2006

Symmetric private information retrieval via additive homomorphic probabilistic encryption

Laura Lincoln

ROCHESTER INSTITUTE OF TECHNOLOGY
Department of Computer Science

Symmetric Private Information Retrieval
via Additive Homomorphic Probabilistic Encryption

by

Laura Beth Lincoln
March 24, 2006

Thesis submitted in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Science

Stanisław Radziszowski, Chairman

Chris Homan, Reader

David Hart, Observer

Abstract

Suppose there is a movie you would be interested in watching via pay-per-view, but you refuse to purchase the feed because you believe that the supplier will sell your information to groups paying for the contact information of all the people who purchased that movie, and the association of your name to that purchase could hinder career, relationships, or increase the amount of time you spend cleaning SPAM out of your mailbox. Private Information Retrieval (PIR) will allow you to retrieve a particular feed without the supplier knowing which feed you actually got, and Symmetric Private Information Retrieval (SPIR) will assure the supplier, if the feeds are equally priced, that you received only the number of feeds you purchased. Now you can purchase without risking your name being associated with a particular feed and the supplier has gained the business of a once paranoid client.

The problem of SPIR can be achieved with the cryptographic primitive Oblivious Transfer (OT). Several approaches to constructing such protocols have been posed and proven to be secure. Most attempts have aimed at reducing the amount of communication, theoretically, but this thesis compares the computational expense of the algorithms through experimentation to show that reduction of communication is less valuable in the effort of achieving a practical protocol than reducing the amount of computation. Further, this thesis introduces new protocols to compete with previous published protocols that derive security from additive homomorphic probabilistic encryption schemes, and explores means to increase the length of data handled by these protocols so that the media is more useful and the time to complete the protocol is reasonable.

Acknowledgments

First, I would like to thank my committee members Staszek for his patience, thoroughness, and confidence, Chris for taking the time, when his life is so busy, to read through this longer than average thesis, and Professor Hart for his willingness to explore the subject matter.
Second, I would like to thank Matt for his persistent nagging, Mom for withholding fulfillment of her desire to visit me, so that I could have time to spend on my thesis, and Dad for not only helping with the proofreading, but for putting up with Mom.

Contents

1 Introduction **1**
1.1 Notations 4

2 Models of Secure Computation **5**
2.1 Models Based on the Honesty of the Parties 5
2.2 Models Based on the Computing Power of the Parties 8

3 A Branch of Probabilistic Encryption **11**
3.1 Background and Definitions 11
3.2 Encryption Schemes 15

4 Previous PIR Protocols **27**
4.1 Generic Protocol 27
4.2 Stern Protocol 29
4.3 Chang Protocol 32
4.4 Lipmaa Protocol 38

5 New PIR Protocols **43**
5.1 Extended Chang Protocol 43
5.2 Generalized Chang Protocol 46
5.3 Chang Protocol with Pre-Split Elements 48
5.4 A Collaboration of the Chang and Lipmaa Protocols 51

6 Comparisons **55**
6.1 Theoretical Comparison 55
6.2 Experimental Comparison 67
6.3 Conclusions and Future Work 71

A Previous PIR Implementation Details **77**
A.1 Stern Protocol 77
A.2 Chang Protocol 78
A.3 Lipmaa Protocol 80

B New PIR Implementation Details **83**
B.1 Extended Chang Protocol 83
B.2 Generalized Chang Protocol 84
B.3 Chang Protocol with Pre-Splitting Elements 86
B.4 A Collaboration of the Chang and LipmaaProtocols 87

C Implementation Documentation **91**
C.1 Package edu.rit.cryptosystems 92
C.2 Client-Side Library 120
C.3 Server-Side Library 133

D Scripts **159**
D.1 Stern Experiment Script 159
D.2 Chang Experiment Script 160
D.3 Pre-Split Experiment Script 161
D.4 Extended Experiment Script 161
D.5 Generalized Experiment Script 162
D.6 Lipmaa Experiment Script 163
D.7 Lipmaa versus Collaboration Experiment Script 165
D.8 Collaboration Experiment Script 166
D.9 Result Collection Script 167

Chapter 1

Introduction

There exist many databases, both public and private, and there exist several users that may be hesitant to obtain information from these sources for various reasons involving the preservation of the client's privacy. The server's knowledge of the data retrieved by the client could be used in a malicious manner such as unwanted profiling, spying, or blackmail. If the database is public, then a Private Information Retrieval Protocol (PIR) could be used to maintain the client's privacy. Private information retrieval is oblivious transfer with the privacy of the server removed and is generally addressed in a more practical setting.

Definition 1.0.1 (Private Information Retrieval [CKGS98]) *A scheme with a protocol which satisfies the following properties*

Inputs	*Client*	$i*$ *the desired index*
	Server	x *the server-side database*
Outputs	*Client*	x_{i*} *the entry in the server-side database at the desired index*
	Server	*Nothing*
Privacy	*Client*	*Received entry unknown to the server*
	Server	*No privacy*

Since the server has no privacy to maintain, the trivial solution is for the client to download the entire database and then read the desired entries. Thus, the goal of private information retrieval research is to reduce the communication complexity. An example could be Google's GMail, where a user's e-mail is scanned, and textual ads are retrieved from Google's database based on the text of the e-mail. The e-mail text could be viewed by the database administrator, or person with access to the data and transaction logs of the server, and when a user pursues a resulting ad, the text of the e-mail could be inferred by someone with access to the database or someone with access to the transaction logs of a destination site linked by one of the chosen ads. This scenario could be mapped to a potential PIR protocol. The e-mail text could be viewed as the private query, and the textual ads retrieved from Google's ad database could be the privately retrieved result. This would also assume the added quality of Private

Information Retrieval by Keyword [CGN]. This term is often confused with the symmetric version.

Symmetric private information retrieval is the setting for the protocols described in this thesis, at least in the semi-honest model with computationally bounded client and server.

Definition 1.0.2 (Symmetric Private Information Retrieval [GYKM00]) *A scheme with a protocol which satisfies the following properties*

Inputs	*Client*	$i*$ *the desired index*
	Server	x *the private server-side database*
Outputs	*Client*	x_{i*} *the entry in the server-side database at the desired index*
	Server	*Nothing*
Privacy	*Client*	*Received entry unknown to the server*
	Server	*Client can only acquire the amount of information allowed*

The symmetric property of this scheme allows the database to be private, and it allows for the restricting of the number of records a given client may retrieve. This is quite an enticing property for e-commerce scenarios. However, ensuring the strict privacy required of a symmetric private information retrieval protocol is potentially very costly both computationally and in the amount of information communicated during the protocol. Several papers have attempted to use symmetric private information retrieval in several e-commerce scenarios [KBS02, BDF00, Bea97], but some have assumed that anonymous retrieval is equivalent to private retrieval. Anonymity is not restricted, but simply hiding the identity of the client does not ensure that the entry the client retrieved was maintained as private [KBS02].

In [Rab81], Rabin introduced the cryptographic primitive [Kil88], Oblivious Transfer (OT), which has also been presented by Yao as Oblivious Circuit Evaluation in [Yao90]. SPIR, by definition can be achieved by the use of Oblivious Transfer.

Definition 1.0.3 (Oblivious Transfer) *An oblivious transfer (OT) scheme is a protocol in which the sending party sends a bit to the receiving party in a way such that the receiving party receives the bit with probability of* $\frac{1}{2}$ *without the sending party knowing whether or not the receiving party received the correct bit. This is equivalent to the sending party having two bits and the receiving party will receive one of the bits without the sending party knowing which of the two bits was received. This protocol is often denoted* OT_1^2*, 1-out-of-2 oblivious transfer.*

There have been many flavors of OT protocols: hardware-based [Aso03], private information retrieval by keyword [CGN], quantum PIR [KdW03], multiple database PIR [CKGS98], PIR with preprocessing [BYT], random sequence retrieval [NP99], etc.

Most attempts have aimed at reducing the amount of communication, theoretically, but this thesis compares the computational expense of the algorithms through experimentation to show that reduction of communication is less valuable in the effort of achieving a practical protocol than reducing the amount of computation. Further, this thesis introduces new protocols to compete with previous published protocols that derive security from additive homomorphic probabilistic encryption schemes, and explores means to increase the length of data handled by these protocols so that the media is more useful and the time to complete the protocol is reasonable.

This thesis will explore the additive homomorphic probabilistic encryption scheme based SPIR protocols with computationally bounded client and server. Encryption schemes explored derive security from the following: quadratic residues, prime residues, p-subgroups, and composite residues. These protocols have evolved through variation and this thesis continues the evolutionary process. Through experimental comparison of these variations, the proof of previous theoretical comparisons will be justified or nullified, and the issue of whether SPIR can be used for practical means will be addressed. As will be shown, given the existing technology, these protocols are impractical, and where most have considered the reduction of computation as a means to developing practical protocols, protocols which reduce computation rather than communication will lead to a more practical solution.

The remainder of this chapter present the notations that will be used through out this thesis. Chapter 2 will provide a brief overview of some secure computational models. Chapter 3 provides a survey of additive homomorphic probabilistic encryption schemes emphasizing the evolution of such schemes. Chapter 4 provides a survey of published PIR protocols that are based on additive homomorphic probabilistic encryption schemes. In Chapter 5 we present new protocols based on the previously published protocols, and in Chapter 6 we compare the protocols theoretically and then through experimentation we see that previous assumptions of communication as the bottleneck of PIR protocols are unjustified.

1.1 Notations

Symbol	Definition
$\|\alpha\|$	order
P_k	public key
S_k	secret (Private) key
$\mathcal{D}$	decryption function
$\mathcal{E}$	encryption function
$m_{[i]}$	a plaintext message
$\mu_{[i]}$	a ciphertext message
r_i	randomly chosen integer
$\mathcal{PT}$	space containing all possible plaintexts
$\mathcal{CT}$	space containing all possible ciphertexts
N	RSA composite, where $N = pq$ and p and q are prime
c	# of dimensions
s	maximum exponent, used by length-flexible cryptosystem
t	# of parts
ν	modulus of ciphertext domain
K	# of elements in the server-side private database
$\{x_i\}_{i=0}^{K-1}$	linear server-side private database
x_i	database element at the i^{th} index
$i*$	index of element in private server-side database desired by the client
$q_{i[j]}$	i^{th} element of Q, an encrypted $0, 1$-indicator, j
Q	private query generated by client
R_s	server-side result

Chapter 2

Models of Secure Computation

When constructing secure multi-party communication protocols, it is helpful to construct the protocol to be secure within a particular model of secure computation. Attempts can then be made to modify a protocol that is secure in one model of secure computation to be secure in a model that allows for parties to either perform more malicious actions before, during, or after the protocol or to allow parties to have greater computational power. This suggests that there are at least two different types of models of secure computation. The two models considered here are those models based on the honesty of the parties, and those models based on the computational prowess of the parties. The protocols explored in this thesis, however, are all secure within the semi-honest model provided that the parties are computationally bounded.

2.1 Models Based on the Honesty of the Parties

This section discusses three models of secure computation that are based on the honesty of the parties, but considering the scenario where only two parties participate and that each party may be honest, semi-honest, or malicious, a protocol could be constructed in which the client and server participate in the protocol with different levels of honesty. The Ideal model is not dependent on the honesty of either participating party because of a trusted third party. The ideal model serves as a goal for all secure computations to meet for both computational and communication complexities. If all participating parties are honest, and all the parties trusted one another, then there is no need to construct a protocol other than the trivial non-secure protocol. In the event that one out of the two parties were honest, then the honest party could serve as the trusted third party in the Ideal model. Thus, when constructing protocols to be secure within a particular model of secure computation based on the honesty of the parties, there really exist four possible models: both parties are semi-honest,

both parties are malicious, the client is semi-honest and the server is malicious, and the server is semi-honest while the client is malicious, where a semi-honest party follows the protocol without deviation except for maintaining records of the entire transaction for later cryptanalysis, and a malicious party may deviate from the protocol at anytime.

2.1.1 Ideal

The ideal model in the two party scenario consists of the two parties that wish to communicate and a trusted third party. A trusted third party is a party apart from the two communicating parties that will leak no information to any opposing communicating party concerning the party's private input or the party's resulting output from the computation. The trusted third party is an intermediary who can see the private inputs and outputs resulting from the private computation, but the trusted third party will do nothing with the inputs other than use them to perform the computation, and will do nothing with the resulting outputs other than send the outputs to the appropriate communicating party. Also note that the channels of communication are assumed to be secure. The standard two-party ideal model protocol proceeds as follows [Gol]:

Sending Inputs to Trusted Third Party Each of the two communicating parties sends his private input to the trusted third party as follows:

Honest First [Second] Party Sends input x [y].

Malicious [Second] Party Send input $x' \in \{0,1\}^{|x|}$ [$y' \in \{0,1\}^{|y|}$].

Trusted Third Party Responds to First Communicating Party Trusted third party answers the first communicating party from inputs (x, y) as follows:

Assume Honest Responds with $f_1(x, y)$.

Assume Malicious Responds with $\perp$.

Trusted Third Party Responds to Second Communicating Party Trusted third party answers the second communicating party as follows:

Malicious First Party In the event the first communicating party is not satisfied with the output and/or was malicious at input phase $\rightarrow$ replies with $\perp$ to second communicating party.

Otherwise Responds with $f_2(x, y)$.

Output Resulting output should be as follows:

Honest Party Message from trusted third party.

Malicious Party Arbitrary function of intial inputs and message from trusted third party.

It would be ideal to have an extraneous party which all communicating parties can trust to maintain the privacy of each party's inputs, perform the correct computation, and send the appropriate output to each party. Provided that the communication is performed over private channels, this model requires no extra computations to hide the information. However, if two or more parties cannot trust one another, how can the communicating parties trust a party extraneous from the communicating group. It is indeed ideal, thus the name of this model, to have a third party. Although the ideal model is near impossible to achieve, the ideal model is not necessarily a model in which to construct new secure multi-party protocols. Rather, this model serves as a metric for which all protocols designed in the other two secure computation models based on the honesty of each party will be compared.

2.1.2 Semi-Honest

The semi-honest model involves one or more semi-honest parties, and the remaining parties, if any, are honest. A semi-honest party is one who follows the protocol properly with the exception that he keeps record of all intermediate computations [Gol].

Almost all secure multi-party protocol constructions are designed initially to meet the correctness and privacy requirements of the ideal model within the semi-honest model, and for most of these constructions, the protocol does not proceed to satisfying the requirements of the ideal model within the malicious model, but the semi-honest model can satisfy some real world situations.

Most users may not have the knowledge base to deviate from the protocol. Most user may not also have access to the same resources as a malicious party might due to lack of user level priviliges. Transparency levels in the party's operating system may contribute to a lack of access to register and memory contents. The party could be forced to use only the application provided by the other party to participate in the protocol. Thus, given that parties are restricted to behaving in a semi-honest manner due to the party's inability to act maliciously, a semi-honest protocol, which is much easier to construct and implement than a corresponding protocol in the malicous model, will suffice in most real world situations.

2.1.3 Malicious

The malicious model consists of one or more malicious parties which may attempt to deviate from the protocol in any manner. A protocol constructed in the malicious model must provide the same level of correctness and privacy as the corresponding protocol in the ideal model. The malicious model protocol should yield it impossible for a malicious party to deviate from the protocol save for the following possibilities [Gol]:

- A party may refuse to participate in the protocol when the protocol is first invoked.

- A party may substitute his local input by entering the protocol with an input other than the one provided to them.
- A party may abort prematurely.

Notice that the ideal model protocol accounts for the above three scenarios by returning $\perp$. If a party refuses to participate in the protocol, then the protocol is not executed. If the party substitutes his local input, then he has chosen to participate in the multi-party computation with a different input, and since the other participating parties are not permitted to discover another party's input, it is even more unreasonable to know another party's intended input, and a machine obviously cannot determine a participating party's intentions, thus the substitution of a local input is permissible. Premature abortion is an allowable deviation because it is improbable to prevent. A party's system could fail, a connection could go down, or a party could decide that he no longer wishes to continue participation in the protocol. Since it is infeasible for any protocol to prevent these three scenarios [Gol], these three scenarios need not be prevented.

2.2 Models Based on the Computing Power of the Parties

Models concerning the computational power of the parties are also often considered for measuring the security of secure multi-party protocols. Since most cryptographic operations are computationally expensive, the computational power of the parties can be considered to simplify the scheme by increasing or decreasing the amount of communication or by using weaker or stronger cryptographic operations so long as the cryptographic operations are secure against adversaries with a defined level of computational power.

2.2.1 Information Theoretically Secure

If a party is computationally unbounded, then if given enough information, no matter how strongly encrypted, unless something unconditionally secure is used, the party will be able to derive the answer. A protocol is information theoretically secure if it can maintain the specified privacy of all the parties when one or more parties is computationally unbounded. Thus the protocol cannot allow a computationally unbounded party to receive more information, whether encrypted or not, than is allowed by the specified privacy.

2.2.2 Computationally Secure

If a party is computationally bounded, then the party can only view information that is received in plaintext, information which the party can decrypt because the party possesses the keys to decrypt the information or the information that is encrypted with a weak encryption scheme. A weak encryption scheme in this case is when the plaintext form of the information encrypted by this encryption

scheme can be extracted by the party without the keys to decrypt the information because the computational power of the party is great enough. Thus a protocol is computationally secure if the information that is not to be received by a party, according to the specified privacy policy, is encrypted with a scheme that the party cannot break with its bounded computational power.

Chapter 3

A Branch of Probabilistic Encryption

Probabilistic encryption was introduced by Goldwasser and Micali in [GM84], and has since split into multiple branches. The most popular branch, consisting of public-key encryption schemes such as ElGamal and Elliptic Curves, derives security from the Computational Diffie-Hellman Assumption, but Diffie-Hellman based schemes allow only for plaintext in multiplicative groups. Another branch, gaining popularity in academic research, derives security from Residue Class Problems. This branch allows for plaintext in additive groups. Additive groups are desirable domains for plaintext over multiplicative groups due to the presence of 0 as a possible plaintext. With bit-by-bit encryption of $\{0,1\}^*$, one can send encrypted indicators of desirable indices to accomplish oblivious transfer, or could tally the number of yes votes to determine the majority vote. This branch of probabilistic encryption, the branch of encryption schemes acting on plaintexts in additive groups that derive security from the intractibility of particular residue class problems, has been used in many research papers. The following chapter is one of few, if any, outlining the evolution of this branch.

3.1 Background and Definitions

Around 1976, Martin Hellman, Ralph Merkle, Whitfield Diffie, and potentially others, developed the notion of public-key encryption [MVO96].

Definition 3.1.1 (Public-Key Encryption) *A cryptosystem which uses two keys, a public key and a private key. The public key and the algorithm for encryption and decryption, may be published so that others may send encrypted messages. The private key is kept secret by the receiver for decryption of messages encrypted by the corresponding public key. The private key cannot be*

easily discovered from knowledge of the public key or encryption and decryption algorithms. Also known as Asymmetric Encryption.

The development of public-key cryptography alleviated the key distribution issues inherent with private-key cryptosystems. Private-key cryptosystems rely on maintaining the privacy of a single private key which is used for both encryption and decryption. Thus, to receive an encrypted message from a sender, the private-key must be shared with that sender by non-public means. But how can the key be shared and not revealed to undesirable senders? Public-key absolves this issue by allowing for the encryption key to be made public without compromising security. The security of most public-key cryptosystems is dependent upon at least one computational assumption.

In 1977, Rivest, Shamir, and Adleman developed the *RSA Cryptosystem*, see [Sti02]. The public key can be defined as (a, N) and the private key can be defined as (b, p, q) where $N = pq$ and $ab \equiv 1 \mod \phi(N)$ for primes p and q. $\phi(\cdot)$ is the Euler-ϕ function where $\phi(N) = (p-1)(q-1)$. Without knowledge of p and q, it is impractical to determine $\phi(N)$, and thus it is difficult to determine b from knowledge of just a and N. It can be easily seen that the security of the RSA cryptosystem is dependent upon the difficulty of factoring large numbers. RSA is one example of a deterministic cryptosystem.

Definition 3.1.2 (Deterministic Encryption) *Given security parameter n, let $\mathcal{PT}_n$ denote the finite domain consisting of all possible plaintexts, and $\mathcal{CT}_n$ denote the finite domain consisting of all possible ciphertexts. A deterministic encryption scheme consists of a one-to-one function $\mathcal{E}\colon \mathcal{PT}_n \mapsto \mathcal{CT}_n$.*

Consider $\mathcal{E}$ to be the encryption algorithm and $\mathcal{D}$ to be the decryption algorithm. Using the keys described above, a plaintext m can be encrypted with $\mathcal{E}(m) = m^b \mod n$. Thus, RSA is a deterministic encryption scheme. Deterministic encryption schemes lack semantic security.

Definition 3.1.3 (Semantic Security) *Given two messages, m_0 and m_1, of equal-size, generated by a computationally-bounded adversary. If the adversary sends these two messages to an encryption oracle which chooses one of these two messages at random to encrypt and return the generated ciphertext to the adversary and the adversary cannot distinguish which of the original messages the returned ciphertext relates to with probability greater than $\frac{1}{2}$, then the encryption algorithm used by the encryption oracle is said to be semantically secure.*

However, probabilistic encryption schemes do not lack semantic security. Introduced in [GM84] by Goldwasser and Micali, probabilistic encryption introduces randomness during the encryption algorithm, so that various ciphertexts can be generated from one plaintext. A string of bits can be sent encrypted bit-by-bit via a deterministic encryption scheme, but an eavesdropper could look at the string of encrypted values and have a $\frac{1}{2}$ chance of guessing the entire string, but the same string encrypted with a probabilistic encryption scheme can be intercepted by an eavesdropper, and the eavesdropper would only have a

$\frac{1}{2}$ chance of guessing a single bit right, so given a bit string of length k encrypted with a probabilistic encryption scheme, an eavesdropper would now only have a $\left(\frac{1}{2}\right)^k$ chance of guessing the entire string. Thus, the introduction of randomness in an encryption scheme ensures semantic security since knowledge of one bit does not imply knowledge of any other bit encrypted by the same encryption scheme.

Definition 3.1.4 (Probabilistic Encryption) *Given security parameter n, let $\mathcal{PT}_n$ denote the finite domain consisting of all possible plaintexts, $\mathcal{RT}_n$ denote the finite domain consisting of all possible randomness, and $\mathcal{CT}_n$ denote the finite domain consisting of all possible ciphertexts. A probabilistic encryption scheme consists of a function $\mathcal{E}\colon \mathcal{PT}_n \times \mathcal{RT}_n \mapsto \mathcal{CT}_n$ such that $\exists \mu_1, \mu_2 \in \mathcal{CT}_n$ where $\mu_1 \neq \mu_2$ but $\mathcal{D}(\mu_1) = \mathcal{D}(\mu_2)$ is possible.*

When Goldwasser and Micali introduced probabilistic encryption in [GM84], the encryption scheme presented was based on the difficulty of computing residue classes, quadratic residues in particular, and led to the development of many other probabilistic encryption schemes based on the difficulty of computing residue classes of other functions within a group.

Definition 3.1.5 (Residue Class) *The residue classes of a function $f(x) \mod n$ are all possible values of the residue, or congruence, $f(x) \mod n$.*

Example 3.1.1 *The residue classes of $x^2 \mod 6$ are $\{0, 1, 3, 4\}$.*

A year later, [Gam85] introduced the ElGamal encryption scheme. The security of the ElGamal encryption scheme relies upon the Computational Diffie-Hellman assumption, or difficulty of computing discrete logarithms.

Assumption 3.1.1 (Diffie-Hellman Assumption [Sti02]) *Given a multiplicative cyclic group $(G, \cdot)$, an element $\alpha \in G$ having order n, and two elements $\beta, \gamma \in \langle\alpha\rangle$. There is no known polynomial time algorithm to compute $\delta \in \langle\alpha\rangle$ such that $log_\alpha \delta \equiv \log_\alpha \beta \times \log_\alpha \gamma \mod N$.*

ElGamal is defined as follows where p is a prime such that it is infeasible to solve a discrete logarithm in $\mathbb{Z}_p^*$.

Encryption Scheme 3.1.1 (ElGamal [Sti02])

Plaintext Space:	$\mathbb{Z}_p^*$
Random Space:	$\mathbb{Z}_p^*$
Ciphertext Space:	$\mathbb{Z}_p^* \times \mathbb{Z}_p^*$
Public Key:	p
	$\alpha \in \mathbb{Z}_p^*$ *where* $\lvert\alpha\rvert = p-1$
	$\beta = \alpha^a \mod p$
Private Key:	a
Encryption:	$\mathcal{E}(m,r) = (\mu_1, \mu_2)$
	$\mu_1 = \alpha^r \mod p$
	$\mu_2 = x\beta^r \mod p$
Decryption:	$\mathcal{D}(\mu_1, \mu_2) = m_2({m_1}^a)^{-1} \mod p$

The Goldwasser-Micali encryption scheme and the El Gamal encryption scheme pioneered the two major branches of probabilistic encryption. Attempts were made using error correcting codes [McE78], lattice problems [NS98b], additive and multiplicative knapsack-type systems [MH88, Vau98, NS97] and multivariate polynomials [MI88, Pat97] to create further types of probabilistic encryption schemes, but these schemes were cryptanalyzed and found to suffer inefficiencies, security weakness, or lack of public scrutiny.

Basing the security of an encryption scheme on assumptions such as the difficulty of computing particular residue classes for certain functions and the difficulty of computing discrete logarithms leads towards the development of encryption schemes that are not just easily adapted to probabilistic encryption schemes, but also leads to encryption schemes that have a homomorphic property.

Definition 3.1.6 (Group Homomorphism) *A group homomorphism is a mapping* $\phi\colon G \mapsto H$ *between two groups such that*

- $f(g_1 g_2) = f(g_1) f(g_2)$ *and*
- $f(e_G) = e_H$

When discussing a probabilistic encryption scheme as being homomorphic, the first property can be stated more loosely as

$$\mathcal{E}(m_1 \oplus m_2, r_3) = \mathcal{E}(m_1, r_1) \otimes \mathcal{E}(m_2, r_2)$$

where $\oplus$ and $\otimes$ can be any valid operations.

Example 3.1.2 *The ElGamal encryption scheme, is multiplicatively homomorphic in that* $\mathcal{E}(m_1 \cdot m_2, r_1 + r_2) = \mathcal{E}(m_1, r_1) \cdot \mathcal{E}(m_2, r_2)$. *Since, using componentwise multiplication*

$$\begin{aligned}
\mathcal{E}(m_1, r_1) \cdot \mathcal{E}(m_2, r_2) &= (\alpha^{r_1}, m_1\beta^{r_1}) \cdot (\alpha^{r_2}, m_2\beta^{r_2}) \\
&= (\alpha^{r_1}\alpha^{r_2}, m_1\beta^{r_1}m_2\beta^{r_2}) \\
&= (\alpha^{r_1}\alpha^{r_2}, m_1m_2\beta^{r_1}\beta^{r_2}) \\
&= (\alpha^{r_1+r_2}, m_1m_2\beta^{r_1+r_2}) \\
&= \mathcal{E}(m_1 \cdot m_2, r_1 + r_2)
\end{aligned}$$

Refer to [Sti02] for the details of the El Gamal encryption scheme.

3.2 Encryption Schemes

The following sections will continue with the branch of probabilistic encryption schemes which derive security from the difficulty of computing residue classes of particular functions in a fashion that highlights each schemes semantic security and additive homomorphic property.

3.2.1 Goldwasser-Micali Encryption Scheme

The Goldwasser-Micali encryption scheme was developed in [GM84], and is regarded as the first probabilistic encryption scheme. The scheme encrypts a single bit and takes a random element of the group $\mathbb{Z}_N^*$to produce a ciphertext within the group $\mathbb{Z}_N^*$where $N = pq$ is an RSA composite, such that p and q are prime. The encryption function requires a fixed element y which is a quadratic nonresidue of $\mathbb{Z}_N^*$.

Definition 3.2.1 (Quadratic Residue) *Let p be an odd prime and $gcd(a, p) = 1$. If $x^2 \equiv a \mod p$ has a solution, then a is a quadratic residue. Otherwise, a is called a quadratic nonresidue modulo p.*

It is easy to see that the product of two quadratic residues is a quadratic residue and that the product of a quadratic residue and a quadratic nonresidue is a quadratic nonresidue. Also, for all numbers $r \in \mathbb{Z}_N^*$, r^2 is a quadratic residue of $\mathbb{Z}_N^*$. Since we chose a fixed element $g \in \mathbb{Z}_N^*$ that is a quadratic nonresidue, raise g to a power equal to the plaintext x, then if $m = 0$, $g^0 = 1$ which is clearly a quadratic residue, and if $m = 1$, $g^1 = g$ which is clearly a quadratic nonresidue. Since r^2 is a quadratic residue, the product $r^2 \cdot g^m$ is a quadratic residue if $m = 0$, and is a quadratic nonresidue if $m = 1$. Thus, a logical encryption function would be $\mathcal{E}(m, r) = r^2g^m \mod N$.

Decryption of the ciphertext is simply a matter of determining whether or not the ciphertext is a quadratic residue or quadratic nonresidue in $\mathbb{Z}_N^*$. If the factorization of N is known, then Legendre's symbol can be used to determine whether the ciphertext μ is a quadratic residue or a quadratic nonresidue.

Definition 3.2.2 (Legendre's Symbol) *For prime p*

$$\left(\frac{a}{p}\right) = \begin{cases} 1 \text{ if } a \text{ is a quadratic residue} \\ 0 \text{ if } a|p \\ -1 \text{ if } a \text{ is a quadratic nonresidue} \end{cases}$$

PROPERTIES

1. *If* $a \equiv b \mod p$, *then* $\left(\frac{a}{p}\right) = \left(\frac{b}{p}\right)$.
2. $\left(\frac{a^2}{p}\right) = 1$
3. $\left(\frac{a}{p}\right) \equiv a^{\frac{p-1}{2}} \mod p$
4. $\left(\frac{ab}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b}{p}\right)$
5. $\left(\frac{1}{p}\right) = 1$ *and* $\left(\frac{-1}{p}\right) = (-1)^{\frac{p-1}{2}}$
6. $\left(\frac{ab^2}{p}\right) = \left(\frac{a}{p}\right)\left(\frac{b^2}{p}\right) = \left(\frac{a}{p}\right)$

Example 3.2.1 *[Bur02] Is* -46 *a quadratic residue* $\mod 17$*?*

$$\begin{aligned} \left(\frac{-46}{17}\right) &= \left(\frac{-1}{17}\right)\left(\frac{46}{17}\right) \text{ by property 4} \\ &= \left(\frac{46}{17}\right) = \left(\frac{12}{17}\right) \text{ by property 5} \\ &= \left(\frac{3 \cdot 2^2}{17}\right) = \left(\frac{3}{17}\right) \text{by property 6} \\ &\equiv 3^{\frac{17-1}{2}} \equiv 3^8 \equiv 81^2 \equiv (-4)^2 \equiv -1 \mod 17 \end{aligned}$$

Thus -46 *is a quadratic nonresidue* $\mod 17$.

To encrypt a plaintext m, g and N must be public. The random number $r \in \mathbb{Z}_N^*$ is unnecessary for decryption and will be generated for each encryption. For decryption, there must be knowledge of N's prime factors, p and q. Thus, p and q must be the private keys in the Goldwasser-Micali encryption scheme. Without the prime factors of N, it is difficult to determine whether or not the ciphertext μ is a quadratic residue or a quadratic nonresidue in $\mathbb{Z}_N^*$. Thus, the security of the Goldwasser-Micali cryptosystem rests upon the Quadratic Residuosity Assumption.

Assumption 3.2.1 (Quadratic Residuosity) *Any polynomial-time algorithm, or polynomial-sized circuit, can only accurately determine whether a negligible fraction of values are quadratic residues modulo N. Where $N = pq$, a RSA composite.*

In summary, the Goldwasser-Micali cryptosystem is as follows.

Encryption Scheme 3.2.1 (Goldwasser-Micali)

Plaintext Space:	$\{0,1\}$
Random Space:	$\mathbb{Z}_N^*$
Ciphertext Space:	$\mathbb{Z}_N^*$
Public Key:	g, N
Private Key:	p, q
Encryption:	$\mu = r^2 g^m \mod N$
Decryption:	$\left(\frac{\mu}{p}\right)$

Note:

$$N = pq \text{ where } p, q \text{ are } k\text{-bit primes}$$

$$g \not\equiv m^2 \mod N$$

$$\left(\frac{\mu}{p}\right) = \begin{cases} 1, & m = 0 \\ \text{-1}, & m = 1 \end{cases}$$

3.2.2 Benaloh Encryption Scheme

In [Ben87], the Benaloh encryption scheme was introduced. The construction of the Benaloh encryption scheme is buried in Benaloh's doctoral thesis which concerns electronic voting. Thus it may be simpler to read this section after Section 3.2.3, which discusses the descendant encryption scheme, Naccache-Stern. The Benaloh encryption scheme performs computation in prime residue groups.

Definition 3.2.3 (Prime Residue Group) *A multiplicative subgroup G of the group $\mathbb{Z}_N$, where $p_i \in G$ if* $\gcd(p_i, N) = 1$.

Some Properties

- $|G| = \phi(N)$
- *G is cyclic if and only if $N = 2$, 4, p^e, or $2p^e$*

The basis of security in the Benaloh encryption scheme is the prime residuosity assumption.

Assumption 3.2.2 (Prime Residuosity) *For all p where p is relatively prime to N, there does not exist a probabilistic or deterministic algorithm which decides p^{th} residues in polynomial time.*

Let public key tuple be (g, k, N) where $g \neq w^k \mod N$ when k is a prime divisor of $\phi(N)$, but k^2 is not a prime divisor of $\phi(N)$, $w \in \mathbb{Z}_N^*$, and $N = p \cdot q$ is an RSA composite. Let the private key tuple consist solely of $\phi(N)$. Consider y

to be a wisely chosen constant analogous to g in the Naccahe-Stern encryption scheme and k analogous to σ. The encryption function follows the common form of all encryption schemes discussed in this chapter.

$$\mathcal{E}(m) = r^k g^m \mod N$$

To recover m, the following decryption function $\mathcal{D}(\mu)$ can be applied

$$\mathcal{D}(\mu) = \left(\log_g \mu^{\frac{\phi(N)}{k}}\right) \cdot \frac{k}{\phi(N)}$$

since

$$\mu^{\frac{\phi(N)}{k}} = r^{\phi(N)} \cdot g^{\frac{m \cdot \phi(N)}{k}}$$

Key generation can mimic the key generation presented as an example in [NS98a]. Select ℓ randomly chosen distinct primes such that when $\{p\}_\ell$ are partitioned into two sets of equal length, which will be referred to as $\{p_1\}_{\frac{\ell}{2}}$ and $\{p_2\}_{\frac{\ell}{2}}$. Now set

$$N = \left(\left(\prod_{i=1}^{\frac{\ell}{2}} p_{1i}\right) + 1\right) \cdot \left(\left(\prod_{i=1}^{\frac{\ell}{2}} p_{2i}\right) + 1\right)$$

so that N is essentially the product of two distinct primes. Thus

$$\phi(N) = \left(\prod_{i=1}^{\frac{\ell}{2}} p_{1i}\right) \cdot \left(\prod_{i=1}^{\frac{\ell}{2}} p_{2i}\right)$$

Finally, let $k = p_{ij}$ where $1 \leq i \leq 2$ and $1 \leq j \leq \frac{\ell}{2}$ and i and j are chosen at random. It is easily seen that $k \mid \phi(N)$ and $k^2 \nmid \phi(N)$. However, N can be often factored with the Pollard-Rho algorithm, see Stinson [Sti02] for description.

In summary, the Benaloh cryptosystem is as follows.

Encryption Scheme 3.2.2 (Benaloh[Ben87])

Plaintext Space:	$m < k$
Random Space:	$\mathbb{Z}_N^*$
Ciphertext Space:	$\mathbb{Z}_N^*$
Public Key:	g, k, N
Private Key:	$\phi(N)$
Encryption:	$\mu = r^k g^m \mod N$
Decryption:	$m = (\log_g \mu^{\frac{\phi(N)}{k}}) \cdot \frac{k}{\phi(N)}$

3.2.3 Naccache-Stern Encryption Scheme

The Naccache-Stern encryption scheme was introduced in [NS98a]. The security of the Naccache-Stern encryption system rests upon the *Higher Residuosity Assumption*.

Assumption 3.2.3 (Higher Residuosity) *It is difficult to distinguish integers of the form* $x^p \mod N$ *when* p *is a prime divisor of* $\phi(N)$.

The Naccache-Stern encryption scheme can be viewed as multiplicative version of the basic (additive) knapsack problem. The transformation of the basic knapsack problem to a multiplicative one is made possible by combining techniques from the multiplicative Merkle-Hellmen Knapsack and Pohlig-Hellman's secret-key cryptosystem. That is, the notion of super increasing sets and determining discrete logarithms via the Pohlig-Hellman algorithm supplement the inner workings of the Naccache-Stern encryption scheme.

Let the public key be represented by the tuple (g, σ, N) and the private key be represented by the tuple $(\phi(N), \{p\}_i, \{q\}_i)$. Choose a k so that $N = pq$, where $p = \left(\prod_{i=0}^{k} p_i\right) + 1$ and $q = \left(\prod_{i=0}^{k} q_i\right) + 1$ are distinct primes, such that the sets p_k and q_k contain distinct primes and the intersection of both sets is empty. Set $\sigma = \frac{\prod_{i=0}^{k} p_i q_i}{p_m \cdot q_n}$. Let $\phi(\cdot)$ denote the Euler-ϕ function, so that $\phi(N) = (p-1)(q-1) = \prod_{i=0}^{k} p_i q_i$. Now choose a g such that the multiplicative order modulo $pq+1$ of g is a multiple of σ. This, however, could make Naccache-Stern breakable by $p-1$ factoring.

Considering the computational assumption concerning higher residuosity, since it is difficult to distinguish integers of the form $x^p \mod N$ when p is a prime divisor of $\phi(N)$ and the prime factorization of σ consists of all but two prime divisors of $\phi(N)$, σ is roughly analogous to 2 in the Goldwasser-Micali encryption scheme. Also, g was chosen wisely in the Goldwasser-Micali encryption scheme to be a quadratic non-residue of N, but here it is more fitting for g to be chosen wisely as having a multiplicative order modulo $pq+1$ which is a multiple of σ. Without loss of generality, we can maintain the form of the encryption function from the previously discussed encryption schemes by letting $\mathcal{E}(m) = r^{\sigma} g^m \mod N$.

Pohlig-Hellman's algorithm for computing discrete logs, helps to compute the discrete logarithm modulus N where N is a family of distinct primes as follows.

Algorithm 3.2.1 (Pohlig-Hellman) *[Sti02] Consider*

$$N = \prod_{i=1}^{k} (p_i)^{c_i}$$

where the p_i*'s are distinct primes. The value of* $a = \log_{\alpha} \beta$ *can be uniquely determined* $\mod N$ *by observing that we can compute* $a \mod p_i^{c_i}$ *for* $1 \leq i \leq k$ *and then compute* $a \mod N$ *by the Chinese Remainder Theorem.*

Since σ was chosen as a family of distinct primes, the set of distinct primes remains private, and we wish to have an encryption scheme with security dependent upon the Higher Residuosity Assumption, the decryption algorithm will be quite similar to the Pohlig-Hellman algorithm for discrete logarithms.

NSDecrypt1$(\{\mu_i\}, \{p_i\}, g, N)$
1 **for** $i \leftarrow 1$ **to** k
2 **do** $\mu_i \leftarrow \mu^{\frac{\phi(N)}{p_i}} \mod N$
3 **for** $j \leftarrow 0$ **to** $p_i - 1$
4 **do if** $\mu_i = g^{\frac{j\phi(N)}{p_i}} \mod N$
5 **then** $m'_i \leftarrow j$
6 $m \leftarrow$ ChineseRemainder$\{m'_i\}, \{p_i\}$
7 **return** m

Since

$$\begin{aligned} \mu_i &= \mu^{\frac{\phi(N)}{p_i}} \\ &= g^{\frac{m\phi(n)}{p_i}} \\ &= g^{\frac{(m'_i + \mu_i p_i)\phi(N)}{p_i}} \\ &= g^{m'_i\phi(N)} p_i g^{\mu_i \phi(N)} \\ &= g^{\frac{m'_i\phi(N)}{p_i}} \mod N \end{aligned}$$

By determining the discrete logarithm of c_i modulus N and multiplying by $\frac{p_i}{\phi(N)}$, we can set m_i to the resulting value. Then by use of the Chinese Remainder Theorem we can compute the original message m from $m'_i \mod p_i$ for $1 \leq i \leq k$. It can easily be seen that the decryption algorithm is a special case of the Pohlig-Hellman algorithm for discrete logarithms. The algorithm above is from [NS98a]. To solve intermediate discrete logarithms, the above algorithm uses a naive algorithm for computing discrete logarithms. If the conditions allowed to supplement this portion with Shank's algorithm[Sti02], we would proceed as follows.

NSDecrypt2$(\{\mu_i\}, \{p_i\}, g, N)$
1 **for** $i \leftarrow 1$ **to** k
2 **do** $\mu_i \leftarrow \mu^{\frac{\phi(N)}{p_i}} \mod N$
3 $m'_i \leftarrow$ Shanks$(N, g, \mu_i) \cdot \frac{p_i}{\phi(N)}$
4 $m \leftarrow$ ChineseRemainder$\{m'_i\}, \{p_i\}$
5 **return** m

This algorithm might also allow for the Pollard-Rho algorithm[Sti02] for discrete logarithms, by replacing the call to Shank's with a call to Pollard Rho.

In summary, the Naccache-Stern cryptosystem is as follows.

Encryption Scheme 3.2.3 (Naccache-Stern[NS98a])

Plaintext Space:	$m < \sigma$
Random Space:	$r < N$
Ciphertext Space:	$\mathbb{Z}_N$
Public Key:	g, N
Private Key:	p, q
Encryption:	$\mu = r^{\sigma} g^{m} \mod N$
Decryption:	NSDecrypt$(\{\mu_i\}, \{p_i\}, g, N)$

3.2.4 Okamoto-Uchiyama Encrytpion Scheme

The Okamoto-Uchiyama encryption scheme was introduced in [OU98]. It performs computations in p-subgroups.

Definition 3.2.4 (p-Group) *Let p be a prime number. G is a p-group when $G = \{x : |x| = p^k\}$. Equivalently, the number of elements in G is a power of p.*

Theorem 3.2.1 (p-Subgroup) *Every finite group has subgroups which are p-groups.*

Let the public key tuple be (g, h, N). The Okamoto-Uchiyama encryption scheme uses a slightly different composite N compared to the previously discussed encryption schemes. Here p and q will be chosen such that $\lceil \log_2 p \rceil = \lceil \log_2 q \rceil = \lceil \frac{\log_2 N}{3} \rceil$, so that $N = p^2 \cdot q$. Choose a random $g < N$, such that

$$g_p = g^{p-1} \mod p^2$$

where g_p has order p in $\mathbb{Z}^*_{p^2}$. Similarly, choose $g' < N$, such that

$$h = {g'}^{N} \mod N$$

The private key tuple is thus (p, q, g_p).

In the other encryption schemes presented in this chapter, there has been a y or g chosen wisely for easy computation of a discrete logarithm with base y or g, this is also true of the Okamoto-Uchiyama; however, there was also a wisely chosen exponent to be applied to the randomness r introduced in the encryption function $\mathcal{E}$. That is not quite the case in the Okamoto-Uchiyama encryption scheme. The public key value h is used as a base and r is used as the exponent. Since $h = {g'}^{N} \mod N$, $h^r = {g'}^{Nr} \mod N$, which is essentially a random number to the N^{th} times r power. The resulting encryption function is as follows

$$\mathcal{E}(m) = g^m h^r \mod N$$

The decryption algorithm $\mathcal{D}$ can be completed in two steps

1. $\mu' = \mu^{p-1} \mod p^2$
2. $m = \log(\mu')\log(g_p)^{-1} \mod p$

Notice that

$$\mu' = \mu^{p-1} \mod p^2 = {g_p}^{m(p-1)} {g'}^{nr(p-1)} = g_p^m \mod p^2$$

so it can be seen that

$$\begin{aligned} \log(\mu') &= m \cdot (p-1) \\ \log(g_p) &= (p-1) \end{aligned}$$

Thus $\log(\mu')\log(g_p)^{-1} \mod p = m$.

In summary, the Okamoto-Uchiyama cryptosystem is as follows.

Encryption Scheme 3.2.4 (Okamoto-Uchiyama[OU98])

Plaintext Space:	$\mathbb{Z}_q$
Random Space:	$\mathbb{Z}^*_{p^2}$
Ciphertext Space:	$\mathbb{Z}^*_N$
Public Key:	N, g, h
Private Key:	p, q
Encryption:	$\mu = h^r g^m \mod N$
Decryption:	$m = \log_p(\mu')\log_g(g_p)^{-1} \mod p$

3.2.5 Paillier Encryption Scheme

The most used Residuosity Class Problem based probabilistic encryption scheme in academic research was introduced by Pascal Paillier in [Pai99]. His encryption scheme is based on the Composite Residuosity Class Problem. A composite residue, or N^{th} residue, is as follows

Definition 3.2.5 (Composite Residue) *(N^{th} residue) A number z is said to be an N^{th} residue modulo N^2 if there exists a number $y \in \mathbb{Z}^*_{N^2}$ such that $z \equiv y^N \mod N^2$. The set of N^{th} residues is a multiplicative subgroup of $\mathbb{Z}^*_{N^2}$ of order $\phi(N)$.*

Considering that the encryption scheme relies upon computation of composite residues, encryption can be treated similar to that of the previously mentioned encryption schemes. For now, consider the Goldwasser-Micali encryption scheme. Since the Goldwasser-Micali scheme relied upon recognizing quadratic residues, the randomness was squared, and since the Pascal encryption scheme

relies upon composite residues, the randomness should be raised to the N^{th} power. For all of the previous encryption schemes, the residue classes were computed modulo N, but composite residue classes are computed modulo N^2. Thus the encryption function will change as follows

$$\mathcal{E}(m) = r^N g^m \mod N^2$$

To be consistent with previously mentioned encryption schemes, g must be chosen such that the discrete logarithm base g modulo N can be easily computed. Consider the N^{th} roots of unity for $\mathbb{Z}^*_{N^2}$.

Definition 3.2.6 (N^{th} Roots of Unity of $\mathbb{Z}^*_{N^2}$) $(1+N)^x = 1+xN \mod N^2$

The discrete logarithm of some value base $(1+N)$ modulo N^2 is $1+xN$, so $\frac{\log_{(1+N)} \mu \mod N^2-1}{N} = m$ where $\mu = (1+N)^m$. Thus it would be advantageous to choose a $g \in \mathbb{Z}^*_N$ such that $g = (1+N)^j x \mod N^2$. By choosing g in this manner, the encryption function $\mathcal{E}$ is now a one-one correspondence.

Consider that if the plaintext $m \in \mathbb{Z}_N$, and the randomness $r \in \mathbb{Z}^*_N$, then $\mathcal{E}\colon \mathbb{Z}_N \times \mathbb{Z}^*_N \mapsto \mathbb{Z}^*_{N^2}$. Since $|\mathbb{Z}_N| = N$ and $|\mathbb{Z}^*_N| = \phi(N)$, then since $|\mathbb{Z}^*_{N^2}| = N \cdot \phi(N)$ it is easily seen that $\mathcal{E}$ is an onto function. Thus, it must be shown that $\mathcal{E}$ is one-to-one due to the choice of g.

Before it can be shown that $\mathcal{E}$ is a one-one correspondence, the private key must be explained. The private key can be simply represented as the tuple (p, q) where $N = p \cdot q$, but for decryption, which will be explained later, let $\lambda = lcm(p-1, q-1)$. Now for verification that $\mathcal{E}$ is indeed a one-one correspondence, consider

$$\text{is } g^{m_1} {r_1}^N \equiv g^{m_2} {r_2}^N \mod N^2?$$

Where $m_1, m_2 \in \mathbb{Z}_N$ and $r_1, r_2 \in \mathbb{Z}^*_N$.

$$\begin{aligned}
&\Rightarrow \quad g^{m_1 - m_2} \left(\frac{r_2}{r_1}\right)^N \equiv 1 \mod N^2 \\
&\Rightarrow \quad |g| \mid \lambda(m_2 = m_1) \\
&\Rightarrow \quad N \mid \lambda(m_2 - m_1) \\
&\Rightarrow \quad gcd(\lambda, N) = 1 \\
&\Rightarrow \quad m_2 - m_1 \mid N \\
&\Rightarrow \quad m_2 - m_1 \equiv 0 \mod N \\
&\Rightarrow \quad \frac{r_2}{r_1} = 1 \text{ uniquely} \\
&\Rightarrow \quad m_1 = m_2 \text{ and } \mu_1 = \mu_2 \\
&\Rightarrow \quad \mathcal{E}(m, r) \text{ is a one-one correspondence.}
\end{aligned}$$

While the encryption function $\mathcal{E}$ does not necessarily have to be a onto, it does have to be one-to-one so that the function is invertible. Thus it is possible to decrypt.

The decryption function $\mathcal{D}$ is as follows

$$\mathcal{D}(\mu) = \frac{L(\mu^\lambda \mod N^2)}{L(g^\lambda \mod N^2)} \mod N$$

where

$$L(u) = \frac{u-1}{N}$$

for $u \in \mathbb{Z}^*_{N^2}$ and

$$\lambda = lcm(p-1, q-1)$$

is chosen such that the decryption algorithm can remove the randomness introduced during encryption. For RSA N we can assume $\lambda = \frac{(p-1)(q-1)}{2}$. Notice

$$\begin{aligned} \mu^\lambda &= (1+N)^{m\lambda} r^{N\lambda} \\ &= (1+N)^{m\lambda} \\ &= (1+N)^{m\lambda} \\ &= 1 + m\lambda N \mod N^2 \end{aligned}$$

and thus

$$L(\mu^\lambda \mod N^2) = m\lambda$$

and since

$$L(g^\lambda \pmod N)^2) = L(1+\lambda N) = \lambda$$

it can easily be seen that $\mathcal{D}(\mathcal{E}(m,r)) = m$.

With knowledge of p and q, a λ could be chosen such that the randomness introduced in encryption was removed and the choice of g allowed for quick computation of discrete logarithms. Thus, the security of Paillier's encryption scheme relies on the Composite Residuosity Assumption.

Assumption 3.2.4 (Composite Residuosity Assumption) *There does not exist a polynomial time distinquisher for N^{th} residues modulo N^2.*

In summary, the Paillier cryptosystem is as follows.

Encryption Scheme 3.2.5 (Paillier[Pai99])

Plaintext Space:	$\mathbb{Z}_N$
Random Space:	$\mathbb{Z}^*_N$
Ciphertext Space:	$\mathbb{Z}^*_{N^2}$
Public Key:	g, N
Private Key:	p, q
Encryption:	$\mu = g^m r^N \mod N^2$
Decryption:	$m = \frac{L(\mu^\lambda \mod N^2)}{L(g^\lambda \mod N^2)} \mod N$

For the remainder of this thesis, the case where $g = N + 1$ will be the only case considered, since this case is much simpler to handle, but gives the same security.

3.2.6 Damgård-Jurik Encryption Scheme

Damgård and Jurik extended Paillier's encryption scheme in [DJ03]. Notice that $(1+N)^x \mod N^2 = 1 + xN \mod N^2$. This is due to the binomial expansion

$$(1+N)^x = 1 + \binom{x}{1}N + \binom{x}{2}N^2 + \cdots + \binom{x}{x}N^x$$

Damgård and Jurik, in [DJ03], consider modulus N^{s+1}, rather than N^2. Thus

$$(1+N)^x \mod N^{s+1} = (1 + \binom{x}{1}N + \binom{x}{2}N^2 + \cdots + \binom{x}{s}N^s) \mod N^{s+1}$$

To find the discrete logarithm base $(1+N)$ of a number a with modulus N^s, the following algorithm can be applied

DiscreteLog(a, N, s)
1 $i \leftarrow 0$
2 **for** $j \leftarrow 1$ **to** s
3 **do** $t_1 \leftarrow \mathrm{L}(a \mod N^{j+1})$
4 $t_2 \leftarrow i$
5 **for** $k \leftarrow 2$ **to** j
6 **do** $i \leftarrow i - 1$
7 $t_2 \leftarrow t_2 \cdot i \mod N^j$
8 $t_1 \leftarrow t_1 - \frac{t_2 \cdot N^{k-1}}{k!} \mod N^j$
9 $i \leftarrow t_1$
10 **return** i

Essentially the public key will be the same tuple as in Paillier (g, N), but since s can vary, the resulting public key tuple will be (g, N, s) where $s > 1$. Since it is easy to find the discrete log base $(1+N)$, $g = (1+N)^j x \mod N^2$ as in Paillier's encryption scheme, and thus the case where $s = 1$, the encryption schemes are essentially the same. To account for the expansion exponent s

$$\mathcal{E}(m) = g^m r^{N^s} \mod N^{s+1}$$

The decryption $\mathcal{D}$ is similar to Paillier's decryption function as well. Let $d \equiv 0 \mod \lambda$ where $\lambda = lcm(p-1, q-1)$ as before. It is easy to compute m from μ^{jd}. Let $g = (1+N)$ which implies $j = 1$, then

$$\begin{aligned} \mu^d &= ((1+N)^m r^{N^s})^d \\ &= (1+N)^{md} \mod N^s (g^m r^{N^s})^d \mod \lambda \\ &= (1+N)^m \mod N^{s+1} \end{aligned}$$

$$\textit{Note: } g = (1+N) = (1+N)^j \mu \mod N^{s+1}$$

In summary, the Damgård-Jurik cryptosystem is as follows.

Encryption Scheme 3.2.6 (Damgård-Jurik[DJ03])

Plaintext Space:	$\mathbb{Z}_{N^s}$
Random Space:	$\mathbb{Z}_N^*$
Ciphertext Space:	$\mathbb{Z}_{N^{s+1}}^*$
Public Key:	g, N, s
Private Key:	p, q
Encryption:	$\mu = g^m r^{N^s} \mod N^{s+1}$
Decryption:	*compute* $m \mod N^s$ *from* μ^d

Chapter 4

Previous PIR Protocols

The protocols covered in this chapter make use of the additive homomorphic probabilistic encryption schemes discussed in Chapter 3. The additive property of the encryption scheme chosen for the protocol allows the client to encrypt $0, 1$ indicators. Multiplicative encryption schemes, like ElGamal, do not allow for encryption of 0s. The probabilistic property of the chosen encryption scheme provides semantic security of the encrypted $0, 1$ indicators. Without semantic security the server would now have a $\frac{1}{2}$ chance of guessing which of the encrypted values were the encryption of 0 and which were the encryption of 1, but since, as will be shown in this chapter, there will be a known number of 0s and 1s and the number of 0s will never be the same as the number of 1s, without the semantic security provided by a probabilisitc encryption scheme, the server will know which of the encrypted values are 0 and which are 1 with absolute certainty. With the semantic security provided by the chosen encryption scheme, the server will have $\frac{1}{2^K}$ chance of guessing which index the server-side private database the set of $0, 1$ indicators is referring to. The homomorphic property of the chosen encryption scheme, $\mathcal{D}(\mathcal{E}(m_0) \cdot \mathcal{E}(m_1) = m_0 + m_1$ where $m_0, m_1 \in \mathcal{PT}$ and $\mathcal{E}$ and $\mathcal{D}$ are respectively the encryption and decryption functions of the chosen encryption scheme, as shown in this chapter, will allow for the encrypted $0, 1$ indicators to filter the entries in the database to an encryption of the desired index which can be sent back to the user as a server-side response R_s, that the client will be capable of extracting the result from with knowledge of its private key S_k.

4.1 Generic Protocol

The following protocols were presented as one-round protocols secure in the semi-honest model. Since the client chooses the public key, private key pair (P_k, S_k) secure under a probabilistic additive homomorphic cryptosystem, as presented in Chapter 3, when the client and server both follow the protocol from start to finish, upon completion of the protocol, the user will have a valid

CLIENT		SERVER
Input: $i*$		**Input:** $\{x_i\}_{i=0}^{K-1}$, c, s
1. Acquire Server Parameters *Request* Database Size # of Dimensions[1] Dimension Length[1] Data Size # of Splits[2] Maximum Exponent[3]	$\xrightarrow{req}$	*Send* Database Size # of Dimensions[1] Dimension Length[1] Data Size # of Splits[2] Maximum Exponent[3]
	$\xleftarrow{K, c, \lceil \sqrt[c]{K} \rceil, \lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil, t, s}$	
2. Generate Query $(\mathcal{P}_k, \mathcal{S}_k)$, Q	$\xrightarrow{Q, N}$	
		3. Process Query Against Database $f_N : Q \times \{x_i\}_{i=0}^{K-1} \rightarrow R_s$
	$\xleftarrow{R_s}$	
4. Extract Result ${f_N}^{-1} : R_s \rightarrow R_c$		
Output: $R_c = x_{i*}$		**Output:** NULL

[1]If the protocol references $\{x_i\}_{i=0}^{K-1}$ as $\{x_{i_0,i_1,\ldots,i_{c-1}}\}_{\forall i_0,i_1,i_{c-1} \in \mathbb{Z}_\ell}$, $i = \lceil \sqrt[c]{K} \rceil$

[2]If the protocol requires intermediate results to be split

[3]If the protocol uses the length-flexible cryptosystem

Figure 4.1: Generic Protocol

output, x_{i*} the element residing at the desired index $i*$ of the server's private linear database, and the server will be guaranteed that the client received only one item from the private linear database and no information about any other entry in the database. The server will also not be able to determine which entry of the private linear database that the client retrieved, unless the server can decrypt the query sent by the client. The client also will not be able to discover any other entries in the database since the well-formed query will return a result that can only be decrypted to one plaintext message, the entry residing at index $i*$ in the server's private linear database. As can be seen later from the structure of the client-side queries, if the client chooses multiple indices, the client will not receive multiple entries, but at best the sum of those entries at those indices.

These protocols were initially presented as one-round protocols, but these protocols assume that the client and server have already agreed upon some public parameters. Assuming that the server and the client are not well acquainted, the client will have to request the public parameters from the server, and the server should return the appropriate parameters. Appropriate parameters allow for the client to produce a query that will properly retrieve any desired index $i*$ from the server's private linear database. If the query is not properly structured for the server's database, then the client will not have access to all of the entries in the database in the event that the server informs the client of a smaller database size. The client may retrieve the wrong index if the server provides the wrong number of dimensions. The client may retrieve *garbage* if the server informs the client that the number of bytes in an entry is smaller than the actual maximum number of bytes in any entry. The number of parts, in the protocols that split intermediate results, however, can be easily inferred from the server-side result R_s, if all other parameters were conveyed correctly, but the maximum exponent s must be agreed upon when using the length-flexible cryptosystem presented by Damgård-Jurik in [DJ03], or the error result will be the same as when the data size was incorrectly conveyed.

Expanding these protocols to two-rounds, with the requesting and retrieving of parameters as round one, the protocols flow as described in Figure 4.1.

4.2 Stern Protocol

The first protocol to use additive homomorphic probabilistic encryption schemes as part of a Private Information Retrieval protocol was the Stern protocol [Ste98]. The Stern protcol is the simplest protocol presented, as it references the server-side private linear database as an array, that is using a single index, which in later protcols is not the case. Understanding of the Stern protocol is necessary as it is the foundation of later protocols.

4.2.1 Encrypting 0's and 1's

The probabilistic property of the encryption schemes mentioned in Chapter 3 provides semantic security. Thus, if a user encrypted $\{0,1\}^*$ there would be a

$\frac{1}{2}$ chance of guess whether a single element in the string was a 0 or a 1 from the ciphertext result of encrypting the string bit-by-bit, and there would be a $\frac{1}{2^K}$ chance of guessing the entire string of length K correctly, as opposed to a $\frac{1}{2}$ chance of guessing all of a $\{0,1\}^*$, if the ciphertext was a result of a non-semantically secure cryptosystem.

The cryptosystems also fulfilled the following property $\forall x \in \mathcal{PT}$

$$\mathcal{D}(\mathcal{E}(0)^x) = 0$$
$$\mathcal{D}(\mathcal{E}(1)^x) = x$$

which when the $\{0,1\}^K = 0^v 10^w$ where $v + w = K - 1$ and $i* = v$ the index of the character 1, the client's desired index, this property can be combined with the homomorphic property of the mentioned cryptosystems

$$\mathcal{D}(\mathcal{E}(m_0) \cdot \mathcal{E}(m_1)) = m_0 + m_1 \mod \nu$$

where the domain of the ciphertext, $\mathcal{CT} = \mathbb{Z}_\nu$, to single out one element in a collection $\{x_i\}_{i=0}^{K-1}$, where $x_i \in \mathcal{PT}$ $\forall i$, as follows

$$\mathcal{D}\left(\prod_{i=0}^{K-1} \mathcal{E}(m_i, r_i)^{x_i} \mod \nu\right) = x_i*$$

Thus, if the collection $\{x_i\}_{i=0}^{K-1}$ is the server-side private database, then the client could generate a query that is a collection $\{q_i\}_{i=0}^{K-1}$ that is the resulting bit-by-bit encryption of $0^v 10^w$ such that

$$q_i = \begin{cases} \mathcal{E}(1) & i = i*(= v) \\ \mathcal{E}(0) & \text{otherwise} \end{cases}$$

and privately retrieve the element x_{i*} by computing

$$R_s = \prod_{i=0}^{K-1} {q_i}^{x_i} \mod \nu.$$

4.2.2 Implementing

See Appendix A.1 for pseudo-code implementation details.

1. Acquire Server Parameters

GetDatabaseSize()

> **Description** Retrieves the number of elements in the database, so that the client can generate a query that corresponds to the length of the database.
>
> **Returns** the number of elements in the database. K

GetDataSize()

Description Retrieves the bit-length of the element with the greatest value in the database.

Returns the suggested minimum bit-length of the client-side public-key parameter N, disregarding the minimum supposed to be secure for RSA. $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$

2. Generate Query

Stern-Query($i*$)

Description Generates a query with one element per each element in the database. Where each element is the encryption of the plaintext 0, unless the element is at index $i*$. The element at index $i*$ is the encryption of the plaintext 1.

Parameters

$i*$ The index the client wishes to privately retrieve from the server.

Returns the private query to be sent to the server for retrieving index $i*$. $Q = \{q_i\}_{i=0}^{K-1}$ where $q_i = \mathcal{E}(0)$ for $i \neq i*$ and $q_i = \mathcal{E}(1)$ for $i = i*$

3. Process Query Against Database

Stern-Perform-Retrieval(Q, ν)

Description Takes the client's private query and processes the query against the database.

Parameters

Q The client's private query.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

Returns the server-side result after processing the query against the private database. R_s

4. Extract Result

Stern-Extract-Result(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database.

Returns The plaintext result which should be the $i*$ entry of the private database.

4.3 Chang Protocol

The Chang protcol was introduced in [Cha04]. The Chang protocol builds upon the Stern protocol by reducing the client to server communication by referencing the server-side linear private database as a hypercube, or by multiple indices of reference. With the use of the Paillier encryption scheme and some manipulation, this protocol essentially finds server-side results similar to the Stern protocol along each dimension of reference. This section will begin with explaining how a hypercube can be referenced. Second this section will discuss how the structure of the query Q has changed, and lastly will describe how Paillier ciphertexts can be manipulated so that intermediate cihpertexts can be treated as plaintexts in an attempt to merge intermediate server-side results into a complete server-side result that the client will be able to extract the desired result from.

4.3.1 Referencing a Linear Database as a Hypercube

To reference a database as though it were a c-dimensional hypercube, simply means to reference each item in the database by c indices. The linear database containing K elements, $\{x_i\}_{i=0}^{K-1}$ as discussed in the Stern Protocol in Section 4.2 could be termed a 1-dimensional hypercube since each element in the database is indexed by a single index i.

This same database could be referenced by 2 indices with the possibility of padding with $\emptyset$ entries when $K < \lceil \sqrt[2]{K} \rceil^2 - 1$ as follows

$$\{x_i\}_{i=0}^{K-1} \cup \{\emptyset_i\}_{i=K}^{\lceil \sqrt[2]{K} \rceil^2 - 1} = \left\{ \{y_{i_0 i_1}\}_{i_0=0}^{\lceil \sqrt[2]{K} \rceil - 1} \right\}_{i_1=0}^{\lceil \sqrt[2]{K} \rceil - 1}$$

where

$$x_i = y_{i_0 i_1} \text{ if } i = i_0 \lceil \sqrt[2]{K} \rceil + i_1.$$

This is a translation of a linear database to a 2-dimensional hypercube database.

In general, any linear database can be translated to a c-dimensional hypercube by the following relation

$$\{x_i\}_{i=0}^{K-1} \cup \{\emptyset_i\}_{i=K}^{\lceil \sqrt[c]{K} \rceil^{c-1}} = \left\{ \ldots \left\{ \{y_{i_0 i_1 \ldots i_{c-1}}\}_{i_0=0}^{\lceil \sqrt[c]{K} \rceil - 1} \right\}_{i_1=0}^{\lceil \sqrt[c]{K} \rceil - 1} \ldots \right\}_{i_{c-1}=0}^{\lceil \sqrt[c]{K} \rceil - 1}$$

where

$$x_i = y_{i_0 i_1 \ldots i_{c-1}} \text{ if } i = \sum_{j=0}^{c-1} i_j \lceil \sqrt[c]{K} \rceil^{c-j-1}$$

4.3.2 The New Query Q Structure

In Section 4.2, the query was

$$Q = \{q_i\}_{i=0}^{K-1}$$

but now that the database is referenced as a hypercube, that is by c indices of reference, the structure of the query must change. Since to reference a single element $x_{i_0 i_1 \ldots i_{c-1}}$ from the server-side database requires $0 \leq i_j < \lceil \sqrt[c]{K} \rceil$ for all $0 \leq j < c$, then it can been seen that there will need to be $c\lceil \sqrt[c]{K} \rceil$ $0, 1$ indicators. To properly index one item in the database, there will be exactly c 1s in the entire collection of indicators, that is

$$q_{i,j} = \begin{cases} \mathcal{E}(1) & j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil \\ \mathcal{E}(0) & \text{otherwise} \end{cases}$$

and thus

$$Q = \left\{ \{q_{ij}\}_{i=0}^{\sqrt[c]{K}-1} \right\}_{j=0}^{c-1}$$

For example, if $c = 3$, $\sqrt[c]{K} = 5$, and $i* = 101$, then

$$Q = \left\{ \begin{array}{l} \{\mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(1)\} \\ \{\mathcal{E}(1), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0)\} \\ \{\mathcal{E}(0), \mathcal{E}(1), \mathcal{E}(0), \mathcal{E}(0), \mathcal{E}(0)\} \end{array} \right\}$$

4.3.3 Encrypting Paillier Ciphertext

As seen in Section 3.2.5, the plaintext domain of the Paillier cryptosystem is $\mathcal{PT} = \mathbb{Z}_N$ where $\mathbb{Z}_N$ is the additive cyclic group on N elements, the ciphertext domain of the Paillier cryptosystem is $\mathcal{CT} = \mathbb{Z}_{N^2}^*$ where $\mathbb{Z}_{N^2}^*$ is the multiplicative group on $N\phi(N) = \phi(p^2q^2)$ elements, and the message $m \in \mathcal{PT}$ and the ciphertext $\mu \in \mathcal{CT}$ are related by

$$\begin{array}{rcll} \mathcal{E}(m, r) & = & \mu & \\ & = & (N+1)^m r^N & \mod N^2 \\ \mathbb{Z}_N \times \mathbb{Z}_N^* & \rightarrow & \mathbb{Z}_{N^2}^* & \end{array}$$

where $r \in \mathbb{Z}_N^*$ is chosen at random.

Since clearly $\mathbb{Z}_{N^2}^* \neq \mathbb{Z}_N$ a ciphertext μ cannot be encrypted, but since $\mathbb{Z}_{N^2}^* \cup \{0\} \equiv \mathbb{Z}_{N^2}$ an additive cyclic group, a ciphertext μ could be split as follows

$$\begin{array}{rcl} \mu & = & m_0 N + m_1 \\ \mathbb{Z}_{N^2}^* & \rightarrow & \mathbb{Z}_N \times \mathbb{Z}_N \end{array}$$

and now that $m_0, m_1 \in \mathbb{Z}_N = \mathcal{PT}$ the ciphtertext c can be encrypted as the pair $(\mathcal{E}(m_0, r_0), \mathcal{E}(m_1, r_1)) = (\mu_0, \mu_1)$.

Decryption of the pair (μ_0, μ_1) proceeds as follows to arrive at the original message m

$$\mathcal{D}(\mathcal{D}(\mu_0) N + \mathcal{D}(\mu_1)) = m$$

where

$$\mathcal{D}\colon \mathbb{Z}^*_{N^2} \to \mathbb{Z}_N$$

Using this technique of splitting, the ciphertext result of a Paillier encryption can easily be encrypted c times.

4.3.4 Implementing

See Appendix A.2 for pseudo-code implementation details.

1. Acquire Server Parameters

GetDatabaseSize()

> **Description** Retrieves the number of elements in the database, so that the client can generate a query that matches the length of the database.
>
> **Returns** the number of elements in the database. K

GetNumberOfDimensions()

> **Description** Retrieves the number of dimensions (indices) used for indexing any given element in the private database.
>
> **Returns** the number of dimensions that index any given element in the private database. c

GetDimensionLength()

> **Description** Retrieves the length of the dimension.
>
> **Returns** the number of elements in any given dimension.

GetDataSize()

> **Description** Retrieves the bit-length of the element with the greatest value in the database.
>
> **Returns** the suggested minimum bit-length of the client-side public-key parameter N, disregarding the minimum suggested bit-length for $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$ (minimum 1024)

GetNumberOfSplits()

Description Retrieves the number of pieces that the each intermediate result will be split into.

RETURNS the number of pieces each intermediate step is split into. $t = 2$ (t-fold split)

2. Generate Query

CHANG-QUERY($i*$)

Description Generates a query with one element encrypted indicators as described in Section 4.3.2. Where each element is the $\mathcal{E}(0)$, unless the element is at index $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$, then the element if $\mathcal{E}(1)$.

Parameters

$i*$ The index the client wishes to privately retrieve from the server.

Returns the private query to be sent to the server for retrieving index $i*$. $Q = \{\{q_{ij}\}_{i=0}^{\sqrt[c]{K}-1}\}_{j=0}^{c-1}$ where $q_{ij} = \mathcal{E}(0)$ for $j \neq \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$ and $q_{ij} = \mathcal{E}(1)$ for $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$

3. Process Query Against Database

PERFORM-RETRIEVAL(Q, N)

Description Takes the client's private query and processes the query against the database.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain used in the context of the Paillier encryption scheme.

Returns the server-side result after processing the query against the private database. R_s

C-HYPERCUBE(Q, N, ν, d, *index*)

Description Internal method used for recursing through all the dimensions (indices) of reference except for the two innermost indices.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

d The distance of the current dimension from the outermost dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the server-side result after processing the query against the private database. R_s

TWO-HYPERCUBE(Q, ν, i, *index*)

Description Internal method used for processing the query against the two innermost dimensions.

Parameters

Q The client's private query.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

i The current element accessed in the second dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the result of processing the query against the two innermost dimensions of the database.

SIGMA(Q, ν, i, *index*)

Description Internal method that basically finds a Stern result along on particular dimension of the private database instead of the entire database as in the Stern protocol.

Parameters

Q The client's private query.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

i The current element accessed in the second dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the result of processing the query against the innermost dimension of the database.

SPLIT(σ, N)

Description Internal method which splits a single intermediate result, a result from a SIGMA call so that each piece of the split may be a valid plaintext.

Parameters

σ A result from a SIGMA call.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

RETURNS The value of σ split into $t = 2$ pieces. $parts$ =(u, v) such that $\sigma = un + v$

FILTER(R_s, Q, $parts$, ν)

Description Internal method which applies the split from the result of processing the query on the innermost dimension to the second innermost dimension.

Parameters

R_s The server-side result from processing the private query against the database.

Q The client's private query.

$parts$ The pieces resulting from the call to SPLIT.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

Returns The server-side result from processing the query against the two innermost dimensions of the database.

C-SPLIT($prevParts$, N)

Description Internal method similar to SPLIT, but applies to outer dimensions. Where C is a particular dimension, $c - d$.

Parameters

$prevParts$ The result from a previous call to TWO-HYPERCUBE or C-HYPERCUBE.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

Returns an array of r times the length of $prevParts$. Splits each of the elements in $prevParts$ into r pieces.

C-FILTER(R_s, Q, $parts$, ν, k, d)

Description Internal method similar to FILTER, but applies to outer dimensions.

Parameters

R_s The server-side result from processing the private query against the database.

Q The client's private query.

parts The pieces resulting from the call to C-SPLIT.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

i The current element accessed in this dimension.

d The distance of the current dimension from the outermost dimension.

Returns The server-side result from processing the query against the $(c - d)$ innermost dimension of the database.

4. Extract Result

CHANG-EXTRACT-RESULT(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database. The number of elements in the array R_s is t^c. Here $t = 2$.

Returns The plaintext result which should be the $i*$ entry of the private database.

4.4 Lipmaa Protocol

The Lipmaa protocol, introduced in [Lip04], continues with referencing the server-side private database as a hypercube, but uses the Damgård-Jurik encryption scheme, a generalization of the Paillier encryption scheme, see Chapter 3, for encryption of intermediate results, and since Dangård-Jurik is length-flexible, as will be seen in this section, the need to split intermediate results is eliminated.

4.4.1 Encrypting Damgård-Jurik Ciphertext

As seen in Section 3.2.6, the plaintext domain of the Damgård-Jurik cryptosystem is $\mathcal{PT} = \mathbb{Z}_{N^s}$ is an additive cyclic group where $s > 0$ is small compared to N, the ciphertext domain of the Damgård-Jurik cryptosystem is $\mathcal{CT} = \mathbb{Z}^*_{N^{s+1}}$ is a multiplicative group, and the message $m \in \mathcal{PT}$ and the ciphertext $\mu \in \mathcal{CT}$ are related by

$$\begin{aligned} \mathcal{E}(m, r) &= \mu \\ &= g^m r^{N^s} \mod N^{s+1} \\ \mathbb{Z}_{N^s} \times \mathbb{Z}^*_N &\to \mathbb{Z}^*_{N^{s+1}} \end{aligned}$$

where $r \in \mathbb{Z}^*_N$ is chosen at random.

The Damgård-Jurik cryptosystem, a generalization of the Paillier cryptosystem, boasts to be a length-flexible cryptosystem. As a length-flexible cryptosystem, the bit-length of the plaintext increases with s, and s can vary so long as every message encrypted with one value s is decrypted with the same value of s.

Let $s > 0$, $m \in \mathcal{PT}$, and $\mathcal{E}(m, r, s) = \mu \in \mathcal{CT}$ where $r \in \mathbb{Z}^*_N$. Since $\mu \in \mathbb{Z}^*_{N^{s+1}} = \mathbb{Z}_{N^{s+1}}$, μ can be encrypted as $\mathcal{E}(\mu, r, s+1) = \mu'$. Thus, μ' can be considered the *double* encryption of the plaintext m, and $\mathcal{D}(\mathcal{D}(\mu', s+1), s) = m$.

In general, a message $m \in \mathcal{PT}$ can be encrypted n times by

$$\mu_n = \mathcal{E}(...\mathcal{E}(\mathcal{E}(m, r_0, s), r_1, s+1)...r_{n-1}, s+n-1) \in \mathbb{Z}_{N^{s+n}}$$

and the plaintext m can be recovered from μ_n by

$$m = \mathcal{D}(...\mathcal{D}(\mathcal{D}(\mu_n, s+n-1), s+n-2)...s).$$

4.4.2 Implementing

See Appendix A.3 for pseudo-code implementation details.

1. Acquire Server Parameters

GetDatabaseSize()

> **Description** Retrieves the number of elements in the database, so that the client can generate a query that matches the length of the database.
>
> **Returns** the number of elements in the database. K

GetNumberOfDimensions()

> **Description** Retrieves the number of dimensions (indices) used for indexing any given element in the private database.

Returns the number of dimensions that index any given element in the private database. c

GetDimensionLength()

Description Retrieves the length of the dimension.

Returns the number of elements in any given dimension.

GetDataSize()

Description Retrieves the bit-length of the element with the greatest value in the database.

Returns the suggested minimum bit-length of the client-side public-key parameter N, disregarding the minimum suggested bit-length for RSA. $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$ (Minimum 1024)

GetMaximumExponent()

Description Retrieves the minimum exponent s will be used in the protocol. The maximum exponent would be $s + c$.

Returns the minimum exponent. s

2. Generate Query

Lipmaa-Query($i*$)

Description Generates a query with one element encrypted indicators as described in Section 4.3.2. Where each element is the $\mathcal{E}(0, s + c - j)$, unless the element is at index $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$, then the element if $\mathcal{E}(1, s + c - j)$.

Parameters

$i*$ The index the client wishes to privately retrieve from the server.

Returns the private query to be sent to the server for retrieving index $i*$. $Q = \{\{q_{ij}\}_{i=0}^{\sqrt[c]{K}-1}\}_{j=0}^{c-1}$ where $q_{ij} = \mathcal{E}(0, s + c - j)$ for $j \neq \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$ for $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$

3. Process Query Against Database

Perform-Retrieval(Q, N)

Description Takes the client's private query and processes the query against the database.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain used in the context of the Paillier encryption scheme.

Returns the server-side result after processing the query against the private database. R_s

C-Hypercube(Q, N, d, *index*)

Description Internal method used for recursing through all the dimensions (indices) of reference except for the innermost indices.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ the plaintext domain.

d The distance of the current dimension from the outermost dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the server-side result after processing the query against the private database. R_s

One-Hypercube(Q, *index*, ν)

DescriptionInternal method that processes the query against a particular dimension of the database similar to the Stern protocol.

Parameters

Q The client's private query.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the result of processing the query against the innermost dimension of the database.

4. Extract Result

Lipmaa-Extract-Result(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database.

Returns The plaintext result which should be the $i*$ entry of the private database.

Chapter 5

New PIR Protocols

This chapter is somewhat misnamed in that the first three protocols lay out a train of thought for the final protocol. However, each of the four protocols is a variation on the existing protocols just as the Chang and Lipmaa protocols were variations on the Stern protocol. Since the Chang protocol relies on the Paillier encryption scheme in which the length of the plaintext is dependent upon the length of N, involving larger plaintexts is possible but not necessarily as plausible since a larger key would have to be generated. The Lipmaa protocol and the final protocol of this chapter however use the length-flexible encryption scheme developed by Damgård-Jurik. The goal is to have a protocol that could potentially handle plaintexts of reasonable length. The bit-length of the operands involved in the modular exponentiation, is directly related to the CPU time spent on a modular exponentiation. The Lipmaa protocol increases the bit-length of the operands as it progresses through the dimensions. The final protocol of this chapter however reduces the bit-length of the operands as it progresses through the dimensions at the cost of increasing the number of modular exponentiations by a factor of 2 per dimension. Through experimentation, with the final protocol versus the Lipmaa protocol, it could become apparent not only whether communication or computational complexity serves as the greater bottleneck in producing a practical protocol, but whether or not it is faster to perform several smaller modular exponentiations or fewer longer modular exponentiations.

5.1 Extended Chang Protocol

The Chang protocol presented in section 4.3 can be extended to support database records $x_i \in \mathbb{Z}_{N^s}$ where $s > 1$ by substituting the Paillier encryption scheme with the Damgård-Jurik encryption scheme. The intermediate ciphertexts however will not be encrypted as in the Lipmaa protocol, but rather the ciphertext will be encrypted in a manner similar to Chang. Considering that the Paillier encryption scheme is the same as the Damgård-Jurik encryption scheme when

$s = 1$ and the number of parts was $s + 1 = 2$, the number of parts in this extension of the Chang protocol will be $s + 1$ as well.

5.1.1 Another Method of Encrypting Damgård-Jurik Ciphertext

In Section 4.4.1, it was shown that a plaintext $m \in \mathbb{Z}_{N^s}$ could be encrypted as $\mu = \mathcal{E}(m, s)$ where $\mathcal{E}$ is the encryption function of the Damgård-Jurik encryption scheme, and that the ciphertext $\mu \in \mathbb{Z}^*_{N^{s+1}}$ can be encrypted once more as $\mu' = \mathcal{E}(\mu, s{+}1)$. The plaintext m can be easily retrieved through double decryption by $m = \mathcal{D}(\mathcal{D}(\mu', s{+}1), s)$. However, in Section 4.3.3 it was seen that a plaintext $m \in \mathbb{Z}_N$ could be doubly encrypted by splitting the intermediate ciphertext $\mu \in \mathbb{Z}^*_{N^2}$ into two parts such that $\mu = m_0 N + m_1$, and the pair $(\mu_0, \mu_1) = (\mathcal{E}(m_0), \mathcal{E}(m_1))$ where $\mathcal{E}$ is the ecryption function of the Paillier encryption scheme. From this doubly encrypted pair (μ_0, μ_1) the original message m could be recovered by $m = \mathcal{D}(\mathcal{D}(\mu_0) + \mathcal{D}(\mu_1))$. This method used in Section 4.3.3 to doubly encrypt a plaintext can be applied to the Damgård-Jurik encryption scheme. Let the plaintext $m \in \mathbb{Z}_{N^s}$ and the resulting ciphertext $\mu = \mathcal{E}(m, s) \in \mathbb{Z}_{N^{s+1}}$, then this ciphertext can be split as follows

$$\mu = \sum_{i=0}^{s} m_i N^{s-i}$$

and each m_i for $0 \leq i \leq s$ can be encrypted as $\mu_i = \mathcal{E}(m_i)$, so from the $(s+1)$-tuple $(\mu_0, \mu_1, ...\mu_s)$ the original plaintext can be retrieved by

$$m = \mathcal{D}\left(\sum_{i=0}^{s} \mathcal{D}(m_i, s) N^{s-i}\right)$$

5.1.2 Implementing

See Appendix B.1 for pseudo-code implementation details.

1. Acquire Server Parameters

See Section 4.3.4 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, and GetDataSize

GetNumberOfParts()

Description Retrieves the number of pieces that the each intermediate result will be split into.

Returns the number of pieces each intermediate step is split into. $t = s + 1$

GetMaximumExponent()

Description Retrieves the maximum exponent s will be used in the protocol.

Returns the maximum exponent. s

2. Generate Query

See Section 4.3.4 for CHANG-QUERY.

3. Process Query Against Database

PERFORM-RETRIEVAL(Q, N)

Description Takes the client's private query and processes the query against the database.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain used in the context of the Paillier encryption scheme.

Returns the server-side result after processing the query against the private database. R_s

See Section 4.3.4 for C-HYPERCUBE, TWO-HYPERCUBE, SIGMA, FILTER, and C-FILTER and override the C-SPLIT and SPLIT with the following.

C-SPLIT($prevParts$, N)

Description Internal method similar to SPLIT, but applies to outer dimensions. Where C is a particular dimension, $c - d$.

Parameters

$prevParts$ The result from a previous call to TWO-HYPERCUBE or C-HYPERCUBE.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

Returns an array of $t = s + 1$ times the length of $prevParts$. parts each of the elements in $prevParts$ into t pieces.

SPLIT(σ, N)

Description Internal method which parts a single intermediate result, a result from a SIGMA call so that each piece of the split may be a valid plaintext.

Parameters

σ A result from a SIGMA call.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

RETURNS The value of σ split into $t = s + 1$ pieces. $parts$ =(u, v) such that $\sigma = \sum_{i=0}^{s} u_i N^{s-i}$

4. Extract Result

CHANG-EXTRACT-RESULT(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database. The number of elements in the array R_s is t^c. Here $t = s + 1$.

Returns The plaintext result which should be the $i*$ entry of the private database.

5.2 Generalized Chang Protocol

The Extended Chang protocol increased the possible bit-length of the plaintexts in the private server-side database without increasing the bit-length of the N, but at the expense of extra parts which translates to a factor of $s-1$ more bits. This generalization of the Chang protocol considers how to reduce the number of parts while maintaining the increase in the potential length of plaintexts made possible by the Extended Chang protocol.

5.2.1 Reducing the Number of Parts

As mentioned, Section 4.4.1 provided a reasonable method of encrypting Damgård-Jurik ciphertext once more, but as in Section 5.1.1 there exists other ways to encrypt Damgård-Jurik ciphertext. However, there is no need to split intermediate ciphertexts into $s + 1$ parts. When a ciphertext $\mu = \mathcal{E}(m, s) \in \mathbb{Z}^*_{N^{s+1}}$, where $m \in \mathbb{Z}_{N^s}$ is the plaintext, is split into $s + 1$ parts $(m_0, m_1, ...m_s)$, each part $m_i \in \mathbb{Z}_N$ for all $0 \leq i \leq s$, but $\mathbb{Z}_N$ is a small subset of $\mathbb{Z}_{N^s}$ when $s > 1$. Through simple algebra it can be proven that for all ℓ such that $0 \leq \ell \leq s - 1$ there exists $\{x_i\}_{i=\ell}^{s}$ and $\{y_j\}_{j=0}^{s}$ such that

$$\sum_{i=\ell}^{s} x_i \cdot N^{s-i} = \sum_{j=0}^{s} y_j \cdot N^{s-j}$$

where $x_\ell \in \mathbb{Z}_{N^{s-\ell}}$ and x_i,$y_j \in \mathbb{Z}_N$ for $\ell < i \leq s$ and $0 \leq \ell \leq s$. With this relation, it can easily be seen that the number of parts can be reduced to $t = s - \ell + 1$.

5.2.2 Implementing

See Appendix B.2 for pseudo-code implementation details.

1. Acquire Server Parameters

See Section 4.3.4 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, and GetDataSize

GetNumberOfParts()

Description Retrieves the number of pieces that the each intermediate result will be split into.

Returns the number of pieces each intermediate step is split into. $t \leq s + 1$

See Section 5.1.2 for GetMaximumExponent.

2. Generate Query

See Section 4.3.4 for Chang-Query.

3. Process Query Against Database

See Section 5.1.2 for Chang-Perform-Retrieval and override C-Split and Split with the following.

C-Split($prevParts$, N)

Description Internal method similar to Split, but applies to outer dimensions. Where C is a particular dimension, $c - d$.

Parameters

prevParts The result from a previous call to Two-Hypercube or C-Hypercube.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

Returns an array of $t \leq s$ times the length of *prevParts*. Splits each of the elements in *prevParts* into t pieces.

Split(σ, N)

Description Internal method which splits a single intermediate result, a result from a SIGMA call so that each piece of the split may be a valid plaintext.

Parameters

σ A result from a SIGMA call.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

RETURNS The value of σ split into $t \leq s$ pieces. $parts = (u, v)$ such that $\sigma = \sum_{i=\ell}^{s} u_i N^{s-i}$

4. Extract Result

CHANG-EXTRACT-RESULT(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database. The number of elements in the array R_s is t^c. Here $t \leq s$.

Returns The plaintext result which should be the $i*$ entry of the private database.

5.3 Chang Protocol with Pre-Split Elements

The Extended and Generalized Chang protocol expanded the possible bit-length of the plaintext records in the server's private linear database by using the Damgård-Jurik encryption scheme with $s > 1$, but modular arithmetic is an operation that becomes more expensive as the operators increase. This section considers another means of privately retrieving plaintexts of greater length without increasing the bit-length of the operators involved in modular arithmetic.

5.3.1 Pre-Splitting Elements

Consider that the query $Q = \{\{q_{ij}\}_{i=0}^{c-1}\}_{j=0}^{\lceil \sqrt[c]{K} \rceil}$ as described in Section 4.3. Since Paillier's encryption scheme is semantically secure, the server has a $\frac{1}{K}$ chance of guessing which entry in the database the client is trying to retrieve. If this query was used on several similar databases, where databases are similar if a query can be generated to satisfy the server-side parameters of all of the databases, then the server could still have a $\frac{1}{K}$ chance of guessing which entry the client

received from all the databases, even though the client is retrieving the same entry from each database because the server only has a $\frac{1}{K}$ chance of guessing which item in the database that the query Q refers to. This suggests query reusability may not be a security risk. If query reusability is not a security risk, then if a database had entries in which the bit-length of the longest entry was $s\lceil \log_2 N \rceil$, where N is the public-key parameter of the client generated key pair for the Paillier encryption scheme, then each entry in the database could be split into s parts, creating s similar databases. The query generated for one of these server-side databases, using the the key pair previously mentioned, could be reused s times for execution on each of the s similar databases. The results from each of the s similar databases will then have to be concatenated on the client-sde after each result extraction. At this time it is not certain whether query reusability is as secure as s unique queries.

5.3.2 Implementing

See Appendix B.3 for pseudo-code implementation details.

1. Acquire Server Parameters

See Section 4.3.4 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, GetNumberOfParts and GetDataSize

2. Generate Query

See Section 4.3.4 for Chang-Query.

3. Process Query Against Database

See Section 4.3.4 for Chang-Perform-Retrieval and override Sigma, Split and Filter with the following.

Sigma(Q, ν, i, *index*)

Description Internal method that basically finds a Stern result along on particular dimension of the private database instead of the entire database as in the Stern protocol. However, the database elements have been split into 2 parts, so a pair will be returned.

Parameters

Q The client's private query.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

i The current element accessed in the second dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the result of processing the query against the innermost dimension of the database, a pair.

SPLIT(σ, N)

Description Internal method which splits a single intermediate result, a result from a SIGMA call so that each piece of the split may be a valid plaintext.

Parameters

σ A result from a SIGMA call.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ is the plaintext domain.

RETURNS The value of σ split into $2t = 4$ pieces. $parts$ =((u_0, v_0), (u_1, v_1)) such that $\sigma_0 = u_0 n + v_0$ and $\sigma_1 = u_1 n + v_1$

FILTER(R_s, Q, $parts$, ν)

Description Internal method which applies the parts from the result of processing the query on the innermost dimension as the second innermost dimension.

Parameters

R_s The server-side result from processing the private query against the database.

Q The client's private query.

$parts$ The pieces resulting from the call to SPLIT.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

Returns The server-side result from processing the query against the two innermost dimensions of the database.

4. Extract Result

CHANG-EXTRACT-RESULT(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database. The number of elements in the array R_s is t^{c+1}. Here $t = 2$.

Returns The plaintext result which should be the $i*$ entry of the private database.

5.4 A Collaboration of the Chang and Lipmaa Protocols

The Extended and Generalized Chang and the Lipmaa protocols used the Damgård-Jurik encryption scheme to increase the possible length of the plaintext records in the server-side private linear database, but the Extended and Generalized Chang held s constant and did not use the full plaintext space for encrypting intermediate ciphertexts, and the Lipmaa protocol increased s to accomodate encryption of intermediate ciphertexts which led to more computational complexity when considering that not all modular exponentiation operations have the same computational complexity. The following protocol is a collaboration in the sense that it uses the ideas of splitting as presented in the Chang protocol and adjustment of the ciphertext, by changing the value of s at each dimension of the hypercube, as presented in the Lipmaa protocol.

5.4.1 Using the Full Plaintext Domain

Both parts in the Chang protocol, $m_0, m_1 \in \mathbb{Z}_N$, and since $\mathbb{Z}_N$ is the plaintext domain $\mathcal{PT}$, it could be said that $\mathcal{PT}$ is fully used. The Extended and Genralized Chang protocols however do not fully utilize $\mathcal{PT}$ unless the Damgård-Jurik encryption scheme is equivalent to the Paillier encryption scheme, that is $s = 1$.

First, consider the splitting of an intermediate Damgård-Jurik ciphertext $\mu \in \mathbb{Z}^*_{N^{s+1}}$ as described in 5.1.1 as

$$\mu = \sum_{i=0}^{s} m_i N^{s-i}$$

Each $m_i \in \mathbb{Z}_N$, and $\mathbb{Z}_N$ becomes a much smaller subset of $\mathbb{Z}_{N^s}$, the plaintext domain $\mathcal{PT}$ of the Damgård-Jurik encryption scheme, as s increases, which is an under-utilization of the Damgård-Jurik $\mathcal{PT}$.

Now, consider the splitting of an intermediate Damgård-Jurik ciphertext $\mu \in \mathbb{Z}^*_{N^{s+1}}$ as described in 5.2.1 as

$$\mu = \sum_{i=\ell}^{s} m_i \cdot N^{s-i}$$

where $0 \leq \ell \leq s-1$. Obviously, $m_0 \in \mathbb{Z}_N$ for all valid choices of ℓ. It can be shown that $m_i \in \mathbb{Z}_N$ for $0 \leq i \leq \ell - 1$, and that $m_\ell \in \mathbb{Z}_{N^{s-\ell}}$. Thus,

if $\ell = s - 1 \Rightarrow t = 2$, then $m_1 \in \mathbb{Z}_{N^s}$ which is a full utilization of potential plaintexts, but since m_0 under-utilizes $\mathcal{PT}$, this shows that even the Generalized Chang protocol will under-utilize $\mathcal{PT}$ when encrypting parts from intermediate ciphertexts.

Since the number of bits transferred from the server to the client was the least in the Generalized Chang protocol when the $t = 2$, not to mention the number of encryptions was less, let $t = 2$. Consider splitting an intermediate Damgård-Jurik ciphertext $\mu \in \mathbb{Z}^*_{N^{s+1}}$ in to 2 parts of potentially equal bit-length

$$\mu = m_0 N^{\lceil \frac{s+1}{2} \rceil} + m_1$$

Notice $m_0, m_1 \in \mathbb{Z}_{N^{\lceil \frac{s+1}{2} \rceil}}$. The Lipmaa protocol increased the value of s so that an intermediate ciphertext could be encrypted. Now consider decreasing s to match $\mathcal{PT}$, that is the encryption of the parts m_0 and m_1 should be the pair

$$(\mu_0, \mu_1) = \left(\mathcal{E}\left(m_0, \left\lceil \frac{s+1}{2} \right\rceil \right), \mathcal{E}\left(m_1, \left\lceil \frac{s+1}{2} \right\rceil \right) \right)$$

As an example, let $c = 5$, and $s_0 = 17$, then the progression of adjustments to $\mathcal{PT}$ and $\mathcal{CT}$ provided by the length-flexibility of the Damgård-Jurik encryption scheme would be

$$(\mu_0, \mu_1) = \left(\mathcal{E}\left(m_0, \left\lceil \frac{17+1}{2} \right\rceil = 9 \right), \mathcal{E}\left(m_1, \left\lceil \frac{17+1}{2} \right\rceil = 9 \right) \right)$$

$$(\mu'_0, \mu'_1) = \left(\mathcal{E}\left(m_0, \left\lceil \frac{9+1}{2} \right\rceil = 5 \right), \mathcal{E}\left(m_1, \left\lceil \frac{9+1}{2} \right\rceil = 5 \right) \right)$$

$$(\mu''_0, \mu''_1) = \left(\mathcal{E}\left(\mu'_0, \left\lceil \frac{5+1}{2} \right\rceil = 3 \right), \mathcal{E}\left(\mu'_1, \left\lceil \frac{5+1}{2} \right\rceil = 3 \right) \right)$$

$$(\mu'''_0, \mu'''_1) = \left(\mathcal{E}\left(\mu''_0, \left\lceil \frac{3+1}{2} \right\rceil = 2 \right), \mathcal{E}\left(\mu''_1, \left\lceil \frac{3+1}{2} \right\rceil = 2 \right) \right)$$

$$(\mu''''_0, \mu''''_1) = \left(\mathcal{E}\left(\mu'''_0, \left\lceil \frac{2+1}{2} \right\rceil = 2 \right), \mathcal{E}\left(\mu'''_1, \left\lceil \frac{2+1}{2} \right\rceil = 2 \right) \right)$$

Since modular exponentiations increase in computational complexity with respect to the length of the operands, $s = 17$ is most likely impractical, considering that $\lceil \log_2 N \rceil \geq 1024$ bits, but taking a smaller example with $c = 3$ (if $c = 2$, then the Collaboration protocol similar to the Chang protocol with a larger N) and $K = 1000$, while the Lipmaa protocol and the Collaboration of Chang and Lipmaa protocol will have 1000 modular exponentiations with a base having $s\lceil \log_2 N \rceil$ bits while operating on the innermost dimension, the Lipmaa will have 100 modular exponentiations with a base having $(s+1)\lceil \log_2 N \rceil$ bits and the Collaboration will have 200 modular exponentiations with a base having $\lceil \frac{s+1}{2} \rceil \lceil \log_2 N \rceil$ bits while operating on the second innermost dimension. If all exponentiations were equal, then there would be an obvious savings in using Lipmaa over the Collaboration, but because all modular exponentiations are not equally complex, there is reason to consider that the Collaboration may be more efficient since the bit-length of the base nearly halves for each dimension of processing.

5.4.2 Implementing

See Appendix B.4 for pseudo-code implementation details.

1. Acquire Server Parameters

See Section 4.3.4 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, GetNumberOfParts and GetDataSize

See Section 4.4.2 for GetMaximumExponent.

2. Generate Query

Collaboration-Query($i*$)

Description Generates a query with one element encrypted indicators as described in Section 4.3.2. Where each element is the $\mathcal{E}(0, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$, unless the element is at index $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$, then the element if $\mathcal{E}(1, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$.

Parameters

$i*$ The index the client wishes to privately retrieve form the server.

Returns the private query to be sent to the server for retrieving index $i*$. $Q = \{\{q_i\}_{i=0}^{\sqrt[c]{K}-1}\}_{i=0}^{c-1}$ where $q_{ij} = \mathcal{E}(0, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$ for $j \neq \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$ and $q_{ij} = \mathcal{E}(1, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$ for $j = \lfloor \frac{i*}{\lceil \sqrt[c]{K} \rceil^{c-i-1}} \rfloor \mod \lceil \sqrt[c]{K} \rceil$

3. Process Query Against Database

Server-side Invocation

See Section 4.3.4 for Chang-Perform-Retrieval and override C-Hypercube, Two-Hypercube with the following.

C-Hypercube(Q, N, ν, d, $index$)

Description Internal method used for recursing through all the dimensions (indices) of reference except for the two inner-most indices.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ the plaintext domain.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

d The distance of the current dimension from the outermost dimension.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the server-side result after processing the query against the private database. R_s

TWO-HYPERCUBE(Q, N, ν, *index*)

Description Internal method used for processing the query against the two innermost dimensions.

Parameters

Q The client's private query.

N The modulus of the plaintext domain, where $\mathbb{Z}_N$ the plaintext domain.

ν The modulus of the ciphertext domain, where $\mathbb{Z}_\nu$ is the ciphertext domain.

index The cummulative index from combining the indexes from within, the previous dimensions.

Returns the result of processing the query against the two innermost dimensions of the database.

4. Extract Result

COLLABORATION-EXTRACT-RESULT(R_s)

Description Extracts the plaintext result of the query processed against the private database from the server-side result. The plaintext result should be the $i*$ entry of the private database.

Parameters

R_s The server-side result from processing the private query against the database. The number of elements in the array R_s is t^c. Here $t \leq s$.

Returns The plaintext result which should be the $i*$ entry of the private database.

Chapter 6

Comparisons

In Stern[Ste98], Chang[Cha04], and Lipmaa[Lip04], the primary goal of each protocol was to reduce the amount of communication. That is, each protocol was deemed an improvement over previous protocols only if the amount of communication between the client and server for the duration of the entire protocol was lower. It will be seen by example that the protocol presented by Lipmaa[Lip04] did not beat out Chang[Cha04] in the realm of communication all the time. Considering a communication channel that will support a consistent $28.8Kb/s$, which is simply a cheap, if not obsolete, dial-up modem running at maximum capacity, the time spent on communication could potentially be quite negligible compared to the time spent on computations. Thus, the better protocol should be the protocol that performs the least number of operations, if modular exponentiation operations are expensive for large moduli, which in most cases is the opposite of the protocol with the least number of bits communicated.

The experiments performed and discussed in this chapter use the multi-presion package `java.math.BigInteger`. Multi-presicion packages are quite efficient to the extent that using this package to increase the readability of the code, will barely, if at all, hinder the performance of the protocols. Through experimentation, it can be seen that the time spent performing a `modPow` operation provided in `java.math.BigInteger` increases polynomially as the length of the operands increases. Figure 6.1 shows that the average length of time, with a sample size of 20, the time spent on a modular exponentiation grows exponentially with bit-length. The same test was performed for modular multiplication using `multiply` followed by `mod`, and it was found that this operation combination takes under 6 milliseconds.

6.1 Theoretical Comparison

Since the Chang protcol is strongly limited to the bit-length of the key, a fair comparison of the three published protocols, Stern [Ste98], Chang [Cha04], and

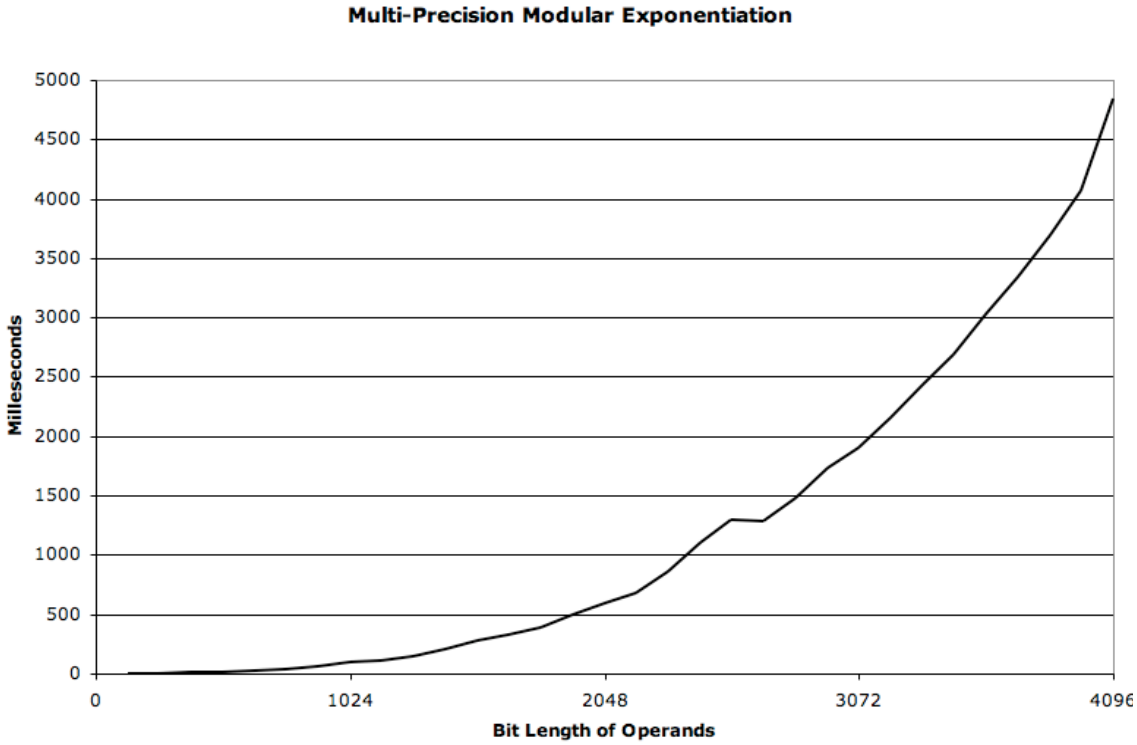

Figure 6.1: Using `java.math.BigInteger` for multi-precision operations such as modular exponentiations, it can be seen that the time spent to complete a `modPow` call grows polynomially as the bit-length of the operands increases.

Lipmaa [Lip04] would be to limit the length of the data to $\lceil \log_2 N \rceil$, or in terms of the Damgård-Jurik cryptosystem maintain $s = 1$. The new protocol, the Collaboration of Chang and Lipmaa Protocol, introduced in this thesis in Section 5.4 was devised to use the length-flexibilty of the Damgård-Jurik cryptosystem for the retrieval of data larger than $\lceil \log_2 N \rceil$, and thus is discussed separately in comparison to the latest published protocol, Lipmaa. Comparisons are made assuming $\lceil \log_2 N \rceil = 1024$. While it is suggested for commercial applications to use at least 2048, 1024 bits is still secure enough for this kind of protocols.

6.1.1 Data with Bit-Length Less Than $\lceil \log_2 N \rceil$

From the descriptions of the client-side phase of query generation in Chapters 4 and 5, the number of encryptions necessary and the number of bits in the resulting query can be determined to be as listed in Table 6.1, and from the descriptions of the client-side phase of extracting the result in Chapters 4 and 5, the number of decryptions necessary can be determined to be as listed in Table 6.1. The same can also be determined from the descriptions of the server-side processing of the query in Chapters 4 and 5. The number of modular exponentiations and the number of bits sent is listed in Table 6.2.

As was agreed in [Cha04] and [Lip04], the more dimensions the smaller the query which results in less client to server communication as can be shown in Figure 6.3 with the $\lceil \log_2 N \rceil = 1024$. The smaller query also had proportionally fewer encryption operations to obtain it, see Figure 6.2. From these figures, it is easy to assume that more dimensions is better because the greater the number of dimensions the less time spent on both computation and communication, but this only covers the phases of Query Generation and Client Communicating to Server.

In section 5.1, it was conjectured that the Extension to the Chang Protocol was not an improvement, but rather serves as a stepping stone to the Collaboration of Chang and Lipmaa Protocol. To assert that conjecture as true, notice in Figure 6.4, that the number of decryptions increases polynomially faster than any of the other Chang-based protocols, except Pre-Split, so the Extended Protocol will only be dragged along through the experiments to further assert the original conjecture as true. Figure 6.4 shows the number of decryptions increases with the number of dimensions and this number of decryptions is roughly proportional to the increase in number of bits sent from the server to the client, see Table 6.3. This would suggest that increasing the number of dimensions hinders the performance of the protocol, but the number of encryptions and bits sent from the client to the server outweigh the number of decryptions and bits sent from server to client vastly, especially as the size of the database K increases, so the time spent in the extraction phase could potentially be negligible.

The last phase to consider is the server-side processing of the private query on the private database. Figure 6.5 shows that as the dimensions increase, the number of modular exponentiations increases, and the Chang protocol in-

	Encryptions	Decryptions	Bits Sents
Stern	K	1	$K\lceil \log_2 m \rceil$
Chang	$c\sqrt[c]{K}$	$\sum_{i=0}^{c-1} 2^i$	$2c\sqrt[c]{K}\lceil \log_2 N \rceil$
Extended	$c\sqrt[c]{K}$	$\sum_{i=0}^{c-1} (s+1)^i$	$(s+1)c\sqrt[c]{K}\lceil \log_2 N \rceil$
Generalized	$c\sqrt[c]{K}$	$\sum_{i=0}^{c-1} r^i$	$(s+1)c\sqrt[c]{K}\lceil \log_2 N \rceil$
Pre-Split	$c\sqrt[c]{K}$	$\sum_{i=0}^{c} 2^i$	$2c\sqrt[c]{K}\lceil \log_2 N \rceil$
Lipmaa	$c\sqrt[c]{K}$	$\sum_{i=0}^{c-1} 2^i$	$(\sum_{i=1}^{c} s+i)\ \sqrt[c]{K}\lceil \log_2 N \rceil$
Collaboration	$c\sqrt[c]{K}$	$\sum_{i=0}^{c-1} 2^i$	$\left(\sum_{i=1}^{c} \frac{s-1}{2^{c-i}} + 2\right)\sqrt[c]{K}\lceil \log_2 N \rceil$

Table 6.1: Client-Side Operation Comparison

	Exponentiations	Bits Sents
Stern	K	$\lceil \log_2 M \rceil$
Chang	$\sum_{i=0}^{c-1} 2^i \sqrt[c]{K}^{c-i}$	$2^c\lceil \log_2 N \rceil$
Extended	$\sum_{i=0}^{c-1} (s+1)^i \sqrt[c]{K}^{c-i}$	$(s+1)^c\lceil \log_2 N \rceil$
Generalized	$\sum_{i=0}^{c-1} r^i \sqrt[c]{K}^{c-i}$	$(s+1)r^{c-1}\lceil \log_2 N \rceil$
Pre-Split	$\sum_{i=0}^{c-1} 2^{i+1} \sqrt[c]{K}^{c-i}$	$2^{c+1}\lceil \log_2 N \rceil$
Lipmaa	$\sum_{i=1}^{c} \sqrt[c]{K}^{i}$	$(s+c)\lceil \log_2 N \rceil$
Collaboration	$\sum_{i=0}^{c-1} 2^i \sqrt[c]{K}^{c-i}$	$2^c\lceil \frac{s+1}{2^c} + \sum_{i=0}^{c-1} \frac{1}{2^i} \rceil \lceil \log_2 N \rceil$

Table 6.2: Server-Side Operation Comparison

c	Stern	Chang-based ($t = 2$)	Lipmaa
1	2056	-	-
2	-	8224	3074
3	-	16448	4096
4	-	32896	5124
5	-	65792	6152

Table 6.3: The number of bits sent from the server to the client is independent of the size of the database; however, since the number of bits sent from the client to server decreases with dimension, a reasonable tactic would be to increase the dimensions of reference which will increase the server to client communication.

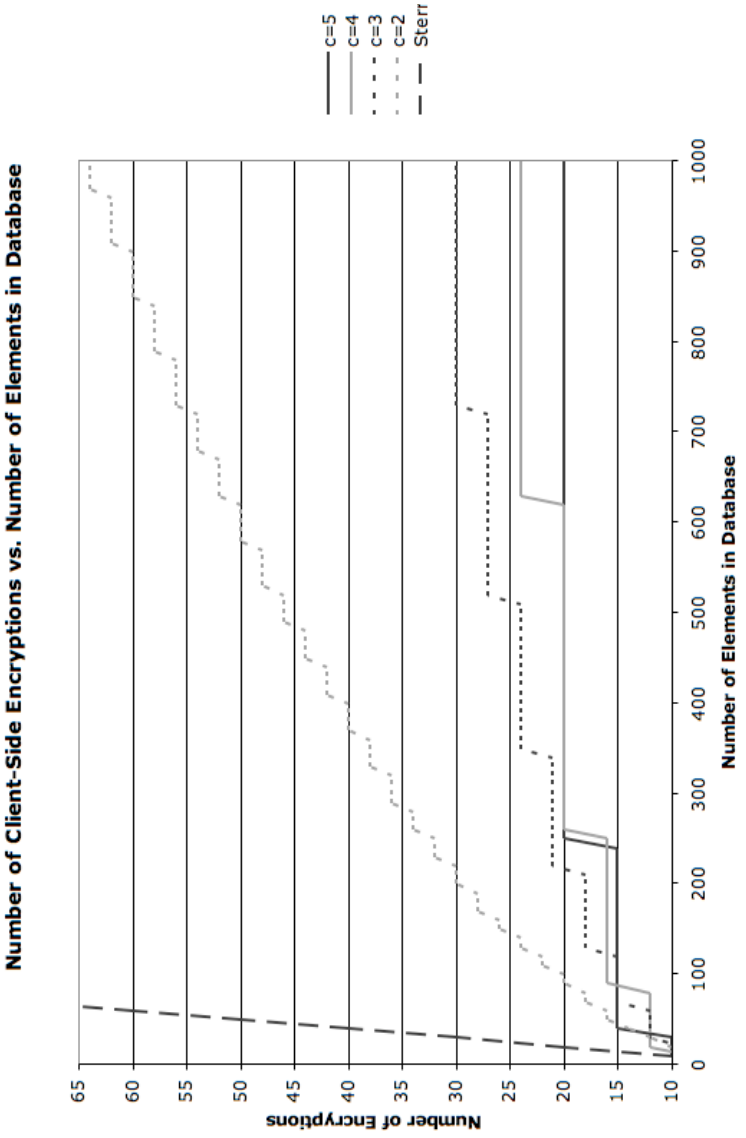

Figure 6.2: A comparison of the number of server-side encryptions. Comparison assumes that the bit-length of the operands involved in the encryption has no affect on the computational complexity of the encryption function.

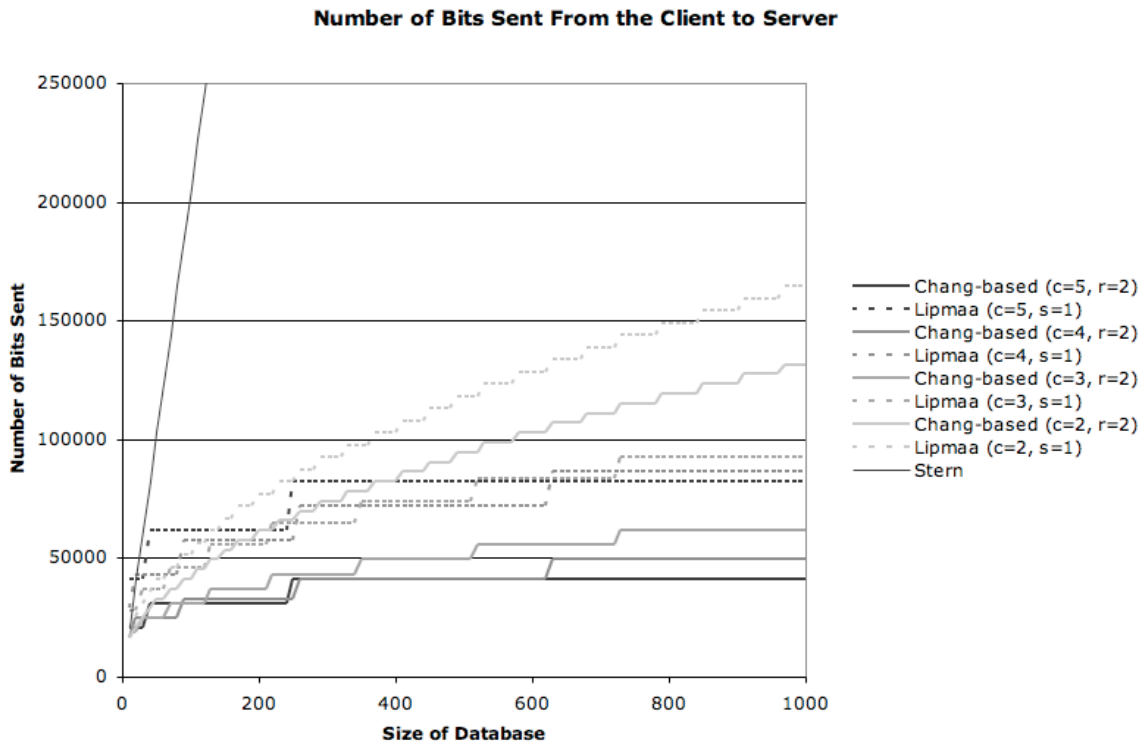

Figure 6.3: A comparison of the number of bits sent from the client to the server. This comparison more accurately compares the computationally complexity, since the bits sent is the sum of the bit-lengths of ciphertexts resulting from each encryption.

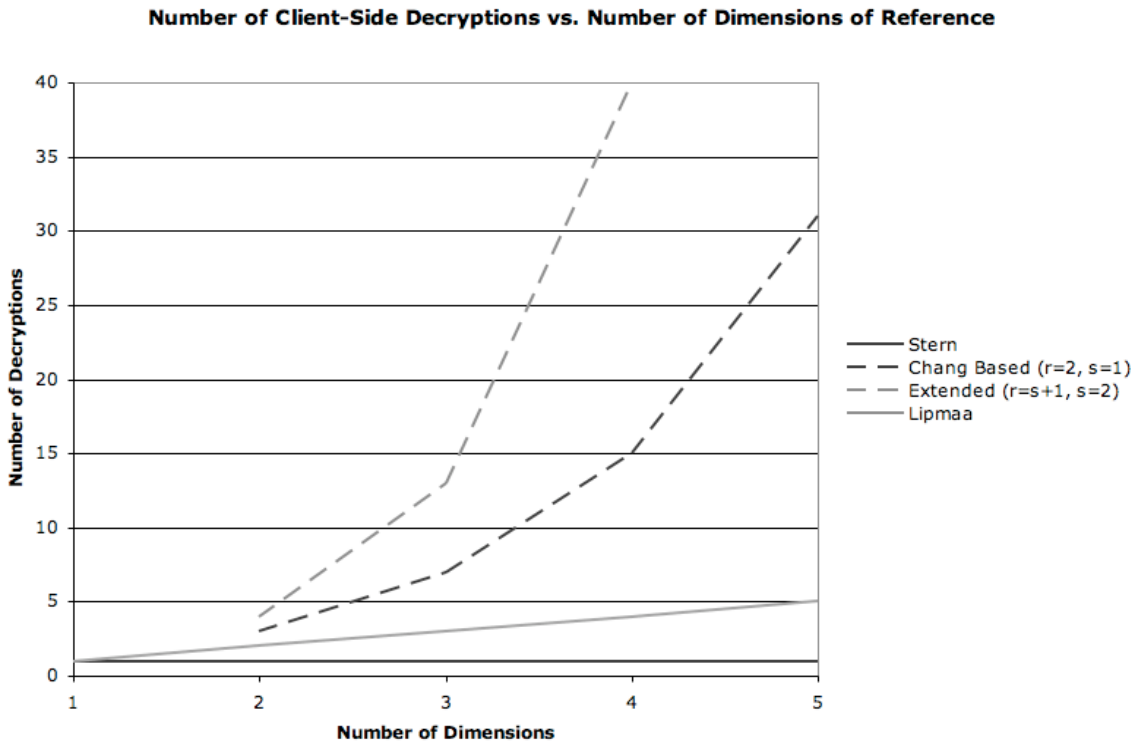

Figure 6.4: The client-side decryptions is directly related to the number of bits sent from the server to the client,a dn as with encryptions, these decryptions are not necessarily of the same bit-length. The number of bits sent from the server to the client is a better comparison. The number of decryptions again is independent of the database size and dependent on the number of dimensions. The number of decryptions is still significantly smaller than the number of encryptions.

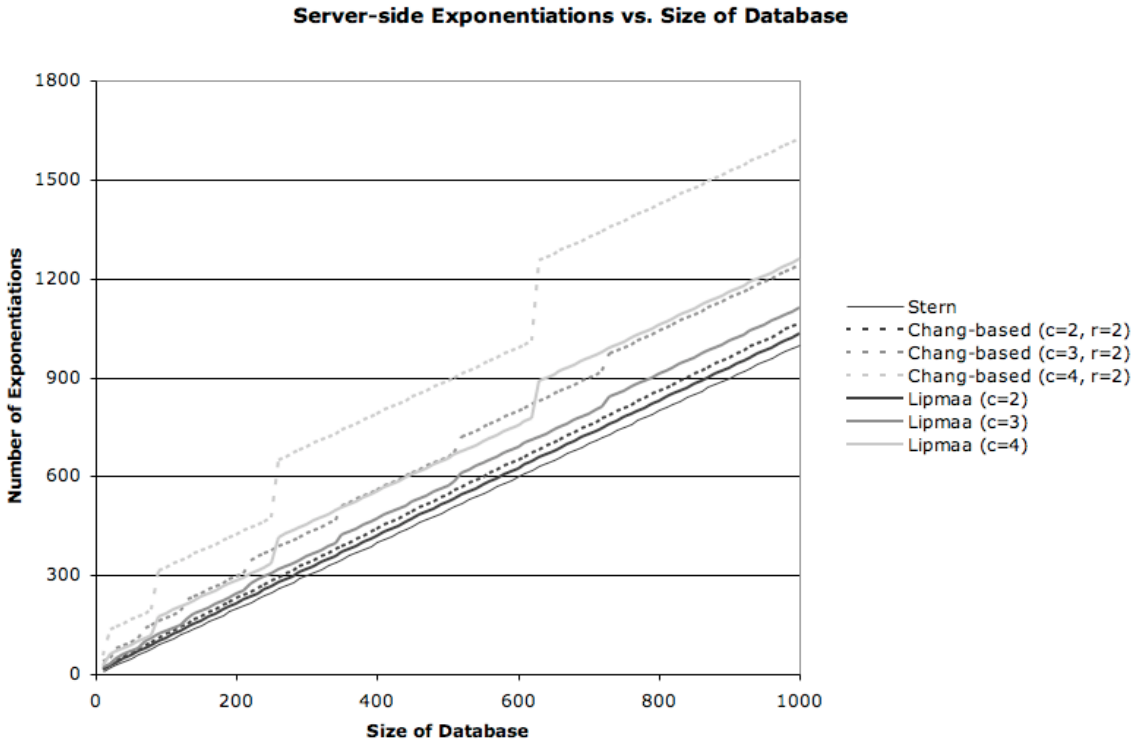

Figure 6.5: As the size of the database and the number of dimensions increase, the number of server-side exponentiations. This raises the question of how many dimensions should be used to balance the decreasing client-side computational complexity with the increasing server-side complexity.

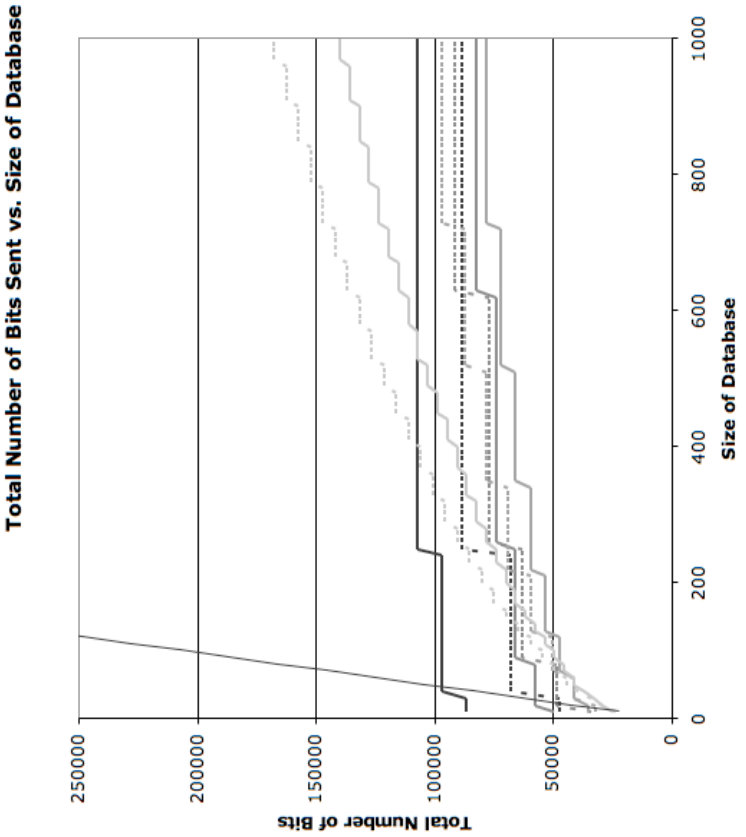

Figure 6.6: The number of bits sent between the two parties is a better indicator of computational complexity than the raw number of modular exponentiation algorithms since the bit-length of the operands vary by protocol setup.

creases much faster than the LIPMAA protocol because the operation of splitting the intermediate results, doubles the number of modular exponentiations per dimension. The number of server-side modular exponentiation operations is far greater than the number of encryptions during client-side query generation, that unlike with the discussion of the extracting the result phase, this phase cannot be considered negligible. This suggests that if the client and server have the same computational power, that the increase in dimensions is a hinderance on the performance. However, if the server is much more computationally inclined than the client, and can dedicate those resources to the single client, then the time spent on client-side query generation and server-side processing of query could be comparable.

Communicationally, since most of the protocols communicate less than $28.8Kb$, the total communication time for each protocol is mere seconds. However, the Stern protocol may be several seconds. Computationally, for the client-side, a CHANG-based protocol with 2 parts and higher dimensions will be faster given the number of server-side elements K is not trivially small, see [Cha04]. If the time to encrypt and decrypt larger operands is small, then the LIPMAA protocol could be computationally faster, since the number of decryptions only increases linearly rather than exponentially with the number of dimensions. Computationally, for the server-side, the Stern protocol is faster since the number of modular exponentiations is equal to the number elements in the private database K, and for all other protocols, the number of modular exponentiations is greater than K.

6.1.2 Data with Bit-Length Greater Than $\lceil \log_2 N \rceil$

Since the number of encryptions performed during client-side query generation and the number of bits sent from the client to the server, that is the number of bits in the private query, are proportional, the bits sent from the client to server is a more accurate indicator of the computational complexity of the client-side query generation phase because not all encryptions in the query generation phase of the Lipmaa and Collaboration of Chang and LIPMAA protocols are equal. The LIPMAA protocol calls for each column of the private query matrix to increase s by 1 and the Collaboration protocol calls for each column of the private query matrix to decrease s by roughly half per column. So as one moves along the columns of the private query during generation, one will notice that the encryptions become gradually longer in Lipmaa, and significantly smaller in Collaboration. As can be seen in Figure 6.7, Lipmaa queries are significantly larger than Collaboration queries for large data. The figure does not display the comparison in lower dimensions because for the cases where $2 \leq c \leq 3$ are roughly the same for $s < 5$.

The number of bits sent from the server to the client, is a reasonable indicator of the number of decryptions involved during client-side extraction of result. Again the number of decryptions is outweighed by the number of encryptions performed by the client and the number of modular exponentiations performed by the server, so this phase is still relatively negligible except that the bit-length

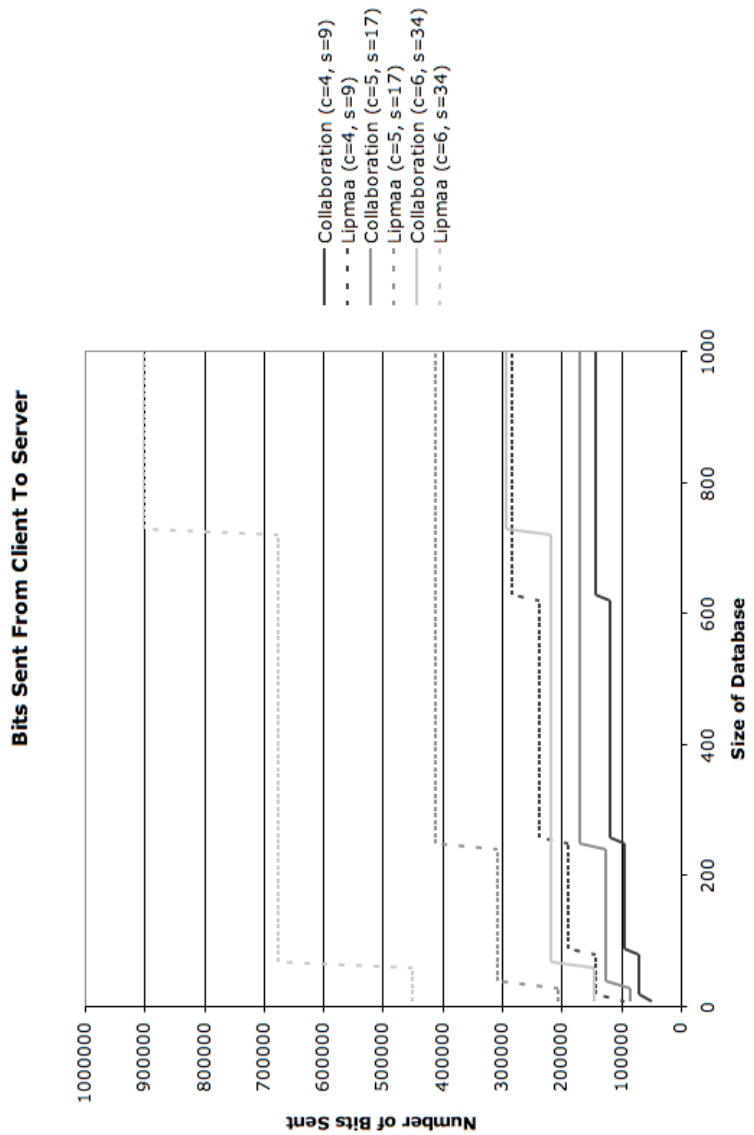

Figure 6.7: Since the bit-length of the operands in each encryption will vary, comparison by the number of encryptions is frivolous. Again the number of bits sent from the client to the server is directly related to the number of client-side encryptions.

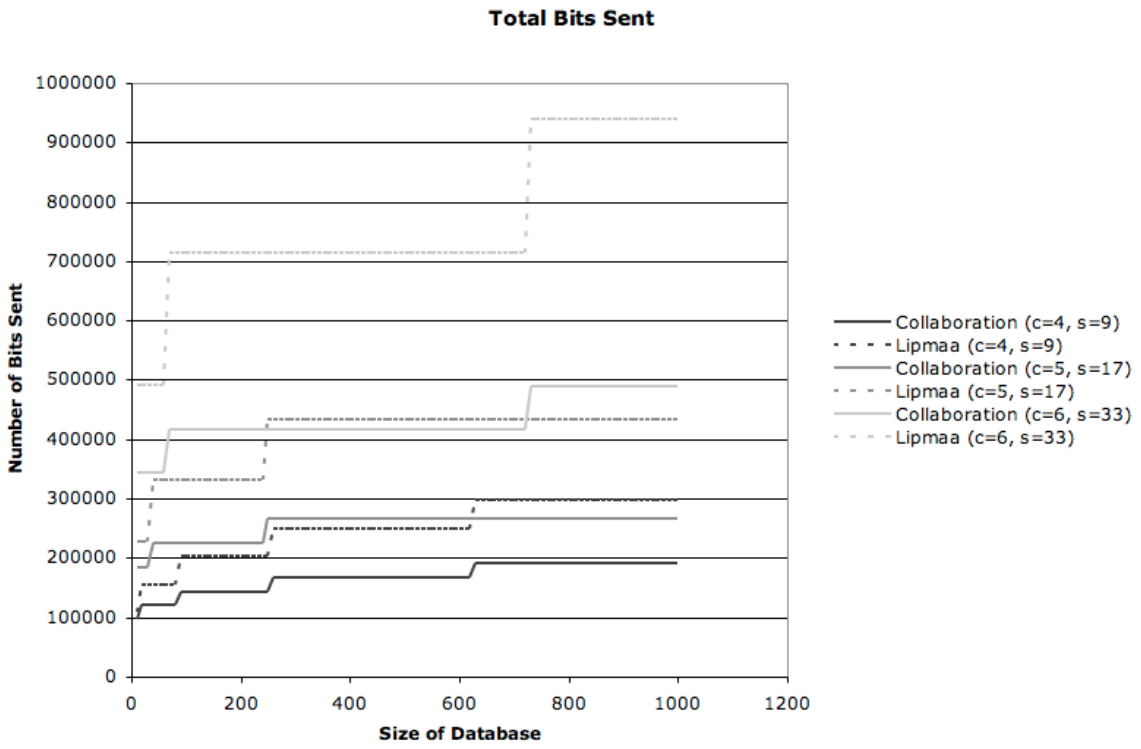

Figure 6.8: The exponential increase in the number of bits sent from the server to the client as the number of dimensions increases, decreases the difference in performance between the LIPMAA and COLLABORATION protocols.

of operands involved in decryption are significantly smaller for the Collaboration protocol than the Lipmaa, so even though there are exponentially many more decryptions for the Collaboration protocol, this could be offset by the decryption operations being much shorter.

The number of modular exponentiations performed by the server are the same as in Figure 6.5, considering the Collaboration protocol as a CHANG-based protocol. This comparison is not completely accurate, for the same reasons as client-side query generation and client-side extraction of result. The bit-length of the operands in the modular exponentiations is not equal. While the Collaboration protocol will have many more modular exponentiations, it will have operations acting on much shorter operands than Lipmaa.

Computationally it is tough to say whether Collaboration or Lipmaa will be faster. It is pretty certain that with the number of encryptions performed by the client being equal that the Collaboration protocol will produce a query faster because the bit-length of the operands is smaller, assuming again that as the bit-length of the operands increases the computational complexity. Communicationally it easy to see that Collaboration will be faster, but since the number of bits involved in communication is smaller than $28.8Kb$, the overall communication time for each will be mere seconds anyways.

6.2 Experimental Comparison

The following experimental results were acquired by running the source code included with the enclosed CD with the scripts in Appendix D on Sun Blade 1500 with a UltraSPARC IIIi 1.0GHz processor and 1GB of RAM. The programs running in the background, and the users logging into the computer were uncontrolled, but effort was made to find the least used workstations which statisfied these specifications. All times in the tables contained in this section are in milleseconds (1/1000s).

The architecture of the software for these protocols is such that the relationship of previous protocols to later protocols is made apparent through polymorphism. From an educational view point, this will allow others to better understand how these protocols work. Unfortunately, the introduction of broad generalizations can diminish the test of performance. While some protocols should theoretically perform better and in actuality might perform better, with the generalizations introduced this may not be the case.

The sections are broken up to compare the protocols based on the length of data that can be retrieved. Some protocols may perform better on a database with longer entries. Within each section, the affect of changing the dimensions, or number of indices of reference, is adjusted to see how the number of dimensions affects performance.

c	s	Collaboration	Lipmaa
2	3	12336	5140
3	5	24672	8224
4	9	49344	13312
5	17	98688	22616
6	33	197376	40092

Table 6.4: Due to splitting, the number of bits sent from the client increases exponentially for the Collaboration protocol. This suggests that if the Collaboration will prevail over Lipmaa a larger number of dimensions could prove to be a hinderance.

Phase	Stern	Chang			Lipmaa		
$c =$	1	2	3	4	2	3	4
Query	7897	5331	5309	7699	22496	25997	23376
Process	6266	9392	19843	59190	32864	26755	17480
Extraction	1272	3797	8822	19850	2086	4488	8331
Total	15435	18520	33974	86739	57446	57240	49187
Phase		Extended			Generalized		
$c =$		2	3	4	2	3	4
Query		5058	5371	6379	6147	8947	6601
Process		9399	16057	36589	11310	39786	55730
Extraction		1914	4826	9512	1960	4493	9440
Total		16371	26254	52480	19417	53226	71771

Table 6.5: $K = 10, s = 1$. The simple and straightforward Stern protocol prevails with the small database size.

6.2.1 Data with Bit-Length Less Than $\lceil \log_2 N \rceil$

From Tables 6.5 and 6.6, several things can be noticed. First, the length of time spent on extraction is clearly independent of the size of the server-side private database. Second, the architecture of the software, that is, using a significant amount of generalization, may not have introduced as many measurement errors as was feared. Since the EXTENDED and GENERALIZED protocols are exactly the same as the CHANG protocol when $c = 2$ and $s = 1$, and the times are roughly the same. Third, the LIPMAA protocol, though published after the CHANG protocol was boasted an improvement, it failed to compete with the CHANG protocol. Fourth, increasing the number of dimensions is indeed a hinderance on performance. It could be argued that 10 and 100 do not have second, third, as well as fourth roots, so you are always acting on a database with varying quantity of null entries, except for $100 = 10^2$, but in the cases where $c = 2$ while $K = 10$ and $c = 4$ while $K = 10$, the product $\ell c = 8$ will the number of null entries being 6 for both cases. The total time for the case where $c = 4$ while $K = 10$ is greater than the total time for the case where $c = 2$ and $K = 10$. Thus, increasing the number of dimensions is a hinderance on performance.

6.2.2 Data with Bit-Length Less Than $2\lceil \log_2 N \rceil$

In Chapter 5, the Chang Protocol was EXTENDED then GENERALIZED and then went back to the Chang to ponder whether you could reuse the client-side generated query several times on Pre-Split entries. These variations serve as stepping stones towards the Collaboration of Chang and LIPMAA protocol, but these could serve as contenders for the LIPMAA protocol when $s > 1$. Since these protocols were equivalent to the CHANG protocol when $s = 1$ and the CHANG protocol outperformed the LIPMAA protocol in Section 6.2.1, there is a reasonable likelihood that at least PRE-SPLIT and the GENERALIZED could perform at least as well as the LIPMAA protocol for $s > 1$.

From Tables 6.7 and 6.8, it can be seen that Lipmaa still fails to compete with the CHANG-based protocols. The extraction time is low, but in general for all of these protocols, the extraction phase has the least number of operations, thus it is easy to have the fastest extraction time and still be the slower protocol.

The EXTENDED, as mentioned before, begins to lose its speed very quickly. By increasing s to 2, the EXTENDED protocol runs nearly three times slower; whereas, the GENERALIZED is running roughly the same with $s = 2$ as it did with $s = 1$. Thus, the EXTENDED will be considered no further. The Pre-Split did not run much faster than the GENERALIZED, and since the security of PRE-SPLIT is questionable, PRE-SPLIT protocol will be dropped from consideration as well.

6.2.3 Data with Bit-Length Less Than $4\lceil \log_2 N \rceil$

Table 6.9 presents further evidence that CHANG-based protocols out perform LIPMAA. This reinforces that the bit-length of the operands involved in modu-

Phase	Stern	Chang			Lipmaa		
$c =$	1	2	3	4	2	3	4
Query	64444	10452	9183	9162	55651	41525	45687
Process	64417	39448	99029	195340	325089	199767	115964
Extraction	1278	3800	8834	18884	2054	4636	8374
Total	130139	53700	117046	223386	382794	245928	170025
Phase		Extended			Generalized		
$c =$		2	3	4	2	3	4
Query		10741	9093	89825	14063	11531	12135
Process		41035	70227	122526	76674	158007	380851
Extraction		1976	4442	9490	1928	4586	9487
Total		53770	83762	140941	92665	174124	402473

Table 6.6: $K = 100, s = 1$. The larger database size significantly increased the time spent generating a query for the Stern protocol, which indicates that saving time by decreasing the query is a strength. Since Chang surpasses Lipmaa, this confirms the suggestion that the impact of more decryptions is not as significant as having a larger query.

Phase	Pre-Split			Lipmaa		
$c =$	2	3	4	2	3	4
Query	5299	5316	6639	57481	64283	57561
Process	16563	29573	82561	78702	97503	90637
Extraction	6311	15755	35261	2027	445	8398
Total	28173	50644	124461	138210	166241	156596
Phase	Extended			Generalized		
$c =$	2	3	4	2	3	4
Query	17802	19132	23498	22593	25830	23389
Process	24859	55193	172399	30219	122618	175816
Extraction	5599	19164	55985	4198	9783	20885
Total	48260	93489	251882	57010	158231	220090

Table 6.7: $K = 10, s = 2$. Pre-Split and Generalized are essentially the same protocol. The Chang-based protcol once again surpasses the later published Lipmaa protocol.

lar exponentiation affects the computational complexity of the operation. This also suggests that the polynomial growth in the complexity of the modular exponentiation operation surpasses the polynomially increasing number of parts to be encrypted during the application of the private query on the private database in a CHANG-based protocol.

6.2.4 Comparisons With Collaboration

The Collaboration of Chang and LIPMAA protocol should provide relatively good performance when s is significantly large. Also, since it was seen that increasing the number of dimensions is more of hinderance than a help, only cases where $c = 2$ will be considered.

Looking at Tables 6.10, it can be seen that when $s = 3$, the COLLABORATION protocol surpasses the GENERALIZED protocol. Thus, it is reasonable to assume that the COLLABORATION protocol is the most efficient CHANG-based protocol when s is large. As in all previous experiments, the CHANG-based protocol surpasses the LIPMAA protocol. As s increases, however, the LIPMAA performance to COLLABORATION performance ratio slowly decreases, but the time spent retrieving data increases much faster to the point where large s becomes unreasonable for the computations.

6.3 Conclusions and Future Work

The hierarchical architecture of both the Cryptosystem and Private Information Retrieval libraries add value as a means for conveying the relationships of newer cryptosystems and protocols to the early cryptosystems and protocols so that the derivations of one cryptosystem or protocol from another is more apparent. However, this did increase the risk for experimental error. It could be beneficial to verify the results found in this thesis with software architected to contain little to no generalizations.

It was shown that larger operands in modular exponentiations do increase the computational complexity of the protocol. Recent advances have created chips that contain a cryptographic coprocessors that can perform faster modular exponentiations with larger operands via the Montgomery Fast Modular Exponentiation algorithm embedded in the coprocessor. This may prove useful in sending larger amounts of data. Since very few files of any meaning are under a megabyte.

Experiments were not performed to prove that the time spent on communication was small, but in future experiments, the networking could be added, though it is highly doubtful that such experiments would disprove what was conjectured here. Communication will only become an issue once these protocols can perform computations on on data entries exceeding maybe a megabyte, but with more and more consumers leasing high speed connections, it is still doubtful that communication will be a bottleneck for such protocols.

Furthermore, [Cha04] and [Lip04] introduced referencing the database as a hypercube to decrease the communication, but at the cost of even more time spent on computation. Since computations consume nearly all of the time spent executing the protocol, this finding produces another reason to not focus so much on reduction of communication.

The CHANG protocol with Pre-Split database entries could be a potential means of increasing the amount of data communicated by the ability to send a single query to retrieve an entry one kilobit at a time, and pipelining the execution of server-side processing on the next chunk with the client-side extraction of the most recently received chunk. Even the possibilty of the server performing multiple processing phases on contiguous chunks could speed up the process of retrieving larger data privately, but it would have to be proven that reuse of a query is indeed secure.

If attempting to find the protocol and the parameters that could accomplish "practical" PIR in one CPU minute, that would be a tough decision. A "practical" scheme could require the data be of a certain size, the database be a certain size, or a balance of the two. From hypotheses and experiments, there is no doubt that the number of dimensions should be fixed at $c = 2$. If the goal is to retrieve from a larger database and the data length is free, then the CHANG protocol (or GENERALIZED or COLLABORATION) with $K = 100$ and $s = 1$ with $N = 1024$ will provide "practical" PIR. However, if the length of the data is more important, then the COLLABORATION protocol with $K = 9$ and $s = 3$ while $N = 1024$ would be practical. Regardless, the value K is still significantly small, considering an online music source such as iTunes©provides access to over 2 million songs.

Eventually there will be a 100-fold increase in the CPU power available to a user. As CPU power increases, the ability to break an N-bit encryption increases. Assuming, for simplicity, that the bit-length necessary to create a nearly invulnerable encryption increases directly with the CPU power, which is not necessarily true, then N will increase from 1024 bits to 12.8kB. A 12.8kB file could be a length text file or a short PDF. That PDF could be a academic journal article, and it could be retrieved with the CHANG protocol. Just by setting $s = 2$, the length of the data retrieved could be as long as 25.6kB.

The problem of Private Information Retrieval consists of much more than was defined in Chapter 1. The problem consists of having to read every piece of data in the database an equal number of times and the improbability of reducing computational complexity because such a reduction in complexity would most likely lead to a reduction in security.

Phase	Pre-Split			Lipmaa		
$c =$	2	3	4	2	3	4
Query	10330	9412	9829	143513	110511	113492
Process	89570	176090	339599	779757	613612	501879
Extraction	6312	15762	36561	2025	4510	8381
Total	106221	201264	385989	925295	728633	623752
Phase	Extended			Generalized		
$c =$	2	3	4	2	3	4
Query	42223	33767	33862	57426	44730	46242
Process	103986	204334	448069	193486	450608	1172553
Extraction	5997	18207	55694	4445	10045	20954
Total	152206	256308	537625	255357	505383	1239749

Table 6.8: $K = 100, s = 2$. The Extended protocol was presented as a stepping stone to Generalized and then Collaboration, and this protocol will be considered no further as the complexity is increasing rapidly.

K=10						
Phase	Lipmaa			Generalized		
$c =$	2	3	4	2	3	4
Query	118030	132292	118527	57791	64587	57472
Process	148064	239979	267822	62341	277336	400020
Extraction	2032	4481	8340	7689	17361	37004
Total	268126	376752	394689	127821	359284	494496

Phase	Lipmaa			Generalized		
$c =$	2	3	4	2	3	4
Query	210465	237269	208484	116845	136122	117068
Process	239715	473152	594116	110875	528578	801187
Extraction	2039	4559	8369	11745	27072	59180
Total	452219	714980	810969	239465	691772	977435

K=100						
Phase	Lipmaa			Generalized		
$c =$	2	3	4	2	3	4
Query	292066	218687	233184	144555	108291	115252
Process	1520228	1329336	1379164	351863	963495	2634033
Extraction	2062	4526	8385	7558	17350	36806
Total	1814356	1552549	1620733	503976	108136	2786091

Phase	Lipmaa			Generalized		
$c =$	2	3	4	2	3	4
Query	580101	395996	417609	312427	223578	233989
Process	2536055	2368644	2914973	58233	1779283	5062250
Extraction	2045	4508	8659	11811	27172	58437
Total	3118201	2769148	3341241	907471	2030033	5354676

Table 6.9: $K = 10, s = 3$, $K = 10, s = 4$, $K = 100, s = 3$, and $K = 100, s = 4$. This is verification that many smaller modular exponentiations is faster than fewer larger modular exponentiations.

Phase	Lipmaa			
s=	3	5	9	17
Query	118557	341786	1509086	7685477
Process	148439	373942	1079521	3526029
Extraction	14061	38875	113068	659771
Total	281057	754603	2701675	11871277
Phase	Collaboration			
s=	3	5	9	17
Query	40578	134243	621103	7518219
Process	46014	110876	358451	1638035
Extraction	5278	10623	28503	229869
Total	91870	255742	1008057	9386123

Table 6.10: Lipmaa vs. Collaboration with K=10 and c=2. The Collaboration protocol shows to be much faster, and the difference shows the true affect of reducing the bit-length of the operands involved in modular exponentiations.

Appendix A

Previous PIR Implementation Details

A.1 Stern Protocol

See Section 4.2.2 for method headers.

A.1.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

GetDatabaseSize()
1 **return** K

GetDataSize()
1 **return** $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$

A.1.2 Generate Query

Client-Side Invocation

Stern-Query($i*$)
1 $Q[K]$
2 **for** $i \leftarrow 0$ **to** K
3 **do if** $i = i*$
4 **then** $Q[i] \leftarrow$ Encrypt(1)
5 **else** $Q[i] \leftarrow$ Encrypt(0)
6 **return** Q

A.1.3 Process Query Against Database

Server-side Invocation

STERN-PERFORM-RETRIEVAL(Q, m)
1 R_s
2 **for** $i \leftarrow 0$ **to** K
3 **do** $R_s = (R_s \cdot Q[i]^{db[i]}) \mod m$
4 **return** R_s

A.1.4 Extract Result

Client-Side Invocation

STERN-EXTRACT-RESULT(R_s)
1 **return** DECRYPT(R_s)

A.2 Chang Protocol

See Section 4.3.4 for method headers.

A.2.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

GETDATABASESIZE()
1 **return** K

GETNUMBEROFDIMENSIONS()
1 **return** c

GETDIMENSIONLENGTH()
1 **return** $\lceil \sqrt[c]{K} \rceil$

GETDATASIZE()
1 **return** $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$

GETNUMBEROFPARTS()
1 **return** 2

A.2.2 Generate Query

Client-Side Invocation

CHANG-QUERY($i*$)
1 $Q[c, \sqrt[c]{K}]$
2 **for** $i \leftarrow 0$ **to** c
3 **do for** $j \leftarrow 0$ **to** $\sqrt[c]{K}$
4 **do if** $j = (i * /(\sqrt[c]{K})^{c-i-1}) \mod (\sqrt[c]{K})$
5 **then** $Q[i, j] =$ ENCRYPT(1)
6 **else** $Q[i, j] =$ ENCRYPT(0)
7 **return** Q

A.2.3 Process Query Against Database

Server-side Invocation

PERFORM-RETRIEVAL($Q[][], n$)
1 **return** C-HYPERCUBE($Q, n, n^2, 0, 0$)

C-HYPERCUBE($Q[][], n, m, d, index$)
1 $R_s[numSplits^{c-d-1}$
2 **if** $c - d = 2$
3 **then** $R_s \leftarrow$ TWO-HYPERCUBE($Q, n, m, d + 1, index \cdot (\sqrt[c]{K})^d$)
4 **else** **for** $k \leftarrow 0$ **to** $\sqrt[c]{K}$
5 **do** $prevParts \leftarrow N - Hypercube(Q, n, m, d + 1, index \cdot (\sqrt[c]{K})^d)$
6 $currParts \leftarrow N - Split(prevParts, n)$
7 $R_s \leftarrow N - Filter(R_s, Q, currParts, m, k, d)$
8 **return** R_s

TWO-HYPERCUBE($Q[][], n, m, index$)
1 $R_s[r]$
2 **for** $i \leftarrow 0$ **to** $\sqrt[c]{K}$
3 **do** $\sigma \leftarrow$ SIGMA($Q[0], m, i, index$)
4 $parts \leftarrow$ SPLIT(σ, n)
5 $R_s \leftarrow$ FILTER($R_s, Q[1][i], parts, m$)
6 **return** R_s

SIGMA($Q[], m, i, index$)
1 $\sigma \leftarrow 1$
2 **for** $i \leftarrow 0$ **to** $length(Q)$
3 **do** $\sigma \leftarrow \sigma \cdot Q[i]^{db[index + d \cdot \sqrt[c]{K} + i \cdot (\sqrt[c]{K})^2]}$
4 **return** σ

SPLIT(σ, n)
1 $R_{parts}[0] \leftarrow \sigma/n$
2 $R_{parts}[1] \leftarrow \sigma \mod n$
3 **return** R_parts

FILTER$(R_s, Q, parts[], m)$
1 $R_s[0] = R_s[0] \cdot Q^{parts[0]} \mod m$
2 $R_s[1] = R_s[1] \cdot Q^{parts[1]} \mod m$
3 **return** R_s

C-SPLIT$(prevParts[], n)$
1 $R_{parts}[length(prevParts) * 2]$
2 **for** $i \leftarrow 0$ **to** $length(prevParts)$
3 **do** $R_{parts}[i * 2] \leftarrow prevParts[i]/n$
4 $R_{parts}[i * 2 + 1] \leftarrow prevParts[i] \mod n$
5 **return** R_{parts}

C-FILTER$(R_s[], Q[][], parts[], m, k, d)$
1 **for** $i \leftarrow 0$ **to** $length(parts)$
2 **do** $R_s[i] \leftarrow R_s[i] \cdot Q[c - d - 1][k]^{parts[i]} \mod m$
3 **return** R_s

A.2.4 Extract Result

Client-Side Invocation

CHANG-EXTRACT-RESULT$(R_s[2^{c-1}])$
1 $R_l \leftarrow R_s$
2 $R_c[length(R_l)/2]$
3 **while** $length(R_c) > 0$
4 **do for** $i \leftarrow 0$ **to** $length(R_c)$
5 **do** $R_c[i] \leftarrow$ DECRYPT$(R_l[i * 2]) \cdot n +$ DECRYPT$(R_l[i * 2 + 1])$
6 $R_l \leftarrow R_c$
7 $R_c[length(R_l)/2]$
8 **return** DECRYPT$(R_l[0])$

A.3 Lipmaa Protocol

See Section 4.4.2 for method headers.

A.3.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

GetDatabaseSize()
1 **return** K

GetNumberOfDimensions()
1 **return** c

GetDimensionLength()
1 **return** $\lceil \sqrt[c]{K} \rceil$

GetDataSize()
1 **return** $\lceil \log_2 \max\{x_i\}_{i=0}^{K-1} \rceil$

GetMaximumExponent()
1 **return** s

A.3.2 Generate Query

Client-Side Invocation

Lipmaa-Query($index$)
1 $Q[c, \sqrt[c]{K}]$
2 **for** $i \leftarrow 0$ **to** c
3 **do for** $j \leftarrow 0$ **to** $\sqrt[c]{K}$
4 **do if** $i * n + j = index$
5 **then** $Q[i, j] \leftarrow$ Encrypt$(1, i)$
6 **else** $Q[i, j] \leftarrow$ Encrypt$(0, i)$
7 **return** Q

A.3.3 Process Query Against Database

Server-side Invocation

Lipmaa-Perform-Retrieval($Q[][], n$)
1 **return** N-Hypercube$(Q, n, 0, 0)$

N-HYPERCUBE$(Q[][], n, d, index)$
1 **if** $(c - d) = 1$
2 **then** $R_s \leftarrow$ ONE-HYPERCUBE$(Q, index, n^2)$
3 **else** $R_s \leftarrow 1$
4 $m \leftarrow n^{s-d-1}$
5 **for** $i \leftarrow 0$ **to** $log_c K$
6 **do** $R_p \leftarrow$ N-HYPRERCUBE$(Q, n, d + 1, index, (i \cdot (log_c K)^d)$
7 $R_s \leftarrow R_s \cdot Q[c - d - 1]^{R_p} \mod m$
8 **return** R_s

ONE-HYPERCUBE$(Q[][], index, m)$
1 $R_s \leftarrow 1$
2 **for** $i \leftarrow 0$ **to** $log_c K$
3 **do** $R_s \leftarrow R_s \cdot Q[0][i]^{db[i \cdot (log_c K)^{c-1} + index]}$
4 **return** R_s

A.3.4 Extract Result

Client-Side Invocation

LIPMAA-EXTRACT-RESULT(R_s)
1 **for** $i \leftarrow c$ **to** 0
2 **do** $R_s \leftarrow$ DECRYPT(R_s, i)
3 **return** R_s

Appendix B

New PIR Implementation Details

B.1 Extended Chang Protocol

See Section 5.1.2 for method headers.

B.1.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

See Appendix A.2 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, and GetDataSize

GetNumberOfParts()
1 **return** s

GetMaximumExponent()
1 **return** s

B.1.2 Generate Query

Client-Side Invocation

See Appendix A.2 for Chang-Query.

B.1.3 Process Query Against Database

Server-side Invocation

CHANG-PERFORM-RETRIEVAL($Q[][], n$)
1 **return** C-HYPERCUBE($Q, n, n^{s+1}, 0, 0$)

See Appendix A.2 for C-HYPERCUBE, TWO-HYPERCUBE, SIGMA, FILTER, and C-FILTER and override the C-SPLIT and SPLIT with the following.

C-SPLIT($prevParts, n$)
1 $R_{parts}[length(prevParts) * (s+1)]$
2 **for** $i \leftarrow 0$ **to** $length(prevParts)$
3 **do for** $j \leftarrow 0$ **to** $s+1$
4 **do** $R_{parts}[i \cdot (s+1) + j] \leftarrow prevParts[i] \cdot (n^{s-j})^{-1} \mod n$
5 **return** R_{parts}

SPLIT(σ, n)
1 $R_{parts}[s+1]$
2 **for** $i \leftarrow 0$ **to** $s+1$
3 **do** $R_{parts}[i] \leftarrow \sigma / n^{s-i}$
4 $\sigma \leftarrow \sigma \mod n^{s-i}$
5 **return** R_{parts}

B.1.4 Extract Result

Client-Side Invocation

CHANG-EXTRACT-RESULT($R_s[2^{c-1}]$)
1 $R_l \leftarrow R_s$
2 $R_c[length(R_l)/2]$
3 **while** $length(R_c) > 0$
4 **do for** $i \leftarrow 0$ **to** $length(R_c)$
5 **do for** $j \leftarrow 0$ **to** s
6 **do** $R_c[i] = R_c[i] +$ DECRYPT($R_l[i * (s+1) + j] \cdot n^{s-j}, s$)
7 $R_l \leftarrow R_c$
8 $R_c[length(R_l)/2]$
9 **return** DECRYPT($R_l[0], s$)

B.2 Generalized Chang Protocol

See Section 5.2.2 for method headers.

B.2.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

See Appendix A.2 for GetDatabaseSize, GetNumberOfDimensions, GetDimensionLength, and GetDataSize

GetNumberOfParts()
1 **return** r

See Appendix B.1 for GetMaximumExponent.

B.2.2 Generate Query

Client-Side Invocation

See Appendix A.2 for Chang-Query.

B.2.3 Process Query Against Database

Server-side Invocation

See Appendix B.1 for Chang-Perform-Retrieval and override C-Split and Split with the following.

C-Split($prevParts, n$)
1 $R_{parts}[length(prevParts) * (r)]$
2 **for** $i \leftarrow 0$ **to** $length(prevParts)$
3 **do for** $j \leftarrow 0$ **to** r
4 **do** $R_{parts}[i \cdot r + j] \leftarrow prevParts[i] \cdot (n^{s-j})^{-1} \mod n$
5 **return** R_{parts}

Split(σ, n)
1 $R_{parts}[r]$
2 **for** $i \leftarrow 0$ **to** r
3 **do** $R_{parts}[i] \leftarrow \sigma / n^{s-i-1}$
4 $\sigma \leftarrow \sigma \mod n^{s-i-1}$
5 **return** R_{parts}

B.2.4 Extract Result

Client-Side Invocation

CHANG-EXTRACT-RESULT($R_s[2^{c-1}]$)
1 $R_l \leftarrow R_s$
2 $R_c[length(R_l)/2]$
3 **while** $length(R_c) > 0$
4 **do for** $i \leftarrow 0$ **to** $length(R_c)$
5 **do for** $j \leftarrow 0$ **to** r
6 **do** $R_c[i] \leftarrow R_c[i] + \text{DECRYPT}(R_l[i * (r) + j] \cdot n^{s-j}, s)$
7 $R_l \leftarrow R_c$
8 $R_c[length(R_l)/2]$
9 **return** DECRYPT($R_l[0], s$)

B.3 Chang Protocol with Pre-Splitting Elements

See Section 5.3.2 for method headers.

B.3.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

See Appendix A.2 for GETDATABASESIZE, GETNUMBEROFDIMENSIONS, GETDIMENSIONLENGTH, GETNUMBEROFPARTS and GETDATASIZE

B.3.2 Generate Query

Client-Side Invocation

See Appendix A.2 for CHANG-QUERY.

B.3.3 Process Query Against Database

Server-side Invocation

See Appendix A.2 for CHANG-PERFORM-RETRIEVAL and override SIGMA, SPLIT and FILTER with the following.

SIGMA($Q[], m, i, index$)
1 $\sigma[] \leftarrow 1, 1$
2 **for** $i \leftarrow 0$ **to** $length(Q)$
3 **do** $\sigma[0] \leftarrow \sigma[0] \cdot Q[i]^{\frac{db[index+d\cdot\lceil\sqrt[c]{K}\rceil+i\cdot(\lceil\sqrt[c]{K}\rceil)^2]}{n}}$
4 $\sigma[1] \leftarrow \sigma[1] \cdot Q[i]^{db[index+d\cdot\lceil\sqrt[c]{K}\rceil+i\cdot(\lceil\sqrt[c]{K}\rceil)^2]} \mod n$
5 **return** σ

SPLIT(σ, n)
1 $R_{parts}[0] \leftarrow \sigma[0]/n$
2 $R_{parts}[1] \leftarrow \sigma[0] \mod n$
3 $R_{parts}[2] \leftarrow \sigma[1]/n$
4 $R_{parts}[3] \leftarrow \sigma[1] \mod n$
5 **return** R_{parts}

FILTER$(R_s, Q, parts[], m)$
1 $R_s[0] = R_s[0] \cdot Q^{parts[0]} \mod m$
2 $R_s[1] = R_s[1] \cdot Q^{parts[1]} \mod m$
3 $R_s[2] = R_s[2] \cdot Q^{parts[2]} \mod m$
4 $R_s[3] = R_s[3] \cdot Q^{parts[3]} \mod m$**return** R_s

B.3.4 Extract Result

Client-Side Invocation

CHANG-EXTRACT-RESULT$(R_s[2^c])$
1 $R_l \leftarrow R_s$
2 $R_c[length(R_l)/2]$
3 **while** $length(R_c) > 0$
4 **do for** $i \leftarrow 0$ **to** $length(R_c)$
5 **do for** $j \leftarrow 0$ **to** $numSplits$
6 **do** $R_c[i] \leftarrow R_c[i] +$ DECRYPT$(R_l[i * (numSplits) + j] \cdot n^{s-j}, s)$
7 $R_l \leftarrow R_c$
8 $R_c[length(R_l)/2]$
9 **return** $R_l[0]$

B.4 A Collaboration of the Chang and LipmaaProtocols

See Section 5.4.2 for method headers.

B.4.1 Acquire Server Parameters

Client-Side Remote Method Invocations on Server-Side Stub

See Appendix A.2 for GETDATABASESIZE, GETNUMBEROFDIMENSIONS, GETDIMENSIONLENGTH, GETNUMBEROFPARTS and GETDATASIZE

See Appendix A.3 for GETMAXIMUMEXPONENT.

B.4.2 Generate Query

Client-Side Invocation

COLLABORATION-QUERY($index$)
1 $Q[c, \sqrt[c]{K}]$
2 **for** $i \leftarrow 0$ **to** c
3 **do for** $j \leftarrow 0$ **to** $\sqrt[c]{K}$
4 **do if** $i * n + j = index$
5 **then** $Q[i,j] \leftarrow$ ENCRYPT$(1, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$
6 **else** $Q[i,j] \leftarrow$ ENCRYPT$(0, \lceil \frac{s-1}{2^{c-i-1}} \rceil)$
7 **return** Q

B.4.3 Process Query Against Database

Server-side Invocation

See Appendix A.2 for CHANG-PERFORM-RETRIEVAL and override C-HYPERCUBE, TWO-HYPERCUBE with the following.

C-HYPERCUBE($Q[][], n, m, d, index$)
1 $R_s[numSplits^{c-d-1}$
2 $t \leftarrow \lceil \frac{s-1}{2^{c-dim+1}} \rceil$
3 **if** $c - d = 2$
4 **then** $R_s \leftarrow$ TWO-HYPERCUBE$(Q, n, m, d+1, index \cdot (\sqrt[c]{K})^d)$
5 **else** **for** $k \leftarrow 0$ **to** $\sqrt[c]{K}$
6 **do** $prevParts \leftarrow N - Hypercube(Q, n, n^{2t}, d+1, index \cdot (\sqrt[c]{K})^d)$
7 $currParts \leftarrow C - Split(prevParts, n^t)$
8 $R_s \leftarrow C - Filter(R_s, Q, currParts, n^{t+1}, k, d)$
9 **return** R_s

TWO-HYPERCUBE($Q[][], n, m, index$)
1 $R_s[numSplits]$
2 **for** $i \leftarrow 0$ **to** $\sqrt[c]{K}$
3 **do** $\sigma \leftarrow$ SIGMA$(Q[0], n^{s+1}, i, index)$
4 $parts \leftarrow$ SPLIT$(\sigma, n^{\frac{s+1}{2}})$
5 $R_s \leftarrow$ FILTER$(R_s, Q[1][i], partsn^{\frac{s+1}{2}+1})$
6 **return** R_s

B.4.4 Extract Result

Client-Side Invocation

COLLABORATION-EXTRACT-RESULT($R_s[2^{c-1}]$)
1 $R_l \leftarrow R_s$
2 $R_c[length(R_l)/2]$
3 **while** $length(R_c) > 0$
4 **do** $s \leftarrow \lceil \frac{s-1}{length(R_l)} \rceil + 1$
5 **for** $i \leftarrow 0$ **to** $length(R_c)$
6 **do** $R_c[i] \leftarrow$ DECRYPT$(R_l[i * 2], s) \cdot n +$ DECRYPT$(R_l[i * 2 + 1], s)$
7 $R_l \leftarrow R_c$
8 $R_c[length(R_l)/2]$
9 **return** DECRYPT$(R_l[0])$

Appendix C

Implementation Documentation

C.1 Package edu.rit.cryptosystems

Package Contents *Page*

Classes

Cryptosystem 93
Generic definition of a Cryptosystem.

DamgardJurikCryptosystem 94
Damgard-Jurik Cryptosystem was introduced in 2003 in Jurik's doctoral thesis.

DamgardJurikKeyGen 96
A key generator should create a single private and public key pair.

DamgardJurikPrivateKey 97
The definition of a private key to be used with a Damgard-Jurik cryptosystem.

DamgardJurikPublicKey 98
The definition of a public key to be used with a Damgard-Jurik cryptosystem.

GoldwasserMicaliCryptosystem 100
The Goldwasser-Micali cryptosystem was presented in the paper known for introducing the idea of probabilistic public-key encryption.

GoldwasserMicaliKeyGen 101
A key generator should create a single private and public key pair.

GoldwasserMicaliPrivateKey 102
The definition of a private key to be used with a Goldwasser-Micali cryptosystem.

GoldwasserMicaliPublicKey 103
The definition of a public key to be used with a Goldwasser-Micali cryptosystem.

KeyGen 104
A key generator should create a single private and public key pair.

NaccacheSternCryptosystem 106
Can be easily seen as an extension of Benaloh's cryptosystem.

NaccacheSternKeyGen 107
A key generator should create a single private and public key pair.

NaccacheSternPrivateKey 109
The definition of a private key to be used with a Naccache-Stern cryptosystem.

NaccacheSternPublicKey 110
The definition of a public key to be used with a Naccache-Stern cryptosystem.

PaillierCryptosystem 111
Paillier's Cryptosystem came out around 1999, and relies upon the Composite Residuosity Assumption.

PaillierKeyGen 113
A key generator should create a single private and public key pair.

PaillierPrivateKey 114
The definition of a private key to be used with a Paillier cryptosystem.

PaillierPublicKey 115
The definition of a public key to be used with a Paillier cryptosystem.

PrivateKey 116
The minimal definition of a private key for an RSA-composite based cryptosystem.

PublicKey 118

The minimal definition of a public key for an RSA-composite based cryptosystem.

A library of homomorphic cryptosystem implementations. Each cryptosystem has a cryptosystem which encrypts and decrypts, a key generated that genrates a private and public key pair for a give key size, and the private and public keys.

C.1.1 Class Cryptosystem

Generic definition of a Cryptosystem. A cryptosystem traditionally consists of key generation, the encyption, and the decryption algorithms, but the key generation has been seperated in this implementation.

Declaration

public abstract class Cryptosystem
extends java.lang.Object

All known subclasses

PaillierCryptosystem (in C.1.15, page 111), NaccacheSternCryptosystem (in C.1.11, page 106), GoldwasserMicaliCryptosystem (in C.1.6, page 100), DamgardJurikCryptosystem (in C.1.2, page 94)

Constructor summary

Cryptosystem()

Method summary

decrypt(BigInteger, PublicKey, PrivateKey) Decrypts a ciphertext given a public key and a private key.
encrypt(BigInteger, PublicKey) Encrpyts a plaintext given a public key.
random(BigInteger) Acquires a random integer value between 1 and max.

Constructors

- **Cryptosystem**
  `public` **Cryptosystem**`( )`

Methods

- **decrypt**
  `public abstract java.math.BigInteger` **decrypt**`(`
  `java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`

  - **Description**
    Decrypts a ciphertext given a public key and a private key.
  - **Parameters**
    * `c` – ciphertext
    * `pub` – the public key
    * `pri` – the corresponding private key
  - **Returns** – the plaintext

- **encrypt**
  `public abstract java.math.BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
  - **Description**
    Encrpyts a plaintext given a public key.
  - **Parameters**
    * `x` – plaintext message
    * `pub` – the public key for the corresponding system
  - **Returns** – the ciphertext

- **random**
  `protected java.math.BigInteger` **random**`( java.math.BigInteger` **max** `)`
  - **Description**
    Acquires a random integer value between 1 and max.
  - **Parameters**
    * `max` – the maximum value
  - **Returns** – a random value between 1 and max

C.1.2 Class DamgardJurikCryptosystem

Damgard-Jurik Cryptosystem was introduced in 2003 in Jurik's doctoral thesis. The cryptosystem is an extension of Paillier's Cryptosystem, introduced in 1999. The computational assumption that the security of this cryptosystem relies on is also Composite Residuosity Assumption.

Declaration

public class DamgardJurikCryptosystem
extends edu.rit.cryptosystems.PaillierCryptosystem (in C.1.15, page 111)

Constructor summary

DamgardJurikCryptosystem()

Method summary

decrypt(BigInteger, PublicKey, PrivateKey) Decrypts a ciphertext given a public key and a private key.
encrypt(BigInteger, PublicKey) Encrpyts a plaintext given a public key.
getLofS(BigInteger, BigInteger, int) Basically retrieves the discrete log of (n+1)∧x in base n∧s.

Constructors

- **DamgardJurikCryptosystem**
  `public` **DamgardJurikCryptosystem**`( )`

Methods

- **decrypt**
  `public java.math.BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
  - **Description**
    Decrypts a ciphertext given a public key and a private key.
  - **Parameters**
    - `c` – ciphertext
    - `pub` – the public key
    - `pri` – the corresponding private key
  - **Returns** – the plaintext
- **encrypt**
  `public java.math.BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
  - **Description**
    Encrpyts a plaintext given a public key.
  - **Parameters**
    - `x` – plaintext message
    - `pub` – the public key for the corresponding system
  - **Returns** – the ciphertext
- **getLofS**
  `public java.math.BigInteger` **getLofS**`( java.math.BigInteger` **a**`, java.math.BigInteger` **n**`, int` **s** `)`
  - **Description**
    Basically retrieves the discrete log of (n+1)∧x in base n∧s.

– **Parameters**

* `a` – the number which we are looking for the discrete log of
* `n` – the RSA composite
* `s` – the block factor

Members inherited from class `edu.rit.cryptosystems.PaillierCryptosystem` (in C.1.15, page 111)

- `public BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
- `public BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
- `protected BigInteger` **L**`( java.math.BigInteger` **u**`, java.math.BigInteger` **n** `)`

Members inherited from class `edu.rit.cryptosystems.Cryptosystem` (in C.1.1, page 93)

- `public abstract BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
- `public abstract BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
- `protected BigInteger` **random**`( java.math.BigInteger` **max** `)`

C.1.3 Class DamgardJurikKeyGen

A key generator should create a single private and public key pair. This corresponds to the Damgard-Jurik cryptosystem.

Declaration

public class DamgardJurikKeyGen
extends edu.rit.cryptosystems.PaillierKeyGen (in C.1.16, page 113)

Constructor summary

DamgardJurikKeyGen() Creates a key generator.

Method summary

generateKey(int)

Constructors

- **DamgardJurikKeyGen**
  `public` **DamgardJurikKeyGen**`( )`

  – **Description**
  Creates a key generator. Default s is 1.

Methods

- **generateKey**
  `public abstract void` **generateKey**`( int keySize )`
  - **Description copied from KeyGen** (**in C.1.10, page 104**)
    Generates a public key and private key pair.
  - **Parameters**
    * `keySize` – the number of bits in the key

Members inherited from class `edu.rit.cryptosystems.PaillierKeyGen` (in C.1.16, page 113)

- `public void` **generateKey**`( int keySize )`

Members inherited from class `edu.rit.cryptosystems.KeyGen` (in C.1.10, page 104)

- `protected BigInteger` **createPandQ**`( int keySize )`
- `public abstract void` **generateKey**`( int keySize )`
- `public PrivateKey` **getPrivateKey**`( )`
- `public PublicKey` **getPublicKey**`( )`
- `protected static final` **P**
- `protected` **pri**
- `protected` **pub**
- `protected static final` **Q**

C.1.4 Class DamgardJurikPrivateKey

The definition of a private key to be used with a Damgard-Jurik cryptosystem.

Declaration

public class DamgardJurikPrivateKey
extends edu.rit.cryptosystems.PaillierPrivateKey (in C.1.17, page 114)

Constructor summary

DamgardJurikPrivateKey(BigInteger, BigInteger) The private key for a Damgard-Jurik cryptosystem.

Method summary

getCryptosystem()

Constructors

- **DamgardJurikPrivateKey**
  `public` **DamgardJurikPrivateKey**`( java.math.BigInteger` **p**`, java.math.BigInteger` **q** `)`
  - **Description**
    The private key for a Damgard-Jurik cryptosystem.
  - **Parameters**
    - `p` – a prime
    - `q` – a prime with equal length

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
  - **Description copied from PrivateKey** (**in C.1.19, page 116**)
    Retrieves an instance of the proper cryptosystem to work with this key.
  - **Returns** – an instance of the appropriate cryptosystem

Members inherited from class `edu.rit.cryptosystems.PaillierPrivateKey` (in C.1.17, page 114)

- `public Cryptosystem` **getCryptosystem**`( )`
- `public BigInteger` **getLambda**`( )`

Members inherited from class `edu.rit.cryptosystems.PrivateKey` (in C.1.19, page 116)

- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public BigInteger` **getP**`( )`
- `public BigInteger` **getQ**`( )`

C.1.5 Class DamgardJurikPublicKey

The definition of a public key to be used with a Damgard-Jurik cryptosystem.

Declaration

public class DamgardJurikPublicKey
extends edu.rit.cryptosystems.PaillierPublicKey (in C.1.18, page 115)

Constructor summary

DamgardJurikPublicKey(BigInteger, BigInteger, int) The public key for a Damgard-Jurik cryptosystem.

Method summary

getCryptosystem()
setExp(int) Sets the exponent.

Constructors

- **DamgardJurikPublicKey**
  `public` **DamgardJurikPublicKey**`( java.math.BigInteger n, java.math.BigInteger y, int s )`
  - **Description**
    The public key for a Damgard-Jurik cryptosystem.
  - **Parameters**
    - `n` – an RSA composite
    - `y` – a wisely chosen constant
    - `s` – the block size

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
- **setExp**
  `public void` **setExp**`( int exp )`
  - **Description**
    Sets the exponent. Since the exponent can chang so that ecrypted values can be encrypted repeated to produce the original plaintext after several equal numbered decryptions, this is present here.
  - **Parameters**
    - `exp` – the exponent

Members inherited from class `edu.rit.cryptosystems.PaillierPublicKey` (in C.1.18, page 115)

- `public Cryptosystem` **getCryptosystem**`( )`

Members inherited from class `edu.rit.cryptosystems.PublicKey` (in C.1.20, page 118)

- `protected` **e**
- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public int` **getExp**`( )`
- `public BigInteger` **getN**`( )`
- `public BigInteger` **getY**`( )`

C.1.6 Class GoldwasserMicaliCryptosystem

The Goldwasser-Micali cryptosystem was presented in the paper known for introducing the idea of probabilistic public-key encryption. The advantage of public-key cryptosystems are that strings of bits can be encrypted bit-by-bit without loss of semantic security.

Declaration

public class GoldwasserMicaliCryptosystem
extends edu.rit.cryptosystems.Cryptosystem (in C.1.1, page 93)

Constructor summary

GoldwasserMicaliCryptosystem()

Method summary

decrypt(BigInteger, PublicKey, PrivateKey) Decrypts a ciphertext given a public key and a private key.
encrypt(BigInteger, PublicKey) Encrpyts a plaintext given a public key.
jacobi(BigInteger, BigInteger) The jacobi symbol, and extension of the legendre symbol.

Constructors

- **GoldwasserMicaliCryptosystem**
  `public` **GoldwasserMicaliCryptosystem**`( )`

Methods

- **decrypt**
  `public java.math.BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
  - **Description**
    Decrypts a ciphertext given a public key and a private key.
  - **Parameters**
    - `c` – ciphertext
    - `pub` – the public key
    - `pri` – the corresponding private key
  - **Returns** – the plaintext
- **encrypt**
  `public java.math.BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`

- **Description**
  Encrpyts a plaintext given a public key.
- **Parameters**
  - `x` – plaintext message
  - `pub` – the public key for the corresponding system
- **Returns** – the ciphertext

- **jacobi**
  `public int` **jacobi**`( java.math.BigInteger` **a**`, java.math.BigInteger` **n** `)`
  - **Description**
    The jacobi symbol, and extension of the legendre symbol. ALgorithm borrwoed from "The Handbook of Applied Cryptography" algoritm 2.149.
  - **Parameters**
    - `a` – the "numerator"
    - `n` – the "denominator"
  - **Returns** – the jacobi symbol of a/n

Members inherited from class `edu.rit.cryptosystems.Cryptosystem` (in C.1.1, page 93)

- `public abstract BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
- `public abstract BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
- `protected BigInteger` **random**`( java.math.BigInteger` **max** `)`

C.1.7 Class GoldwasserMicaliKeyGen

A key generator should create a single private and public key pair. This corresponds to the Goldwasser-Micali cryptosystem.

Declaration

public class GoldwasserMicaliKeyGen
extends edu.rit.cryptosystems.KeyGen (in C.1.10, page 104)

Constructor summary

GoldwasserMicaliKeyGen() Creates a key generator.

Method summary

generateKey(int)

Constructors

- **GoldwasserMicaliKeyGen**
  `public` **GoldwasserMicaliKeyGen**`( )`
  - **Description**
    Creates a key generator.

Methods

- **generateKey**
  `public abstract void` **generateKey**`( int` **keySize** `)`
  - **Description copied from KeyGen** (**in C.1.10, page 104**)
    Generates a public key and private key pair.
  - **Parameters**
    * `keySize` – the number of bits in the key

Members inherited from class `edu.rit.cryptosystems.KeyGen` (in C.1.10, page 104)

- `protected BigInteger` **createPandQ**`( int` **keySize** `)`
- `public abstract void` **generateKey**`( int` **keySize** `)`
- `public PrivateKey` **getPrivateKey**`( )`
- `public PublicKey` **getPublicKey**`( )`
- `protected static final` **P**
- `protected` **pri**
- `protected` **pub**
- `protected static final` **Q**

C.1.8 Class GoldwasserMicaliPrivateKey

The definition of a private key to be used with a Goldwasser-Micali cryptosystem.

Declaration

public class GoldwasserMicaliPrivateKey
extends edu.rit.cryptosystems.PrivateKey (in C.1.19, page 116)

Constructor summary

GoldwasserMicaliPrivateKey(BigInteger, BigInteger) The private key for a Goldwasser-Micali cryptosystem.

Method summary

getCryptosystem()

Constructors

- **GoldwasserMicaliPrivateKey**
  `public` **GoldwasserMicaliPrivateKey**`( java.math.BigInteger p, java.math.BigInteger q )`
  - **Description**
    The private key for a Goldwasser-Micali cryptosystem.
  - **Parameters**
    - `p` – a prime
    - `q` – a prime with equal length

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
  - **Description copied from PrivateKey** (**in C.1.19, page 116**)
    Retrieves an instance of the proper cryptosystem to work with this key.
  - **Returns** – an instance of the appropriate cryptosystem

Members inherited from class `edu.rit.cryptosystems.PrivateKey` (in C.1.19, page 116)

- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public BigInteger` **getP**`( )`
- `public BigInteger` **getQ**`( )`

C.1.9 Class GoldwasserMicaliPublicKey

The definition of a public key to be used with a Goldwasser-Micali cryptosystem.

Declaration

public class GoldwasserMicaliPublicKey
extends edu.rit.cryptosystems.PublicKey (in C.1.20, page 118)

Constructor summary

GoldwasserMicaliPublicKey(BigInteger, BigInteger) The public key for a Goldwasser-Micali cryptosystem.

Method summary

getCryptosystem()

Constructors

- **GoldwasserMicaliPublicKey**
  `public` **GoldwasserMicaliPublicKey**`( java.math.BigInteger` **n**`, java.math.BigInteger` **y** `)`
  - **Description**
    The public key for a Goldwasser-Micali cryptosystem.
  - **Parameters**
    - `n` – an RSA composite
    - `y` – a wisely chosen constant

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`

Members inherited from class `edu.rit.cryptosystems.PublicKey` (in C.1.20, page 118)

- `protected` **e**
- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public int` **getExp**`( )`
- `public BigInteger` **getN**`( )`
- `public BigInteger` **getY**`( )`

C.1.10 Class KeyGen

A key generator should create a single private and public key pair.

Declaration

public abstract class KeyGen
extends java.lang.Object

All known subclasses

PaillierKeyGen (in C.1.16, page 113), NaccacheSternKeyGen (in C.1.12, page 107), GoldwasserMicaliKeyGen (in C.1.7, page 101), DamgardJurikKeyGen (in C.1.3, page 96)

Field summary

P
pri
pub
Q

Constructor summary

KeyGen()

Method summary

createPandQ(int) Generates p and q of equal length such that the length of p is keySize/2.
generateKey(int) Generates a public key and private key pair.
getPrivateKey() Returns the private key.
getPublicKey() Returns the public key.

Fields

- protected static final int **P**
- protected static final int **Q**
- protected PrivateKey **pri**
- protected PublicKey **pub**

Constructors

- **KeyGen**
  `public KeyGen( )`

Methods

- **createPandQ**
  `protected java.math.BigInteger[] createPandQ( int keySize )`
  - **Description**
    Generates p and q of equal length such that the length of p is keySize/2.
  - **Parameters**
    - `keySize` – the size of n given n=pq
  - **Returns** – teo primes, p and q
- **generateKey**
  `public abstract void generateKey( int keySize )`
  - **Description**
    Generates a public key and private key pair.
  - **Parameters**
    - `keySize` – the number of bits in the key

- **getPrivateKey**
  `public PrivateKey` **getPrivateKey**`( )`
  - **Description**
    Returns the private key.
  - **Returns** – private key

- **getPublicKey**
  `public PublicKey` **getPublicKey**`( )`
  - **Description**
    Returns the public key.
  - **Returns** – public key

C.1.11 Class NaccacheSternCryptosystem

Can be easily seen as an extension of Benaloh's cryptosystem. The Naccache-Stern cryptosystem came about in 1998 and is based on the Higher Residuosity Assumption as opposed to the Prime Residuosity Assumpiton which the Benaloh Cryptosystem relies on.

Declaration

public class NaccacheSternCryptosystem
extends edu.rit.cryptosystems.Cryptosystem (in C.1.1, page 93)

Constructor summary

NaccacheSternCryptosystem()

Method summary

decrypt(BigInteger, PublicKey, PrivateKey) Decrypts a ciphertext given a public key and a private key.
encrypt(BigInteger, PublicKey) Encrpyts a plaintext given a public key.

Constructors

- **NaccacheSternCryptosystem**
  `public` **NaccacheSternCryptosystem**`( )`

Methods

- **decrypt**
  `public java.math.BigInteger` **`decrypt(`** `java.math.BigInteger` **`c`**`, PublicKey` **`pub`**`, PrivateKey` **`pri`** **`)`**
  - **Description**
    Decrypts a ciphertext given a public key and a private key.
  - **Parameters**
    - `c` – ciphertext
    - `pub` – the public key
    - `pri` – the corresponding private key
  - **Returns** – the plaintext
- **encrypt**
  `public java.math.BigInteger` **`encrypt(`** `java.math.BigInteger` **`x`**`, PublicKey` **`pub`** **`)`**
  - **Description**
    Encrpyts a plaintext given a public key.
  - **Parameters**
    - `x` – plaintext message
    - `pub` – the public key for the corresponding system
  - **Returns** – the ciphertext

Members inherited from class `edu.rit.cryptosystems.Cryptosystem` (in C.1.1, page 93)

- `public abstract BigInteger` **`decrypt(`** `java.math.BigInteger` **`c`**`, PublicKey` **`pub`**`, PrivateKey` **`pri`** **`)`**
- `public abstract BigInteger` **`encrypt(`** `java.math.BigInteger` **`x`**`, PublicKey` **`pub`** **`)`**
- `protected BigInteger` **`random(`** `java.math.BigInteger` **`max`** **`)`**

C.1.12 Class NaccacheSternKeyGen

A key generator should create a single private and public key pair. This corresponds to the Naccache-Stern cryptosystem.

Declaration

public class NaccacheSternKeyGen
extends edu.rit.cryptosystems.KeyGen (in C.1.10, page 104)

Field summary

DEFAULT_K

Constructor summary

NaccacheSternKeyGen() Generates public and private key pair for a Naccache-Stern cryptosystem.

Method summary

generateKey(int)
setNumPrimes(int) Sets the number of primes that comprises the public key N.

Fields

- public static final int **DEFAULT_K**

Constructors

- **NaccacheSternKeyGen**
  `public` **NaccacheSternKeyGen**`( )`
  - **Description**
    Generates public and private key pair for a Naccache-Stern cryptosystem.

Methods

- **generateKey**
  `public abstract void` **generateKey**`( int keySize )`
  - **Description copied from KeyGen** (**in C.1.10, page 104**)
    Generates a public key and private key pair.
  - **Parameters**
    * `keySize` – the number of bits in the key
- **setNumPrimes**
  `public void` **setNumPrimes**`( int numPrimes )`
  - **Description**
    Sets the number of primes that comprises the public key N.
  - **Parameters**
    * `numPrimes` – number of primes in N

Members inherited from class `edu.rit.cryptosystems.KeyGen` (in C.1.10, page 104)

- `protected BigInteger` **createPandQ**`( int` **keySize** `)`
- `public abstract void` **generateKey**`( int` **keySize** `)`
- `public PrivateKey` **getPrivateKey**`( )`
- `public PublicKey` **getPublicKey**`( )`
- `protected static final` **P**
- `protected` **pri**
- `protected` **pub**
- `protected static final` **Q**

C.1.13 Class NaccacheSternPrivateKey

The definition of a private key to be used with a Naccache-Stern cryptosystem.

Declaration

public class NaccacheSternPrivateKey
extends edu.rit.cryptosystems.PrivateKey (in C.1.19, page 116)

Constructor summary

NaccacheSternPrivateKey(BigInteger, BigInteger, BigInteger, BigInteger[]) The private key for a Naccache-Stern cryptosystem.

Method summary

getCryptosystem()
getPhi() Euler-phi function of n
getPs() A family of primes

Constructors

- **NaccacheSternPrivateKey**
  `public` **NaccacheSternPrivateKey**`( java.math.BigInteger` **p**`, java.math.BigInteger` **q**`, java.math.BigInteger` **phi**`, java.math.BigInteger[]` **ps** `)`
  - **Description**
    The private key for a Naccache-Stern cryptosystem.
  - **Parameters**
    - `p` – a prime
    - `q` – a prime with equal length
    - `phi` – Euler-phi function of n
    - `ps` – the family of primes

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
  - **Description copied from PrivateKey** (**in C.1.19, page 116**)
    Retrieves an instance of the proper cryptosystem to work with this key.
  - **Returns** – an instance of the appropriate cryptosystem
- **getPhi**
  `public java.math.BigInteger` **getPhi**`( )`
  - **Description**
    Euler-phi function of n
  - **Returns** – the number of relatively prime numbers below n
- **getPs**
  `public java.math.BigInteger[]` **getPs**`( )`
  - **Description**
    A family of primes
  - **Returns** – a family of primes

Members inherited from class `edu.rit.cryptosystems.PrivateKey` (in C.1.19, page 116)

- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public BigInteger` **getP**`( )`
- `public BigInteger` **getQ**`( )`

C.1.14 Class NaccacheSternPublicKey

The definition of a public key to be used with a Naccache-Stern cryptosystem.

Declaration

public class NaccacheSternPublicKey
extends edu.rit.cryptosystems.PublicKey (in C.1.20, page 118)

Constructor summary

NaccacheSternPublicKey(BigInteger, BigInteger, BigInteger) The public key for a Naccache-Stern cryptosystem.

Method summary

getCryptosystem()
getSigma() The wisely chosen constant sigma

Constructors

- **NaccacheSternPublicKey**
  `public` **NaccacheSternPublicKey**`( java.math.BigInteger n, java.math.BigInteger y, java.math.BigInteger sig )`
  - **Description**
    The public key for a Naccache-Stern cryptosystem.
  - **Parameters**
    - `n` – an RSA composite
    - `y` – a wisely chosen constant
    - `sig` – a wisely chosen constant

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
- **getSigma**
  `public java.math.BigInteger` **getSigma**`( )`
  - **Description**
    The wisely chosen constant sigma
  - **Returns** – the wisely chosen constant sigma

Members inherited from class `edu.rit.cryptosystems.PublicKey` (in C.1.20, page 118)

- `protected` **e**
- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public int` **getExp**`( )`
- `public BigInteger` **getN**`( )`
- `public BigInteger` **getY**`( )`

C.1.15 Class PaillierCryptosystem

Paillier's Cryptosystem came out around 1999, and relies upon the Composite Residuosity Assumption.

Declaration

public class PaillierCryptosystem
extends edu.rit.cryptosystems.Cryptosystem (in C.1.1, page 93)

All known subclasses

DamgardJurikCryptosystem (in C.1.2, page 94)

Constructor summary

PaillierCryptosystem()

Method summary

decrypt(BigInteger, PublicKey, PrivateKey) Decrypts a ciphertext given a public key and a private key.
encrypt(BigInteger, PublicKey) Encrpyts a plaintext given a public key.
L(BigInteger, BigInteger) L(u) = (u-1)/n

Constructors

- **PaillierCryptosystem**
  `public` **PaillierCryptosystem**`( )`

Methods

- **decrypt**
  `public java.math.BigInteger` **decrypt**`( java.math.BigInteger` **c**`, PublicKey` **pub**`, PrivateKey` **pri** `)`
  - **Description**
    Decrypts a ciphertext given a public key and a private key.
  - **Parameters**
    - `c` – ciphertext
    - `pub` – the public key
    - `pri` – the corresponding private key
  - **Returns** – the plaintext
- **encrypt**
  `public java.math.BigInteger` **encrypt**`( java.math.BigInteger` **x**`, PublicKey` **pub** `)`
  - **Description**
    Encrpyts a plaintext given a public key.
  - **Parameters**
    - `x` – plaintext message
    - `pub` – the public key for the corresponding system
  - **Returns** – the ciphertext
- **L**
  `protected java.math.BigInteger` **L**`( java.math.BigInteger` **u**`, java.math.BigInteger` **n** `)`

- **Description**
  L(u) = (u-1)/n
- **Parameters**
  - `u` – the parameter of this function
  - `n` – the cryptosystem modulus
- **Returns** – L(u) at n

Members inherited from class `edu.rit.cryptosystems.Cryptosystem` (in C.1.1, page 93)

- `public abstract BigInteger` **`decrypt`**`( java.math.BigInteger` **`c`**`, PublicKey` **`pub`**`, PrivateKey` **`pri`** `)`
- `public abstract BigInteger` **`encrypt`**`( java.math.BigInteger` **`x`**`, PublicKey` **`pub`** `)`
- `protected BigInteger` **`random`**`( java.math.BigInteger` **`max`** `)`

C.1.16 Class PaillierKeyGen

A key generator should create a single private and public key pair. This corresponds to the Paillier cryptosystem.

Declaration

public class PaillierKeyGen
extends edu.rit.cryptosystems.KeyGen (in C.1.10, page 104)

All known subclasses

DamgardJurikKeyGen (in C.1.3, page 96)

Constructor summary

PaillierKeyGen() Creates a key generator.

Method summary

generateKey(int)

Constructors

- **PaillierKeyGen**
  `public` **`PaillierKeyGen`**`( )`
  - **Description**
    Creates a key generator.

Methods

- **generateKey**
  `public abstract void` **generateKey**`( int` **keySize** `)`
  - **Description copied from KeyGen** (**in C.1.10, page 104**)
    Generates a public key and private key pair.
  - **Parameters**
    * `keySize` – the number of bits in the key

Members inherited from class `edu.rit.cryptosystems.KeyGen` (in C.1.10, page 104)

- `protected BigInteger` **createPandQ**`( int` **keySize** `)`
- `public abstract void` **generateKey**`( int` **keySize** `)`
- `public PrivateKey` **getPrivateKey**`( )`
- `public PublicKey` **getPublicKey**`( )`
- `protected static final` **P**
- `protected` **pri**
- `protected` **pub**
- `protected static final` **Q**

C.1.17 Class PaillierPrivateKey

The definition of a private key to be used with a Paillier cryptosystem.

Declaration

public class PaillierPrivateKey
extends edu.rit.cryptosystems.PrivateKey (in C.1.19, page 116)

All known subclasses

DamgardJurikPrivateKey (in C.1.4, page 97)

Constructor summary

PaillierPrivateKey(BigInteger, BigInteger) The private key for a Paillier cryptosystem.

Method summary

getCryptosystem()
getLambda() Returns lambda.

Constructors

- **PaillierPrivateKey**
  `public` **PaillierPrivateKey**`( java.math.BigInteger p, java.math.BigInteger q )`
  - **Description**
    The private key for a Paillier cryptosystem.
  - **Parameters**
    - `p` – a prime
    - `q` – a prime with equal length

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
  - **Description copied from PrivateKey** (**in C.1.19, page 116**)
    Retrieves an instance of the proper cryptosystem to work with this key.
  - **Returns** – an instance of the appropriate cryptosystem
- **getLambda**
  `public java.math.BigInteger` **getLambda**`( )`
  - **Description**
    Returns lambda.
  - **Returns** – lambda

Members inherited from class `edu.rit.cryptosystems.PrivateKey` (in C.1.19, page 116)

- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public BigInteger` **getP**`( )`
- `public BigInteger` **getQ**`( )`

C.1.18 Class PaillierPublicKey

The definition of a public key to be used with a Paillier cryptosystem.

Declaration

public class PaillierPublicKey
extends edu.rit.cryptosystems.PublicKey (in C.1.20, page 118)

All known subclasses

DamgardJurikPublicKey (in C.1.5, page 98)

Constructor summary

PaillierPublicKey(BigInteger, BigInteger) The public key for a Paillier cryptosystem.

Method summary

getCryptosystem()

Constructors

- **PaillierPublicKey**
  `public` **PaillierPublicKey**`( java.math.BigInteger n, java.math.BigInteger y )`
  - **Description**
    The public key for a Paillier cryptosystem.
  - **Parameters**
    - `n` – an RSA composite
    - `y` – a wisely chosen constant

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`

Members inherited from class `edu.rit.cryptosystems.PublicKey` (in C.1.20, page 118)

- `protected e`
- `public abstract Cryptosystem` **getCryptosystem**`( )`
- `public int` **getExp**`( )`
- `public BigInteger` **getN**`( )`
- `public BigInteger` **getY**`( )`

C.1.19 Class PrivateKey

The minimal definition of a private key for an RSA-composite based cryptosystem.

Declaration

public abstract class PrivateKey
extends java.lang.Object

All known subclasses

PaillierPrivateKey (in C.1.17, page 114), NaccacheSternPrivateKey (in C.1.13, page 109), GoldwasserMicaliPrivateKey (in C.1.8, page 102), DamgardJurikPrivateKey (in C.1.4, page 97)

Constructor summary

PrivateKey(BigInteger, BigInteger) The private key for a generic RSA-Composite based cryptosystem.

Method summary

getCryptosystem() Retrieves an instance of the proper cryptosystem to work with this key.
getP() The prime factor p.
getQ() The prime factor q.

Constructors

- **PrivateKey**
  `public` **PrivateKey**`( java.math.BigInteger` **p**`,`
  `java.math.BigInteger` **q** `)`
  - **Description**
    The private key for a generic RSA-Composite based cryptosystem.
  - **Parameters**
    - `p` – a prime
    - `q` – a prime with equal length

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
  - **Description**
    Retrieves an instance of the proper cryptosystem to work with this key.
  - **Returns** – an instance of the appropriate cryptosystem
- **getP**
  `public java.math.BigInteger` **getP**`( )`
  - **Description**
    The prime factor p.
  - **Returns** – the prime factor p
- **getQ**
  `public java.math.BigInteger` **getQ**`( )`
  - **Description**
    The prime factor q.
  - **Returns** – the prime factor q

C.1.20 Class PublicKey

The minimal definition of a public key for an RSA-composite based cryptosystem.

Declaration

public abstract class PublicKey
extends java.lang.Object

All known subclasses

PaillierPublicKey (in C.1.18, page 115), NaccacheSternPublicKey (in C.1.14, page 110), GoldwasserMicaliPublicKey (in C.1.9, page 103), DamgardJurikPublicKey (in C.1.5, page 98)

Field summary

e

Constructor summary

PublicKey(BigInteger, BigInteger, int) The public key for a generic RSA-Composite based cryptosystem.

Method summary

getCryptosystem()
getExp() The exponent for the RSA composite.
getN() The RSA composite n.
getY() The specially chosen constant for easy backdoor computation.

Fields

- protected int **e**

Constructors

- **PublicKey**
  `public` **PublicKey**`( java.math.BigInteger n, java.math.BigInteger y, int e )`
  - **Description**
    The public key for a generic RSA-Composite based cryptosystem.
  - **Parameters**
    * `n` – an RSA composite

  - `y` – a wisely chosen constant
  - `e` – the RSA composites exponent

Methods

- **getCryptosystem**
  `public abstract Cryptosystem` **getCryptosystem**`( )`
- **getExp**
  `public int` **getExp**`( )`
  - **Description**
    The exponent for the RSA composite.
  - **Returns** – exponent
- **getN**
  `public java.math.BigInteger` **getN**`( )`
  - **Description**
    The RSA composite n.
  - **Returns** – the RSA composite
- **getY**
  `public java.math.BigInteger` **getY**`( )`
  - **Description**
    The specially chosen constant for easy backdoor computation.
  - **Returns** – the constant y

C.2 Client-Side Library

Package Contents *Page*

Interfaces

IClient 120
Client interface for local operations on a client.

Classes

ChangClient 122
Implementation of client side operations for Chang protocol.

CollaborationClient 123
Implementation of client side operations for Chang protocol.

ExtendedChangClient 124
The client side functionality of Chang extended for use with the Damgard-Jurik cryptosystem to use the block size factor for larger data transfer in one execution with a smaller key.

GeneralizedChangClient 126
The client side functionality of Chang generalized for use with the Damgard-Jurik cryptosystem to use the block size factor for larger data transfer in one execution with a smaller key, but with the ability to limit the number of splits.

HypercubeClient 127
Implementation of the client-side query creation for all of the hypercube based protocols.

LipmaaClient 130
Implementation of client side operations for Lipmaa protocol.

PreSplitChangClient 131

SternClient 132
Implementation of client side operations for Stern protocol.

The local functions of a client in a Private Information Retrieval protocol.

C.2.1 Interface IClient

Client interface for local operations on a client. Q represents the type of the query, SR represents the type of the server-side result to be extracted, and R is the type of the result queried for.

Declaration

public interface IClient

All known subinterfaces

SternClient (in C.2.9, page 132), PreSplitChangClient (in C.2.8, page 131), LipmaaClient (in C.2.7, page 130), HypercubeClient (in C.2.6, page 127), GeneralizedChang-

Client (in C.2.5, page 126), ExtendedChangClient (in C.2.4, page 124), CollaborationClient (in C.2.3, page 123), ChangClient (in C.2.2, page 122)

All classes known to implement interface

SternClient (in C.2.9, page 132), HypercubeClient (in C.2.6, page 127)

Field summary

KEY_LENGTH_BUFFER

Method summary

extractResult(SR) Extracts the result from the encrypted server-side result.
generateQuery(int) Generates a query for a server to process.
getDomain() The domain for modular arithmetic to be performed by a server while processing the private query.

Fields

- int **KEY_LENGTH_BUFFER**

Methods

- **extractResult**
  `R` **extractResult**`( SR` **serverResult** `)`
  - **Description**
    Extracts the result from the encrypted server-side result.
  - **Parameters**
    - `serverResult` – the server-side result
  - **Returns** – the desired result of the private query
- **generateQuery**
  `Q` **generateQuery**`( int` **index** `)`
  - **Description**
    Generates a query for a server to process.
  - **Parameters**
    - `index` – the index of the element to be retrieved
  - **Returns** – a private query
- **getDomain**
  `java.math.BigInteger` **getDomain**`( )`

- **Description**
  The domain for modular arithmetic to be performed by a server while processing the private query. The ciphertext domain.
- **Returns** – modulus for modular arithmetic

C.2.2 Class ChangClient

Implementation of client side operations for Chang protocol.

Declaration

public class ChangClient
extends edu.rit.local.simple.client.HypercubeClient (in C.2.6, page 127)

All known subclasses

PreSplitChangClient (in C.2.8, page 131)

Constructor summary

ChangClient(int, int, int, int, Cryptosystem, KeyGen) Creates a client to interact with a ChangServer.

Method summary

extractResult(BigInteger[])

Constructors

- **ChangClient**
  `public` **ChangClient**`(` `int` **dbSize**`,` `int` **dataSize**`,` `int` **numDim**`,` `int` **dimLength**`,` `edu.rit.cryptosystems.Cryptosystem` **cry**`,` `edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a ChangServer. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `cry` – the cryptosystem to be used with this client, Paillier
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
  `public java.math.BigInteger` **extractResult**`(`
  `java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.3 Class CollaborationClient

Implementation of client side operations for Chang protocol.

Declaration

public class CollaborationClient
extends edu.rit.local.simple.client.HypercubeClient (in C.2.6, page 127)

Constructor summary

CollaborationClient(int, int, int, int, int, Cryptosystem, KeyGen) Creates a client to interact with a ChangServer.

Method summary

extractResult(BigInteger[])

Constructors

- **CollaborationClient**
  `public` **CollaborationClient**`( int` **dbSize**`, int` **dataSize**`, int` **numDim**`, int` **dimLength**`, int` **maxExp**`,`
  `edu.rit.cryptosystems.Cryptosystem` **cry**`,`
  `edu.rit.cryptosystems.KeyGen` **gen** `)`

– **Description**
Creates a client to interact with a ChangServer. All parameters except cry and gen should be retrieved from the server object.

– **Parameters**
* `dbSize` – number of records in the database
* `dataSize` – the size of each record in the database
* `numDim` – the number of dimensions
* `dimLength` – the length of a dimension
* `cry` – the cryptosystem to be used with this client, Paillier
* `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
`public java.math.BigInteger` **extractResult**`(`
`java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.4 Class ExtendedChangClient

The client side functionality of Chang extended for use with the Damgard-Jurik cryptosystem to use the block size factor for larger data transfer in one execution with a smaller key.

Declaration

public class ExtendedChangClient
extends edu.rit.local.simple.client.HypercubeClient (in C.2.6, page 127)

Constructor summary

ExtendedChangClient(int, int, int, int, int, Cryptosystem, KeyGen) Creates a client to interact with a ChangServer.

Method summary

extractResult(BigInteger[])

Constructors

- **ExtendedChangClient**
  `public` **ExtendedChangClient**`( int` **dbSize**`, int` **dataSize**`, int` **numDim**`, int` **dimLength**`, int` **maxExp**`,`
  `edu.rit.cryptosystems.Cryptosystem` **cry**`,`
  `edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a ChangServer. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `maxExp` – the exponent for the cryptosystem, s
    - `cry` – the cryptosystem to be used with this client, Damgard-Jurik
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
  `public java.math.BigInteger` **extractResult**`(`
  `java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**

- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.5 Class GeneralizedChangClient

The client side functionality of Chang generalized for use with the Damgard-Jurik cryptosystem to use the block size factor for larger data transfer in one execution with a smaller key, but with the ability to limit the number of splits.

Declaration

public class GeneralizedChangClient
extends edu.rit.local.simple.client.HypercubeClient (in C.2.6, page 127)

Constructor summary

GeneralizedChangClient(int, int, int, int, int, int, Cryptosystem, KeyGen) Creates a client to interact with a ChangServer.

Method summary

extractResult(BigInteger[])

Constructors

- **GeneralizedChangClient**
  `public` **GeneralizedChangClient**`(` `int` **dbSize**`,` `int` **dataSize**`,` `int` **numDim**`,` `int` **dimLength**`,` `int` **numSplits**`,` `int` **maxExp**`,` `edu.rit.cryptosystems.Cryptosystem` **cry**`,` `edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a ChangServer. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `numSplits` – the number of splits, so that the ciphertext can be in the domain of the plaintext
    - `maxExp` – the exponent for the cryptosystem, s

* `cry` – the cryptosystem to be used with this client, Damgard-Jurik
* `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
  `public java.math.BigInteger` **extractResult**`(`
  `java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.6 Class HypercubeClient

Implementation of the client-side query creation for all of the hypercube based protocols. In general design practice, this class would probably not encompass the Lipmaa protocol as well, because of the varying plaintext.

Declaration

public abstract class HypercubeClient
extends java.lang.Object
implements IClient

All known subclasses

PreSplitChangClient (in C.2.8, page 131), LipmaaClient (in C.2.7, page 130), GeneralizedChangClient (in C.2.5, page 126), ExtendedChangClient (in C.2.4, page 124), CollaborationClient (in C.2.3, page 123), ChangClient (in C.2.2, page 122)

Field summary

cry
dataSize
dbSize
decreasing
dimLength
gen
increasing
maxExp
numDim
numSplits
pri
pub

Constructor summary

HypercubeClient(int, int, int, int, int, int, boolean, boolean, Cryptosystem, KeyGen) Creates a client to interact with a server that retrieves data from the database in a hypercube fashion.

Method summary

generateQuery(int)
getDomain()

Fields

- protected int **dbSize**
- protected int **dataSize**
- protected int **numDim**
- protected int **dimLength**
- protected int **maxExp**
- protected int **numSplits**
- protected edu.rit.cryptosystems.Cryptosystem **cry**
- protected edu.rit.cryptosystems.KeyGen **gen**
- protected edu.rit.cryptosystems.PublicKey **pub**
- protected edu.rit.cryptosystems.PrivateKey **pri**
- protected boolean **increasing**
- protected boolean **decreasing**

Constructors

- **HypercubeClient**
  `public` **HypercubeClient(** `int` **dbSize,** `int` **dataSize,** `int` **numDim,** `int` **dimLength,** `int` **numSplits,** `int` **maxExp,** `boolean` **increasing,** `boolean` **decreasing,** `edu.rit.cryptosystems.Cryptosystem` **cry,** `edu.rit.cryptosystems.KeyGen` **gen)**
  - **Description**
    Creates a client to interact with a server that retrieves data from the database in a hypercube fashion. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `numSplits` – the number of splits, so that the ciphertext can be in the domain of the plaintext
    - `maxExp` – the exponent for the cryptosystem, s
    - `increasing` – basically for the Lipmaa server
    - `cry` – the cryptosystem to be used with this client, Damgard-Jurik
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **generateQuery**
  `Q` **generateQuery(** `int` **index)**
  - **Description copied from IClient** (**in C.2.1, page 120**)
    Generates a query for a server to process.
  - **Parameters**
    - `index` – the index of the element to be retrieved
  - **Returns** – a private query
- **getDomain**
  `java.math.BigInteger` **getDomain(**)
  - **Description copied from IClient** (**in C.2.1, page 120**)
    The domain for modular arithmetic to be performed by a server while processing the private query. The ciphertext domain.
  - **Returns** – modulus for modular arithmetic

C.2.7 Class LipmaaClient

Implementation of client side operations for Lipmaa protocol.

Declaration

public class LipmaaClient
extends edu.rit.local.simple.client.HypercubeClient (in C.2.6, page 127)

Constructor summary

LipmaaClient(int, int, int, int, int, Cryptosystem, KeyGen)
Creates a client to interact with a Lipmaa Server.

Method summary

extractResult(BigInteger)

Constructors

- **LipmaaClient**
`public` **LipmaaClient**`( int` **dbSize**`, int` **dataSize**`, int` **numDim**`, int` **dimLength**`, int` **maxExp**`, edu.rit.cryptosystems.Cryptosystem` **cry**`, edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a Lipmaa Server. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `maxExp` – the exponent for the cryptosystem, s
    - `cry` – the cryptosystem to be used with this client, Damgard-Jurik
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
`public java.math.BigInteger` **extractResult**`( java.math.BigInteger` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.8 Class PreSplitChangClient

Declaration

public class PreSplitChangClient
extends edu.rit.local.simple.client.ChangClient (in C.2.2, page 122)

Constructor summary

PreSplitChangClient(int, int, int, int, Cryptosystem, KeyGen) Creates a client to interact with a ChangServer.

Method summary

extractResult(BigInteger[])

Constructors

- **PreSplitChangClient**
  `public` **PreSplitChangClient**`( int` **dbSize**`, int` **dataSize**`, int` **numDim**`, int` **dimLength**`, edu.rit.cryptosystems.Cryptosystem` **cry**`, edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a ChangServer. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `numDim` – the number of dimensions
    - `dimLength` – the length of a dimension
    - `cry` – the cryptosystem to be used with this client, Paillier
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
  `public java.math.BigInteger` **extractResult**`(`
  `java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.ChangClient` (in C.2.2, page 122)

- `public BigInteger` **extractResult**`( java.math.BigInteger[]` **serverResult** `)`

Members inherited from class `edu.rit.local.simple.client.HypercubeClient` (in C.2.6, page 127)

- `protected` **cry**
- `protected` **dataSize**
- `protected` **dbSize**
- `protected` **decreasing**
- `protected` **dimLength**
- `protected` **gen**
- `public BigInteger` **generateQuery**`( int` **index** `)`
- `public BigInteger` **getDomain**`( )`
- `protected` **increasing**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `protected` **pri**
- `protected` **pub**

C.2.9 Class SternClient

Implementation of client side operations for Stern protocol.

Declaration

public class SternClient
extends java.lang.Object
implements IClient

Constructor summary

SternClient(int, int, Cryptosystem, KeyGen) Creates a client to interact with a Stern Server.

Method summary

extractResult(BigInteger)
generateQuery(int)
getDomain()

Constructors

- **SternClient**
  `public` **SternClient**`( int` **dbSize**`, int` **dataSize**`,`
  `edu.rit.cryptosystems.Cryptosystem` **cry**`,`
  `edu.rit.cryptosystems.KeyGen` **gen** `)`
  - **Description**
    Creates a client to interact with a Stern Server. All parameters except cry and gen should be retrieved from the server object.
  - **Parameters**
    - `dbSize` – number of records in the database
    - `dataSize` – the size of each record in the database
    - `cry` – the cryptosystem to be used with this client
    - `gen` – the key generated to create keys for the cryptosystem

Methods

- **extractResult**
  `public java.math.BigInteger` **extractResult**`(`
  `java.math.BigInteger` **serverResult** `)`
- **generateQuery**
  `Q` **generateQuery**`( int` **index** `)`
  - **Description copied from IClient** (**in C.2.1, page 120**)
    Generates a query for a server to process.
  - **Parameters**
    - `index` – the index of the element to be retrieved
  - **Returns** – a private query
- **getDomain**
  `java.math.BigInteger` **getDomain**`( )`
  - **Description copied from IClient** (**in C.2.1, page 120**)
    The domain for modular arithmetic to be performed by a server while processing the private query. The ciphertext domain.
  - **Returns** – modulus for modular arithmetic

C.3 Server-Side Library

Package Contents *Page*

Interfaces

IServer 134

Interface for a simple server, the local functions.

Classes

BigIntegerServer 136
Provides all of the database functionality for those servers using a BigInteger[] as the database.

ChangServer 139
Uses a BigInteger[] as the database, but implements the server side functionality 9 in the Chang Private Information Retrieval protocol.

ChangServer.NHypercubeThread 143
Thread helper to execute recusive nHypercube calls as seperate threads.

CollaborationServer 145
Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol.

ExtendedChangServer 148
Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol, but extended in the sense that the cryptosystem used in Damgard-Jurik's extension to Paillier's cryptosystem, and thus rather than considering maximum exponent as 1 as in Paillier's, the maximum exponent can now be greater than 1.

GeneralizedChangServer 150
Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol, but extended in the sense that the cryptosystem used in Damgard-Jurik's extension to Paillier's cryptosystem, and thus rather than considering maximum exponent as 1 as in Paillier's, the maximum exponent can now be greater than 1.

LipmaaServer 153
Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Lipmaa Private Information Retrieval protocol.

PreSplitChangServer 154

SternServer 157
Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Stern Private Information Retrieval protocol.

The local functions of a server in a Private Information Retrieval protocol.

C.3.1 Interface IServer

Interface for a simple server, the local functions. D represents the database type, Q represents the query type, and R represents the result type. But D, Q and R should only be BigInteger, BigInteger[], or BigInteger[][].

Declaration

public interface IServer

All known subinterfaces

SternServer (in C.3.10, page 157), PreSplitChangServer (in C.3.9, page 154), LipmaaServer (in C.3.8, page 153), GeneralizedChangServer (in C.3.7, page 150), ExtendedChangServer (in C.3.6, page 148), CollaborationServer (in C.3.5, page 145), ChangServer (in C.3.3, page 139), BigIntegerServer (in C.3.2, page 136)

All classes known to implement interface

BigIntegerServer (in C.3.2, page 136)

Method summary

getDatabaseDimensions() The number of dimensions this database is referenced by.
getDatabaseSize() The size of the database.
getDataSize() The size of the data in bytes.
getDimensionLength() The length of a dimension.
getMaxExponent() The maximum exponent, s, used in the Damgard-Jurik cryptosystem.
getNumberOfSplits() The number of splits, necessary for the Chang protocol.
performRetrieval(Q, BigInteger) Public function for performing the retrieval.
setDatabase(D) Sets the database for this server.

Methods

- **getDatabaseDimensions**
  `int` **getDatabaseDimensions**`( )`
  - **Description**
    The number of dimensions this database is referenced by.
  - **Returns** – number of dimensions for referencing an item in this database
- **getDatabaseSize**
  `int` **getDatabaseSize**`( )`
  - **Description**
    The size of the database. The number of entries in the database.
  - **Returns** – size of the database
- **getDataSize**
  `int` **getDataSize**`( )`
  - **Description**
    The size of the data in bytes.

  - **Returns** – size of data

- **getDimensionLength**
  `int` **getDimensionLength**`( )`
  - **Description**
    The length of a dimension.
  - **Returns** – number of records in a dimension

- **getMaxExponent**
  `int` **getMaxExponent**`( )`
  - **Description**
    The maximum exponent, s, used in the Damgard-Jurik cryptosystem.
  - **Returns** – maximum exponent, s, used

- **getNumberOfSplits**
  `int` **getNumberOfSplits**`( )`
  - **Description**
    The number of splits, necessary for the Chang protocol.
  - **Returns** – the number of splits per dimension

- **performRetrieval**
  `R` **performRetrieval**`( Q` **query**`, java.math.BigInteger` **domain** `)`
  - **Description**
    Public function for performing the retrieval.
  - **Parameters**
    * `query` – user query
    * `domain` – BigInteger for modular arithmetic
  - **Returns** – the result of the retrieval, encrypted

- **setDatabase**
  `void` **setDatabase**`( D` **db** `)`
  - **Description**
    Sets the database for this server.
  - **Parameters**
    * `db` – the database

C.3.2 Class BigIntegerServer

Provides all of the database functionality for those servers using a BigInteger[] as the database. Extending classes will then only need to implement the protocol for private information retrieval.

Declaration

public abstract class BigIntegerServer
extends java.lang.Object
implements IServer

All known subclasses

SternServer (in C.3.10, page 157), PreSplitChangServer (in C.3.9, page 154), LipmaaServer (in C.3.8, page 153), GeneralizedChangServer (in C.3.7, page 150), ExtendedChangServer (in C.3.6, page 148), CollaborationServer (in C.3.5, page 145), ChangServer (in C.3.3, page 139)

Field summary

dataSize
db
dbSize
dimLength
maxExp
numDim
numSplits

Constructor summary

BigIntegerServer(int, int, int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

getDatabaseDimensions()
getDatabaseSize()
getDataSize()
getDimensionLength()
getMaxExponent()
getNumberOfSplits()
setDatabase(BigInteger[])

Fields

- protected java.math.BigInteger **db**
- protected int **dbSize**
- protected int **dataSize**
- protected int **numDim**

- protected int **dimLength**
- protected int **numSplits**
- protected int **maxExp**

Constructors

- **BigIntegerServer**
  `public` **BigIntegerServer**`( int` **numDim**`, int` **maxExp**`, int` **numSplits** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a GeneralizedChangClient.
  - **Parameters**
    - `numDim` – the number of dimensions to reference to reference the database by
    - `numSplits` – the number of splits per dimension of the hypercube
    - `maxExp` – the maximum exponent, s, to be used in a Lipmaa Public Key

Methods

- **getDatabaseDimensions**
  `int` **getDatabaseDimensions**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The number of dimensions this database is referenced by.
  - **Returns** – number of dimensions for referencing an item in this database
- **getDatabaseSize**
  `int` **getDatabaseSize**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The size of the database. The number of entries in the database.
  - **Returns** – size of the database
- **getDataSize**
  `int` **getDataSize**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The size of the data in bytes.
  - **Returns** – size of data

- **getDimensionLength**
  `int` **getDimensionLength**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The length of a dimension.
  - **Returns** – number of records in a dimension
- **getMaxExponent**
  `int` **getMaxExponent**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The maximum exponent, s, used in the Damgard-Jurik cryptosystem.
  - **Returns** – maximum exponent, s, used
- **getNumberOfSplits**
  `int` **getNumberOfSplits**`( )`
  - **Description copied from IServer** (**in C.3.1, page 134**)
    The number of splits, necessary for the Chang protocol.
  - **Returns** – the number of splits per dimension
- **setDatabase**
  `public void` **setDatabase**`( java.math.BigInteger[] db )`

C.3.3 Class ChangServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol.

Declaration

public class ChangServer
extends edu.rit.local.simple.server.BigIntegerServer (in C.3.2, page 136)

All known subclasses

PreSplitChangServer (in C.3.9, page 154), GeneralizedChangServer (in C.3.7, page 150), ExtendedChangServer (in C.3.6, page 148), CollaborationServer (in C.3.5, page 145)

Constructor summary

ChangServer(int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

filter(BigInteger[], BigInteger, BigInteger[], BigInteger) Filters the split from the two-dimensional hypercube method.
getResultLength(int) Determines the length for a result, at a given dimension
nFilter(BigInteger[], BigInteger[][], BigInteger[], BigInteger, int, int) Filters the recent split split from an nHypercube call split
nHypercube(BigInteger[][], BigInteger, BigInteger, int, int) A recursive method that performs the private retrieval of a query on the databse.
nSplit(BigInteger[], BigInteger) Splits the split from the previous split.
performRetrieval(BigInteger[][], BigInteger)
sigma(BigInteger[], BigInteger, int, int) Computes sigma using only the top two dimension of the database
split(BigInteger, BigInteger) Splits a sigma in the two-dimensional hypercube method.
twoHypercube(BigInteger[][], BigInteger, BigInteger, int) The base case of the nHypercube method.

Constructors

- **ChangServer**
  `public` **ChangServer**`( int` **numDim** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a ChangClient.
  - **Parameters**
    - `numDim` – the number of dimensions to reference to reference the database by

Methods

- **filter**
  `protected java.math.BigInteger[]` **filter**`(`
  `java.math.BigInteger[]` **result**`, java.math.BigInteger` **query**`,`
  `java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus** `)`
  - **Description**
    Filters the split from the two-dimensional hypercube method.
  - **Parameters**

    - `result` – the result thus far, for further manipulation
    - `query` – the query presented by the client, that particular one
    - `split` – the current split operating with
    - `modulus` – the cipher text domain
  - **Returns** – the filtered result, through the current sigma

- **getResultLength**
  `protected int` **getResultLength**`( int` **dim** `)`
  - **Description**
    Determines the length for a result, at a given dimension
  - **Parameters**
    - `dim` – the current dimension this is to be computed for
  - **Returns** – the length of a result

- **nFilter**
  `protected java.math.BigInteger[]` **nFilter**`(`
  `java.math.BigInteger[]` **result**`, java.math.BigInteger[][]`
  query`, java.math.BigInteger[]` **split**`, java.math.BigInteger`
  modulus`, int` **k**`, int` **dim** `)`
  - **Description**
    Filters the recent split split from an nHypercube call split
  - **Parameters**
    - `result` – the result being manipulated further
    - `query` – the client issued query
    - `split` – the split to be filtered
    - `modulus` – the ciphertext domain
    - `k` – the current in dex in the current dimension of the query
    - `dim` – the current dimension being considered
  - **Returns** – the recent manipulation of the filtered result

- **nHypercube**
  `protected java.math.BigInteger[]` **nHypercube**`(`
  `java.math.BigInteger[][]` **query**`, java.math.BigInteger`
  domain`, java.math.BigInteger` **n2**`, int` **dim**`, int` **index** `)`
  - **Description**
    A recursive method that performs the private retrieval of a query on the databse.
  - **Parameters**
    - `query` – the client generated query
    - `domain` – the public key RSA composite from the client

    - `n2` – the domain of the ciphertext
    - `dim` – the current dimension we are on - should call with 0
    - `index` – the index thus far - should call with 0
  - **Returns** – the filtered result of the current dimension

- **nSplit**
  `protected java.math.BigInteger[] nSplit( java.math.BigInteger[] prevSplit, java.math.BigInteger domain )`
  - **Description**
    Splits the split from the previous split.
  - **Parameters**
    - `prevSplit` – previous split from the last recursive nHypercube call
    - `domain` – the plaintext domain, RSA composite of the public key
  - **Returns** – the new split

- **performRetrieval**
  `public java.math.BigInteger[] performRetrieval( java.math.BigInteger[][] query, java.math.BigInteger domain )`

- **sigma**
  `protected java.math.BigInteger sigma( java.math.BigInteger[] query, java.math.BigInteger modulus, int dim, int index )`
  - **Description**
    Computes sigma using only the top two dimension of the database
  - **Parameters**
    - `query` – the client generated query, top dimension
    - `modulus` – the ciphertext domain
    - `dim` – the index of the second level dimension
    - `index` – the index thus far minus the first and second dimensions
  - **Returns** – sigma for that particular second dimension

- **split**
  `protected java.math.BigInteger[] split( java.math.BigInteger sigma, java.math.BigInteger domain )`
  - **Description**
    Splits a sigma in the two-dimensional hypercube method.
  - **Parameters**
    - `sigma` – the sigma to split

    * `domain` – the RSA composite of the client's public-key
  - **Returns** – the results of the split

- **twoHypercube**
  `protected java.math.BigInteger[] twoHypercube( java.math.BigInteger[][] query, java.math.BigInteger domain, java.math.BigInteger n2, int index )`

  - **Description**
    The base case of the nHypercube method. Actually references the database.
  - **Parameters**
    * `query` – the client generated query
    * `domain` – the public key RSA composite from the client
    * `n2` – the domain of the ciphertext
    * `index` – the index thus far
  - **Returns** – the filtered result of the top two dimensions

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions( )**
- `public int` **getDatabaseSize( )**
- `public int` **getDataSize( )**
- `public int` **getDimensionLength( )**
- `public int` **getMaxExponent( )**
- `public int` **getNumberOfSplits( )**
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase(java.math.BigInteger[] db)**

C.3.4 Class ChangServer.NHypercubeThread

Thread helper to execute recusive nHypercube calls as seperate threads.

Declaration

protected class ChangServer.NHypercubeThread
extends java.lang.Thread

Constructor summary

ChangServer.NHypercubeThread(BigInteger[][], BigInteger, BigInteger, int, int) Constructor that compies the parameters into this thread space.

Method summary

getResult() Retrieves the result, or null if no result has been computed.
run() Overriden.

Constructors

- **ChangServer.NHypercubeThread**
  `public` **ChangServer.NHypercubeThread(** `java.math.BigInteger[][]` **query,** `java.math.BigInteger` **domain,** `java.math.BigInteger` **modulus,** `int` **dim,** `int` **index)**
  - **Description**
    Constructor that compies the parameters into this thread space.
  - **Parameters**
    * `query` – client made query
    * `domain` – plaintext doamin
    * `modulus` – cipher text domain
    * `dim` – current dimension of operation
    * `index` – the index thus far from previous dimensions traversed

Methods

- **getResult**
  `public java.math.BigInteger[]` **getResult(** **)**
  - **Description**
    Retrieves the result, or null if no result has been computed.
  - **Returns** – the result of the recursive call
- **run**
  `public void` **run(** **)**
  - **Description**
    Overriden. Executes the recusive call and stores the result in the thread space for later retrieval.

Members inherited from class `java.lang.Thread`

- `public static int` **activeCount(** **)**
- `public final void` **checkAccess(** **)**
- `public native int` **countStackFrames(** **)**
- `public static native Thread` **currentThread(** **)**
- `public void` **destroy(** **)**
- `public static void` **dumpStack(** **)**
- `public static int` **enumerate(** `Thread[]` **arg0)**
- `public static Map` **getAllStackTraces(** **)**
- `public ClassLoader` **getContextClassLoader(** **)**

- public static Thread.UncaughtExceptionHandler **getDefaultUncaughtExceptionHandler**( )
- public long **getId**( )
- public final String **getName**( )
- public final int **getPriority**( )
- public StackTraceElement **getStackTrace**( )
- public Thread.State **getState**( )
- public final ThreadGroup **getThreadGroup**( )
- public Thread.UncaughtExceptionHandler **getUncaughtExceptionHandler**( )
- public static native boolean **holdsLock**(Object **arg0**)
- public void **interrupt**( )
- public static boolean **interrupted**( )
- public final native boolean **isAlive**( )
- public final boolean **isDaemon**( )
- public boolean **isInterrupted**( )
- public final void **join**( ) throws InterruptedException
- public final synchronized void **join**(long **arg0**) throws InterruptedException
- public final synchronized void **join**(long **arg0**, int **arg1**) throws InterruptedException
- public static final **MAX_PRIORITY**
- public static final **MIN_PRIORITY**
- public static final **NORM_PRIORITY**
- public final void **resume**( )
- public void **run**( )
- public void **setContextClassLoader**(ClassLoader **arg0**)
- public final void **setDaemon**(boolean **arg0**)
- public static void **setDefaultUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler **arg0**)
- public final void **setName**(String **arg0**)
- public final void **setPriority**(int **arg0**)
- public void **setUncaughtExceptionHandler**(Thread.UncaughtExceptionHandler **arg0**)
- public static native void **sleep**(long **arg0**) throws InterruptedException
- public static void **sleep**(long **arg0**, int **arg1**) throws InterruptedException
- public synchronized void **start**( )
- public final void **stop**( )
- public final synchronized void **stop**(Throwable **arg0**)
- public final void **suspend**( )
- public String **toString**( )
- public static native void **yield**( )

C.3.5 Class CollaborationServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol.

Declaration

public class CollaborationServer
extends edu.rit.local.simple.server.ChangServer (in C.3.3, page 139)

Constructor summary

CollaborationServer(int, int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

nHypercube(BigInteger[][], BigInteger, BigInteger, int, int) A recursive method that performs the private retrieval of a query on the databse.
performRetrieval(BigInteger[][], BigInteger)
twoHypercube(BigInteger[][], BigInteger, BigInteger, int) The base case of the nHypercube method.

Constructors

- **CollaborationServer**
  `public` **CollaborationServer(** `int` **numDim,** `int` **maxExp)**
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a ChangClient.
  - **Parameters**
    - `numDim` – the number of dimensions to reference to reference the database by

Methods

- **nHypercube**
  `protected java.math.BigInteger[]` **nHypercube(** `java.math.BigInteger[][]` **query,** `java.math.BigInteger` **domain,** `java.math.BigInteger` **n2,** `int` **dim,** `int` **index)**
  - **Description**
    A recursive method that performs the private retrieval of a query on the databse.
  - **Parameters**
    - `query` – the client generated query
    - `domain` – the public key RSA composite from the client
    - `n2` – the domain of the ciphertext
    - `dim` – the current dimension we are on - should call with 0
    - `index` – the index thus far - should call with 0
  - **Returns** – the filtered result of the current dimension

- **performRetrieval**
  `public java.math.BigInteger[] performRetrieval( java.math.BigInteger[][] query, java.math.BigInteger domain )`

- **twoHypercube**
  `protected java.math.BigInteger[] twoHypercube( java.math.BigInteger[][] query, java.math.BigInteger domain, java.math.BigInteger n2, int index )`

  - **Description**
    The base case of the nHypercube method. Actually references the database.
  - **Parameters**
    - `query` – the client generated query
    - `domain` – the public key RSA composite from the client
    - `n2` – the domain of the ciphertext
    - `index` – the index thus far
  - **Returns** – the filtered result of the top two dimensions

Members inherited from class `edu.rit.local.simple.server.ChangServer` (in C.3.3, page 139)

- `protected BigInteger` **filter**`( java.math.BigInteger[]` **result**`, java.math.BigInteger` **query**`, java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus** `)`
- `protected int` **getResultLength**`( int` **dim** `)`
- `protected BigInteger` **nFilter**`( java.math.BigInteger[]` **result**`, java.math.BigInteger[][]` **query**`, java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus**`, int` **k**`, int` **dim** `)`
- `protected BigInteger` **nHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **nSplit**`( java.math.BigInteger[]` **prevSplit**`, java.math.BigInteger` **domain** `)`
- `public BigInteger` **performRetrieval**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **sigma**`( java.math.BigInteger[]` **query**`, java.math.BigInteger` **modulus**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **split**`( java.math.BigInteger` **sigma**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **twoHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **index** `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**

- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

C.3.6 Class ExtendedChangServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol, but extended in the sense that the cryptosystem used in Damgard-Jurik's extension to Paillier's cryptosystem, and thus rather than considering maximum exponent as 1 as in Paillier's, the maximum exponent can now be greater than 1. and the number of splits increases equally with the size of the maximum exponent.

Declaration

public class ExtendedChangServer
extends edu.rit.local.simple.server.ChangServer (in C.3.3, page 139)

Constructor summary

ExtendedChangServer(int, int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

filter(BigInteger[], BigInteger, BigInteger[], BigInteger)
nSplit(BigInteger[], BigInteger)
split(BigInteger, BigInteger)

Constructors

- **ExtendedChangServer**
  `public` **ExtendedChangServer**`( int` **numDim**`, int` **maxExp** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a ExtendedhangClient.
  - **Parameters**

  - `numDim` – the number of dimensions to reference to reference the database by
  - `maxExp` – the maximum exponent, s, to be used in a Damgard-Jurik Public Key

Methods

- **filter**
  `protected java.math.BigInteger[] filter( java.math.BigInteger[] result, java.math.BigInteger query, java.math.BigInteger[] split, java.math.BigInteger modulus )`
  - **Description copied from ChangServer** (in C.3.3, page 139)
    Filters the split from the two-dimensional hypercube method.
  - **Parameters**
    - `result` – the result thus far, for further manipulation
    - `query` – the query presented by the client, that particular one
    - `split` – the current split operating with
    - `modulus` – the cipher text domain
  - **Returns** – the filtered result, through the current sigma
- **nSplit**
  `protected java.math.BigInteger[] nSplit( java.math.BigInteger[] prevSplit, java.math.BigInteger domain )`
  - **Description copied from ChangServer** (in C.3.3, page 139)
    Splits the split from the previous split.
  - **Parameters**
    - `prevSplit` – previous split from the last recursive nHypercube call
    - `domain` – the plaintext domain, RSA composite of the public key
  - **Returns** – the new split
- **split**
  `protected java.math.BigInteger[] split( java.math.BigInteger sigma, java.math.BigInteger domain )`
  - **Description copied from ChangServer** (in C.3.3, page 139)
    Splits a sigma in the two-dimensional hypercube method.
  - **Parameters**
    - `sigma` – the sigma to split
    - `domain` – the RSA composite of the client's public-key
  - **Returns** – the results of the split

Members inherited from class `edu.rit.local.simple.server.ChangServer` (in C.3.3, page 139)

- `protected BigInteger` **filter**`( java.math.BigInteger[]` **result**`, java.math.BigInteger` **query**`, java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus** `)`
- `protected int` **getResultLength**`( int` **dim** `)`
- `protected BigInteger` **nFilter**`( java.math.BigInteger[]` **result**`, java.math.BigInteger[][]` **query**`, java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus**`, int` **k**`, int` **dim** `)`
- `protected BigInteger` **nHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **nSplit**`( java.math.BigInteger[]` **prevSplit**`, java.math.BigInteger` **domain** `)`
- `public BigInteger` **performRetrieval**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **sigma**`( java.math.BigInteger[]` **query**`, java.math.BigInteger` **modulus**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **split**`( java.math.BigInteger` **sigma**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **twoHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **index** `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

C.3.7 Class GeneralizedChangServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Chang Private Information Retrieval protocol, but extended in the sense that the cryptosystem used in Damgard-Jurik's extension to Paillier's cryptosystem, and thus rather than considering maximum exponent as 1 as in Paillier's, the maximum exponent can now be greater than 1. and the number of splits is at most one less than the size of the exponent, and is user prescribed.

Declaration

public class GeneralizedChangServer
extends edu.rit.local.simple.server.ChangServer (in C.3.3, page 139)

Constructor summary

GeneralizedChangServer(int, int, int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

filter(BigInteger[], BigInteger, BigInteger[], BigInteger)
nSplit(BigInteger[], BigInteger)
split(BigInteger, BigInteger)

Constructors

- **GeneralizedChangServer**
  `public` **GeneralizedChangServer**`( int` **numDim**`, int` **numSplits**`, int` **maxExp** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a GeneralizedChangClient.
  - **Parameters**
    - `numDim` – the number of dimensions to reference to reference the database by
    - `numSplits` – the number of splits per dimension of the hypercube
    - `maxExp` – the maximum exponent, s, to be used in a Damgard-Jurik Public Key

Methods

- **filter**
  `protected java.math.BigInteger[]` **filter**`(`
  `java.math.BigInteger[]` **result**`, java.math.BigInteger` **query**`,`
  `java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus** `)`
  - **Description copied from ChangServer** (**in C.3.3, page 139**)
    Filters the split from the two-dimensional hypercube method.
  - **Parameters**
    - `result` – the result thus far, for further manipulation

* `query` – the query presented by the client, that particular one
* `split` – the current split operating with
* `modulus` – the cipher text domain

– **Returns** – the filtered result, through the current sigma

- **nSplit**
  `protected java.math.BigInteger[] nSplit( java.math.BigInteger[] prevSplit, java.math.BigInteger domain )`

  – **Description copied from ChangServer** (in C.3.3, page 139)
  Splits the split from the previous split.

  – **Parameters**

  * `prevSplit` – previous split from the last recursive nHypercube call
  * `domain` – the plaintext domain, RSA composite of the public key

  – **Returns** – the new split

- **split**
  `protected java.math.BigInteger[] split( java.math.BigInteger sigma, java.math.BigInteger domain )`

  – **Description copied from ChangServer** (in C.3.3, page 139)
  Splits a sigma in the two-dimensional hypercube method.

  – **Parameters**

  * `sigma` – the sigma to split
  * `domain` – the RSA composite of the client's public-key

  – **Returns** – the results of the split

Members inherited from class `edu.rit.local.simple.server.ChangServer` (in C.3.3, page 139)

- `protected BigInteger filter( java.math.BigInteger[] result, java.math.BigInteger query, java.math.BigInteger[] split, java.math.BigInteger modulus )`
- `protected int getResultLength( int dim )`
- `protected BigInteger nFilter( java.math.BigInteger[] result, java.math.BigInteger[][] query, java.math.BigInteger[] split, java.math.BigInteger modulus, int k, int dim )`
- `protected BigInteger nHypercube( java.math.BigInteger[][] query, java.math.BigInteger domain, java.math.BigInteger n2, int dim, int index )`
- `protected BigInteger nSplit( java.math.BigInteger[] prevSplit, java.math.BigInteger domain )`
- `public BigInteger performRetrieval( java.math.BigInteger[][] query, java.math.BigInteger domain )`

- `protected BigInteger` **sigma**`( java.math.BigInteger[]` **query**`, java.math.BigInteger` **modulus**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **split**`( java.math.BigInteger` **sigma**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **twoHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **index** `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

C.3.8 Class LipmaaServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Lipmaa Private Information Retrieval protocol.

Declaration

public class LipmaaServer
extends edu.rit.local.simple.server.BigIntegerServer (in C.3.2, page 136)

Constructor summary

LipmaaServer(int, int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

performRetrieval(BigInteger[][], BigInteger)

Constructors

- **LipmaaServer**
  `public` **LipmaaServer**`( int` **numDim**`, int` **maxExp** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a LipmaaClient.
  - **Parameters**
    - `numDim` – the number of dimensions to reference to reference the database by
    - `maxExp` – the maximum exponent, s, to be used in a Damgard-Jurik Public Key

Methods

- **performRetrieval**
  `public java.math.BigInteger` **performRetrieval**`(`
  `java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**
  `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

C.3.9 Class PreSplitChangServer

Declaration

public class PreSplitChangServer
extends edu.rit.local.simple.server.ChangServer (in C.3.3, page 139)

Constructor summary

PreSplitChangServer(int) Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

filter(BigInteger[], BigInteger, BigInteger[], BigInteger) Filters the split from the two-dimensional hypercube method.
getResultLength(int) Determines the length for a result, at a given dimension
split(BigInteger[], BigInteger) Splits a sigma in the two-dimensional hypercube method.
twoHypercube(BigInteger[][], BigInteger, BigInteger, int) The base case of the nHypercube method.

Constructors

- **PreSplitChangServer**
  `public` **PreSplitChangServer**`( int` **numDim** `)`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a ChangClient.
  - **Parameters**
    * `numDim` – the number of dimensions to reference to reference the database by

Methods

- **filter**
  `protected java.math.BigInteger[]` **filter**`(`
  `java.math.BigInteger[]` **result**`, java.math.BigInteger` **query**`,`
  `java.math.BigInteger[]` **split**`, java.math.BigInteger` **modulus** `)`
  - **Description**
    Filters the split from the two-dimensional hypercube method.
  - **Parameters**
    * `result` – the result thus far, for further manipulation
    * `query` – the query presented by the client, that particular one
    * `split` – the current split operating with
    * `modulus` – the cipher text domain
  - **Returns** – the filtered result, through the current sigma
- **getResultLength**
  `protected int` **getResultLength**`( int` **dim** `)`
  - **Description**
    Determines the length for a result, at a given dimension

  - **Parameters**
    - `dim` – the current dimension this is to be computed for
  - **Returns** – the length of a result

- **split**
  `protected java.math.BigInteger[] split( java.math.BigInteger[] sigma, java.math.BigInteger domain )`

  - **Description**
    Splits a sigma in the two-dimensional hypercube method.
  - **Parameters**
    - `sigma` – the sigma to split
    - `domain` – the RSA composite of the client's public-key
  - **Returns** – the results of the split

- **twoHypercube**
  `protected java.math.BigInteger[] twoHypercube( java.math.BigInteger[][] query, java.math.BigInteger domain, java.math.BigInteger n2, int index )`

  - **Description**
    The base case of the nHypercube method. Actually references the database.
  - **Parameters**
    - `query` – the client generated query
    - `domain` – the public key RSA composite from the client
    - `n2` – the domain of the ciphertext
    - `index` – the index thus far
  - **Returns** – the filtered result of the top two dimensions

Members inherited from class `edu.rit.local.simple.server.ChangServer` (in C.3.3, page 139)

- `protected BigInteger filter( java.math.BigInteger[] result, java.math.BigInteger query, java.math.BigInteger[] split, java.math.BigInteger modulus )`
- `protected int getResultLength( int dim )`
- `protected BigInteger nFilter( java.math.BigInteger[] result, java.math.BigInteger[][] query, java.math.BigInteger[] split, java.math.BigInteger modulus, int k, int dim )`
- `protected BigInteger nHypercube( java.math.BigInteger[][] query, java.math.BigInteger domain, java.math.BigInteger n2, int dim, int index )`
- `protected BigInteger nSplit( java.math.BigInteger[] prevSplit, java.math.BigInteger domain )`

- `public BigInteger` **performRetrieval**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **sigma**`( java.math.BigInteger[]` **query**`, java.math.BigInteger` **modulus**`, int` **dim**`, int` **index** `)`
- `protected BigInteger` **split**`( java.math.BigInteger` **sigma**`, java.math.BigInteger` **domain** `)`
- `protected BigInteger` **twoHypercube**`( java.math.BigInteger[][]` **query**`, java.math.BigInteger` **domain**`, java.math.BigInteger` **n2**`, int` **index** `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

C.3.10 Class SternServer

Uses a BigInteger[] as the database, but implements the server side functionality prescribed in the Stern Private Information Retrieval protocol.

Declaration

public class SternServer
extends edu.rit.local.simple.server.BigIntegerServer (in C.3.2, page 136)

Constructor summary

SternServer() Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme.

Method summary

performRetrieval(BigInteger[], BigInteger)

Constructors

- **SternServer**
  `public` **SternServer**`( )`
  - **Description**
    Creates a server object that contains the methods required for the server in homomorphic encryption based private information retrieval scheme. To be used in conjunction with a SternClient.

Methods

- **performRetrieval**
  `public java.math.BigInteger` **performRetrieval**`(`
  `java.math.BigInteger[]` **query**`, java.math.BigInteger` **domain** `)`

Members inherited from class `edu.rit.local.simple.server.BigIntegerServer` (in C.3.2, page 136)

- `protected` **dataSize**
- `protected` **db**
- `protected` **dbSize**
- `protected` **dimLength**
- `public int` **getDatabaseDimensions**`( )`
- `public int` **getDatabaseSize**`( )`
- `public int` **getDataSize**`( )`
- `public int` **getDimensionLength**`( )`
- `public int` **getMaxExponent**`( )`
- `public int` **getNumberOfSplits**`( )`
- `protected` **maxExp**
- `protected` **numDim**
- `protected` **numSplits**
- `public void` **setDatabase**`( java.math.BigInteger[]` **db** `)`

Appendix D

Scripts

D.1 STERN Experiment Script

```
#!/usr/bin/bash
#Stern

K=10
while [ $K -le 100 ]; do
                #Start the server
                java -cp ../../thesis/. edu.rit.remote.
                    server.ServerImpl $K 1024 0 &
                sleep 10

                #Run the clients
                j=1
                while [ $j -le 20 ]; do
                        echo $j$K$c
                        let r=$RANDOM%$K
                        java -cp ../../thesis/. edu.rit.
                            remote.client.Client localhost
                             $r

                        let j=$j+1
                done

                #Kill the current server
                pid=`pgrep java`
                kill $pid

                #Move *.log files to another directory
                mkdir Stern$K
```

```
            mv *.log Stern$K/.

        let K=$K*10
done

exit 0
```

D.2 CHANG Experiment Script

```
#!/usr/bin/bash
#Chang

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    #Start the server
    java -cp ../../thesis/. edu.rit.remote.server.
        ServerImpl $K 1024 1 $c &
    sleep 10

    #Run the clients
    j=1
    while [ $j -le 20 ]; do
      echo $j$K$c
      let r=$RANDOM%$K
      java -cp ../../thesis/. edu.rit.remote.client.
          Client localhost $r

      let j=$j+1
    done

    #Kill the current server
    pid=`pgrep java`
    kill $pid

    #Move *.log files to another directory
    mkdir Chang$K$c
    mv *.log Chang$K$c/.

    let c=$c+1
  done

  let K=$K*10
done
```

```
exit 0
```

D.3 Pre-Split Experiment Script

```
#!/usr/bin/bash
#Pre-Split

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    #Start the server
    java -cp ../../thesis/. edu.rit.remote.server.
        ServerImpl $K 1024 5 $c &
    sleep 10

    #Run the clients
    j=1
    while [ $j -le 20 ]; do
      echo $j
      let r=$RANDOM%$K
      java -cp ../../thesis/. edu.rit.remote.client.
          Client localhost $r

      let j=$j+1
    done

    #Kill the current server
    pid=`pgrep java`
    kill $pid

    #Move *.log files to another directory
    mkdir PreSplit$K$c
    mv *.log PreSplit$K$c/.

    let c=$c+1
  done

  let K=$K*10
done

exit 0
```

D.4 Extended Experiment Script

```
#!/usr/bin/bash
#Extended

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    s=1
    while [ $s -le 4 ]; do
      #Start the server
      java -cp ../../thesis/. edu.rit.remote.server.
          ServerImpl $K 1024 3 $c $s &
      sleep 10

      #Run the clients
      j=1
      while [ $j -le 20 ]; do
        echo $j
        let r=$RANDOM%$K
        java -cp ../../thesis/. edu.rit.remote.client.
            Client localhost $r

        let j=$j+1
      done

      #Kill the current server
      pid=`pgrep java`
      kill $pid

      #Move *.log files to another directory
      mkdir Extended$K$c$s
      mv *.log Extended$K$c$s/.

      let s=$s+1
    done

    let c=$c+1
  done

  let K=$K*10
done

exit 0
```

D.5 Generalized Experiment Script

```
#!/usr/bin/bash
#Generalized

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    s=1
    while [ $s -le 4 ]; do
      #Start the server
      java -cp ../../thesis/. edu.rit.remote.server.
          ServerImpl $K 1024 4 $c 2 $s &
      sleep 10

      #Run the clients
      j=1
      while [ $j -le 20 ]; do
        echo $j
        let r=$RANDOM%$K
        java -cp ../../thesis/. edu.rit.remote.client.
            Client localhost $r

        let j=$j+1
      done

      #Kill the current server
      pid=`pgrep java`
      kill $pid

      #Move *.log files to another directory
      mkdir Generalized$K$c$s
      mv *.log Generalized$K$c$s/.

      let s=$s+1
    done

    let c=$c+1
  done

  let K=$K*10
done

exit 0
```

D.6 LIPMAA Experiment Script

```
#!/usr/bin/bash
#Lipmaa

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    s=1
    while [ $s -le 4 ]; do
      #Start the server
      java -cp ../../thesis/. edu.rit.remote.server.
          ServerImpl $K 1024 2 $c $s &
      sleep 10

      #Run the clients
      j=1
      while [ $j -le 20 ]; do
        echo $j
        let r=$RANDOM%$K
        java -cp ../../thesis/. edu.rit.remote.client.
            Client localhost $r

        let j=$j+1
      done

      #Kill the current server
      pid=`pgrep java`
      kill $pid

      #Move *.log files to another directory
      mkdir Lipmaa$K$c$s
      mv *.log Lipmaa$K$c$s/.

      let s=$s+1
    done

    let c=$c+1
  done

  let K=$K*10
done

exit 0
```

D.7 LIPMAA versus COLLABORATION Experiment Script

```
#!/usr/bin/bash
#LipmaaVCollab

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    s=1
    while [ $s -le 4 ]; do
      let exp=2**$s+1
      #Start the server
      java -cp ../../thesis/. edu.rit.remote.server.
          ServerImpl $K 1024 2 $c $exp &
      sleep 10

      #Run the clients
      j=1
      while [ $j -le 20 ]; do
        echo $j
        let r=$RANDOM%$K
        java -cp ../../thesis/. edu.rit.remote.client.
            Client localhost $r

        let j=$j+1
      done

      #Kill the current server
      pid=`pgrep java`
      kill $pid

      #Move *.log files to another directory
      mkdir LipmaaVCollab$K$c$s
      mv *.log LipmaaVCollab$K$c$s/.

      let s=$s+1
    done

    let c=$c+1
  done

  let K=$K*10
done
```

```
exit 0
```

D.8 COLLABORATION Experiment Script

```
#!/usr/bin/bash
#Collab

K=10
while [ $K -le 100 ]; do
  c=2
  while [ $c -le 4 ]; do
    s=1
    while [ $s -le 4 ]; do
      let exp=2**$s+1
      echo exp$exp
      #Start the server
      java -cp ../../thesis/. edu.rit.remote.server.
          ServerImpl $K 1024 6 $c $exp &
      sleep 10

      #Run the clients
      j=1
      while [ $j -le 20 ]; do
        echo $j
        let r=$RANDOM%$K
        java -cp ../../thesis/. edu.rit.remote.client.
            Client localhost $r

        let j=$j+1
      done

      #Kill the current server
      pid=`pgrep java`
      kill $pid

      #Move *.log files to another directory
      mkdir Collab$K$c$s
      mv *.log Collab$K$c$s/.

      let s=$s+1
    done

    let c=$c+1
  done
```

```
  let K=$K*10
done

exit 0
```

D.9 Result Collection Script

```
#!/bin/bash
#Thesis Results Collection

topDirs=`ls `
echo $topDirs

for top in $topDirs
do

    if [ $top != ThesisResults.sh ]; then
    cd $top
    echo top:`pwd`

    subDirs=`ls `

    for sub in $subDirs
    do

        let query=0
        queryMin=9999999999
        queryMax=0

        extraction=0
        extractionMin=9999999999
        extractionMax=0

        if [ $sub != Local$top.sh ]; then
           echo $sub
           cd $sub
           rm Results.txt
           echo sub:`pwd`

           files=`ls Client*.log`

           for INFILE in $files
           do
              echo $INFILE

              lines=`wc -l $INFILE`
```

```
    exec < $INFILE
        #Skip to third line
        read LINE
        read LINE

        #Get query time
        read LINE
        first=$LINE
        read LINE
        second=$LINE
        if [ $first -ge $second ]; then
            let second=$second+86400000
        fi
        let diff=$second-$first

        let query=$query+$diff
        if [ $diff -ge $queryMax ]; then
            queryMax=$diff
        fi

        if [ $diff -le $queryMin ]; then
            queryMin=$diff
        fi

        #Get extraction time
        read LINE
        first=$LINE
        read LINE
        second=$LINE
        if [ $first -ge $second ]; then
            let second=$second+86400000
        fi
        let diff=$second-$first
        let extraction=$extraction+$diff
        if [ $diff -ge $extractionMax ]; then
            extractionMax=$diff
        fi

        if [ $diff -le $extractionMin ]; then
            extractionMin=$diff
        fi

done
echo $query
```

```
INFILE=`ls Server*.log`
echo $INFILE
lines=`wc -l $INFILE`
firstLines=0

process=0
processMax=0
processMin=999999999

exec < $INFILE
while read LINE
do
    if [ $firstLines = 0 ]; then
        echo Skip
        #Skip to third line
        read LINE
        read LINE
        firstLines=1
    fi

    first=$LINE
    read LINE
    second=$LINE
    if [ $second -le $first ]; then
        let second=$second+86400000
    fi

    let diff=$second-$first
    let process=$process+$diff

    if [ $diff -ge $processMax ]; then
        processMax=$diff
    fi

    if [ $diff -le $processMin ]; then
        processMin=$diff
    fi
done

sampleSize=20
let process=$process/$sampleSize
let query=$query/$sampleSize
let extraction=$extraction/$sampleSize

echo $query > Results.txt
echo $queryMin >> Results.txt
```

```
            echo $queryMax >> Results.txt
            echo $process >> Results.txt
            echo $processMin >> Results.txt
            echo $processMax >> Results.txt
            echo $extraction >> Results.txt
            echo $extractionMin >> Results.txt
            echo $extractionMax >> Results.txt
            let totalAvg=$query+$extraction+$process
            echo $totalAvg >> Results.txt
            cd ..
        fi

    done

    cd ..
    fi
done
```

Bibliography

[Aso01] D. Asonov. Private information retrieval. In *GI Jahrestagung*, pages 889–894, 2001.

[Aso03] D. Asonov. *Querying Databases Privately*. PhD thesis, Humboldt University Berlin, 2003.

[BDF00] F. Bao, R. H. Deng, and Peirong Feng. An efficient and practical scheme for privacy protection in the e-commerce of digital goods. In *ICISC*, pages 162–170, 2000.

[Bea97] D. Beaver. Commodity-based cryptography. In *Proc. of the 29th ACM Sym. on Theory of Computing*, pages 446–455, 1997.

[Ben87] J. D. C. Benaloh. *Verifiable secret-ballot elections*. PhD thesis, 1987.

[BFG03] R. Beigel, L. Fortnow, and W. Gasarch. A nearly tight lower bound for private information retrieval protocols. Technical Report TR03-087, Electronic Colloqium on Computational Complexity, 2003.

[BS02] A. Beimel and Y. Stahl. Robust information-theoretic private information retrieval. In *3rd Conference on Security in Communications Networks*, pages 326–341. 3rd Conference on Security in Communication Networks, 2002.

[Bur02] David M. Burton. *Elementary Number Theory*. McGraw-Hill Higher Education, 5th edition, 2002.

[BYT] A. Beimel, Y.Ishai, and T.Malkin. Reducing the servers' computation in private information retrieval: Pir with preprocessing. Journal of Cryptography. To appear. Prelim version was in CRYPTO00.

[Cas] A. Castner. Survey of single database private information retrieval.

[CG00] B. Chor and N. Gilboa. Computationally private information retrieval. In *32th ACM Sym. on Theory of Computing*. 32nd ACM Sym on Theory of Computing, 2000.

[CGH98] R. Canetti, O. Goldreich, and S. Halevi. The random oracle methodology, revisited (preliminary version). In *STOC '98: Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 209–218. ACM Press, 1998.

[CGN] B. Chor, N. Gilboa, and M. Naor. Private information retrieval by keywords. Unpublished manuscript.

[Cha04] Y-C Chang. Single database private information retrieval with logarithmic communication. Cryptology ePrint Archive, Report 2004/036, 2004. http://eprint.iacr.org/.

[CKGS98] B. Chor, E. Kushilevitz, O. Goldreich, and M. Sudan. Private information retrieval, 1998.

[DFKR97] D.Beaver, J. Feigenbaum, J. Killian, and P. Rogaway. Locally random reductions: Improvements and applications. In *Journal of Cryptology*, volume 10, 1997.

[DJ01] I. Damgård and M. Jurik. A generalisation, a simplification and some applications of paillier's probabilistic public-key system. In *PKC '01: Proceedings of the 4th International Workshop on Practice and Theory in Public Key Cryptography*, pages 119–136. Springer-Verlag, 2001.

[DJ03] I. Damgård and M. Jurik. A length-flexible threshold cryptosystem with applications, 2003.

[DMO00] G. DiCrescenzo, T. Malkin, and R. Ostrovsky. Single database private information retrieval implies oblivious transfer, 2000.

[DYO01] G. DiCrescenzo, Y.Ishai, and R. Ostrovsky. Universal service-providers for private information retrieval, 2001.

[Gam85] T. El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Proceedings of CRYPTO 84 on Advances in cryptology*, pages 10–18, New York, NY, USA, 1985. Springer-Verlag New York, Inc.

[GGM98] Y. Gertner, S. Goldwasser, and T. Malkin. A random server model for private information retrieval or information theoretic pir avoiding database replication. In *2nd RANDOM*, 1998.

[GM84] S. Goldwasser and S. Micali. Probabilistic encryption, 1984.

[Gol] O. Goldreich. Foundations of cryptography, volume 2. http://www.wisdom.weizmann.ac.il/%7Eoded/PSBookFrag/prot.ps.

[Gol01] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2001.

[GYKM00] Y. Gertner, Y.Ishai, E. Kushilevitz, and T. Malkin. Protecting data privacy in private information retrieval schemes, 2000.

[IL89] R. Impagliazzo and M. Luby. One-way functions are essential for cryptography. In *30th IEEE Sym, on Found. of Comp. Sci.*, pages 230–235. IEEE Computer Society Press, 1989.

[Ito99] T. Itoh. Efficient private information retrieval. In *IEICE Trans. Fundamentals*, volume ES2-A(1), 1999.

[Jur03] M. Jurik. Extensions to the paillier cryptosystem with applications to cryptological protocols, 2003.

[KBS02] D. Kesdogan, M. Borning, and M. Schmeink. Unobservable surfing on the world wide web: Is private information retrieval and alternative to the mix aproach? In *2nd Workshop on Privacy Enhancing Technologies*, volume PET2002, 2002.

[KdW03] I. Kerenidis and R. de Wolf. Exponential lower bound for 2-query locally decodable codes. In *Proceedings of the 35th ACM Sym. on Theory of Computing*, pages 106–115, 2003.

[Kil88] J. Killian. Basing cryptography on oblivious transfer. In *STOC*, volume 20-31, 1988.

[KO97] E. Kushilevitz and R. Ostrovsky. Replication is not neeeded: Single database, computationally-private information retrieval. In *38th IEEE Sym. on Found. of Comp. Sci.*, pages 364–373, 1997.

[KO00] E. Kushilevitz and R. Ostrovsky. One-way trapdoor permutations are sufficient for non-trivial single-server privae information retrieval. In *EUROCRYPT00*, 2000.

[Lin] C. Lin. Survey of private information retrieval systems.

[Lip04] H. Lipmaa. An oblivious transfer protocol with log-squared communication. Technical Report 2004/063, International Association for Cryptologic Research, February 25 2004.

[Mal00] T. Malkin. *A Study of Secure Database Access and General Two-Party Computation.* PhD thesis, Massachusetts Institute of Technology, 2000.

[Man98] E. Mann. *Private Access to Distributed Information.* PhD thesis, Technion - Israel Institute of Technology, Haifa, 1998.

[McE78] R. McEliece. A public-key cryptosystem based on algebraic coding theory, 1978.

[MH88] R. Merkle and M. Hellman. Hiding information and signatures in trapdoor knapsacks, 1988.

[MI88] T. Matsumoto and H. Imai. Public quadratic polynomial-tuples for efficient signature-verification and message-encryption, 1988.

[Mis00] S. Mishra. *Symmetrically Private Information Retrieval*. PhD thesis, Indian Statistical Institute, Calcutta, 2000.

[MVO96] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[NP99] M. Naor and B. Pinkas. Oblivious transfer and polynomial evaluation. In *31st ACM Sym. on Theory of Computing*, 1999.

[NP01] M. Naor and B. Pinkas. Efficient oblivious transfer protocols. In *SODA '01: Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 448–457. Society for Industrial and Applied Mathematics, 2001. http://www.pinkas.net/PAPERS/effot.ps.

[NS97] D. Naccache and J. Stern. A new public-key cryptosystem, 1997.

[NS98a] D. Naccache and J. Stern. A new public key cryptosystem based on higher residues, 1998.

[NS98b] P. N'Guyen and J. Stern. Cryptanalysis of the ajtai-dwork cryptosystem, 1998.

[NW94] N. Nisan and A. Wigderson. Hardness vs randomness, 1994.

[oC88] R Conf. on Crypto. Equivalent between two flavors of oblivious transfer. In *Proc. of the 8th IAC*, volume 403, 1988.

[OU98] T. Okamoto and S. Uchiyama. A new public-key cryptosystem as secure as factoring. In *Eurocrypt '98, LNCS 1403*, pages 308–318, 1998.

[Pai99] P. Paillier. Public-key cryptosystems based on composite degree residuosity classes. In *EUROCRYPT*, pages 223–238, 1999.

[Pat97] J. Patarin. The oil and vinegar algorithm for signatures, 1997.

[Rab81] M. Rabin. How to exchange secreats by oblivious transfer. Technical report, Aiken Computation Laboratory, Harvard University, 1981.

[Ste98] J. P. Stern. A new efficient all-or-nothing disclosure of secrets protocol. In *ASIACRYPT '98: Proceedings of the International Conference on the Theory and Applications of Cryptology and Information Security*, pages 357–371, London, UK, 1998. Springer-Verlag.

[Sti02] Douglas R. Stinson. *Cryptography: Theory and Practice*. CRC Press, 2nd edition, 2002.

[Vau98] S. Vaudenay. Cryptanalysis of the chor-rivest cryptosystem, 1998.

[Yao90] A. Yao. An application of communication complexity to cryptography. In *Lecture DIMACS Worhshop on Structural Complexity and Cryptography*, 1990.