國 立 交 通 大 學

電信工程研究所

碩 士 論 文

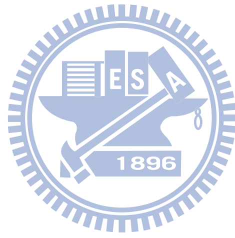
應用同形保密技術於社群適地性行動雲端服務之研究

Preserving Privacy for Community Location-Based Mobile Cloud Applications Using Homomorphic Encryption

研究生：魯旺

指導教授：王蒞君 教授

中 華 民 國 100 年 2 月 15 日



Preserving Privacy for Community Location-Based Mobile Cloud Applications Using Homomorphic Encryption

Student: W.K.Ruwan Indika Prasanna

Advisor: Prof. Li-Chun Wang

A THESIS

Submitted to Graduate Institute of Communications Engineering

College of Electrical and Computer Engineering

National Chiao Tung University

in Partial Fulfillment of the Requirements

for the Degree of

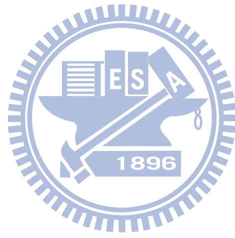
Master of Science

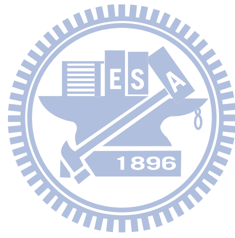
in

Communications Engineering

Hsinchu, Taiwan, Republic of China

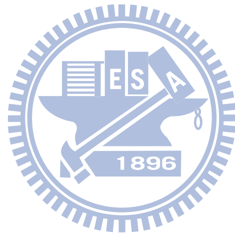
February 15, 2011





Abstract

Cloud computing provides computing infrastructures and applications as a service. Since the infrastructure of cloud computing is hosted by a third party, security has become a major concern. In this thesis we discuss the security issues of hosting servers for a location based social network service in the cloud computing infrastructure. The limitation of traditional encryption is that once data is encrypted, other processing cannot be done without decrypting the cipher text. This results in a conflict of interest in cloud computing environments. Users require cloud computing infrastructures to perform data processing, but they do not trust the cloud computing infrastructure with their sensitive information. A location sharing protocol, which permits server side calculations on encrypted location information, is proposed in this thesis. Based on the XOR homomorphic encryption technique, the proposed scheme allows the users to send their location as the cipher and obtain location based services without revealing location information to the cloud computing service providers.

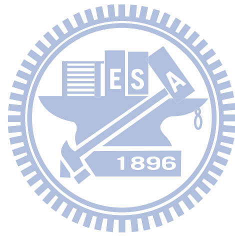


Acknowledgements

I am heartily thankful to my supervisor Prof. Li-Chun Wang for providing me insight into an important research problem, encouragement and kind support throughout the master program. This work would not have been possible without his advice and guidance.

I would like to thank my laboratory mates in the cloud computing research lab for their support and sharing their ideas with me. I would like to extend my great gratitude to Hitron Technologies for the support throughout my studies. I would like to thank my family, even though they are faraway, they have been a big source of motivation for me to continue on this path.

Lastly, I offer my regards to those who supported me in any respect during the completion of this thesis.

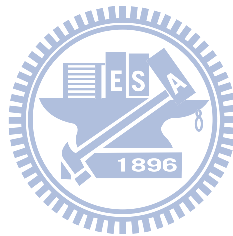


Contents

Abstract i

Acknowledgements iii

List of Tables xii



List of Figures xv

List of Abbreviations xvii

1 Introduction 1

1.1 Restrictions of Current Encryption Techniques 3

1.2 Homomorphic Encryption 4

1.3 Problem and Solution 5

1.4 Thesis Outline 7

2 Background 9

2.1 Existing Approaches for Preserving Privacy of Location Data . 10
2.1.1 Query over Encrypted Data 10
2.1.2 Distributed Architecture for Secure Database Services . 14
2.1.3 Order Preserving Encryption 16
2.1.4 Private Information Retrieval 18
2.1.5 Anonymization in Proactive Location-Based Community Services 18
2.1.6 Privacy Protection in Location-Based Services Through Public-Key Privacy Homomorphism 20
2.1.7 Multiplicative Homomorphism in RSA Algorithm . . . 21
2.1.8 Comparison Table 24
2.2 Comparison of Different Cryptographic Schemes 26
2.2.1 Symmetric Key Encryption 26
2.2.2 Asymmetric Key Encryption 27
2.2.3 One-Way HASH Functions 28
2.2.4 Digital Signatures 29
2.2.5 Homomorphic Encryption 30
2.2.6 Comparison of Security Properties of Encryption Schemes 31
3 System Model and Problem Formulation 33
3.1 System Model 33

3.1.1 Model 33

3.1.2 Assumptions 34

3.2 Problem Formulation 35

3.2.1 Goal 35

3.2.2 Overview of the Proposed XOR Homomorphic Encryption Secure Location Sharing Scheme 36

4 XOR Homomorphic Encrypted Location Sharing Scheme 41

4.1 Traditional Location Sharing Method 41

4.2 Introduction to GDHV's Fully Homomorphic Encryption . . . 42

4.3 Considerations of Selecting Homomorphic Encryption Algorithm for LBS 45

4.3.1 XOR Homomorphic Encryption Algorithm 45

4.3.2 Other Consideration in Selecting the Homomorphic Encryption Algorithm 47

4.4 Proposed XOR Homomorphic Encrypted Secure Location Sharing Protocol 51

4.4.1 Detailed Protocol Steps 51

4.4.2 Illustrative Example Using Pseudo Location Data . . . 52

4.4.3 Illustrative Example Using GPS Location Data 54

5 Security Enhancements for XOR Homomorphic Encrypted

Location Sharing Scheme **57**

5.1 RSA Digital Signature Scheme 57

5.2 Group Key Distribution 59

5.3 Protocol Security Analysis 60

5.3.1 Cryptography *vs* Security Protocols 60

5.3.2 Dolev-Yao Model 60

5.3.3 Security Verification of the Proposed Protocol using AVISPA Tool 61

5.3.4 Security Verification of the Proposed Protocol using ProVerif Tool 65

6 Performance Issues of XOR Homomorphic Encrypted Secure Location Sharing Protocol **69**

6.1 Concept Verification of XOR Homomorphic Encryption Algorithm by MATLAB 69

6.2 Implementation of the Proposed Protocol in Android Platform 73

6.2.1 Implementation at the Client Side 73

6.2.2 Implementation at the Server Side 83

6.3 Cipher-Text Growth Issue 86

6.4 Time Complexity Issue 89

6.5 Bandwidth Issue 93

7 Conclusions 95

7.1 XOR Homomorphic Encrypted Secure Location Sharing Protocol . . . 95

7.2 Suggestions for Future Research . . . 96

Bibliography 99

Appendices 109

Appendix A Proposed Protocol in HLPSL 111

Appendix B Details of the Verification Result using AVISPA 117

B.1 OFMC . . . 117

B.2 CL-AtSe . . . 118

B.3 CL-AtSe . . . 119

B.4 TA4SP . . . 120

Appendix C Proposed Protocol in Spi-Calculus 123

Appendix D Verification Result using PROVERIF 127

Appendix E MATLAB Simulation Code 131

E.1 Encryption Function . . . 131

E.2 Main Function . . . 132

Appendix F Android Mobile Client JAVA Code **135**

F.1 Main Class 135

F.2 Itemized Overlay Class 148

F.3 Homomorphic Encryption Class 150

F.4 JAVA Socket Class 152

F.5 Homomorphic Decryption Class 154

F.6 Two's Complement to Integer Class 157

F.7 XOR Operation Class 158

F.8 RSA Digital Signature Class 159

Appendix G Server JAVA Code **163**

G.1 Main Class 163

G.2 Homomorphic Addition Class 170

List of Tables

2.1 Query over encrypted data Table 1 12

2.2 Query over encrypted data Table 2 12

2.3 Query over encrypted data Table 3 13

2.4 Query over encrypted data Table 4 13

2.5 Order preserving encryption Table 1 16

2.6 Comparison of the existing location privacy protection. 25

2.7 Characteristics of the security properties of cryptographic algorithms. 32

4.1 Desired characteristics of the secure community LBS encryption algorithm. 47

4.2 Computational complexity of asymmetric vs symmetric encryption. 49

5.1 Formal approach vs cryptographic approach of protocol verification. 61

6.1 Server computation reduction in the proposed protocol. 91

6.2 Bandwidth overhead reduction of two-user case using the proposed protocol. 94

List of Figures

1.1 Traditional location sharing protocol. 5

1.2 XOR homomorphic encrypted secure location scheme. 7

2.1 Distributed architecture for secure database services. 14

2.2 Order preserving encryption. 17

2.3 Private information retrieval. 19

2.4 Privacy protection in LBS through public-key privacy homomorphism. 23

2.5 Symmetric key encryption. 27

2.6 Asymmetric key encryption. 28

2.7 One way hash function. 29

2.8 Digital signature. 30

2.9 Homomorphic encryption. 31

3.1 Application overview 34

3.2 Group key distribution. 35

3.3 The proposed XOR homomorphic encrypted secure location sharing protocol for the two-user case. 37

3.4 Flow of the two-user case. 39

3.5 Flow chart of using XOR homomorphic encryption for location data exchange in the n-user case. 40

4.1 Traditional location sharing mechanism. 42

4.2 Locations known by each user before location sharing 52

4.3 Locations known by each user after location sharing 53

5.1 Client digital signature generation and verification. 58

5.2 Group key distribution. 59

5.3 Dolev-Yao model. 61

5.4 Protocol written in HLPSL animated using the SPAN tool. . . 63

5.5 Result of verifying the proposed protocol using AVISPA tool. . 65

6.1 MATLAB simulation flow chart for the XOR homomorphic encryption in community LBS. 70

6.2 The Android client programme flow. 74

6.3 Android application GUI 75

6.4 Client RSA digital signature generation and verification. . . . 80

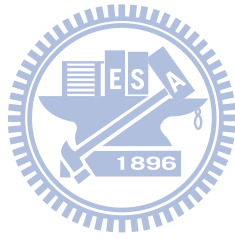
6.5 The server programme flow. 84

6.6 Encryption key size vs cipher-text growth. 87

6.7 Number of users vs cryptographic operations at the server. . . 92

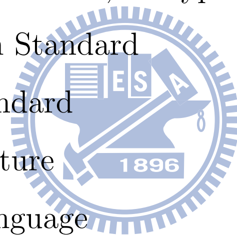
6.8 Number of users vs server bandwidth utilization. 94

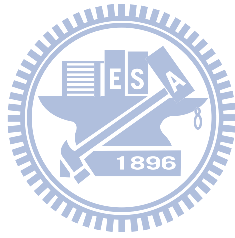
7.1 Using network coding for security. 98



List of Abbreviations

LBS	Location Based Service
GPS	Global Positioning System
POI	Points Of Interest
RSA	Rivest, Shamir and Adleman, encryption algorithm
AES	Advanced Encryption Standard
DES	Data Encryption Standard
PKI	Public Key Infrastructure
SQL	Structured Query Language
PLBCS	Proactive Location Based Community Services
API	Application Programming Interface
TCP	Transmission Control Protocol
IP	Internet Protocol
GUI	Graphical User Interface
SDK	Software Development Kit
IDE	Integrated Development Environment
SHA	Secure Hash Algorithm





Chapter 1

Introduction

Cloud computing is the network-based computing where shared servers provide resources including software, platform and infrastructure as a service to computers and other devices on demand. Cloud computing is an evolution of virtualization. Although the industry has started selling cloud computing products, a multitude of research challenges are present in various areas. One of the key concerns is the data security of organisations which outsource their computing and data storage to cloud computing service providers. Most organizations are hesitant to outsource their data, because data outsourced to cloud computing infrastructure could be located anywhere in the world, in computers which are shared by many other users and organizations. Even though the cloud computing service may be provided by a reputed company, there is no guarantee that the local data center administrators will not misuse the access to users' sensitive information.

Location-based services (LBS) are booming with the increase in popularity of smartphones with built-in location sensing hardware like global

positioning system (GPS) receivers. In general, someone's location is an extremely sensitive piece of information, which usually is not shared with an unknown person. With newly launched social network applications, users are able to share their locations with their friends and at the same time obtain services from LBS providers in the Internet. In this thesis, we discuss the security issue for providing location-based social network services in cloud computing infrastructure. In this scenario, location sharing among friends and points of interest (POI) searches etc, are expected to be performed by the virtual machines which are hosted in the untrusted cloud computing infrastructure. The widely accepted data secrecy solution is encryption. However, once data is encrypted processing cannot be performed without decrypting the cipher-text. The users require the cloud computing to process the location data, but the cloud computing infrastructure may not be totally trustable. This results in a dilemma in proving LBS in cloud computing.

In this research we focus on developing a secure protocol based on the homomorphic encryption technique, which can assist smartphone users to securely share their locations with other users in online location-based community services. Our proposed homomorphic encryption scheme for mobile clients can relieve the heavy computation issue of fully homomorphic encryption schemes [1, 2].

1.1 Restrictions of Current Encryption Techniques

Traditional encryption algorithms are capable of providing data secrecy in transition and in storage. If cloud computing is not used, all the data analysis and processing can be done in secured local information system facilities. There was no major security concern in decrypting sensitive information and processing those locally. Since cloud computing shifts the paradigm of possessing own local information systems infrastructure, more and more organizations are looking for the third parties who will provide computation and storage capacity. The security problem arises when data storage and processing are outsourced. Corporates would like their data to be stored and processed by a third-party service provider, but they are not willing to reveal confidential information to the service provider. Data encryption will hide the data, but it makes it impossible for the cloud computing service providers to perform computations on the data. This defeats the advantage of cloud computing. A number of researches have been done on developing techniques to query and search encrypted data [3–7], but all these methods introduce significant network overheard and requires trusted computing infrastructure in order to filter out and obtain the final result. One of the most promising solutions for the security problem in cloud computing is homomorphic encryption technique [1, 2], which will be discussed in the next section.

1.2 Homomorphic Encryption

The idea of fully homomorphic encryption has been discussed since 1978 with proposal of the Rivest, Shamir and Adleman (RSA) asymmetric key encryption scheme [8] which is multiplicatively homomorphic. However, developing such a scheme has seemed obscure for more than three decades [9] because it was uncertain whether fully homomorphic encryption is even a possibility. In 2006 Dan Boneh, Eu-Jin Goh, Kobbi Nissim [10] proposed a scheme which permitted evaluation of multiple additions but only one multiplication. In year 2009, Craig Gentry using lattice-based cryptography [1] revealed the first fully homomorphic encryption scheme. Gentry's scheme is impracticable for most applications with current technology because the cipher text size and the computation time increase drastically if the security level needs to be increased. Following the first fully homomorphic encryption scheme, Craig Gentry, Marten Van Dijk, Shai Halevi and Vinod Vaikuntanathan [2] presented a second fully homomorphic encryption scheme, which use many aspects of Gentry's construction without requiring ideal lattices. They demonstrated that the homomorphic portion of Gentry's encryption algorithm can be interchanged with a simple encryption algorithm based on integers. The scheme in [2] is simpler in concept than Gentry's ideal lattice based encryption algorithm, but has similar homomorphic encryption's property. We will implement the symmetric homomorphic scheme in reference [2] for the community location-based mobile cloud computing environment.

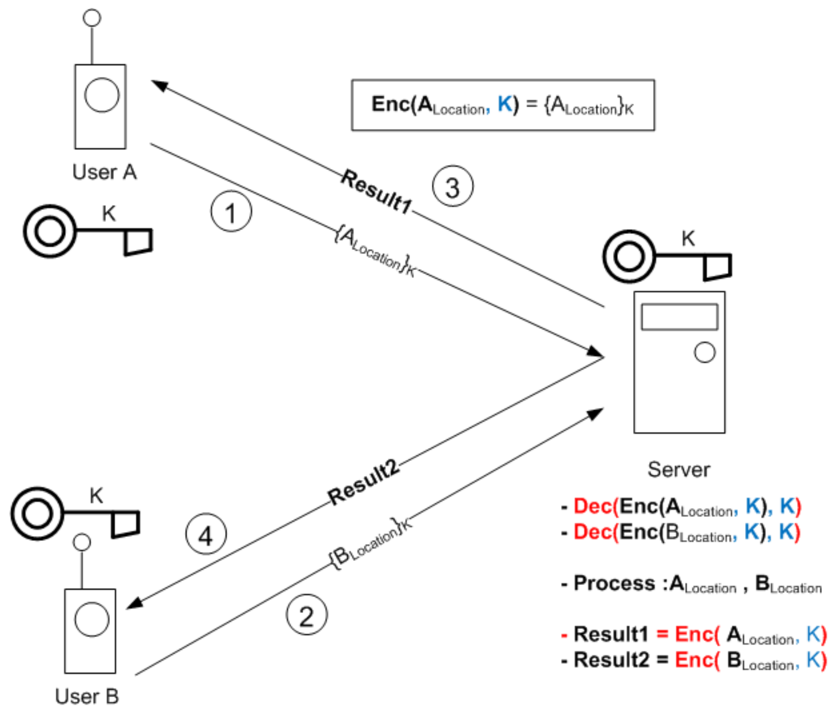


Figure 1.1: Traditional location sharing protocol.

1.3 Problem and Solution

Most traditional location sharing protocols are designed with a trusted service provider. Therefore, these protocols did not have a mechanism to hide user's location from the service provider. As expressed in Fig. 1.1, the service provider can decrypt the location information, process and send it back to the user. In the cloud computing paradigm, the service provider may not be totally trusted. Location information need to be encrypted before sending to the server. Then the server cannot process the encrypted data to provide LBS.

In this thesis, we have developed **"XOR Homomorphic Encrypted Secure Location Sharing Protocol"**, which can assist mobile clients to

securely share locations with other mobile devices. The protocol is graphically expressed in Fig. 1.2. Our protocol permits the server to perform a simple XOR mathematical operations on the cipher text and can achieve the goal of location information exchange. In the proposed protocol, the user location data is encrypted with a secret key among users before sending to the service provider. The location data are processed by the server in the encrypted form while providing location sharing services to a group of users. The two main concepts used in our scheme are homomorphic encryption and the concept of network coding based on the XOR operation [11]. For example, if there are two users, they can send their locations encrypted to the social network server. The server adds user$A$'s location and user$B$'s location together $(\{A_{Location}\}_K + \{B_{Location}\}_K)$ and sends the result back to both users. In homomorphic domain, addition results in a bitwise XOR in the plain text $(A_{Location} \oplus B_{Location})$. User A XOR the result of $(\{A_{Location}\}_K + \{B_{Location}\}_K)$ and obtain the user B's location, and vice versa.

Our proposed protocol is light enough for a mobile client to perform, which can provide confidentiality, integrity, authentication and non-repudiation in the cloud computing environment. Further more, it can reduce communication bandwidth by broadcasting the XOR results of two user locations.

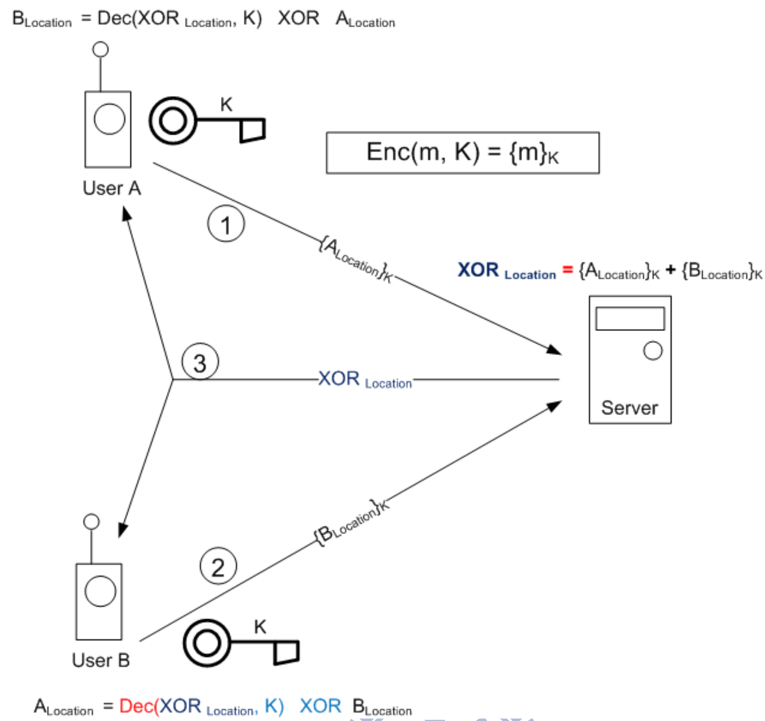
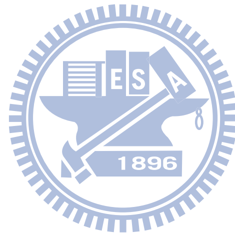


Figure 1.2: XOR homomorphic encrypted secure location scheme.

1.4 Thesis Outline

The rest of the thesis is organized as follows. Chapter 2 is a literature survey on existing solutions for the problem of processing encrypted data and secure location sharing. We also give a survey on the properties of different tools used in cryptography. Chapter 3 describes the system model and problem formulation. In Chapter 4 we discuss the proposed "XOR Homomorphic Encrypted Secure Location Sharing Protocol". Chapter 5 presents the RSA digital signature scheme, key distribution and the security analysis of the proposed protocol using secure transaction protocol analysis tools. In Chapter 6, the implementation issues of the proposed protocol in an Android [12] application are reported. Finally, Chapter 7 gives the concluding remarks

and suggests some future research topics.



Chapter 2

Background

Many research works have been done on the subject of processing and preforming calculations over the cipher-text. Homomorphism in encryption algorithms like RSA was identified over three decades ago [9]. The limiting factor of using homomorphic encryption as a solution for all applications is the level of homomorphism in the existing schemes. For example, RSA is only multiplicatively homomorphic. Therefore, not all the mathematical operations can be performed for homomorphic encryption. Since outsourcing databases to third party service providers become popular, the issue of performing structured query language (SQL) queries over encrypted data has been reported in [3]. Many researchers have proposed methods to perform queries over the encrypted databases. Some of the methods will be overviewed in this chapter.

Both corporates and individual customers are concerned about privacy. As a solution to the problem of user privacy, "Private Information Retrial" method was proposed by Rafail-Victor [13]. Location privacy becomes a

major concern with the popularity of social network services and location-based services (LBS). Users are concerned about revealing their location to untrusted parties. For this particular issue, most proposed solutions point to anonymizing the user's location by diluting accuracy of the location provided to the service providers. Here we give an overview on the technologies for secure LBS [14–16].

2.1 Existing Approaches for Preserving Privacy of Location Data

2.1.1 Query over Encrypted Data

Hosting a database in cloud computing infrastructure presents many challenges to database security. Corporates hold their most valuable assets in databases. When enterprises deploy databases in cloud computing virtual environments, they need to face the risk of exposing highly-sensitive data to a broad base of internal and external threats. For secrecy, data is encrypted before being stored in the cloud computing infrastructure. Unfortunately, due to the nature of encryption, virtual machines are unable to perform queries and other operation on encrypted data. Hakan-Bala-Chen [3] described a method to perform querying over encrypted data. This technique relies on sharing the computation over client and server. The clients stores encrypted data and meta data in the database. The database can only use metadata for query processing due to the fact that the actual data is encrypted. The server performs queries over meta data and produces a super set of the results and communicates those data to the client. The client de-

crypts all the received data and locates the required record. Tables 2.1 $\sim$ 2.4 show the example of encrypted data querying.

Table 2.1 shows the salary of employees before encryption. The Table 2.2 shows the salary range split in to partitions and assigned a randomly generated index (metadata). In the Table 2.3, all the data are encrypted and stored in the database with metadata. The human resource manager would like to search the database for employees whose earnings are USD 150K. This value is in the range of (140-160]. The manager sends a query as, "SQL $>$ SELECT name FROM Table WHERE metadata = 11" and receives two rows as in the Table 2.4(a). The manager decrypts the received cipher-text as represented in Table 2.4(b). In Table 2.4(c), the manager filters out the unnecessary rows and obtains the required information.

This method is suitable for preserving privacy in outsourced databases. Since the exact search result cannot be obtained in one transaction, the network utilizations is significantly higher than normal.

Table 2.1: Employee salary before encryption.

ID	Name	Salary	Position
1	Employee 1	110K	Manager
2	Employee 2	130K	CTO
3	Employee 3	150K	Engineer
4	Employee 4	1600K	Accountant
5	Employee 5	180K	Engineer

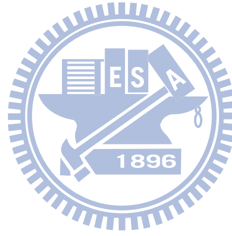


Table 2.2: Salary partitioning.

Range	Index
[100-120]	2
(120-140]	5
(140-160]	11
(160-180]	9
(180-200]	7

Table 2.3: Employee salary after encryption, where “XXXXXX” indicates cipher-text.

ID	**Name**	**Salary**	**Position**	**Metadata**
1	XXXXX	XXXXX	XXXXX	2
2	XXXXX	XXXXX	XXXXX	5
3	XXXXX	XXXXX	XXXXX	11
4	XXXXX	XXXXX	XXXXX	11
5	XXXXX	XXXXX	XXXXX	9

Table 2.4: Result of the query.

(a) Encrypted query result

ID	**Name**	**Salary**	**Position**	**Metadata**
3	XXXXX	XXXXX	XXXXX	11
4	XXXXX	XXXXX	XXXXX	11

(b) Query result after decryption

ID	**Name**	**Salary**	**Position**
3	Employee 3	150K	Engineer
4	Employee 4	1600K	Accountant

(c) The final result

ID	**Name**	**Salary**	**Position**
3	Employee 3	150K	Engineer

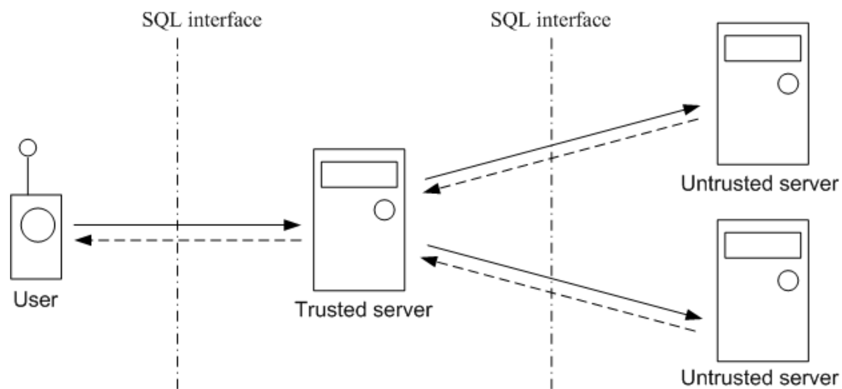


Figure 2.1: Distributed architecture for secure database services.

2.1.2 Distributed Architecture for Secure Database Services

The general architecture of a distributed secure database service is illustrated in Fig. 2.1 consisting of a trusted server as a mediator and two untrusted servers that provide database services. The database servers provide reliable content storage and data management, but are not trusted to preserve confidentiality [4].

There are different techniques to partition data across the two servers with distributed architectures. It should be a lossless decomposition where it is possible to reconstruct the original data using only the contents in the two servers. Some methods of data partitioning are given below.

1. One-time pad: $a_1 = a \oplus r\,; a_2 = r$, where r is a random value.

2. Deterministic encryption: $a_1 = E(a; k); a_2 = k$, where E is a deterministic encryption.

3. Random addition: $a_1 = a + r; a_2 = r$, where r is a random number

drawn from a domain much larger than that of a.

4. Paulo-Lusa-Tiago-Joo-Muriel [5] has proposed coding as a means to achieve a prescribed level of confidentiality by using the algebraic structure of Vandermonde matrix. The Vandermonde matrix is used for splitting input blocks before they are stored in different locations. This scheme ensures that adversary whom only has access to one of the two networks is not able to extract any symbol, even if they succeed in guessing some of the blocks.

The user splits the file in to n blocks $b_1, b_2, ..., b_n$, and encodes each block. There are two networks E_1 and E_2. Components 1 $\sim$ k are stored in network E_1, and components $(k+1) \sim n$ are stored in network E_2.

$$[A_{i,j}] = (a_j^{(i-1)}) \text{ is a } n \times n \text{ Vandermonde matrix.} \tag{2.1}$$

The coefficients a_j are distributed over all nonzero elements of a finite field $\mathbb{F}_q, q = 2^u > n$.

$$\forall i, l \in \{1, ..., n\}, i \neq l \rightarrow a_i \neq a_l. \tag{2.2}$$

Let the original data, or plain-text, be a vector $b = (b_1, ..., b_n)^T$. Then, the encoded data vector is represented by

$$c = (c_1, ..., c_n)^T = Ab;$$

$$c_i = \sum_{j=1}^{n} a_j^{i-1} b_j; \tag{2.3}$$

To recover the original information, the legitimate user receives $n-k$ and k contiguous components of c from the networks E_1 and E_2 and preforms the operation in (2.4):

$$b = A^{-1}a. \tag{2.4}$$

Table 2.5: Order preserving encryption.

(a) Salary data before encryption.

Name	Salary
Employee 1	10
Employee 2	34
Employee 3	23
Employee 4	44
Employee 5	67
Employee 6	88
Employee 7	67
Employee 8	43
Employee 9	55

(b) Salary data after encryption.

Name	Salary
Employee 1	10105
Employee 2	1187015
Employee 3	59977285
Employee 4	2561605
Employee 5	3750037
Employee 6	280375
Employee 7	24014905
Employee 8	1187015
Employee 9	9837637

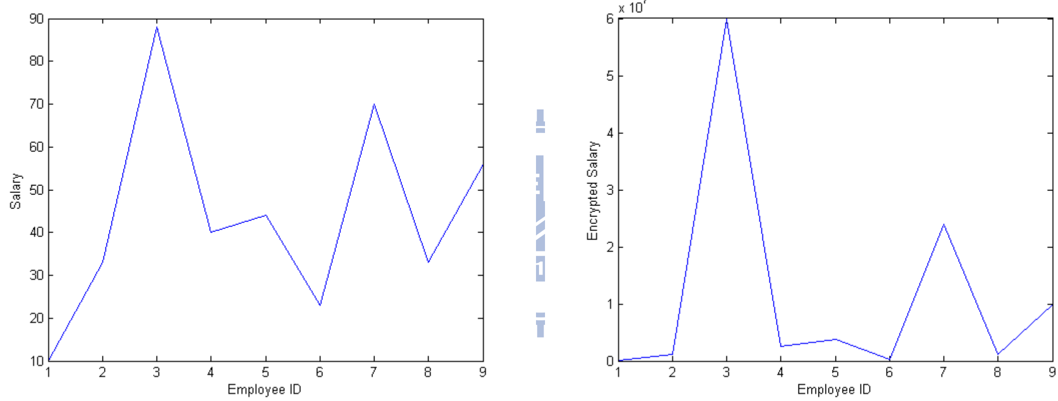
2.1.3 Order Preserving Encryption

The concept behind this scheme is to map data into another plane based on a polynomial [6] such as $y = x^3 + x^2 + 5$. Table 2.5(a) and Fig. 2.2(a) show the salary data and plot before encryption. Table 2.5(b) and Fig. 2.2(b) show the salary data and plot after encryption. The following SQL commands is an example of quarry over the encrypted data.

SQL > SELECT name FROM tablename WHERE Y=1187015 ;

SQL > query response : **Employee 2**

Figure 2.2: Order preserving encryption.



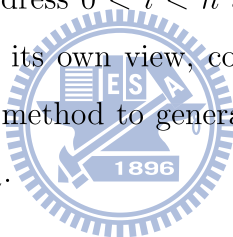
(a) Salary plot before encryption.

(b) Salary plot after encryption.

2.1.4 Private Information Retrieval

Fig. 2.3 shows the basic idea of private information storage. This approach efficiently and privately stores and retrieves data which is distributed and maintained in several databases, which do not communicate with one another. This minimizes the communication complexity while maintaining privacy. Thus each individual database does not get any information about the data or the nature of the users' queries [13].

The composite database maintains a bit vector of length n. There exists a "K" number of non-communicating constituent databases. User may perform a read operation, read(i), for a given address $0 < i < n$. Write operation write($i; b$), for a given address $0 < i < n$ and bit b belongs to $\{0, 1\}$. Each constituent database has its own view, consisting of the messages received from the user. A simple method to generate the bit string of length "n" is $D = D_1 \oplus D_2 \oplus ..... \oplus D_n$.



2.1.5 Anonymization in Proactive Location-Based Community Services

This method uses pseudonyms in conjunction with coordinate transformation. It is based on the idea that for proactive location based community services (PLBCS), the real position of a user is irrelevant. What is necessary for this kind of applications is the knowledge about relative spatial position of the community member or the player with respect to other members. This approach is suitable for "Buddy Tracking" and "Mobile Gaming" applications. It uses distance preserving coordinate transformations to hide user

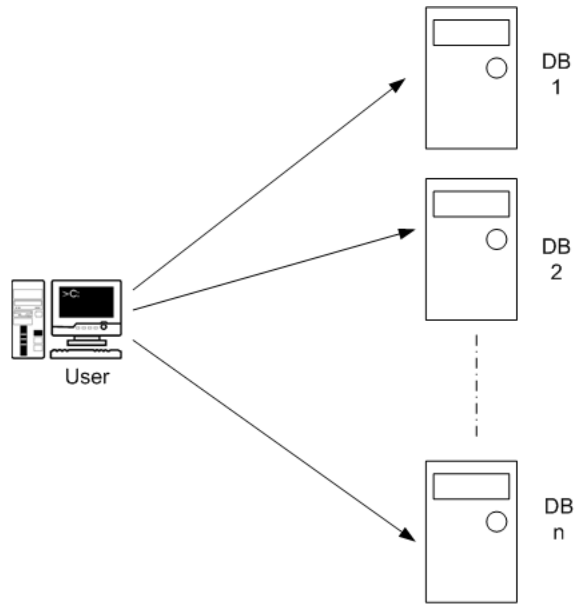


Figure 2.3: Private information retrieval.

locations [15].

Let user's actual location be,

$$(x_i, y_i) = (Long_i, Latt_i). \tag{2.5}$$

Consider a shared value among group members

$$N = Noise.$$

Then the user's location is mapped to a new plane

$$(\hat{x}_i, \hat{y}_i) = (Long_i + N, Latt_i + N). \tag{2.6}$$

2.1.6 Privacy Protection in Location-Based Services Through Public-Key Privacy Homomorphism

Agusti and Antoni [14] proposed a method for location privacy based on $k-$ *anonymity* approach. The method is to hide the position of the user within other users. This makes the user indistinguishable among $k-1$ other users. A mathematical discussion of the method is given below and a graphical representation is given in Fig. 2.4. In the figure $skUp$ is the secret(private) key of the LBS provider and $pkUp$ is the public key of the LBS provider.

- **First Step: All users mask their locations, sign the locations with LBS provider's public key $pklbs$, and send to user A on request**

N_i is Gaussian noise with null average $\sim N(0,\sigma)$. User A encrypts her own masked location with $pklbs$.

$$\text{Masked location } (\hat{x}_i, \hat{y}_i) = (x_i, y_i) + (N_i^x, N_i^y), \quad (2.7)$$

$$\text{Encrypted masked location } E_{pklbs}\{(\hat{x}_i, \hat{y}_i)\} = E_{pklbs}\{(x_i, y_i) + (N_i^x, N_i^y)\}. \quad (2.8)$$

- **Second Step: User A performs homomorphic addition on all the encrypted locations**

Let "Γ" be the mathematical operation on the cipher-text which corresponds to addition in the plain-text. i.e.,

$$\cup = (\Gamma_{i=1}^{k} E_{pklbs}\{\hat{x}_i\}, \Gamma_{i=1}^{k} E_{pklbs}\{\hat{y}_i\}). \quad (2.9)$$

Due to additively homomorphic property of the cryptography algorithm used, we can have

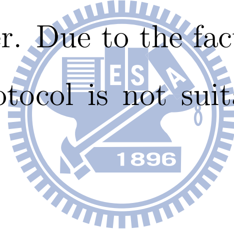
$$(\Gamma_{i=1}^{k} E_{pklbs}\{\hat{x}_i\}, \Gamma_{i=1}^{k} E_{pklbs}\{\hat{y}_i\}) = (E_{pklbs}\{\Sigma_{i=1}^{k}\hat{x}_i\}, E_{pklbs}\{\Sigma_{i=1}^{k}\hat{y}_i\}). \quad (2.10)$$

- **Third Step: User A send "∪" to the LBS provider's server**

The server decrypts the message using its private key *sklbs* and obtains $\Sigma_{i=1}^{k}\hat{x}_i$ and $\Sigma_{i=1}^{k}\hat{y}_i$. Then the server calculates the centroid as below.

$$(\bar{x}, \bar{y}) = (\frac{\Sigma_{i=1}^{k}\hat{x}_i}{k}, \frac{\Sigma_{i=1}^{k}\hat{y}_i}{k}). \quad (2.11)$$

Based on the accuracy diluted location of user A, the server provides location based services to the user. Due to the fact that the LBS server can decrypt users' locations, this protocol is not suitable for cloud computing environment.



2.1.7 Multiplicative Homomorphism in RSA Algorithm

The multiplicative homomorphic property of RSA was discovered shortly after the proposal of the RSA algorithm. Given blow is a mathematical explanation of this property [9].

Consider modulus

$$m = q \times p,$$

where p and q are prime number, and exponent. Then the encrypted message x_1 and x_2 are

$$Enc(x_1) = x_1^e \bmod m,$$

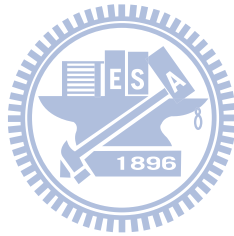
and

$$Enc(x_2) = x_2^e \bmod m,$$

where Enc() is the encryption function.

The multiplicatively homomorphic property of the RSA algorithm can be shown as follows:

$$\begin{aligned} Enc(x_1) \times Enc(x_2) &= (x_1^e \bmod m) \times (x_2^e \bmod m), \\ &= (x_1 \times x_2)^e \bmod m, \\ &= Enc(x_1 \times x_2). \end{aligned} \tag{2.12}$$



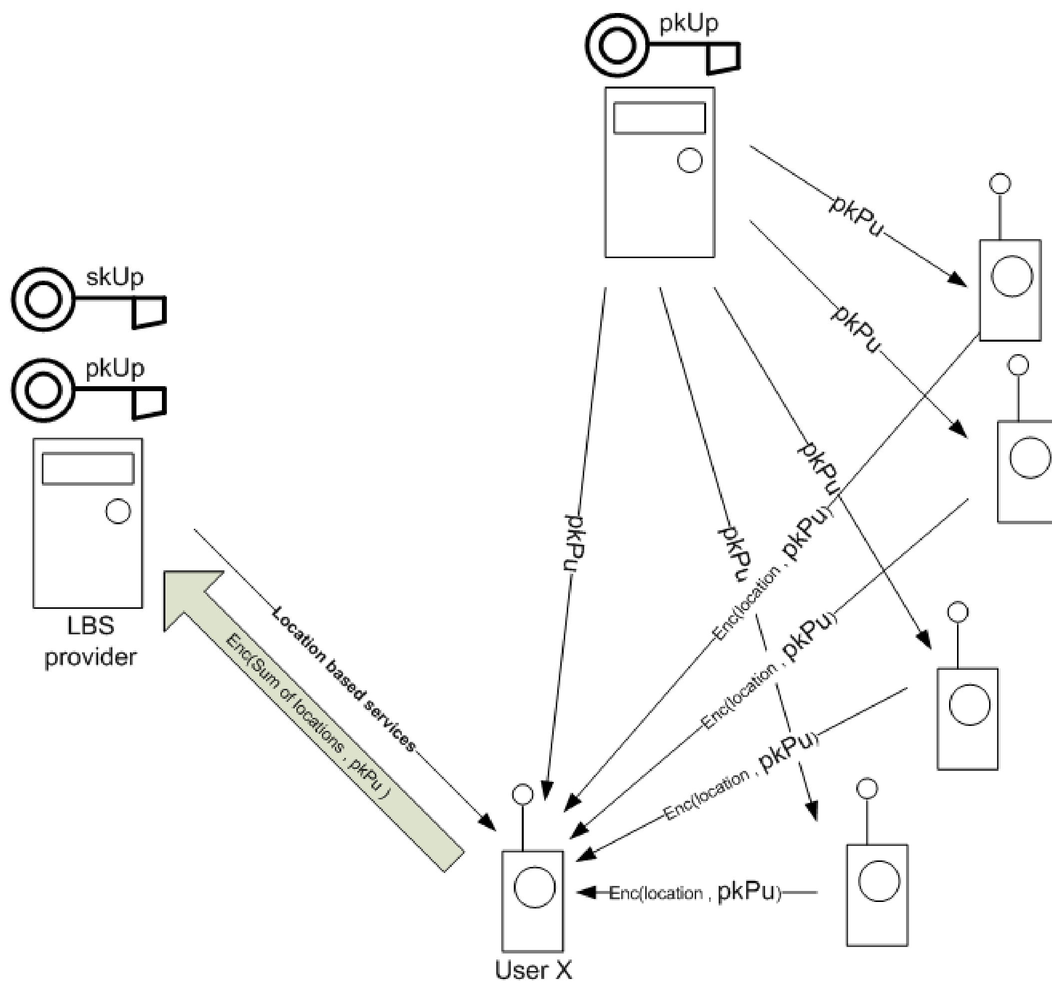


Figure 2.4: Privacy protection in LBS through public-key privacy homomorphism.

2.1.8 Comparison Table

In Table 2.6, all seven schemes which were discussed are compared with the proposed "XOR Homomorphic Encrypted Secure Location Sharing Protocol".

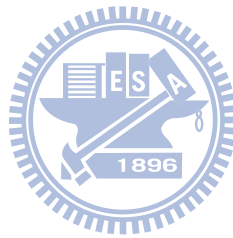


Table 2.6: Comparison of the existing location privacy protection.

.	Confidentiality	Integrity	Authenticity	Non-Repudiation	Homomorphism	Time complexity
Order Preserving Encryption [6]	**YES**	NO	NO	NO	NO	Degree of the polynomial - $O(n^k)$
Private Information Retrieval [13]	**YES**	NO	NO	NO	NO	XOR - $O(n)$, Symmetric encryption - $O(n), O(n^2)$
Privacy Protection in Location-Based Services Through a Public Key Privacy Homomorphism [14]	NO	**YES**	**YES**	**YES**	NO	Public key crypto - $O(n^2), O(n^3)$, Homomorphic multiplication - $O(n^2)$
Anonymization in Proactive Location Based Community Services [15]	**YES**	NO	NO	NO	NO	Addition - $O(n)$
Query over Encrypted Data [3]	**YES**	NO	NO	NO	NO	Symmetric encryption - $O(n), O(n^2)$
Distributed Architecture for Secure Database Services [4]	**YES**	NO	NO	NO	NO	Matrix multiplication - $O(n^3)$
Multiplicative Homomorphism in RSA Algorithmm [9]	NO	**YES**	**YES**	**YES**	PARTIAL	public key operations - $O(n^2)$, private key operations - $O(n^3)$

2.2 Comparison of Different Cryptographic Schemes

In cryptography, encryption is the process of transforming data in to unreadable bits with no significant statistical properties using an algorithm. The encrypted bits can be converted back to the original data using a secret value call the "key", which was used at the point of encryption. Certain algorithms like one-way HASH functions are not considered encryption, but posses other important properties like error detection capabilities. Different cryptographic algorithms are designed for different purposes. In this chapter we compare different cryptographic schemes and discuss their properties and applications [9].

2.2.1 Symmetric Key Encryption

Symmetric-key encryption, is a type of encryption which uses a single key for both encryption and decryption. Some examples of popular symmetric algorithms are twofish, advanced encryption standard (AES) and data encryption standard (DES). Symmetric encryption is commonly used in situations where a large amount of data need to be encrypted quickly and efficiently [9]. Fig. 2.5 shows the basic idea of symmetric key encryption, and the following are the related mathematical property.

$$E_k(m) \longrightarrow c, \tag{2.13}$$

$$D_K(c) \longrightarrow m, \tag{2.14}$$

$$D_k(E_k(m)) = m. \tag{2.15}$$

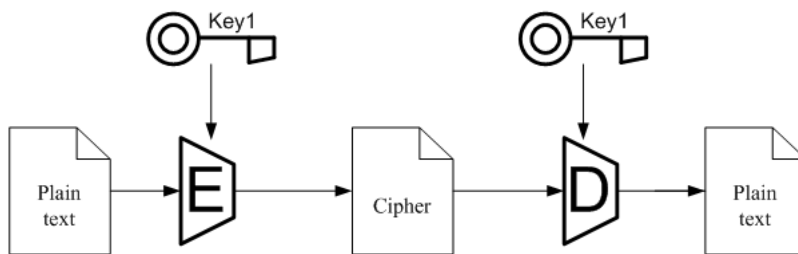


Figure 2.5: Symmetric key encryption.

ES A 1896

2.2.2 Asymmetric Key Encryption

Asymmetric key encryption uses two different keys for encryption and decryption of messages. The privet key is never sent outside the user's device and the public key is distributed using a trusted server. Some common asymmetric key algorithms are Rivest, Shamir and Adleman (RSA) and El-Gamal. The computational complexity of asymmetric encryption is significantly higher than that of symmetric key encryption algorithms. Therefore asymmetric encryption is usually used for applications which require small amount of data to be encrypted, like key exchange and for digital signatures. Fig. 2.6 and the following equations show the basic idea of asymmetric key encryption, where *sk* and *pk* are the secret private-key and the public-key respectively [9].

$$E_{sk}(m) \longrightarrow c, \tag{2.16}$$

$$D_{pk}(c) \longrightarrow m, \tag{2.17}$$

$$D_{pk}(E_{pk}(m)) = m. \tag{2.18}$$

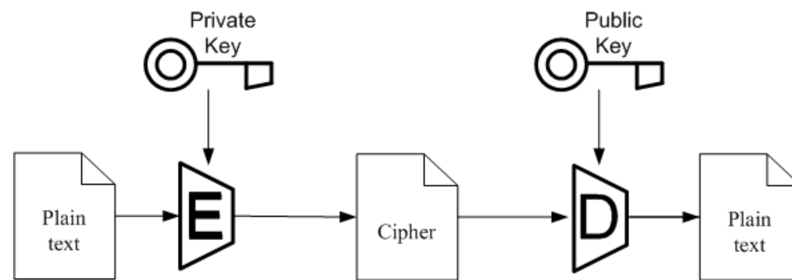


Figure 2.6: Asymmetric key encryption.

ESA

2.2.3 One-Way HASH Functions

A cryptographic hash function take any size block of data and returns a fixed-size bit string. Cyclic redundancy check is one of the most simple hash functions in use.

The ideal cryptographic hash function is expected to posses four main properties:

1) Easy to compute the hash of a given message

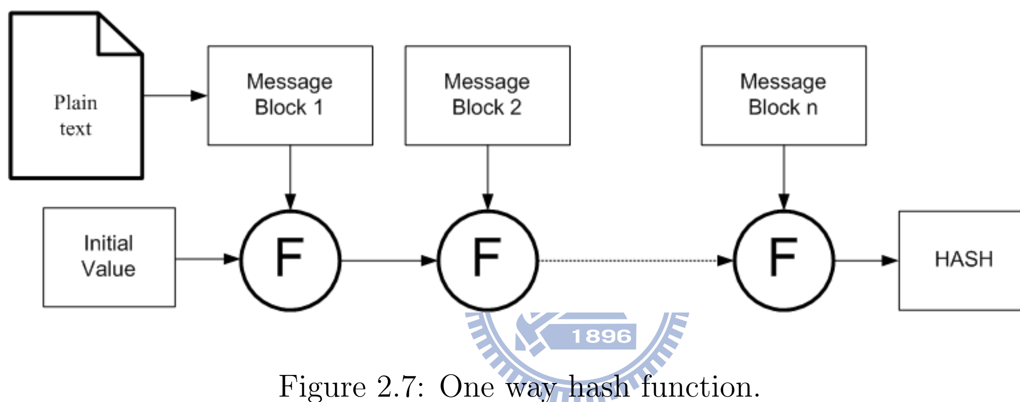
2) Computationally infeasible to find the message related to a given hash value

3) It is infeasible to change any part of a message and generate the same has value as before the changes were done.

4) It is computationally infeasible to find two meaningful messages with the same hash vale.

A graphical representation of the hash function is give in Fig. 2.7 and the mathematical property is given in (2.19). The function F is a HASH function like secure hash algorithm (SHA) [9].

$$F(m) \longrightarrow c. \tag{2.19}$$



1896

Figure 2.7: One way hash function.

2.2.4 Digital Signatures

Digital signature scheme is mainly used for achieving authenticity, integrity and non-repudiation. The two main components in such a scheme are the public, private key pair and a one-way hash function. The user generates a hash over the plain text message and encrypts the result using his private-key, and append this at the end of the plain-text message. The receiver can decrypt the signature, obtain the original hash value and compare it with a newly generated hash value over the plain-text message. A graphical representation of the digital signature is give in Fig. 2.8 and given in (2.20)

and (2.21) is the mathematical representation. The function F is a one-way HASH function like SHA.

$$E_{sk}(F(m)) = Digital\, Signature, \tag{2.20}$$

$$D_{pk}(Digital\, Signature) = \Gamma(m). \tag{2.21}$$

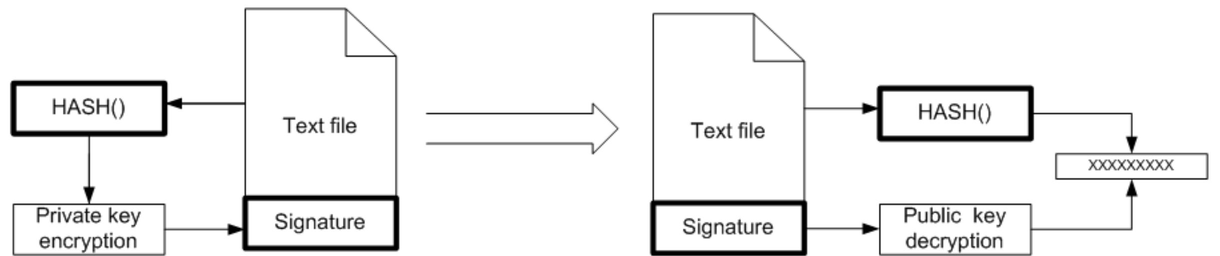


Figure 2.8: Digital signature.

2.2.5 Homomorphic Encryption

An encryption scheme is considered homomorphic if it allows mathematical operations over cipher-text which will result in a meaningful transformation on the plain text message. Using homomorphic encryption, untrusted servers are able to process users' data without decrypting it first. A graphical representation of homomorphic encryption is give in Fig. 2.9.

$$E_k(m_1) = c_1, \tag{2.22}$$

$$E_k(m_2) = c_2, \tag{2.23}$$

$$E_k(m_1 + m_2) = c_1 + c_2, \tag{2.24}$$

where Ψ is used to symbolize mathematical operations in general

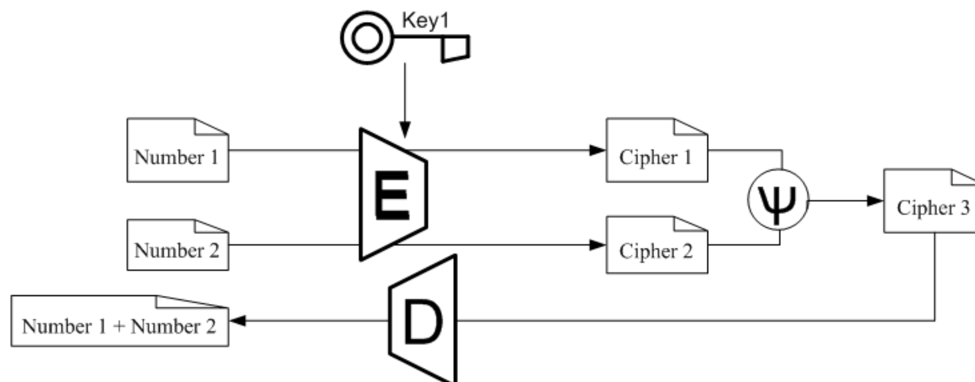


Figure 2.9: Homomorphic encryption.

2.2.6 Comparison of Security Properties of Encryption Schemes

Table 2.7 gives a comparison of different security properties of cryptographic primitives used in secure protocols. The schemes are compared based on the five basic security requirements confidentiality, integrity, authenticity, non-repudiation and homomorphism.

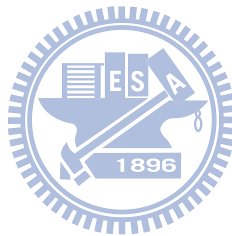
From the table, one can see that it is necessary to combine homomorphic encryption or symmetric encryption with other schemes to achieve fully secure protection. In our cryptographic proposed scheme, we integrate homomorphic encryption and signature to achieve the goal of full security.

Table 2.7: Characteristics of the security properties of cryptographic algorithms.

.	Confidentiality	Integrity	Authenticity	Non-Repudiation	Homomorphism
Symmetric Encryption	**YES**	NO	NO	NO	NO
Asymmetric Encryption	**YES**	NO	**YES**	**YES**	NO
HASH	NO	**YES**	NO	NO	NO
Signatures	NO	**YES**	**YES**	**YES**	NO
Homomorphic Encryption	**YES**	NO	NO	NO	**YES**
Our Goal	**YES**	**YES**	**YES**	**YES**	**YES**

Chapter 3

System Model and Problem Formulation



3.1 System Model

3.1.1 Model

The system model under consideration in this research is location-based community services in cloud computing infrastructures. There are multiple user groups and each group has several members. Each member is given two types of keys, one groups shared key (G-key) and a pair of asymmetric keys. This is illustrated in Fig. 3.1. The G-key is distributed using the asymmetric key pair as illustrated in Fig. 3.2. The server of the location-based community service is not trusted due to the fact that it is in the cloud computing infrastructure.

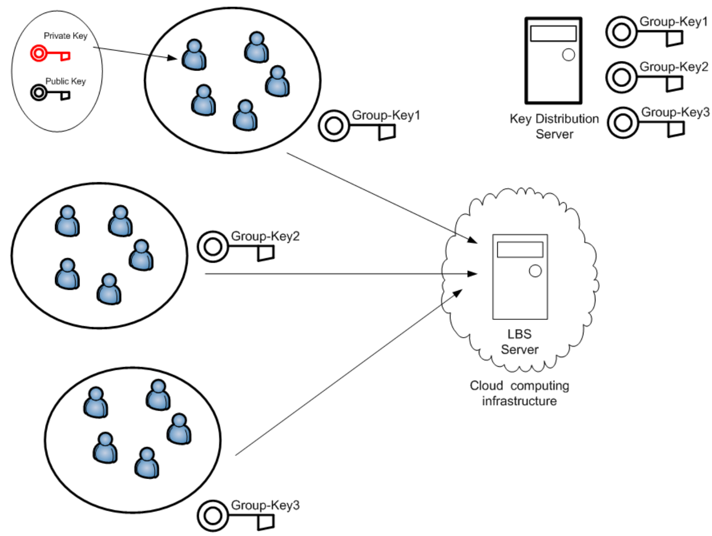


Figure 3.1: Each group is given a shared key in the proposed algorithm.

3.1.2 Assumptions

In this model we have made following assumptions,

- A location-based community service
- The service is hosted in the cloud computing infrastructure
- The cloud computing service provider is not trusted
- There are multiple user groups
- Each group has n number of users
- Users possess two types of keys
  - A shared key for location sharing
  - An asymmetric key pair for Digital signature and key distribution

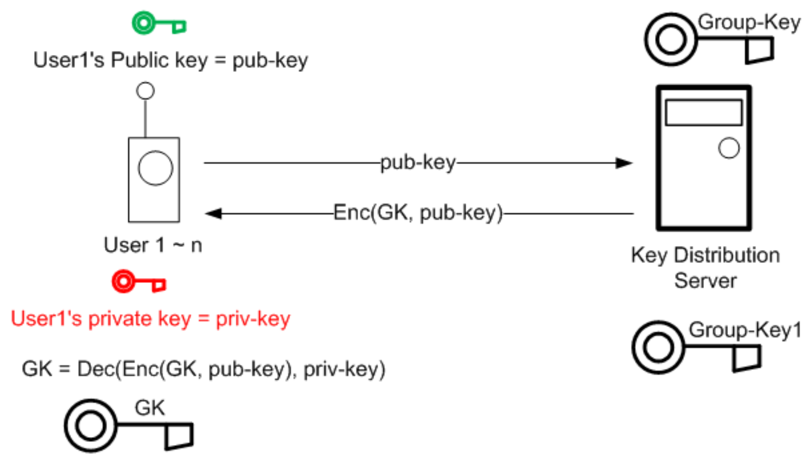


Figure 3.2: Group key distribution.

we have assumed that users in the group need to share their locations with the other members of the group. The location-based community service server is not trusted due to the fact that it is in cloud computing infrastructure. The server is expected to process the location information before being shared with other users in the group.

3.2 Problem Formulation

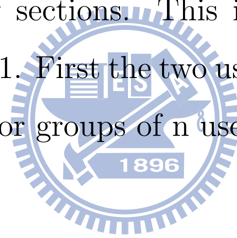
3.2.1 Goal

The goal of our research is to propose a location sharing protocol which can preserve the four basic security properties confidentiality, authenticity, integrity and non-repudiation. The encryption algorithm used in the protocol is required to be homomorphic in order for the server to process location information of the users. A comparison of different cryptographic tools which can be used for achieving these properties are given in Table 2.7. We have

divided these goals in to two tasks. Task-1 which is described in Section 3.2.2 is to achieve confidentiality and homomorphism using the encryption scheme described in [2]. Task-2 which is described in Section 5.1 is to achieve authenticity, integrity and non-repudiation using RSA digital signature algorithm.

3.2.2 Overview of the Proposed XOR Homomorphic Encryption Secure Location Sharing Scheme

An overview of how confidentiality and homomorphism is achieved in the proposed "XOR Homomorphic Encrypted Secure Location Sharing Protocol" is given in the next few sections. This is the task-1 of our two tasks as mentioned in Section 3.2.1. First the two user case of the protocol is discussed and then it is extended for groups of n users.



The Two-user Case

Users encrypt their locations using the symmetric key and send the cipher-text to the server. The server performs addition on the cipher-text and send the result back to both users. Then the users decipher the answer, perform XOR operation with their own locations and recover the location of the other user.

Given below is the protocol written in Alice Bob notation. User A sends her location encrypted with the symmetric key. User B sends his location encrypted with the same symmetric key as well. Then the server calculates XOR (which is adding $A_{Location}$ and $B_{Location}$ together under homomorphic

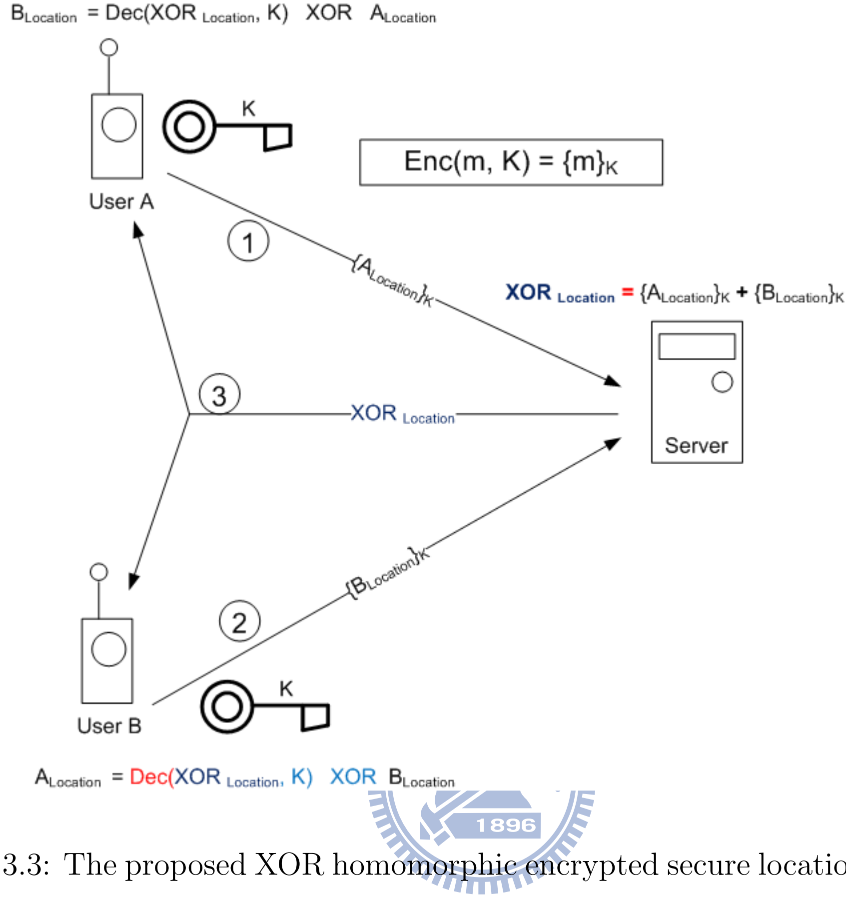


Figure 3.3: The proposed XOR homomorphic encrypted secure location sharing protocol for the two-user case.

encryption scheme) and sends the result to user A and user B. The proposed procedures are illustrated in Fig. 3.3. The XOR Homomorphic Secure Location Sharing protocol is expressed in standard notation in (3.1) and (3.2). First the user A and B encrypt their locations, generate the digital signature, concatenate the two values together and transmit to the server.

$$Message\,1: A \longrightarrow S \quad :\{A_{Location}\}_K, \{H(A, A_{Location})_{KAs}\}, \tag{3.1}$$

$$Message\,2: B \longrightarrow S \quad :\{B_{Location}\}_K, \{H(B, B_{Location})_{KBs}\}, \tag{3.2}$$

Then the server adds the user A and B's encrypted locations together, concatenates the two digital signature with the result and broadcasts to both user as expressed in (3.3).

$$Message\,3 : S \longrightarrow A, B :\{A_{Location}\}_K + \{B_{Location}\}_K,$$
$$\{H(A, A_{Location})_{KAs}\}, \{H(B, B_{Location})_{KBs}\}. \quad (3.3)$$

To generate the digital signature, let user A's identity and user B's identity be values A and B respectively. User A's location is $A_{Location}$ and user B's location is $B_{Location}$. The group shared secret key is K. To generate the digital signature for a given location of a given user, first the identity and the location is concatenated and a HASH value is generated over the result as $H(A, A_{Location})$. The HASH value is then encrypted with users private key as $\{H(A, A_{Location})_{KAs}\}$, which is the digital signature of the user A for the location $A_{Location}$. Digital signature of user B at the location $B_{Location}$ is $\{H(B, B_{Location})_{KBs}\}$. The significance of the HASH function here is that, it acts as an error checking code for the location data. If the location data is corrupted or altered during the transmission, the HASH generated at the receiving end will not match the HASH value generated at the origin.

Fig. 3.4 shows the flowchart of using XOR homomorphic encryption for secure location data exchange.

The n-user Case

Extending the two-user case to n-user case is done by instructing the server to perform XOR between each pair of user locations in the server. The computational complexity of this operation at the server is $O(n^2)$, which

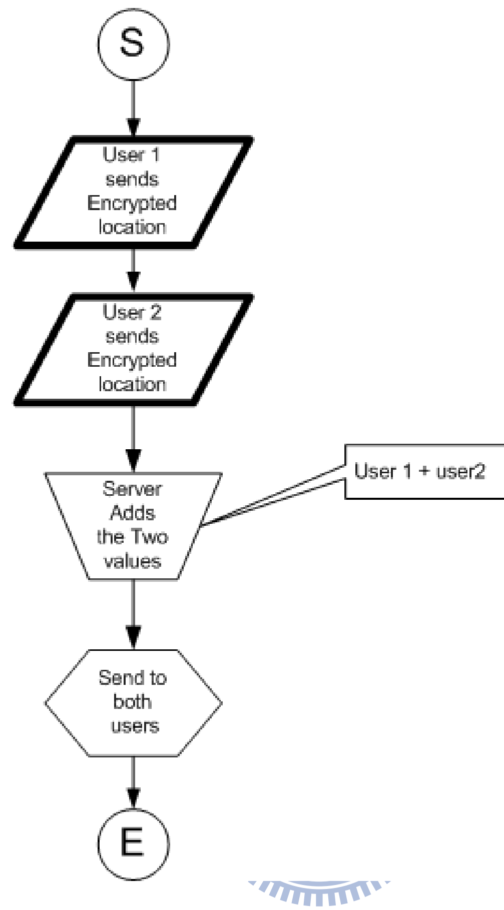


Figure 3.4: Flow of the two-user case.

means that the complexity is polynomial time. Therefore the processing time required for this particular operation is within the capacity of today's computers. A detailed analysis of the system time complexity is given in Section 6.4. Since the service is hosted in the cloud computing infrastructure, increasing the computing power on-demand is a trivial matter. Fig. 3.5 gives the flowchart for n-user case.

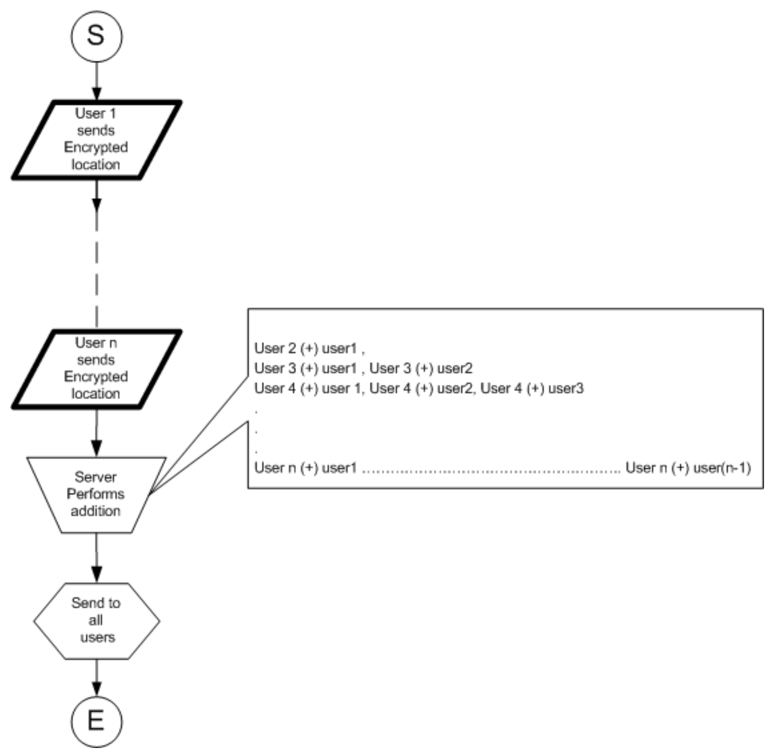
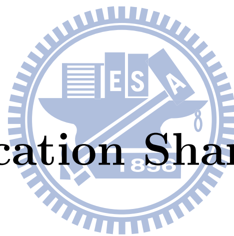


Figure 3.5: Flow chart of using XOR homomorphic encryption for location data exchange in the n-user case.

Chapter 4

XOR Homomorphic Encrypted Location Sharing Scheme



4.1 Traditional Location Sharing Method

Traditional location sharing mechanisms are based on the trusted server model. The service provider's server is trusted, therefore decrypting location information at the server is not considered as a security issue. Servers were usually in private secure information systems facilities. Users will encrypt their locations and send it to the server. The server will decrypt the messages, process it and distributes it to the other users. This process is illustrated in Fig. 4.1.

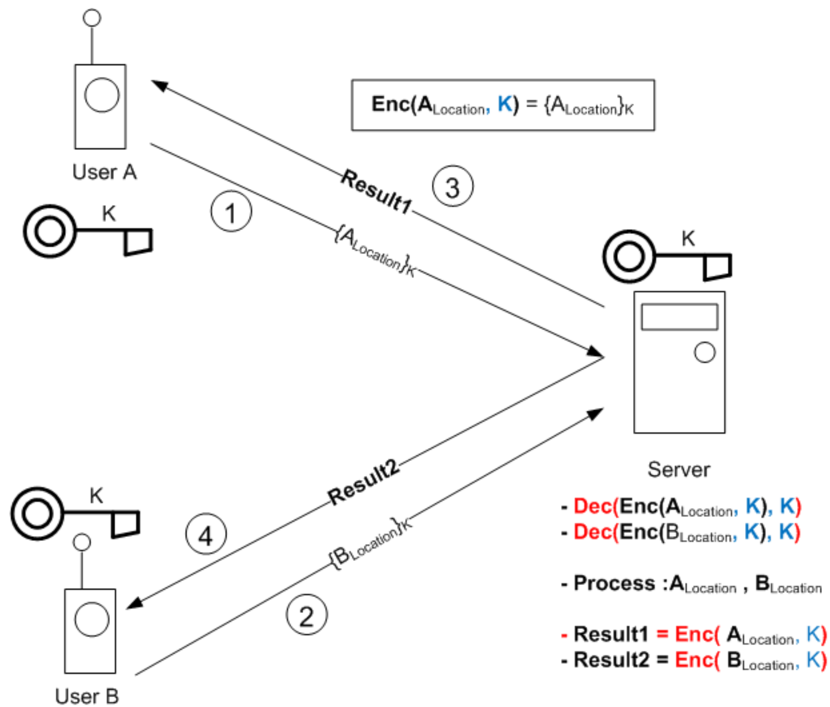


Figure 4.1: Traditional location sharing mechanism.

4.2 Introduction to GDHV's Fully Homomorphic Encryption

The homomorphic symmetric key encryption scheme proposed by Gentry-Dijk-Halevi-Vaikuntanathan [2] is described below. The parameters must be selected according to the restrictions mentioned in (4.3), (4.4) and (4.5).

Shared secret key : odd number "p",

Small "r" , large "q" which are chosen randomly,

Message is a bit, $m \in \{0, 1\}$,

$$\textbf{Encryption function}: c = m + 2r + pq, \tag{4.1}$$

$$\textbf{Decryption function}: m = (c \bmod p) \bmod 2. \tag{4.2}$$

$$p \in [2^{(\eta-1)}, 2^{\eta}), \tag{4.3}$$

$$r \approx 2^{\sqrt{\eta}}, \tag{4.4}$$

$$q \approx 2^{\eta^3}, \tag{4.5}$$

where η is the security parameter,

If $\eta = 64$ **,** $q = 262144$ bits.

1896

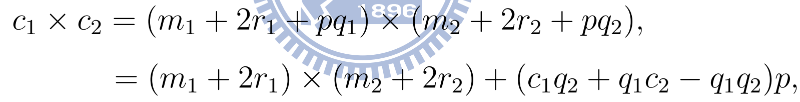
This algorithm is homomorphic given that, the magnitude of the noise generated by random numbers r_1 and r_2 do not exceed the size of the key during calculations. The technique can be seen as mapping the cipher-text between two multiples of "p". Thus when "*cipher* mod p" is performed, the multiples of p gets nullified, and the result contains the message and the noise added in the beginning and during the calculations. Noise can be removed by performing "mod 2" on the result due to the fact that the random number r_x was multiplied by two in the encryption function. Now we explain why this algorithm is additive and multiplicative homomorphic in (4.6) and (4.7).

To begin with, let $c_1 = m_1 + 2r_1 + pq_1$ and $c_2 = m_2 + 2r_2 + pq_2$

1) Additive Homomorphism

$$
\begin{aligned}
& c_1 + c_2 = (m_1 + m_2) + 2(r_1 + r_2) + p(q_1 + q_2), \\
& \text{If } (m_1 + m_2) + 2(r_1 + r_2) << p, \\
& \quad \text{then we have,} \\
& (c_1 + c_2) \bmod p = (m_1 + m_2) + 2(r_1 + r_2), \\
& \quad \text{and,} \\
& ((c_1 + c_2) \bmod p) \bmod 2 = m_1 + m_2.
\end{aligned}
\tag{4.6}
$$

2) Multiplicative Homomorphism

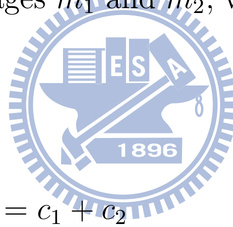


$$
\begin{aligned}
& c_1 \times c_2 = (m_1 + 2r_1 + pq_1) \times (m_2 + 2r_2 + pq_2), \\
& \qquad\quad = (m_1 + 2r_1) \times (m_2 + 2r_2) + (c_1 q_2 + q_1 c_2 - q_1 q_2) p, \\
& \text{If } (m_1 + 2r_1) \times (m_2 + 2r_2) << p, \\
& \quad \text{then we have,} \\
& c_1 \times c_2 \bmod p = (m_1 + 2r_1) \times (m_2 + 2r_2), \\
& \quad \text{and,} \\
& (c_1 \times c_2 \bmod p) \bmod 2 = m_1 \times m_2.
\end{aligned}
\tag{4.7}
$$

4.3 Considerations of Selecting Homomorphic Encryption Algorithm for LBS

4.3.1 XOR Homomorphic Encryption Algorithm

Traditional encryption algorithms define three functions: key generation, encryption and decryption. In homomorphic encryption, there is an additional function called "Evaluate". For instance, the XOR homomorphic operation in the protocol we have proposed is actually addition in the cipher-text resulting in XOR in the plain-text message. Therefore, we define one evaluation function, which is called "XOR homomorphic operation". Specifically, for the group key K shared by messages m_1 and m_2, we define $\Gamma(x, y)$ the XOR homomorphic operator.



$$\begin{aligned}\Gamma(c_1, c_2) &= c_1 + c_2 \\ &= \{m_1\}_k + \{m_2\}_k \\ &= \{m_1 \oplus m_2\}_k,\end{aligned} \tag{4.8}$$

where c_1 and c_2 are the cipher-text corresponding to m_1 and m_2 encrypted by key K.

In the protocol, where the cipher-text c_1 and c_2 are added together at the server, the addition resulting in a XOR operation on the plain-text. As defined above, the plain-text message $m \in \{0, 1\}$, therefore $c_1 + c_2 = m_1 \oplus m_2$

Since $A_{Location}$ and $B_{Location}$ are XORed at the server, the end users must perform another XOR in order to recover the other user's location. For

example, user A should perform the operation in (4.9) to obtain the location of user B from the cipher-text.

$$\text{Message } 3 : S \longrightarrow A : \{A_{Location}\}_K + \{B_{Location}\}_K,$$
$$\text{user A} : \{\{A_{Location}\}_K + \{B_{Location}\}_K\}_{K^{-1}},$$
$$\text{user B's location} = \{\{\{A_{Location}\}_K + \{B_{Location}\}_K\}_{K^{-1}}\} \oplus \{A_{Location}\}. \quad (4.9)$$

There are several XOR homomorphic encryptions schemes. In general, most additive homomorphic encryption algorithms can be used for obtaining the XOR homomorphism by bitwise encryption and applying $(\bmod 2)$ at the decryption. The GoldwasserMicali crypto system [17] is a public key scheme which has XOR homomorphic property. The security of this scheme is based on the quadratic residuosity problem and it is considered to be the first provably secure probabilistic public-key encryption scheme [9]. As other bitwise encryption schemes, this also suffer from the cipher-text expansion. The XOR homomorphism of the algorithm is mathematically expressed in (4.10),

where x is a quadratic residue mod N and $x = y^2 \bmod N$ for some y. The encryption function: $c_i = y^2 x^{m_i} \bmod N$. Let c_1 and c_2 be the corresponding cipher text of the plain text messages m_1 and m_2 where $m \in \{0, 1\}$,

$$Enc(m_1 \oplus m_2) = c_1 c_2 \bmod N. \quad (4.10)$$

The proposed protocol requires a symmetric key XOR homomorphic algorithm. Almost all of the homomorphic encryption algorithms are public key systems. Therefore we stick to the scheme described in [2], which is a

Table 4.1: Desired characteristics of the secure community LBS encryption algorithm.

Required Property	**Reason**
Additive homomorphism	XOR operation
Symmetric key encryption	Shared group key
Minimal computation complexity	Resource challenged mobile clients
Minimal cipher text growth	Bandwidth efficiency

symmetric key scheme with additive and multiplicative homomorphic properties.

4.3.2 Other Consideration in Selecting the Homomorphic Encryption Algorithm

In Table 2.7 we have compared security properties of the proposed protocol with the properties of existing individual cryptographic tools. As mentioned, our goal is to propose a protocol which is capable of providing all five primary security requirements: confidentiality, integrity, authenticity, non-repudiation and homomorphism.

When selecting the homomorphic encryption algorithm we take the four criteria as described in Table 4.1 as decision factors.

Additive Homomorphism

There are several additively homomorphic encryptions algorithms including Elliptic curve elgamal encryption scheme [18], Paillier cryptosystem [19], Okamoto-Uchiyama encryption [20], Ragevs homomrphic scheme [21] and Gentry-Dijk-Halevi-Vaikuntanathan scheme based on Ragevs [2]. The following homomorphic encryption algorithms require asymmetric key, such as Elliptic curve elgamal [18], Paillier cryptosystem [19] and Okamoto-Uchiyama [20].

Majority of the existing homomorphic algorithms are partially homomorphic. For example RSA is multiplicatively homomorphic and Okamoto-Uchiyama encryption is additively homomorphic. For the proposed scheme we only require additive homomorphism, but considering future expansions of server-side processing we preferred choosing an algorithm which is both additively and multiplicatively homomorphic.

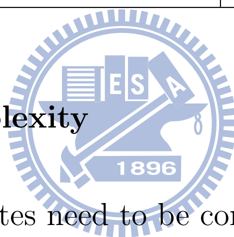
Therefore we have picked Gentry-Dijk-Halevi-Vaikuntanathan [2] scheme based on Ragevs, which is multiplicative and additive homomorphic and symmetric.

Symmetric vs Asymmetric Key Encryption

In our protocol, we require the user groups to use a group key called G-Key to encrypt their locations. This must be a symmetric key encryption due to the fact that, the server needs to perform the addition between different users' locations

Table 4.2: Computational complexity of asymmetric vs symmetric encryption.

Encryption Scheme	**Mathematical Operation**	**Complexity**
Public Key	Modular exponentiation by repeated multiplication and reduction, n-bit exponent	public key operations - $O(n^2)$, private key operations - $O(n^3)$. where n is the number of bits in the modulus.
Symmetric Key	Addition, Multiplication, Modulo	$O(n), O(n^2), O(n^2)$

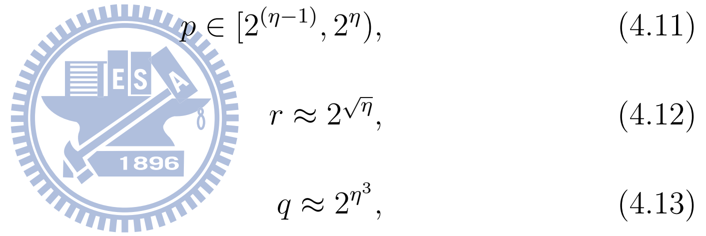


Minimal Computation Complexity

In our application, location updates need to be communicated as frequently as within the scale of a few seconds. This requires considerable amount of computation. The cryptographic algorithm adds extra complexity in to the process. Modulo exponentiation in public key algorithms has a high computational complexity compared to simple addition, multiplication and modulo based symmetric key schemes. Therefore we have selected a symmetric key algorithm for location encryption. Table 4.2 shows the complexity comparison of public key *vs* symmetric key encryption.

Minimal Cipher Text Growth

This property is required for efficient bandwidth utilization. Cipher-text growth is one of the main concerns of existing homomorphic encryption schemes. If the Gentry-Dijk-Halevi-Vaikuntanathan [2] scheme is used with parameter selection for best possible security, the size of cipher-text can grow n^3 times where n is the number of bits in the encryption key. That is, it is implied that one bit in the plain text will become n^3 bits after encryption. The choice of p, q and r in the encryption algorithm of Gentry-Dijk-Halevi-Vaikuntanathan [2] are suggested to be,



$$p \in [2^{(\eta-1)}, 2^{\eta}), \tag{4.11}$$

$$r \approx 2^{\sqrt{\eta}}, \tag{4.12}$$

$$q \approx 2^{\eta^3}, \tag{4.13}$$

where η is the security parameter,

If $\eta = 64$ **,** $q = 262144 bits.$

Security of the algorithm is based on "Approximate-GCD problem" and "Sparse subset problem". For these two problems to remain computationally infeasible, q and r should be selected accordingly based on (4.12)(4.13). To prevent a brute force attack it is safe to select a key which is longer than 64 bits.

4.4 Proposed XOR Homomorphic Encrypted Secure Location Sharing Protocol

4.4.1 Detailed Protocol Steps

The proposed protocol is expressed by a twelve step process in this section. There are three parties involved in the process of location sharing. User 1, user 2 and the server. Steps $1 \sim 3, 9, 10$ are performed by user 1. Steps $4 \sim 6, 11, 12$ are performed by user 2. The steps 7 and 8 are performed by the server.

Step 1 : User-1 converts the location to binary

Step 2 : User-1 encrypts the location using homomorphic encryption

Step 3 : User-1 sends the Encrypted location to the server

Step 4 : User-2 converts the location to binary

Step 5 : User-2 encrypts the location using homomorphic encryption

Step 6 : User-2 sends the Encrypted location to the server

Step 7 : Server adds user 1's encrypted location and user 2's encrypted location

Step 8 : Server broadcasts the result to the users

Step 9 : User-1 decrypts the received result

Step 10 : User-1 XOR the decrypted result with his own location to obtain user 2's location

Step 11 : User-2 decrypts the received result

Step 12 : User-2 XOR the decrypted result with his own location to obtain user 1's location

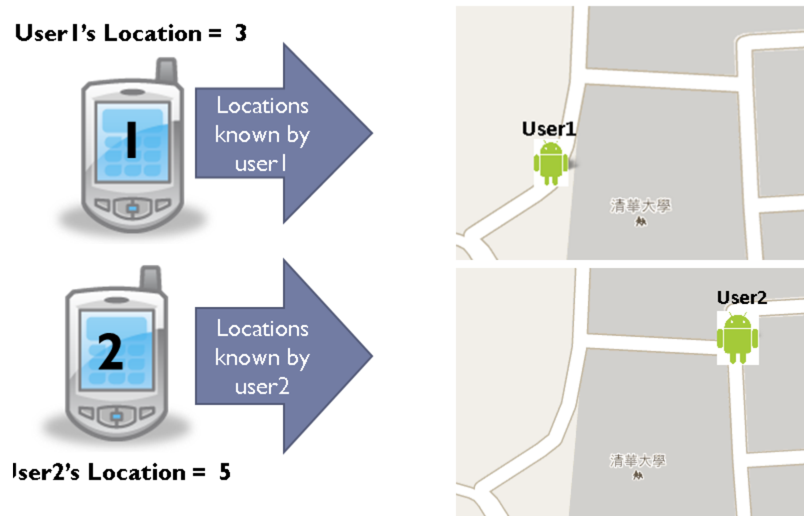
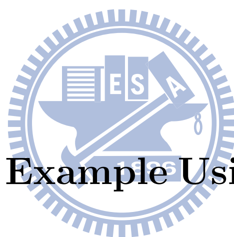


Figure 4.2: Locations known by each user before location sharing .



4.4.2 Illustrative Example Using Pseudo Location Data

In this example we assume that the GPS coordinates are single digit integers, and exchange these two value between the two users. In the next section, sharing real GPS coordinates between users is outlined. Fig. 4.2 gives a snapshot of the locations known by each user before location sharing, and Fig. 4.3 shows the locations known by each user after the location sharing.

Step 1 : Let user 1's location be "3". When it is converted to binary it is 011_b.

Step 2 : User 1 encrypts his location bit by bit as follows. Let the shared key be 203. Parameters q and r are random values 5,6,4 and 9,10,13 consecutively. Ones the the bits are encrypted using the equation $c = m +$

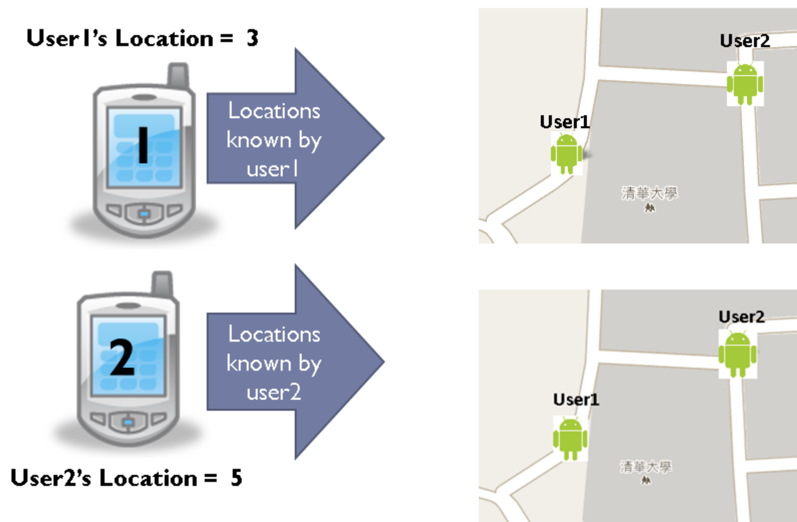


Figure 4.3: Locations known by each user after location sharing .

$2r + pq$, the result is 1837, 2043, 2648.

Step 3 : User 1 sends the location to the sever

Step 4 : Let user 2's location be "5". When it is converted to binary it is 101_b.

Step 5 : User 2 encrypts his location bit by bit as follows. Let the shared key be 203. Parameters q and r are random values 7,8,9 and 7,15,18 consecutively. Ones the the bits are encrypted using the equation $c = m + 2r + pq$, the result is 1436, 3061, 3673.

Step 6 : User 2 sends the location to the sever

Step 7 : Server adds the two locations together integer by integer. $1837, 2043, 2648$ added to $436, 3061, 3673$ is $3273, 5307, 6321$.

Step 8 : Server broadcasts the result to both users.

Step 9 : User 1 receives 3675 , 4491 , 5296. He decrypts the message

using the shared key 203 and the decryption formula $m = (c \bmod 203) \bmod 2$ and obtains the binary bits 110_b.

Step 10 : User 1 XOR the result 110_b with his own location 011_b and obtains the user 2's location, which is $101_b = 5$.

Step 11 : User 2 receives 3675 , 4491 , 5296. He decrypts the message using the shared key 203 and the decryption formula $m = (c \bmod 203) \bmod 2$ and obtains the binary bits 110_b.

Step 12 : User 2 XOR the result 110_b with his own location 101_b and obtains the user 1's location, which is $011_b = 3$.

4.4.3 Illustrative Example Using GPS Location Data

GPS locations are expressed in degrees of longitude and latitude. These values need to be converted in to 32 bit two's complement binary before being encrypted. Therefore, when dealing with real GPS locations, the encryptions, decryption and XOR operation are performed as follows. let the user's location be latitude 24.789657 and longitude 120.999749. Then these values are multiplied by 10^6 to convert in to micro degrees. When the resulting values are converted to binary two's complement, it becomes,

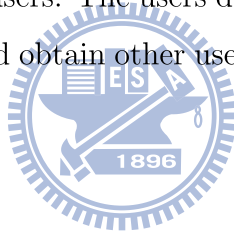
$$\textbf{Latitude} : 000000010111110100100001010011001_b$$

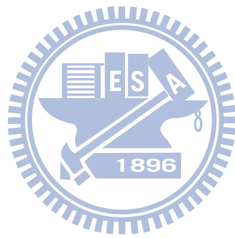
$$\textbf{Longitude} : 000000000000000000000000001111000_b$$

Once the longitude and latitude are encrypted bitwise using the shared key 203 and randomly generated q and r, the result is,

$$Encypted\,latitude = \begin{bmatrix} 999754 & 1109723 & 459914 & 339948 \\ 779832 & 339956 & 819820 & 709844 \\ 279972 & 319957 & 439921 & 249982 \\ 1139725 & 689851 & 1099725 & 309963 \\ 589887 & 599873 & 329951 & 1159718 \\ 259982 & 549887 & 219999 & 369937 \\ 929782 & 649865 & 499908 & 389946 \\ 379949 & 829807 & 279980 & 979773 \end{bmatrix}$$

Then the users send these encrypted information to the server. The server adds user 1's and user 2's locations together as described in step-7 and broadcasts the result to the users. The users decrypt the result, perform XOR with their own locations and obtain other users' locations following the steps described in Section 4.4.1.





Chapter 5

Security Enhancements for XOR Homomorphic Encrypted Location Sharing Scheme

5.1 RSA Digital Signature Scheme

The RSA digital signature scheme is used in achieving authenticity, Integrity and non-repudiation. This was described as the task-2 in Section 3.2.1. Users generate a digital signature based on their current location and transmits it with the location sharing message as described in (5.1). A is user 1's identity, KA_s is user 1's private key, KA_p is user 1's public key and $H()$ is a secure hash function.

$$\{H(A, A_{Location})_{KA_s}\} \tag{5.1}$$

At the receiving end, user 2 verifies the signature by generating a hash value on decrypted user 1's location and comparing it with the hash value in the

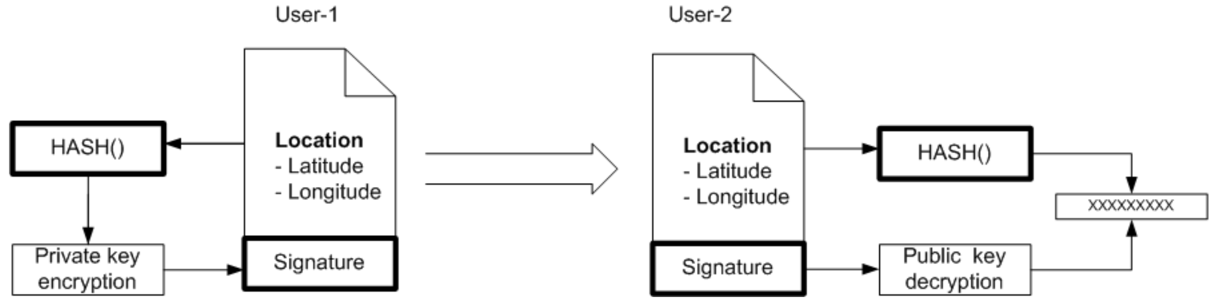


Figure 5.1: Client digital signature generation and verification.

digital signature sent by user 1, which is described in (5.3). A graphical representation of this process is given in Fig. 5.1

$$H(A, A_{Location}) = Dec(\{H(A, A_{Location})_{KAs}\}, KA_p), \tag{5.2}$$

$$H(A, A_{Location}) = H(A, Decrypted\, A_{Location}). \tag{5.3}$$

This verification process guarantees three properties authenticity, integrity and non-repudiation. Once the signature is verified, user 2 at the receiving end can be sure that the message was sent by user 1, because the digital signature was signed by user 1's private key. Only user 1 posses this particular private key. The integrity of the location update is confirmed by the SHA-1 hash. It is similar to a cyclic redundancy check (CRC), which will confirm whether the transmitted message and the received messages are the same. If an adversary had modified the massage during transmission, the hash generated at the receiving end and the hash generated at the origin will not match. If the hash check fails at the receiving end, the location update need to be discarded as it has been altered during the transmission. The third property is non-repudiation. This property is a by product of authentication. If user 1

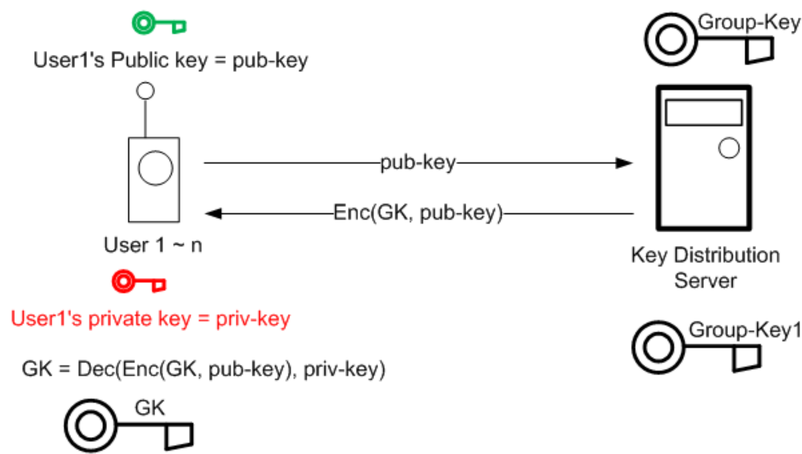


Figure 5.2: Group key distribution.

sends a location update signed using his private key, he cannot deny sending that update at a later time, because only user 1 posses that particular private key. Implementation details of the RSA digital signature scheme is given in Section 6.2.1.

5.2 Group Key Distribution

The group shared key ($G - Key$) is distributed by a trusted key distribution server. Clients obtain the group key through the trust relationship between the key distribution server and them. User's public-key private-key pair is used in this operation according to the Fig. 5.2.

5.3 Protocol Security Analysis

5.3.1 Cryptography *vs* Security Protocols

One interesting analogy to think about encryptions algorithms and security protocols is to equate the encryption algorithm to a bicycle lock [22]. You can have the best possible lock, however if you tie your bicycle front wheel to an iron pole, someone can unscrew the front wheel out of the bicycle frame and steal rest of the bicycle. This is like using the perfect encryption algorithm in an insecure protocol. An adversary can get access to secret information despite the strength of underlying encryption [22].

Security protocols are the key to secure communication. Security protocols can appear extremely simple, but the security properties that they have to preserve are extremely delicate. Therefore, it is difficult to define protocols correctly by informal reasoning. There have been many incidences where security flows in popular security protocols has been fund years later [23, 24]. Table 5.1 provides a comparison between the two approaches of secure protocol analysis.

In this chapter we will analyse the proposed security protocol using two industry standard secure protocol analysis tools.

5.3.2 Dolev-Yao Model

Dolev-Yao model [23, 25], which is also called the formal model is shown in Fig. 5.3. It is used for modelling the network in secure protocol analysis. This model defines an omnipotent intruder who has full control over the network.

Table 5.1: Formal approach vs cryptographic approach of protocol verification.

.	**Formal approach**	**Cryptographic approach**
Messages	Terms	Binary-strings
Encryption	Idealized	Algorithm
Adversary	Idealized	Algorithm
Proof	Automated	Cumbersome and difficult

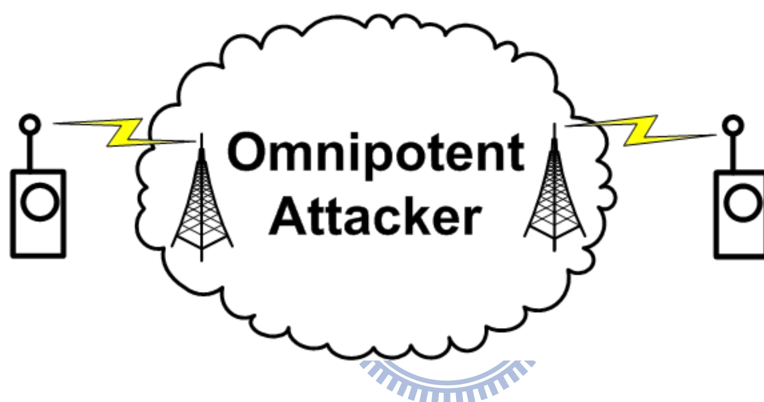


Figure 5.3: Dolev-Yao model.

The only thing stopping the intruder from obtaining secret information is the strength of encryption, which is idealised in this model.

5.3.3 Security Verification of the Proposed Protocol using AVISPA Tool

AVISPA [26] stands for automated validation of Internet security protocols and applications. It is a project for developing an automated technology for analysing secure transaction protocols. AVISPA is the first tool that intro-

duced algebraic operations like XOR on the cipher-text. Over 80 protocols are analysed by using this tool for IETF and Siemens [27].

To verify a security protocol using AVISPA tool, the examined protocols need to be expressed in High Level Protocol Specification Language (HLPSL) [28]. The first step is to write down the protocol in Alice and Bob notation. Specifically, the following four messages will be sent between server S and users A and B, which are

$$Message\,1: A \longrightarrow S: \{A_{Location}\}_K, \{H(A, A_{Location})_{KAs}\},$$

$$Message\,2: B \longrightarrow S: \{B_{Location}\}_K, \{H(B, B_{Location})_{KBs}\},$$

$$Message\,3: S \longrightarrow A: \{A_{Location}\}_K + \{B_{Location}\}_K,$$
$$\{H(A, A_{Location})_{KAs}\}, \{H(B, B_{Location})_{KBs}\},$$

$$Message\,4: S \longrightarrow B: \{A_{Location}\}_K + \{B_{Location}\}_K,$$
$$\{H(A, A_{Location})_{KAs}\}, \{H(B, B_{Location})_{KBs}\}.$$

The next step is to express the protocol in HLPSL. It is designed to be easy to read and write HLPSL specifications. HLPSL provides a high level of abstraction and has many features that are found in most protocol specifications languages [29]. The full protocol specification in HLPSL is given in appendix A.

Fig. 5.4 is a graphical representation of the protocol flow. This was generated using the SPAN tool [30], which is a software that takes HLPSL as input and produces protocol animations.

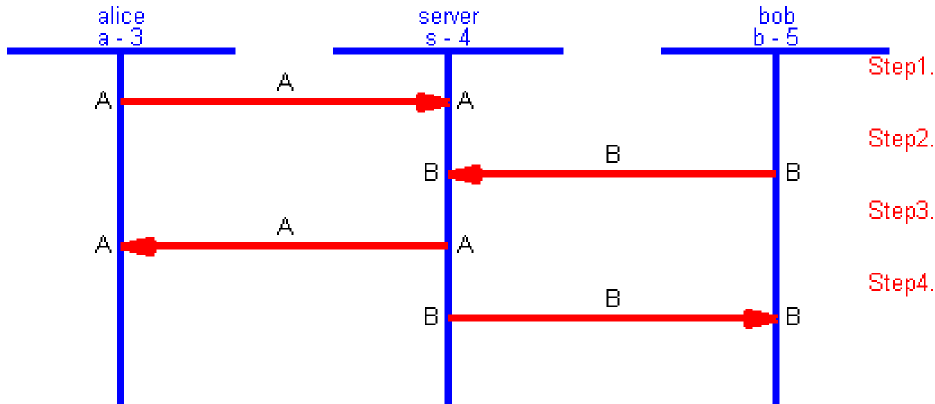


Figure 5.4: Protocol written in HLPSL animated using the SPAN tool.

Defining homomorphic encryption in protocol verification tools is challenging because a formal definition for homomorphic encryption is not available in HLPSL. However there are definitions for some mathematical operations such as XOR and exponentiation. Therefore we have used the predefined XOR algebraic operation in expressing our protocol in HLPSL.

$$\textbf{XOR}\ (A_{Location}, B_{Location}). \tag{5.4}$$

AVISPA tool first converts the protocol defined in HLPSL into an intermediate format, and analyses using four protocol analysers:

1) OFMC (The On-the-Fly Model-Checker),
2) CL (Constraint-Logic-based model-checker),
3) SATMC (SAT-based Model-Checker),
4) TA4SP (Tree Automata based Analysis of Security Protocols).

In the following, we introduce the functions of the above four tools.

- **OFMC On-the-Fly Model-Checker** - Is based on automata-based

linear temporal logic(LTL) [31] model checking [32].

- **Constraint-Logic-based Model-Checker** - Is an OCaml-based deduction rules implementation developed in the AVISPA project [26].
- **SAT-based Model Checker** - SATMC (SAT-based Model Checker) [33] is a bounded model checker for security protocols. [34]
- **Tree Automata based Automatic Analysis of Security Protocols** - This tool analyses a given protocol by rewriting it in tree languages for unbounded number of sessions [26, 35].

The XOR homomorphic secure location sharing protocol was analysed using the AVISPA web tool and the result is given below. The first responses confirm that the protocol is secure and the last tool does not get started due to the nature of the protocol. A complete log of the result is given in appendix B.

AVISPA Tool Summary (Fig. 5.5)

OFMC : SAFE
CL-AtSe : SAFE
SATMC : SAFE
TA4SP : N/A

Figure 5.5: Result of verifying the proposed protocol using AVISPA tool.

5.3.4 Security Verification of the Proposed Protocol using ProVerif Tool

ProVerif [36] is an automatic cryptographic protocol verifier in Dolev-Yao model (formal model). ProVerif tool converts the protocol written in spi-calculus to horn clauses and then analyse for unbounded number of sessions [37]. It can handle many cryptographic operations, including symmetric and asymmetric key encryption and key exchange protocols. It can handle an unbounded number of sessions of the protocol efficiently using a small amount of memory. This analyser was picked to confirm the verification by the AVISPA tool. ProVerif provides methods to express homomorphic encryption as a custom function.

To verify a security protocol using ProVerif, the protocol needs to be written in spi-calculus, which is an extension of π-calculus to express cryptographic protocols.

There is no syntax to express homomorphic operations in spi-calculus. A work around to express security protocols using homomorphic encryption is, to define a custom function to specify the required homomorphic properties. The protocol expressed using spi-calculus is given in Appendix C.

Required Properties of Addition

$$A + B = N,$$

$$N - A = B,$$

$$N - B = A. \tag{5.5}$$

ProVerif function which represent this property

(* Homomorphic addition function *)

fun addH/2,

reduc deduct$(addH(p, q), q) = p$,

reduc deduct$(addH(p, q), p) = q$.

We attempted verifying the strong secrecy of Alice's and Bob's locations ($A_{Location}$ and $B_{Location}$). This requirement is expressed in spi-calculus as below.

(* noninterf = Prove strong secrecy *)

noninterf secretA,secretB.

noninterf secretA.

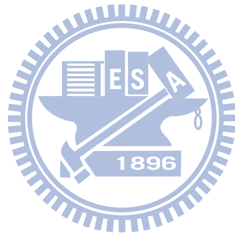
noninterf secretB.

(* Test whether "secretA" and "secretB" are secret *)

query attacker:secretA.

query attacker:secretB.

According to the analysis by ProVerif tool, the protocol is secure against strong secrecy of secretA ($A_{Location}$) and secretB ($B_{Location}$). The result is given in Appendix D.



Chapter 6

Performance Issues of XOR Homomorphic Encrypted Secure Location Sharing Protocol

6.1 Concept Verification of XOR Homomorphic Encryption Algorithm by MATLAB

MATLAB implementation issues of the process will be discussed in this chapter. The encryption and decryption functions are implemented in MATLAB in order to verify the XOR homomorphic property of the encryption algorithm in [2].

Fig. 6.1 shows the flow chart of implementing XOR homomorphic en-

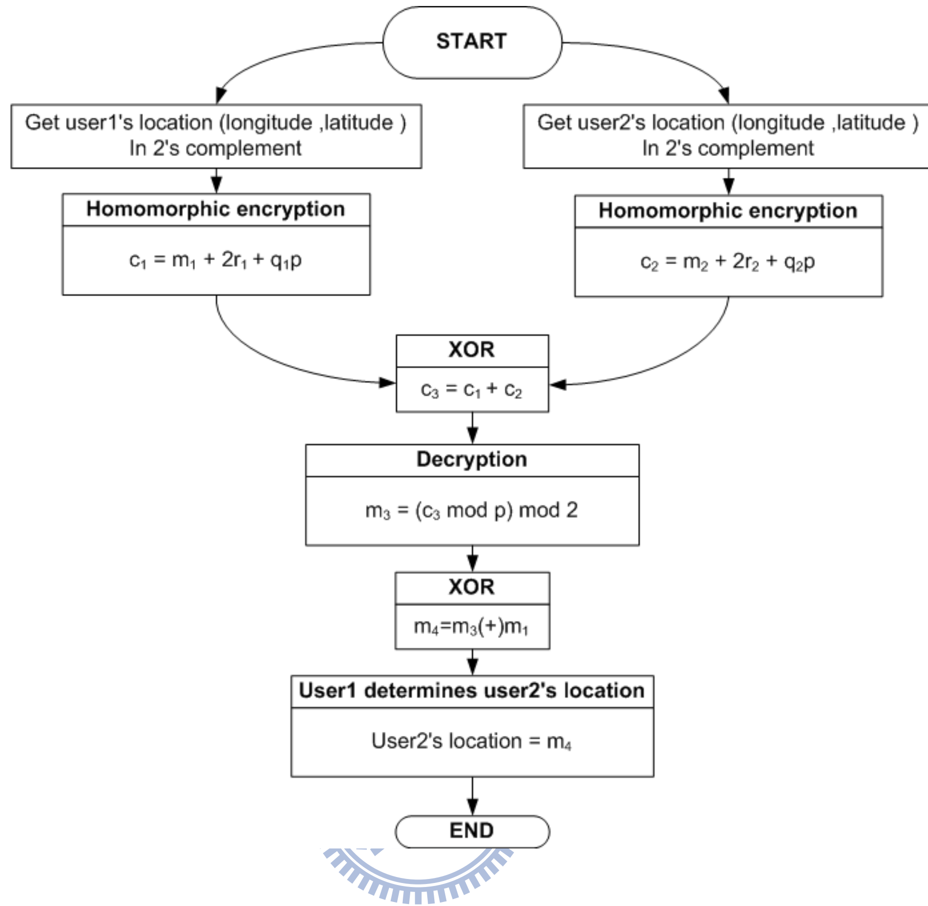


Figure 6.1: MATLAB simulation flow chart for the XOR homomorphic encryption in community LBS.

cryption in community LBS. Our goal is that user 1 can determine user 2's location based on the response from the server. The server adds user 1's location and user 2's location together. In this scheme addition in the cipher-text means bitwise XOR in the plain text domain. Therefore, when user 1 receives $\{$user 1's location $\oplus$ user 2's location$\}_K$, the user 1 can XOR the received value with his own location to obtain user 2's location.

First, we represent the location data in the form of two's complement as follows.

Calculating Two's Compliment

```
twosX = dec2bin(mod((x),2^16),16);  % 16bit twos compliment
twosY = dec2bin(mod((y),2^16),16);  % 16bit twos compliment
```

Then, the data bits of location are encrypted based on the equation (4.1).

Bitwise Encryption of Locations in Two's Compliment

```
for lx = 1:16
     r = round(rand(1)*100);
     lengthXenc(lx) = twosX(lx) + 2*r + oddkey*q;
     % encryption  c = m + 2r +qp
end

for ly = 1:16
   r = round(rand(1)*200);
   lengthYenc(ly) = twosY(ly) + 2*r + oddkey*q;
   % encryption c = m + 2r +qp
end
```

Then, the server adds the encrypted location data bits as follows.

Server Performs XOR by Adding the Cipher Text Together

```
% server does the XOR
x3 = x1 + x2;
y3 = y1 + y2;
```

Next, the server sends the sum of the encrypted data of the two users' location to both users. Each user will decrypt the data according to (4.2). last, each user performs XOR operation with his own location to get the other user's location.

User_1 Receives the Result from the Server and Decrypts

```
% key p = 99997
% decryption equation m = (c mod p)mod 2
x4 = mod(mod(x3,99997),2);
y4 = mod(mod(y3,99997),2);
```

User_1 Performs XOR on the Decrypted Value

```
% User1 does the XOR in order to recover user2's location
user2TwosX = xor(x4,user1X1);
user2TwosY = xor(y4,user1Y1);
```

The complete MATLAB code for this algorithms are given in Appendix E.

6.2 Implementation of the Proposed Protocol in Android Platform

This section describes the implementation issues of the proposed protocol in an Android application, using JAVA programming language [38]. The mobile client is targeted to be run in an Android mobile device, and the server is hosted by the cloud computing infrastructure. In the application development, Google application programming interface(API) level 7, Android software development kit(SDK) [39], Bouncy castle crypto APIs [40] and Eclipse integrated development environment(IDE) [41] were used [42] to implement our algorithm. The application was tested in a HTC hero mobile handset [43] operating on Android version 2.1 [44].

6.2.1 Implementation at the Client Side

The client codes mainly consists of the following components. The two's complement arithmetic class, encryption and decryption classes, JAVA TCP socket, Google maps API, Bouncy castle crypto APIs RSA implementation and the Graphical user interface(GUI). The program flowchart at the client side is shown in Fig. 6.2 [45].

Graphical User Interface

The GUI is XML based and follows Android relative layout, which embeds a Google mapview [46] as shown in Fig. 6.3

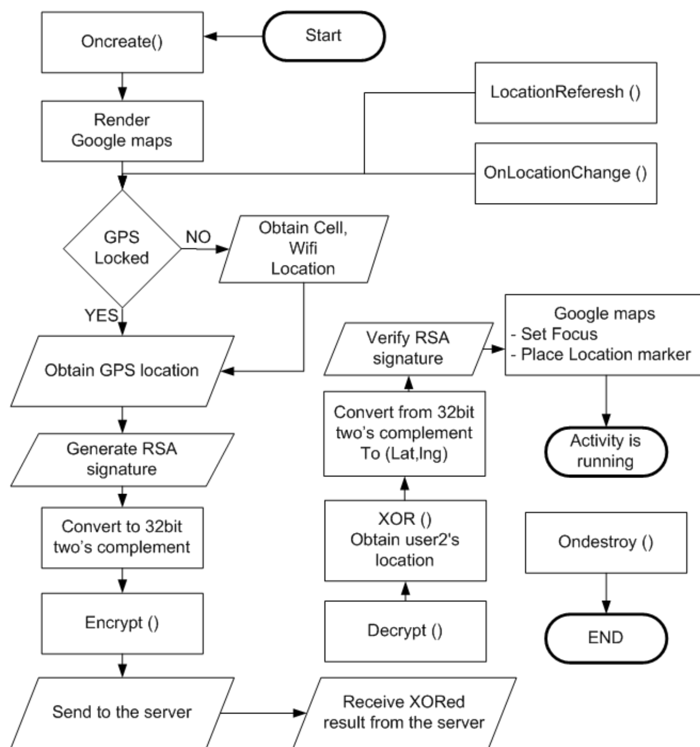


Figure 6.2: The Android client programme flow.

```
<?xml version="1.0" encoding="utf-8"?>
<RelativeLayout xmlns:android=
"http://schemas.android.com/apk/res/android"
    android:id="@+id/mainlayout"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:text="@string/hello">
    <com.google.android.maps.MapView
    android:id="@+id/mapview"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent"
    android:clickable="true"
    android:apiKey="0hF1EsCi1r0WSg7ZcE_Y19d4XaBImlQKEkKTPng"/>
</RelativeLayout>
```

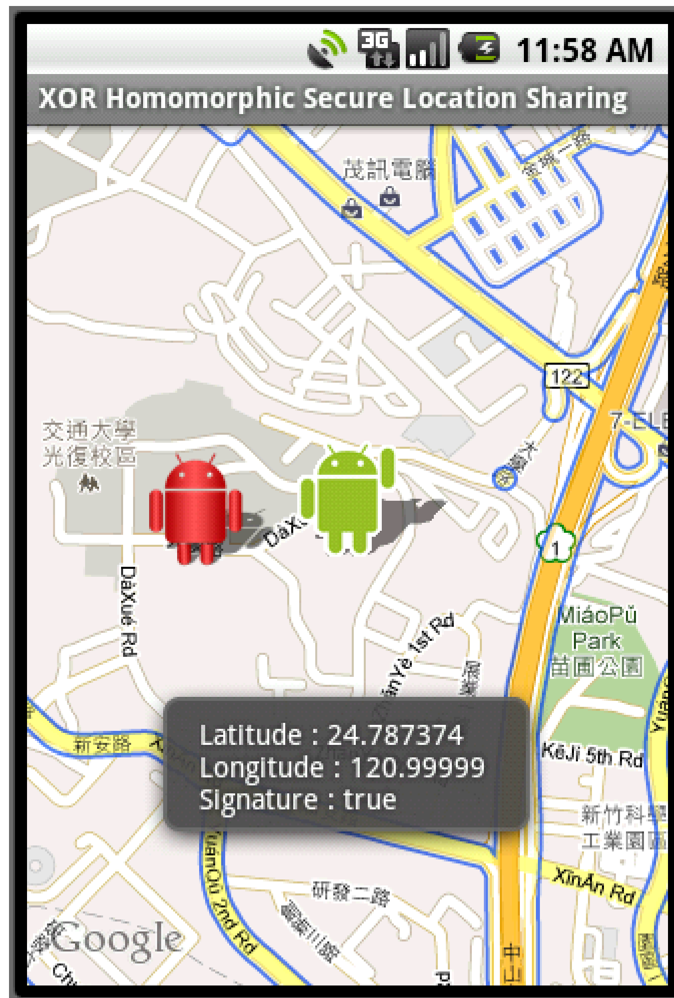


Figure 6.3: Android application GUI

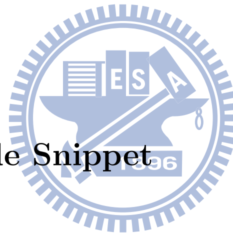
Two's Complement Arithmetic Class

The underlying mechanism in JAVA handles binary numbers in two's complement arithmetic, but there are not many classes available for accessing these native operations. Therefore, we had to defined our own two's complement arithmetic class. Given below is the algorithm behind the operation. The complete code is given in Appendix F [47].

```
if (the value is <  0){
        bitwise invert;
        add 1;
}else{return the original value;}
```

Encryption and Decryption Classes

The code segment below illustrates how the homomorphic encryption and decryption are performed. The complete JAVA class is given in the appendix F. The random numbers "r" and "q" are generated using the "java.util.Random" class. The biginteger data type was used for handling large cryptographic parameters.



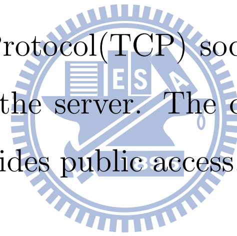
Encryption Class Code Snippet

```
qLarge = new BigInteger(4096,rand);
rLarge = new BigInteger(8,rand);
BigInteger c1 = BigInteger.valueOf(encDigitInt[k]);
BigInteger c2 = pLarge.multiply(qLarge);
BigInteger c3 = rLarge.multiply(BigInteger.valueOf(2));
BigInteger c4 = c1.add(c2).add(c3);
outputStringBig[k] = c4.toString();
```

Decryption Class Code Snippet

```
for (int i = 0;i<32;i++)
{
locationLarge = new BigInteger(location[i]);
BigInteger mm = locationLarge.mod(keyLarge).mod(BigInteger.valueOf(2));
locationDec[i] = mm.intValue();
}
```

Client TCP Socket



A JAVA Transmission Control Protocol(TCP) socket is utilised in communication between the client and the server. The clients initiates TCP connections to the server which provides public access. A code snippet from the client TCP socket is given here.

```
try {
Socket clientSocket = new Socket("yourdoamin.com", 2004);
ObjectOutputStream outToServer = new
ObjectOutputStream(clientSocket.getOutputStream());
ObjectInputStream inFromServer = new
ObjectInputStream(clientSocket.getInputStream());
outToServer.writeObject(msgToServer[k]);'
// close the socket after transmission
try{
clientSocket.close();
   }
   catch(IOException ioException){
    ioException.printStackTrace();
}
}catch (Exception e) {
printScr("TCP Error: " + e.toString());
}
```

Google Maps Application Programming Interface

Google maps API provides the *com.google.android.maps* class with functionalities for map-rendering, caching map tiles and many other map manipulation methods. This package is used in rendering the maps in this application. The code snippet of calling for Google API [46] is given next.

```
mapView = (MapView) findViewById(R.id.mapview);
mapView.setBuiltInZoomControls(true);
mapView.setStreetView(true);
mapController = mapView.getController();
mapController.setZoom(16);
locationManager = (LocationManager)
getSystemService(Context.LOCATION_SERVICE);
```

RSA Digital Signature Class

The Bouncy castle crypto API [40] was used in generating the digital signature. Users generate digital signatures by encrypting the SHA1 hash of their locations using the RSA private key. The other users verify the signature by generating SHA1 hash of the received location and comparing it with the hash value in the digital signature. A graphical illustration of this process is given in Fig. 6.4. The first code snippet next shows the signature generation process and the second code snippet shows the verification process.

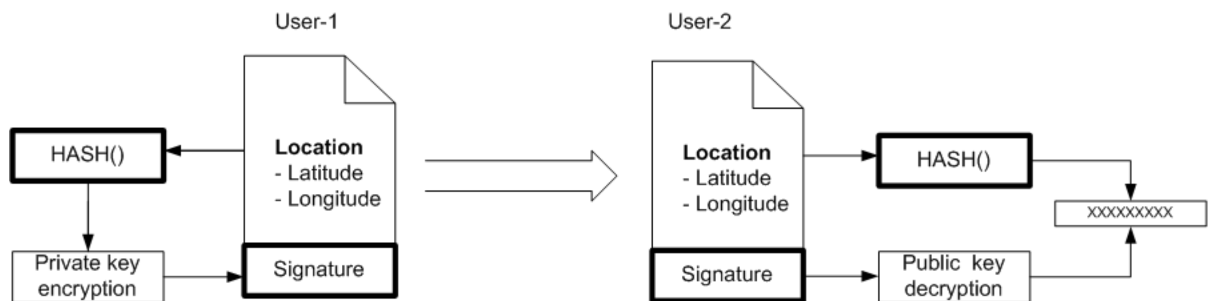


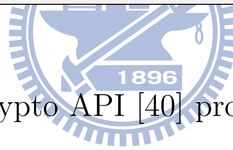
Figure 6.4: Client RSA digital signature generation and verification.

```
public static byte[] signLocation(String location)
 throws Exception {
 Security.addProvider(new
 org.bouncycastle.jce.provider.BouncyCastleProvider());
 KeyFactory keyFactory =
 KeyFactory.getInstance("RSA", "BC");
 RSAPrivateKey privKey =
 RSAPrivateKey) keyFactory.generatePrivate(privKeySpec);
 Signature signature=Signature.getInstance("SHA1withRSA","BC");
 signature.initSign(privKey, new SecureRandom());
 signature.update(location.getBytes());
 byte[] sigBytes = signature.sign();
 return sigBytes;
}
```

```
public static boolean signVerify
(byte[] sigBytes, String location)
throws Exception {
Security.addProvider(new
org.bouncycastle.jce.provider.BouncyCastleProvider());
KeyFactory keyFactory = KeyFactory.getInstance
("RSA", "BC");
RSAPublicKeySpec pubKeySpec = new
RSAPublicKeySpec(modulus,exponentPub);
RSAPublicKey pubKey = (RSAPublicKey)
keyFactory.generatePublic(pubKeySpec);
    Signature signature = Signature.getInstance
    ("SHA1withRSA","BC");
signature.initVerify(pubKey);
signature.update(location.getBytes());
boolean verificationResult = signature.verify(sigBytes);
return verificationResult;
}
```

The RSA key was generated using the JAVA code given in the next code snippet. Two pairs of keys were generated and placed in the user 1's and user 2's JAVA codes.

```
public static void main(String[] args) throws Exception {
Security.addProvider
(new org.bouncycastle.jce.provider.BouncyCastleProvider());
KeyPairGenerator keyGen = KeyPairGenerator
.getInstance("RSA", "BC");
keyGen.initialize(1024, new SecureRandom());
KeyPair keyPair = keyGen.generateKeyPair();
KeyFactory fact = KeyFactory.getInstance("RSA");
RSAPublicKeySpec pub = fact.getKeySpec(keyPair.getPublic(),
RSAPublicKeySpec.class);
RSAPrivateKeySpec priv = fact.getKeySpec(keyPair.getPrivate(),
RSAPrivateKeySpec.class);
```



The Bouncy castle crypto API [40] produces a signature which is a byte array of 1024 bits. When this is converted in to a string in order to be transmitted using the TCP socket, the bits are encoded according to some character format like UTF-8, UCS-2 etc. These encoded strings results in modifications to the original bits of the digital signature. Therefore we have implemented a method in order to transmit the bits of the digital signature without any modification over the network. The first step is to convert all bytes(128Bytes) in to signed decimal integers.

01111000 00001010 00101101 = 120 10 45

Then the numbers are concatenated with the "," as a separator.

120,10,45

Then it is converted in to a character string and transmitted over the network. At the client end, the string is split at each "," and the individual integers are obtained. Then these are converted in to binary and concatenated to reconstruct the original byte array.

120 10 45 = 01111000 0000s1010 00101101

By using this method we are able to transmit the original digital signature bits without being altered during the transmission.

6.2.2 Implementation at the Server Side

The server codes consist of the main class which contains the TCP socket and the homomorphic XOR class, which performs the XOR homomorphic addition on the data received from the clients. The complete JAVA code of the server implementation is given in Appendix G. Fig. 6.5 shows the flowchart of the program implemented at the server side.

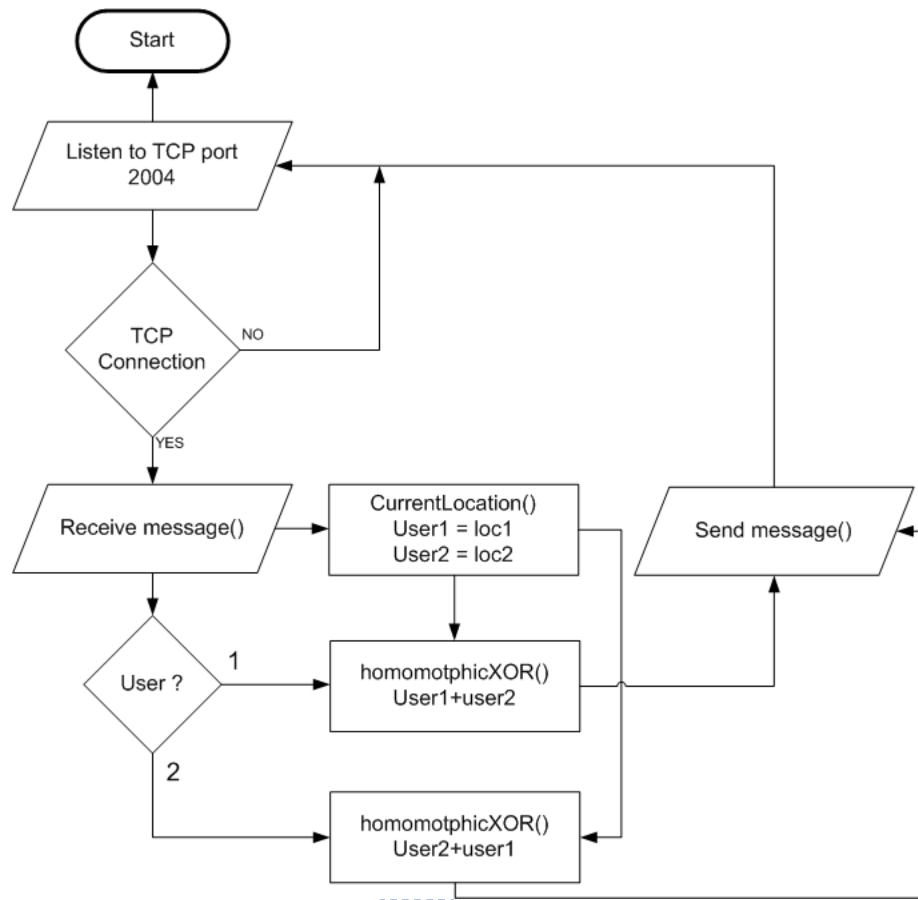


Figure 6.5: The server programme flow.

Server TCP Socket

TCP socket code snippet [48] at the server side is given below. The server socket binds to the TCP port 3105 of the server Internet protocol(IP) interface.

```
Provider(){}
void run(){
try{
providerSocket = new ServerSocket(3105, 10);
System.out.println("Waiting for connection");
connection = providerSocket.accept();
System.out.println("Connection received from " +
 connection.getInetAddress().getHostName());
out = new ObjectOutputStream(connection.getOutputStream());
in = new ObjectInputStream(connection.getInputStream());
do{
try{
message = (String)in.readObject(); }
}while(!message.equals("bye"));}
catch(IOException ioException){
ioException.printStackTrace();}
```

Homomorphic XOR Operation

The homomorphic XOR operation is performed in the server by adding the received user_1's and user_2's location data together. A **for** loop is used for adding the integer valued cipher-text. The code snippet is given next.

```
for(int i=0;i<66;i++)
{
userLoc1Big = new BigInteger(a1.elementAt(i));
userLoc2Big = new BigInteger(b1.elementAt(i));
outputBigArray.add((userLoc2Big.add(userLoc1Big)).toString());
}
```

6.3 Cipher-Text Growth Issue

Selecting Parameters

The encryption scheme in [2] can produce very large cipher-text based on the selection of p, q and r. The parameter selection is a trade-off between security and system performance. For the best security, parameters should be selected according to (6.1)(6.2)(6.3).

$$p \in [2^{(\eta-1)}, 2^{\eta}), \tag{6.1}$$

$$r \approx 2^{\sqrt{\eta}}, \tag{6.2}$$

$$q \approx 2^{\eta^3}, \tag{6.3}$$

where η is the security parameter.

For protection against brute force attack we have selected the key length of 64 bits. That is $\eta = 64$. Therefore, according to (6.2)(6.3), $r = 8$ bits and

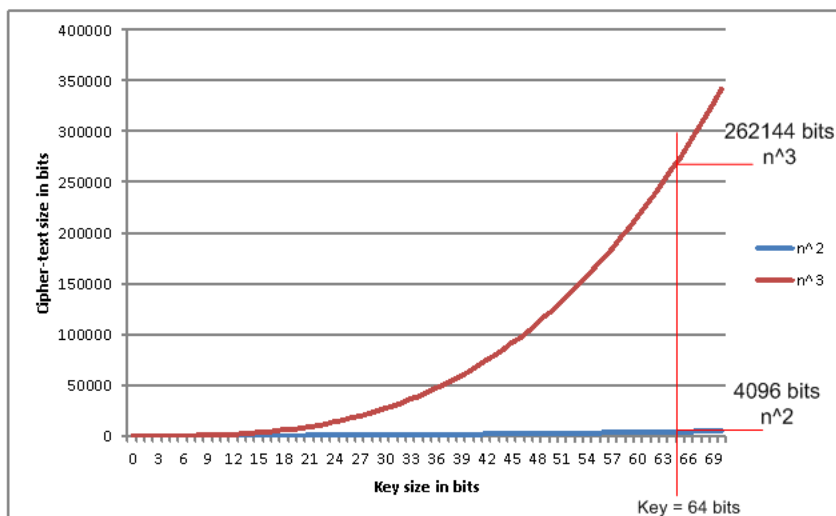


Figure 6.6: Encryption key size vs cipher-text growth.

$q = 262144$ bits. Since q is the largest integer in the encryption formula, the resulting cipher-text will be around the size of 262144 bits per one bit of plain text. The two's complement representation of GPS coordinates will take 64 bits, therefore 64 bits of plain text will become 16777216 bits (16Mbits) of cipher-text. These parameters were tested during the implementation. Fig. 6.6 gives a graph which shows the cipher-text size increase with the length of the encryption key. The code snippet below shows the implementation of encryption and decryption formulas implemented using JAVA bigintger type.

```
p = new BigInteger("18446744073709551617");
// p is 64 bit
q = new BigInteger(262144,rand);
r = new BigInteger(8,rand);

BigInteger c1 = ((p.multiply(q)));
BigInteger c2 = c1.add(m);
BigInteger c3 = r.multiply(BigInteger.valueOf(2));
c = c2.add(c3);
// "c" is cipher-text
BigInteger mm = c.mod(p).mod(BigInteger.valueOf(2));
// "mm" is plain-text obtained by
decrypting the cipher-text
```

Two problems arise due to the large cipher-text size. First, there is a 16MB memory allocation limit per application in Android operating system [49]. The total memory usage comes close to this limit when other components of the application like google maps, TCP socket etc are added in to the equation. The second problem is the bandwidth available over HSDPA, 3G and GPRS links. HSDPA in smartphones usually supports 7.2 Mbps downlink and 384 Kbps uplink with best radio reception. Therefore the mobile client will take at least 42 seconds ($16777216/(384 \times 1024)$) for a location update. Due to these practical considerations, we have loosened up the selection of q to be $q = 2^{\eta^2}$. With this selection of q, the location update size reduces significantly. One bit of plain-text will become 4096 bits of cipher-text, therefore 64 bits will become 262144 bits (32 KB), which is a

reasonable value for the mobile device memory and HSDPA, 3G and GPRS data links.

JAVA regular data types like double and float can handle 64 bit numbers [50], but this is not sufficient for the requirement of implementing the encryption and decryption functions. Therefore we have used the data type "biginteger". This data type can handle very large integers. The upper limit is only limited by the amount of memory available for the process. we have used the "BigInteger(int numBits, Random rnd)" method to generate random values of a specific bit length for q and r as shown in the next code snippet.

```
q = new BigInteger(262144,rand);
r = new BigInteger(8,rand);
```

6.4 Time Complexity Issue

In this subsection, we discuss the time complexity issue of the proposed XOR homomorphic encrypted secure location sharing protocol at the client side and the server side.

Time Complexity at the Client

The time complexity of the proposed protocol's mathematical operations at the client is analysed below. The number of users is considered to be the input of this analysis.

If there are m-bit location data, each user will perform $m(n-1)$ XOR operations in a group of n users. Specifically that is ,

2 users $\longrightarrow m$ XOR operations,

3 users $\longrightarrow 2m$ XOR operations,

.

n users $\longrightarrow (n-1)m$ XOR operations,

Denote the time complexity of the proposed algorithm as $T_{client}(n)$. Then, $T_{client}(n) = O(n)$

Time Complexity at the Server

Let each user's GPS location data be expressed by m digits. If each bit is encrypted and become c-bits. For

2 users $\longrightarrow c$ additions,

3 users $\longrightarrow 3c$ additions,

4 users $\longrightarrow 6c$ additions,

n users

$$\begin{aligned}
&\longrightarrow \frac{n!}{2!(n-2)!} \times c, \\
&\longrightarrow \frac{(n-2)! \times (n-1) \times n}{2! \times (n-2)!} \times c, \\
&\longrightarrow \frac{(n-1) \times n}{2!} \times c, \\
&\longrightarrow O(n^2). \qquad (6.4)
\end{aligned}$$

1896 ES

Table 6.1: Server computation reduction in the proposed protocol.

.	**Encryption**	**Decryption**	**XOR/Addition**
Traditional method	2	2	0
Proposed protocol	0	0	1

Thus, the time complexity of the proposed algorithm at the server becomes $T_{Server}(n) = O(n^2)$.

Cryptographic Operations Complexity Consideration

In the traditional method, the number of cryptographic operations increase with the number of users. The server need to perform "n" number of decryptions and "n" number of encryptions in order to share locations among the users. In the proposed scheme, the server does not need to perform any cryptographic operation. The server simply adds the received values together and broadcast to all the users. A comparison of the cryptographic complexity of traditional approach and the proposed protocol is given in Fig. 6.7, and the Table 6.1 shows a number of cryptographic operations at the server for traditional method and the proposed protocol in the two-user case. As observed in Fig. 6.7, for 10 users, the overall system require 120 cryptographic operations in traditional method compared to the 100 cryptographic operations required by the proposed method. Therefore the proposed scheme reduce the cryptographic computation burden on the system by 16.67% at 10 users.

In the calculation below, the number of cryptographic operations per-

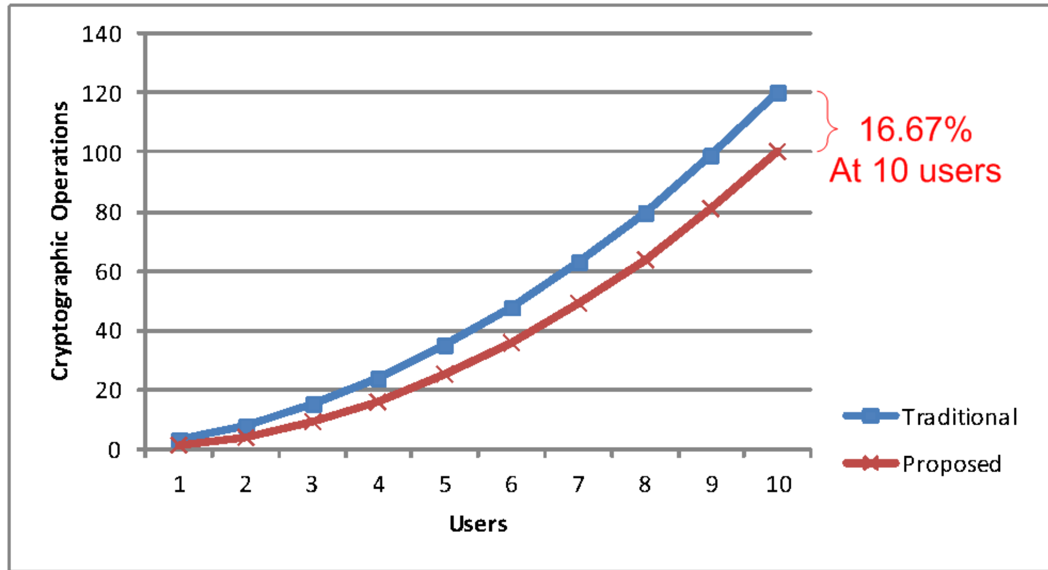


Figure 6.7: Number of users vs cryptographic operations at the server.

formed in the overall system is considered as the metric. We have made the assumptions that each cryptographic operation is considered as one unit of CPU time and the XOR homomorphic operation is not a cryptographic operation due to the fact that it is merely a numerical addition at the server.

In the traditional approach for n users, the client require one encryption and $n-1$ decryptions. At the server, one encryption and one decryption is required for each user. Therefore the total cryptographic operations for n users is $(1+(n-1)) \times n + (1+1) \times n = n^2 + 2n$.

In the proposed solution, for n users, each client requires one encryption and $n-1$ decryptions. The server does not require to perform any cryptographic operations. Therefore the total number of cryptographic operations for n users is $(1+(n-1)) \times n = n^2$.

6.5 Bandwidth Issue

In the proposed algorithm, the bandwidth is required at the server for sending and receiving messages. In traditional location sharing method, in the case of n users, each user will send his or her location to the server and the server need to send $(n-1)$ other users' locations to each user. This results in a significant amount of traffic. In the proposed solution, the users' locations are combined using the XOR homomorphic encryption technique and broadcasted to the whole group.

The metric used for this analysis is the number of messages need to be communicated [51] per user. We have made these assumptions in the process, each message has a fixed length (This is true due to the fact that the GPS coordinates need to be converted to fixed length two's complement in both schemes), underlying cryptographic algorithm generates the same number of bits per one bit of plain text and a protocol like Internet relay chat (IRC) is used with the proposed scheme in order to broadcast messages to user groups. Table 6.2 shows a comparison of the number of messages in traditional method vs the proposed protocol in two-user case.

Formula :

n = number of users

Traditional method bandwidth usage = $n \times n$,

Proposed protocol bandwidth usage = $n \times (n-1)/2$.

The number of messages is plotted against the the number of users in Fig. 6.8. According the result, it is observed that for 10 users, the proposed

Table 6.2: Bandwidth overhead reduction of two-user case using the proposed protocol.

.	**Number of message**
Traditional method	4
Proposed protocol	3

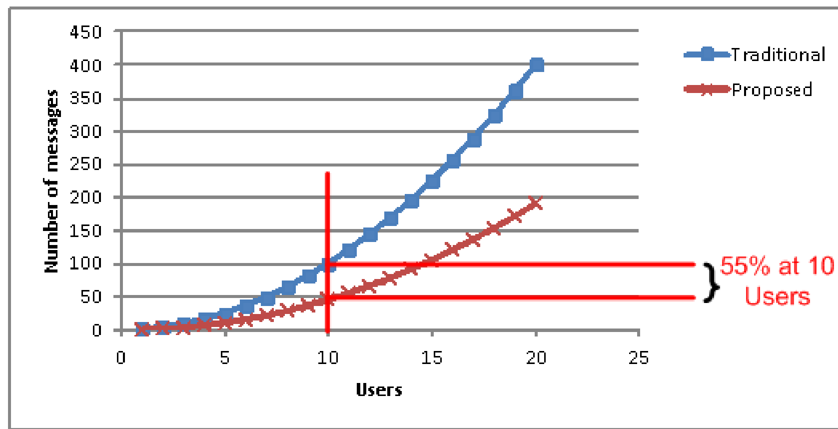


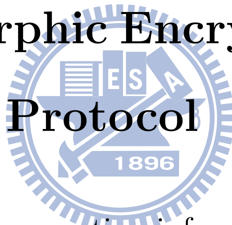
Figure 6.8: Number of users vs server bandwidth utilization.

scheme only require 45 messages compared to the 100 messages required by the traditional method, which is a 55% bandwidth saving.

Chapter 7

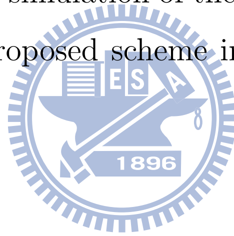
Conclusions

7.1 XOR Homomorphic Encrypted Secure Location Sharing Protocol



When outsourcing data to cloud computing infrastructure, one of the main concerns for any organization is security. For this end, all data are encrypted and stored in cloud computing infrastructure. The problem arises when the organization requires the virtual servers in the cloud computing infrastructure to perform computations on encrypted data. In this thesis we have discussed the methods of performing calculations on the cipher-text, and its applications for location sharing services in cloud computing. As described in the literature survey in Chapter 2, we have analysed several leading solutions to this problem and has picked homomorphic encryption as the best fit for sharing location information securely, in untrusted cloud computing infrastructure.

We have proposed the “XOR homomorphic encrypted secure location sharing protocol” in Chapter 4. This security protocol can be used for supporting location based cloud computing applications to provide data privacy using homomorphic encryption scheme [2]. Thus, location sharing among users can preserve confidentiality, authenticity, integrity and non-repudiation. The proposed protocol requires only a limited number of additions and multiplications on the cipher text by the cloud computing servers. In addition to homomorphic encryption, a one-way hash function and an asymmetric key encryption scheme are used to provide authenticity, integrity and non-repudiation. In Chapter 5 we have verified the security of proposed protocol using two industry standard secure protocol analysers namely AVISPA and ProVerif. A simulation of the protocol using MATLAB and the implementation of the proposed scheme in an Android application are discussed in Chapter 6.



7.2 Suggestions for Future Research

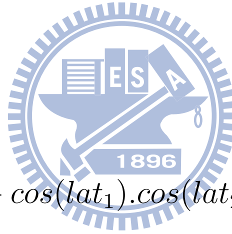
The fully homomorphic scheme proposed by Gentry [1] prove the existence of cryptographic algorithms that can allow any mathematical operation on the cipher-text. At this moment the fully homomorphic encryption scheme impractical to be implemented in all application because of its high computation overhead and the growth of the cipher text size with each operation. We provide the following suggestions to further continue our work.

- Extend our scheme and perform further mathematical operations on the cipher-text:

One of the interesting things we analysed in our research is to calculate distance between two users at the server. The challenge is the Haversine formula for calculating the distance between two GPS coordinates [52]. This formula includes functions such as $sin, cos, atan2$ and the $squareroot$, which need to be defined in the homomorphic domain. One possible solution would be to replace the GPS cordinates with a cartician system like UTM coordinates [53], which will simplify the distance calculation to Pythagoras theorem.

Haversine formula

$$R = \textbf{earths radius (mean radius = 6,371km)}$$



$$\begin{aligned} \Delta lat &= lat_1 - lat_2 \\ \Delta long &= long_1 - Long_2 \\ a &= sin^2(\Delta lat/2) + cos(lat_1).cos(lat_2).sin^2(\Delta long/2) \\ c &= 2 \times atan2(\sqrt{a}, \sqrt{(1-a)}) \\ d &= R \times c \end{aligned} \tag{7.1}$$

- Application of homomorphic encryption in databases hosted in cloud computing infrastructure:

Another popular topic we looked into during our research was database security in cloud computing. Most methods descried in the literature survey are inspired by the requirement of performing SQL querries over the ciphertext [3, 6].

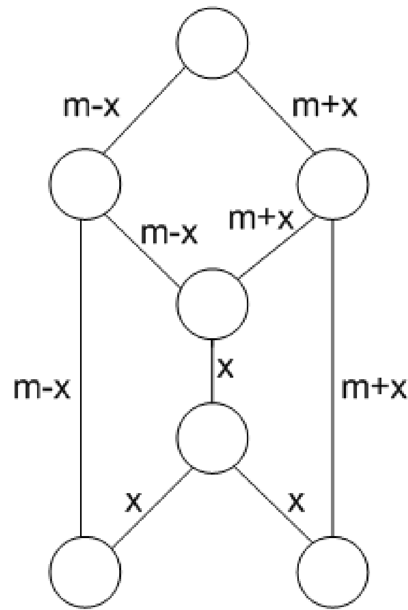


Figure 7.1: Using network coding for security.

ES A 1896

- Applying homomorphic encryption in network coding:

The reference [54] gives several applications of network coding in order to achieve security. What makes this interesting is the fact that encryption is not used in these applications. Several variations of the original message are transmitted through several paths in the network, trusted nodes on the way perform predefined mathematical operations on the messages so that the receiver is provided with a set of messages which will help him to extract the original message. This process is illustrated in Fig. 7.1. The main concern in this architecture is that the nodes need to be trusted. Using homomorphic encryption to overcome the need for trust could be an interesting research area.

Bibliography

[1] C. Gentry, "Fully Homomorphic Encryption Using Ideal Lattices," *The 41st ACM Symposium on Theory of Computing (STOC)*, pp. 169–178, May 2009.

[2] M. van Dijk, C. Gentry, S. Halevi, and V. Vaikuntanathan, "Fully Homomorphic Encryption Over the Integers," *The 29th Annual International Conference on the Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, pp. 24–43, May 2010.

[3] H. Hacigümüş, B. Iyer, C. Li, and S. Mehrotra, "Executing SQL Over Encrypted Data in the Database-Service-Provider Model," *ACM SIGMOD International Conference on Management of Data*, pp. 216–227, May 2002.

[4] G. Aggarwal, M. Bawa, P. Ganesan, H. Garcia-Molina, K. Kenthapadi, R. Motwani, U. Srivastava, D. Thomas, "Two Can Keep A Secret: A Distributed Architecture for Secure Database Services," in *The Conference on Innovative Data Systems Research (CIDR)*, pp. 186–199, Jan. 2005.

[5] P. F. Oliveira, L. Lima, T. T. V. Vinhoza, J. Barros, and M. Médard, "Trusted Storage over Untrusted Networks," *IEEE GLOBECOM*, pp. 1–5, Dec. 2010.

[6] Agrawal, Kiernan, Srikant and Xu, "Order Preserving Encryption for Numeric Data," *ACM SIGMOD The Interantional Conference on Management of Data*, pp. 563–574, May 2004.

[7] D. X. Song, D. Wagner, and A. Perrig, "Practical Techniques for Searches on Encrypted Data," *IEEE Symposium on Security and Privacy*, pp. 44–55, May 2000.

[8] R. L. Rivest, L. Adleman, and M. L. Dertouzos, "On Data Banks and Privacy Homomorphisms," *Foundations of Secure Computation*, pp. 169–180, 1978.

[9] Wikipedia the Free Encyclopedia. [Online]. Available: http://en.wikipedia.org/wiki/Main_Page

[10] Boneh, Goh, and Nissim, "Evaluating 2-DNF Formulas on Ciphertexts," *Second Theory of Cryptography Conference (TCC)*, pp. 325–341, Feb. 2005.

[11] S. Katti, H. Rahul, W. Hu, D. Katabi, M. Médard, and J. Crowcroft, "XORs in the Air: Practical Wireless Network Coding," *IEEE/ACM Transactions on Networking*, Vol.16, No. 3, pp. 497–510, 2008.

[12] Android. [Online]. Available: http://www.android.com/

[13] Ostrovsky and Shoup, "Private Information Storage," *Annual Symposium on Theory of Computing*, Vol. 16, pp. 294–303, May 1997.

[14] A. Solanas and A. Martínez-Ballesté, "Privacy Protection in Location-Based Services Through a Public-Key Privacy Homomorphism," *European Public Key Infrastructure Workshop*, pp. 362–368, June 2007.

[15] P. Ruppel, G. Treu, A. Küpper, and C. Linnhoff-Popien, "Anonymous User tracking for location-based community services," *Second International Workshop on Location and Context Awareness (LoCA)*, pp. 116–133, May 2006.

[16] M. Gruteser and D. Grunwald, "Anonymous Usage of Location-Based Services through spatial and temporal cloaking," *The First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[17] S. Goldwasser and S. Micali, "Probabilistic Encryption and how to Play Mental Poker Keeping Secret all partial information," *Symposium on Theory of Computing*, pp. 365–377, May 1982.

[18] O. Ugus, D. Westhoff, R. Laue, A. Shoufan and S. A. Huss, "Optimized Implementation of Elliptic Curve based additive homomorphic encryption for wireless sensor networks," *An Informal publication in Computing Research Repository*, March 2009.

[19] Paillier, "Public-Key Cryptosystems based on Composite Degree Residuosity classes," *International Conference on the Theory and Applications of Cryptographic Techniques*, pp. 223–238, May 1999.

[20] E. Dawson and S. Vaudenay, "A New Public-Key Cryptosystem as Secure as Factoring," *First International Conference on Cryptology*, pp. 308–318, June 2005.

[21] O. Regev, "New Lattice-Based Cryptographic Constructions," *The Journal of the ACM*, Vol. 5, No. 6, pp. 899–942, Feb. 2004.

[22] Christian Haack, Verification of Security Protocols, [Online]. Available: http://www.cs.ru.nl/~chaack/teaching/2IF02-Spring08/

[23] P. Ryan and S. A. Schneider, *Modelling and Analysis of Security Protocols*, Addison-Wesley, 2001.

[24] Q. Chen, C. Zhang, and S. Zhang, *Secure Transaction Protocol Analysis*, Springer, 2008.

[25] D. Dolev and C. Yao, "On the Security of Public Key Protocols," *IEEE Transactions on Information Theory*, Vol. 29, No. 2, pp. 198–207, Jan. 1983.

[26] A. Armando, D. Basin , Y. Boichut, Y. Chevalier, and al Et, "The AVISPA Tool for the Automated Validation of Internet Security Protocols and Applications," *International Conference on Computer Aided Verification*, pp. 1–5, July 2005.

[27] Cassis, Loria and Nancy, "Verification of Cryptographic Protocols: Techniques and tools," *French/Japanese Symposium on Computer Security*, Sept. 2005.

[28] D. Oheimb, "High Level Protocol Specification Language". [Online]. Available:http://www.avispa-project.org/package/tutorial.pdf

[29] A. Armand, The High Level Protocol Specification Language. [Online]. Available: http://www.avispa-project.org/delivs/2.1/d2-1.pdf

[30] T. Genet, Span Tool. [Online]. Available: http://www.irisa.fr/celtique/genet/span/

[31] K. Korovin, Linear Temporal Logic LTL. [Online]. Available: http://www.cs.man.ac.uk/~korovink/cs2142/chapter14.pdf

[32] Basin, Modersheim, and Vigano, "An On-The-Fly Model-Checker for Security Protocol Analysis," *European Symposium on Research in Computer Security*, pp. 253–270, Dec. 2003.

[33] K. L. Mcmillan, "Interpolation and SAT-Based Model Checking," *International Conference on Computer Aided Verification*, pp. 1–13, Oct. 2003.

[34] A. Armando and L. Compagna, "SAT Based Model-Checking for Security Protocols Analysis," *International Journal of Information Security*,Vol. 7, No. 1, pp. 3–32, 2008.

[35] H. Comonm, M Dauchet, "Tree Automata Techniques and Applications". [Online]. Available: http://www.cs.aau.dk/~srba/courses/FS-07/tata.pdf

[36] B. Blanchet, "Proverif: Cryptographic Protocol Verifier in the Formal Model". [Online]. Available: http://www.proverif.ens.fr

[37] B. Blanchet, "Automatic Proof of Strong Secrecy for Security Protocols," *IEEE Symposium on Security and Privacy*, Oakland, California, May 2004.

[38] Java Tutorials. [Online]. Available: http://download.oracle.com/javase/tutorial/

[39] Android SDK. [Online]. Available: http://developer.android.com/sdk/index.html

[40] The Bouncy Castle Crypto APIs. [Online]. Available: http://www.bouncycastle.org/

[41] Eclipse IDE. [Online]. Available: http://www.eclipse.org/

[42] E. Burnette, *Hello, Android: Introducing Google's Mobile Development Platform*, Pragmatic Bookshelf, 2009.

[43] HTC Hero Mobile Handset. [Online]. Available: http://www.htc.com/www/product/hero/specification.html

[44] Android IS 2.1. [Online]. Available: http://developer.android.com/sdk/android-2.1.html

[45] L. Vogel, "Location API and Google Maps in Android - tutorial". [Online]. Available:http://www.vogella.de/articles/AndroidLocationAPI/article.html

[46] Google Maps External Library. [Online]. Available: http://developer.android.com/guide/topics/location/index.html

[47] D. Rorvik, Two's Complement Representation for Signed Integers. [Online]. Available: http://academic.evergreen.edu/projects/biophysics/technotes/program/2s_comp.htm

[48] K.Zerioh, Sockets Example. [Online]. Available: http://zerioh.tripod.com/ressources/sockets.html

[49] Android Memory Heap. [Online]. Available: http://developer.android.com/resources/articles/avoiding-memory-leaks.html

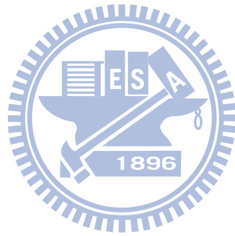
[50] Java Primitive Data Types. [Online]. Available: http://download.oracle.com/javase/tutorial/java/nutsandbolts/datatypes.html

[51] K. Welke and K. Rechert, "Spontaneous Privacy-Aware Location Sharing," *Joint Conferences on Pervasive Computing (JCPC)*, pp. 395–398, Dec 2009.

[52] E. Williams, "Aviation Formulary". [Online]. Available: http://williams.best.vwh.net/avform.htm

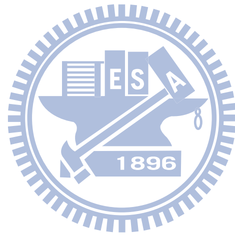
[53] UTM Map Coordinate System. [Online]. Available:http://www.maptools.com/UsingUTM/

[54] L. Robert. Network Coding - A Paradigm Shift in Data Transport. [Online]. Available: http://www.ie.cuhk.edu.hk/people/bobli.html

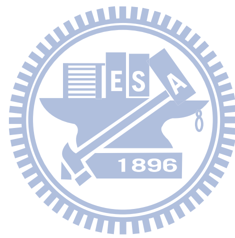


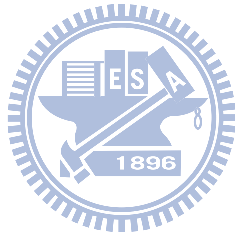
Vita

W.K.Ruwan Indika Prasanna was born in Galle, Sri Lanka in 1981. He received his B.Sc degree from the Electronic and Telecommunication Engineering Department at University of Moratuwa in Sri Lanka in 2005. From September 2007 to January 2011, he worked on his master degree in the Wireless Systems Laboratory of the Department of Communications Engineering at National Chiao Tung University, Taiwan. His research interests are in the fields of secure communication protocols and network security. You can contact him via e-mail: ruwanindika@gmail.com.



Appendices





Appendix A

Proposed Protocol in HLPSL

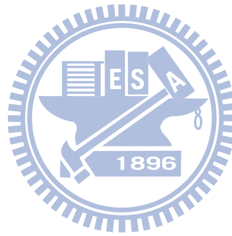
```
role alice(A,B,S : agent,
           K          : symmetric_key,
           Hash     : hash_func,
           Ka,Kb : public_key,
           Snd,Rcv : channel (dy)) played_by A def=

        local
                State :nat,
                Na,Nb    : message


        init State := 0

        transition
```

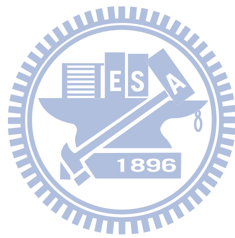
```
        1. State = 0 /\ Rcv(start) =|>
          State':= 2 /\ Na' := new()
                         /\ Snd(A,{Na'}_K,{Hash(A,Na')}_inv(Ka))

        2. State = 2 /\ Rcv(A,{Na'}_K.{Nb'}_K,{Hash(A,Na')}_inv(Ka),
                            {Hash(B,Nb')}_inv(Kb))   =|>
         State':= 4 /\ secret (Na,na,{A,B})



end role

role bob(A,B,S : agent,
        K       : symmetric_key,
        Hash      : hash_func,
        Ka,Kb         : public_key,
        Snd,Rcv : channel (dy)) played_by B def=



        local
                State :nat,
                Na,Nb    : message

        init State := 1

        transition
                1. State = 1 /\ Rcv(start) =|>
                   State':= 3 /\ Nb':=new()
```

```
                                     /\ Snd(B,{Nb'}_K,{Hash(B,Nb')}_inv(Kb))


   2. State = 3 /\  Rcv(B,{Na'}_K.{Nb'}_K,{Hash(A,Na')}_inv(Ka),
                    {Hash(B,Nb')}_inv(Kb)) =|>
      State':= 5 /\  secret (Nb,nb,{A,B})


end role


role server(A,B,S : agent,
                   Snd,Rcv : channel (dy)) played_by S def=

        local
                State :nat,
                X1    : message,
                X2    : message,
                SIG1  : message,
                SIG2  : message


        init State := 7

        transition
```

```
          1. State = 7 /\ Rcv(A,X1',SIG1) /\ Rcv(B,X2',SIG2)=|>
             State':= 9

          2. State = 9 /\ Rcv(A,X1',SIG1)
                       /\ Rcv(B,X2',SIG2)=|>
             State':= 11 /\ Snd(A,X1'.X2',SIG1,SIG2)
                         /\ Snd(B,X1'.X2',SIG1,SIG2)


end role

role session(A, B, S : agent,K : symmetric_key,
              Hash : hash_func, Ka : public_key,
              Kb : public_key)
def=
local
SAS, RSA,
SBS, RSB,
SSA, RAS,
SSB, RBS : channel (dy)

composition
alice (A, B, S, K,Hash, Ka,Kb, SAS, RSA)
/\ server(A, B, S, SSA, RAS)
/\ bob (A, B, S, K, Hash, Ka,Kb, SBS, RSB)

end role
```

```
role environment()
def=
const
  a, b, s : agent,
  k1 , kis      : symmetric_key,
  ka , kb , ki : public_key,
  h             : hash_func,
  na,nb   : protocol_id

  intruder_knowledge = {a, b, s,kis,ki,ka,kb}

composition
session(a,b,s,k1,h,ka,kb)
/\session(i,b,s,kis,h,ki,kb)
/\session(a,i,s,kis,h,ka,ki)
/\session(a,b,i,kis,h,ka,kb)


end role

goal

secrecy_of na,nb

end goal

environment()
```

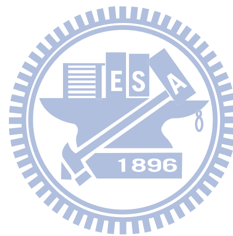




Appendix B

Details of the Verification Result using AVISPA

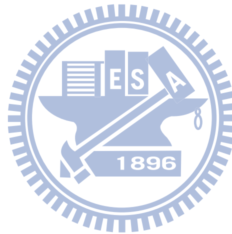
B.1 OFMC



```
OFMC
Version of 2006/02/13
SUMMARY
  SAFE
DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
PROTOCOL
  /home/avispa/web-interface-computation/./tempdir/workfile41o3Wk.if
GOAL
  as_specified
BACKEND
```

```
  OFMC
COMMENTS
STATISTICS
  parseTime: 0.00s
  searchTime: 23.66s
  visitedNodes: 19683 nodes
  depth: 18 plies
```

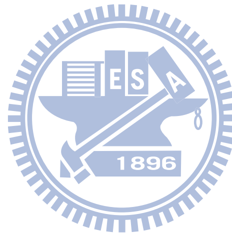
B.2 CL-AtSe



```
SUMMARY
  SAFE

DETAILS
  BOUNDED_NUMBER_OF_SESSIONS
  TYPED_MODEL

PROTOCOL
  /home/avispa/web-interface-computation/./tempdir/workfile41o3Wk.if

GOAL
  As Specified

BACKEND
  CL-AtSe
```

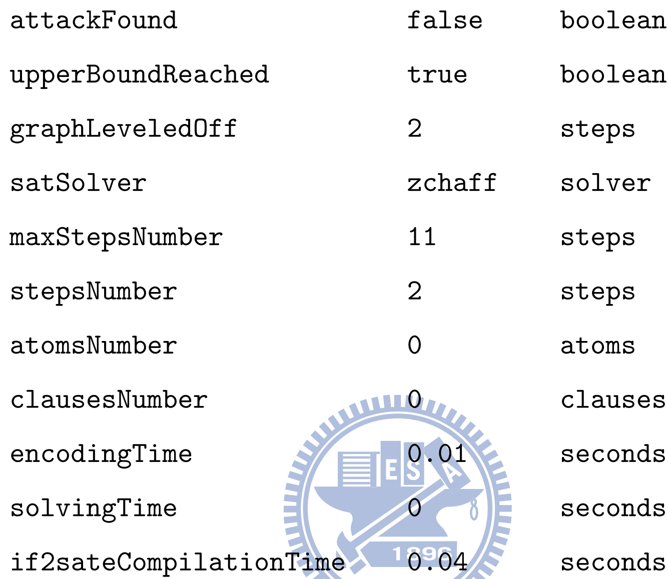
```
STATISTICS

  Analysed    : 0 states
  Reachable   : 0 states
  Translation: 0.02 seconds
  Computation: 0.00 seconds
```

B.3 CL-AtSe



```
SUMMARY
  SAFE

DETAILS
  STRONGLY_TYPED_MODEL
  BOUNDED_NUMBER_OF_SESSIONS
  BOUNDED_SEARCH_DEPTH
  BOUNDED_MESSAGE_DEPTH

PROTOCOL
  workfile41o3Wk.if

GOAL
  %% see the HLPSL specification..

BACKEND
```

```
  SATMC

COMMENTS

STATISTICS
  attackFound                 false     boolean
  upperBoundReached           true      boolean
  graphLeveledOff             2         steps
  satSolver                   zchaff    solver
  maxStepsNumber              11        steps
  stepsNumber                 2         steps
  atomsNumber                 0         atoms
  clausesNumber               0         clauses
  encodingTime                0.01      seconds
  solvingTime                 0         seconds
  if2sateCompilationTime      0.04      seconds

ATTACK TRACE
  %% no attacks have been found..
```

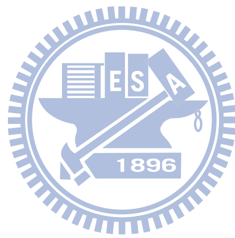


B.4 TA4SP

```
SUMMARY
   INCONCLUSIVE
```

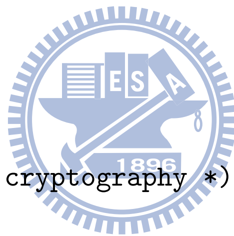
```
DETAILS
   NOT_SUPPORTED

PROTOCOL
   /home/avispa/web-interface-computation/./tempdir/workfile41o3Wk.if

GOAL
   SECRECY

BACKEND
   TA4SP

COMMENTS
   Some rules may be not fired so TA4SP does not do the verification.
```

```
STATISTICS
   Translation: 0.00 seconds
```



Appendix C

Proposed Protocol in Spi-Calculus

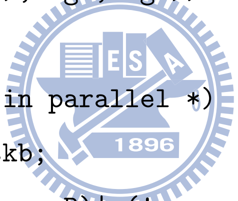


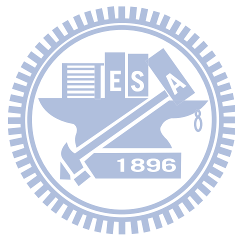
```
(* probablistic Shared key cryptography *)
fun encrypt/3.
reduc decrypt(encrypt(x,y,r),y) = x.

(* public key cryptography *)
fun pk/1.
fun Pencrypt/2.
reduc Pdecrypt(Pencrypt(x,y),y) = x.

(* Hash function *)
fun hash/1.

(* Homomorphic addition function *)
```

```
fun addH/2.
reduc deduct(addH(p,q),q) = p;
      deduct(addH(p,q),p) = q.

(* Signature Aggregation function *)
fun sigAG/2.

(* Secrecy assumptions *)
not k.                (* shared symetric key *)
not ska.        (* A's private key for signatures *)
not skb.        (* B's private key for signatures *)

(* Dolev-Yao model network with omnipotent attacker *)
free net.
private free secretA,secretB.

(* noninterf = prove strong secracy *)
noninterf secretA,secretB.
noninterf secretA.
noninterf secretB.

(* Test whether "secretA" and "secretB" are secret *)
query attacker:secretA.
query attacker:secretB.

let processA =
        new na;
        out(net, (encrypt((secretA), k,na) ,
```

ES A 1896

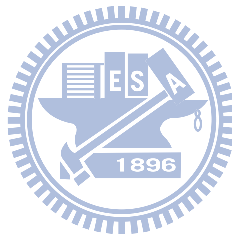
```
        Pencrypt(hash(secretA),ska));
        in(net,a).

let processB =
        new nb;
        out(net, (encrypt((secretB), k,nb) ,
        Pencrypt(hash(secretB),skb));
        in(net,b).

let processS =
        in(net,(s1,sig1));
        in(net,(s2,sig2));
        out(net,(addH(s1,s2),sig1,sig2)).

(* run the three processes in parallel *)
process new k;new ska;new skb;
        ((!processA) |(!processB)| (!processS))  (* !P = p | p | p...*)
```





Appendix D

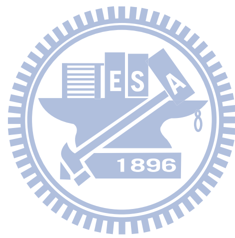
Verification Result using PROVERIF



```
Process:
new k_13;
new ska_14;
new skb_15;
(
    {8}!
    new na_22;
    {9}out(net, (encrypt((secretA),k_13,na_22),
    Pencrypt(hash(secretA),ska_14)));
    {10}in(net, a_23);
    0
) | (
    {5}!
```

```
    new nb_20;
    {6}out(net, (encrypt((secretB),k_13,nb_20),
    Pencrypt(hash(secretB),skb_15)));
    {7}in(net, b_21);
    0
) | (
    {1}!
    {2}in(net, (s1_16,sig1_17));
    {3}in(net, (s2_18,sig2_19));
    {4}out(net, (addH(s1_16,s2_18),sig1_17,sig2_19));
    0
)

-- Query not attacker:secretB[]
Completing...
ok, secrecy assumption verified: fact unreachable attacker:k_13[]
ok, secrecy assumption verified: fact unreachable attacker:ska_14[]
ok, secrecy assumption verified: fact unreachable attacker:skb_15[]
Starting query not attacker:secretB[]
RESULT not attacker:secretB[] is true.
-- Query not attacker:secretA[]
Completing...
ok, secrecy assumption verified: fact unreachable attacker:k_13[]
ok, secrecy assumption verified: fact unreachable attacker:ska_14[]
ok, secrecy assumption verified: fact unreachable attacker:skb_15[]
Starting query not attacker:secretA[]
RESULT not attacker:secretA[] is true.
-- Non-interference secretB
```

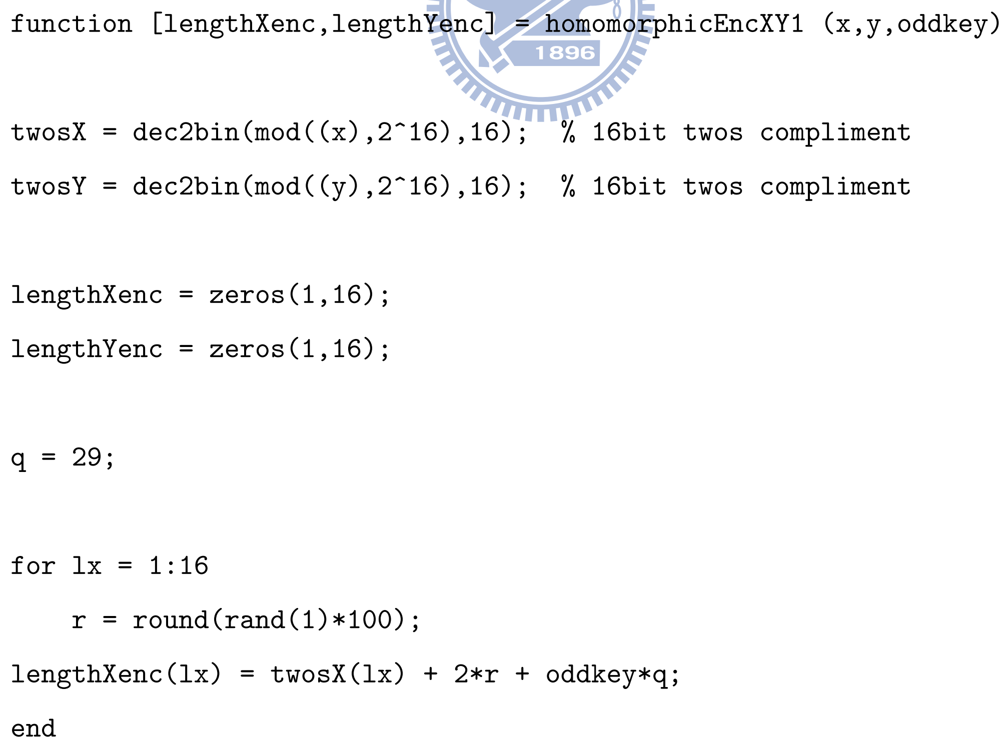
```
Completing...
ok, secrecy assumption verified: fact unreachable attacker:k_13[]
ok, secrecy assumption verified: fact unreachable attacker:ska_14[]
ok, secrecy assumption verified: fact unreachable attacker:skb_15[]
RESULT Non-interference secretB is true (bad not derivable).
-- Non-interference secretA
Completing...
ok, secrecy assumption verified: fact unreachable attacker:k_13[]
ok, secrecy assumption verified: fact unreachable attacker:ska_14[]
ok, secrecy assumption verified: fact unreachable attacker:skb_15[]
RESULT Non-interference secretA is true (bad not derivable).
-- Non-interference secretA, secretB
Completing...
ok, secrecy assumption verified: fact unreachable attacker:k_13[]
ok, secrecy assumption verified: fact unreachable attacker:ska_14[]
ok, secrecy assumption verified: fact unreachable attacker:skb_15[]
RESULT Non-interference secretA, secretB is true (bad not derivable).
```



Appendix E

MATLAB Simulation Code

E.1 Encryption Function



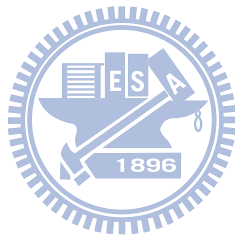
```
function [lengthXenc,lengthYenc] = homomorphicEncXY1 (x,y,oddkey)

twosX = dec2bin(mod((x),2^16),16);  % 16bit twos compliment
twosY = dec2bin(mod((y),2^16),16);  % 16bit twos compliment

lengthXenc = zeros(1,16);
lengthYenc = zeros(1,16);

q = 29;

for lx = 1:16
    r = round(rand(1)*100);
lengthXenc(lx) = twosX(lx) + 2*r + oddkey*q;
end
```

```
for ly = 1:16
   r = round(rand(1)*200);
lengthYenc(ly) = twosY(ly) + 2*r + oddkey*q;
end
```

E.2 Main Function

```
user1X = double((zeros(1,16)));
user1Y = double((zeros(1,16)));

user1X1 = double((zeros(1,16)));
user1Y1 = double((zeros(1,16)));

% user2's location
[x1 y1] = homomorphicEncXY1(120,24,99997);
% user2 encrypts his location and send to the server

% user1's location
[x2 y2] = homomorphicEncXY1(123,33,99997);
% user1 encrypts his location and send to the server

% server does the XOR
x3 = x1 + x2;
y3 = y1 + y2;

% server sends x3 and y3 to both user1 and user2 ,
% they decrypt the message
```

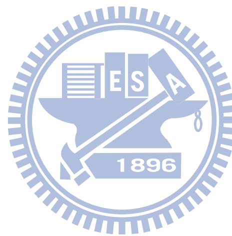
```
x4 = mod(mod(x3,99997),2);
y4 = mod(mod(y3,99997),2);

% user1 calculate user2's location
user1X = dec2bin(mod((123),2^16),16);
user1Y = dec2bin(mod((33),2^16),16);

for i = 1:16
    user1X1(i) = bin2dec(user1X(i));
end

for i = 1:16
    user1Y1(i) = bin2dec(user1Y(i));
end
% user1 does the XOR in order to recover user2's location
user2TwosX = xor(x4,user1X1);
user2TwosY = xor(y4,user1Y1);
% convert the longitude Latitude from twos compliment to Decimal
user2LocationLongitude = twostoDeci(user2TwosX)
user2LocationLatitude = twostoDeci(user2TwosY)
```



Appendix F

Android Mobile Client JAVA Code

F.1 Main Class



```
package xor.homomorphic.encrypted.secure.location.sharing;

import java.math.BigInteger;
import java.util.List;

import android.content.Context;
import android.graphics.drawable.Drawable;
import android.location.Location;
import android.location.LocationListener;
import android.location.LocationManager;
import android.os.Bundle;
```

```
import android.view.Menu;
import android.view.MenuItem;
import android.widget.RelativeLayout;
import android.widget.Toast;

import com.google.android.maps.GeoPoint;
import com.google.android.maps.MapActivity;
import com.google.android.maps.MapController;
import com.google.android.maps.MapView;
import com.google.android.maps.Overlay;
import com.google.android.maps.OverlayItem;

public class ShowMap extends MapActivity {

        private MapController mapController;
        private MapView mapView;
        private LocationManager locationManager;
        private int lat;
        private int lng;
        private OverlayItem overlayitem;
        private OverlayItem overlayitem2;
        private GeoPoint point;
        private GeoPoint point1;
        private GeoPoint point2;
        private HelloItemizedOverlay itemizedoverlay;
        private HelloItemizedOverlay itemizedoverlay2;
        private String[] latCipher;
    private String[] latCipherU;
```

```
    private String[] lngCipher;
private String[] latCipherS = new String[32];
private String[] lngCipherS = new String[32];
private Location locationLast;
private static final int MENU_REFRESH = 0; // Menu button
private static final int MENU_QUIT = 1; // Menu button
private BigInteger good;
private boolean result1;
private byte[] signature;
private String locationForSign;

    private BigInteger keyBig =
    new BigInteger("18446744073709551617");

    public void onCreate(Bundle bundle){
            super.onCreate(bundle);
            setContentView(R.layout.main);
            // bind the layout to the activity
            // create a map view
            RelativeLayout linearLayout =
            (RelativeLayout) findViewById(R.id.mainlayout);
            mapView = (MapView) findViewById(R.id.mapview);
            mapView.setBuiltInZoomControls(true);
            mapView.setStreetView(true);
            mapController = mapView.getController();
            mapController.setZoom(16); // Zoon 1 is world view
            locationManager = (LocationManager)
            getSystemService(Context.LOCATION_SERVICE);
```

```
        boolean isGPS =
        locationManager.isProviderEnabled
        (LocationManager.GPS_PROVIDER);

        locationManager.requestLocationUpdates
        (LocationManager.GPS_PROVIDER, 5000,
                        50, new GeoUpdateHandler());

        if (isGPS==false){
        locationManager.requestLocationUpdates
        (LocationManager.NETWORK_PROVIDER, 5000,
                50, new GeoUpdateHandler());
        }

        //System.out.println(result1);

        //--- overlay ---
List<Overlay> mapOverlays = mapView.getOverlays();
Drawable drawable = this.getResources().getDrawable
(R.drawable.androidmarkerred);
itemizedoverlay = new HelloItemizedOverlay(drawable);
Drawable drawable1 =
 this.getResources().getDrawable(R.drawable.androidmarker);
itemizedoverlay2 = new HelloItemizedOverlay(drawable1);
point = new GeoPoint(24792345,120999813);
point2 = new GeoPoint(24792345,120999813);
overlayitem = new OverlayItem(point, "Ruwan!", "I'm in");
```

```
        overlayitem2 = new OverlayItem(point2, "Ruwan!", "I'm in");
        mapController.animateTo(point);

        itemizedoverlay.addOverlay(overlayitem);
        itemizedoverlay2.addOverlay(overlayitem2);
    mapOverlays.add(itemizedoverlay);
    mapOverlays.add(itemizedoverlay2);
      }

        public void onLocationChangedRefresh(Location location) {


        lat = (int) (location.getLatitude() * 1E6);
        lng = (int) (location.getLongitude() * 1E6);

                //----- signature -----


        locationForSign =
        Integer.toString((int)Math.round(lat/(1E6)))+","+
        Integer.toString((int)Math.round(lng/(1E6)));


try {
signature = digiSign.signLocation(locationForSign);
        } catch (Exception e) {

                e.printStackTrace();
```

```
        }
try {
result1 = digiSign.signVerify(signature, locationForSign );
} catch (Exception e) {
                e.printStackTrace();
        }
StringBuffer result = new StringBuffer();
for (int i=0; i < 128; i++) {
result.append(Integer.toString(signature[i])+",");
        }

String signatureUser1 = result.toString();

                //--- encrypt -----------
latCipher = homomorphicEnc.encryptH(lat,keyBig);
lngCipher = homomorphicEnc.encryptH(lng,keyBig);

                 //---send it to the server ----

                String[] latCipherUserX = new String[66];
                latCipherUserX[0] = "1";
                // add "1" to indicate user1
                for(int i = 1;i<33;i++){
                        latCipherUserX[i] = latCipher[i-1];
                        //latCipherUserX[33+i] = lngCipher[i-1];
                }
                latCipherUserX[33] = signatureUser1;
                // add "1" to indicate user1
```

```
                for(int j = 1;j<33;j++){
                        //latCipherUserX[i] = latCipher[i-1];
                        latCipherUserX[33+j] = lngCipher[j-1];
                }

                latCipherU = mySecretaryClasses.
                socketConnect(latCipherUserX);



//---split to lng and lat--------

                for (int k = 0;k<32;k++){
                        latCipherS[k] = latCipherU[k+2];
                }

                for (int k = 0;k<32;k++){
                        lngCipherS[k] = latCipherU[k+35];
                }

                // --- decrypt -------------
                String latDeCipher = homomorphicDec.
                decryptH(lat,latCipherS,keyBig);
                String lngDeCipher = homomorphicDec.
                decryptH(lng,lngCipherS,keyBig);

                //----toast--------
                Context context1 = getApplicationContext();
                int duration1 = Toast.LENGTH_SHORT;
```

```
            Toast toast1 = Toast.makeText
            (context1,"Latitude : "+
            (Float.valueOf(latDeCipher)/1000000)+"\nLongitude :
             "+(Float.valueOf(lngDeCipher)/1000000)+
            "Sign : "+locationForSign,duration1);
            //arrayToString2(latCipher)
            //latDeCipher+","+lngDeCipher
            toast1.show();

            //----- move the man ------
            point = new GeoPoint(Integer.parseInt(latDeCipher),
            Integer.parseInt(lngDeCipher));
            point2 = new GeoPoint(lat,lng);

            mapController.animateTo(point);
            // mapController.setCenter(point);

                     overlayitem = new OverlayItem
                     (point, "Ruwan!", "I'm in ???!");
                 overlayitem2 = new OverlayItem
                 (point2, "Ruwan!", "I'm in ???!");

                 itemizedoverlay.addOverlay(overlayitem);
                 itemizedoverlay2.addOverlay(overlayitem2);

    }

    //-------------
```

1896

```
        @Override
        protected boolean isRouteDisplayed() {
                return false;
        }

        public class GeoUpdateHandler implements LocationListener {

                @Override
                public void onLocationChanged(Location location) {
                        locationLast=location;

                lat = (int) (location.getLatitude() * 1E6);
                lng = (int) (location.getLongitude() * 1E6);

                        //----- signature -----

                        locationForSign =  Integer.toString
                        ((int)Math.round(lat/(1E6)))+ Integer.
                        toString((int)Math.round(lng/(1E6)));

try {
signature = digiSign.signLocation
(locationForSign);
} catch (Exception e) {
                e.printStackTrace();
}
try {
```

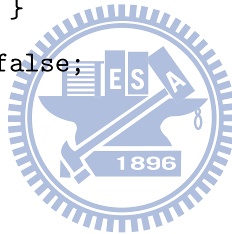
```
result1 = digiSign.signVerify(signature, locationForSign );
} catch (Exception e) {
        e.printStackTrace();
}
        StringBuffer result = new StringBuffer();
    for (int i=0; i < 128; i++) {
                result.append(Integer.toString(signature[i])+",");
        }
                        String signatureUser1 = result.toString();
        //--- encrypt -----------
        latCipher = homomorphicEnc.encryptH(lat,keyBig);
        lngCipher = homomorphicEnc.encryptH(lng,keyBig);

        //---send it to the server ----

        String[] latCipherUserX = new String[66];
        latCipherUserX[0] = "1"; // add "1" to indicate user1
        for(int i = 1;i<33;i++){
                latCipherUserX[i] = latCipher[i-1];
        }
                latCipherUserX[33] = signatureUser1;
                 // add "1" to indicate user1
                for(int j = 1;j<33;j++){
                        //latCipherUserX[i] = latCipher[i-1];
                        latCipherUserX[33+j] = lngCipher[j-1];
                }

                latCipherU =
```

```
    mySecretaryClasses.socketConnect(latCipherUserX);

    //---split to lng and lat

    for (int k = 0;k<32;k++){
            latCipherS[k] = latCipherU[k+2];
    }

    for (int k = 0;k<32;k++){
            lngCipherS[k] = latCipherU[k+35];
    }

    // start tracing to "/sdcard/calc.trace"
//Debug.startMethodTracing("decrypotion1");
    // --- decrypt -------------
    String latDeCipher = homomorphicDec.
    decryptH(lat,latCipherS,keyBig);
    String lngDeCipher = homomorphicDec.
    decryptH(lng,lngCipherS,keyBig);

     // stop tracing
     //  Debug.stopMethodTracing();
    //----toast--------
    Context context1 = getApplicationContext();
    int duration1 = Toast.LENGTH_SHORT;
    Toast toast1 = Toast.makeText(context1,
    "Latitude : "+(Float.valueOf(latDeCipher)/1000000)
    +"\nLongitude : "+(Float.valueOf
```

```
(lngDeCipher)/1000000),duration1);
//arrayToString2(latCipher)
//latDeCipher+","+lngDeCipher
toast1.show();

        //----- move the man ------
point = new GeoPoint(Integer.
parseInt(latDeCipher),
Integer.parseInt(lngDeCipher));
point2 = new GeoPoint(lat,lng);

mapController.animateTo(point);
// mapController.setCenter(point);

         overlayitem = new OverlayItem
         (point, "Ruwan!", "I'm in ???!");
     overlayitem2 = new OverlayItem
     (point2, "Ruwan!", "I'm in ???!");

     itemizedoverlay.addOverlay(overlayitem);
     itemizedoverlay2.addOverlay(overlayitem2);

        //------------------------

}
@Override
public void onProviderDisabled(String provider) {
}
```

```
        @Override
        public void onProviderEnabled(String provider) {
        }

        @Override
        public void onStatusChanged
        (String provider, int status, Bundle extras) {
        }
}

public static String arrayToString2(String[] a) {
    StringBuffer result = new StringBuffer();
    if (a.length > 0) {
        for (int i=1; i<a.length; i++) {
            result.append(a[i]);
        }
    }
    return result.toString();
}

/* Creates the menu items */
public boolean onCreateOptionsMenu(Menu menu) {
        menu.add(0, MENU_REFRESH, 0, "Refresh");
        menu.add(0, MENU_QUIT, 0, "QUIT");
    return true;
}
/* Handles item selections */
```

```
        public boolean onOptionsItemSelected(MenuItem item) {
            switch (item.getItemId()) {

            case MENU_REFRESH:

                    onLocationChangedRefresh(locationLast);

                return true;

            case MENU_QUIT:
                    System.exit(0);
                return true;
                    }
            return false;
        }
}
```



F.2 Itemized Overlay Class

```
package xor.homomorphic.encrypted.secure.location.sharing;

import java.util.ArrayList;

import android.app.AlertDialog;
import android.content.Context;
import android.graphics.drawable.Drawable;
```

```
import com.google.android.maps.ItemizedOverlay;
import com.google.android.maps.OverlayItem;

public class HelloItemizedOverlay extends ItemizedOverlay {

        private ArrayList<OverlayItem> mOverlays
        = new ArrayList<OverlayItem>();
        private Context mContext;

        public HelloItemizedOverlay(Drawable defaultMarker) {
                  super(boundCenterBottom(defaultMarker));
                }

        public void addOverlay(OverlayItem overlay) {

        for (int i = 0; i < mOverlays.size(); i++) {
                        mOverlays.remove(i);
            }

            mOverlays.add(overlay);
            populate();
        }

        public void cleanAllMarker(){

                for (int i = 0; i < mOverlays.size(); i++) {
                        mOverlays.remove(i);
```

```
        }
    }

    @Override
    protected OverlayItem createItem(int i) {
      return mOverlays.get(i);
    }

    @Override
    public int size() {
      return mOverlays.size();
    }

    public HelloItemizedOverlay
    (Drawable defaultMarker, Context context) {
              super(defaultMarker);
              mContext = context;
    }
}
```

F.3 Homomorphic Encryption Class

```
package xor.homomorphic.encrypted.secure.location.sharing;
import java.math.BigInteger;
import java.util.Random;
```

```
import android.os.Debug;

public class homomorphicEnc {



private static String toBin;
private static BigInteger qLarge;
private static BigInteger rLarge;
private static BigInteger pLarge;

public static String[] encryptH(int location,BigInteger key) {

        toBin = Integer.toBinaryString(location);
        pLarge = key;
        String[] cipherLarge = new String[32];
    Random generator = new Random(19580421);
        for (int i=0; i<toBin.length(); i++) {

                encDigitInt[32-toBin.length()+i] =
                Character.getNumericValue(toBin.charAt(i));

                          }

String[] outBig = null;

                for (int k=0;k<32;k++){
                        Random rand = new Random();
                        qLarge = new BigInteger(4096,rand);
```

```
                        rLarge = new BigInteger(8,rand);

                        BigInteger c1 = BigInteger.valueOf
                        (encDigitInt[k]);
                        BigInteger c2 = pLarge.multiply(qLarge);
                        BigInteger c3 = rLarge.multiply
                        (BigInteger.valueOf(2));
                        BigInteger c4 = c1.add(c2).add(c3);

                        outputStringBig[k] = c4.toString();

                        }

                        return outputStringBig;

                }

        }
```

F.4 JAVA Socket Class

```
-
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.Socket;
public class mySecretaryClasses {
        static String[] msgFromServer ;
```

```
        public static String[] socketConnect(String[] msgToServer) {
                int arraySizeRx = 0;
                try {
              Socket clientSocket = new Socket("yourdomain.com", 3105);
              ObjectOutputStream outToServer = new
              ObjectOutputStream(clientSocket.getOutputStream());
              ObjectInputStream inFromServer = new
              ObjectInputStream(clientSocket.getInputStream());
              int arraySizeTx = msgToServer.length;
              String arraySizeTxStr = Integer.toString(arraySizeTx);
              outToServer.writeObject(arraySizeTxStr);
              outToServer.flush();
              int k=0;

          do{
          outToServer.writeObject(msgToServer[k]);
          outToServer.flush();
          k++;
          }while(k < msgToServer.length);
          outToServer.writeObject("bye");
          outToServer.flush();
// the first value the server send is the size of the transmission,
// need  to get that value to run the loop
        arraySizeRx = Integer.parseInt((String)
        inFromServer.readObject());
        msgFromServer = new String[arraySizeRx+1];
        msgFromServer[0] = Integer.toString(arraySizeRx);
        // receive info from server
```

```
            for(int i=1; i<arraySizeRx+1; i++){
            msgFromServer[i] = (String)inFromServer.readObject();
                                 }
            // close the socket after transmission
            try{
                 inFromServer.close();
                     outToServer.close();
                     clientSocket.close();
                    }
                                   catch(IOException ioException){
                                           ioException.printStackTrace();
                                   }
                    }catch (Exception e) {

            }
                    return msgFromServer;
            }
    }
```

F.5 Homomorphic Decryption Class

```
    package xor.homomorphic.encrypted.secure.location.sharing;


    import java.math.BigInteger;



    public class homomorphicDec {
```

```
    private static int locationDec[] = new int[32];
    private static int locationDecXOR[] = new int[32];
    private static String[] latStringArray = new String[32];
    private static String latString;
    private static String toBinLatLng;
    private static BigInteger locationLarge;
    private static BigInteger keyLarge;

    public static String decryptH
    (int latlng,String[] location,BigInteger key) {

            keyLarge = key;
            for (int i = 0;i<32;i++)        {
            locationLarge = new BigInteger(location[i]);
            BigInteger mm = locationLarge.mod(keyLarge)
            .mod(BigInteger.valueOf(2));
            locationDec[i] = mm.intValue();

            }

            for (int j=0;j<32;j++){latStringArray[j]
            = Integer.toString(locationDec[j]);}
        latString = arrayToString2(latStringArray);

//--- XOR with current users location

            toBinLatLng = Integer.toBinaryString(latlng);
            locationDecXOR = xorclass.twostoint
```

```
            (locationDec ,toBinLatLng);

            // --- chenge negetive values from twos
            compliment to normal form ---
            //---convert to decimal

            String latStringDec = TwosToInt.
            twostoint(locationDecXOR);

            return latStringDec;

        }

        public static String arrayToString2(String[] a) {
            StringBuffer result = new StringBuffer();
            if (a.length > 0) {
        for (int i=0; i<a.length; i++) {
        result.append(a[i]);
                }
            }
            return result.toString();
        }

}
```

F.6 Two's Complement to Integer Class

```
public class TwosToInt {

        private static int[] latIntArrayNew = new int[32];
        private static String[] latStringArray = new String[32];
        private static int latDec;
        private static String latStringString;
        /**
         * @param args
         */
        public static String twostoint(int[] latIntArray) {
        int kk = latIntArray[0];
                if(kk==1){
                //---convert----------
                for(int i=0;i<32;i++){
                //--- binary NO operation
                if(latIntArray[i]==0){latIntArrayNew[i]=1;}
                else{latIntArrayNew[i]=0;}
          }
                  }else{
          latIntArrayNew =  latIntArray;
                          }
for(int j=0;j<32;j++){latStringArray[j] =
Integer.toString(latIntArrayNew[j]);}
                latStringString = arrayToString2(latStringArray);
```

```
        //--add 1 after the NO operation
        latDec = 0;
        if(kk==1){
latDec = (( Integer.parseInt(latStringString,2))+1)*(-1);
        }else{
                latDec = ( Integer.parseInt(latStringString,2)+1);
        }
        return Integer.toString(latDec);
                //Integer.toString(latDec);
                //latStringString;
        }
        public static String arrayToString2(String[] a) {
            StringBuffer result = new StringBuffer();
if (a.length > 0) {
for (int i=0; i<a.length; i++) {
result.append(a[i]);
                }
            }
return result.toString();
        }
}
```

F.7 XOR Operation Class

```
public class xorclass {
        private static int[] output = new int[32];
        public static int[] twostoint
```

```
        (int[] latIntArray,String latlong)  {

        for (int i=0; i<latlong.length(); i++) {
        encDigitInt[32-latlong.length()+i] =
        Character.getNumericValue(latlong.charAt(i));
                  }

                for (int j=0;j<32;j++){
                output[j] = encDigitInt[j] ^ latIntArray[j];
                }
                return output;
        }
}
```

F.8 RSA Digital Signature Class

```
package xor.homomorphic.encrypted.secure.location.sharing;

import java.math.BigInteger;
import java.security.KeyFactory;
import java.security.SecureRandom;
import java.security.Security;
import java.security.Signature;
import java.security.interfaces.RSAPrivateKey;
import java.security.interfaces.RSAPublicKey;
import java.security.spec.RSAPrivateKeySpec;
import java.security.spec.RSAPublicKeySpec;
```

```
public class digiSign {

public static byte[] signLocation
(String location) throws Exception {
 Security.addProvider
 (new org.bouncycastle.jce.provider
 .BouncyCastleProvider());

  KeyFactory keyFactory = KeyFactory.
  getInstance("RSA", "BC");

  BigInteger modulusPri =
  new BigInteger
  ("8a6cbe671a7b2ac818a8a0af86ce
  edc08278c1b2c610512bd5342928aea5235b909b66a
  cdcbf20cd303be020dd2f51f4d501fa18a266c65598e
  9d969714e589d3a38d5466d752d41315d379e46028c4a
  78a83a26d4db1f164e2f957cbe19ecc2f6c120f69687b3
  d431647aa2eaae86ea03f081d166ab8e3846815cfc1efe2ea5",16);

  BigInteger exponentPri =
  new BigInteger("8f4a01ceb89ac6ae5ad8337d7f0eb50d92016e80a08643
  801c32e2683a60b7391177cbd124b0b443b2aa4857bc9e3f383146da9
  ca57fbdcedd7b3492f1b728020bbbbf4d8e032c7a503e24a38915ccc8
  d57156aae83bb25cec4b3185aa29623d99063e4721801993a27bcedaeb
  023974b5d4cc985a89e4faca74c249054dad",16);
  RSAPrivateKeySpec privKeySpec = new RSAPrivateKeySpec
 (modulusPri, exponentPri);
```

```
 RSAPrivateKey privKey = (RSAPrivateKey) keyFactory.
 generatePrivate(privKeySpec);

Signature signature = Signature.getInstance("SHA1withRSA", "BC");

signature.initSign(privKey, new SecureRandom());

signature.update(location.getBytes());

byte[] sigBytes = signature.sign();

return sigBytes;}

public static boolean signVerify(byte[] sigBytes,
String location)throws Exception {
Security.addProvider
(new org.bouncycastle.jce.provider.
BouncyCastleProvider());

KeyFactory keyFactory = KeyFactory.
getInstance("RSA", "BC");

                    BigInteger modulus =
new BigInteger("8a6cbe671a7b2ac818a8a0af86ce
edc08278c1b2c610512b
d5342928aea5235b909b66acdcbf20cd303be020dd2f
51f4d501fa18a266
```

```
c65598e9d969714e589d3a38d5466d752d41315d379
e46028c4a78a83a26d4db
1f164e2f957cbe19ecc2f6c120f69687b3d431647aa2
eaae86ea03f081d166ab8e38
46815cfc1efe2ea5",16);
BigInteger exponentPub = new
BigInteger("10001",16);
RSAPublicKeySpec pubKeySpec = new
RSAPublicKeySpec(modulus,exponentPub);

  RSAPublicKey pubKey = (RSAPublicKey)
  keyFactory.generatePublic(pubKeySpec);


 Signature signature = Signature.
 getInstance("SHA1withRSA", "BC");

 signature.initVerify(pubKey);
 signature.update(location.getBytes());
 boolean verificationResult = signature.verify(sigBytes);

 return verificationResult;}

  }
```

Appendix G

Server JAVA Code

G.1 Main Class

```
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.net.ServerSocket;
import java.net.Socket;
import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
import java.util.Vector;


public class Provider{
        ServerSocket providerSocket;
```

```
        Socket connection = null;
        ObjectOutputStream out;
        ObjectInputStream in;
        String message;
        Vector<String> tableInfo = new Vector<String>()  ;
        String[] test1 = new String[5];
        String[] test2 = new String[5];
        Vector<String> vector = new Vector<String>();
        Vector<String> lastLocUser1 =
        new Vector<String>();
        Vector<String> lastLocUser2 =
        new Vector<String>();
        Vector<String> lastLocUser11 =
        new Vector<String>();
        Vector<String> lastLocUser22 =
        new Vector<String>();

private static
Vector<String> vLat = new Vector<String>
(Arrays.asList(mocLocUser2latEnc));
private static Vector<String> vLng = new
Vector<String>(Arrays.asList(mocLocUser2lngEnc));
private static Vector<String> vLatLng1 =
new Vector<String>();
private static Vector<String> vLatLng2 =
new Vector<String>();

        private String signatureUser1="";
```

```
        private String signatureUser2="";
        private String signature="";

        Provider(){}
        @SuppressWarnings("unchecked")
        void run()
        {
                try{
providerSocket = new ServerSocket(2004, 10);
System.out.println("Waiting for connection");
connection = providerSocket.accept();
System.out.println
("Connection received from " + connection
.getInetAddress().getHostName());
out = new ObjectOutputStream
(connection.getOutputStream());
out.flush();
in = new ObjectInputStream
(connection.getInputStream());

int k = 0;
vector.clear();
                do{
try{
message = (String)in.readObject();
if(k==34){

        if (Integer.parseInt
```

```
        (vector.elementAt(1).toString())==1)
        {
        signatureUser1 = message;
        }

if (Integer.parseInt
(vector.elementAt(1).toString())==2)
{
signatureUser2 = message;
}
vector.add("5");
}else {
vector.add(message);
}
```

```
        k++;
        }
catch(ClassNotFoundException classnot){
System.err.println
("Data received in unknown format");
}
}while(!message.equals("bye"));

                }
catch(IOException ioException){
ioException.printStackTrace();
                }

if (Integer.parseInt(vector.elementAt(1).toString())==1){
```

```
lastLocUser1 = (Vector<String>) vector.clone();

System.out.println("client>" + lastLocUser1.elementAt(1));

                }else{}

if (Integer.parseInt(vector.elementAt(1).toString())==2){
lastLocUser2 = (Vector<String>) vector.clone();

System.out.println("client>" + lastLocUser2.elementAt(1));

                }else{}

                //---------- mock locations ----------------
                vLatLng1.clear();
                vLatLng2.clear();

                vLatLng2.add("2");
                vLatLng2.add("2");
                vLatLng2.addAll(vLat);
                vLatLng2.add("2");
                vLatLng2.addAll(vLng);
                vLatLng2.add("2");

                vLatLng1.add("1");
                vLatLng1.add("1");
                vLatLng1.addAll(vLat);
                vLatLng1.add("1");
```

```
vLatLng1.addAll(vLng);
vLatLng1.add("1");

//----null check-------------------------------

if(lastLocUser1.isEmpty()){lastLocUser1=vLatLng1;}

if(lastLocUser2.isEmpty()){lastLocUser2=vLatLng2;}

tableInfo = homomorphicAddition.add
(vector,lastLocUser1,lastLocUser2);

tableInfo.remove(33);
tableInfo.add(33, signatureUser1+"--"+signatureUser2);

sendMessage(Integer.toString(tableInfo.size()+2));

int i=0;
do{

sendMessage(tableInfo.elementAt(i)
.toString());
System.out.println("server>"+i+">"+
tableInfo.elementAt(i).toString());

i++;
```

```
                }while(i < tableInfo.size());

                sendMessage("bye");

try{
                in.close();
                out.close();
                providerSocket.close();
        }
        catch(IOException ioException){
                ioException.printStackTrace();
        }
        }

        void sendMessage(String msg)
        {
                try{
        out.writeObject(msg);
        out.flush();

                }
                catch(IOException ioException){
                ioException.printStackTrace();
                }
        }
        public static void main(String args[])
        {
                Provider server = new Provider();
```

```
                while(true){
                        server.run();
                }
        }

}
```

G.2 Homomorphic Addition Class

```
import java.math.BigInteger;
import java.util.Arrays;
import java.util.Vector;



public class homomorphicAddition {
private static int[] user1Lat = new int[32];
private static int[] user1Lng = new int[32];
private int[] user2Lat = new int[32];
private int[] user2Lng = new int[32];
private static BigInteger userLoc1Big;
private static BigInteger userLoc2Big;

private static Vector<String>
vLat = new Vector<String>(Arrays.asList
(mocLocUser2latEnc));
private static Vector<String>
 vLng = new Vector<String>
 (Arrays.asList(mocLocUser2lngEnc));
```

```
private static Vector<String> vLatLng1 =
new Vector<String>();
private static Vector<String> vLatLng2 =
new Vector<String>();

        @SuppressWarnings("unchecked")
        public static Vector<String> add(Vector<String>
        locationInfoIn,Vector<String> u1,Vector<String> u2)
        {

Vector<String> locationInfoOut = new Vector<String>();
Vector<String> a = new Vector<String>() ;
Vector<String> a1 = new Vector<String>();
Vector<String> b = new Vector<String>();
Vector<String> b1 = new Vector<String>();
Vector<String> calcResult = new Vector<String>();
Vector<String> outputBigArray = new Vector<String>();

vLatLng2.add("2");
vLatLng2.addAll(vLat);
vLatLng2.add("2");
vLatLng2.addAll(vLng);

vLatLng1.add("1");
vLatLng1.addAll(vLat);
vLatLng1.add("1");
vLatLng1.addAll(vLng);
```

```
a1 = (Vector<String>) u1.clone();
b1 = (Vector<String>) u2.clone();

a1.remove(0);
a1.remove(66);

b1.remove(0);
b1.remove(66);

switch (Integer.parseInt(locationInfoIn.
elementAt(1).toString()))
 // cast to an integer

              {
              case 1:
```

```
 a = (Vector<String>)locationInfoIn.clone();

a.remove(0);
a.remove(66);
a1 = a;
  break ;

              case 2:

b = (Vector<String>)locationInfoIn.clone();
b.remove(0);
b1 = b;
```

```
         break ;
}
 for(int i=0;i<66;i++){

userLoc1Big = new BigInteger(a1.elementAt(i));
 userLoc2Big = new BigInteger(b1.elementAt(i));

outputBigArray.add((userLoc2Big.add(userLoc1Big))
].toString());

              }

locationInfoOut = outputBigArray;
return locationInfoOut;

        }
}
```