Searching in encrypted data

Richard Brinkman

This research is conducted within the Secure Meta Data project supported by Philips Research and the University of Twente.

University of Twente
The Netherlands

Group of Distributed and Embedded Systems,
P.O. Box 217, 7500 AE Enschede, The Netherlands.

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Keywords: encryption, searching, database encryption.
Printed by Ipskamp PrintPartners, Enschede, The Netherlands.
ISSN 1381-3617 (CTIT Ph.D.-thesis Series No. 07-98)
ISBN 978-90-365-2488-9
IPA 2007-09

SEARCHING IN ENCRYPTED DATA

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof. dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Friday, June 1, 2007 at 13:15

by

Richard Brinkman

born on 27 October 1978,

in Ede, The Netherlands

This dissertation is approved by:

Prof. Dr.	W.	Jonker	(promotor)	and
Prof. Dr.	P. H.	Hartel	(promotor)	

Abstract

This PhD thesis addresses the problem of securing data stored on an untrusted server. There are situations in which personal data or other sensitive information has to be stored on an untrusted system. For instance, if someone else has a cheaper means to store large amounts of data or offers a better network connectivity, it is beneficial to outsource your data to that system. In the literature we find different approaches to secure data. Some approaches use access control while others use encryption. In this thesis we focus on the latter approach. We do not assume that the storage system itself is secure.

In this PhD thesis we envisage the following scenario. There exists a server with a large storage capacity and a large bandwidth. This server is considered honest but curious. This means that on the one hand we trust that it stores the data correctly and follows the protocols. On the other hand it cannot be trusted to refuse access to unauthorised people. Since the security of the system itself cannot be trusted, the data should be stored in encrypted form at the server. Authorised people should still be able to query the encrypted database efficiently. The goal of the search process is to perform the majority of workload at the server, allowing low power devices to connect to the database.

Three solutions are presented. The first solution uses a trapdoor mechanism. The data is encrypted in such a way that it is possible to search for a certain word. The server is given a key that is specific for that particular word. With this key the server is able to scan the encrypted text and find occurrences of the word. Although the server does not know which word it is being asked for, it will learn the location where the word can be found, if it can be found at all. The server does not learn anything else about the text.

The second solution uses secret sharing. The text to be stored is split in two (or more) shares. Both shares are needed to reconstruct the original text. The text is split in such a way that it is possible for the data owner to regenerate his own share, so that he does not actually have to store it. The other share is stored at the server. The search process consists of an interactive protocol between the data owner and the server. The server does not learn the location where the answer can be found, as in the first solution, but the client has more work to do.

A third category of solutions uses homomorphic encryption functions. Homomorphic encryption makes it possible to perform simple operations like addition and multiplication directly on the encrypted data, without the need to decrypt it first. We explore possibilities to use this type of encryption functions to search in encrypted data.

The thesis ends with a storage technique, based on the principles of a lucky dip, in which the security not solely relies on the computational complexity, like in standard cryptography, but also on information theoretic security. The information will be torn into shreds by using secret sharing before they are are mixed with similar shreds from other documents. The security of the lucky dip containing all those shreds is based on the fact that many combinations of shreds result in correctly readable texts. The number of combinations increases dramatically with the number of shreds. Enumerating all possible combinations is not feasible in practice. Even if we assume that an attacker has unlimited time or has unlimited computational power, the attacker still does not have certainty which messages are stored in the lucky dip. Although the attacker finds all the stored messages, he also 'finds' almost every other text imaginable. The attacker does not know which combination results in a readable text by accident and which one is stored deliberately.

Summarising, we offer three approaches for a client to query a database such that the server neither learns the query nor the stored data:

- An encrypted XML text can be searched efficiently for the occurrences of a word. The search takes place entirely at the server. The server learns only the locations of the word, if it occurs at all, but nothing more about the text or the search word.

- Using a secure protocol between the server and the client and data represented as shared polynomials, we can secure the data, the query as well as the answer, at the cost of more work for the client.

- The simple operations that homomorphic encryption can perform on encrypted data, are sufficient to search the encrypted data.

Samenvatting

Dit proefschrift behandelt het probleem van het beveiligen van gegevens die op een niet vertrouwd systeem moeten worden opgeslagen. Er zijn situaties denkbaar waarin persoonlijke data of anderszins gevoelige informatie moeten worden opgeslagen op een niet-vertrouwd systeem. Als iemand bijvoorbeeld een manier aanbiedt om goedkoper en beter bereikbaar grote hoeveelheden data te bewaren, is het aanbevelenswaardig om de data opslag uit te besteden. In de literatuur vinden we verschillende aanpakken om gegevens te beveiligen. Sommige gebruiken toegangscontrole, terwijl andere encryptie gebruiken. In dit proefschrift zullen we de nadruk leggen op de laatste aanpak. We zullen niet vertrouwen op de veiligheid van het opslagsysteem zelf.

In dit proefschrift hebben we steeds het volgende scenario voor ogen. Er bestaat een server met een grote opslagcapaciteit en een grote bandbreedte. We beschouwen deze server als eerlijk maar nieuwsgierig. Dit houdt in dat we er aan de ene kant vanuit gaan dat de server de data correct bewaart en zich aan de regels van het protocol houdt, maar aan de andere kant gaan we er niet vanuit dat ongeautoriseerde personen de toegang wordt ontzegd. Aangezien de veiligheid van het systeem zelf niet vertrouwd wordt, moet de data in geëncrypte vorm op de server worden opgeslagen. Geautoriseerde personen moeten nog steeds in staat zijn om op een efficiënte wijze de database te bevragen. Het doel is om het zoekproces zoveel mogelijk op de server uit te voeren, zodat het voor simpele mobiele apparaten mogelijk wordt om met de database te communiceren.

Er worden drie oplossingen gepresenteerd. De eerste oplossing gebruikt een zogenaamde achterdeur. De data wordt op een zodanige manier vercijferd dat het mogelijk is om op een bepaald woord te zoeken. De server krijgt dan een sleutel die specifiek is voor dit woord. Deze sleutel stelt de server in staat om een vercijferde tekst te doorzoeken naar het vóórkomen van het woord. Hoewel de server dus niet weet naar welk woord er gezocht wordt, leert hij wel waar het woord voorkomt of niet, als het al voorkomt. Van de rest van de vercijferde tekst leert hij niets.

De tweede oplossing gebruikt ‘secret sharing’. De te bewaren tekst wordt gesplitst in twee (of meer) delen. Beide delen zijn noodzakelijk om de originele tekst te reconstrueren. De opdeling van de tekst is zodanig dat de eigenaar van de data zijn eigen deel kan hergenereren, waardoor hij dit niet hoeft te bewaren. Het andere deel wordt op de server opgeslagen. Het zoekproces is een interactief protocol tussen de data eigenaar en de server. De server leert niet de locatie waar het antwoord staat, zoals het geval is in de eerste oplossing, maar de client heeft wel meer werk te doen.

Een derde categorie oplossingen gebruikt homomorfe encryptie functies. Homomorfe encryptie staat ons toe om sommige simpele operaties als optellen en vermenigvuldigen direct op de geëncrypte data toe te passen, zonder daarbij de data eerst te hoeven decrypten. We maken een inventarisatie in hoeverre deze categorie encryptie functies gebruikt kan als methode om te zoeken in geëncrypte data.

Het proefschrift wordt afgesloten met een opslagtechniek dat gebaseerd is op een grabbelton, waarbij de veiligheid niet alleen berust op de computationele complexiteit, zoals bij normale cryptografie, maar ook op informatie theoretische veiligheid. De informatie wordt door middel van ‘secret sharing’ in snippers verscheurd alvorens ze worden gemixt met soortgelijke snippers van andere documenten. De veiligheid van de grabbelton die al deze snippers bevat, berust op het feit dat er zeer veel snippercombinaties zijn die een goed leesbare tekst opleveren. Het aantal combinaties neemt drastisch toe met het aantal snippers. Het aflopen van alle mogelijke combinaties is in de praktijk niet mogelijk. Maar zelfs als we aannemen dat een aanvaller de beschikking heeft over oneindig veel tijd of over een oneindige rekenkracht, dan nog kan hij niet met zekerheid zeggen welke berichten er in de grabbelton zitten. De aanvaller vindt weliswaar alle berichten die zijn opgeslagen, maar hij ‘vindt’ ook zo’n beetje iedere andere denkbare tekst. De aanvaller weet niet welke combinatie toevallig een leesbare tekst oplevert en welke er bewust is ingestopt.

Samengevat bieden we drie technieken waarmee een gebruiker de database kan bevragen op een zodanige manier dat de server de vraag noch de opgeslagen data leert.

- Een versleuteld XML bestand kan efficiënt doorzocht worden naar het vóórkomen van een specifiek woord. Het zoekproces wordt volledig op de server uitgevoerd. De server leert enkel de locaties waar het woord voorkomt, als het al voorkomt. De server leert verder niets over de tekst

of over het zoekwoord.

- Gebruik makend van een beveiligd protocol tussen de server en de client en een manier om de gegevens te splitsen in meerdere polynomen, zijn we in staat om de gegevens, de vraag, en het antwoord te beveiligen. Het kost alleen wel wat meer werk voor de client.

- De simpele operaties die homomorfe encryptie kan toepassen op de vercijferde gegevens, zijn voldoende om te kunnen zoeken in de vercijferde gegevens.

Acknowledgements

Researchers, and especially PhD students, are often regarded as individualists working day and night in their ivory tower. Having been a PhD student for four years myself, I have to contradict this. My tower was not made of ivory and I did not spend all nights there on my own. This thesis could not have been made without the help of others. Therefore I would like to thank everybody who has contributed to this thesis.

I will start with my daily supervisors. I started with two supervisors: Ling from the database group and Sandro from the DIES group. Although they already had lots of work to do for themselves, they accepted me as their PhD student. They just worked a bit harder to teach me how to do research. After a while Jeroen took over the hard job of taming me. And I have to say, he has been quite successful. We have had plenty of fruitful brainstorm sessions. We spoke at the same frequency, understanding each others scribbles on the white board. This does not mean we did not have different opinions, but with good scientific reasoning we always agreed in the end.

I also would like to thank my two promotors: Pieter and Wim. Pieter was able to let me look at my research challenges from a completely different angle than I was looking before, rescuing me from the tunnel vision those PhD students in ivory towers suffer from. Wim has amazed me for his efficiency. He is capable of reading my papers within a minute and have very good comments on them afterwards. Also when I explained things to him that took me weeks to understand, he immediately grasped the point. Although his agenda was always overbooked he always found time to have regular meetings with me. With respect to this, I would like to thank his secretaries Suse, Sandra, Ida and Anne for making those appointments.

Our group secretaries, Marlous, Nicole and Thelma were always happy to help me with my financial businesses and booking conference trips. I really appreciated their help.

I would also like to thank my parents and my brother for their support in me. My father helped to correct the English of my thesis, although he did not understand most of the technical parts. Even my cat Jikke contributed to my thesis, by walking over my keyboard from time to time.

Staying focused for 4 years was not possible without the necessary relaxation. The Twentse Studenten Alpen Club (TSAC) has been quite capable of filling my spare time. I enjoyed the many weekends and vacations I spend with my fellow climbers. I especially would like to thank Martin and Timo who accepted the offer to be my paranimphs.

Contents

Chapter 1

Introduction

When private information is stored in databases that are under the control of others, a typical way to protect the data, is to encrypt the data before storing it. To retrieve the data efficiently, a search mechanism is needed that still works over the encrypted data. This chapter gives a brief overview of several search strategies that exist in the literature and introduces our own techniques which will be further investigated in subsequent chapters. Some techniques add meta-data to the database and do the searching only in the meta-data, while others search in the data itself, use secret sharing or homomorphic encryption methods to solve the problem. Each strategy has specific advantages and disadvantages.

1.1 Problem statement

In a thesis about *searching in encrypted data* we should first ask ourselves the questions:

- Why should we want to protect our data using encryption?
- Why not use access control?
- Why should we want to search in encrypted data?
- Why not decrypt the data first and then search in it?

Access control is a perfect way to protect your data as long as you trust the access control enforcement. And exactly that condition often makes access control simply impossible.

Consider a database on your friend's computer. You store your data on his computer because he has bought a brand new large capacity hard drive. Furthermore, he leaves his computer always on, so that you can access your data from everywhere with an Internet connection. You trust your friend to store your data and to make daily backups. However, your data may contain some information you do not want your friend to read (for instance, letters to your girlfriend). In this particular setting you cannot rely on the access control of your friend's database, because your friend has administrator privileges. He can always circumvent the access control or simply turn it off.

Fortunately, there is an alternative. You can encrypt all your sensitive information before storing it in the database. Now you can use your friend's bandwidth and storage space without fearing that he is reading your private data.

A problem arises when more and more information is being stored. Although storing it is not problematic, retrieval is. In the situation before you encrypted your data you were able to send a precise query to the server and to retrieve only the information you needed. But in the situation where all the information is stored in encrypted form you cannot make the selection on the server any more. So, for each query you have to download the whole database and do the decryption and querying on your own computer. Since you may have a slow Internet connection, you get tired of waiting for the download to finish. Of course, you can send your encryption key to your friend's database and ask it to do the decryption for you, but then you end up in almost the same situation as you started with. If the database can decrypt your data, your friend can read it.

We see a similar trend to outsource data in the hosting of Internet websites. Often special centres are being used that are administered by external system administrators. These system administrators have full access rights to all of the data, which is not a problem when dealing with publicly accessible websites. However, this changes drastically when it comes to sensitive information.

Not only companies outsource their data, also consumers do it. People used to store their e-mails and photos on their own computers. Nowadays, people use more and more web-based solutions to store their e-mails (hotmail, gmail, imap), photos and even home-made movies (youtube). All this outsourced private content should be made searchable.

From these examples we can distill the following research question:

> *"Can we store private data securely on a database server, when we cannot rely on its access control mechanism, in such a way that it is possible to search the data efficiently?"*

After having found a method to securely outsource the data, we would like the data to stay secure in the future. Our second research question, therefore, is:

> *"Can data be stored in such a way that it stays secure forever without relying on computational assumptions?"*

1.2 Literature overview

Traditionally, databases are protected by means of some kind of access control mechanism. Those mechanisms work fine under the assumption that the database runs on a trusted server. In this thesis we will weaken this assumption. To keep the data hidden from the prying eyes of non-authorised users, many of the publicly available database systems offer the opportunity to encrypt records. However, none of those systems provide a way to efficiently query the encrypted records.

In the literature some solutions to our research question have been proposed. We can categorise them in four classes:

Using indices Instead of searching in the encrypted data itself, the actual search is performed in an added index. The index contains for example the hashes of the encrypted records [30–33].

Using trapdoor encryption Trapdoor encryption makes it possible to give a user a way to perform some operation on the encrypted data without the need to give him the encryption key. A possible goal of trapdoor encryption is to allow a user to search for a particular keyword, without giving him the opportunity to find any other keyword [9,10,25,46,50].

Using secret sharing Data can be stored securely by distributing it over several servers. If the servers do not collude, the data will be secured forever. The data is queried by using a secure protocol between the client and the servers [13,14,35,36].

Using homomorphic encryption Some encryption functions give the ability to perform some simple operations directly on the encrypted data without the need to decrypt the data first. This property can be used also to search in the encrypted data [13,14,18].

The rest of this section will categorise the existing solutions into one of these categories. For each category the most cited solution will be explained in more detail.

1.2.1 Using indices

Relational databases use tables to store the information. Rows of the table correspond to records and columns to fields. Often hidden fields or even complete tables are added which act as an index. This index does not add information; it is only used to speed up the search process. Hacıgümüş et al. [30–32] use the index idea to solve the problem of searching in encrypted data. To illustrate their approach we will use the example of table 1.1, which is stored on the server as shown in table 1.2.

id	*name*	*salary*
23	Tom	70000
860	Mary	60000
320	Tony	50000
875	Jerry	5600

Table 1.1: Plain text *salary* table.

etuple	id^S	$name^S$	$salary^S$
010101011...	4	28	10
000101101...	2	5	10
010111010...	8	28	2
110111101...	2	7	1

Table 1.2: Encrypted *salary* table.

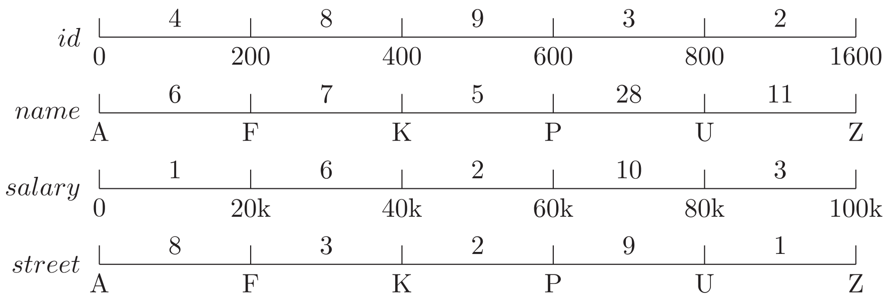


Figure 1.1: Partitioning of the *id*, *name*, *salary* and *street* fields.

The first column of the encrypted table contains the encryptions of whole records. Thus $etuple = E_k(id, name, salary)$, where $E_k(\cdot)$ is the encryption function with key k. The extra columns are used as an index, enabling the server to prefilter records. The fields are named the same as the plaintext labels, but are annotated with the superscript S which stands for 'server' or 'secure'. The values for these fields are calculated by using partitioning functions drawn as intervals in figure 1.1. The labels of the intervals are chosen randomly. For example, consider Tony's salary. It lies in the interval $[40k, 60k\rangle$. This interval is mapped to the value 2 which is stored as the $salary^S$ field of Tony's record. It is the client's responsibility to keep these partitioning functions secret.

Querying the data is performed in two steps. Firstly, the server tries to give an answer as accurately as it can. Secondly, the client decrypts this answer and post-processes it. For this two-stage-approach it is essential that the client splits a query Q into a server part Q^S (working on the index only) and a client part Q^C (which post-processes the answer retrieved from the server). Several methods of splitting are possible. The goal is to reduce the workload of the client and the network traffic. To have a realistic query example, let us first add a second table containing addresses to the database. The plain *address* table is shown in table 1.3. It is stored encrypted on the server as shown in table 1.4.

id	*street*
23	Avenue 4
860	Owl street 4
320	Downing street 10
875	Longstreet 100

Table 1.3: Plain text *address* table.

etuple	id^S	$street^S$
110111100...	4	8
110111110...	2	2
000111010...	8	8
001110110...	2	2

Table 1.4: Encrypted *address* table.

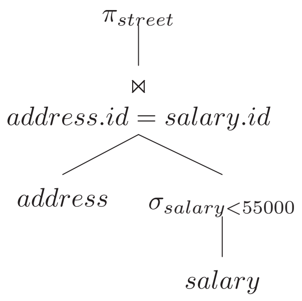


Figure 1.2: Optimal query evaluation on unencrypted data.

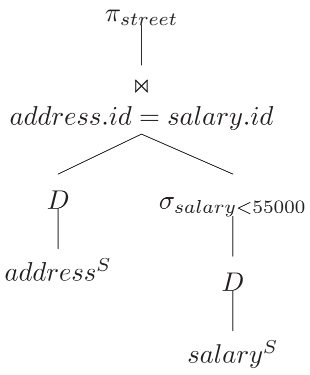


Figure 1.3: Inefficient evaluation on encrypted data.

As an example query we choose the following SQL query:

```
SELECT street
FROM address, salary
WHERE address.id=salary.id AND salary<55000
```

SQL is a declarative query language. It does not dictate the database *how* the result should be calculated only *what* the result should be. The database has freedom in the sequence of operations (selection (σ), projection (π), join ($\bowtie$), etc.). In this case the optimal evaluation is the one drawn in figure 1.2.

The direct translation of the query tree to the encrypted domain is drawn in figure 1.3. The tables are decrypted before the normal query evaluation is performed. It clearly calculates the correct result but misses our goal of reducing network bandwidth and client computation. Because the decryption can only be done at the client the encrypted tables have to be transmitted over the network and decrypted on the client. Therefore the operators should be pushed below the decryption operator D as much as possible, doing the majority of the work at the server side. To prove the correctness of those transformations Hacıgümüş et al. [30–32] have designed a theoretic algebra similar to the relational algebra.

In figure 1.4 the selection on the salary is pushed below the decryption. Notice that the selection $\sigma^S_{salary^S \in \{1,6,2\}}$ returns also salaries between 55000 and 60000, so the client side selection $\sigma_{salary<55000}$ cannot be left out. After the client selection is pulled above the join (not shown), the join can be pushed

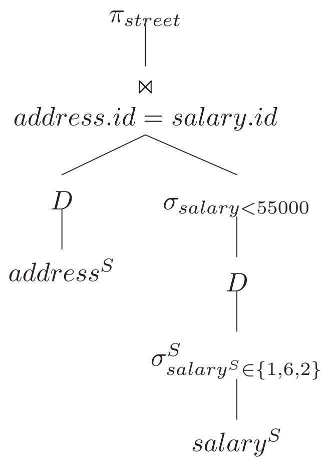


Figure 1.4: Selection pushed down.

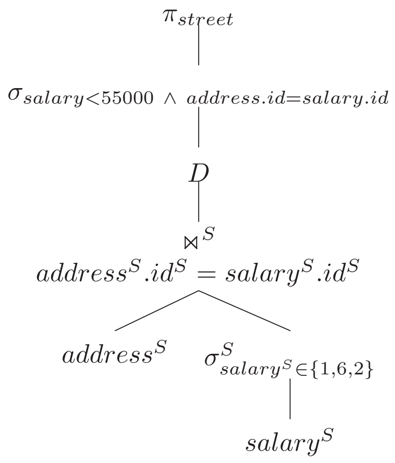

π_{street}
$\sigma_{salary<55000 \ \wedge \ address.id=salary.id}$
D
$\bowtie^S$
$address^S.id^S = salary^S.id^S$
$address^S$
$\sigma^S_{salary^S \in \{1,6,2\}}$
$salary^S$

Figure 1.5: Efficient evaluation on encrypted data.

below the decryption as shown in figure 1.5.

The original strategy as described in [31] has two drawbacks: it cannot handle aggregate functions like SUM, COUNT, AVG, MIN and MAX very well and frequency analysis attacks are possible.

In a follow up paper [33] Hacıgümüş et al. extend the method described in this section with privacy homomorphisms [18], allowing operations like addition and multiplication to work on encrypted data directly, without the need to decrypt first.

The second drawback of the original method is dealt with by Damiani et al. [16]. Instead of using an encrypted invertible index, they use a hash function that is designed to have collisions. This way, an attacker has no certainty that two records are equal when they have the same index. The proposed indexing mechanism, which is based on the B+ tree indexing method, can balance the trade-off between efficiency and security. This solution however, still suffers from linkability. Two different hashes means that the corresponding plaintexts are different too. And although the same hashes do not guarantee equal plaintexts, it is still a strong indication that they are equal. Our solutions in chapters 3 and 4 do not suffer from this linkability.

1.2.2 Using trapdoor encryption

In contrast to the approach of Hacıgümüş et al., Song, Wagner and Perrig [46] do not need extra meta-data. In their approach the search is done in the encrypted data itself. They use a protocol that uses several encryption steps which will be explained in section 2.2.

The protocol has two drawbacks:

- The plaintext is split into fixed sized words which is not natural, especially not for natural languages.
- The search time complexity is linear in the length of the whole database. It does not scale up to large databases.

We solve both drawbacks in section 2.3. There we use XML as a data format and exploit its tree structure to get a logarithmic search complexity instead of a linear complexity.

Both Boneh et al. [10] and Goh [25] combine the index based approach with the trapdoor encryption method. They encrypt a message sent by Alice to Bob with the public key of Bob. In order for intermediate nodes, like the mail server, to find particular keywords, Alice may append a *Public Key Encryption with Keyword Search (PEKS)* entry for each keyword. When Alice sends a message M with keywords $W_1, \ldots, W_m$, she transmits $\langle E_{B_{pub}}(M)||\text{PEKS}(W_1)|| \cdots ||\text{PEKS}(W_m)\rangle$. Bob may want his mail server to filter his mails according to some keywords. Bob can give his mail server the ability to find a predefined set of keywords by giving it a trapdoor for each keyword W. With such a trapdoor and a PEKS, the server can test whether the PEKS matches the trapdoor. The approaches of Boneh et al. and Goh only differ in the implementation.

All these keyword based search techniques can only be used to find exact matches. Agrawal et al. [9] provide an order-preserving scheme for numeric data that allows any comparison operation directly applied on the encrypted data.

Waters et al. [50] use a similar technique which is based on the work of Song et al. [46], to secure audit logs. Audit logs contain detailed and probably sensitive information about past execution. It should therefore be encrypted. Only when there is a need to find something in the encrypted audit log, a trusted party can generate a trapdoor for a specific keyword.

1.2.3 Using secret sharing

A third solution to our problem uses secret sharing [2, 7]. In this context, sharing a secret does not mean that several parties know the same secret. In cryptography secret sharing means that a secret is split over several parties in such a way that no single party can retrieve the secret. The parties have to collaborate in order to retrieve the secret.

Secret sharing can be very simple. To share, for instance, the secret value 5 over 3 parties a possible split can be 12, 4 and 26. To find the value back all the 3 parties should collaborate and sum their values modulo 37 ($5 \equiv 12 + 4 + 26 \pmod{37}$).

A typical usage of secret sharing is Private Information Retrieval (PIR) [14]. PIR aims at letting a user query the database without leaking to the database which data was queried. The idea behind PIR is to replicate the data among several non-communicating servers. A client can hide his query by asking all servers for a part of the data in such a way that no server will learn the whole query by itself. Chor et al. [14] prove that PIR with a single server can only be done by sending all data to the client for each query. Computational PIR [13, 14, 35] uses cryptographic techniques to achieve a similar goal as information theoretic PIR. Lin and Candan [36] use a single server scheme which is a compromise between total privacy and efficiency. A query is hidden by asking for more data than required. The server cannot tell which data is really needed and which data is just added garbage. To avoid replay attacks and server learning, all data elements in the retrieved set are shuffled and stored at different locations after each query.

The database scheme that is described in chapter 3, uses the idea of secret sharing to accomplish the task of storing data such that you need both the server and the client to collaborate in order to retrieve the data. Further requirements are:

- The server should not benefit from the collaboration. Its knowledge about the data should not increase (much) during the collaboration.
- The data split should be unbalanced, meaning that the server share is heavier (in terms of storage space) than the client share.

In chapter 3 the encoding of the data is described in full detail, including a protocol to search the data efficiently.

1.2.4 Using homomorphic encryption

Homomorphic encryption is a type of encryption with a special property. This property makes it possible to calculate with encrypted values. Using the Paillier encryption function [40], for example, it is possible to calculate the sum of two encrypted values by multiplying the two encryptions. Thus,

$$E(x) \cdot E(y) = E(x + y). \tag{1.1}$$

While it is possible to argue that this property weakens the security of the encryption, it certainly has its purpose. It is often used for secure electronic voting or for private information retrieval (PIR) [13,14]. The latter aims at hiding the query from the database server. The server stores the data in plaintext, but does not know what data is being asked for.

Homomorphic encryption has not been used to search in encrypted data, yet. In chapter 4 an investigation is made whether it is possible to store the data in encrypted form (by using homomorphic encryption), while still being able to use the PIR techniques to hide the query.

1.3 Contributions

Chapters 2-4 give new or improved solutions for our first research question while chapter 5 addresses the second research question.

We summarise the contributions of this thesis here.

Classification of the field of searching in encrypted data We have made a classification of the techniques that can be used to search in encrypted data. This resulted in a book chapter [1]. Parts of it are being reused in this thesis (especially in this introductory chapter).

Tree extension to the Song et al. scheme Chapter 2 improves the technique that was introduced by Song et al. [46]. The original scheme of Song et al. has a linear time complexity for searching unstructured text. We have extended their scheme to make it more suitable for tree structured data. The efficiency is improved from linear to logarithmic complexity. The research has been carried out in close collaboration with Jeroen Doumen, Willem Jonker, Ling Feng and Pieter Hartel and has resulted in a journal paper [4].

Secure multi party search protocol based on secret sharing Chapter 3 builds on the idea of secret sharing to make an interactive protocol between a client and a server system. The protocol ensures the secrecy of the query while an encoding based on polynomials, ensures the secrecy of the stored data. The research has been carried out together with Jeroen Doumen, Willem Jonker, Berry Schoenmakers and Pieter Hartel and has resulted in two papers and a patent application [3]. The first paper [2] gives the fundamental theoretical basis, while the second paper [7] presents experimental data from tests we carried out with our developed prototype.

Exploration of the use of homomorphic encryption in the domain of searching in encrypted data Chapter 4 explores ways to use homomorphic encryption functions to solve the research challenge addressed in this thesis. Together with Jeroen Doumen, Willem Jonker and Pieter Hartel we investigated means to extend Private Information Retrieval (PIR). PIR is a way to hide a query to the database system while keeping the stored data in the clear. Our extensions aim at encrypting the stored data too.

Secure long term storage Chapter 5 gives a solution for our second research question. The research has been carried with Willem Jonker and Stefan Maubach and has resulted in a patent application [6].

The three solutions of chapters 2, 3 and 4 answer our first research question with a 'yes', whereas chapter 5 answers our second research question with a 'yes'. Although both research questions are answered affirmatively, not all solutions are equally efficient. In the concluding chapter the different solutions are compared to each other with respect to efficiency, security and practicality. Which solution is best depends on the system architecture, the structure of the data, the query complexity and the preferred balance between security and efficiency.

Chapter 2

Linear versus tree search

Song, Wagner and Perrig (SWP) have published a theoretical paper about keyword search in encrypted textual data. We describe a prototype implementing their theory. Tests are carried out with this prototype to analyse efficiency. As expected encryption and search times are linear in the size of the database. More interestingly they also depend on the block sizes used in the protocol.

Since the search speed is linear in the size of the document, SWP does not scale well to a large database. We have developed a tree search algorithm based on the linear search algorithm that is suitable for XML databases. Our schema is more efficient than SWP since it exploits the tree structure of an XML document. We have built a similar prototype implementation for the tree search case. Experiments show a reduction in search time from linear to logarithmic in the size of the database.

2.1 Introduction

Song, Wagner and Perrig (SWP) [46] describe a protocol to store sensitive data on an untrusted server. A client (Alice) can store data on the untrusted server (Bob) and search in it, without revealing the plain text of either the stored data or the query. Only the query result is known to both Alice and Bob when the protocol finishes.

The data that is being searched is unstructured text. The search process is therefore a linear process. To investigate the scalability, we have made a prototype (section 2.2.1) implementing the original scheme. Our test results (section 2.2.2) show, as expected, a linear connection between the size of the database and the search time.

In section 2.3 we introduce an extension to the original scheme. Our extension uses structured XML documents instead of unstructured text. The tree structure of an XML document is exploited to improve the search speed from linear to logarithmic in the size of the database. This comes at the price of handing the server the tree structure. Another prototype (section 2.3.1) demonstrates the scalability of our tree extension (section 2.3.2).

In section 2.4 we compare the test results of both prototypes.

2.2 Linear search strategy

The original SWP protocol of Song et al. [46] consists of three parts: storage, search and retrieval. After summarising the protocol we will discuss our implementation and test results showing the influence of various parameters on the encryption and search times.

Storage

Before Alice can store information on Bob she has to do some calculations. First of all she has to fragment the whole plaintext W into several fixed sized words W_i. Each W_i has length n. She also generates encryption keys k' and k'' (which are used for every word) and a sequence of reproducible random numbers S_i using a pseudo-random bit generator. Then she has or calculates the following for each block W_i:

$$\begin{array}{ll} W_i & \text{plaintext block} \\ k'' & \text{encryption key} \\ X_i = E_{k''}(W_i) = \langle L_i, R_i \rangle & \text{encrypted text block} \\ k' & \text{key for } f \text{ (see below)} \\ k_i = f_{k'}(L_i) & \text{key for } F \text{ (see below)} \\ S_i & \text{random number } i \\ T_i = \langle S_i, F_{k_i}(S_i) \rangle & \text{tuple used by search} \\ C_i = X_i \oplus T_i & \text{value to be stored} \end{array} \tag{2.1}$$

Here E is a standard symmetric block cipher and f and F are pseudo-random functions:

$$\begin{array}{l} E : key_{64} \times int_n \rightarrow int_n \\ f : key_{64} \times int_{n-m} \rightarrow key_{64} \\ F : key_{64} \times int_{n-m} \rightarrow int_m \end{array} \tag{2.2}$$

The encrypted word X_i has the same block length as W_i (i.e. n). L_i has length $n-m$ and R_i has length m (see Figure 2.1). The parameters n and m may be chosen freely. Section 2.2.3 gives guidelines for efficient values for n and m. The value C_i can be sent to Bob and stored there. Alice may now forget the values W_i, X_i, L_i, R_i, k_i, T_i and C_i, but she should remember k', k'' and S_i.

Search

After the encrypted data is stored on Bob in the previous phase Alice can ask Bob queries. Alice can provide Bob with an encrypted version of the plaintext word W and ask him if and where W occurs in the original document. If W has been found at location j (i.e. $W = W_j$) then $\langle j, C_j \rangle$ is returned. Alice has or calculates:

$$\begin{array}{ll} k'' & \text{encryption key} \\ k' & \text{key for } f \\ W & \text{plaintext block to look for} \\ X = E_{k''}(W) = \langle L, R \rangle & \text{encrypted block} \\ k = f_{k'}(L) & \text{key for } F \end{array} \tag{2.3}$$

Then Alice sends the value of X and k to Bob. Having X and k Bob is able to compute for each ciphertext block in the database (C_p):

$$\begin{array}{l} T_p = C_p \oplus X = \langle S_p, S'_p \rangle = \begin{cases} \langle S_p, F_k(S_p) \rangle & \text{if } W_p = W_j \\ \text{garbage} & \text{otherwise} \end{cases} \\ \text{IF } S'_p = F_k(S_p) \text{ THEN RETURN } \langle p, C_p \rangle \end{array} \tag{2.4}$$

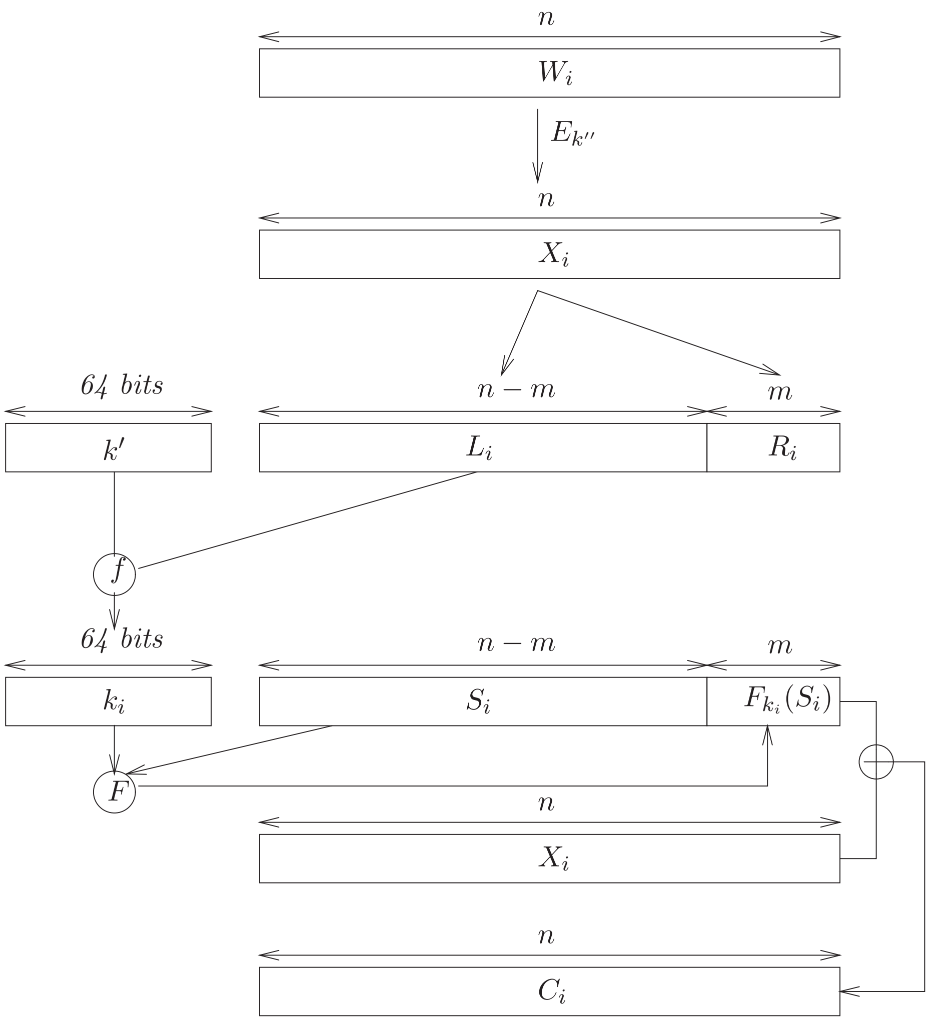


Figure 2.1: Encryption schema.

Note that all locations with a correct T_p value are returned. However there is a small probability that T satisfies $T = \langle S_q, F_k(S_q)\rangle$ but $S_q \neq S_p$. Therefore Alice should check each answer if the correct random value is used.

Retrieval

Alice can also ask Bob for the ciphertext at any position p. Alice, knowing k', k'' and the seed for S, can recalculate W_p by

$$\begin{array}{ll}
k' & \text{key for } f \\
k'' & \text{encryption key} \\
p & \text{desired location} \\
C_p = \langle C_{p,l}, C_{p,r}\rangle & \text{stored block} \\
S_p & \text{random value used for block } p \\
X_{p,l} = C_{p,l} \oplus S_p & \text{left part of encrypted block} \\
k_p = f_{k'}(X_{p,l}) & \text{key for } F \\
T_p = \langle S_p, F_{k_p}(S_p)\rangle & \text{check tuple} \\
X_p = C_p \oplus T_p & \text{encrypted block} \\
W_p = D_{k''}(X_p) & \text{plaintext block}
\end{array} \qquad (2.5)$$

Here D is the decryption function $D : key_{64} \times int_n \rightarrow int_n$ such that $D_{k''}(E_{k''}(W_i)) = W_i$.

This is all Alice needs. She can store, find and read the text while Bob cannot read anything of the plaintext. The only information Bob gets from Alice is C in the storage phase and X and k in the search phase. Since C and X are both encrypted with a key only known to Alice and k is only used to hash one particular random value, Bob does not learn anything of the plaintext.

The only information Bob learns from a search query is the location where an encrypted word is stored and the number of occurences.

2.2.1 Implementation

Section 2.2 introduces three functions: E, f and F. Figure 2.1 shows how they are connected to each other. E could be a block cipher in ECB mode and f and F pseudo-random functions. For our prototype we chose DES for all three of them, but any other symmetric block cipher could have been used instead. E is exactly DES in ECB mode. Since DES works on blocks of 64 bits n should be a multiple of 64 bits.

f and F are pseudo-random functions with variable sized output values. The output values are used as kind of hash values. Standard hash functions like SHA-1 have a fixed sized hash value. The last (or the first) m bits of the hash value could be used, but then m should be less than the size of the hash value (160 bits for SHA-1). To allow a larger value for m our prototype uses DES in CBC mode. To hash a data block of length $n - m$ to a hash value of length m the block is encrypted with the specified key (64 bits DES key) but only the last m bits are used as hash value. The only restriction for m is that $n - m \geq m$ and thus $n \geq 2m$. See Menezes et al. [37] for a more detailed description of the used hash algorithm.

The prototype implementation is split into two programs, one for the encryption and one for the search. Both programs share the same parameters (n, m, S, k', k''). The search program uses the output of the encryption program (i.e. the encrypted XML document) and the search word W to produce a list of locations where the word occurs.

2.2.2 Experimental data

The Encrypt and Search tools give us the opportunity to experiment with the parameters used in the protocol. We are especially interested in the influence the parameters n and m have on the encryption and search speed. We use the XML benchmark[1] to generate three sample files of sizes 1 MB, 10 MB and 100 MB. Although these files are XML files the tree structure is not used in the protocol. The tools just consider them as large text files. The benchmark is only used to compare the results with previous and with future experiments where we intend to exploit the tree structure for more efficient queries on encrypted data.

Changing the parameters n and m also influences the correctness of the result. Therefore, also the number of collisions has been measured (see figure 2.3(a)). Collisions are the false hits that occur because of the collisions in the hash function F. F hashes the random value S_i of size $n - m$ to a hash value of length m, where $n - m \geq m$. For $n - m > m$ collisions are unavoidable.

Tests are carried out $\forall n \in \{8, 16, 24, 32, 40, 48, 56, 64\}$ where these values are the number of bytes and not bits. Because we use DES in ECB mode for the encryption function E we only use multiples of 8 bytes. m should be less than or equal to $\frac{n}{2}$ so $m \in \{1, 2, \ldots, \frac{n}{2}\}$. Measurement results are plotted in figures 2.4-2.6. The absolute values are not interesting because they depend on

[1] http://www.xml-benchmark.org

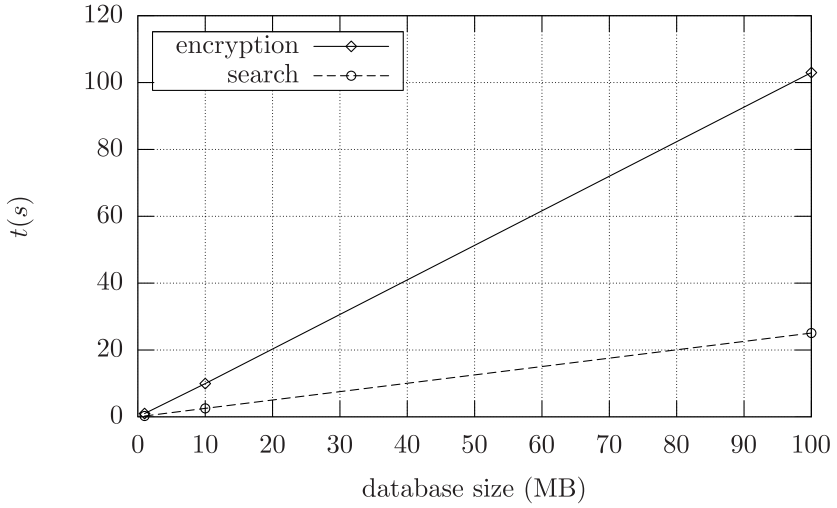


Figure 2.2: Encryption and search times with different database sizes for $n = 64$ and $m = 32$.

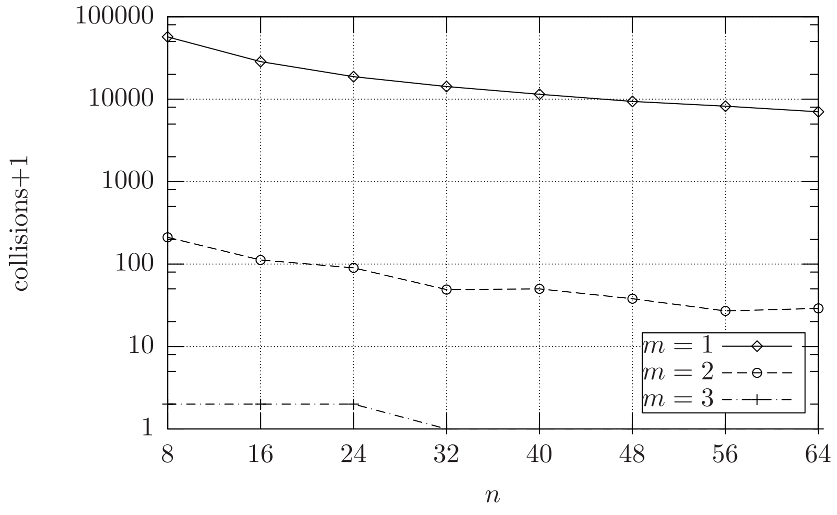
the physical hardware. However, differences between the various configurations are interesting. All tests were carried out on a Pentium IV 2.4 MHz with 512 MB memory.

2.2.3 Results

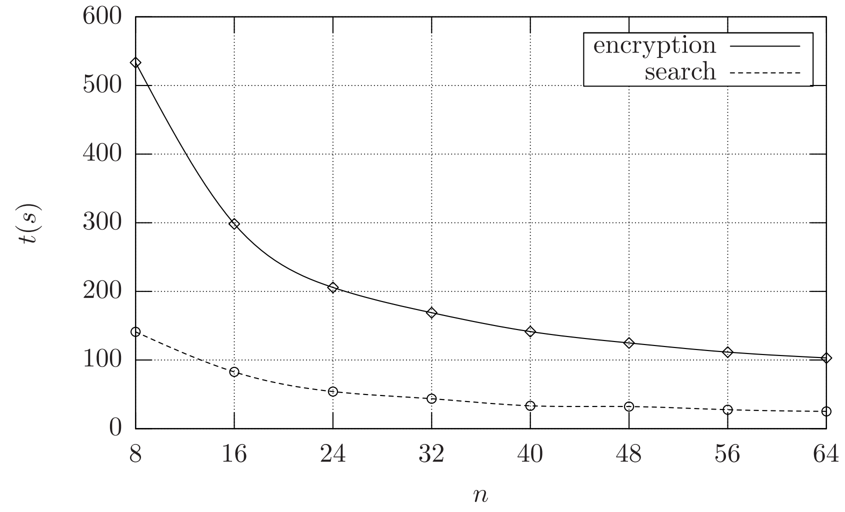
From the experiments we conclude that:

- The larger the dataset the larger the encryption and search times. As expected from the SWP theory the encryption and search time grow linear in the size of the dataset. Therefore the protocol does not scale well and can only be used for reasonable small databases (see figure 2.2).

- The larger n the shorter the encryption and search times (figures 2.4-2.6). This can be explained by looking at the number of blocks. The larger n the fewer blocks there are. For each block a fixed number of steps are taken. Most of these steps do not depend on the length of the blocks. Therefore less time is needed for the whole database.

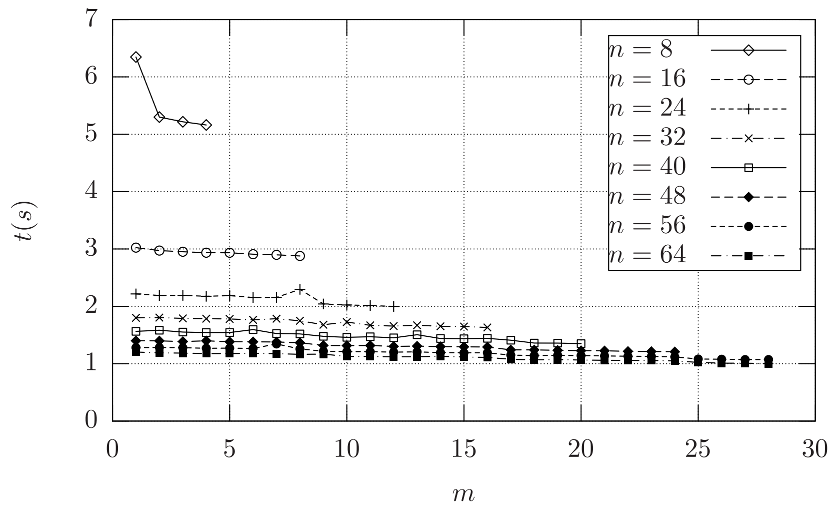


(a) Number of Measured Collisions

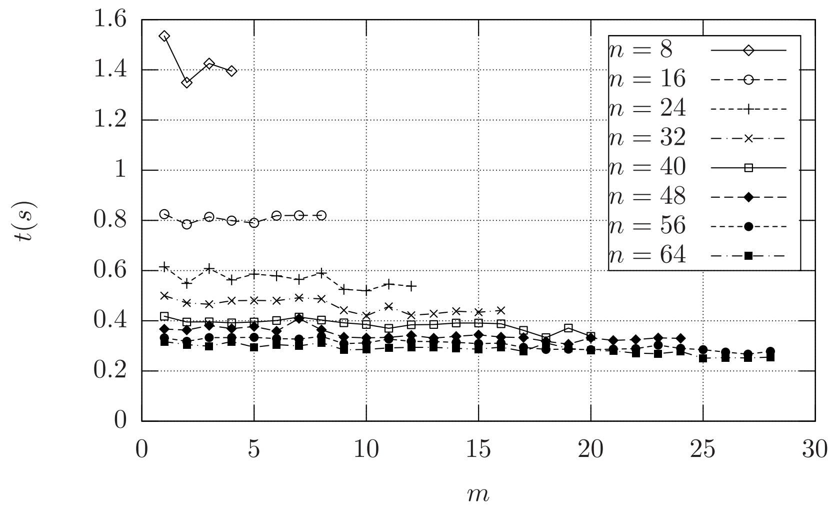


(b) Encryption and Search Times ($m = \frac{n}{2}$)

Figure 2.3: Measurement Results of Linear Search Prototype for the 100 MB Case.

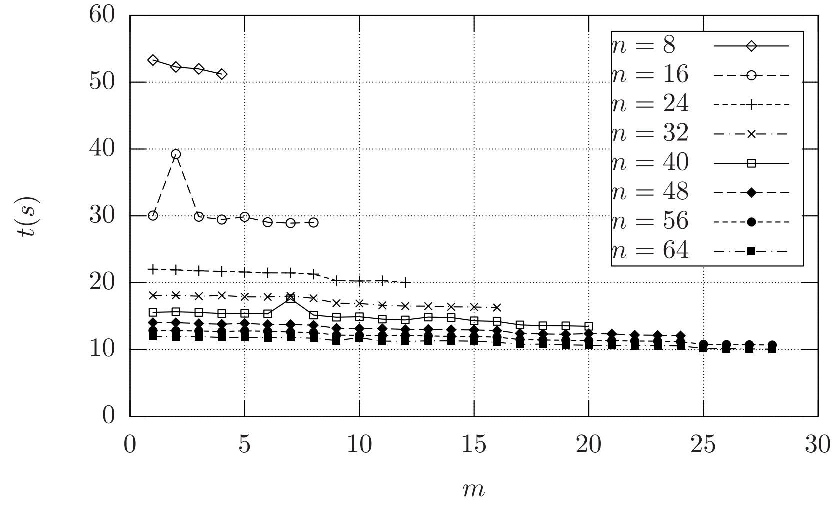


(a) Encryption speed

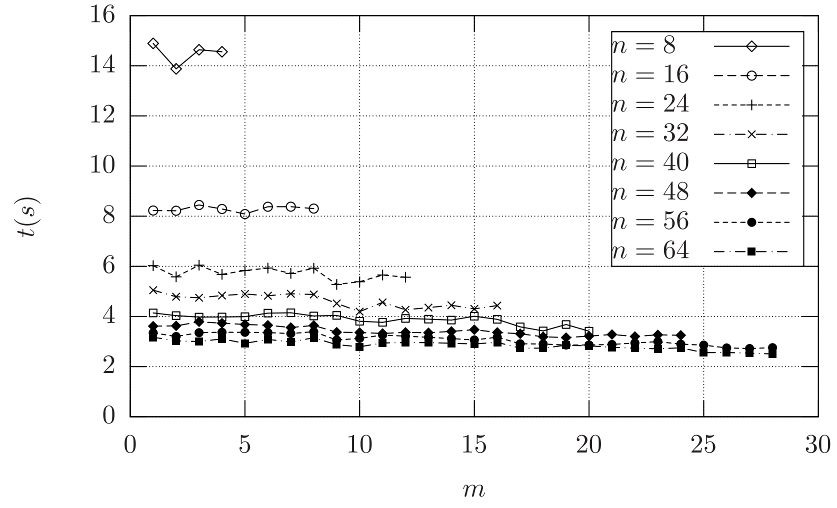


(b) Search speed

Figure 2.4: Measurement results for small dataset (1 MB).

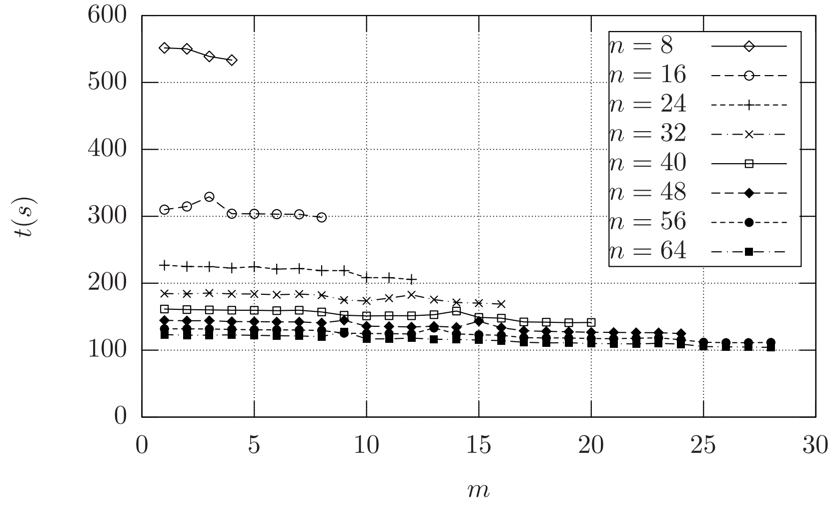


(a) Encryption speed

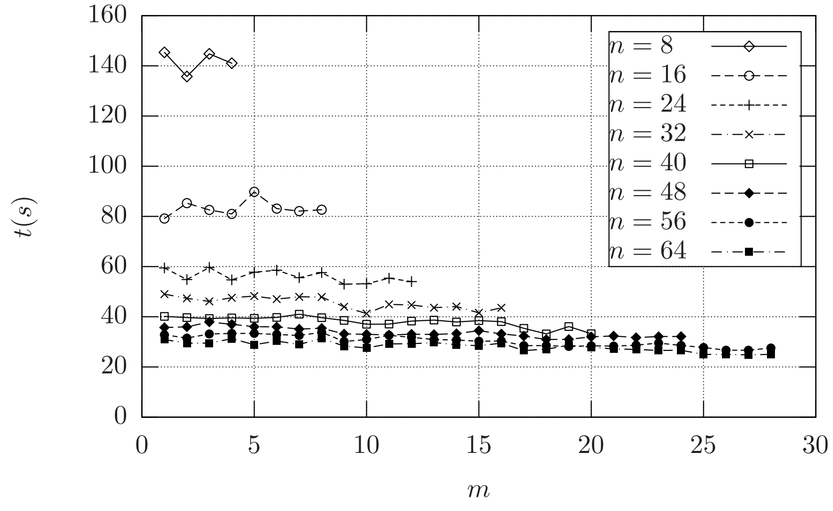


(b) Search speed

Figure 2.5: Measurement results for medium sized dataset (10 MB).



(a) Encryption speed



(b) Search speed

Figure 2.6: Measurement results for large dataset (100 MB).

- The larger n the fewer collisions occur (figure 2.3(a)). This can be explained by the smaller number of blocks too.
- For a fixed value of n the encryption and search times hardly depend on the value of m (horizontal lines in figures 2.4-2.6). Higher values for m are slightly better. Since the number of collisions is lower for higher values of m it is best to choose m maximally high (that is, $m = \frac{n}{2}$). This is also the reason why figure 2.2 and 2.3 are drawn at an m value that is half the size of n.
- Searching is faster than encryption, because fewer operations have to be calculated for each block (see figure 2.2 and 2.3).
- Collisions can be avoided by choosing a sufficiently large value of m. The largest value for m is $\frac{n}{2}$ which is also the most optimal one. But also for smaller values of m the number of collisions is negligible. Only for m values equal to 1 or 2 bytes, there are many collisions.

That the encryption and search times are linear in the size of the text, does not come as a surprise, since it can be predicted from the theory. The influence of the chosen bit lengths n and m on the encryption and search times, however, could not have been predicted from the theory alone.

2.3 Tree search strategy for XML documents

So far, we considered only text files. Using structured XML data can improve the efficiency.

Grust [28, 29] introduces a way to store XML data in a relational database such that search queries can be handled efficiently. An XML document is translated into a relational table with a predefined structure. Each record consists of the name of the tag or attribute and its corresponding value. The information about the tree structure of the original XML document is captured in the pre, post and parent fields. All fields can be computed in a single pass over the XML document. The pre and post fields are sequence numbers that count the open tags respectively the close tags. The parent value is the pre value of the parent element (see figure 2.7(a)).

It is common to use XPath [47] to localise elements within XML documents. Although the syntax of XPath is similar to the syntax used for directory names or for the addresses of a web page, an XPath expression is more than just a name. XPath is often used in conjunction with XQuery [48]. XQuery is a query language for XML documents. XQuery has more control over the output of a query than XPath. XQuery uses XPath to localise the XML elements and builds new XML documents from these elements. In this thesis we focus only on the search part. Therefore we will use XPath to express our search queries.

The XPath axes like *descendant*, *ascendant*, *child*, etc. can be expressed as simple expressions over the pre, post and parent fields. For instance:

- v is a child of $v' \iff v.parent = v'.pre$
- v is a descendant of $v' \iff v'.pre < v.pre \wedge v'.post > v.post$
- v is following $v' \iff v'.pre < v.pre \wedge v'.post < v.post$

Some XPath axes can be drawn in a pre/post plane. Each XML element has a pre and a post value, which can be plotted in a 2-D drawing as in figure 2.7(b). The solid circle indicates a single element (E), which is taken as the starting point. Horizontal and vertical axes can be drawn through this point (the dashed lines), creating four quadrants. For instance, the upper left quadrant contains all the elements with smaller pre but larger post values than the chosen starting point E. This means that all these elements have an open tag that lies before the open tag of E and have a close tag after the close tag of E. In other words, they enclose E and are therefore the ascendants of E. The other three quadrants form the XPath axes: descendants, previous siblings and following siblings.

Not all updates are efficient. Modification and deletion are no problem, but element insertion causes the need to recalculate the pre, post and parent values for all following elements. The number of recalculations can be reduced by leaving gaps in the numbering. Thus, instead of numbering the pre and post values like 1, 2, 3, . . ., number them like 100, 200, 300, ...

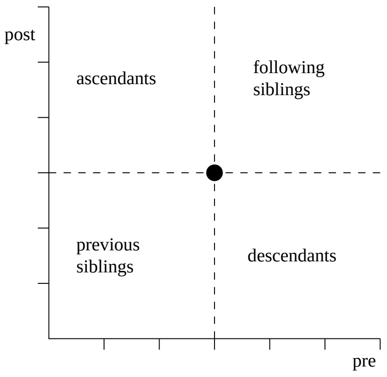
Grust aims at storing XML data in the clear. To protect the data cryptographically we combine his strategy with the linear search approach of Song, Wagner and Perrig (SWP) [46]. Only some slight modifications to the SWP approach are necessary:

1. The input file is not an unstructured text file but a tree structured XML document. The division of the data into fixed sized blocks does not seem natural. Therefore, we use variable block lengths that depend on the lengths of the tag names, attribute names, attribute values and the text between tags.

2. The sequence number of a block is no longer appropriate to define the location within a document. We use the pre value instead.

	pre	post	parent
<a>	1		0
<b>	2		1
</b>		1	
<c	3		1
d="...">	4	2	3
<e/>	5	3	3
</c>		4	
</a>		5	

(a) Pre/Post/Parent calculation



(b) Visualisation of XPath Axes in a Pre/Post Plane

Figure 2.7: Calculation and Usage of Pre, Post and Parent fields.

The equations of section 2.2 can be rewritten to the equations below. Note that all subscripts have changed. For simplicity we only describe the encryption of tag names. Exactly the same scheme is used for attribute names (prefixed with a @ sign) or the data itself by simply substituting value for tag.

Storage

Storage is analogous to the original SWP scheme. Only the subscripts have been changed.

$$\begin{array}{ll} W_{tag} & \text{plaintext block} \\ k'' & \text{encryption key} \\ X_{tag} = E_{k''}(W_{tag}) = \langle L_{tag}, R_{tag} \rangle & \text{encrypted text block} \\ k' & \text{key for } f \\ k_{tag} = f_{k'}(L_{tag}) & \text{key for } F \\ S_{pre} & \text{pseudo-random number } pre \\ T_{pre,tag} = \langle S_{pre}, F_{k_{tag}}(S_{pre}) \rangle & \text{tuple used by search} \\ C_{pre,tag} = X_{tag} \oplus T_{pre,tag} & \text{value to be stored} \end{array} \tag{2.6}$$

Note that the random value S_{pre} does not depend on the tag name but on the location (expressed in the pre field) because all elements with the same tag name should be encrypted to different values when stored.

Search

An XPath query like $/tag_1//tag_2[tag_3 = "value"]$ is encrypted to $/\langle X_{tag_1}, k_{tag_1}\rangle//\langle X_{tag_2}, k_{tag_2}\rangle[\langle X_{tag_3}, k_{tag_3}\rangle = "\langle X_{value}, k_{value}\rangle"]$ before sending it to the server. The server calculates the result traversing the XPath query from left to right. Each step consists of two or three sub steps:

- Evaluating the XPath axis /, //, [and] using the pre, post and parent fields. It is possible to find all children (/) or all descendants (//) of elements found in a previous step by just using the pre, post and parent field. See section 2.3.1 for an example.
- Filtering out the records that do not satisfy $S'_p = F_{k_{tag}}(S_p)$ in $T_{p,tag} = C_{p,tag} \oplus X_{tag} = \langle S_p, S'_p\rangle$.
- Eventually filtering out the records with an incorrect value field.

Retrieval

Also the retrieval is analogous to the original scheme. Also here, only the subscripts have been changed.

$$\begin{array}{ll}
k' & \text{key for } f \\
k'' & \text{encryption key} \\
pre & \text{desired location} \\
C_{pre,tag} = \langle C_{pre,tag,l}, C_{pre,tag,r}\rangle & \text{stored block} \\
S_{pre} & \text{random value} \\
X_{tag,l} = C_{pre,tag,l} \oplus S_{pre} & \text{left part of encrypted block} \\
k_{tag} = f_{k'}(X_{tag,l}) & \text{key for } F \\
T_{tag} = \langle S_{pre}, F_{k_{tag}}(S_{pre})\rangle & \text{check tuple} \\
X_{tag} = C_{pre,tag} \oplus T_{tag} & \text{encrypted block} \\
W_{tag} = D_{k''}(X_{tag}) & \text{plaintext block}
\end{array} \tag{2.7}$$

Example 2.3.1 *Figure 2.8 shows an XML tree. Like the server that stores the tree, we do not see any node names. The colouring of the nodes is the result of an XPath evaluation. In this example we use the XPath expression* `/a/*/b//c/d`*. White nodes do not have to be checked. The black node is the end result. Grey nodes indicate whether the check using $\langle X, k\rangle$ resulted in a match (dark grey) or a miss (light grey). As we can see, it is sufficient to check only a few nodes.*

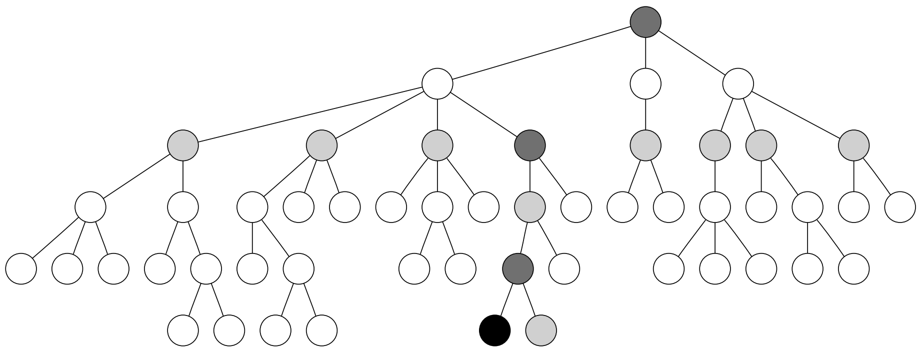


Figure 2.8: XPath evaluation of the query `/a/*/b//c/d`. Dark grey nodes indicate a match with a part of the query and light grey nodes a miss. The end result is coloured black and the nodes that have not been touched are white.

2.3.1 Implementation

Like the linear prototype the tree search prototype is split into two parts: one for encryption and one for searching.

The Encrypt tool uses a SAX parser to read the input XML document. In one pass over the input, the pre, post and parent values can be calculated. When an end tag is encountered all the information to encrypt the element is available. Attributes are handled as tags with a leading @ sign. A new record $\langle pre, post, parent, C_{pre,tag}, C_{pre,value}\rangle$ is inserted into the relational database, where $C_{pre,tag}$ and $C_{pre,value}$ are calculated as in section 2.3. In our prototype we use a MySQL database to store the encrypted document.

In contrast with the linear prototype there are no predefined block sizes n and m. Instead of using a fixed sized block, n is simply set to the length of the tag name. m is a predefined fraction of n (for example 0.5).

To speed up the search process, indices are added to the MySQL table for the pre, post and parent fields.

The search tool evaluates the XPath expression step by step. Preliminary results are stored in a result table. Each step consists of two or three sub steps:

1. Evaluate the path delimiter (/, //, [or]). For this step only the pre, post and parent fields are needed. For example // (descendants) is translated into the SQL query:

   ```
   CREATE TABLE new_result
   ```

```
SELECT data.*
FROM data, previous_result
WHERE data.pre  > previous_result.pre AND
      data.post < previous_result.post
```

2. Filter out the records in the preliminary result with the wrong tag/attribute names by applying the steps of the original linear search method.

3. When the step consists of an equation expression the previous step is repeated but now for the value instead of the name.

2.3.2 Experimental data

For the search query a word guaranteed to be in at least one location was chosen. The search engine does not stop when one occurrence is found; the whole document is scanned for each query, giving a complete answer to the query.

For the tree search prototype the only configurable parameters are m and the data size. The block length n depends on the tag names and values. Encryption tests are carried out on the same XML documents as in the linear prototype. In this case m is relative to n; $m \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$. The encryption times for the 1 MB, 10 MB and the 100 MB files are 21.5, 188 and 1195 s and do not depend on m.

Search tests with different values for m show that m does not influence the search speed. The results shown in this section are carried out with a fixed $m = 0.5$. Some queries are shown in table 2.1. Also the number of elements in the result is shown for each query (table 2.2). All three files have approximately the same tree depth but have different branch factors (average number of children per element).

2.3.3 Results

From the tree search prototype we can conclude that:

- The encryption time is linear in the size of the input.

- The encryption time and the search time hardly depend on the chosen value for m.

- The search time depends both on the structure of the XML document and the search query. The search time is of order $O(p)$ where p is the number

Table 2.1: Search times calculated for search queries with different depth and branch factor.

query	$t(ms)$ 1 MB	$t(ms)$ 10 MB	$t(ms)$ 100 MB
/site	1281	1506	1285
/site/regions	1266	1380	1321
/site/regions/asia	1358	1435	1342
/site/regions/asia/item	1409	1687	2464
/site/regions/asia/item/description	1518	2030	4135
/site/regions/africa/item/description	1376	1591	2442
/site/regions/europe/item/description	1448	2777	9059
/site/regions/australia/item/description	1455	2098	4577
/site/regions/namerica/item/description	1654	3226	13672
/site/regions/samerica/item/description	1336	1817	3028
//*	1398	2382	18530
//item	3639	21775	191899

Table 2.2: Result sizes calculated for search queries with different depth and branch factor.

query	count 1 MB	count 10 MB	count 100 MB
/site	1	1	1
/site/regions	1	1	1
/site/regions/asia	1	1	1
/site/regions/asia/item	20	200	2000
/site/regions/asia/item/description	20	200	2000
/site/regions/africa/item/description	5	55	550
/site/regions/europe/item/description	60	600	6000
/site/regions/australia/item/description	22	220	2200
/site/regions/namerica/item/description	100	1000	10000
/site/regions/samerica/item/description	10	100	1000
//*	21048	206130	2048180
//item	217	2175	21750

Table 2.3: Block Sizes (in bytes).

data size	avg tag length	standard deviation	avg text size	standard deviation	avg all blocks	standard deviation
1 MB	9.8	3.4	28	70	18	48
10 MB	9.8	3.4	29	70	18	48
100 MB	9.8	3.4	29	70	19	49

of elements to be read. For queries without // the search time is $O(bd)$ where b is the branch factor (the average number of subelements) and d is the depth in the tree where the answer is found. Figure 2.8 visualizes this. All the nodes on the path from the root to the requested node have to be examined. All siblings of those nodes have to be examined too.

- The wildcard operator (`*`), indicating any tag name, is very efficient. This can be explained by the fact that no cryptographic steps are involved. The search engine only uses the pre, post and parent values.

2.4 Benefits of using tree structure

From the experiments with the linear search method we know that the encryption time depends on the block size. Therefore, to make a fair comparison between the linear text encryption and the tree encryption, we have to take into account the block size of the tree search method. In our tree based extension, a block is formed by either a tag/attribute name or the textual information between the open and close tag. The properties of our sample document are shown in table 2.3.

As we can see from the table, the average block size of our sample XML document lies around 18 bytes. To make a fair comparison between the linear and the tree based scheme, we choose n to be equal to 18. From the linear scheme (figure 2.3) we expect an encryption time of around 275 s to encrypt a 100 MB database. In reality, however, the encryption takes 1195 s. The reason why the tree based protocol is so much slower than the linear protocol is the added complexity of the program. Whereas in the linear case the data is just unstructured data, in the tree case the data is parsed, an index is added and translated into SQL queries to fill a database. Thus much more work is done.

The major benefit of using a tree structure is the increase in search speed. Only a small part of the whole tree has to be searched. Because the search time totally depends on the data and the query, a straight comparison between the linear and the tree case is impossible. However, when we take n to be equal to 18 again, it takes the linear prototype approximately 75 s to search a 100 MB database. If we look at the last column of table 2.1 we see times are much smaller. Only the worst case query `//item` is slower. Again, this can be explained by the greater complexity of the tree implementation.

Theoretically, the linear search complexity is linear in the number of nodes that have to be examined. For an average query, only the nodes on the path from the root node to the answer and all their siblings have to be examined. With a path length of d and an (average) branch factor of b, the normal tree search complexity is $O(bd)$. Only in the worst case (with queries like `//item`) the whole tree has to examined. In that case the search complexity is $O(b^d)$ which is similar to the linear search approach.

2.5 Conclusions

We have implemented a prototype for the theory described by Song et al. [46]. We show that the search complexity is linear in the size of the text. We also have defined a new protocol for semi-structured XML data that exploits the tree structure. Experiments with the implementations of both protocols show that the encryption speed remains linear in the size of the input, but that a major improvement in the search speed can be achieved. Our contributions are:

Faster search strategies

The tree structure of the XML data can be exploited to increase efficiency. Whereas linear search is necessary in order to search for a word in an unstructured text, faster search strategies are possible when looking for a specific path in structured XML data. Tree search search decreases search time dramatically.

Variable block size

The original protocol works with a fixed block size. Words in a natural language like English have variable lengths. Therefore the English words should be padded or split which make it more difficult to search for it. Our new tree based scheme does not use fixed size blocks any more.

2.6 Future work

Currently our prototype treats the text within the XML tags as single blocks. In fact, it does not distinguish between tags/attributes and the unstructured text. A future implementation should be hybrid. The part of the query dealing with tag names and attribute names should use our tree based extension, whereas the part that deals with the unstructured text should use the original SWP scheme [46]. Our current prototype does not accept all the functions that can be used in XPath. All functions that 'calculate' over the textual information (like *contains, substring, starts-with, string-length, concat, not, sum, floor, ceiling* and *round*) are not supported. The hybrid scheme will be able to handle more of these functions, although maybe not to their full extent. For example, in the hybrid scheme the *contains* function can only be used to check whether a text contains a word or a sequence of consecutive words. It cannot be used to check whether a part of a word can be found in the text. The same holds for the functions *substring* and *starts-with*. Functions like *sum, floor, ceiling* and *round* interpret the data as numbers, which is something the SWP scheme does not support. Therefore, it is not likely that the hybrid scheme will support all the XPath functions.

Another deficiency of the current scheme is the lack of relational, additive and multiplicative expressions. Currently, only the equality operator '=', the inequality operator '!=' and the logical operators 'and' and 'or' can be used to form an expression. Further research is needed to support the full expressiveness of XPath.

As XPath is a part of XQuery, our tree extension to SWP can also be used for XQuery. Whereas XPath can only point to a location within an XML document, XQuery can build another XML document as the answer to a query. A follow-up project will investigate the possibility to make this answer searchable as well.

Chapter 3

Using secret sharing to search in encrypted data

In this chapter we present a method, inspired by secure multi-party computation, to search efficiently in encrypted data. We will encrypt an XML documents by encoding a tree of XML elements as a tree of polynomials. Each polynomial is split into two parts: a random polynomial for the client and the difference between the original polynomial and the client polynomial for the server. Since the client polynomials are generated by a random sequence generator, only the seed has to be stored on the client. In a combined effort of both the server and the client a query can be evaluated without traversing the whole tree and without the server learning anything about the data or the query.

3.1 Introduction

We propose a method that looks like secure multi-party computation where two parties, a client and the database server, together evaluate a query on an XML document. Before we will present our solution (section 3.3) we will say a few things about secure multi-party computation in general (section 3.2).

3.2 Secure multi-party computation

We speak of secure multi-party computation when several parties calculate a function result without giving the other parties access to their input. More precisely, the parties want to evaluate the function result $(y_1, \ldots, y_n) = f(x_1, \ldots, x_n)$ where each parameter x_i is the private input of party P_i and y_i its private output. It is also possible that all y's are equal. In that case it is written as $y = f(x_1, \ldots, x_n)$. In principle there exist schemes that can evaluate any function securely using secure multi-party computation [26]. However, no efficient general schemes are known to us at the moment of writing.

For example, let f be an anonymous voting function. Each voter P_i can vote for a decision ($x_i = 1$) or against it ($x_i = 0$). The function f can be defined as the function $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$ (in case of a majority vote) or as $f(x_1, \ldots, x_n) = \prod_{i=1}^{n} x_i$ (in case of a veto system).

One characteristic of secure multi-party computation is the lack of a trusted third party. In our example there is no need for a trusted party to count the votes.

Many secure multi-party computation protocols are based on Shamir's secret sharing scheme [44]. These protocols have at least two phases. In the first phase each party P_i splits up its input x_i in such a way that at least $t \leq n$ shares are needed to reconstruct x_i. In the second phase each party P_i calculates its share of the function result given only his own input and the shares of the other parties. Now, the complete function result is shared over all parties.

We will now give the implementation of one specific secure multi-party computation protocol. In this protocol P_i shares its input variable x_i by choosing a random polynomial g_i of degree t such that $g_i(0) = x_i$. P_i sends to each other party P_j the value of $g_i(j)$. When t parties collaborate they can reconstruct the original polynomial g_i by interpolating the t points $(j, g_i(j))$. With the polynomial it is easy to recalculate $x_i = g_i(0)$.

The second phase consists of the local computations with the distributed shares $g_i(j)$ and depends on the function f. For simplicity reasons we consider only our voting case where $f(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i$. Each party P_j locally calculates the sum $h(j) = \sum_{i=1}^{n} g_i(j)$. Having at least t collaborating parties and thus t points $\langle j, h(j)\rangle$ it is possible to construct the polynomial $h = \sum_{i=1}^{n} g_i$ and also $f(x_1, \ldots, x_n) = h(0)$.

3.3 Searching in encrypted data

The solution presented in this chapter has been inspired by secure multi-party computation. One way to look at the problem of searching in encrypted data [4,21] is to consider the search algorithm as a *search* function that is to be evaluated in the sense of secure multi-party computation. The $search(data, query)$ function takes two arguments, *data* and *query*, as input. Unlike secure multi-party computation both inputs originate from the same party (the client), although the *data* part is stored on the server. Our solution use the same building blocks secure multi-party computation is based on (secret sharing and a secure distributed protocol), but cannot be considered as a secure multi-party protocol.

We use a very simple form of secret sharing: addition. The original XML document is transformed to a tree of polynomials (section 3.3.1). Each polynomial is split into a random part and a server part such that the sum equals the original polynomial. We will generate the client polynomials by using a pseudo-random bit generator. Since we can rerun the generator with the same seed, all the client polynomials can be regenerated. Therefore, there is no need to store them at all. Section 3.3.2 proposes a distributed protocol to search in the data.

Damiani et al. [15] use a similar strategy in the relational setting.

3.3.1 Data representation

Secure multi-party computation works best with simple algebraic expressions like polynomials. It is possible to map the tree of elements from an XML file to a tree of polynomials. We will demonstrate this mapping by way of the example shown in figure 3.1.

A plaintext XML document is being transformed into an encrypted database by following the steps below.

1. First we introduce a function $map : node \rightarrow \mathbb{F}_p$, which maps the tag names of the nodes to values of the finite field $\mathbb{F}_p$, where p is a prime that is larger than the total number of different tag names. The mapping function may

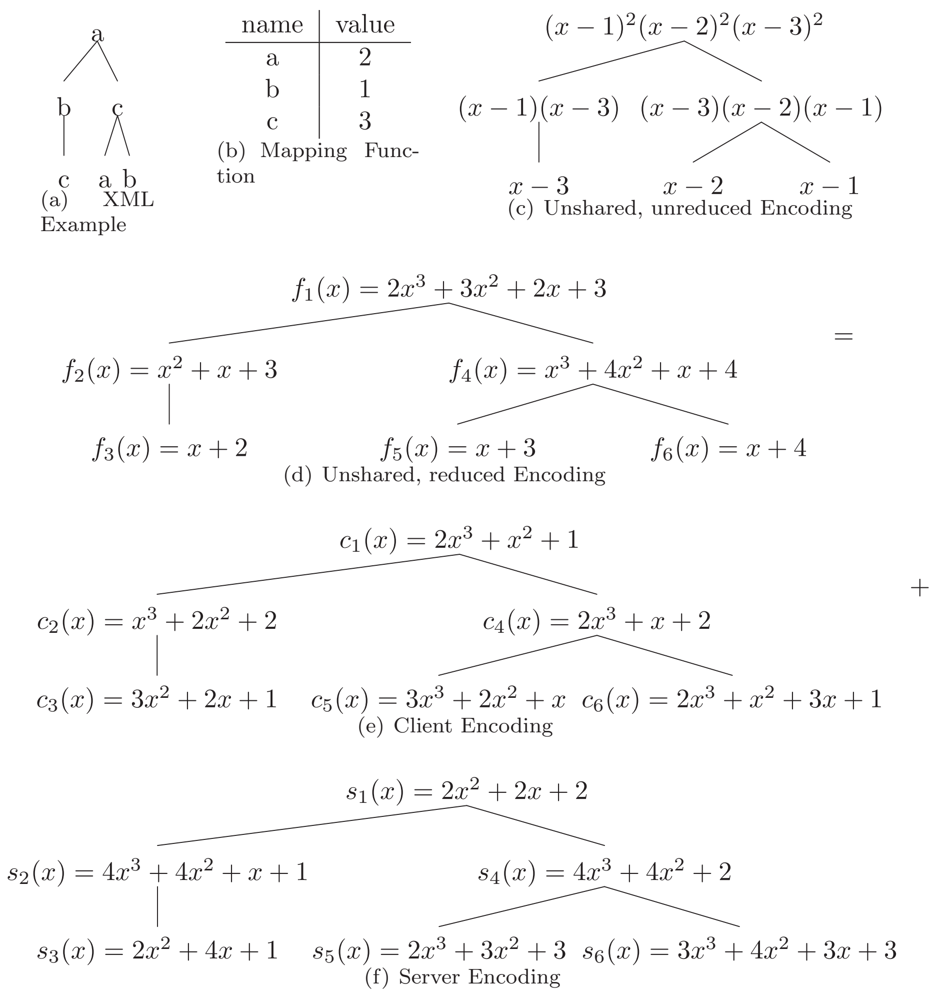

a
b
c
c
a
b
(a) XML Example
name | value
a | 2
b | 1
c | 3
(b) Mapping Function
$(x-1)^2(x-2)^2(x-3)^2$
$(x-1)(x-3)$
$(x-3)(x-2)(x-1)$
$x-3$
$x-2$
$x-1$
(c) Unshared, unreduced Encoding
$f_1(x) = 2x^3 + 3x^2 + 2x + 3$
$f_2(x) = x^2 + x + 3$
$f_4(x) = x^3 + 4x^2 + x + 4$
$f_3(x) = x + 2$
$f_5(x) = x + 3$
$f_6(x) = x + 4$
(d) Unshared, reduced Encoding
=
$c_1(x) = 2x^3 + x^2 + 1$
$c_2(x) = x^3 + 2x^2 + 2$
$c_4(x) = 2x^3 + x + 2$
$c_3(x) = 3x^2 + 2x + 1$
$c_5(x) = 3x^3 + 2x^2 + x$
$c_6(x) = 2x^3 + x^2 + 3x + 1$
(e) Client Encoding
+
$s_1(x) = 2x^2 + 2x + 2$
$s_2(x) = 4x^3 + 4x^2 + x + 1$
$s_4(x) = 4x^3 + 4x^2 + 2$
$s_3(x) = 2x^2 + 4x + 1$
$s_5(x) = 2x^3 + 3x^2 + 3$
$s_6(x) = 3x^3 + 4x^2 + 3x + 3$
(f) Server Encoding

Figure 3.1: The mapping function (3.1(b)) maps each name of an input document (3.1(a)) to an integer. The XML document is first encoded to a tree of polynomials (3.1(c)) before it is reduced to the finite field $\mathbb{F}_5[x]/(x^4-1)$ (3.1(d)) and split into a client (3.1(e)) and a server (3.1(f)) part.

be chosen arbitrarily. For our example we choose the mapping function displayed in figure 3.1(b). The mapping function should be private to avoid the server to see the query (see section 3.3.2).

2. The tree of XML elements (figure 3.1(a)) is represented as a tree of polynomials (figure 3.1(c)). The tree is built from the leaves up to the root node. A leaf node X is translated to the monomial $x - map(X)$. Every non-leaf node is calculated as the product of the polynomials of all its children times its own monomial.

   The following function maps every XML tag to a polynomial:

$$f(node) = \begin{cases} x - map(node) & \text{if } node \text{ is a leaf node} \\ (x - map(node)) \prod_{d \in child(node)} f(d) & \text{otherwise} \end{cases} \tag{3.1}$$

   The polynomials are stored in a tree (figure 3.1(c)) which has the same structure as the XML tree (figure 3.1(a))

3. To avoid large degree polynomials we will work in the finite ring $\mathbb{F}_q[x]/(x^{q-1} - 1)$, where q is a prime power $q = p^e$. For the reader's convenience, all proofs will be given for q prime. The coefficients of the polynomials are reduced modulo q. If p is prime then $\forall a \in \mathbb{F}_p^* : a^{p-1} \equiv 1 \pmod{p}$. Since these polynomials will only be used for evaluation in points of $\mathbb{F}_p[x]$, it makes sense to store the polynomials modulo $x^{p-1} - 1$. In effect, this means we are working in $\mathbb{F}_p[x]/(x^{p-1} - 1)$. In order to avoid zero divisors, we will avoid mapping a tagname to $p - 1$. Thus we reduce every polynomial to a polynomial of degree less than $p - 1$ with coefficients in $\mathbb{F}_p$.

   Although we calculate in a finite ring, no information about the original tag names is lost. We will prove this in theorem 3.3.4.

   Figure 3.1(d) shows the reduction to the finite ring $\mathbb{F}_p[x]/(x^{p-1} - 1)$ with $p = 5$.

4. This step will introduce the actual security. Uptil now all the steps were merely transformation from one encoding to another. In this step the tree from the previous step is split into a client (figure 3.1(e)) and a server tree (figure 3.1(f)). Both trees have the same structure as the original one. The polynomials of the client tree are generated by a pseudo-random bit generator. The polynomials of the server tree are chosen such that the sum of a client node and the corresponding server node equals the original

polynomial. Look for example to the top nodes of figure 3.1(e) and 3.1(f). The sum $(2x^3+x^2+1)+(2x^2+2x+2)$ equals the root node of figure 3.1(d) $(2x^3+3x^2+2x+3)$.

Note that this is a direct application of a basic secret sharing scheme (as is often used in secure multi-party computations). This can easily be extended to a model with multiple servers, in which the client together with n servers can reconstruct the shared secret polynomial.

5. Since the client tree is generated by a pseudo-random bit generator it suffices to store the seed on the client. The client tree can be discarded. When necessary, it can be regenerated using the pseudo-random bit generator and the seed value.

Before we can prove theorem 3.3.4 we need some lemmas.

Lemma 3.3.1 *If p is prime then $\prod_{i=1}^{p-1}(x-i) \equiv x^{p-1}-1 \pmod{p}$.*

Proof Let $f(x)=\prod_{i=1}^{p-1}(x-i)$ and $g(x)=x^{p-1}-1$. Two polynomials are the same if they have exactly the same roots with the same multiplicity. All elements of $\mathbb{F}_p^* = \{1,\ldots,p-1\}$ are roots of $f(x)$. By Fermat's little theorem, for p prime all these $p-1$ roots of $f(x)$ are also roots for $g(x)$. Thus the two polynomials are equal. □

Lemma 3.3.2 *Let p be prime and $f(x)\in\mathbb{F}_p[x]$. If $f(x)$ is non-zero mod $x-(p-1)$ then $f(x)$ is also non-zero modulo $x^{p-1}-1$.*

Proof From $f(x)\equiv 0 \pmod{x^{p-1}-1} \iff (x^{p-1}-1)|f(x)$ and from lemma 3.3.1 it follows that $x-(p-1)|x^{p-1}-1$ in $\mathbb{F}_p[x]$. From that we can conclude that $x-(p-1)|f(x)$ and thus also that $f(x)\equiv 0 \pmod{x-(p-1)}$. This proves that $f(x)\equiv 0 \pmod{x^{p-1}-1} \Longrightarrow f(x)\equiv 0 \pmod{x-(p-1)}$, which is equivalent to the statement of the lemma. □

Lemma 3.3.3 *Let p be prime, and let $f(x)\in\mathbb{F}_p[x]$ be defined as $f(x)=\prod_{i=1}^{p-2}(x-i)^{e_i}$, where $e_i\in\mathbb{N}$. Then $f(x)\not\equiv 0 \pmod{x^{p-1}-1}$.*

Proof Consider the evaluation of $f(x)$ at $p-1$:

$$f(p-1) = \prod_{i=1}^{p-2} ((p-1)-i)^{e_i}$$

Because $\forall i \in \{1, \ldots, p-2\} : i \neq p-1$, $f(p-1) \neq 0$. Thus $x-(p-1)$ cannot be a factor of $f(x)$, and we have that $f(x) \not\equiv 0 \pmod{x-(p-1)}$. By lemma 3.3.2 this implies that $f(x) \not\equiv 0 \pmod{x^{p-1}-1}$. □

Now we are ready to prove that the mapped values can be retrieved uniquely:

Theorem 3.3.4 *Given a polynomial $f(x)$ in $\mathbb{F}_p[x]/(x^{p-1}-1)$ (p prime) of an element* node *and all polynomials $(q_1, \ldots, q_n)$ of its children, the mapped value map(node) can be retrieved uniquely.*

Proof Because of the way the polynomial $f(x)$ of the element *node* was constructed, we know at least one solution exists for the equation

$$f(x) \equiv q_1(x) \cdots q_n(x)(x-t),$$

where t is the mapped value to be retrieved. To prove that the solution is unique, suppose there are two solutions t_1 and t_2 to this equation: $f(x) \equiv q_1(x) \cdots q_n(x)(x-t_1)$ and $f(x) \equiv q_1(x) \cdots q_n(x)(x-t_2)$. Then $q_1(x) \cdots q_n(x)(x-t_1) \equiv q_1(x) \cdots q_n(x)(x-t_2)$. This can be rewritten to

$$q_1(x) \cdots q_n(x)(t_1-t_2) \equiv 0 \pmod{p}.$$

Thus either $q_1(x) \cdots q_n(x) \equiv 0 \pmod{p}$ or $(t_1-t_2) \equiv 0 \pmod{p}$. Since we know that $q_1(x) \cdots q_n(x) \not\equiv 0 \pmod{p}$ by lemma 3.3.3 (the q_i's match the required form by construction), we can conclude that $t_1 \equiv t_2 \pmod{p}$. □

Note that the actual solution for t can easily be found by solving t in the equation $f(x) \equiv q_1(x) \cdots q_n(x)(x-t)$.

3.3.2 Retrieval

Now that the data has been shared on both the client and the server, we will describe how to query the data. First we will discuss simple element lookups: find an element given its tag name. In the second half of this section we will look at more difficult XPath queries.

Element lookup

We assume that the document of figure 3.1(a) has been shared as described in section 3.3.1. Let's further assume that we would like to evaluate the query `//c`. This XPath expression means that we want to find node elements tagged `c` somewhere in the tree. Normally (even in the non-encrypted case) this boils down to traversing the whole tree and comparing the tag names with the name `c`. We will do it smarter than that.

First we use the mapping function to translate the tag name `c` to $x = 3$ (see figure 3.1(b)). The client sends this value of x to the server. If we want to keep the query secret for the server the mapping function should be private to the client.

The server evaluates the polynomials in the given point ($x = 3$). Each time a polynomial has been evaluated the calculated value is sent back to the client (see figure 3.2).

The client does the same thing on its own side. Furthermore it calculates the sum of the client element and the server element. If this sum equals zero then the element contains a factor $(x - 3)$, meaning either that the element has tag name `c` or that it contains a descendant named `c`. A sum different from zero means that the branch is dead. If this is the case the client informs the server so that the server can stop evaluating polynomials for elements in the tree starting with that branch.

Each zero element in the sum tree that does not have a zero subelement represents an answer to the query. All other zero's in the sum tree may or may not represent correct answers. To find out whether the element itself or one of its descendants is named `c`, the non-shared polynomials of both the element and all its direct children have to be reconstructed.

To reconstruct the element value, let f be the sum of the polynomials on the server and the client of an element and $q_1, \ldots, q_n$ the combined polynomials of all its direct children.

By construction we know that f can be written as

$$f(x) = (x - t) \prod_{i=1}^{n} q_i(x) \pmod{p} \tag{3.2}$$

To check the correctness of an answer we have to solve t in $f(x) = 0$. In our example t should be 3.

Theorem 3.3.4 proves that there is just a single solution for t. It is solved by:

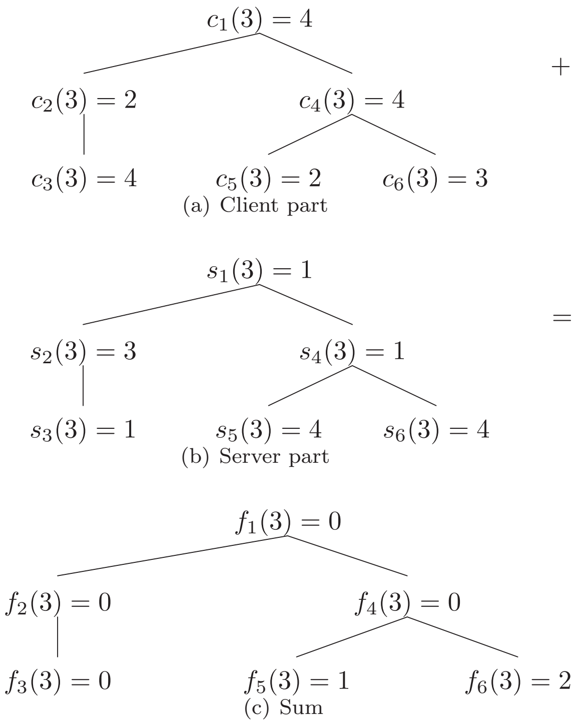


Figure 3.2: Query result for the query '$x = 3$'. Both the server and the client evaluate the polynomials for the given value of x in the finite ring $\mathbb{F}_p[x]/(x^{p-1} - 1)$. The server sends its values to the client which adds it to its own calculated value. A branch is a dead end if the sum is not 0.

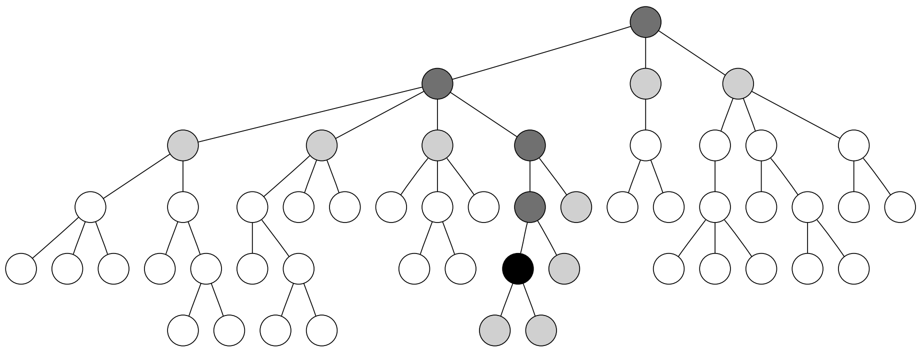


Figure 3.3: Execution path for a simple element lookup. The black node is the requested node. The grey nodes have been evaluated. The dark grey nodes resulted in a zero and the light grey nodes in a non-zero value. All the white nodes have been left untouched.

$$\begin{aligned} f(x) = 0 &\iff \\ (x-t)(q_1(x)\cdots q_n(x)) = 0 &\iff \\ a_{p-1}x^{p-1} + a_{p-2}x^{p-2} + \cdots + a_1 x + a_0 = 0 & \end{aligned} \tag{3.3}$$

Where each a_i is a linear function in t.

Equation (3.3) can be rewritten as

$$\begin{cases} a_{p-1}(t) = 0 \\ \cdots \\ a_0(t) = 0 \end{cases} . \tag{3.4}$$

A single (non-trivial) equation in (3.4) is enough to solve t. The other equations may be used to verify the result. Remember that we did not trust the server. We now have at least a way to check the answer. If, however, we trust the server to give correct answers, only the last equation is enough. In that case only the constant factor (without x) of each polynomial stored on the server has to be transmitted. This reduces bandwidth and increases efficiency but decreases security.

Figure 3.3 shows a typical evaluation path for a simple element lookup. The tree shows which nodes should be evaluated. All the white nodes does not have to be touched.

Advanced querying

So far we evaluated only queries like `//tagname`. But also more elaborate XPath queries can be performed. It is of course possible to evaluate a query like `//a/b//c/d/e` from left to right. That is, search the tree for occurrences of '`a`', then search within the found branches for '`b`', etc. But it is more efficient to evaluate the whole query at once. Since every polynomial in the tree consists of the roots of all its descendants, a single query can find all elements that contain the elements `a`, `b`, `c`, `d` and `e` (in any order). In this case a search consists of the following steps:

1. from the root node find all '`a`' elements that have `b`, `c`, `d` and `e` elements somewhere deeper in the tree
2. from the found nodes find all direct children '`b`' that have elements `c`, `d` and `e` as descendants
3. etc.

Using this strategy elements are filtered out in a very early stage and therefore the efficiency is increased.

In a real query evaluation you start at the XML root node and walk downwards until you encounter a dead branch. Whether you choose to traverse the tree depth- or breadth-first, the strategy remains the same: try to find dead branches as early as you can. Fortunately, each node contains information about all the subnodes. Therefore, it's almost always the case that you find dead branches (where the unshared evaluation returns a non-zero value) before reaching the leaves.

To illustrate the search process we will follow the execution run with the example query `//c/a`. This XPath query should be read as: start at the root node, go 1 or more steps down to all `c` nodes that have an `a` node as child. The roman numbers in figure 3.4 correspond to the following sequence of operations:

(*i*) We start the evaluation process at the root nodes of the server and the client. In parallel, they can substitute the values in the root polynomials. Both $s_1(map(\mathtt{c})) = s_1(3)$ and $s_1(map(\mathtt{a})) = s_1(2)$ should be evaluated, but it does not matter in which order (analogously for $c_1(\cdot)$). To mislead the server we choose to evaluate first the `a` nodes and then the `c` node, although the query suggests otherwise.

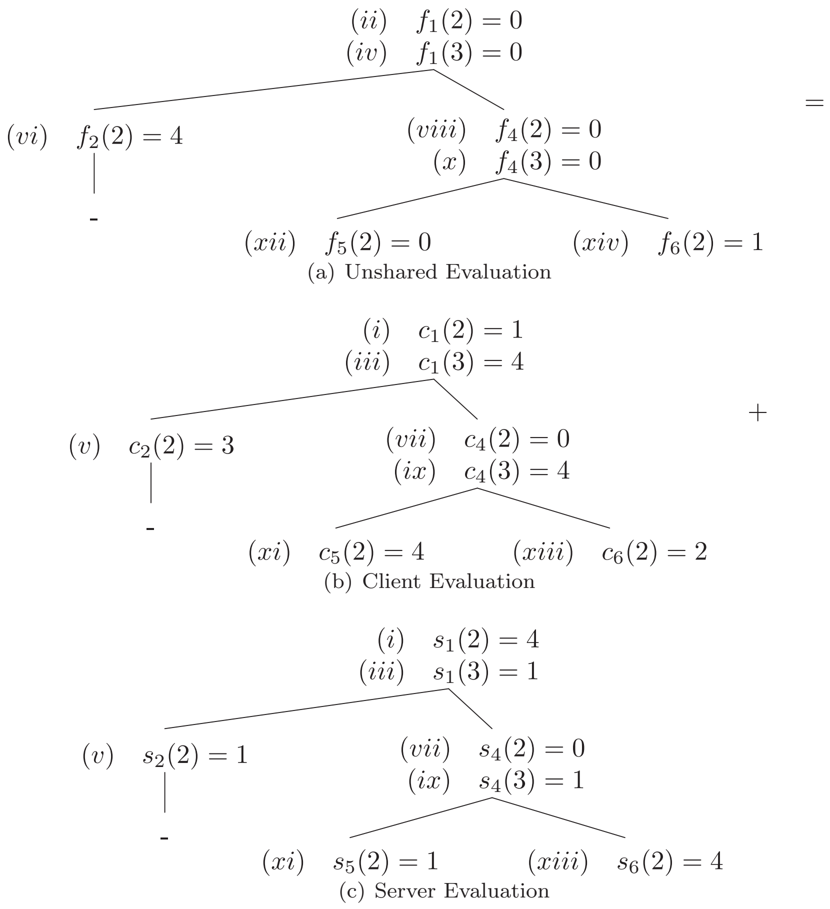


Figure 3.4: Evaluation process of the query `//c/a` using the same mapping function and data encoding as in figure 3.1. The Roman numbers indicate the sequence of operations.

(*ii*) Each time the server has substituted a value for x in one of its polynomials, it sends the result to the client, who can add the server result to its own. In this example $f_1(2) = c_1(2) + s_1(2) = 1 + 4 \equiv 0 \pmod 5$, which means that either the original root node was `a` or the root node has a descendant `a`.

(*iii*) Next thing to do is check that the root node is or contains `c`.

(*iv*) $f_1(3) = 0$. Now we know that the root node contains both `a` and `c`, a prerequisite of our query. Thus, we proceed one step down in the tree.

(*v*) The left child is checked for `a`.

(*vi*) This time $f_2(2) = 4 \neq 0$. Thus the left subtree does not contain an `a` node. Apparently this is a dead branch. It is not even necessary to check for a `c` node; the query `//c/a` can never hold in this branch. We can stop evaluating it and backtrack to the right subtree.

(*vii*) In the right subtree we start checking for a `c` node.

(*viii*) Since $f_4(2) = 0$, the right subtree seems promising.

(*ix*) Therefore we check also for an `a` node.

(*x*) The right tree still seems promising so we walk one level down.

(*xi*) Since the client knows the structure of the tree (if not, he can ask the server for it), he knows that we have reached a leaf node. Therefore, it is unnecessary to check for a `c` node.

(*xii*) Since this is a leaf node and $f_5(2) = 0$ we now know for sure that node 5 *is* an `a` node.

(*xiii*) The rightmost leaf node is also checked for an `a` node.

(*xiv*) But it is not.

Up till now, we have two possible matches:

1. node 1 matches `c` and node 4 matches `a`
2. node 4 matches `c` and node 5 matches `a`

It is sufficient to check the exact value of node 4 only. If this node *is* a `c` node then solution 1 holds, if this node *is* an `a` node solution 2 holds. If it is neither, then there are no matches. The exact value of a node n can be found in two different ways:

- Ask the server for the polynomial $s_n(x)$ and the polynomials of all its children (let us name them $s_n^{(1)}(x), \ldots, s_n^{(k)}(x)$). In the meantime calculate $c_n(x)$ and its children $c_n^{(1)}(x), \ldots, c_n^{(k)}(x)$. The exact value can be calculated by dividing $f_n(x)$ by $\prod_{i=1}^{k} f_n^{(i)}(x)$. The result will be a monomial $x - t$ where t is the node's value.
- If $f_n(a) = 0$ for some value a and for all children i of n, $f_i(a) \neq 0$ then you know that node n *is* a. Note that for recursive Document Type Definitions (such as our example) there is no guarantee that this method works.

3.3.3 *Trie* enhancement

The approach sketched in section 3.3.1 is only efficient when p^e is small. This is no problem for tag names that are chosen from a fixed sized set (described in a DTD), but cannot be used for the data because the number of different data nodes is unbounded. And since each polynomial takes $(p^e - 1)\log_2 p^e$ bits of storage space, it is important to keep p^e as small as possible.

In this chapter we propose a representation of XML documents allowing for efficient searching in data nodes. Basically, all data nodes are transformed to their *trie* representation [22].

A data string in the original XML document is translated to a path of nodes where each node is chosen from a small set. Assume this set contains $a, b, \ldots, z$. With this set we can translate the tree shown in figure 3.5(a) to an equivalent *trie* 3.5(b) or an uncompressed *trie* 3.5(c). An uncompressed *trie* stores exactly the same information as the original data string, whereas the compressed *trie* loses the order and cardinality of the words. If this is a problem an encryption of the data string may be added to the node. In this example we first split a string into words, represented by paths, and then each path is split into several characters. Other ways of splitting the string into nodes are possible.

On average removing duplicate words from a text reduces the size by 50%. Reducing a text into a compressed *trie* reduces the size by 75-80%. However each node is converted into a polynomial of size $(p^e - 1)\log_2 p^e$ bits. In case $p = 29$, a polynomial costs 17 bytes. Due to the *trie* compression the 'encryption' of a single letter will cost approximately $3\frac{1}{2} - 4\frac{1}{2}$ bytes.

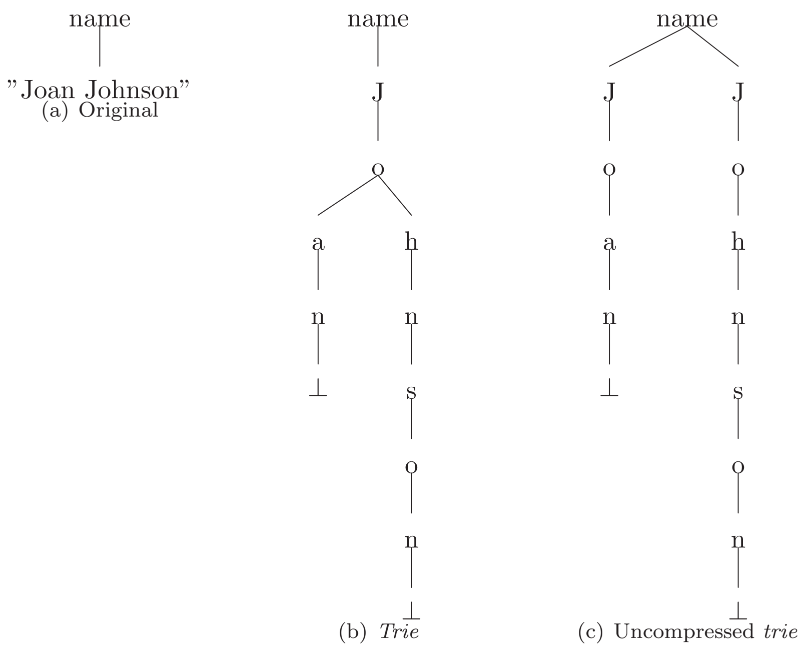


Figure 3.5: Transformation of an XML document tree into either a compressed or an uncompressed *trie*.

Having translated the original XML tree into a (compressed) *trie*, the same strategy as in section 3.3.1 can be used to encode the document. Like the document, also the queries should be pre-tuned to the new scheme. A query like

`/name[contains(text(), "Joan")]`

is first translated to

`/name[//J/o/a/n]`

before it is translated to

/*map*(`name`)`[//`*map*(`J`)`/`*map*(`o`)`/`*map*(`a`)`/`*map*(`n`)`]`.

Simple regular expressions like `.` and `.*` can be mapped to their *trie*-equivalents `*` and `//`.

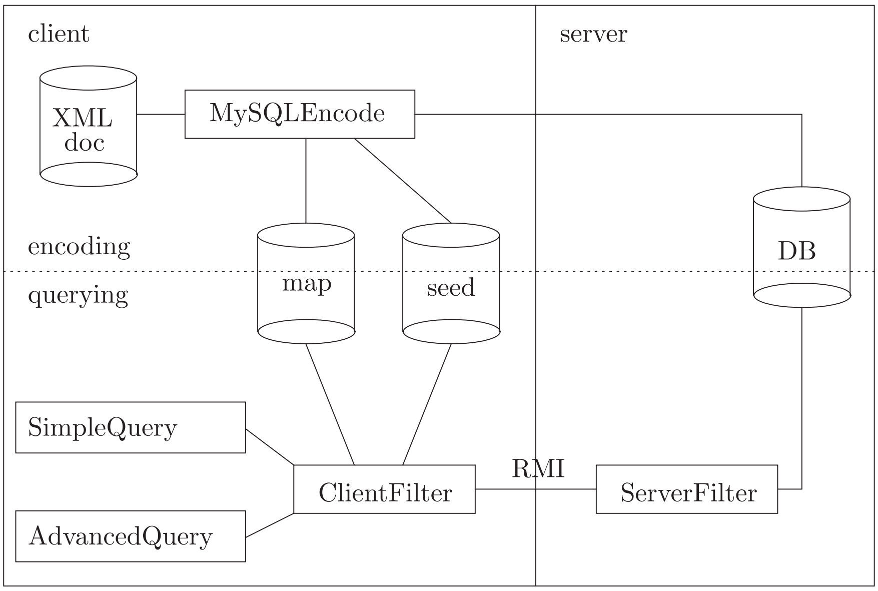


Figure 3.6: Client/Server Architecture.

3.4 Implementation

In the previous sections we described our theory of searching in encrypted data by using secret sharing and a special kind of encoding/encryption. To demonstrate that searching in encrypted data is not only possible in theory, but also in practice, we have built a prototype implementing the encoding and search strategy described in section 3.3.

The implementation is written in Java and set up using a client/server model. Figure 3.6 shows the architecture. We will elaborate on each component in the following sections.

The server stores all the polynomials in a database. The database is not protected and can be considered publicly readable. However, the client encodes the original plaintext XML document into encoded polynomials by using the `MySQLEncode` class. The encoder needs a private seed and a private map file which will be re-used by the query engines. The map file is just a text file which stores the mapping between tag names and corresponding values from $\mathbb{F}_{p^e}$.

The prototype consists of two different query engines: `SimpleQuery` and

`AdvancedQuery`. Both engines share the same filtering technique. The filter is distributed over the client and the server. The filter classes perform basic operations like function evaluation and tree reconstruction.

3.4.1 MySQLEncode

Since the server should not learn the information it is storing, it is the client's responsibility to fill the database.

The `MySQLEncode` class acts on three files which are provided on the command-line:

1. A map file
2. A seed file
3. The original XML document

The map file is a property file where each line is of the form $name = value$, where $name$ is one of the tag-names as specified by the DTD or XML schema and $value \in \mathbb{F}_{p^e}$ is the value it is mapped to.

The seed file acts as the encryption key and should therefore be kept secure. Without the seed file it is impossible to regenerate the client tree, and without the client tree the data on the server is meaningless.

The original XML document is parsed by a SAX parser[1]. This means that there is no need for a big client machine with lots of memory. This fits nicely into our philosophy of small clients (cell phones, for example) and big servers. The parser linearly reads the document and constructs the tree on the fly. It only needs memory proportional to the depth of the tree. The tree structure is stored by adding *pre*, *post* and *parent* values to each polynomial. The *pre* and *post* fields are sequence number that count the open tags respectively close tags. The *parent* fields refers to the *pre* value of its parent. This is a common way to store a tree structure into a flat relational table [2, 28]. In our prototype we use MySQL[2] as the database back-end. In order to speed up the search process the *pre*, *post* and *parent* fields are indexed by a B-tree.

3.4.2 The filter implementation

Each different query engine (see section 3.4.3) will use the same set of basic operations. These operations are offered by `ServerFilter` and `ClientFilter`.

[1] `www.saxproject.org/`

[2] `www.mysql.com`

Both classes implement a common interface `Filter` but are adapted to work on the server site respectively the client site. The two objects communicate with each other using Java's Remote Method Invocation (RMI). The operations consist of functions to query the tree structure as well as to evaluate the polynomials. `ServerFilter` will evaluate the polynomials stored in the database for the given values. `ClientFilter` first regenerates the client polynomial by using the pseudo-random bit generator with the secret seed and the *pre* location of the polynomial. After the evaluation of its generated polynomial it will add the result to the retrieved value from the server. Only when the sum equals zero, the location is returned to the invoking query engine, otherwise the next candidate node is generated/retrieved, evaluated and added together.

With the evaluation method only the *containment* of a node in a subtree is tested. To be sure that the node is *equal* to the root of the subtree there is an option to check the first factor of a node. Let $children(f)$ be a function that retrieves the set of polynomials representing all the children of the node represented as the polynomial f. To retrieve the factor $(x - t)$ in $f(x) = (x - t) \prod_{c \in children(f)} c(x)$ it is necessary to reconstruct the node's polynomial and all its child polynomials. Because the equality test is expensive it should only be invoked when absolutely necessary.

The operator `nextNode()` acts as a pipeline. The thin client only needs to have one node in memory at a time. The big server will do the buffering of the intermediate results.

3.4.3 Query engines

Since it was not a priori clear which search strategy is the best, we have decided to implement two query engines, called `SimpleQuery` and `AdvancedQuery`, each using a different search strategy, as explained below.

SimpleQuery

The most simple search strategy parses the XPath[3] query into steps where each step consists of a direction (child (`/`) or descendant (`//`)) and a tag name. Two special tag names exist: `..` matches the parent and `*` matches every child.

In this example we make use of the containment test only. In section 3.5 we will also use the equality test. There we will compare the two tests to see whether one is preferable to the other. We will sketch the algorithm by using an XML document generated by the XMark benchmark [43] and the example

[3] `www.w3.org/TR/xpath`

query `/site/*/person//city`. See appendix 3.7 for the DTD. This query is parsed into the following steps:

`/site`
: The first slash instructs the search engine to locate the root node (i.e. the only node without a parent (parent=0)). Since the parent field is indexed this is done in constant time. After the root node has been located both the stored polynomial on the server and the generated polynomial on the client are being evaluated at *map*(*site*). Only when the sum equals zero the next steps are carried out.

`/*`
: At this point the preliminary result set (implemented as a `Queue` on the server) will consist of only a single element. This step will change the result set into all children of the root node (i.e. regions, categories, catgraph, people, open_auctions and closed_auctions). The `*` reduces the workload because no additional filtering is needed.

`/person`
: All children of the 6 nodes in the result set are being examined in this step. Evaluation at *map*(*person*) is done for all the polynomials found. Only those nodes for which the sum of the server and client evaluations equals zero remain in the result set.

`//city`
: This step is quite expensive in terms of execution time. The result of the previous step is already quite large and this step even increases the number of possible nodes that have to be checked. All the descendants of the person-nodes (i.e. name, emailaddress, phone, address, homepage, creditcard, profile, watches, street, city, country, province, zipcode, interest, education, gender, business, age, watch, category, open_auction and description) have to be checked against *map*(*city*).

AdvancedQuery

The `AdvancedQuery` takes the tree as the starting point and parses it from root to leaf nodes. In contrast to the `SimpleQuery` the whole remaining query is taken into account at each step. We take advantage of the fact that nodes have knowledge of all descendants. This way it is possible to identify dead branches early in the search process at the cost of more evaluations for each node.

For easy comparison we use the same query and the same test (containment) as before.

`/site/*/person//city`

The `AdvancedQuery` engine always starts at the root node. This node is checked against *map*(*site*), *map*(*person*) and *map*(*city*). Only when all three sums are zero the next steps are carried out. Note that we can only check for the existence of a node. The structure of the query cannot be taken into account since the nodes don't store the structure of the subtree.

`/*/person//city`

The engine proceeds by consuming the `/site` part of the query and traversing the tree one step down to find the root's children. This unfiltered set of nodes are regions, categories, catgraph, people, open_auctions and closed_auctions. After filtering only the people, open_auctions and closed_auctions remain; all the other nodes do not contain person or city nodes. Thus we may skip these branches.

`/person//city`

In this step the `/*` has been removed. This means we traversed the tree one step downwards. The children of people, open_auctions and closed_auctions are person, open_auction and closed_auction. Because open_auction and closed_auction contain person and city nodes they remain in the result set even after filtering. The implementation does not check if the node *is* a person but if it *contains* it. This is done because we chose to use the containment test instead of the equality test. In section 3.5 we investigate whether this was a good choice or not.

`//city`

From the person, open_auction and closed_auction nodes we interactively walk downwards in the tree evaluating the polynomials at *map*(*city*) until this results in a non-zero sum. The result set now contains all nodes having a city inside. If we had chosen the equality test only the city nodes would have been in the result set.

3.5 Experiments

The goal of the prototype is to perform experiments with it. With the experiments described in this section we would like to find out what the practical impact of our encrypted database scheme is. We investigated the storage space overhead (section 3.5.1), the influence of the different search engine algorithms (section 3.5.2) and the difference between the equality and containment tests

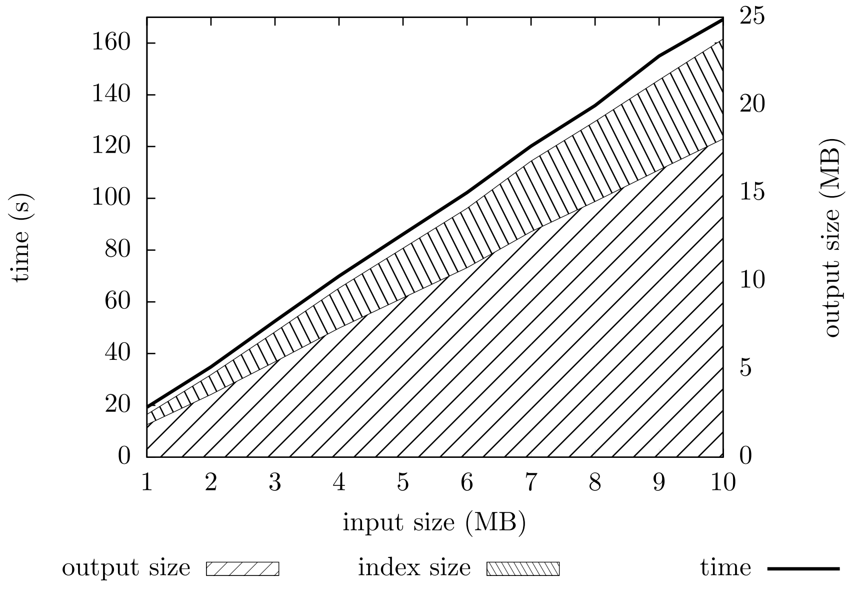


Figure 3.7: Encoding.

(section 3.5.3). All experiments act on an auction database synthesized by the XMark benchmark [43]. The DTD (see appendix 3.7) contains 77 elements. We chose $p = 83$ and $e = 1$ throughout this section.

3.5.1 Encoding

Encoding an XML document as polynomials requires extra storage space. This is due to the fact that each polynomial not only stores the information of its own node but also of all its descendants. Figure 3.7 plots the encoded database size against the input XML size. Approximately 17% of the output size is caused by the pre, post and parent values (not plotted in the figure). The remainder is thus approximately 1.5 times the size of the input. To speed up the search process we added indices to the pre, post and parent fields using B-trees. The size of these indices is added on top of the output size. As expected both the storage space and the encoding time are strictly linear in the input size.

1. `/site`
2. `/site/regions`
3. `/site/regions/europe`
4. `/site/regions/europe/item`
5. `/site/regions/europe/item/description`
6. `/site/regions/europe/item/description/parlist`
7. `/site/regions/europe/item/description/parlist/listitem`
8. `/site/regions/europe/item/description/parlist/listitem/text`
9. `/site/regions/europe/item/description/parlist/listitem/text/keyword`

Table 3.1: Queries with increasing length. The numbers correspond to figure 3.8.

3.5.2 Query Engines

One of the main reasons for building the prototype was that it was not a priori clear what the most efficient query engine algorithm is. Is it best to evaluate a polynomial at as many points as possible at each node to find an early dead branch or should one evaluate at a single point at a time? To answer this question we performed two tests: one with the simplest of all queries at increasing length and one with more advanced queries containing `//` and `*`.

The first test is the worst case scenario for the advanced query engine. The queries in table 3.1 are chosen in such a way that there is no gain for the advanced algorithm. For instance it is a waste of effort to check whether a europe node contains an item, description, parlist, listitem, text and keyword node, because the DTD (see appendix 3.7) dictates it to be always the case.

As can be seen in figure 3.8, where the number of evaluations is plotted against the queries of increasing length shown in table 3.1, the two search algorithms are comparable. They differ by at most a constant factor.

The second test with queries containing `//` and `*` was performed in conjunction with the strictness test. The test results are given in the next section.

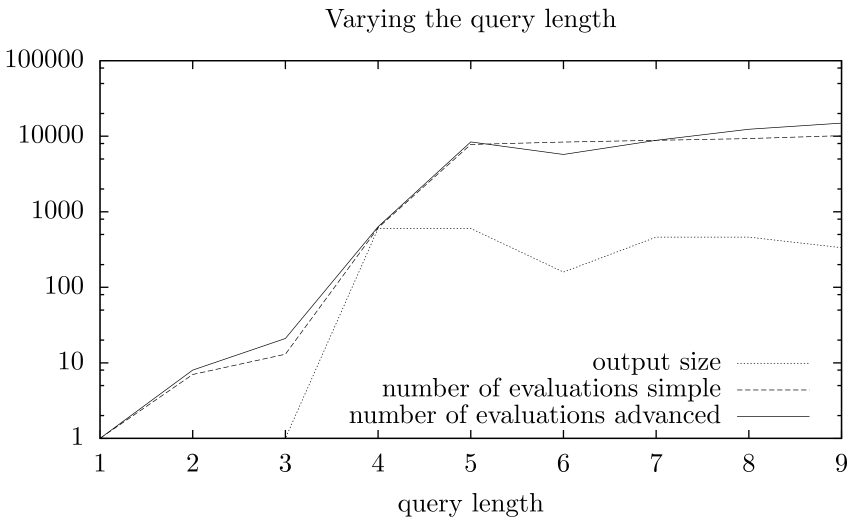


Figure 3.8: Several queries with increasing query length. The query numbers refer to the queries summed up in table 3.1.

1. `/site//europe/item`
2. `/site//europe//item`
3. `/site/*/person//city`
4. `/*/*/open_auction/bidder/date`
5. `//bidder/date`

Table 3.2: Queries for the strictness checks. The numbers correspond to figure 3.9.

3.5.3 Strictness

Another aspect that is hard to predict is the difference between the equality test and the containment test. On the one hand, it can be argued that, since the reconstruction of the first factor of a polynomial is computationally more expensive than a simple function evaluation, it is preferable to use the containment test. On the other hand, the reduced accuracy causes more nodes to be examined. Therefore we used our prototype to compare the two tests using both search algorithms.

For each query in table 3.2 four experiments were performed. Each algorithm (simple and advanced) was run twice: once with the equality test (strict checking) and once with the containment test (non-strict checking). The results are plotted in figure 3.9. For all queries the advanced algorithm outperforms the simple algorithm. Furthermore, it can be noticed that sometimes the strict checking pays off and sometimes it does not. In general, the equality test may cause a slight overhead or a major improvement.

Of course it is unfair to compare the equality test, which always gives the exact answer, with the containment test without considering the accuracy. Figure 3.10 shows the accuracy of the containment test. It plots the percentage of the nodes in the containment test's result that also pass the equality test. Notice that the accuracy drops for each `//` in the query. For absolute queries which do not contain `//`, the accuracy of the containment test reaches 100%.

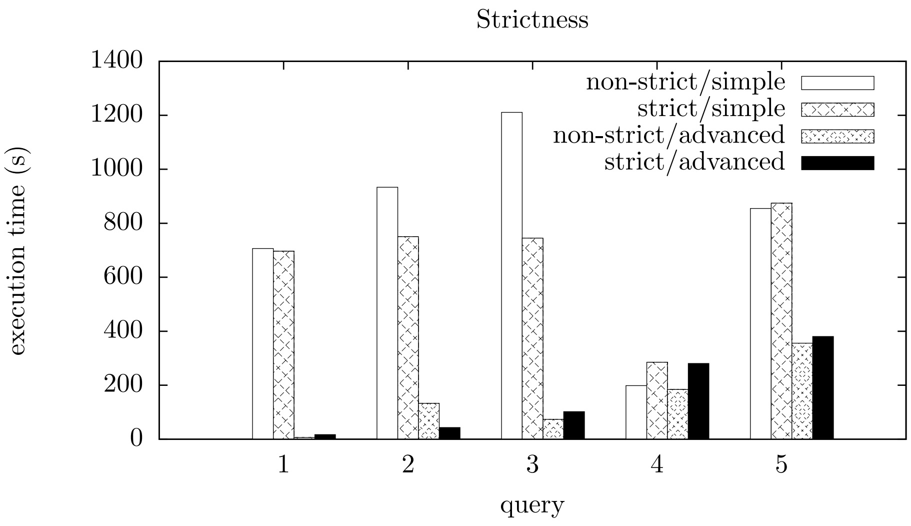


Figure 3.9: Equality test versus containment test.

3.6 Conclusions and future work

We have developed a method to store a tree of XML elements as a tree of polynomials, where the polynomials reside in the finite ring $\mathbb{F}_q[x]/(x^{q-1} - 1)$, where q is a prime power (i.e. $q = p^e$ for some prime p and integer e). This tree of polynomials is split in a server and a client part. Both parts are needed to retrieve the original data. The created trees can be used to query the data in a secure way. Our scheme has only a small penalty in storage space compared to the unencrypted case. To store an XML tree with n elements and q different tagnames in an unencrypted way we need a storage space in the order of $n \log q$. In the encrypted case the storage space is $n(q - 1) \log q$.

The extra amount of storage space is used as a smart index which enables an efficient search strategy. Each element has some knowledge of its descendants. When searching the tree for an element, a branch can be marked as a dead-end in a very early stage. Thus, only a small portion of the tree has to be examined.

Although more storage space is used than the information theoretic minimum, the storage space is 50% less (measured with our prototype (using $p = 83$ and $e = 1$)) than the textual XML document. The mapping function acts as a

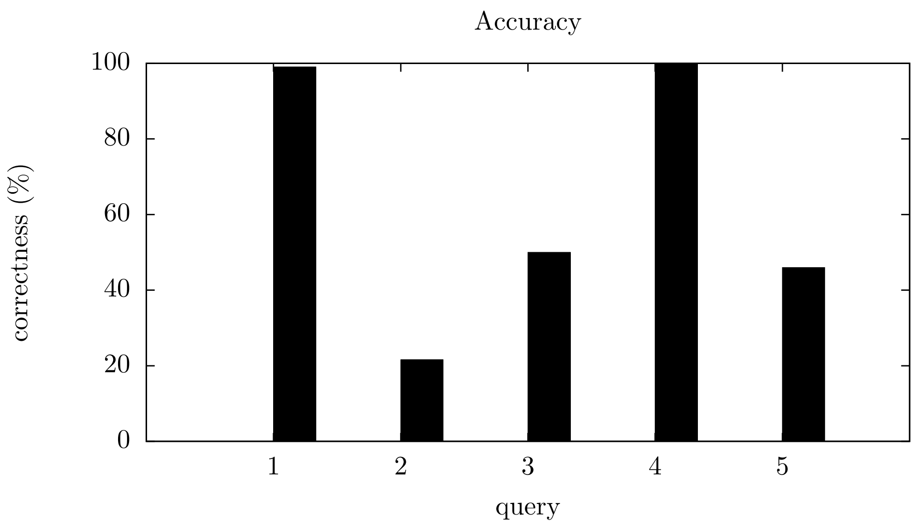


Figure 3.10: Accuracy of the containment test as defined by the quotient $\frac{E}{C}$, where E is the size of the result set using the equality test and C is the size of the result set using the containment test.

compression function. Also, it is necessary to store both the start and the end tag in our encoding. The encoding time is linear in the size of the input.

The prototype can choose between two different search algorithms. The simple algorithm reads a query from left to right carrying out a single evaluation at each node. The more advanced algorithm uses a look-ahead strategy where the whole remaining query is taken into account. Experiments show that the advanced algorithm outperforms the simple algorithm in the majority of cases. Only for the most simple queries it is slightly slower.

The search algorithms can use two comparison tests: the equality test and the containment test. The containment test is just a cheap evaluation whereas the equality test is more expensive because a node's own polynomial should be divided by all its child polynomials. The cost of a single equality test depends on the number of children, whereas the costs of a containment test is always constant. All the child nodes should be retrieved from the server and added to the pseudo-randomly generated client polynomials. The accuracy of the containment test is reasonable but it does not result in a major improvement in the running time. On the contrary, it is often better to use the equality test to reduce the number of nodes to check, especially for the simple algorithm.

Using a *trie* to represent data content enables querying of the data inside the XML tags. The *trie*-representation is not yet part of the current prototype but we expect a major improvement especially in the advanced algorithm. Queries over the data are more precise than those over the tag labels and thus the number of nodes to be examined is being reduced. Since knowledge of the data is present at high level nodes, the query engine can find the path to the answer almost immediately.

3.7 Appendix: XMark's auction DTD

```
<!ELEMENT site           (regions, categories, catgraph, people,
                          open_auctions, closed_auctions)>
<!ELEMENT categories     (category+)>
<!ELEMENT category       (name, description)>
<!ELEMENT name           (#PCDATA)>
<!ELEMENT description    (text | parlist)>
<!ELEMENT text           (#PCDATA | bold | keyword | emph)*>
<!ELEMENT bold           (#PCDATA | bold | keyword | emph)*>
<!ELEMENT keyword        (#PCDATA | bold | keyword | emph)*>
<!ELEMENT emph           (#PCDATA | bold | keyword | emph)*>
```

```
<!ELEMENT parlist         (listitem)*>
<!ELEMENT listitem        (text | parlist)*>
<!ELEMENT catgraph        (edge*)>
<!ELEMENT edge            EMPTY>
<!ELEMENT regions         (africa, asia, australia, europe,
                           namerica, samerica)>
<!ELEMENT africa          (item*)>
<!ELEMENT asia            (item*)>
<!ELEMENT australia       (item*)>
<!ELEMENT namerica        (item*)>
<!ELEMENT samerica        (item*)>
<!ELEMENT europe          (item*)>
<!ELEMENT item            (location, quantity, name, payment,
                           description, shipping, incategory+,
                           mailbox)>
<!ELEMENT location        (#PCDATA)>
<!ELEMENT quantity        (#PCDATA)>
<!ELEMENT payment         (#PCDATA)>
<!ELEMENT shipping        (#PCDATA)>
<!ELEMENT reserve         (#PCDATA)>
<!ELEMENT incategory      EMPTY>
<!ELEMENT mailbox         (mail*)>
<!ELEMENT mail            (from, to, date, text)>
<!ELEMENT from            (#PCDATA)>
<!ELEMENT to              (#PCDATA)>
<!ELEMENT date            (#PCDATA)>
<!ELEMENT itemref         EMPTY>
<!ELEMENT personref       EMPTY>
<!ELEMENT people          (person*)>
<!ELEMENT person          (name, emailaddress, phone?, address?,
                           homepage?, creditcard?, profile?,
                           watches?)>
<!ELEMENT emailaddress    (#PCDATA)>
<!ELEMENT phone           (#PCDATA)>
<!ELEMENT address         (street, city, country, province?,
                           zipcode)>
<!ELEMENT street          (#PCDATA)>
<!ELEMENT city            (#PCDATA)>
<!ELEMENT province        (#PCDATA)>
```

```
<!ELEMENT zipcode         (#PCDATA)>
<!ELEMENT country         (#PCDATA)>
<!ELEMENT homepage        (#PCDATA)>
<!ELEMENT creditcard      (#PCDATA)>
<!ELEMENT profile         (interest*, education?, gender?,
                           business, age?)>
<!ELEMENT interest        EMPTY>
<!ELEMENT education       (#PCDATA)>
<!ELEMENT income          (#PCDATA)>
<!ELEMENT gender          (#PCDATA)>
<!ELEMENT business        (#PCDATA)>
<!ELEMENT age             (#PCDATA)>
<!ELEMENT watches         (watch*)>
<!ELEMENT watch           EMPTY>
<!ELEMENT open_auctions   (open_auction*)>
<!ELEMENT open_auction    (initial, reserve?, bidder*, current,
                           privacy?, itemref, seller, annotation,
                           quantity, type, interval)>
<!ELEMENT privacy         (#PCDATA)>
<!ELEMENT initial         (#PCDATA)>
<!ELEMENT bidder          (date, time, personref, increase)>
<!ELEMENT seller          EMPTY>
<!ELEMENT current         (#PCDATA)>
<!ELEMENT increase        (#PCDATA)>
<!ELEMENT type            (#PCDATA)>
<!ELEMENT interval        (start, end)>
<!ELEMENT start           (#PCDATA)>
<!ELEMENT end             (#PCDATA)>
<!ELEMENT time            (#PCDATA)>
<!ELEMENT status          (#PCDATA)>
<!ELEMENT amount          (#PCDATA)>
<!ELEMENT closed_auctions (closed_auction*)>
<!ELEMENT closed_auction  (seller, buyer, itemref, price, date,
                           quantity, type, annotation?)>
<!ELEMENT buyer           EMPTY>
<!ELEMENT price           (#PCDATA)>
<!ELEMENT annotation      (author, description?, happiness)>
<!ELEMENT author          EMPTY>
<!ELEMENT happiness       (#PCDATA)>
```

Chapter 4

Exploring cryptographic extensions to PIR

Private Information Retrieval (PIR) aims at hiding a query to the database system. Although the server can read and understand the stored data, it cannot understand the query or the answer. In this chapter we explore possibilities to go one step further by encrypting the stored data too. The server should neither understand the stored data, the query nor the answer. This chapter explores the use of homomorphic encryption to accomplish this.

4.1 Introduction

Private Information Retrieval (PIR) deals with a similar problem as this thesis. PIR hides the query and the answer to a database server but leaves the stored data in the clear. In other words the server knows the data that it stores and knows who is querying, but not what he asks for.

In some situations the protection of only the query is sufficient. A good example of where PIR would be useful is to protect corporate research laboratories when connecting to a public patent database server. The patents should be publicly available (by law). Corporate research laboratories tend to keep their research activities secret for their competitors. A query for a specific patent leaks the interest for a particular technology. Therefore, a competitor should not be able to link a researcher to the patent he asks for. PIR solves this.

In other situations, however, also the stored data should be protected. This chapter investigates some possibilities to extend PIR with cryptographic techniques in order to make not only the query and the answer invisible for an attacker (including the server itself), but also the stored data. The data is encrypted with a homomorphic encryption function. Section 4.2 will summarise the most common homomorphic encryption functions. One of them, the Goldwasser-Micali scheme (section 4.2.3), forms the basis for a PIR scheme (section 4.3) which is used by most of our extensions.

In this chapter the database is simply a set of stored integer values. Using standard PIR, it is possible to ask the database for the value that is stored on a known location. The opposite query is not possible. If we know the value and want to know whether and where it is stored, standard PIR techniques cannot be used. Our extensions to PIR (section 4.4) aim at this second kind of queries.

4.2 Homomorphic encryption

Homomorphic encryption is a form of public key encryption with the property that one can perform an operation on the plaintext by performing a (possibly different) operation on the ciphertext, without using the decryption key. More precisely, an encryption function E is called homomorphic if there exist two (possibly the same) operations ($\oplus$ and $\otimes$), such that

$$E(a \oplus b) = E(a) \otimes E(b). \tag{4.1}$$

Several homomorphic encryption functions exist with different operators.

The rest of this section summarises the most famous ones. Except for RSA all the presented encryption methods are probabilistic, meaning that when two identical messages are encrypted with the same key, the corresponding ciphertexts will be different. This is a nice property and can be used to make a correlation between requests in the PIR scheme impossible.

4.2.1 RSA

RSA [42], which is named after its inventors Rivest, Shamir and Adleman, is one of the most famous public key encryption algorithms.

Key generation

1. Choose large prime numbers p and q.
2. Compute the modulus $n = pq$.
3. Compute the totient $\phi(n) = (p-1)(q-1)$.
4. Choose an integer e such that $1 < e < \phi(n)$ and coprime with $\phi(n)$ $(\gcd(e, \phi(n) = 1))$.
5. Compute d such that $de \equiv 1 \pmod{\phi(n)}$.
6. Publish public key (n, e) and keep private key d secret.

Encryption
: The encryption of a message m is $c = E(m) = m^e \bmod n$.

Decryption
: The ciphertext c is decrypted by calculating $c^d \bmod n = m^{ed} \bmod n = m$.

Homomorphic property
: $E(m_1) \cdot E(m_2) = m_1^e m_2^e \bmod n = (m_1 m_2)^e \bmod n = E(m_1 \cdot m_2)$.

4.2.2 ElGamal

ElGamal [20] is a public key encryption algorithm which is based on the Diffie-Hellman key agreement protocol [17].

Key generation

1. Generate a cyclic group $G = \langle g \rangle$ with order $q = |G|$.
2. Randomly choose $x \in_R \{0, \ldots, q-1\}$.

3. Compute $h = g^x$.
4. Publish public key (G, q, g, h) and keep private key x secret.

Encryption

1. Randomly choose $y \in_R \{0, \dots, q-1\}$.
2. The encryption of a message m is $c = (c_1, c_2) = E(m) = (g^y, m \cdot h^y)$.

Decryption
The ciphertext $c = (c_1, c_2)$ is decrypted by calculating

$$\frac{c_2}{c_1^x} = \frac{m \cdot h^y}{g^{xy}} = \frac{m \cdot g^{xy}}{g^{xy}} = m. \tag{4.2}$$

Homomorphic property
$E(m_1) \cdot E(m_2) = (g^{y_1}, m_1 \cdot h^{y_1}) \cdot (g^{y_2}, m_2 \cdot h^{y_2}) = (g^{y_1+y_2}, (m_1 \cdot m_2) h^{y_1+y_2}) = E(m_1 \cdot m_2)$.

4.2.3 Goldwasser-Micali

The encryption algorithm of Goldwasser and Micali [27] was the first probabilistic public key encryption algorithm. Although it is not very efficient (the ciphertexts are several hundred times larger than the plaintext), it is often used as a proof of concept.

Key generation

1. Choose large prime numbers p and q.
2. Compute $n = pq$.
3. Choose a quadratic non-quadratic residue $x \in \mathbb{Z}_n$ with Jacobi symbol $\left(\frac{x}{n}\right) = +1$. This means that the Legrendre symbols $\left(\frac{x}{p}\right) = \left(\frac{x}{q}\right) = -1$.
4. Publish public key (x, n) and keep private key (p, q) secret.

Encryption

1. Choose a random $y \in_R \{0, \dots, n-1\}$.
2. The encryption of a bit $m \in \{0, 1\}$ is $c = y^2 x^m \bmod n$.

Decryption

Using the factorisation of n it can easily be determined whether the ciphertext c is a quadratic residue ($m = 0$) or not ($m = 1$).

Homomorphic property

$E(m_1) \cdot E(m_2) = y_1^2 x^{m_1} \cdot y_2^2 x^{m_2} = (y_1 y_2)^2 x^{m_1+m_2} = E(m_1 \oplus m_2)$, where $\oplus$ is the addition modulo 2 (xor).

4.2.4 Paillier

Paillier's probabilistic public key encryption algorithm [40] is based on the composite residuosity assumption and is often used because of its additive homomorphic property.

Key generation

1. Choose large prime numbers p and q.
2. Compute the modulus $n = pq$ and $\lambda = \mathrm{lcm}(p-1, q-1)$.
3. Select a random integer $g \in_R \mathbb{Z}^*_{n^2}$.
4. Ensure that n divides the order of g by checking the existence of the multiplicative inverse $\mu = (L(g^\lambda \bmod n^2))^{-1} \bmod n$, where $L(u) = \frac{u-1}{n}$.
5. Publish the public key (n, g) and keep the private key (λ, μ) secret.

Encryption

1. Randomly choose $r \in_R \mathbb{Z}^*_{n^2}$.
2. The encryption of a message $m \in \mathbb{Z}_n$ is $c = g^m \cdot r^n \bmod n^2$.

Decryption

The ciphertext c is decrypted by calculating $L(c^\lambda \bmod n^2) \cdot \mu = m$.

Homomorphic property

$E(m_1) \cdot E(m_2) = (g^{m_1} \cdot r_1^n) \cdot (g^{m_2} \cdot r_2^n) = g^{m_1+m_2} \cdot (r_1 r_2)^n = E(m_1 + m_2)$.

4.2.5 Boneh-Goh-Nissim

The public key encryption algorithm of Boneh, Goh and Nissim [11] is currently one of the few encryption algorithm with both additive and multiplicative homomorphic properties. It allows multiple addititions and a single multiplication to be performed directly on the encrypted values.

Key generation

1. Choose two primes p and q.
2. Generate two multiplicative groups $\mathbb{G}$ and $\mathbb{G}_1$ of order $n = pq$ and a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_1$ such that for all $u, v \in \mathbb{G}$ and $a, b \in \mathbb{Z}$, we have that $e(u^a, v^b) = e(u, v)^{ab}$. It is also required that if g is a generator of group $\mathbb{G}$ then $e(g, g)$ is a generator of group $\mathbb{G}_1$.
3. Choose two random generators $g, u \in_R \mathbb{G}$.
4. Calculate the generator $h = u^q$ of a subgroup of $\mathbb{G}$ of order p.
5. Publish public key $(n, \mathbb{G}, \mathbb{G}_1, e, g, h)$ and keep private key p secret.

Encryption

1. Choose a random $r \in_R \{0, \ldots n - 1\}$.
2. The encryption of a message m is $c = g^m h^r \in \mathbb{G}$.

Decryption

To decrypt the ciphertext c first compute $c^p = (g^m h^r)^p = (g^p)^m = \hat{g}^m$ and then use Pollard's ρ-method [41] to calculate the discrete log to retrieve m.

Homomorphic property

Unlike other homomorphic encryption schemes, Boneh, Goh and Nissim support both an unlimited number of additions and a single multiplication. It is additive homomorphic in $\mathbb{G}$ because $E(m_1) \cdot E(m_2) = g^{m_1} h^{r_1} \cdot g^{m_2} h^{r_2} = g^{m_1+m_2} h^{r_1+r_2} = E(m_1 + m_2)$.

The bilinear map e is used for the multiplication. Let $c_1 = g^{m_1} h^{r_1}$ and $c_2 = g^{m_2} h^{r_2}$ two encryptions in $\mathbb{G}$. Further define $g_1 = e(g, g) \in \mathbb{G}_1$, $h_1 = e(g, h) \in \mathbb{G}_1$ and $r \in_R \mathbb{Z}_n$. The multiplication is then calculated as follows:

$$\begin{aligned}
c &= e(E(m_1), E(m_2))h_1^r = e(c_1, c_2)h_1^r \\
&= e(g^{m_1}h^{r_1}, g^{m_2}h^{r_2})h_1^r \\
&= e(g^{m_1+\alpha q_2 r_1}, g^{m_2 \alpha q_2 r_2})h_1^r \\
&= e(g, g)^{(m_1+\alpha q_2 r_1)(m_2 \alpha q_2 r_2)}h_1^r \\
&= g_1^{m_1 m_2 + \alpha q_2 (m_1 r_2 + m_2 r_1 + \alpha q_2 r_1 r_2)} h_1^r \\
&= g_1^{m_1 m_2} h_1^{m_1 r_2 + m_2 r_1 + \alpha q_2 r_1 r_2 + r} \\
&= g_1^{m_1 m_2} h_1^{\hat{r}} \\
&= E(m_1 m_2)
\end{aligned} \tag{4.3}$$

Note that the additive homomorphic property also holds for $\mathbb{G}_1$.

Both additive and multiplicative properties combined, result in a homomorphic encryption scheme that can calculate

$$E\left(\sum_j \left(\left(\sum_i x_{i,j}\right) \cdot \left(\sum_i y_{i,j}\right)\right)\right), \tag{4.4}$$

given the encryptions of the $x_{i,j}$'s and $y_{i,j}$'s. The second and third summations are performed within the group $\mathbb{G}$. With the multiplication we jump from $\mathbb{G}$ to $\mathbb{G}_1$. The leftmost summation is performed within the group $\mathbb{G}_1$. Equation (4.4) can be simplified by moving all the summations to $\mathbb{G}_1$ using the distributive property to

$$E\left(\sum_{i',j'} x_{i'} \cdot y_{j'}\right). \tag{4.5}$$

4.2.6 Domingo-Ferrer

The privacy homomorphisms of Domingo-Ferrer [18, 19] are both additive and multiplicative homomorph. Originally, they were designed to withstand known-plaintext attacks. However, they were succesfully attacked by Cheon and Nam [12] and by Wagner [49] in the known-plaintext scenario. They are still secure in the ciphertext-only scenario.

Key generation

1. Choose a positive integer $d > 2$ and a large integer n ($\approx 10^{200}$ or larger, having many small divisors).

2. Choose secret $r \in \mathbb{Z}_n$ (such that $r^{-1} \bmod n$ exists) and n' which is a small divisor of n.
3. Publish public key (n, d) and keep private key (r, n') secret.

Encryption

1. Randomly split the message m into secrets $m_1, \ldots, m_d$ such that $m = m_1 + \cdots + m_d \bmod n'$ and $a_i \in \mathbb{Z}_n$.
2. The encryption of message m is $c = (m_1 r \bmod n, \ldots, m_d r^d \bmod n)$.

Decryption

1. Multiply each of the coordinates with r^{-i} where i is the index, to retrieve $(m_1 \bmod n, \ldots, m_d \bmod n)$.
2. The decription is the sum $m_1 + \cdots + m_d \bmod n'$.

Homomorphic property

1. $E(a)+E(b) = (a_1 r_1 \bmod n, \ldots, a_d r_1^d \bmod n)+(b_1 r_2 \bmod n, \ldots, b_d r_2^d \bmod n) = ((a_1 + b_1)r' \bmod n, \ldots, (a_d + b_d)r'^d \bmod n) = E(a + b)$.
2. Multiplication works like in the case of polynomials: all terms are cross-multiplied in $\mathbb{Z}_n$. A d_1th degree term times a d_2th degree term yields a $d_1 + d_2$ degree term. Terms of equal degree are added together.

4.3 Private information retrieval

One of the applications of homomorphic encryption is Private Information Retrieval (PIR). In this section we give an example which uses the Goldwasser-Micali scheme (section 4.2.3) [39]. Goldwasser-Micali is the first probabilistic public-key encryption scheme which is secure under standard cryptographic assumptions and therefore often used as a proof of concept. It should be noted that more efficient solutions exists today.

Notation 4.3.1 *In this chapter a shorthand notation for a homomorphic encryption is being used. The homomorphic encryption of an element x is written as $\boxed{x}$, which should be read as $\boxed{x} \in_R E_k(x)$ (i.e. $\boxed{x}$ is a randomised encryption of x) for some public key k and encryption function E. Note that since almost all homomorphic encryption schemes are also probabilistic, it is not always the* *case that two encryptions of the same element are the same. Thus* $x = y \not\Rightarrow \boxed{x} = \boxed{y}$.

When a variable x *holds the encryption of a value* v *it will be written as the equality* $x = \boxed{v}$, *rather than the more cumbersome* $x \in_R \boxed{v}$.

Each database is essentially a list of bits. In this section we will group the bits to form m-bit values. We partition a database into an $m \times n$-matrix. Each column is an m-bit value.

$$D = \begin{pmatrix} d_{1,1} & \cdots & d_{1,i} & \cdots & d_{1,n} \\ \vdots & & \vdots & & \vdots \\ d_{m,1} & \cdots & d_{m,i} & \cdots & d_{m,n} \end{pmatrix}, \tag{4.6}$$

where $d_{k,j} \in \{0, 1\}$. This database D is stored in plaintext on the server.

To privately retrieve the ith column, the user creates a vector

$$q = \begin{pmatrix} q_1 \\ \vdots \\ q_n \end{pmatrix}, \tag{4.7}$$

where q_j is the tuple

$$q_j = \langle v_j, w_j \rangle = \begin{cases} \left\langle \boxed{0}, \boxed{1} \right\rangle & \text{if } i = j \\ \left\langle \boxed{0}, \boxed{0} \right\rangle & \text{if } i \neq j. \end{cases} \tag{4.8}$$

This vector of tuples is sent to the server. Since the server cannot distinguish $\boxed{0}$ from $\boxed{1}$ it does not learn which element is requested. The server replaces each value $d_{k,j}$ in the database by v_j if it is 0 and with w_j otherwise. The computed database then looks like

$$D' = \begin{pmatrix} \boxed{0} & \cdots & \boxed{0} & \boxed{d_{1,i}} & \boxed{0} & \cdots & \boxed{0} \\ \vdots & & \vdots & \vdots & \vdots & & \vdots \\ \boxed{0} & \cdots & \boxed{0} & \boxed{d_{m,i}} & \boxed{0} & \cdots & \boxed{0} \end{pmatrix}. \tag{4.9}$$

In the next step the server multiplies all elements of a row together. With the homomorphic property $\boxed{x} \cdot \boxed{y} = \boxed{x \oplus y}$ the encryption of the requested column can be calculated:

$$a = \begin{pmatrix} \boxed{d_{1,i}} \\ \vdots \\ \boxed{d_{m,i}} \end{pmatrix}. \tag{4.10}$$

This answer vector is sent to the user who can decrypt it with his private key.

4.4 Cryptographic extensions to PIR

Although PIR protects the query and the query result, it does not protect the stored data. In this section some extensions to PIR are being investigated. All those extensions have one thing in common: they all try to use techniques that are similar to PIR, but work on encrypted data instead of plaintext. Figure 4.1 shows all the extensions in a graph. The nodes represent the extensions and will be explained later in this section. An edge from node `A` to node `B` denotes that `B` fixes a problem that exists in solution `A`.

PIR leaves the stored data in the clear. Both the bitmap approach and the dual homomorphic approach encrypt the stored data. The drawback of the bit map approach is the size of the queries. Range queries and storing pre-loaded query vectors tackle this problem. Storing the query vectors on the server takes valuable space. Using stored query templates reduces the storage costs. Another problem of the stored query vector approach is that duplicate queries can be detected. Three techniques (replacement, shift and addition) can be used to refresh the stored queries.

The second branch in figure 4.1 consists of two techniques that do not use a bit map. One uses a dual homomorphic encryption scheme which is based on Domingo-Ferrer [18, 19]. The other uses a polynomial encoding.

In the rest of this section the proposed extensions to PIR will be explained in more detail.

For a database that consists of a set of integers, there are two kinds of queries:

1. 'Give me the data that is stored at this location'.

2. 'Tell me whether (and possibly where) this value is stored in the database'.

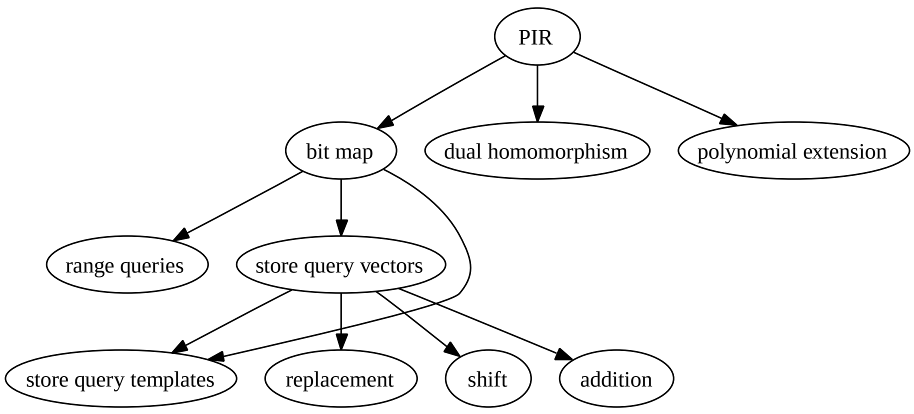


Figure 4.1: Research directions.

The first query can be answered by standard PIR systems even when the integers are encrypted. The answer will of course be the encrypted value which only the requestor can decrypt.

To answer the second query, several extensions are being proposed in this section. Each extension stores the same database D of n values. Each value $d_i \in \mathbb{Z}_N$ is unique.

The same notation for homomorphic encryption will be used as in 4.3.1.

4.4.1 Bit map

Our bit map extension is based on the PIR system that is described in section 4.3. That PIR system is only capable of answering the first kind of queries. We therefore transform the data of that system into a bit map. If the original database contains the set of values $D = \{d_0, \ldots, d_{n-1}\}$ (with $d_i \in \mathbb{Z}_N$), it is possible to encode this with the bit map $\hat{D} = \{\hat{d}_0, \ldots, \hat{d}_{N-1}\}$ (with $\hat{d}_i \in \{0,1\}$), such that $d_i \in D \iff \hat{d}_{d_i} = 1$. In other words the bitmap $\hat{D}$ has one bit for every element in $\mathbb{Z}_N$. If the bit at location i is 1 this means that the value i is in the database and 0 means that it is not.

This bit map can easily be encrypted by any semantically secure encryption algorithm. Any algorithm of section 4.2 except RSA (which is not semantically secure) will do. The algorithm is required to be semantically secure because otherwise there would only be two values in the encryption of $\hat{D}$: the encryption of 1 and the encryption of 0, which would make it very simple for an attacker to guess (with 50%-probability) which elements are in D and which ones are not. The algorithm is not required to have a homomorphic property.

Let $\boxed{\cdot} : \{0,1\} \to \{0,1\}^m$ be a semantically secure encryption algorithm, where m is an integer that depends on the chosen homomorphic encryption function. The encrypted database is simply the list of all the encrypted bits:

$$\boxed{\hat{D}} = \left\{ \boxed{\hat{d}_0}, \ldots, \boxed{\hat{d}_{N-1}} \right\}. \tag{4.11}$$

Since all these values are semantic secure encryptions, an attacker cannot distinguish the encryptions of 0's from the encryptions of 1's.

The encrypted database $\boxed{\hat{D}}$ can be encoded as an $m \times N$-matrix of bits. Standard PIR, like the one presented in section 4.3, can be used to obliviously retrieve the i-th column, i.e. $\boxed{\hat{d}_i}$.

The bit map has the following advantages and disadvantages:

pro's

- Due to the PIR-method used, the server does not learn which value is being asked for.
- Because the values in $\boxed{\hat{D}}$ are encrypted semantically secure, the server does not learn the stored values either.

con's

- Compared to the size of the unencrypted database D, which is $n \log_2 N$ bits, the size of the encrypted database $\boxed{\hat{D}}$, which is Nm, is rather big.
- The communication costs are unacceptably high. A single query costs $2Nm$ bits. The answer, which is the vector a from section 4.3, contains m encryptions of size m bits each. The total costs are $2Nm + m^2$.

  The plaintext database or a database encrypted with a normal block cipher like DES or AES only takes $n \log_2 N$ bits. Therefore the communication costs exceed the storage size of such a database. Using range queries (section 4.4.2) or storing query vectors (section 4.4.3) or query templates (section 4.4.4) on the server reduce the size of the queries.

4.4.2 Range queries

An effective way to shorten the query q (see section 4.3) is to query only over a particular range of the database. If you want to know if an element e is in the database ($e \in D$ or in other words $\hat{d}_e = 1$), a range $\hat{R} = \{\hat{d}_a, \ldots, \hat{d}_b\}$ can be chosen such that $a \leq e \leq b$. Instead of querying over the complete database $\hat{D}$ we restrict ourselves to the much shorter range $\hat{R}$. The query

$$q = \begin{pmatrix} q_0 \\ \vdots \\ q_{N-1} \end{pmatrix}, \tag{4.12}$$

which was used for searching in the complete database, can be cut off at both sides to

$$q' = \begin{pmatrix} q_a \\ \vdots \\ q_b \end{pmatrix}. \tag{4.13}$$

Of course we also have to give the range variables a and b to the server, giving away a bit of information. It is up to the user how much privacy he is willing to sacrifice for better efficiency. This brings us to the following pro's and con's:

pro's

- The query length is linear in the size of the range. The query vector has $b - a + 1$ elements, each of which is a tuple of two encryptions. The query size is therefore $|q'| = 2m(b - a + 1)$ and is adjustable by choosing a and b. The answer takes m^2 bits, which leads to the total communications costs of $2m(b - a + 1) + m^2$ bits.

con's

- The server learns the interval in which the requested element e lies. Both the security and the efficiency are affected by the choice for a and b. A smaller range increases the efficiency but decreases security.

4.4.3 Stored query vectors

Another way to reduce the size of a query is to preload the server with all possible query vectors. The server stores, apart from the data itself, the following set of 'unit vectors'.

$$V = \{v_0, \ldots, v_{N-1}\}, \tag{4.14}$$

where

$$v_i = \begin{pmatrix} v_{i,0} \\ \cdots \\ v_{i,N-1} \end{pmatrix} \text{ and } v_{i,j} = \begin{cases} \langle \boxed{0}, \boxed{1} \rangle & \text{if } i = j \\ \langle \boxed{0}, \boxed{0} \rangle & \text{if } i \neq j \end{cases} \tag{4.15}$$

The set of vectors V is stored in a permuted order to hide as much information from the server as possible. The only thing the server knows about V is that it stores all the 'unit vectors', but it does not know which is which.

When a client wants to use one of the query vectors it does not have to transmit the whole vector. The client can merely give the (permuted) index to one of the vectors. The server then uses the associated vector in the same way as if the vector was transmitted (see section 4.4.1).

The costs have been shifted from communication to storage. More specifically, using stored query vectors has the following pro's and con's:

pro's

- The server stores N vectors. To point to one of them, an index of size $\log_2 N$ bits is needed. This is much less than the $2mN$ bits that is needed to transmit a whole vector and therefore much more efficient. The answer is still m^2 bits, which brings the total communication costs to $\log_2 N + m^2$.

con's

- An obvious drawback of storing the set of vectors V is that it takes valuable storage space. An extra $2mN^2$ (i.e. N 'unit vectors' of N tuples of 2 encryptions each) bits is needed. Using query templates (section 4.4.4) reduces the needed storage space.
- When the same element is queried twice, the same vector (and therefore the same index) is used for each query. The server learns that these queries are equal. In most situations this may not be a problem, but in some other situations the extra information the server learns may be unwanted. If, for instance, two different users ask for the same element, the server can link the two. If this linkability is unwanted, the set of query vectors should be refreshed from time to time in order to prevent double usage. Sections 4.4.6-4.4.7 give some suggestions how to refresh V.

4.4.4 Stored query templates

Instead of storing all the query vectors on the server, query templates can be stored. Let T be a set of l query templates. Each template is a vector of variable names.

$$T = \{t_i \,|\, 0 \leq i < l; t_i = (t_{i,0}, \ldots, t_{i,N-1}); t_{i,j} \in W\} \tag{4.16}$$

$W = \{w_0, \ldots, w_{c-1}\}$ is a set of variable names with $1 \leq c \leq N$. Each variable which will bind to a tuple of two encrypted values $\langle \boxed{0}, \boxed{x} \rangle$ for some concrete value $x \in \mathbb{Z}$ in an actual query.

T is stored at the server. The client can either store a copy of it or query the server when it needs some of the templates.

When a client wants to query for the occurrence of an element $e \in D$ (or equivalent $\hat{d}_e = 1$), it should somehow construct the eth 'unit vector'. It can do so by choosing an appropriate subset $T' = \{t'_0, \ldots, t'_{k-1}\} \subseteq T$ and a binding for all the free variables in T'. The query vector v_e is then the linear combination

$$v_e = \lambda_0 t'_0 + \cdots + \lambda_{k-1} t'_{k-1}, \tag{4.17}$$

where the λ's are calculated by the client.

Notation 4.4.1 *In the following example we will use the following notation:*

$$\overline{x} = \langle \boxed{0}, \boxed{x} \rangle. \tag{4.18}$$

For vectors of this kind of tuples, the following concatenation is used

$$\overline{x_1 \cdots x_n} = \begin{pmatrix} \overline{x_1} \\ \vdots \\ \overline{x_n} \end{pmatrix} \tag{4.19}$$

For an expression expr under a binding $W = \{w_0 \mapsto x_0, \ldots, w_{c-1} \mapsto x_{c-1}\}$ we will use the notation

$$expr[W \mapsto \overline{x_0 \cdots x_{c-1}}]. \tag{4.20}$$

Example 4.4.2 *Consider a server that stores the following query vector templates:*

$$T = \begin{pmatrix} t_0 \\ t_1 \\ t_2 \end{pmatrix} = \begin{pmatrix} w_0 & w_0 & w_1 & w_2 & w_1 \\ w_1 & w_2 & w_0 & w_1 & w_1 \\ w_2 & w_1 & w_0 & w_0 & w_0 \end{pmatrix} \tag{4.21}$$

It can be shown that all 'unit vectors' can be constructed as a binding for w_0, w_1, w_2 and a linear combination of a subset of queries from T. The list below is not complete. Only a small number of possible linear combinations are shown.

$$\begin{array}{l} v_0 = \overline{10000} = t_2[W \mapsto \overline{001}] \\ v_1 = \overline{01000} = t_2[W \mapsto \overline{010}] = t_1[W \mapsto \overline{001}] \\ v_2 = \overline{00100} = t_1[W \mapsto \overline{100}] = \\ \frac{1}{2}t_1 + \frac{1}{2}t_2[W \mapsto \overline{1(-1)1}] = \frac{2}{21}t_0 + \frac{3}{21}t_1 + \frac{2}{21}t_2[W \mapsto \overline{5(-2)(-2)}] \\ v_3 = \overline{00010} = t_0[W \mapsto \overline{001}] \\ v_4 = \overline{00001} = \frac{1}{3}t_0 + \frac{1}{3}t_1 + \frac{1}{3}t_2[W \mapsto \overline{(-1)2(-1)}] \end{array} \tag{4.22}$$

A user can tune the balance between communication and storage costs by choosing an appropriate set of templates. Storing more templates takes space but can make the linear combination with the corresponding variable binding simpler and thus smaller to transmit. The user has to ensure that with the chosen set of templates all the 'unit vectors' can be built. This brings us to the pro's and con's of using stored query templates:

pro's

- The server stores only the query templates. For l query templates and c different variable names, the storage costs are $l \log_2 c$ bits. Since typically $l \ll N$ and $c \ll N$ this is far better than the $2mN^2$ bits that are necessary to store all the 'unit vectors'.
- There is a trade-off between storage and transmission costs. If more templates are stored, the chances are high that there exists a subset of templates with only a few free variables. If, on the other hand, the stored templates form a minimal basis, then the chances are high that you need nearly all templates and need to bind almost all variables. It is up to the user to make the trade-off.
- There are multiple ways to construct a 'unit vector'. Asking the same element twice can be hidden by choosing two different linear combinations with two different bindings. This reduces the need to refresh the stored vectors considerably.

con's

- The transmission costs are higher than in the case where the 'unit vectors' are stored. Choosing an appropriate set of query templates, however, will ensure that the number of bindings will not be too big.

- Both the server and the client have some work to do. The client should choose a suitable linear combination. The number of free variables should be minimized, because the majority of the transmission costs are made up of the bindings. The server has to construct the 'unit vector' before it can be used to find the desired element.

4.4.5 Replacement

The problem of the storing preloaded queries on the server is that queries that are asked more than once can be linked to each other. This happens because the stored queries are reused over and over again. To prevent this reuse, the stored queries should be refreshed from time to time. A very easy way to do so is by replacing all the stored queries from time to time. This replacement is expensive in terms of bandwidth and should therefore not be used more than strictly necessary. The bandwidth can be spread over time by replacing only parts of the stored vectors.

pro's

- The server cannot link the query vectors before and after the replacement. Therefore queries that are being asked for after a replacement cannot be linked to earlier queries.

con's

- For the replacement of all N vectors (with N tuples of 2 encryptions), $2mN^2$ bits have to be transmitted over the network.

4.4.6 Shift

The major drawback of the previous method to refresh the stored query vectors, is the large network bandwidth that is needed. It can be reduced to 'only' $4mN$ bits by the following shift method.

The set of stored 'unit vectors' V can be written in matrix notation:

$$V = \begin{pmatrix} v_0 \\ \vdots \\ v_{N-1} \end{pmatrix} = \begin{pmatrix} v_{0,0} & \cdots & v_{0,N-1} \\ \vdots & & \vdots \\ v_{N-1,0} & \cdots & v_{N-1,N-1} \end{pmatrix}. \tag{4.23}$$

This matrix can be shifted one position to the left. The client knows how the 'unit vectors' are permuted and therefore knows which value of the left most column is the encryption of 1. The client encrypts a fresh column. The encryptions of the zeros and ones stay at the same place as the old column. Because a semantically secure encryption algorithm is used, the re-encrypted column has become different. This re-encrypted column is transmitted back to the server and will be the new right column. V has been transformed to

$$V' = \begin{pmatrix} v_{0,1} & \cdots & v_{0,N-1} & v'_{0,0} \\ \vdots & & \vdots & \vdots \\ v_{N-1,1} & \cdots & v_{N-1,N-1} & v'_{N-1,0} \end{pmatrix}. \tag{4.24}$$

The shift method is probably the best way to refresh the stored query vectors. However, besides the advantages it also have some slight disadvantages:

pro's

- The shift method is less expensive in terms of transmitted bits than a total replacement.

con's

- After N shifts the 'unit vectors' are in the same order again. If the same query is asked after exactly a multiple of N shifts the server can detect that they are the same. This is only problematic when the same query has been asked for more than N times (since the vectors can always be shifted one more time). If this is the case, the stored vectors should be refreshed in another way (for instance with addition (section 4.4.7) or substitution (section 4.4.5).
- The client should not only remember how the 'unit vectors' are permuted, but also how many times the server has shifted its vectors.

4.4.7 Addition

Another way to refresh the stored vectors is to add a newly transmitted vector to all the stored vectors. However, the stored vectors are no longer 'unit vectors' after an addition. It is not even guaranteed that the vectors form a basis any more.

pro's

- The number of bits to transmit is halved compared to the shift method.

con's

- The stored vectors are no longer 'unit vectors'. More than one vector is involved for each query. The client should come up with a suitable linear combination.
- The client should have a good bookkeeping to know which vectors can be used for a query and which query should be added to ensure that the stored vectors still form a basis. If the stored vectors does not form a basis any more than not all 'unit vectors' can be reconstructed.

4.4.8 Dual homomorphic encryption

In this section we propose a different solution than the bit map approach. This approach does not need to transform the set of integers to a bit map prior to the storage at the server. The database $D = \{d_0, \dots, d_{n-1}\}$ with $d_i \in \mathbb{Z}_N$ can be directly encrypted to $\boxed{D} = \{\boxed{d_0}, \dots, \boxed{d_{n-1}}\}$, where $\boxed{\cdot} : \mathbb{Z}_N \to \{0,1\}^m$ is a homomorphic encryption function which needs the property of equation (4.25).

$$\boxed{\prod_{\substack{1 \le i \le n \\ 1 \le j \le m}} (x_i + y_j)} = f(\boxed{x_1}, \dots, \boxed{x_n}, \boxed{y_1}, \dots, \boxed{y_m}) \tag{4.25}$$

This equation states the homomorphic property that (the encryption of the) the product of the sum of two elements can be calculated given only the encryptions of the elements. In other words, the homomorphic encryption function can multiply multiple times but can only add once in a sequence.

The encryption method of Domingo-Ferrer [18,19] allows us to calculate this product of sums, given only the encryptions of the components.

A query $d \overset{?}{\in} D$ is encrypted to $\boxed{-d}$ before it is transmitted to the server. Using the single additive homomorphic property, this value is added to each element in the encrypted database forming $\boxed{D'} = \{\boxed{d_0 - d}, \dots, \boxed{d_{n-1} - d}\}$. If d is in the database then one of these encryptions is $\boxed{0}$. Using the multiplicative homomorphic property all these values can be multiplied together. The product is either $\boxed{0}$ indicating that $d \in D$ or the encryption of a non-zero value indicating that $d \notin D$.

Both communication and storage costs are low. More specifically:

pro's

- Because only the elements that are in the database are stored, the storage costs are kept low. More precisely, n encryptions of m bits each have to be stored, which brings the total storage costs to nm bits.
- The query consists of a single encrypted value (of m bits). The answer consists of a single encryption too. Thus the transmission costs are kept low (i.e. $2m$ bits).

con's

- The security is based on the security of the underlying encryption method of Domingo-Ferrer. The encryption function has been broken for the known-plaintext scenario [12,49]. However, it is still supposed to be secure in the ciphertext-only scenario.

4.4.9 Polynomial extension

With a standard additive homomorphic encryption function like Paillier (section 4.2.4) it is possible to evaluate an encrypted polynomial in a given (plaintext) point. With an encrypted polynomial we mean a polynomial of which the coefficients are encrypted. For instance, a polynomial

$$f(x) = \alpha_0 + \alpha_1 x + \alpha_2 x^2 + \cdots + \alpha_{n-2} x^{n-2} + \alpha_{n-1} x^{n-1} \tag{4.26}$$

can be represented as a list of encrypted coefficients $\{\boxed{\alpha_0}, \ldots, \boxed{\alpha_{n-1}}\}$.

Paillier is used for the encryption. Therefore, $\boxed{x+y} = \boxed{x} \cdot \boxed{y}$ and for a constant plain text value c: $\boxed{cx} = \boxed{x}^c$. When the encrypted coefficients and a plaintext value v are given to the server , it can calculate the encryption of the evaluation of the polynomial f in point v, that is

$$\begin{aligned} \boxed{f(v)} &= \boxed{\alpha_0 + \alpha_1 v + \alpha_2 v^2 + \cdots + \alpha_{n-2} v^{n-2} + \alpha_{n-1} v^{n-1}} \\ &= \boxed{\alpha_0} \cdot \boxed{\alpha_1 v} \cdot \boxed{\alpha_2 v^2} \cdots \boxed{\alpha_{n-2} v^{n-2}} \cdot \boxed{\alpha_{n-1} v^{n-1}} \\ &= \boxed{\alpha_0} \cdot \boxed{\alpha_1}^{v} \cdot \boxed{\alpha_2}^{v^2} \cdots \boxed{\alpha_{n-2}}^{v^{n-2}} \cdot \boxed{\alpha_{n-1}}^{v^{n-1}} \end{aligned} \tag{4.27}$$

Using the Horner scheme [34] this polynomial can be calculated with only n additions and n multiplications. This homomorphic polynomial evaluation can be used in a PIR setting. Consider a database $D = \{d_0, \ldots, d_{n-1}\}$ with

$d_i \in \mathbb{Z}_N$. Instead of storing the values in plaintext like PIR, the values are used to form the polynomial

$$f(x) = \prod_{i=0}^{n-1} (x - d_i) = \sum_{i=0}^{n} \alpha_i x^i. \tag{4.28}$$

The encrypted coefficients $\{\boxed{\alpha_0}, \ldots, \boxed{\alpha_n}\}$ are given to the server. A query in the form of 'does a value v exists in the database' ($v \overset{?}{\in} D$) is translated to $f(v) \overset{?}{=} 0$. The server cannot evaluate f in v directly. However, it can calculate $\boxed{f(v)}$. This way, the server does not learn the answer to the query, but it still learns the query. The latter can easily be solved by not using the d_i's and v directly. Instead use the encryptions $E(d_i)$ and $E(v)$. Here the encryption function can be any deterministic encryption function. The most efficient however, is a traditional symmetric block cipher like AES. In order for it to work, the function $f(x)$ should be changed to

$$\hat{f}(x) = \prod_{i=0}^{n-1} (x - E(d_i)) = \sum_{i=0}^{n} \hat{\alpha}_i x^i. \tag{4.29}$$

The query $v \overset{?}{\in} D$ will now be translated to $\hat{f}(E(v)) \overset{?}{=} 0$. The server can calculate $\boxed{\hat{f}(E(v))}$ just like before.

This polynomial extension to PIR is very efficient for static databases. For dynamic databases it is less efficient, because for each update all the encrypted coefficients will change. Since the client has to calculate them, they have to be transmitted (twice) over the network. In summary, the pro's and con's are:

pro's

- The storage costs are low. Assuming that the homomorphic encryption is a function $\boxed{\cdot} : \mathbb{Z}_N \to \mathbb{Z}_M$, the server should store n coefficients of $\log_2 M$ bits each. The total storage therefore becomes $n \log_2 M$. In the case of Paillier $M = N^2$. Compared to the storage of the plaintext values $d_0, \ldots, d_{n-1}$ the storage costs are doubled.
- Communication cost of a query is low. The client transmits $E(v)$ which takes $\log_2 N$ bits. It receives an encryption which takes $\log_2 M$ bits. Total communication costs are therefore $\log_2 N + \log_2 M$ bits. If the server was storing the plain text values, the communication cost would have been $\log_2 N + 1$.

con's

- When a client asks the same query twice, the server will notice this.
- Updates to the database are expensive. Each time a value is changed, deleted or added, a new polynomial should be constructed. The server cannot do this, so all the coefficients should be transmitted to the client, which performs the update and sends the updated coefficients back. Although the client does not need the storage capacity for all the coefficients (the operation can be streamed), the communication costs are proportional to the size of the database, which makes this solution only useful for static databases.

4.5 Conclusion and future work

There are several methods to extend PIR with encryption of the stored data. However, none of the presented solutions is perfect. Each of them has one or more drawbacks. Some of them have high storage requirements while others have high communications costs. Table 4.1 gives the storage requirements and the communication costs of all the extensions based on the PIR method of section 4.3. Note that the PIR method used throughout this chapter, which has communication complexity that is in the order of the square root of the number of bits in the database, is not the most efficient one that is around. For instance, the PIR method used by Gentry and Ramzan [24] has a communication complexity of $O(k + d)$ where k is a security parameter that is larger than the logarithm of the number of bits in the database and d is the size of the bit blocks. Adapting our extensions to the PIR method of Gentry and Ramzan may decrease the communication costs considerably.

The solutions that use an encrypted bit map to represent the data, all need a large storage capacity. It depends on the context whether this is a real problem or just an inconvenience. The communication costs can be reduced by using range queries or stored query (template) vectors. Also a combination is possible. For instance, the combination of range queries and stored query vectors is better than each of the solutions separately. The query is shorter than in either solution. Also, the combined solution is much less sensitive to the detection of duplicate queries than the stored query vector approach alone. The same element can be queried with different 'unit vectors' if the range is shifted to the left or the right. Therefore less refreshments of the vectors are needed.

The stored query approach solves the problem of the large transmission costs

Table 4.1: Storage requirements and communication costs of all the presented extensions to PIR. The parameters used are n (number of stored values), N (values $d_i \in \mathbb{Z}_N$), m (size in bits of a single encryption), l (number of stored query templates), c (number of variable names used in the query templates). The communication costs of the stored query template approach depend on the stored query templates. Therefore, a lower and an upper bound is given.

	storage size	communication costs
plaintext	$n \log_2 N$	$\log_2 N + 1$
PIR	$n \log_2 N$	$2nm + m^2$
bitmap	Nm	$2Nm + m^2$
range query over $\{\hat{d}_a, \ldots, \hat{d}_b\}$	Nm	$2m(b - a + 1) + m^2$
stored query vectors	$Nm + 2mN^2$	$\log_2 N + m^2$
stored query templates	$Nm + Nl \log_2 c$	$[\log_2 l + m, lm + cm]$
dual homomorphism	nm	$2m$
polynomial extension	$(n + 1)m$	$2m$
replacement		$2mN^2$
shift		$2mN$
addition		$2mN$

but introduces the problem of duplicate query detection. The latter may or may not be a problem. It depends much on the usage of the system. If, for instance, the system is a single user database, then the detection of a duplicate query does not leak much information. In a multi user database however, the linkage between two persons asking the same query may be undesirable. Duplicate query detection can be avoided by refreshing the stored query vectors from time to time. It is best to use a combination of shifting and replacing. The shift should be used until the cycle is complete. After N shifts a total replacement is needed.

The dual homomorphic encryption approach seems ideal. It has low storage and transmission costs.

The approach using the encrypted polynomials does not have such a doubtful assumption. It has low storage and communication costs. For static databases this solution is very efficient. Updates, however, are much less efficient.

Concluding, we can say that using homomorphic encryption to encrypt the stored data of a PIR database is possible, but that further research is needed to increase the efficiency.

Chapter 5

A lucky dip as a secure data store

Most crypto systems rely on the computational complexity of breaking them. Historical evidence suggests that all such systems in use nowadays will be broken some day, it is just a matter of time. Even though this time may be long, it may very well be possible that data remains sensitive for this very long time. In this chapter we propose the principle of a lucky dip as a data store, that is secure even under the assumption that an attacker has unlimited computational power. Before a message is put into the lucky dip it is compressed and split into multiple shares. All these shares are mixed with shares of the other messages already in the lucky dip. Due to the large number of shares it is (1) infeasible to try all possible combinations (computational assumption) and (2) impossible, even with infinite computational power, to distinguish actual messages from recombined shares that look genuine but which have never been inserted as such (information theoretic assumption).

5.1 Introduction

The security of almost all crypto systems (except the one-time-pad) used today is based on the computational complexity of a brute force attack. These systems assume that the encryption function is computationally not invertible, whereas we know there exists at least one inverse: the decryption function. Every crypto algorithm that uses a key can be broken by trying all possible keys. It is just a matter of waiting long enough for the computation to finish or for the computers to become fast enough. According to Moore's Law [38] the processing power of computers is doubled every 18 months. Thus, what seems unbreakable now, will eventually be broken somewhere in the future.

Normally this causes no problem since most data gradually loses its value and secrecy when time elapses. Other data, however, stays sensitive indefinitely. A typical example is medical data, for instance DNA, which should never be revealed to the public. It can contain highly sensitive data, for many generations to come.

In this paper we introduce a secure storage system that differs from the standard encryption methods in the sense that we do not solely rely on the computational complexity of the underlying cryptographic principles. We even assume that adversaries have infinite computational power.

Informally, our secure storage system splits the data into multiple parts, mixes them with the parts of other data and puts all those parts into a large lucky dip. Of course, an attacker with infinite computer power can reconstruct the original data from all the parts. However, he can also 'reconstruct' messages that were never put in it. And since he cannot distinguish genuine from fake messages he has no way of knowing which of the reconstructed messages are genuine.

For example, if an attacker wants to find the account balance of Mr. Smith in the financial database of a bank, he will find several messages of the form 'The account balance of Mr. Smith is XXX' with many different values for XXX. Although the attacker learns some information about Mr. Smith's account balance, namely that it is one of the found possibilities, it is still pretty useless, because the attacker has no certainty which of the found possibilities is the correct one.

In section 5.2 we will explain precisely how the lucky dip works. Section 5.3 analyses its security aspects.

5.2 A lucky dip

The basic idea is to store several messages owned by different users into a single lucky dip. Each message is split into multiple parts, which are mixed with parts of other messages, obscuring which parts belong together. Without any additional information it is computationally hard to reconstruct the messages. The only way to reconstruct the messages is to do a brute force search, trying all possible subsets. The number of guesses grows exponentially in the number of shares.

Furthermore, the parts can be combined in so many ways that many of those recombinations look legitimate, although they have never been put into the lucky dip. An adversary cannot do better than guessing which one is genuine and which one is fake.

In order, for a legitimate user, to be able to retrieve the messages efficiently, the parts are annotated by labels. The labels are generated by the user and act as private keys. The labels, which typically take less space than the messages, are stored at the client site and will be used to retrieve the parts belonging to the same message. Typically, only a small fraction of all possible messages is actually stored in the database. Therefore, the storage requirements for the index containing the labels is considerably less than that of the messages themselves. To hide the relation between the labels from an eavesdropper, genuine labels can be mixed with bogus labels.

5.2.1 Data storage

We assume that each message is divided into blocks of numbers over a finite field $\mathbb{F}$. In a typical application this finite field will be the binary finite field, i.e. $\{0, 1\}$ with binary addition. Each such block is an element of $\mathbb{F}^n$ where n is the block length. For ease of discussion we will assume that all messages have a fixed length equal to the block length of the shares. That is, all messages m_i are taken from $M \subseteq \mathbb{F}^n$. Each m_i is split into $k \in \mathbb{N}$ parts: $m_i = m_i^{(1)} \oplus \cdots \oplus m_i^{(k)}$. The XOR notation ($\oplus$) is used but any secret sharing scheme will do. Since we use secret sharing to split the messages into parts we will use the term 'share' instead of 'part' from now on.

A share $m_i^{(j)}$ gets label $l_i^{(j)}$. The labels can be of any type, but it is most practical if the labels are elements of $L \subseteq \mathbb{F}^s$ where s is the size of the labels, which is typically much smaller than n. The server stores the lucky dip containing the tuples $(l_i^{(j)}, m_i^{(j)})$ and the client keeps track of which labels belong together: $(i, \{l_i^{(1)}, \ldots, l_i^{(k)}\})$.

5.2.2 Private information retrieval in our setting

When retrieving a message m_i from the database, the user first retrieves the corresponding labels $l_i^{(1)}, \ldots, l_i^{(k)}$ from its own data store. These legitimate labels are mixed with bogus labels before they are sent to the server. The bogus labels should be in use in the lucky dip. This way the fact that $l_i^{(1)}, \ldots, l_i^{(k)}$ belong together is hidden from the server. The server retrieves both the legitimate shares $m_i^{(j)}$ and the bogus shares. The latter ones can easily be filtered out by the client.

Let the total number of labels requested be ck ($c \in \mathbb{N}$) of which only k are legitimate. Then, an attacker has $\binom{ck}{k}$ possibilities of putting together shares (i.e. $\approx \mathcal{O}((ck)^k)$ choices).

It would be bad if the $(c-1)k$ labels which are sent along with the real labels to retrieve m_i would be different each time the same message m_i is retrieved, because if an attacker is aware of the fact that the user is retrieving the same message twice, then he will simply take the intersection of the labels sent the first time and the second time. To prevent this, when requesting the same message twice, one should make sure that the requested ck labels will always be the same for a specific message. There are various ways to accomplish this. For example, one can put each possible label in a pre-set group of c labels; when desiring one of the labels in this group, one asks for the data connected to each label in this group. For example, if requesting the data connected with label $l \in \{0,1\}^{50}$, then one always requests the data connected with all labels l' that have the first 40 bits in common.

5.2.3 Reusing shares

To further increase the chaos in the lucky dip, different messages, possibly owned by different users, can share each others shares. For instance let $m_1 = m_1^{(1)} \oplus m_1^{(2)} \oplus m_1^{(3)}$, then m_2 may be defined by $m_2 = m_1^{(1)} \oplus m_1^{(2)} \oplus m_2^{(1)}$, reusing $m_1^{(1)}$ and $m_1^{(2)}$. The purpose of reusing shares is twofold. On the one hand it reduces the size of the lucky dip, since fewer shares are stored. On the other hand security is increased.

To quantify the effect of reusing shares with respect to the security and the size of the lucky dip, we compare two lucky dips: one with and one without reuse. Assume that each message, except the first one, will be composed of

$k-1$ shares that are already in the lucky dip, plus a new share. In this case the lucky dip reusing shares stores $k+h-1$ shares (where h is the total number of messages) whereas the non-reusing lucky dip stores all hk shares. Thus reusing shares approximately costs a factor k less in size.

On the other hand: fewer shares reduce the security, since fewer k-tuples can be taken from the smaller lucky dip. However, it is not as bad as it looks like. In the non reusing case, the lucky dip randomly partitions its hk shares into h partitions, whereas in case of reuse an attacker does not have the advantage of a nice partitioning. In section 5.2.5 we exploit reuse for securing updates.

In the analysis below we assume that the attacker has retrieved all the data in the lucky dip and tries to find out which shares belong together in order to retrieve all messages back. In fact, he tries to 'decrypt' the entire database. But, since he has no additional information, there are quite a number of possible descriptions of which only one is correct.

Without reuse

An attacker does not know which particular partition is chosen, so he has to investigate all possible partitions. The number of possibilities is calculated as follows:

1^{st} k-tuple: $\binom{hk}{k}$
2^{nd} k-tuple: $\binom{hk-k}{k}$
...
i^{th} k-tuple: $\binom{hk-(i-1)k}{k}$
...
h^{th} k-tuple: 1

Which makes the total number of possible partitions $\prod_{i=0}^{h-1}\binom{hk-ik}{k} = \prod_{j=1}^{h}\binom{jk}{k}$.

With reuse

In case of reuse an attacker cannot rely on a nice partition. He has to take h different k-tuples out of a lucky dip of size $h+k-1$. Thus, the total number of possibilities is $\binom{h+k-1}{k}^h$. This number is less than the number of possibilities in case without reuse, but is still huge.

5.2.4 Threat model

In this paper we categorise attackers according to their capabilities:

type	see lucky dip at a certain moment	see lucky dip at every moment	access to communication
I	✓		
II	✓	✓	
III	✓	✓	✓

An attacker of type I (for instance an employee who steals a hard disk) cannot see any communication, while an attacker of type II (for instance a backup operator who can make frequent copies of the database) can see updates and one of type III (for instance a system operator with full control over the system) can see both updates and read operations. All attackers in our model are passive. We do not investigate active attackers who modify data in transit or data stored in the lucky dip. Further research is required to allow the presence of active attackers. Active attackers may try to corrupt the stored messages by modifying or deleting shares. Future research is needed to prevent them from doing so or by detecting such fraudulent actions.

5.2.5 Database operations

Standard database operations are:

- read
- add
- delete
- (modify)

where the last one can be modelled as a ⟨delete, add⟩ sequence and will thus not be dealt with here explicitly.

A database system based on the lucky dip principles should take care that the information leakage is kept low for all these operations. A trade-off should be decided on between security and efficiency. The lucky dip parameters allow this trade-off to be specified precisely.

All operations have their own security threats and consequences. Each of them is summarised below:

read When only attackers of types I and II (see section 5.2.4) are to be taken care of, no special precautions are needed. However, if there are type III attackers around, just asking for the k shares leaks the whole message. To hide the fact that the k shares belong together, noise can be introduced by adding b bogus labels to the query. This way, the information leakage is restricted to the fact that within the $k+b$ shares a message (split over k shares) is hidden. However, the total number of possible messages is $\binom{k+b}{k}$ and can be very large for a sufficiently large b, which acts as the trade-off parameter between security and efficiency. When a message is being retrieved multiple times, it is advisable to use the same set of $k+b$ shares each time. Not doing so, an attacker may intersect two sets of shares belonging to two messages guessed to be the same. If the messages are indeed the same the intersection will almost certainly reveal the k shares.

add A type I attacker is unable to see any updates. Therefore, no precautions are needed against him.

A type II attacker is best misled by allowing reuse of shares. When $k-s$ shares are taken from the ones already in the lucky dip, only s (for example $s=1$) shares have to be added. A type II attacker has no clue which other shares they belong to. This is not true for a type III attacker, since he can see the retrieval of the $k-s$ shares preceding the update.

To mislead a type III attacker, it is preferable to add many messages at once. Mixing t messages will result in tk shares. The total number of recombinations is $\prod_{j=1}^{t}\binom{jk}{k}$ which may be enough when t is sufficiently large. When the number of messages to be added is small, then mixing the real messages with bogus shares will increase the security. To prevent that the bogus shares allocate valuable storage space, the bogus shares may be chosen from the ones already in the lucky dip. When the lucky dip allows reuse of shares, an attacker cannot distinguish a bogus share and a reused share.

delete

Although the messages to be deleted are old or incorrect (otherwise: why bother to delete them?), it is still not a good idea to reveal them.

If the number of messages to be deleted (t) is sufficiently large, the messages are mixed well enough to prevent repartitioning the tk shares into the t original messages.

When reuse of shares is allowed, deletion of a single share may cause many messages to get corrupted. Since there is no single entity knowing which share belongs to whom, it is impossible to safely delete a share without taking extra measures. One such measure is adding a reference counter to each share. Each time a share is used as part of a newly added message, the counter is increased and every time a corresponding message is deleted it is decreased. To avoid that a type II or III attacker can figure out which shares belong together by looking at the increase and decrease operations, these operations should be spread over time. For instance, a client may reserve a bunch of shares early in time by asking the server to increase their reference counters. Each time he wants to add a message he can use some of these reserved shares while not telling the server so. When deleting, he can mix the real shares with enough reserved (but not used) shares, to provide enough security. The lucky dip cannot distinguish a reserved share and a share in use. It will only actually delete the share when the reference counter reaches zero. A time-out mechanism is another technique to store only shares that are actually in use.

A time-out mechanism can be used to delete shares which have been expired. To ensure that his messages are not deleted, a user has to refresh his shares from time to time. If the user refreshes all its shares at once, the server cannot link the shares to the individual messages.

5.3 Security aspects

In this section we will use the following notation:

- D is the set of size h of (unshared) messages that are to be stored in the database.
- S is the set of shares. That is $S = \{s_1, \ldots, s_k \mid d \in D, \{s_1, \ldots, s_k\} = \text{share}(d)\}$, where share$(d)$ is a secret sharing algorithm, splitting a message d into shares $s_1, \ldots, s_k$ such that $d = s_1 \oplus \cdots \oplus s_k$. The size of S is hk in case without reuse of shares and $k + (k - \tilde{k})(h-1)$ in case $\tilde{k}$ shares are reused for each message. We define $s = |S|$ as the size of S.
- R is the set of messages than can be reconstructed from the shares in S, that is $R = \{m \mid s_1, \ldots, s_k \in S; m = s_1 \oplus \cdots \oplus s_k\}$.
- M is the set of possible messages (for instance, all correct English texts).
- $T = R \cap M$ is the part of R that makes sense.
- $U = \mathbb{F}_2^n$ is the universe containing all strings of n bits.

Further, we assume that messages are represented as fixed sized bit strings, thus $D \subseteq T \subseteq M \subseteq U$, $T \subseteq R$ and $S \subseteq U$.

Stochastic variables are notated using calligraphic letters. Thus $\mathcal{D}$, $\mathcal{S}$, $\mathcal{R}$, $\mathcal{M}$, $\mathcal{T}$ and $\mathcal{U}$ are used to denote the stochastic variables belonging to the sets D, S, R, M, T and U respectively.

5.3.1 Entropy

Definition 5.3.1 (Shannon entropy) *The Shannon entropy [45] of a variable $\mathcal{X}$ over a set X with probability function Pr is defined as:*

$$H(\mathcal{X}) = -\sum_{x \in X} Pr(\mathcal{X} = x) \log_2 Pr(\mathcal{X} = x). \tag{5.1}$$

If a set X of size $|X|$ is uniformly distributed, the entropy is just

$$H(\mathcal{X}) = \log_2 |X|. \tag{5.2}$$

Assuming that D, S, R, M, T and U are uniformly distributed we have the following entropies:

$$\begin{array}{l} H(\mathcal{D}) = \log_2 h \leq \\ H(\mathcal{T}) = \alpha \log_2 \binom{s}{k} \leq \\ H(\mathcal{M}) = \alpha n \leq \\ H(\mathcal{U}) = n \\ H(\mathcal{R}) = \log_2 \binom{s}{k} \leq n \end{array} \tag{5.3}$$

where $0 < \alpha \leq 1$ is a compression factor. An English text has an information value of around 1.3 bits per character. This means that if a perfect compression algorithm would exist, it will use 1.3 bits to store a character. Thus, $\alpha = 1.3/8$ if the plaintext uses an 8-bit ASCII encoding. α reaches 1 if all elements of U are correct values.

5.3.2 Difficulty of finding a message by an attacker

Section 5.2.3 dealt with the difficulty of illegally 'decrypting' the entire database. In this section a more plausible, but less sophisticated, attack is considered: finding only a single message. We use the same premise as before; the attacker has retrieved all the data but has not wiretapped any conversation (attack type I).

We assume that the attacker has an oracle O which states whether a recombination $m = s_1 \oplus \cdots \oplus s_k$ adds up to a legitimate message or not. The oracle is defined as:

$$O(m) = \begin{cases} 1 & \text{if } m \in M \\ 0 & \text{otherwise.} \end{cases} \tag{5.4}$$

Furthermore, we assume that the attacker has access to a computer with unlimited processing power and memory. Given the set of shares S, the attacker can compute all the recombinations $R = \{m \mid s_1, \ldots, s_k \in S \wedge m = s_1 \oplus \cdots \oplus s_k\}$. Using the oracle he can even compute the set of possible messages $T = \{r \mid r \in R \wedge O(r) = 1\}$. However, he cannot tell which elements of T are stored intentionally. In other words, the probability $Pr(t \in D \mid t \in T)$ is rather small:

$$Pr(t \in D \mid t \in T) = \frac{h}{2^{H(\mathcal{T})}} = \frac{h}{2^{\alpha \log_2 \binom{s}{k}}} = \frac{h}{\binom{s}{k}^{\alpha}} \tag{5.5}$$

Example 5.3.2 *To get a feeling for this probability, let's give a concrete example. Suppose that the lucky dip contains $h = 2^{20} \approx 1$ million different messages. Each message, of size $n = 2^{10} = 1$ kb is split into $k = 16$ shares. When reusing $k-1$ shares for each message, the lucky dip S contains $h+k-1 = 2^{20}+2^{6}-1 \approx 2^{20}$ shares, thus $s \approx 2^{20}$. Then, the probability of guessing correctly whether a random recombination is an intentionally stored message is*

$$Pr(t \in D \mid t \in T) = \frac{2^{20}}{\binom{2^{20}}{16}^{\frac{1.3}{8}}} \approx 2^{-25} \approx 10^{-8} \tag{5.6}$$

Without reusing shares the lucky dip contains $s = hk = 2^{24} \approx 16$ million shares. In that case the probability is much smaller:

$$Pr(t \in D \mid t \in T) = \frac{2^{20}}{\binom{2^{24}}{16}^{\frac{1.3}{8}}} \approx 2^{-35} \approx 10^{-11} \tag{5.7}$$

5.3.3 Using compression

The existence of the oracle O gives an attacker a great advantage. Many recombinations in R are not in M, i.e. are not correct English texts. In order to reduce this advantage, compression can be used prior to the sharing phase. A good compression algorithm removes all the redundancy from the input data. In case of a perfect compression algorithm, our factor α reaches 1. In fact, there is no difference any more between R and T; the advantage of using the oracle has gone. The entropy of $\mathcal{T}$ is now increased to

$$H(\mathcal{T}) = \log_2 \binom{s}{k} = H(\mathcal{R}). \tag{5.8}$$

As a consequence, the probability of guessing correctly whether a recombination is in the data set D is reduced to

$$Pr(t \in D \mid t \in T) = \frac{h}{2^{H(\mathcal{T})}} = \frac{h}{\binom{s}{k}} \tag{5.9}$$

Example 5.3.3 *Using the same values for h, k and s as in example 5.3.2, the probabilities are*

$$Pr(t \in D \mid t \in T) = \frac{2^{20}}{\binom{2^{20}}{16}} \approx 2^{-256} \approx 10^{-77} \tag{5.10}$$

with reusing shares and

$$Pr(t \in D \mid t \in T) = \frac{2^{20}}{\binom{2^{24}}{16}} \approx 2^{-320} \approx 10^{-96} \tag{5.11}$$

without reusing shares.

5.3.4 Trade-off between security and efficiency

In the previous section we saw that the probability of finding a message put into the lucky dip can be made as small as you want by choosing a high number of shares (k) for each message. However, choosing a value for the security parameter k has great impact on the efficiency of computation, bandwidth and storage space. With many shares per message a client has to perform more work to recombine the shares, it takes more time for all the shares to travel over the network and, probably most important, the client should store more information. A client has to remember all the labels of the shares of a message. If there are more shares, also more labels should be stored at the client site. (The storage space on the server is not influenced by the security parameter k when shares are being reused.)

For practical purposes, all labels in use should be unique. This causes the size of a label to be $l \geq \log_2 s$ bits, where s is the total number of shares in the lucky dip. For each message, the client stores kl bits in its label database. Thus,

k should be as small as possible when optimising for efficiency, but maximized (but not greater than $\frac{1}{2}s$) when optimising for security. It is up to the user what is most important: efficiency or security.

Obviously, we do not want to store more bits for the labels than the length of the message itself. The upper bound for k is given by the equation: $kl \leq n$, where n is the size (in bits) of a single message. This translates to

$$1 < k < \frac{n}{l} \leq \frac{n}{\log_2 s} = \begin{cases} \frac{n}{\log_2 h} & \text{, with reusing shares} \\ \frac{n}{\log_2 hk} = \frac{n \ln 2}{W(hn \ln 2)}, & \text{without reusing shares} \end{cases} \tag{5.12}$$

where W is Lambert's W function. A function $W(x)$ is called a Lambert's W function iff $W(x)$ is the inverse of $f(x) = xe^x$.

Example 5.3.4 *Using the same $h = 2^{20}$ and $n = 2^{10}$ as in our running example k is bounded by*

$$1 < k \leq \frac{n}{l} \leq \frac{2^{10}}{\log_2 s} \approx \begin{cases} 51, & \textit{with reusing shares} \\ 40, & \textit{without reusing shares} \end{cases} \tag{5.13}$$

5.4 Conclusion and future work

Without relying on the assumption that an adversary's processing power is bounded, the concept of the lucky dip can be used to store data securely for an indefinite period of time. The concept consists of three phases: compression, secret sharing and mixing with other shares.

There is a balance between efficiency and security, which can be tuned by carefully choosing the security parameter k (i.e. the number of shares per message).

Reusing shares helps to keep the size of the lucky dip small, i.e. not substantially larger than the plaintext. Furthermore, update operations are better protected against an eavesdropper when reusing shares, because the reused shares do not have to travel over the network. The counter side of having fewer shares in the lucky dip, is that there are fewer recombinations possible. However, the number of recombinations is still large enough to safeguard security. Only in a situation where no attackers listening to the communication are to be expected and where wasting storage space is not a problem, a non-reusing lucky dip is favourable.

In this chapter we only considered passive attackers which do not alter messages in transit and do not alter the data in the lucky dip. Other safeguards are needed in order to protect the security against active attackers, especially when shares are being reused, since, in that case, changing one single share may corrupt several messages at once. This is still being investigated as ongoing research.

Chapter 6

Conclusions and future work

Having seen different ways to query encrypted data, one may ask which one is the best. This is not easy to answer, since each method has its own advantages and disadvantages. It depends on the requirements which one is the most appropriate. In this last concluding chapter we will sum up the strong points as well as the weaknesses of all the solutions.

6.1 Introduction

In the previous chapters we described a number of search techniques over encrypted data and a method of storing data securely for a longer period of time. In this concluding chapter we compare the search strategies with each other. In the next section we give all the advantages and disadvantages of the different search strategies and give guidelines when to use which strategy. Section 6.3 concludes our findings about the secure long term storage.

6.2 Search techniques

Both the solutions that exist in the literature and our new solutions to the first research question are compared in this section. All solutions have their own advantages and disadvantages. The solutions that are being compared are:

- The indexing technique of Hacıgümüş et al. [30–33].
- The trapdoor technique of Song, Wagner and Perrig (SWP) [46].
- Our own tree based extension of SWP (chapter 2).
- Our own solution using secret sharing (chapter 3).
- Our own solutions based on homomorphic encryption (chapter 4).

6.2.1 Hacıgümüş et al.

Hacıgümüş et al. encrypt the records of a relational database. Instead of searching in those encrypted records, some meta-data is added. This meta-data consists of the hashes of the plaintext values. The search takes place within this meta-data. To allow operators like ‘less than’ and ‘greater than’, a user-made hash function is used instead of a standard cryptographic hash function. The range of the input data is partitioned into intervals. Each interval is mapped to a unique value. This unique value acts as the hash. See section 1.2.1 for a more detailed discussion of Hacıgümüş et al.

Advantages

The index based solution uses a relational database as back-end. Since relational databases have been around for quite some time, there exist a huge theoretical background and all kinds of efficient indexing mechanisms. Hacıgümüş takes advantage of this to create an efficient solution, pushing as much of the workload to the server.

Disadvantages

The efficiency comes at a price, though. The storage cost doubles compared to the plaintext case. Apart from the encrypted data also the hash values for each searchable field need to be stored. These hashes are almost as big as the original values.

Another disadvantage is the fact that the server can link records together without the cooperation of the client. Values that are equal in the plaintext domain are also equal in the encrypted domain. Although the opposite does not hold, the server still learns which records are not the same. Therefore, it can estimate the number of different values and it can join tables fairly accurately.

A more practical disadvantage is that the user should choose the hash map in such a way that the intervals are not getting too big or too small. The hash map strongly depends on the distribution of the plain text values. When the distribution changes drastically, also the hash map should be redesigned.

6.2.2 SWP

SWP encrypt a text in such a way that it is possible to search for a particular keyword. The encryption of the keyword is accompanied with a cryptographic key that depends on the keyword. The key acts as a trapdoor with which the server can scan through the encrypted text to find the keyword. Since both the keyword and the stored text are encrypted, the server does not learn which word was search for. It only learns the locations where the word is found, if it is found at all.

Advantages

The encryption method of SWP does not need a larger storage space than in the plaintext case.

When a word occurs multiple times, the encryptions are different, which makes frequency analysis hard.

Almost the whole workload is done at the server site. Only the encryption of the keyword and a single hash operation are performed at the client site. This fact makes this strategy especially useful for lightweight devices like mobile phones.

Disadvantages

Song's strategy may be efficient when you only look at storage space, it is not when looking at computation time. For each query the whole data is being searched linearly. Thus this strategy does not scale well.

Another disadvantage is that all the words should have the same length. Padding is used to create equally sized words. However, padding increases the storage size.

6.2.3 Tree based extension of SWP

In chapter 2 an improvement to the SWP scheme is presented which reduces the computation time from linear to logarithmic by using more structured data as input. Instead of unstructured text, XML documents are used. A query engine supporting the full core XPath has been implemented. It shows that the search time is small enough for practical use, even for large databases. The query engine is still in the phase of a prototype. It has been built as a proof of concept and a way to test the efficiency not to be a commercial product. It can be extended from core XPath to the full XPath. It is also a good idea to mix our tree based extension with the original scheme. The tags and attributes can use our tree based extension, whereas the unstructured text that resides between the open and close tags can be search by the original scheme.

Advantages

The tree structure of the stored data makes it possible to search in logarithmic time instead of the linear search time of the original SWP technique. It is not longer necessary to search through all the text but only the nodes (and their siblings) that lead from the root node to the answer.

In our solution we also have dropped the requirement of fixed sized keywords, which is another disadvantage of the original scheme.

Disadvantages

Unfortunately, the reduction of the computation time also causes a slight increase in the communication costs. An XPath query is somewhat longer than just a single word. Thus instead of transmitting a single (encrypted) keyword with the corresponding trapdoor, an encrypted XPath query has to be transmitted. Depending on the complexity of the XPath expression this is a constant factor larger, but is still very small.

6.2.4 Secret sharing technique

In chapter 3 we presented a way to represent an XML tree as a tree of polynomials. This tree is split into a client tree and a server tree. Because the client tree is generated by a pseudo random generator it can be discarded, provided that the seed is remembered. The search algorithm consists of a secure multi-party protocol. The polynomials are constructed in such a way that not only information of the node itself is used, but also information of all the node's children. This makes it possible for the search algorithm to skip entire parts of the tree, making it quite efficient.

Advantages

The main advantage of the secret sharing strategy is its security. Since all the data stored on the server is randomly generated, it is just garbage for an attacker. Even two identical nodes are encrypted differently.

Another advantage is the efficient storage. Although knowledge about the whole subtree is stored at each node, the storage remains similar in size to the plaintext.

Disadvantages

A disadvantage, though, are the communication costs. Each node that is being traversed costs a round trip communication (with very little data) between the client and the server. Also the workload on the client is similar to the workload at the server.

6.2.5 Homomorphic encryption techniques

Homomorphic encryption makes it possible to calculate within the encrypted domain. It therefore makes sense to assume it is suitable to search in encrypted data. However, our research did not result in one search technique with only advantages. Instead, we presented several techniques; all with their own advantages and disadvantages.

PIR has been used as a starting point. PIR hides the query and the answer to the database system. PIR already has two of the three ingredients for a fully privacy aware database. Only the stored data is in the clear. In chapter 4 the stored data is encrypted too. Standard PIR does not work with encrypted data. That is to say, it can retrieve encrypted values if you know where they are stored. It is not possible, however, to search for a value if the location is not known. Therefore some extensions to PIR have been proposed. As said, no extension is perfect. But some of them are useful in some situations.

One class of extensions uses a bit map, which is a list of zeros and ones, where the ones represent the values that are in the database and the zeros that are not. For sparse databases (i.e. only a small number of all the possible values is stored) this is somewhat inefficient, but for dense databases (i.e. almost every possible value is stored) it is more efficient. The bit map is encrypted with a semantic secure encryption algorithm. A PIR method has been described to query this encrypted bit map. Transmitting a query is expensive. Several techniques can reduce the transmission costs. Range queries shorten the transmitted vectors but leak some information about the data. Preloading the server with query vectors is efficient in terms of needed bandwidth, but enables an attacker to discover duplicate queries. Detection of duplicate queries can be made impossible at the cost of more transmission. To reduce storage costs, query templates can be stored instead of the query vectors itself. The more templates are stored the shorter the queries can be. In summary we can say that the user has some means of tuning the system. He has to find the balance between security, storage space and transmission costs.

Another class of extensions stores the values as separate entities. In contrast with the bit maps, more data values means more storage space, which is a more natural behaviour for a database. The most ideal solution assumes a homomorphic encryption algorithm that can do multiple multiplication and a single addition within the encrypted domain. It is still an open question whether such an algorithm really exists. Therefore, this extension has no practical relevance yet. Another extension in this class represents the values in one large polynomial of which the coefficients are encrypted. Querying this polynomial is efficient. Updates, however, are much less efficient. The whole database should be re-encrypted for every modification.

In summary: using homomorphic encryption to extend PIR schemes is possible in theory, but further research is needed to make it usable in practice.

6.2.6 Search solutions compared

We have seen several strategies to search in encrypted data. It depends on the context which one is the best. The context consists of the architecture, the structure of the data, the complexity of the queries and the preferred balance between security and efficiency.

Architecture The kind of devices and the way they are connected to each other influences the choice for a particular search technique. If both client and server are fast devices and they are connected by a fast network, all the techniques described in this thesis can be used.

SWP (with or without our tree extension) is the best solution when the network bandwidth is low. The query is a single search word with a trapdoor and the answer is a list of locations. Both the technique of Hacıgümüş et al. and our secret sharing scheme use more bandwidth because data is transmitted for some nodes that are not in the result set.

For lightweight clients SWP (with or without tree extension) is best, because the workload is almost entirely on the server. Hacıgümüş et al. is also good because the workload can be shifted to either the client or the server.

Data structure Data can be structured in several ways. We identify the following data types, ordered by the degree of structure:

- set of objects/words/integers
- text of which the order of the words matter
- relational data
- tree data

If the data is organised as a set of unordered objects or an unstructured text, it should be searched in its entirety. Due to a lack of structure, it is not possible to skip parts of the data. Both the original SWP scheme and the various homomorphic solutions search the entire database anyhow. Thus, for this kind of data, both schemes are efficient enough. For more structured data, however, searching through the entire database is much less efficient. It is most natural to use Hacıgümüş et al. for data that is stored in a relational database and either the tree extension of SWP or the secret sharing scheme for tree structured data.

Query complexity Closely related to the structure of the data is the query complexity. The more structure the database has, the more complex the queries can be. The queries in the homomorphic solutions and the original SWP scheme are simple element lookups: check whether and where an element (a word or an

Table 6.1: Comparison of the different search strategies in terms of security/linkability, storage/conmmunication costs and the workload on the server and the client.

	linkability			costs		workload	
search method	data	query	answer	storage	com.	server	client
Hacıgümüş	-	-	-	+/-	+/-	+	+
SWP	+	-	-	+	+	-	+
tree ext. SWP	+	-	-	+	+	+	+
secr. sharing	+	-	+/-	+	+/-	+	+/-
homom. enc.	+	+	+	-	-	-	+/-

integer) occurs. The other 3 solutions (tree extension of SWP, Hacıgümüş et al. and the secret sharing scheme) are based on more complex query languages like SQL and XPath.

Balance between security and efficiency The presented solutions are not equally secure. Most secure are the homomorphic encryption solutions. But, unfortunately, they are also the least efficient. An attacker does not learn the stored data (because the data is encrypted) nor the query and the answer (due to the PIR method).

The most efficient solution is the index based solution of Hacıgümüş et al. Unfortunately, this is also the least secure solution, since it suffers from linkability. Records that are the same have equal hashes and therefore an attacker learns with a certain probability which records are equal.

SWP suffer from linkability too, although in a lesser extent. With only the stored data, an attacker is not able to find equal words, but if he sees an answer to a query, he knows that the retrieved locations contain the same word.

With our secret sharing scheme the user can balance the security (i.e. the linkability) and efficiency. When the client stops evaluating polynomials in a certain branch in the XML tree, the server learns that the answer is not in the skipped part of the tree. To improve security the client can therefore go on evaluating polynomials in a branch the client already knows does not contain the answer, just to mislead the server.

Table 6.1 summarises the comparison between the different search strategies. A plus indicates a strength and a minus a weakness. With linkability we mean the ability to relate one data element, query or answer to another. As we can see, all strategies have weaknesses. Therefore, we cannot recommend a single technique. The user should choose the technique that suits him most. His choice depends on the given architecture, the complexity of the data and the queries and his own judgement with regard to the balance between security and efficiency.

6.3 Long term storage

Having a nice searchable encrypted database is one thing. Keeping it secure over a longer period of time is another. Most encryption algorithms can be broken given enough equipment and/or enough time. Simply trying all possible keys will break the system sooner or (probably) later. In chapter 5 we use a lucky dip as a secure data store. We use the inherent chaos in our favour. We split messages into shares, throw them in the lucky dip and mix them with the shares of other messages. The greater the chaos, the better the security. With millions of shares we have a huge number of possible ways to recombine the shares to messages. In fact there are so many ways that we will find messages we can perfectly read but which have never been stored. It might even be possible to find a piece of Shakespeare's Hamlet in the financial administration of a company.

An attacker with an unlimited supply of computers and no time limit, can try all the possible recombinations. In the end he finds all the stored messages. However, those genuine messages are hidden between a huge number of other messages. An attacker has no means to distinguish genuine messages and messages that are found 'by accident'.

6.4 Conclusion and future work

In the introductory chapter two research questions are formulated:

1. "Can we store private data securely on a database server, of which we cannot rely on its access control mechanism, in such a way that it is possible to search the data efficiently?"

2. "Can data be stored in such a way that it stays secure forever without relying on computational assumptions?"

Both questions can be answered with a simple “yes”, although not all the presented solutions are equally efficient. In a follow-up project we would like to come up with more efficient search techniques without sacrificing the security.

In this thesis the focus is on searching in encrypted data, not querying over encrypted data. In the same follow-up project we would like to extend the search techniques to full query engines. Our current tools for searching in encrypted XML documents, for instance, use XPath to find the desired elements. XQuery goes one step further than XPath by generating a new document using the found elements. Making this generated document secure and searchable as well, is another research challenge. A full query engine should also support operators like ‘greater than’ and ‘less than’ or fuzzy ones such as ‘like’ or ‘similar to’. Ideas from the field of private fuzzy matching [23] may be used within the secure database world as well.

Another interesting idea is to combine our long term storage with one of the search techniques. Both our shared polynomial tree and our lucky dip use secret sharing. Combining the two approaches may lead to a database system that is searchable and secure for a longer period of time.

Ending this thesis does not mean that the research in the direction of searching in encrypted data stops. In the contrary, this thesis has proven that this research area is very interesting. We have proven that searching in encrypted data is possible. The next step is to do it more efficient and more secure using a more expressive query language.

Bibliography

Publications by the author

[1] R. Brinkman. *Security, Privacy, and Trust in modern data management*, chapter Searching in encrypted data, pages 183–196. Data Centric Systems and Applications. Springer Verlag, Berlin, May 2007.

[2] R. Brinkman, J. M. Doumen, P. H. Hartel, and W. Jonker. Using secret sharing for searching in encrypted data. In W. Jonker and M. Petković, editors, *Secure Data Management VLDB 2004 workshop*, volume LNCS 3178, pages 18–27, Toronto, Canada, August 2004. Springer-Verlag, Berlin.

[3] R. Brinkman, J. M. Doumen, W. Jonker, and B. Schoenmakers. Method of and device for querying of protected structured data. US patent application 04102375.5, May 2004.

[4] R. Brinkman, L. Feng, J. M. Doumen, P. H. Hartel, and W. Jonker. Efficient tree search in encrypted data. *Information Systems Security Journal*, 13(3):14–21, July 2004.

[5] R. Brinkman and J.H. Hoepman. Secure method invocation in Jason. In *USENIX Smart Card Research and Advanced Application Conference (CARDIS)*, pages 29–40, San Jose, CA, USA, November 2002.

[6] R. Brinkman, S. Maubach, and W. Jonker. Secure storage system and method for secure storing. European patent application EP06113192.6, April 2006.

[7] R. Brinkman, B. Schoenmakers, J. M. Doumen, and W. Jonker. Experiments with queries over encrypted data using secret sharing. In W. Jonker

and M. Petković, editors, *Secure Data Management VLDB 2005 workshop*, volume LNCS 3674, pages 33–46, Trondheim, Norway, Sep 2005. Springer-Verlag, Berlin.

[8] R. van Rein and R. Brinkman. Home-grown case tools with XML and XSLT. In *Int. Workshop on Model Engineering (IWME)*, pages 105–112, Sophia Antipolis, France, June 2000.

Other references

[9] R. Agrawal, J. Kieman, R. Srikant, and Y. Xu. Order-preserving encryption for numeric data. In *Proc. of the ACM SIGMOD 2004 Conference*, Paris, France, June 2004.

[10] D. Boneh, G. Di Crescenzo, R. Ostrovsky, and G. Persiano. Public key encryption with keyword search. In *Advances in Cryptology, Eurocrypt*, pages 506–522. Springer Verlag, Berlin, 2004.

[11] Dan Boneh, Eu-Jin Goh, and Kobbi Nissim. Evaluating 2-dnf formulas on ciphertexts. In *Theory of Cryptography: Second Theory of Cryptography Conference, TCC 2005*, Cambridge, MA, USA, Feb 2005. Springer Verlag.

[12] Jung Hee Cheon and Hyun Soo Nam. Known-plaintext cryptanalysis of the Domingo-Ferrer algebraic privacy homomorphism scheme. *Information Processing Letters*, 97(3):118–123, Feb. 2006.

[13] B. Chor and N. Gilboa. Computationally private information retrieval (extended abstract). In *Proceedings of the twenty-ninth ACM Symposium on the Theory of Computing*, pages 304–313, El Paso, Texas, United States, 1997.

[14] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval. *Journal of the ACM*, 45(6):965–981, November 1998.

[15] E. Damiani, S. de Capitani di Vimercati, S. Paraboschi, and P. Samarati. Computing range queries on obfuscated data. In *Proc. of IPMU 2004*, Perugia, Italy, 2004.

[16] E. Damiani, S. De Capitani di Vimercati, S. Jajodia, S. Paraboschi, and P. Samarati. Balancing confidentiality and efficiency in untrusted relational

DBMSs. In *Proc. of the 10th ACM Conference on Computer and Communications Security*, pages 93–102, Washington, DC, USA, October 2003. ACM Press New York, NY, USA.

[17] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, Nov 1976.

[18] Josep Domingo-Ferrer. A new privacy homomorphism and applications. *Information Processing Letters*, 60(5):277–282, Dec 1996.

[19] Josep Domingo-Ferrer. A provably secure additive and multiplicative privacy homomorphism. In *ISC'2002*, volume LNCS 2433, pages 471–483. Springer-Verlag, Berlin, Sep. 2002.

[20] Taher ElGamal. A public-key cryptosystem and a signature scheme based on discrete logarithms. *IEEE Transactions on Information Theory*, IT-31(4):469–472, 1985.

[21] Ling Feng and Willem Jonker. Efficient processing of secured XML metadata. In *Proceedings of Intl. Workshop on Security for Metadata*, Catania, Italy, Nov 2003.

[22] Edward Fredkin, Bolt Beranek, and Newman. Trie memory. *Communications of the ACM*, 3(9):490–499, September 1960.

[23] Michael J. Freeman, Kobbi Nissim, and Benny Pinkas. Efficient private matching and set intersection. In *Advances in Cryptology - EUROCRYPT*, volume LNCS 3027, pages 1–19, April 2004.

[24] Craig Gentry and Zulfikar Ramzan. *Automata, Languages and Programming*, volume 3580 of *Lecture Notes in Computer Science*, chapter Single-Database Private Information Retrieval with Constant Communication Rate, pages 803–815. Springer Verlag, 2005.

[25] Eu-Jin Goh. Secure indexes. Cryptology ePrint Archive, Report 2003/216, 2003.

[26] O. Goldreich. *Foundations of Cryptography*, volume 2. Cambridge University Press, May 2004. ISBN 0-521-83084-2.

[27] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, 1984.

[28] Torsten Grust. Accelerating XPath location steps. In *Proceedings of the 21st ACM International Conference on Management of Data (SIGMOD 2002)*, pages 109–120. ACM Press, Madison, Wisconsin, USA, June 2002.

[29] Torsten Grust, Maurice van Keulen, and Jens Teubner. Staircase join: Teach a relational DBMS to watch its (axis) steps. In *Proceedings of the 29th Int'l Conference on Very Large Databases (VLDB 2003)*, Berlin, Germany, Sep 2003.

[30] H. Hacıgümüş, B. Iyer, C. Li, and S. Mehrotra. SSQL: Secure SQL in an insecure environment. *VLDB journal*, 2006.

[31] H. Hacıgümüş, B. Iyer, and S. Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In *Proc. of the 9th International Conference on Database Systems for Advanced Applications*, Jeju Island, Korea, March 2004.

[32] H. Hacıgümüş, Balakrishna R. Iyer, Chen Li, and Sharad Mehrotra. Executing SQL over encrypted data in the database service provider model. In *SIGMOD Conference*, 2002.

[33] Hakan Hacıgümüş, Bala Iyer, and Sharad Mehrotra. Efficient execution of aggregation queries over encrypted relational databases. In YoonJoon Lee, Jianzhong Li, Kyu-Young Whang, and Doheon Lee, editors, *Database Systems for Advanced Applications: 9th International Conference, DASFAA 2004*, volume LNCS 2973, pages 125–136, Jeju Island, Korea, March 2003. Springer Verlag, Berlin.

[34] William George Horner. A new method of solving numerical equations of all orders, by continuous approximation. *Philosophical Transactions of the Royal Society of London*, pages 308–335, Jul. 1819.

[35] E. Kushilevitz and R. Ostrovsky. Replication is not needed: single database, computationally-private information retrieval. In *IEEE Symposium on Foundations of Computer Science*, pages 364–373, 1997.

[36] P. Lin and K. S. Candan. Ensuring privacy of tree structured data and queries from untrusted data stores. *Information Systems Security Journal*, May/June 2004.

[37] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography.* CRC Press, October 1996.

[38] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), april 19 1965.

[39] Rafail Ostrosky. Foundations of cryptography. Lecture 8, March 2005.

[40] Pascal Paillier and David Pointcheval. Efficient public-key cryptosystems provably secure against active adversaries. In *Advances in cryptology - AsiaCrypt*, volume LNCS 1716, pages 165–179. Springer Verlag, Berlin, 1999.

[41] J. M. Pollard. A Monte Carlo method for factorization. *BIT Numerical Mathematics*, 15(3):331–334, September 1975.

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

[43] A. R. Schmidt, F. Waas, M. L. Kersten, D. Florescu, I. Manolescu, M. J. Carey, and R. Busse. The XML Benchmark Project. Technical Report INS-R0103, CWI, Amsterdam, The Netherlands, April 2001. `http://monetdb.cwi.nl/xml/index.html`.

[44] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, November 1979.

[45] C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:379–423 and 623–656, July and October 1948.

[46] Dawn Xiaodong Song, David Wagner, and Adrian Perrig. Practical techniques for searches on encrypted data. In *IEEE Symposium on Security and Privacy*, pages 44–55, 2000.

[47] World Wide Web Consortium (W3C). XML path language (XPath) version 1.0, Nov 1999. `http://www.w3.org/TR/xpath`.

[48] World Wide Web Consortium (W3C). Xquery 1.0: An XML query language, Jan 2007. `http://www.w3.org/TR/xquery/`.

[49] David Wagner. In *Information Security*, volume LNCS 2851, pages 234–239. Springer-Verlag, Berlin, 2003.

[50] B. Waters, D. Balfanz, G. Durfee, , and D. K. Smetters. Building an encrypted and searchable audit log. In *Network and Distributed Security Symposium (NDSS) '04*, San Diego, California, 2004.

Index

Titles in the IPA Dissertation Series since 2002

Y. Qian. *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05

F. Bartels. *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06

L. Cruz-Filipe. *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07

E.H. Gerding. *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08

N. Goga. *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09

M. Niqui. *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10

A. Löh. *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11

I.C.M. Flinsenberg. *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12

R.J. Bril. *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13

J. Pang. *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14

F. Alkemade. *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15

E.O. Dijk. *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16

S.M. Orzan. *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17

M.M. Schrage. *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18

E. Eskenazi and A. Fyukov. *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19

P.J.L. Cuijpers. *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20

N.J.M. van den Nieuwelaar. *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

E. Ábrahám. *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

R. Ruimerman. *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

C.N. Chong. *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

H. Gao. *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04

H.M.A. van Beek. *Specification and Analysis of Internet Applications.* Faculty of Mathematics and Computer Science, TU/e. 2005-05

M.T. Ionita. *Scenario-Based System Architecting - A Systematic Approach to Developing Future-Proof System Architectures.* Faculty of Mathematics and Computing Sciences, TU/e. 2005-06

G. Lenzini. *Integration of Analysis Techniques in Security and Fault-Tolerance.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-07

I. Kurtev. *Adaptability of Model Transformations.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-08

T. Wolle. *Computational Aspects of Treewidth - Lower Bounds and Network Reliability.* Faculty of Science, UU. 2005-09

O. Tveretina. *Decision Procedures for Equality Logic with Uninterpreted Functions.* Faculty of Mathematics and Computer Science, TU/e. 2005-10

A.M.L. Liekens. *Evolution of Finite Populations in Dynamic Environments.* Faculty of Biomedical Engineering, TU/e. 2005-11

J. Eggermont. *Data Mining using Genetic Programming: Classification and Symbolic Regression.* Faculty of Mathematics and Natural Sciences, UL. 2005-12

B.J. Heeren. *Top Quality Type Error Messages.* Faculty of Science, UU. 2005-13

G.F. Frehse. *Compositional Verification of Hybrid Systems using Simulation Relations.* Faculty of Science, Mathematics and Computer Science, RU. 2005-14

M.R. Mousavi. *Structuring Structural Operational Semantics*. Faculty of Mathematics and Computer Science, TU/e. 2005-15

A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. Faculty of Mathematics and Computer Science, TU/e. 2005-16

T. Gelsema. *Effective Models for the Structure of pi-Calculus Processes with Replication*. Faculty of Mathematics and Natural Sciences, UL. 2005-17

P. Zoeteweij. *Composing Constraint Solvers*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-18

J.J. Vinju. *Analysis and Transformation of Source Code by Parsing and Rewriting*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2005-19

M.Valero Espada. *Modal Abstraction and Replication of Processes with Data*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2005-20

A. Dijkstra. *Stepping through Haskell*. Faculty of Science, UU. 2005-21

Y.W. Law. *Key management and link-layer security of wireless sensor networks: energy-efficient attack and defense*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-22

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of

Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data.* Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space.* Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices.* Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization.* Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML.* Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models.* Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications.* Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols.* Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition.* Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems.* Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra.*

Stellingen behorende bij het proefschrift getiteld ‘Searching in encrypted data’

R. Brinkman

01-06-2007

1. Hoewel we beveiliging en efficiëntie niet kunnen vergelijken, moeten ze wel met elkaar in evenwicht zijn.
2. Door de tweede wet van de thermodynamica kan de chaos en daarmee de veiligheid van de lucky dip alleen maar toenemen.
3. Indien een encryptiefunctie een homomorfe eigenschap bezit, dan kan dit worden gezien als een zwakte van de encryptiefunctie; toch kan deze zwakte de veiligheid juist vergroten.
4. Onderling wantrouwen is een goede basis om een geheim te delen.
5. Ook al wordt iedere slag tussen hackers en ontwikkelaars van Digital Rights Management systemen gewonnen door

de hackers, toch zullen het niet de hackers zijn die de strijd winnen.

6. De wereld bestaat uit 10 groepen mensen: zij die binair kunnen rekenen en zij die dit niet kunnen.

7. Aangezien de beste ideeën ontstaan op het grensgebied van diep nadenken en totale ontspanning moeten wetenschappers verplicht worden hun werk periodiek te verruilen voor ontspannende activiteiten.

8. Stress wordt niet veroorzaakt door het werk dat we doen, maar door het werk dat blijft liggen.

9. Om de wetgeving consistenter te maken is het noodzakelijk om naast de kopieerheffing op lege CD's en MP3-spelers ook verkeersboetes te gaan heffen bij de aanschaf van een auto die harder kan rijden dan de maximum snelheid.

10. Om de klap op het lichaam van een vallende klimmer te verkleinen, kan het gunstig zijn om de klimmer zover mogelijk te laten vallen.