htm#BABJJAIG.

[7] IBM DB2 encryption. https://www-01.ibm.com/support/knowledgecenter/SSEPGG _10.5.0/com.ibm.db2.luw.admin.sec.doc/doc/c0061758.html.

[8] Mattsson, U., Protegrity Corporation, 2005. *Combined hardware and software based encryption of databases.* U.S. Patent 6,963,980.

[9] Encrypted File System Windows http://windows.microsoft.com/en-us/windows/what-is-encrypting-file-system#1TC=windows-7

[10] Yang, Z., Zhong, S. and Wright, R.N., 2006. *Privacy-preserving queries on encrypted data.* In Computer SecurityESORICS 2006 (pp. 479-495). Springer Berlin Heidelberg. Vancouver

[11] zsoyoglu, G., Singer, D.A. and Chung, S.S., 2003, August. *Anti-Tamper Databases: Querying Encrypted Databases.* In DBSec (pp. 133-146).

[12] Gentry, C., 2009. *A fully homomorphic encryption scheme* (Doctoral dissertation, Stanford University).

[13] Smart, N.P. and Vercauteren, F., 2010. *Fully homomorphic encryption with relatively small key and ciphertext sizes.* In Public Key CryptographyPKC 2010 (pp. 420-443). Springer Berlin Heidelberg.

[14] Song, D.X., Wagner, D. and Perrig, A., 2000. *Practical techniques for searches on encrypted data.* In Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on (pp. 44-55). IEEE.

[15] Openssl EVP encryption documentation. https://wiki.openssl.org/index.php/EVP _Symmetric_Encryption_and_Decryption.

[16] Stefanov, E., Shi, E., & Song, 2011. *Towards practical oblivious RAM.* arXiv preprint arXiv:1106.3652.

[17] Stefanov, Emil, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. *"Path ORAM: an extremely simple oblivious RAM protocol."* In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 299-310. ACM, 2013.

[18] Vimercati, S. D. C., Foresti, S., Paraboschi, S., Pelosi, G., & Samarati, P. (2011, June). *Efficient and private access to outsourced data.* In Distributed Computing Systems (ICDCS), 2011 31st International Conference on (pp. 710-719). IEEE.