Secure Multi-party based Cloud Computing Framework for Statistical Data Analysis of Encrypted Data

by

G. P. Harsha Sandaruwan
P. S. Ranaweera

Supervisor
Professor Vladimir Oleshchuk

This Master's Thesis is carried out as a part of the education at the University of Agder and is therefore approved as a part of this education. However, this does not imply that the University answers for the methods that are used or the conclusions that are drawn.

University of Agder, 2013
Faculty of Engineering and Science
Department of Information & Communication Technology

Abstract

Secure Multi-party Computation (SMC) is a paradigm used to accomplish a common computation among multiple users while keeping the data of each party secret from others. In recent years there has been a keen interest among the research community to look for techniques that can be adopted for the evolvement of SMC based solutions for improving its efficiency and performance. Cloud computing is a next generation computing solution in the field of Information and Communication Technology (ICT) which allows its users to use high speed infrastructure and services provided by Cloud Service Providers (CSP) in a cost effective manner with a higher availability. Therefore, deployment of cloud based architecture for SMCs would aid in improving its performance and efficiency. However, cloud based solutions raises concerns over security of users' private data, since data is handled by an external party that cannot be trusted. Hence, it is necessary to incorporate necessary security measures to ensure the security of users' private data.

In this master's thesis we have addressed this issue by proposing a *Secure Multi-party based Cloud Computing Framework* which can ensure security, privacy and anonymity of users private data. In order to achieve this, we have formulated a case involving sales data analysis of a certain organization through computing statistical parameters of sales persons private sales data on a cloud environment. Furthermore, we have implemented a prototype of the proposed security framework which aids us to evaluate its performance. Moreover, considering the results that we have obtained, it is conclusive that cloud platforms can be successfully deployed to improve efficiency of SMCs while ensuring the security of users' private data; which in turn provides evidence for the practicability of multi-party based cloud computing solutions.

Keywords: *secure multi-party computation, cloud computing, homomorphic encryption, data security, privacy, anonymity*

Preface

This report serves as a Master's Thesis in Information Security to fulfill the requirements of the Master's Program in Information and Communication Technology (ICT) at the Faculty of Engineering and Science, University of Agder (UiA) in Grimstad, Norway. The research was started on 7^{th} January 2013 and ended on 3^{rd} June 2013. The main objective of this thesis is to propose and implement a Secure Multi-party based Cloud Computing Framework which can cater the security requirements of the associated users.

We are delighted to take this opportunity to show our sincere gratitude to all the ones who have helped us in achieving our goals. We are greatly obliged to our supervisor, Professor Vladimir A. Oleshchuk for his valuable assistance in giving feedback on project work throughout the semester. We are also grateful to Indika Anuradha Mendis who is a Doctoral Fellow in the department of ICT for his valuable thoughts especially on the report writing. It is our duty to also thank Sigurd K. Brinch who is working as a Chief Engineer at the University of Agder for allowing us to use university computing resources which was very important for us to carry out our experiments. Last, but certainly not the least, we would like to express our gratefulness and humility to our parents and friends for their support during the Master's Thesis.

Grimstad
June, 2013
G.P. Harsha Sandaruwan
P.S. Ranaweera

Contents

List of Figures

List of Tables

Abbreviations

3DES	Triple Data Encryption Standard
CSP	Cloud Service Provider
FHE	Fully Homomorphic Encryption
HMAC	Hashed Message Authentication Code
IaaS	Infrastructure as a Service
ICT	Information and Communication Technology
IDS	Intrusion Detection System
IHC	Iterated Hill Cipher
IT	Information Technology
MAC	Message Authentication Code
MA	Mutual Authentication
MD5	Message Digest Algorithm 5
MRS	Modified Rivest Scheme
P-box	Permutation Box
PaaS	Platform as a Service
PC	Personal Computer
RO	Research Objectives
RQ	Research Questions
S-box	Substitution Box

SaaS	Software as a Service
SDS	Secure Data Sharing
SHA1	Secure Hash Algorithm 1
SMCC	Secure Multi-party Cloud Computing
SMC	Secure Multi-party Computation
TRN	Token Ring Networks
TS	Timestamps
TTP	Trusted Third Party
VPP	Virtual Party Protocols
WSN	Wireless Sensor Networks

Chapter 1

Introduction

In this chapter, we attempt to provide background information regarding cloud computing and Secure Multi-party Computations (SMC) while emphasizing the significance of moving towards secure multi-party based cloud computing solutions. Then we have stated the problem statement along with a case which we are going to address during our thesis followed by relevant research work carried out in the area of interest. Finally, an outline describing the structure of the upcoming chapters is provided.

1.1 Cloud Computing

Cloud computing is vast developing and a most discussed topic among the individuals and business organizations who utilize and research over the newest trends in Information Technology (IT). Some of the leading IT companies in the world such as IBM, Google, Yahoo, Amazon have already developed large scale cloud systems for providing various types IT services through the cloud. The term cloud is analogical to the Internet. Hence cloud computing can be visualized as computing over Internet. More precisely it is a set of resources and facilities offered via the Internet [1].

Cloud architecture consist of a large number of shared servers distributed all over the world providing software, infrastructure, platform, devices and other required resources and hosting to subscribers on a "pay as you use it basis". The services provided over a cloud are categorized into three service models depending on the type of resources allocated by the cloud service provider for the customers. They are Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS).

A cloud makes it possible for a user to access his information in the cloud from anywhere, anytime through the Internet. On the other hand users do not need to worry about the maintenance and availability of resources purely because it is the responsibility of the cloud service provider. More importantly cloud computing is an on demand service, where users are charged only based on their resource consumption. Because of these aspects, cloud computing has become more and more popular among public as well as organizations.

1.2 Secure Multi-party Computation

The concept of Secure Multi-party Computation (SMC) was first introduced by Yao in 1982, in which he explained this through a well-known problem called "millionaire problem" [2].

"Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?"

In general terms, a SMC can be defined as a situation where n parties who are having private inputs x_1, x_2 ..., x_n interested in computing the value of the public function $f(x_1, x_2, ..., x_n)$ in such a way that at the end of the computation no party is revealed any of the private inputs of other parties [3]. This process can be theoretically represented with the existence of a Trusted Third Party (TTP), where each user party sends its private data to the TTP to compute the value of the function of common interest as shown in the left side of Figure 1.1.

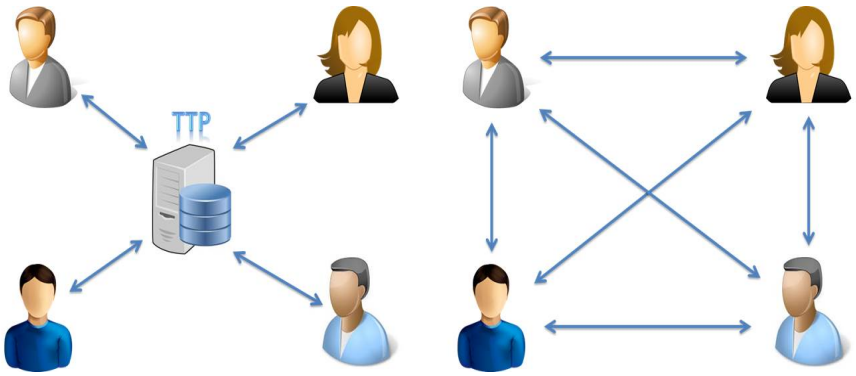

Figure 1.1: Ideal and Practical SMC Scenarios

The right side of Figure 1.1 illustrates the way in which a practical SMC works without the existence of a TTP, which is equivalent in functionality to the ideal representation with a TTP. Therefore, SMC is all about finding appropriate cryptographic protocols that can replace the use of a TTP, to carry out a certain user intended function while ensuring data privacy of users [4]. Furthermore, SMCs are used in a variety of other applications other than the "millionaire problem" as listed below.

- Electronic voting and elections
- Privacy preserving data mining
- Auctions and bidding systems
- Business related computations

1.3 Motivation for Secure Multi-party based Cloud Computing

Even though SMC algorithms capable of achieving privacy of user data even at the computation time, most of the existing multi-party computation approaches incur a lot of communication overhead [5]. Thus, a greater interest is evolved among the research personals in recent times regarding the possibility of outsourcing the computations to a Cloud Service Provider (CSP) which in turn will reduce the expenditure and operation overhead while improving the efficiency. However, outsourcing data to clouds for computational purposes may lead to privacy issues due to the fact that data is handled by an un-trusted external entity. So, it is clear that users simply cannot send private data as raw data to the cloud. This makes way for an interested topic of "how to do the computations on encrypted data". If this is achieved then it is possible for users to send the encrypted data to the cloud and carry out the required computation on encrypted data while decrypting computational result at the user end revealing the raw answer of the intended computation. Furthermore, this type of a framework capable of reducing the computational overhead on the user side, since the bulk of the work is carried out at the cloud. On the other hand maintaining and managing computing resources are under the control of the CSP which is another plus for the end users. So, it is a challenging task to come up with such a protocol which is capable of performing computations on encrypted data while ensuring security requirements.

1.4 Problem Statement

In the previous section we discussed the importance of using cloud based solutions for SMCs, considering the reduction of communication and computational overhead which ultimately leads to an improved efficiency. Even though such benefits exist, it is necessary to overcome the security concerns associated with cloud systems simply because users cannot send the unencrypted private data to the cloud to carry out the computations. Once data is encrypted, in normal circumstances it is not possible to perform the user intended computations.

Therefore, it is necessary to encrypt the data in such a way that computations can be carried out on the encrypted data. Moreover, it is important to provide identity anonymization at the cloud, since it is not necessary to provide any additional information to the cloud except what is required for the computation.

1.4.1 Research Objectives (RO)

In order to address this issue we are going to consider a case where an organization requires computing and analyzing statistical parameters about their sales. In this case, organization is required to compute statistical parameters such as mean, variance, standard deviation, skewness and kurtosis of sales values which were gathered from sales personnel. Furthermore, an external entity is responsible for analyzing computed statistical results. Moreover, data privacy and identity anonymization of sales persons are vital aspects. In the process of providing a solution for this case, following objectives must be accomplished.

RO 1 Proposing an appropriate secure multi-party based cloud computing framework which satisfies the requirements of the above mentioned case.

RO 2 Implement a prototype of the framework.

RO 3 Evaluate the performance of the framework through the implementation.

1.4.2 Research Questions (RQ)

In order to achieve our research goals, it is necessary to find solutions to following research questions.

RQ 1 Which data encryption method allows computation of required statistical parameters on encrypted data?

RQ 2 How to hide the identity of the user parties from the cloud?

RQ 3 How to develop the framework in such a way that it can withstand the security threats such as replay attacks and integrity violations ?

RQ 4 How to keep a satisfactory level of efficiency in the proposed framework while ensuring the security requirements?

RQ 5 What parameters we can use to measure the performance of our implemented framework?

1.5 Related Work

This section describes the researches been conducted in the area of our interest. We have categorized the related researches into cloud security, computations on encrypted data, secure multi-party statistical computations and secure cloud based computations.

1.5.1 Security Issues Related to Cloud Computing

The growth and wide spread of Information and Communication Technologies (ICT) promoted the development of cloud computing which is based on the concept of distributed computing. According to La'Quata "The rise in the scope of cloud computing is continuously increasing" [6]. Though cloud techniques seem quite lucrative for the users, it was found that the cloud architecture and its' communication protocols do not guarantee the level of safety that the users typically expected to have. Consumers of the cloud computing services have serious concerns about the availability of their data when required. Users also concern about the confidentiality, integrity of the data that has been uploaded in the cloud servers [7]. Oren et al. [8] mentioned that advantages of cloud computing is shadowed with data security, safety, privacy and anonymity challenges. Therefore, the adoption of cloud computing has been inhibited to a great extent.

In the quest of finding solutions to enhance security of the data resides in a cloud, researchers have followed different methods. Qingkai Ma et al. [5] introduced a new protocol for secure data protection in cloud computing, which yields better performance at normal execution time while still assures data protection at the presence of security threats. Research carried out by Jun Feng et al. [9] took alternative perspective and proposed a protocol to enhance cloud storage security interms of repudiation, fairness and rollback attacks.

Pearson came up with a solution by introducing a privacy manager for cloud computing environments based on multi-party protocols, which can reduce the risk of cloud computing user by stealing or misusing his or her private data [10]. Afterwards, he dug it further and revealed that the most important obstacle to wide acceptance of cloud computing is services security and privacy issues [11]. In this paper he has discussed some real and practical scenarios, where the use of sensitive information must be minimized when data is processed on clouds in order to assure the privacy of end users.

According to D. K. Mishra [12], it is quite important to keep the user's identity ambiguous, since no party would like to expose their confidential data that can be exploited by its competitors. In order to fulfil this requirement; he has used a secure external entity or a TTP. In addition, anonymity could be essential for the safety, security and personal integrity of the users' data. Richard Mortier et al. [13] described a method of using Dust Clouds to enhance anonymous communication in cloud computing, which can effectively facilitate anonymity communications over un-trusted networks such as the Internet.

1.5.2 Computations on Encrypted Data

Issues with data confidentiality of cloud users have tempted the requirement for encrypting the user data before sending it to cloud. As a solution homomorphic encryption schemes have been introduced. Homomorphic encryption schemes could be either public key or symmetric key. According to M. Tebaa et al. [14], operations like addition and multiplication could be performed on encrypted data by Paillier and RSA public key encryption systems. Furthermore, A. Chan [15] suggested two additive symmetric key homomorphic schemes called as Iterated Hill Cipher (IHC) and Modified Rivest Scheme (MRS). Once such a scheme is supporting addition, it should support scalar multiplication as well [15].

Deploying homomorphic encryption schemes in cloud systems would aid to enhance security in computational as well as data storage applications. The research carried out by B.K. Samanthula et al. [16] have proposed a Secure Data Sharing (SDS) framework for such a data outsourcing application. SDS framework includes three main parties. They are data owners, data consumers and cloud server. Data owners are encrypting the data and outsourcing them in the cloud whereas consumers have to access data through the cloud. Cloud is responsible for the authorization process. Cryptosystem adopted in this paper is Paillier cryptosystem.

When we consider practical implementations of encrypted data based computations, Wireless Sensor Network (WSN) based applications are a prime example. Since, sensors are tiny devices with limited energy resources; it is infeasible to carry out the computations on sensor data within the sensor network itself [17]. In the process of addressing this issue, C. Castelluccia et al. [17] have presented a method by aggregating sensor data and forwarding them to an entity which has higher computational power to find out statistical measures such as mean and variance of sensor measurements. Their framework consists with a set of sensors, intermediate aggregator nodes and a computational entity (sink node) arranged in the structure of a balanced tree. The use of aggregator nodes meant that sensor data must be encrypted otherwise it would reveal raw data to aggregators which are deemed to be un-trusted. So, they have introduced an additive homomorphic scheme based on a symmetric key cryptosystem to encrypt the data at the sensor nodes. Then aggregator nodes simply add up the encrypted sensor values and forward to the sink node where the values decrypted and computations is carried out to find statistical parameters. A similar approach was also presented by J.Wei et al. [18] to address this issue, while the only difference is being using a public key based scheme for achieving data privacy.

1.5.3 Secure Multi-party Statistical Computations

In cooperative environments, multi-party computation has a wide range of applications. Over the years researchers have emphasized the need for securing such computational procedures to deploy them under practical circumstances. But such methods to secure multi-party computations are application specific. The possibility of universally adoptable approach has become unviable. W. Luo et al. [19] suggested a secure k-party real product protocol to perform statistical analysis by extending the secure two party real product protocols.

Various cryptographic primitives such as 1-out-of-k oblivious transfer protocol, secure two party vector scalar protocol and RSA public key streams have been used to develop this method.

Virtual Party Protocols (VPP) has been adopted to develop and enhance the security of multi-party computation schemes by researchers. Such a scenario has been followed by D.K. Mishra et al. [20] to implement a SMC protocol using Token Ring Networks (TRN). According to the method that has been suggested in [20], computation is performed in a token ring in which all the parties involved along with the TTP who is responsible for conducting computation are placed in the ring. Original data is first encrypted and conveyed to TTP through the virtual party layer. The computations are performed on encrypted data. This paper has focused on calculating statistical parameters such as mean, variance, standard deviation, skewness and kurtosis. The proposed protocol is proven secured, due to its lower hacking probability.

1.5.4 Secure Cloud based Computations

Cloud based computations often raises concerns over the confidentiality of users' private data. So, as a solution for overcoming such security issues A. Bouti et al. [21] presented a protocol to delegate computations to clouds with encrypted data. In this protocol, they have used Paillier and RSA cryptosystems to provide additive and multiplicative homomorphism; which allows carrying out a combination of additive and multiplicative operations. Their framework contains a user party and another party with high computational power (cloud server). According to the protocol, user party first encrypts the data using the corresponding encryption scheme depending on the operation to be performed on data and sends the encrypted values to the delegated party where the computation is carried out. After the computation, result is forwarded to the user end which is decrypted to find the answer of the computation. Since this protocol uses two independent homomorphic algorithms to provide additive and multiplicative homomorphism, it is only possible to carry out a single operation (additive or multiplicative) at a time.

When we consider the development of multi-party computations on cloud environments, N. Maheshwari et al. [3] carried out a research where they have proposed a Secure Multi-party Cloud Computing (SMCC) architecture which allows multiple cloud users to jointly compute a function of common interest. The architecture that they have proposed contains three layers as data encryption, data anonymization and computational.

According to their framework, each user encrypts its private data using a specific homomorphic cryptographic scheme and forwards the encrypted data stream to a randomly selected anonymizer on the anonymization layer. Anonymizers disguise the identities of users who have sent the data and forward the data to the private cloud where the computation takes place. At the reception of data at the cloud, one of the available servers is allocated for the computation and after the completion of the computation; the result is broadcasted to all the users. Finally, each user decrypts the result to determine the result of the computation. On the positive side this framework capable of providing data privacy and identity anonymization to the users while its main limitation is that they have not validated their work considering specific practical scenarios. Furthermore, they also have not considered about possible security threats such as replay attacks and integrity violations during information exchange between users and the cloud.

1.6 Thesis Outline

This report is organized into six chapters. The first chapter provides a brief introduction on cloud computing and its associated security concerns along with SMC while illustrating the motivation behind the concept of Multi-party based Cloud Computing. Then we have defined the problem statement, research objectives and research questions that we must answer during the progression of our thesis, followed by a literature study on the area of our interest.

In Chapter 2, the background knowledge required to understand the solution is discussed in detail. We have explained all the cryptographic theories which we have used when developing the proposed solution while concluding the chapter by illustrating the Client-Server architecture which provides the basis for our implementation.

The Chapter 3 is solely dedicated to illustrate the proposed Secure Multi-party based Cloud Computing Framework. We start the Chapter 3 by explaining the case which we have used to build around our solution and then the complete solution is elucidated by dividing it to several functional components using both theoretical aspects and code fragments of the implemented framework.

In Chapter 4, our focus is to analyze the performance of the implemented framework. We have measured the performance of the framework by using a set of performance parameters through a variety of experiments which would allow us to come to conclusions on the efficiency of the implementation.

The Chapter 5 of our thesis is dedicated to discuss some important aspects of our proposed framework as well as the implementation. In the beginning we have illustrated how the proposed framework is capable of living up to the security requirements expected by the users. Then a brief analysis is given on the performance characteristics of our implementation considering the insights gathered from the experiments that we have carried out using the implementation. Finally, we have discussed the associated limitations in our proposed solution.

In Chapter 6, we have summarized the main findings of our thesis and pointed out our contribution to the scientific community. Finally, important extensions to the thesis are presented as future work.

Appendix A: The paper which is prepared based on this thesis work and which is planned to be submitted to 14^{th} IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014).

Appendix B: The complete JAVA programs related to implemented prototype of the framework.

Chapter 2

Theoretical Background

In this chapter our main focus is to introduce the theoretical aspects which provide the necessary foundation to understand the Secure Multi-party based Cloud Computing solution which we are going to put forward in the following chapter. In the first part of this chapter, we closely look at the cryptographic primitives which we have considered when developing our security framework. Finally, we conclude the chapter by introducing the concepts related to client-server architecture which provides the base to our implementation of the proposed security framework.

2.1 Cryptographic Primitives used in the Proposed Solution

In the process of developing our solution, we have incorporated several cryptographic algorithms such as homomorphic cryptosystems, RSA PublicKey encryption scheme, Triple Data Encryption Standard (3DES) Symmetric Key cryptosystem, Message Authentication Codes (MAC) and Timestamps (TS). This section provides a general overview of those cryptographic fundamentals.

2.1.1 Homomorphic Cryptosystems

It is a well known fact that the main obstacle for the wide acceptance of cloud computing based solutions is the associated security risk of handing over the unencrypted private data of users to public CSPs.

Although, it is possible to send the data in an encrypted form to cloud providers' data centers; the cloud servers were not capable of processing the encrypted data prior to the emergence of '*homomorphic encryption*'. In order to allow the processing of encrypted data, the key is to encrypt the data in such a way that performing a mathematical operation on the encrypted information and then decrypting the result produces the same answer as performing an analogous operation on the unencrypted data. This corresponding relationship between the operations performed on unencrypted data and operations performed on encrypted data is known as a homomorphic scheme. The following general expression shows an encryption scheme which is homomorphic with respect to an operation '$\diamond$' for some operation '$*$' on encrypted data.

$$Decrypt(Encrypt(m_1) * Encrypt(m_2)) = Decrypt(Encrypt(m_1 \diamond m_2)) = m_1 \diamond m_2; \tag{2.1}$$

where, m_1 and m_2 represent unencrypted data.

There are two main operations that could be performed on ciphertexts which are encrypted using a homomorphic scheme. They are additive and multiplicative operations. We call a homomorphic cryptosystem as partially homomorphic, if it is capable of performing either one of these operations. For instance, RSA public key cryptosystem exhibits multiplicative homomorphism whereas Paillier cryptosystem exhibits additive homomorphism. Moreover, a homomorphic system is deemed as fully homomorphic if it can perform an unlimited number of both types of operations. However, the development of Fully Homomorphic Encryption (FHE) schemes is not yet being completely perfected, though there have been several encryption schemes proposed which exhibits fully homomorphism under certain conditions.

In our case study, we require a homomorphic cryptosystem which is additively homomorphic as well as exhibiting homomorphism on powers of the values, in order to carry out the necessary statistical computations on encrypted data. Therefore, we have used a somewhat fully homomorphic encryption scheme, which is extended from the ElGamal Public Key Cryptosystem [22] to cater our requirements. Furthermore in the subsections to follow, we have illustrated the concepts of extended ElGamal encryption scheme while proving its relevant homomorphic properties.

2.1.2 Extended ElGamal Encryption Scheme

This encryption scheme is also a public key based and similar to the original ElGamal encryption scheme; except that it exhibits more homomorphic properties than the original scheme which only satisfies multiplicative homomorphism. Before we come to terms with encryption and decryption processes, it is necessary to find out how the public, private key pair for a particular user is generated according to this scheme.

First of all, it is necessary to select two large secure prime numbers p, q and compute $N = p.q$. Then, a generator (g) which is the root of finite field $GF(p)$ and a secret value x must be selected in such a way that $g, x < p$.

After that it is possible to compute y as;

$$y = g^x \, mod \, p \tag{2.2}$$

Then, the following public and private key pair can be achieved.

Public Key : *(y, g, p, N)*
Private Key : x

Since, the public key is used to encrypt the data it is send to all the other users while the private key is kept as a secret, which is required by the user to decrypt the data encrypted from the public key.

Let us consider that a message (M) needs to be encrypted using this encryption scheme. In order to proceed with the encryption process, first a user needs to select two random positive integers r and k. If the ciphertext of M is given by $E_g(M)$;

$$E_g(M) = (a, b) = (g^k \, mod \, p, y^k E_I(M) \, mod \, p), \tag{2.3}$$

where, $E_I(M)$ is given by;

$$E_I(M) = (M + r \times p) \, mod \, N \tag{2.4}$$

According to the above relation, it is clear that the encryption scheme generates two cipertext components a and b for a particular message.

In order to illustrate the relation for the decryption process, consider the ciphertext to be decrypted as (a, b) while the related private key is given by x. If the decrypted message is represented by $D_g(a, b)$;

$$D_g(a, b) = b \times (a^x)^{-1} \, mod \, p \tag{2.5}$$

Additive Homomorphism of Extended ElGamal Scheme

In order to illustrate how additive homomorphism can be achieved through this encryption scheme; consider two plaintexts M_1, M_2 and their encryptions be denoted by $E_g(M_1)$ and $E_g(M_2)$ respectively. Furthermore, assume that both encryptions are made with the same public key (y, g, p, N) and use the same constant k.

According to Equation 2.3, $E_g(M_1)$ and $E_g(M_1)$ can be written as;

$$E_g(M_1) = (a_1, b_1) = (g^k \, mod \, p, y^k E_I(M_1) \, mod \, p) \tag{2.6}$$

$$E_g(M_2) = (a_2, b_2) = (g^k \, mod \, p, y^k E_I(M_2) \, mod \, p) \tag{2.7}$$

So, it is clear that;

$$a_1 = a_2 = g^k \, mod \, p = a \tag{2.8}$$

Furthermore, the property of additive homomorphism for two plaintexts in extended ElGamal scheme is defined as follows.

$$M_1 + M_2 = D_g(E_g(M_1) \oplus E_g(M_2)) \tag{2.9}$$

where;

$$E_g(M_1) \oplus E_g(M_2) = (a, b_1 + b_2) \tag{2.10}$$

In order to prove this property, consider the decryption of both sides in Equation 2.10.

$$D_g(E_g(M_1) \oplus E_g(M_2)) = D_g(a, b_1 + b_2) \tag{2.11}$$

By substituting for a, b_1, b_2 from Equation 2.6 and Equation 2.7;

$$D_g(E_g(M_1) \oplus E_g(M_2)) = D_g(g^k \, mod \, p, \; y^k E_I(M_1) \, mod \, p + y^k E_I(M_2) \, mod \, p) \tag{2.12}$$

$$D_g(E_g(M_1) \oplus E_g(M_2)) = D_g(g^k \, mod \, p, \; y^k (E_I(M_1) + E_I(M_2)) \, mod \, p) \tag{2.13}$$

By considering the decryption relation in Equation 2.5;

$$\begin{aligned} D_g(E_g(M_1) \oplus E_g(M_2)) = (y^k (E_I(M_1) + E_I(M_2)) \, mod \, p) \\ \times ((g^k \, mod \, p)^x)^{-1} \, mod \, p \end{aligned} \tag{2.14}$$

Then, by substituting $y = g^x \ mod \, p$;

$$D_g(E_g(M_1) \oplus E_g(M_2)) = ((g^x)^k (E_I(M_1) + E_I(M_2)) \ mod \, p) \times ((g^k \ mod \, p)^x)^{-1} \ mod \, p \tag{2.15}$$

$$D_g(E_g(M_1) \oplus E_g(M_2)) = (E_I(M_1) + E_I(M_2)) \ mod \, p \tag{2.16}$$

After that by substituting for $E_I(M_1)$ and $E_I(M_2)$ using the relation given in Equation 2.4;

$$D_g(E_g(M_1) \oplus E_g(M_2)) = ((M_1 + r_1 \times p \ ) mod \, N + (M_2 + r_2 \times p \ ) mod \, N) \ mod \, p \tag{2.17}$$

Then, it is possible to deduce that;

$$D_g(E_g(M_1) \oplus E_g(M_2)) = M_1 + M_2 \tag{2.18}$$

Hence, Equation 2.18 proves that extended ElGamal encryption scheme exhibits additive homomorphism.

Homomorphism on Powers of Values

In this section, our focus is to prove that the extended ElGamal encryption scheme possesses homomorphism on powers of values. In order to prove this property, consider a plaintext message denoted by M.

Then, the encryption of M is given by;

$$E_g(M) = (g^k \ mod \, p, \ y^k E_I(M) \ mod \, p) \tag{2.19}$$

Let us define j^{th} power of $E_g(M)$ as;

$$(E_g(M))^j = ((g^k \ mod \, p)^j, \ (y^k E_I(M) \ mod \, p)^j) \tag{2.20}$$

Then by using the Equation 2.5, we can write $D_g[(E_g(M))^j]$ as;

$$D_g((E_g(M))^j) = (y^k E_I(M) \ mod \, p)^j \times (((g^k \ mod \, p)^j)^x)^{-1} \ mod \, p \tag{2.21}$$

After substituting $y = g^x \ mod \, p$, we can get;

$$D_g((E_g(M))^j) = ((g^x)^k E_I(M) \ mod \, p)^j \times (((g^k \ mod \, p)^j)^x)^{-1} \ mod \, p \tag{2.22}$$

$$D_g((E_g(M))^j) = (E_I(M))^j \; mod \, p \tag{2.23}$$

Since, the function $E_I()$ is multiplicatively homomorphic, we can write;

$$(E_I(M))^j = E_I(M^j) \tag{2.24}$$

Then from Equation 2.23, we can get;

$$D_g((E_g(M))^j) = E_I(M^j) \; mod \, p \tag{2.25}$$

By substituting for $E_I(M^j)$ using the relation given in Equation 2.4;

$$D_g((E_g(M))^j) = (M^j + r \times p) \; mod \, N \; mod \, p \tag{2.26}$$

Then, it is possible to deduce that;

$$D_g((E_g(M))^j) = M^j \tag{2.27}$$

So, the Equation 2.28 depicts that the extended ElGamal encryption scheme is homomorphic for powers of values.

We have used the homomorphic properties that we have proved above, to compute the required statistical parameters of user inserted data by processing the corresponding encrypted data in our solution.

2.1.3 RSA Public Key Cryptosystem

Ron Rivest, Adi Shamir and Leonard Adleman created a public key encryption scheme called RSA in 1977. The RSA algorithm involves three steps, which are key generation, encryption and decryption.

The key generation phase, deals with producing two keys namely a public key and a private key. The public key is advertised to everyone where as the private key is kept as a secret. In order to generate a public, private key pair for a user, it is required to select two different random prime numbers p and q and compute $n = p.q$. Then a smaller odd integer a is selected which is relatively prime with $\phi(n)$.

$$gcd\,(a,\ \phi(n)) = 1, \tag{2.28}$$

where;

$$\phi(n) = (p-1).(q-1) \tag{2.29}$$

After that it is possible to compute b which is a multiplicative inverse to $a \, mod \, \phi(n)$.

$$b = a^{-1} \, mod \, \phi(n) \tag{2.30}$$

Then, it is possible to use (a, n) as the public key and b as the secret or private key for encryption and decryption respectively. So, we can define the encryption function $e_k(x)$ and decryption function $d_k(y)$ as given below.

$$e_k(x) = x^a \, mod \, n \tag{2.31}$$

$$d_k(y) = y^b \, mod \, n \tag{2.32}$$

where;
x - Data to be encrypted
y - Data to be decrypted

Moreover, it is also possible for the owner of a particular RSA key pair to use the private key to create a signed digest, which makes it unique since the private key is only known to the owner while the relevant public key can be used to verify the digest at the other end. If the signed digest is given by $S_k(x)$;

$$S_k(x) = x^b \, mod \, n \tag{2.33}$$

$$x = [S_k(x)]^a \, mod \, n, \tag{2.34}$$

where x is the information to be signed.

We have used both RSA encryption and signing procedures to implement mutual authentication between the entities in our proposed solution. Furthermore, it is important to note that we have used the conventional notations of RSA encryption and signing which are given below in the rest of the report.

$\{data\}_A$ - Data is encrypted using the public key of A
$[data]_A$ - Data is signed using the private key of A

2.1.4 Data Encryption Standard (DES)

DES is a symmetric key encryption scheme which was first introduced by IBM in 1970s based on a Feistel cipher called as Lucifer cipher [23]. It is a simple block cipher scheme in which plaintext is split into fixed sized blocks and fixed sized ciphertext blocks being generated in an iterative manner. In a block cipher, algorithm is mainly contributed by a function called as a '*Round function*', which is used in each of the iteration of the process. The output of the round function depends on the output of the previous iteration and a sub key which is generated from the main symmetric key.

DES uses a 56-bit symmetric key which is responsible for generating 48-bit sub keys in each round. The sub key is consisting with a subset of the main key. There are 16 rounds to DES algorithm in which it uses 64-bit block length for plaintext. In each round the plaintext block is segmented into two parts of equal bit length and processed under different steps. Right side block of plaintext is expanded, X-ORed with the key before it is processed in the Substitution box (S-box). Then, it is processed through the Permutation box (P-box) and finally X-ORed with the left side segment of the plaintext to obtain the ciphertext [23].

DES has been found to be insecure due to its relatively smaller size 56-bit key. Novel methods of key cracking could easily figure out the key and hence the plaintext of the DES encrypted data. To overcome this issue, a variant of DES has been developed with two 56-bit keys which is known as "3DES". In this method, same DES encryption and decryption algorithms are followed. In order to obtain the ciphertext, plaintext is initially encrypted with first key K_1. Then it is decrypted with second key K_2. Finally, encryption is followed again with K_1. The decryption process is the reverse process of this phenomenon. The encryption process of 3DES is mentioned below.

$$C = (E(D(E(P, K_1), K_2), K_1) \tag{2.35}$$

where;
C - 3DES Ciphertext
P - Plaintext
E( ) - DES Encryption function
D( ) - DES Decryption function

Furthermore, we have used 3DES for establishing symmetric key communication channels between the entities in our implementation.

2.1.5 Timestamp (TS)

A TS is a string value which indicates the occurrence of a certain event in time. Apparently, TS is the date and time of a specific event as recorded in the corresponding computer system. This value is considered to be accurate up to the levels of milliseconds. Furthermore, TS is represented according to the standards of *ISO8601* (e.g.: TS = 2013-04-27 21:08:35.146 which is in the format YYYY-MM-DD HH:MM:SS.MS).

TSs could be used for number of reasons in cryptography. It is possible to deploy TSs instead of a randomly generated onetime number (nonce) in authentication protocols, since a current TS ensures freshness. Furthermore, the usage of TSs improves the efficiency of such protocol, due to the fact that it is not necessary to carry out any message exchange of nonces. Other than that, usage of TSs will provide a precautionary measure for avoiding 'Replay attacks'. Replay attacks are kind of a network attack which allows a malicious agent to send a valid data transmission repeatedly or delayed to sabotage a security protocol. Although TSs provide such benefits, one problem attached is that it makes time a security critical parameter. Furthermore, we cannot expect that clocks at the sender of a message and a receiver are always synchronized meaning that a received TS must be accepted within a certain time range from the current time. This timing window is defined as a "*clock skew*". Moreover, it is important to select the clock skew intelligently, since a larger clock skew will open up a window for replay attacks whereas very small clock skew will lead for the rejection of fresh messages.

Figure 2.1 shows how we have deployed validation of TSs for determining replay attacks. In the figure, $getCurrentTimestamp()$ function returns the current TS of the system at message initiation while $CheckTimestamp()$ function computes the difference between the current TS at reception of the message and TS attached to the received message (TS_D). Then the system will check the value of TS_D with the clock skew to determine whether the received frame is a fresh one or not.

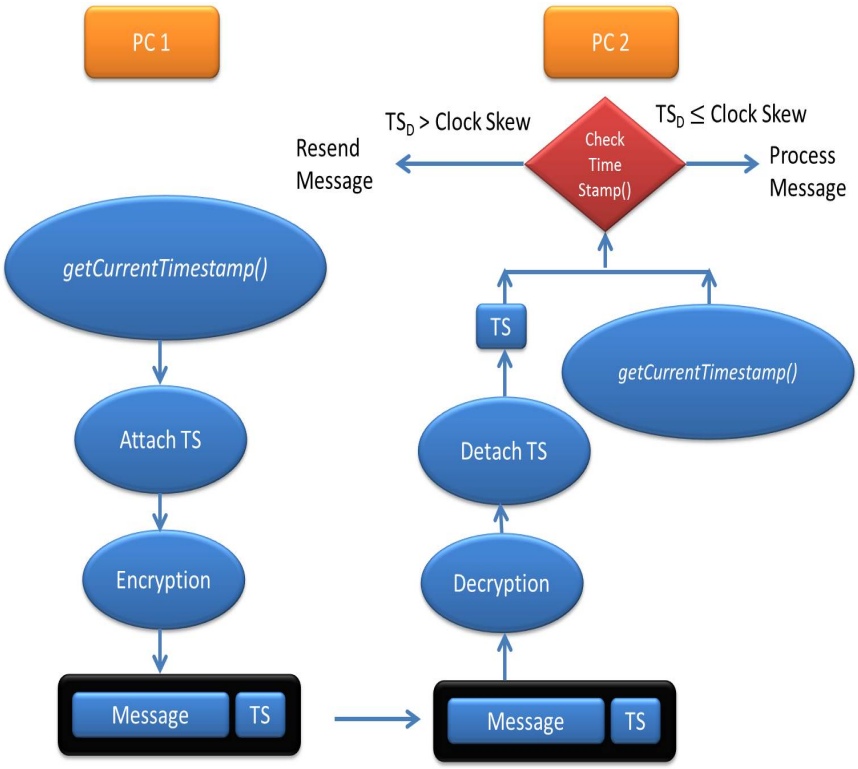

Figure 2.1: Process of checking TS to detect Replay Attacks

2.1.6 Message Authentication Codes (MAC)

MACs are used for the purpose of integrity protection and authentication of messages as the name derives. The principle of such a code is to generate a digest of a message which can uniquely identify the corresponding message. In fact there are several known approaches to create message digests such as symmetric key block cipher modes and hash functions.

The method of computing MAC using a hash function is known as Hashed Message Authentication Code (HMAC). The deployment of hash functions is quite popular, since it is efficient and does not require a key to generate it. There are several hash functions that can be used to compute a HMAC. Some of the hash functions are Message Digest Algorithm 5 (MD5), Secure Hash Algorithm 1 (SHA1), SHA256 and SHA512. In our solution we have used SHA1 hashing algorithm to generate the message digests to integrity protect the data sent over the un-trusted networks. Figure 2.2 illustrates how we have used HMAC for validation of integrity of messages.

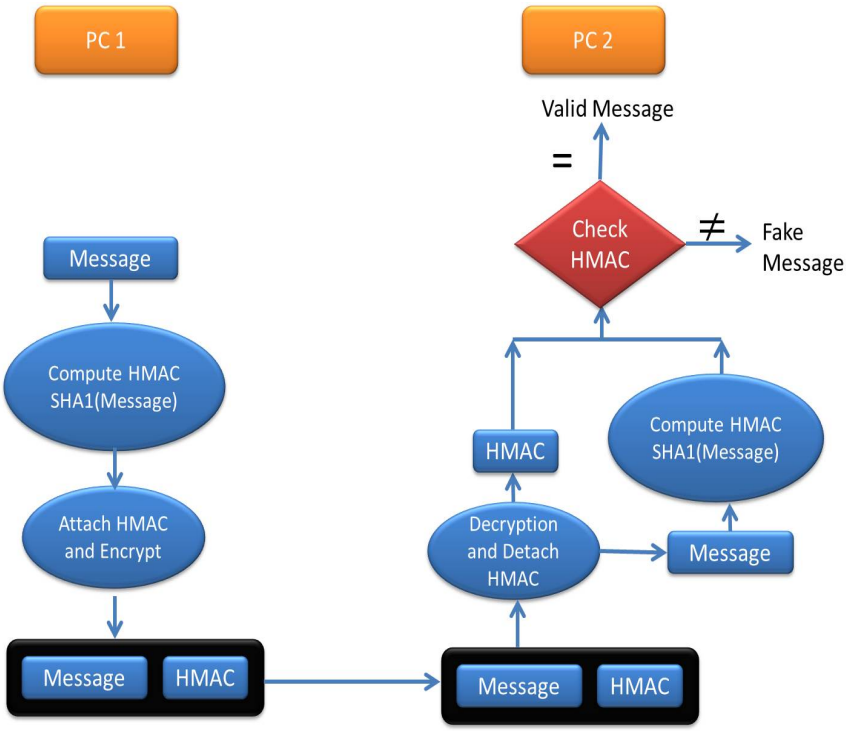

Figure 2.2: Process of checking HMAC to detect Integrity Violations

As seen from Figure 2.2, the HMAC is computed from the function SHA1( ) by feeding the 'Message' as the input parameter. The computed HMAC is sent to the receiving party along with the message in an encrypted environment. At the reception, system will compute the HMAC from the received message and checks it with the received HMAC. Hence, the received message can be identified as a valid one or a tampered one.

2.2 Client-Server Architecture

The client-server model is a computer network architecture which enables the operation of most of the current day web applications. Such web applications could be file sharing, network printing, webmail, collaborating applications, etc. In this system, each process is either a client or a server. Client is a computer terminal or a process which requires accessing or maneuvering a certain web service and its resources. On the other hand, server is a device which shares its resources through a web service or a function. Under normal circumstances, client is usually a computer program or a web application which possess a network connection while server is a high performing computer with a large storage capacity.

Furthermore, client-server architecture could be either two-tier or three-tier. Figure 2.3 illustrates a two-tier client-server architecture in which clients are communicating with the server directly. In three-tier architecture, a middleware or an application server exists between clients and the server.

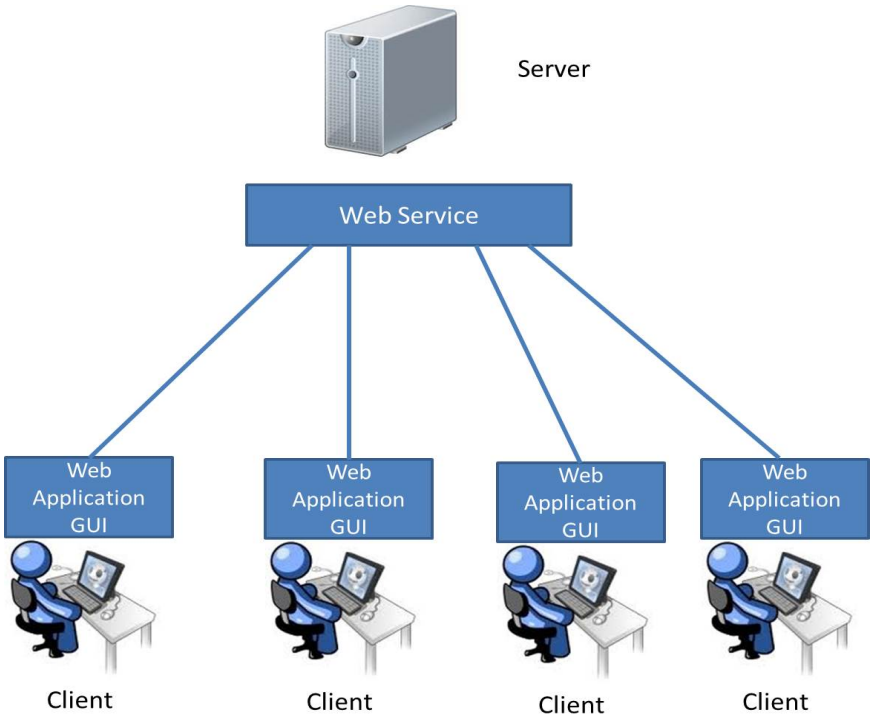

Figure 2.3: Client Server Architecture Illustration

As seen from Figure 2.3, a server serves number of clients simultaneously. Number of client which can be handled by the server is dependent upon the performance and resource capacity of the server. Availability of the server is a vital factor. In order to establish communication between client and a server, certain communication protocols should be adopted. Most of such client-server protocols are implemented as application layer protocols. The methodology of "*socket communication*" is a commonly developed inter-process bi-directional communication method which is suited for client-server architectures. A "*socket*" is defined for a process where an IP address along with a PORT number forms a socket. There are different port numbers being defined for different processes. Once a client socket is initiated, it will send a request to the server which is running the specific web service under the given PORT number. Then the server will respond to the request and establishes the communication between client and server after authentication procedure. Developing such communication protocols using socket communication concepts is called as "*socket programming*".

Most of the programming languages such as Java, C and Python are supporting socket implementations with their inbuilt libraries and packages.

The *Secure Multi-party based Cloud Computing Framework* which we are going to put forward in the next chapter follows a certain resemblance to client-server architectural properties. Therefore when implementing the proposed framework, we have adopted socket communication concepts to achieve higher efficiency from the implemented system.

Chapter 3

Secure Multi-party based Cloud Computing Solution

This chapter of the report mainly describes our solution of Multi-party based Cloud Computing Framework and its implementation scenario in a descriptive manner. Preliminarily, the case which we have formulated is elaborated furthermore. Afterwards, our solution is explicated under different phases of the proposed framework while exemplifying our application by different code segments we used in the implementation.

3.1 Case Description

Before we start the discussion on the proposed secure multi-party based cloud computing model, it is quite important to first remind the case which we build around. The formulated case deals with sales management in a particular organization called 'ABC'. This organization is interested in calculating mean, variance, standard deviation, skewness and kurtosis of their overall sales income at a certain instance of time. Employees of the ABC organization are represented by *Sales Person 1, Sales Person 2,..., Sales Person n* and they are responsible for selling the products manufactured by them. As shown in Figure 3.1, employees are entitled to send the daily sales income details to the servers in the organization's computer network via a software designed for managing daily sales of ABC. Furthermore, the organization wants to outsource the statistical computations to an external entity with higher computational power due to the fact that it lacks computational resources and as well as to overcome the burden of maintaining such resources inside the organization.

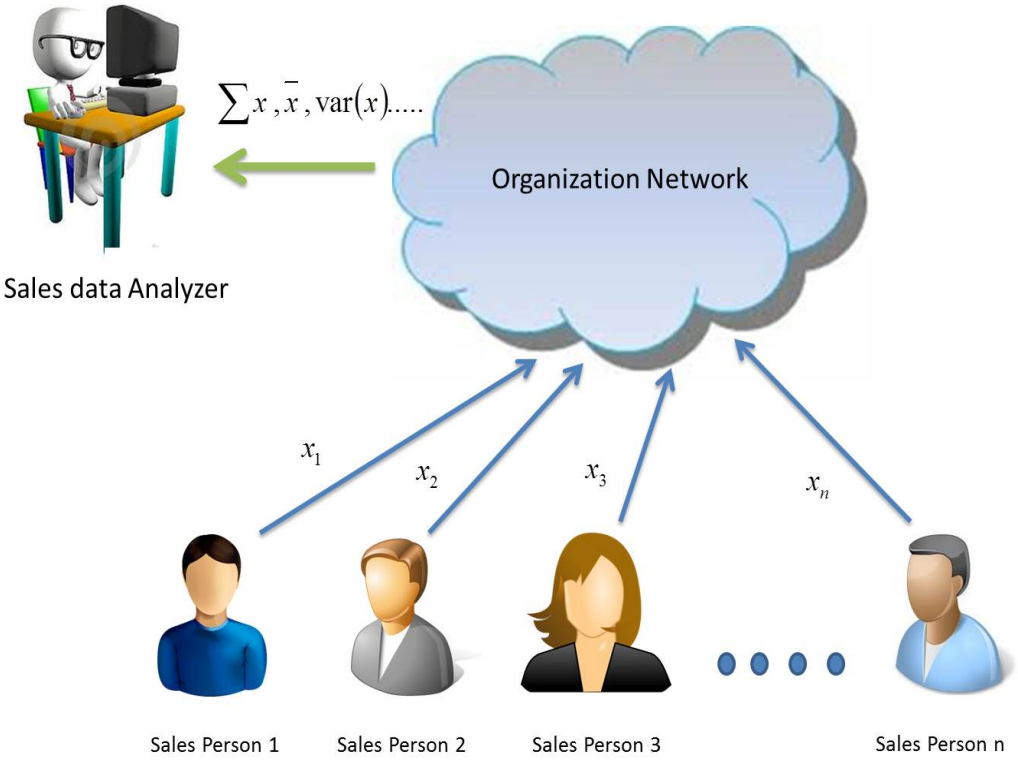

Figure 3.1: Illustration of the Case Study

However, employees value the privacy regarding their sales income which meant that it is not feasible to feed the external party with sales data as plaintext. Moreover, the organization has assigned a third party (Analyzer) to ultimately analyze the computed statistical parameters to aid in the cause of organization's sales related decision making. On the other hand, security and privacy of user inserted data becomes a crucial aspect due to the engagement of external parties invoking the necessity of an efficient security protocol. We are going to consider this case for presenting our solution throughout this report.

3.2 Secure Multi-party based Cloud Computing Framework

Figure 3.2 illustrates the architecture of secure multi-party based cloud computing framework that we have considered to address the case which is discussed in the previous section.

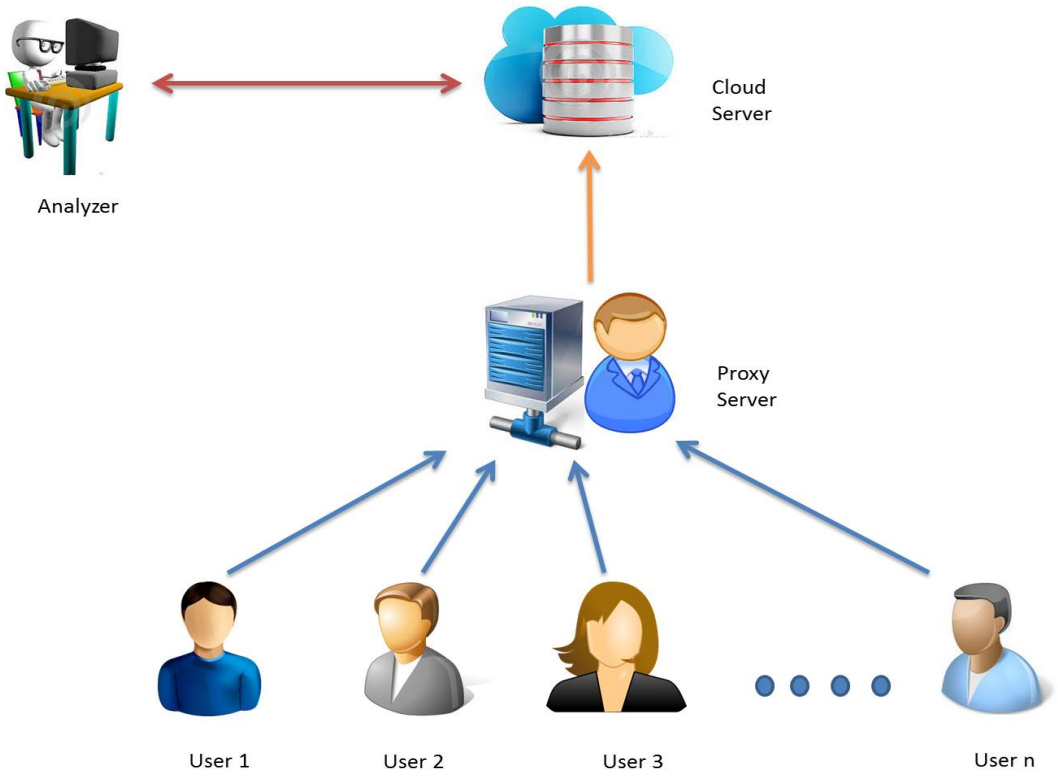

Figure 3.2: Proposed Secure Multi-party based Cloud Computing Framework

The proposed architecture contains four major entities namely cloud server, analyzer, proxy server and parties who provide data for the statistical computations. In order to fulfil the computational requirements of the organization, we have deployed a cloud server considering the benefits of cloud computing such as computational power, high availability, low cost and no burden of maintenance on the perspective of the organization.

As a means of achieving the privacy requirements of users, we are encrypting users' private sales data with a homomorphic encryption scheme which allows the required computations to be carried out on the encrypted data at the cloud. These encrypted user sales data are transferred to the cloud through a proxy server which is the exit point of the organization's internal network. The main functionality of the proxy is to obscure the identities of user data from the cloud server. The analyzer is the external party which receives the statistical parameters of user sales data which is used for analytical purposes. In the sections to follow, we have discussed the operations of each of those entities while explaining how this architecture can achieve a secure cloud computation by enhancing security, privacy and anonymity of user data.

3.3 Key Generation Phase

Before we explain the process of our security framework, it is important to have an idea about the cryptographic keys which we have incorporated with the proposed system.

- Authentication process requires RSA key pairs for proxy server, cloud server and for the analyzer which are generated at the entity itself. For an instance cloud server's RSA key pair is generated at the cloud server, etc. Furthermore, we have shared the RSA public keys among entities before initiating the authentication. Theoretical aspects behind generation of RSA key pairs were stated in Subsection 2.1.3.

- We have used extended ElGamal encryption scheme to encrypt users' private data. Since the computed statistical results of encrypted user data are displayed at the analyzer, an ElGamal key pair for the analyzer is generated at its end while encryption parameters are transferred to the users' end. Furthermore, we have explained the generation of ElGamal parameters in Subsection 2.1.2.

- We also require two 3DES symmetric keys to encrypt data transferred between proxy server and cloud server (K_{DES_PC}) as well as cloud server and analyzer (K_{DES_CA}). We generate the symmetric key K_{DES_PC} at the proxy server while K_{DES_CA} is generated at the cloud server. Furthermore the theoretical concepts behind generation of 3DES keys were explained in Subsection 2.1.4.

3.4 Authentication and Key Exchange Phase

When we consider from the organization's perspective cloud server and the analyzer in our framework are external entities. Therefore, it is necessary to mutually authenticate proxy server to the cloud server as well as cloud server to the analyzer prior to the initiation of information flow between those entities. Furthermore, we also used the process of mutual authentication to share the symmetric keys which are required to encrypt data when sending data from proxy server to the cloud server, cloud server to the analyzer and vice versa. Moreover, we have also established an authentication process for user parties to make sure that only the valid users are involved for the computations. In the following subsections we have put forward those authentication methods in detail.

3.4.1 Mutual Authentication between Cloud Server and Analyzer

The diagram shown in Figure 3.3 illustrates how we have achieved mutual authentication between analyzer and the cloud server.

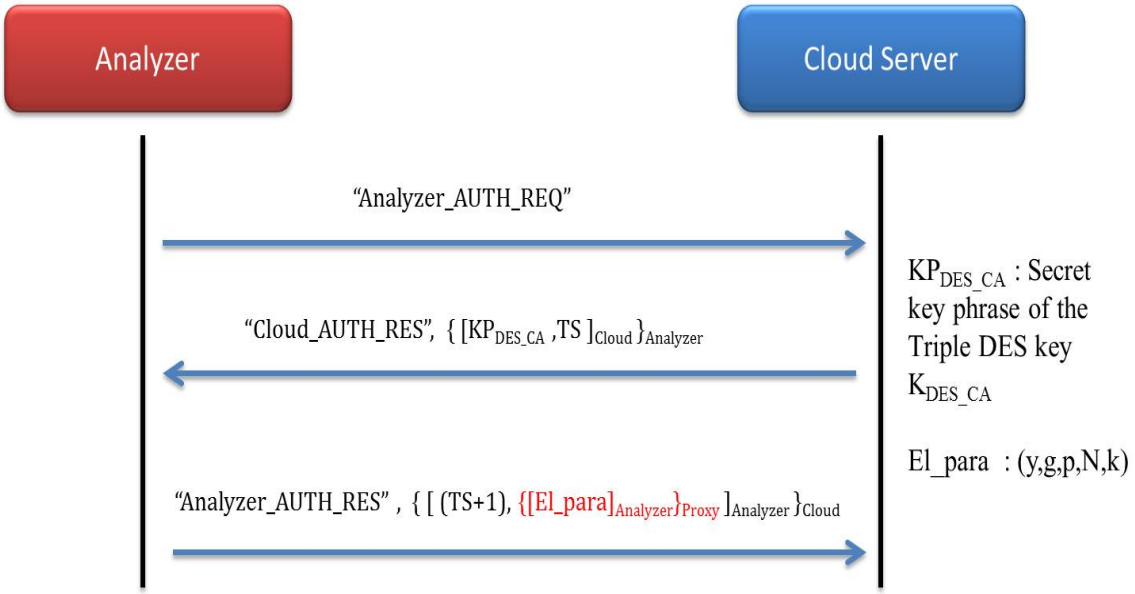

Figure 3.3: Mutual Authentication between Analyzer and Cloud Server

The authentication process is initiated by the analyzer, by sending authentication request message ($Analyzer\_AUTH\_REQ$). After the reception of this message, cloud server generates a message that includes the current timestamp (TS) and the secret key phrase (KP_{DES_CA}) to generate the 3DES key to be shared with the analyzer(K_{DES_CA}). Then the complete message is first signed with its' RSA private key and encrypt again with the RSA public key of the analyzer in order to be forwarded to the analyzer. At the analyzer end, it decrypts the above message by first decrypting it with the RSA private key of the analyzer and thereafter by the public key of the cloud server. After that by considering the content of the decrypted message which is the timestamp (TS); if it is within the defined clock skew, analyzer can authenticate the cloud server and the secret key phrase is saved and shared 3DES key, K_{DES_CA} is being generated.

In order to authenticate itself to the cloud server, analyzer then generates a message by incrementing received timestamp (TS) value by one and attaching ElGamal parameters (y, g, N, p, k) after signing with its private key and then by encrypting it with the public key of the proxy server.

It is important to note that we are sending ElGamal parameters which are required to encrypt user data, in an encrypted environment due to the fact that, the component k is a security critical parameter. Thereafter, the complete message is signed and encrypted with RSA private key of the analyzer and RSA public key of the cloud server respectively. After receiving this message at the cloud server, it decrypts the complete message (first by the private key of the cloud server then by the public key of the analyzer) and extracts the timestamp value. Finally, it checks whether the received value is equals to the value of the original timestamp (TS) value sent to the analyzer incremented by one. If it is verified, analyzer is authenticated to the cloud server whereas the encrypted ElGamal parameters (El_para) are retrieved and saved to be sent to the proxy server.

3.4.2 Mutual Authentication between Proxy Server and Cloud Server

In order to authenticate proxy server to the cloud server and vice versa, we have used a similar approach as discussed above and it is graphically represented in the following Figure 3.4.

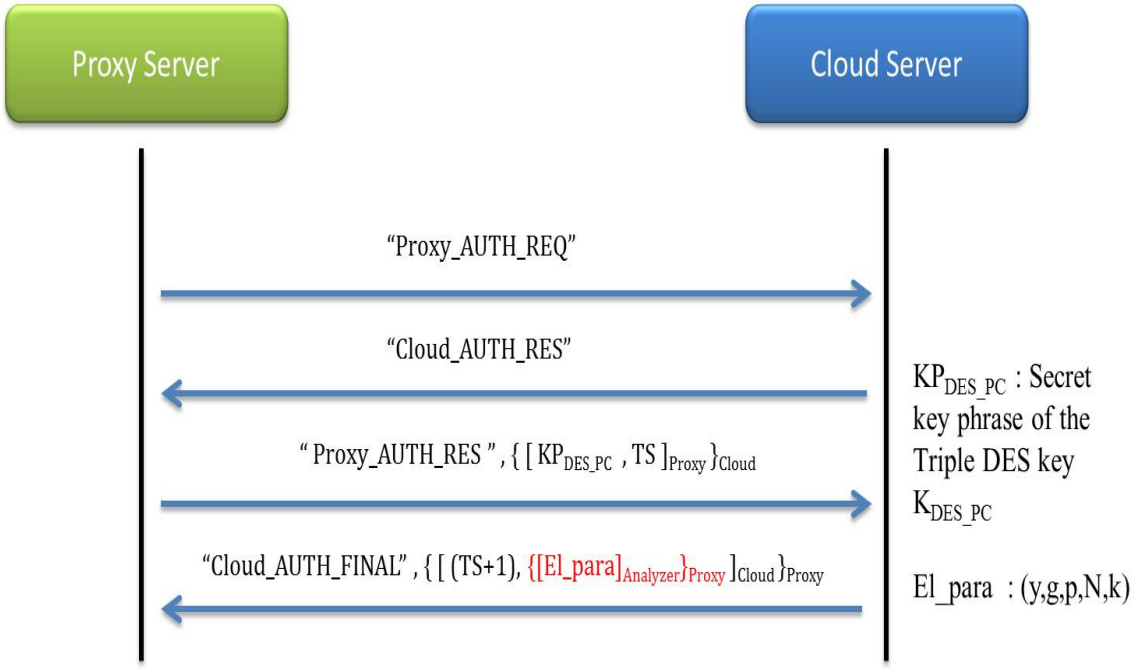

Figure 3.4: Mutual authentication between Proxy server and Cloud server

In this case, authentication is initiated by the proxy server by sending an authentication request ($Proxy\_AUTH\_REQ$). At the reception of this request, cloud server generates a similar response ($Cloud\_AUTH\_RES$) and sends it to the proxy server. Then, the proxy server generates a message by including the current timestamp (TS) and the secret key phrase (KP_{DES_PC}) which is required to generate the 3DES symmetric key shared between proxy and the cloud server (K_{DES_PC}). After that, the complete message is signed and encrypted using the private key of the proxy and public key of the cloud server respectively. The rest of the authentication process is identical to what we have discussed in the previous section. So, ultimately when mutual authentication is established, cloud server will be able to generate the 3DES symmetric key, K_{DES_PC} from the shared secret key phrase while proxy server attains the ElGamal parameters (El_para). Then the proxy server could attain the unencrypted ElGamal parameters by first decrypting it with its' RSA private key and thereafter by the public key of the analyzer.

3.4.3 Authentication of Users

It is quite important to authenticate users which assure only the valid users are allowed to take part in the computations. Furthermore, the diagram shown in Figure 3.5 illustrates how we have established the authentication procedure for user parties.

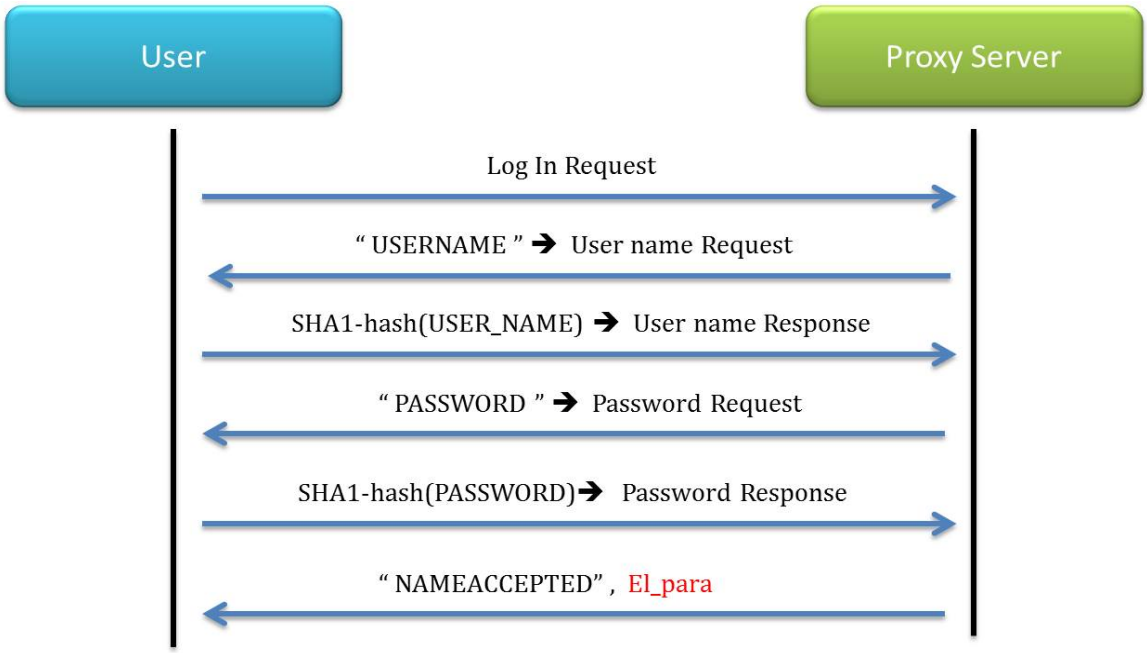

Figure 3.5: User Authentication Procedure with Proxy Server

In order to accomplish this, we have created a file in the proxy server which includes SHA1 hashes of username and password pairs of valid users. When a user logs into the system, SHA1 hashes of the entered username and password are transferred to the proxy server in order to be validated. If login information is validated, then the user will be authenticated and the proxy server will forward the ElGamal parameters to the user end which are necessary for the users to encrypt the private data that are sent for the computation.

3.5 User Data Encryption

Each user can start sending private data for computations after being successfully authenticated into the system as we have discussed in the previous section. When data is entered by a user party, first it is encrypted using the extended ElGamal homomorphic cryptosystem which exhibits somewhat fully homomorphic properties [22]. Furthermore, the concepts related to this encryption scheme were given in Subsection 2.1.2. The following code fragment shows how we have implemented extended ElGamal encryption function in our system.

Code Fragment 3.1: Extended ElGamal Encryption Function

```
1   public void Encryption(String SalesValue){
2   BigInteger N;     // N
3   N = p.multiply(q); //    N = p * q
4   BigInteger SV = new BigInteger(SalesValue);
5   // bx = SV + (r * p) mod N
6   BigInteger bx = SV.add(r.multiply(p)).mod(N);
7   // b = ( bx * ( y ^ k ) mod p ) mod p
8   BigInteger b = bx.multiply(y.modPow(k, p)).mod(p);
9   // a = (g ^ k) mod p
10  BigInteger a = g.modPow(k, p);
11  A = a.toString();
12  B = b.toString();
13    }
```

According to the encryption scheme, p, q, k, y, g, and r values are pre-computed before the encryption step. In our program, sales value of a certain sales person is fetched into the encryption function (e.g.: Encryption ($SALES\_VALUE$);).

First of all, the value N is computed according to the formula, $N = p.q$. It is important to note that all these values are represented as Big Integers. This data type is the convenient option for modular operations. Line 4 in Code Fragment 3.1, converts the input string 'sales value' into a 'BigInteger'. Lines 6, 8 and 10 of the above code fragment corresponds to the Equations 2.3 and Equation 2.4 in the extended ElGamal algorithm. Finally, generated encrypted values a and b are converted to String data type values A and B in order to be conveyed to the proxy server. Thereafter each user party will transmit the encrypted information to the proxy server. Figure 3.6 illustrates the structure of the message sent from a user party to the proxy server.

Figure 3.6: Structure of User Data Message sent to the Proxy Server

3.6 Proxy Server Functionality

The main idea of having a proxy server is to hide the identity of data sent from each user from the cloud server. At the beginning of a computation, proxy server waits till it receives encrypted data from all the users. After receiving all of them, it will create a new message including encrypted data of all the users, number of users (n) and uses the SHA 1 algorithm to generate a Hashed Message Authentication Code (HMAC) of the preceding message. Then the generated message digest is appended to the original message along with the current timestamp. It is important to note that the theoretical aspects related to timestamps and SHA 1 message digest were given in Subsections 2.1.5 and 2.1.6.

Finally, the complete message is encrypted with the 3DES symmetric key shared between the proxy and cloud server (K_{DES_PC}) and forwarded to the cloud server. Furthermore, the structure of the message sent from the proxy server is illustrated through the following Figure 3.7.

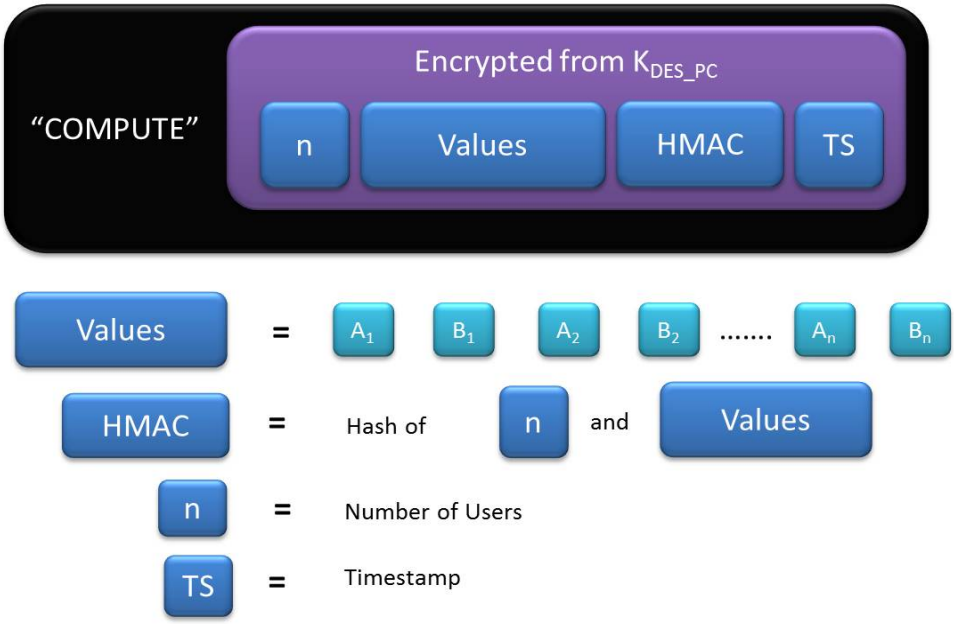

Figure 3.7: Structure of the Packet sent from Proxy Server to Cloud Server

3.7 Cloud Server and Analyzer Functionality

In order to explain the rest of the process, consider that the cloud server is already authenticated to the analyzer as well as to the proxy server. Then, it continuously waits for encrypted data of users from the proxy end which needs to be handled. After receiving such a data set, cloud server first decrypts it with the 3DES key shared with the proxy (K_{DES_PC}) and separates the three components of the complete message which are timestamp, HMAC of the original message and the original message. Furthermore, original message refers to the ElGamal encrypted private data of all users which must be statistically processed and the number of users (n). Firstly, the cloud server checks the received timestamp is within the defined clock skew to make sure that it is not a replayed frame and if it is verified, then a HMAC is created with the original message part and check whether the computed and received HMACs are matching. If so, the private data of users are accepted and otherwise a message will be sent to the proxy server to resend the message. The process that we have discussed is represented by the following Figure 3.8.

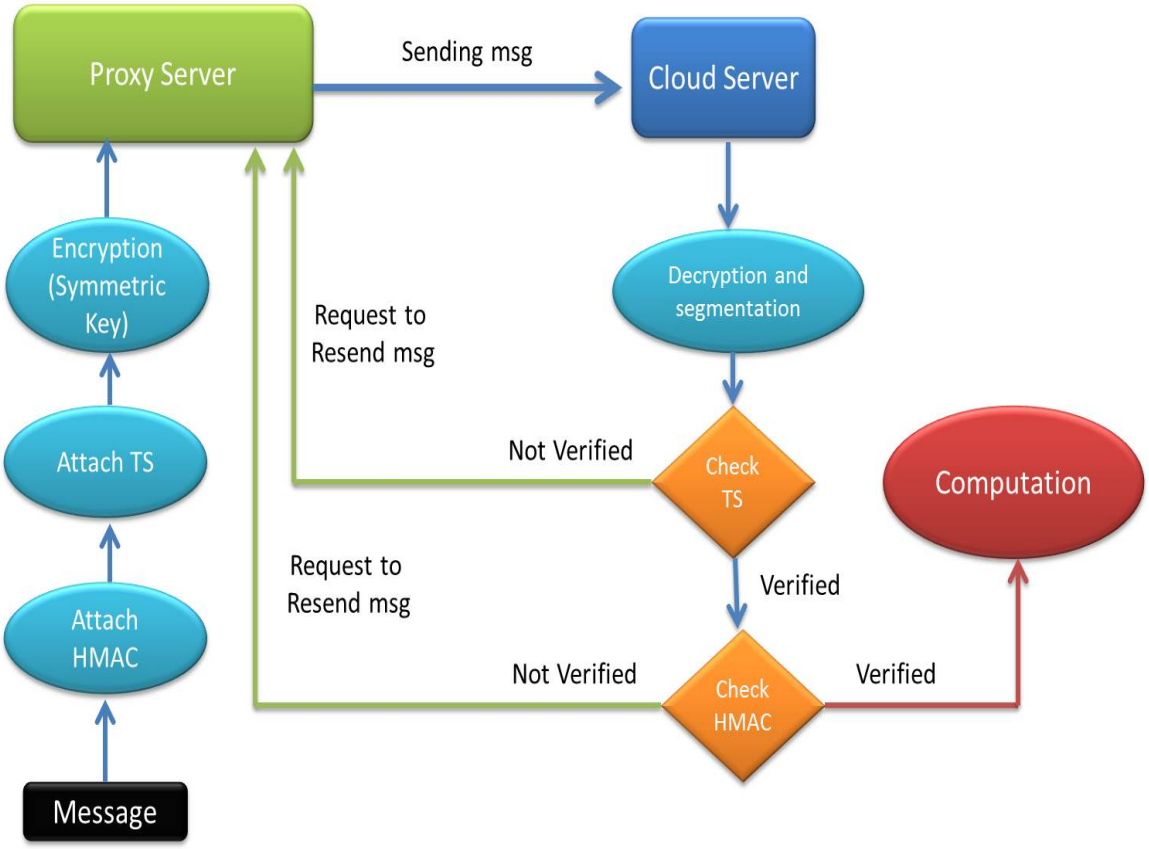

Figure 3.8: Flow Diagram of the Process when the Proxy Server Message is received at Cloud Server

Now we can discuss the procedure that we have followed to compute statistical parameters from encrypted user data. Let us consider that the ElGamal encrypted user data received at the cloud server of the n users who involved in the computation are represented as (a_1, b_1), (a_2, b_2),..., (a_n, b_n). However, according to the ElGamal encryption formula given in Equation 2.3 which compute the first encryption component of a particular sales data of a user $(a_1, a_2, ..., a_n)$;

$$a_i = g^k \ mod \ p, \ \ for \ \ i = 1, 2, ..., n$$

Since, we have kept g, k and p common for all the users; it is clear that;

$$a_1 = a_2 = a_3 = ... = a_n = a \tag{3.1}$$

Then, ElGamal encrypted sales data received at the cloud server can be simplified to (a, b_1), (a, b_2),..., (a, b_n).

In the subsections to follow we have discussed how we can compute statistical parameters of sales data by processing encrypted sales data at the cloud server and made them available at the analyzer end. First of all, we start with the computation of mean value of sales data.

3.7.1 Computing Mean Value

By using the property of additive homomorphism of the extended ElGamal encryption scheme, we process the sales data at the cloud server according to the formula given below.

$$b = b_1 + b_2 + b_3 + ... + b_n \tag{3.2}$$

Implementation of the property of additive homomorphism in extended ElGamal algorithm is achieved from the following *Addition*() function given in the Code Fragment 3.2.

Code Fragment 3.2: Function for Computing Summation over Encrypted Data

```
1 public static String Addition(BigInteger A[], BigInteger B[], Integer n){
2       BigInteger BIA = A[0];
3       BigInteger BIB = new BigInteger("0");
4       for(int j = 0 ; j < n ; j++){
5               BIB = BIB.add(B[j]);
6       }
7  return (BIA.toString()+" "+BIB.toString());
8  }
```

This function is fed with arrays of A and B, which represent the components of encrypted values along with total number of users n (i.e. Components $A[i]$ and $B[i]$ represent the encrypted value of the user party i). According to the Equation 3.2, only $B[\,]$ array components are added under the $for()$ loop whereas $A[\,]$ is unaltered (i.e. $A[i] = A_{Sum}\ \forall\, i$). Finally, the result is returned as a combined String of A_{Sum} and B_{Sum} components.

Then, cloud server generates a new message by including the values A_{Sum}, B_{Sum} and the number of users n. It is important to note that the exteded ElGamal encryption scheme does not possess homomorphism for division, so that the mean is calculated from the summation at the analyer end.

After that HMAC of the created message is generated through SHA 1 hash algorithm and it is appended to the original message along with the current timestamp. Finally, the complete frame is encrypted with the 3DES symmetric key shared with the analyzer (K_{DES_CA}) and transmit it to the analyzer. Moreover, Figure 3.9 represents the above mentioned transmitted data frame by the cloud server.

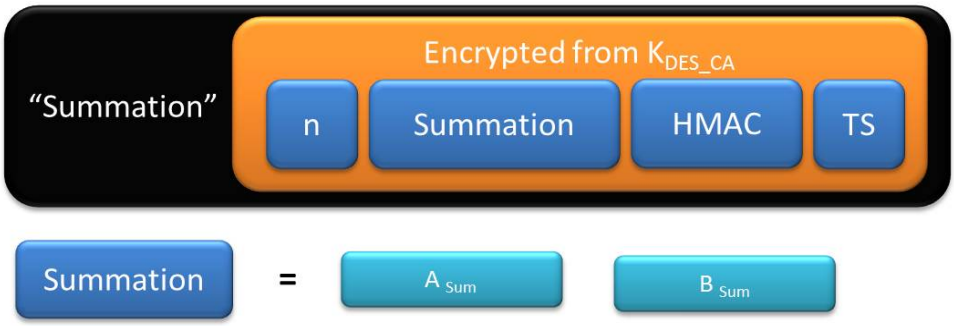

Figure 3.9: Structure of the Summation Frame sent from Cloud to Analyzer

After receiving the data frame from the cloud server, as usual the analyzer segment the complete message to timestamp, HMAC and the original message by decrypting it with K_{DES_CA}. Then the analyzer verifies the timestamp as well as the HMAC by recreating the HMAC from the received original message and matching it with the received HMAC. If the process is successful, the original message is saved, otherwise a message is sent to the cloud server asking to resend the message. Assuming the process is successful, analyzer then separates the encrypted value (A_{Sum}, B_{Sum}) and the value for number of users n. After that, the encrypted value (A_{Sum}, B_{Sum}) is decrypted using the analyzer's private key of the extended ElGamal cryptographic scheme.

The following Code Fragment 3.3 represents how we have implemented the *Decryption*() function of the extended ElGamal encryption scheme.

Code Fragment 3.3: Decryption Function of Extended ElGamal Scheme

```
1  public BigInteger Decryption(BigInteger A, BigInteger B,BigInteger p,
   BigInteger x){
2  B = B.mod(p);
3  BigInteger ZX = A.modPow(x,p);   // ZX = A ^ x mod p
4  // Z  = B * ((ZX^(-1)mod p) mod p
5  BigInteger Z = B.multiply(ZX.modInverse(p)).mod(p);
6  return Z;
7  }
```

This function requires the input parameters A, B, public ElGamal constant p and x which is the private key as BigIntegers. First, the modular value of B is taken w.r.t the base value p. Line 3 corresponds to the formula $ZX = A^x \ mod \ p$. In the same way line 5 corresponds to the Equation 2.5. Finally, the decrypted value is returned as a 'BigInteger'.

If the resulting decrypted value is given by M_1, it is possible to recover the mean value of sales data as given below.

$$Mean\ value\ of\ user\ sales\ data\ = \frac{M_1}{n} \tag{3.3}$$

In order to compute the rest of the statistical parameters variance, standard deviation, skewness and kurtosis; it is necessary to obtain the mean value which is encrypted using the extended ElGamal scheme. Therefore, after recovering the mean value by the analyzer, it re-encrypts the mean value using the same ElGamal parameters (y, g, N, p, k) which gives us the encrypted components A_{Mean} and B_{Mean}. Furthermore, the analyzer generates a HMAC for the newly encrypted mean and appends it to the encrypted mean. Finally, the current timestamp is also attached to the tail of the message and forwards it to the cloud server after encrypting the complete frame with the shared 3DES symmetric key, K_{DES_CA}. The generated new data frame is illustrated in Figure 3.10.

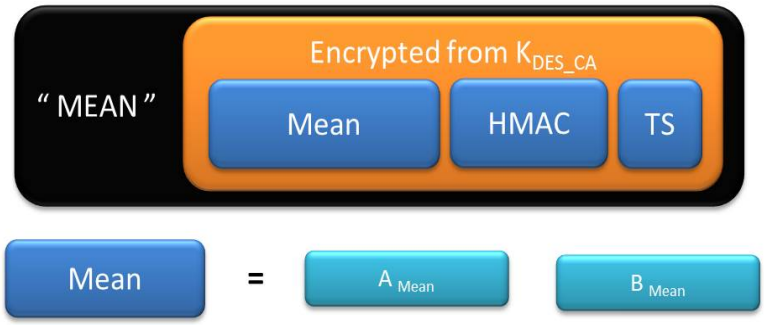

Figure 3.10: Structure of the message which carries encrypted Mean value to Cloud

After receiving the above data frame, cloud server decrypts it with the shared 3DES key, K_{DES_CA} and extracts the encrypted mean value, if the timestamp and HMAC of encrypted mean is validated. Now, it is possible for us to discuss how we can compute the rest of the statistical parameters.

3.7.2 Computing Variance, Standard deviation, Skewness and Kurtosis

According to the definitions of variance, standard deviation, skewness and kurtosis given below;

$$Variance : \mathbf{S^2} = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2 \tag{3.4}$$

$$Standard\ Deviation : \mathbf{S} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2} \tag{3.5}$$

$$Skewness : \gamma_\mathbf{1} = \sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{S}\right)^3 \tag{3.6}$$

$$Kurtosis : \gamma_\mathbf{2} = \sum_{i=1}^{n}\left(\frac{x_i - \bar{x}}{S}\right)^4 \tag{3.7}$$

where;

x_i - Sample value of i^{th} sample
$\bar{x}$ - Mean of sample values
n - Number of samples

It is required to calculate the deviation values $(x_i - \bar{x})$ in order to compute the above mentioned statistical parameters. So, we need to use subtraction property which is obtained from the additive homomorphism of the extended ElGamal cryptosystem. Code Fragment 3.4 given below explains the *Subtraction*() function that we have implemented.

Code Fragment 3.4: Function for Computing Subtraction over Encrypted Data

```
1  public static String Subtraction(BigInteger A1, BigInteger
   B1,BigInteger A2, BigInteger B2){
2   BigInteger BIA = A1;
3   BigInteger BIB = B1.subtract(B2);
4   return BIA.toString()+" "+BIB.toString();
5  }
```

This function will result the subtraction of two values V_1 and V_2 in which input parameters A_1, B_1 and A_2, B_2 represent the encrypted components of V_1 and V_2 respectively. Line 3 of the above code segment computes the value $(B_1 - B_2)$ while A component of the answer remains common same as the *Addition*() function discussed earlier. The output will be returned as a combined String value. Then we can find the mean deviations of encrypted user data from the following expression.

$$String\ Deviation = Subtraction(A[i], B[i], MeanA, MeanB)$$

As for the above relation, A_i and B_i components (same as $A[i]$ and $B[i]$ array elements) of the i^{th} user's encrypted sales value and $MeanA$, $MeanB$ components of the encrypted mean are inserted as input parameters for the function *Subtraction*() which results the required deviation. Since, *Subtraction*() function returns a *String* value, the variable *Deviation* acquires the returning value of the function. Once the deviation is calculated, it is necessary to compute $(x_i - \bar{x})^2$, $(x_i - \bar{x})^3$ and $(x_i - \bar{x})^4$. For that, computing 2^{nd}, 3^{rd} and 4^{th} powers of encrypted values is a necessity. If the encrypted (with ElGamal) value of V_i is represented by A_i and B_i components; then, j^{th} power of V_i (or V_i^j) is obtained by ${A_i}^j$ and ${B_i}^j$ components. Accordingly, 2^{nd}, 3^{rd} and 4^{th} powers of encrypted values are implemented through following Code Fragment 3.5.

Code Fragment 3.5: Functions for Computing 2^{nd}, 3^{rd} and 4^{th} Powers of Encrypted Data

```
1   public static String Square(BigInteger A, BigInteger B){
2       BigInteger BIA = A.pow(2);
3       BigInteger BIB = B.pow(2);
4       return BIA.toString()+" "+BIB.toString();
5   }
6   public static String Cube(BigInteger A, BigInteger B){
7       BigInteger BIA = A.modPow(new BigInteger("3"), p);
8       BigInteger BIB = B.modPow(new BigInteger("3"), p);
9       return BIA.toString()+" "+BIB.toString();
10  }
11  public static String Biquadrate(BigInteger A, BigInteger B){
12      BigInteger BIA = A.modPow(new BigInteger("4"), p);
13      BigInteger BIB = B.modPow(new BigInteger("4"), p);
14      return BIA.toString()+" "+BIB.toString();
15  }
```

Code Fragment 3.6 shows the computation of square, cubic and biquadratic mean deviations of each A_i and B_i values which are acquired by $SDA[\,]$, $SDB[\,]$, $CDA[\,]$, $CDB[\,]$, $QDA[\,]$ and $QDB[\,]$ arrays respectively.

Code Fragment 3.6: Functions for Computing Square, Cube and Biquadratic Mean Deviations on Encrypted Data

```
1   for(int j = 0 ; j < i ; j++){
2     String Deviation = Subtraction(A[j],B[j],MeanA,MeanB);
3        String Z[] = Deviation.split(" ");
4          BigInteger DA = new BigInteger(Z[1]);
5          BigInteger DB = new BigInteger(Z[2]);
6     String SquareDeviation = Square(DA,DB);
7        String O[] = SquareDeviation.split(" ");
8          SDA[j] = new BigInteger(O[1]);
9          SDB[j] = new BigInteger(O[2]);
10    String CubicDeviation = Cube(DA,DB);
11       String O1[] = CubicDeviation.split(" ");
12         CDA[j] = new BigInteger(O1[1]);
13         CDB[j] = new BigInteger(O1[2]);
14    String BiquadrateDeviation = Biquadrate(DA,DB);
15       String O2[] = BiquadrateDeviation.split(" ");
16         QDA[j] = new BigInteger(O2[1]);
17         QDB[j] = new BigInteger(O2[2]);
18  }
```

Once these values are calculated, all the array elements are added to obtain variance and other values as illustrated by the following Code Fragment 3.7.

Code Fragment 3.7: Computation of Variance, Skewness and Kurtosis

```
1   String Variance = Addition(SDA,SDB,i);
2   String Skewness = Addition(CDA,CDB,i);
3   String Kurtosis = Addition(QDA,QDB,i);
```

The addition of array elements will produce the two encrypted components of encrypted variance, encrypted skewness and encrypted kurtosis as ($A_{Variance}$, $B_{Variance}$), ($A_{Skewness}$, $B_{Skewness}$) and ($A_{Kurtosis}$, $B_{Kurtosis}$) respectively.

Finally, these computed values are sent to the analyzer by attaching the HMAC of the computed set of values, current TS while encrypting the complete frame with the symmetric key K_{DES_CA}. Furthermore, the structure of the message sent to the analyzer is shown in Figure 3.11.

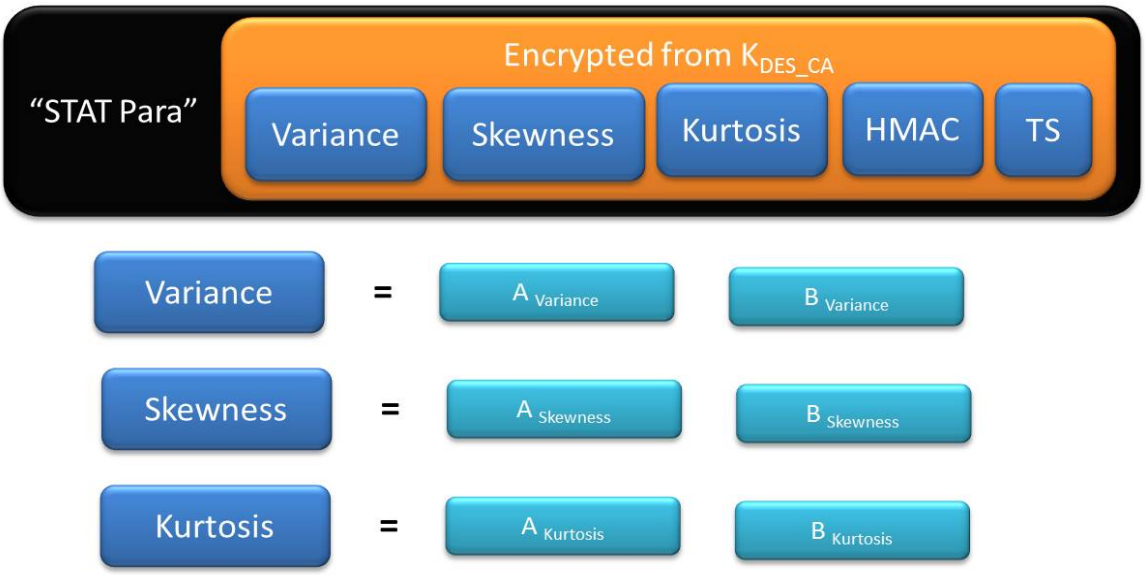

Figure 3.11: Structure of the Message sent from Cloud to Analyzer with Computed Statistical Values

When the above message is received at the analyzer, it will first decrypt the frame using the shared symmetric key K_{DES_CA} and check the validity of TS and HMAC. Once those checks are cleared, rest of the message will be segmented to obtain the encrypted statistical values. Then, those values are decrypted from the ElGamal private key which is possessed by the analyzer.

If the resulting decrypted values are given by V_1, S_1 and K_1; then,

$$Variance\ of\ the\ users'\ sales\ data\colon Variance = \frac{V_1}{n} \tag{3.8}$$

$$Standarad\ Deviation\ of\ the\ users'\ sales\ data\colon S.D. = \sqrt{\frac{V_1}{n}} \tag{3.9}$$

$$Skewness\ of\ the\ users'\ sales\ data\colon Skewness = \frac{S_1}{(S.D.)^3} \tag{3.10}$$

$$Kurtosis\ of\ the\ users'\ sales\ data\colon Kurtosis = \frac{K_1}{(S.D.)^4} \tag{3.11}$$

3.8 Overall Process

The complete algorithm of the proposed framework that we have discussed throughout the previous sections is summarized in the following flow diagram shown in Figure 3.12.

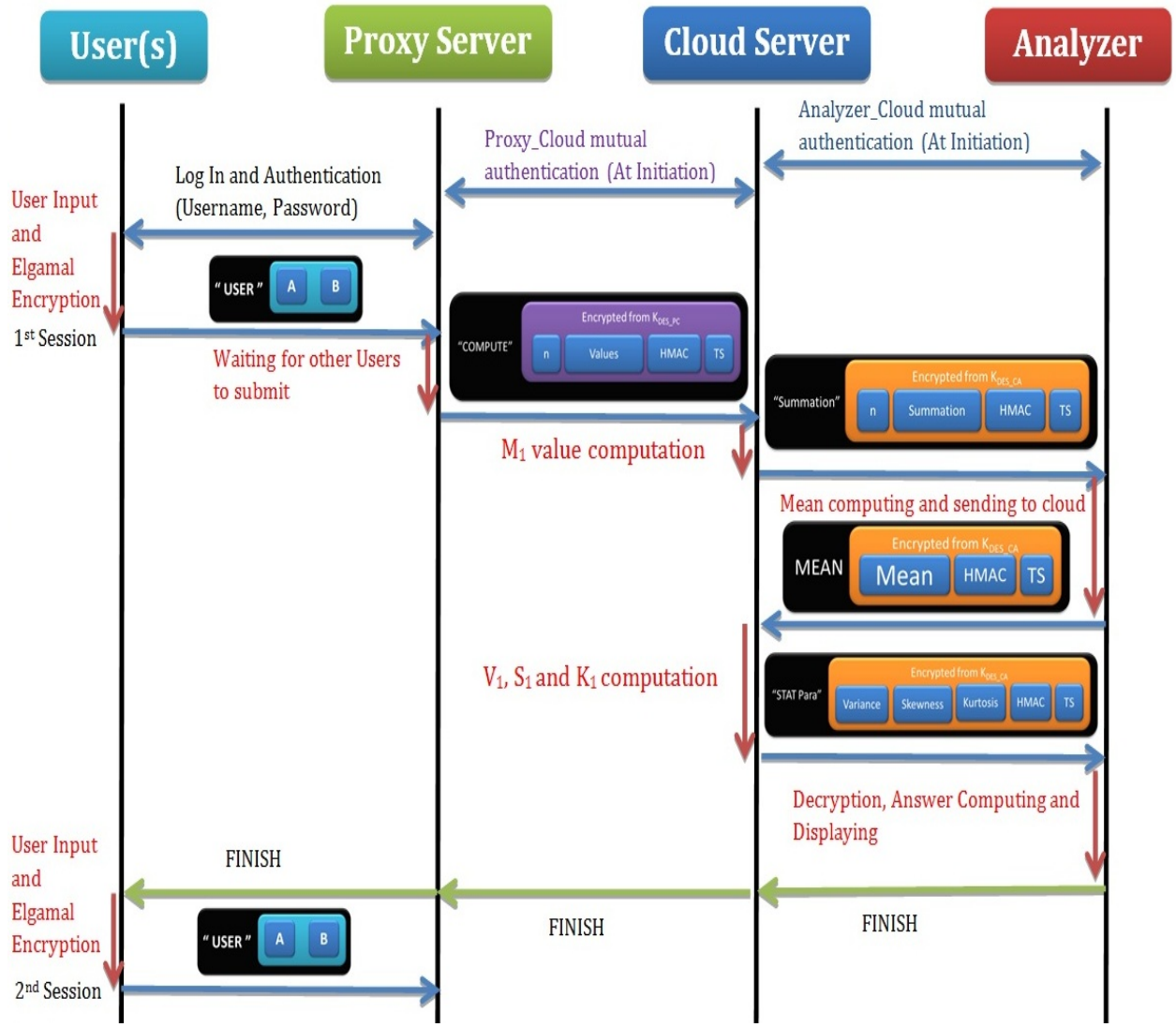

Figure 3.12: Overall Process of the Proposed Model

As shown in Figure 3.12, at the initiation it is necessary to first mutually authenticate all the external entities. After that users who are willing to participate for the computation must authenticate themselves to the proxy server using their usernames and passwords. When users insert their private sales values, these values will be encrypted using the extended ElGamal encryption scheme and forwarded to the proxy server. After receiving encrypted values from all the users, it creates a new message by combining all those data and transmits to the cloud server.

At the cloud server, it first computes the summation of received encrypted data and sends it along with the number of users (n) to the analyzer. Then, the analyzer can compute the mean sales value by decrypting the received encrypted summation value. In order to compute the other statistical parameters, it is necessary to have encrypted mean value of sales data. So, the analyzer re-encrypts the calculated mean value using the same extended ElGamal parameters and transmits to the cloud server. Finally, the cloud server computes the parameters V_1, S_1, K_1 and forwards to the analyzer, where those values are decrypted and computed to obtain the actual variance, standard deviation, skewness and kurtosis of user inserted sales data.

After the completion of a computing session, analyzer will send a message to all the users through the proxy server to initiate the next session. Moreover, it is not necessary to carry out mutual authentication between entities again at the start of a new session. However, to ensure the freshness, we flush all existing keys after consecutive 10 sessions. At that time, it is necessary to carry out the mutual authentication process again before starting a computing session.

Chapter 4

Performance Analysis

In this chapter our main focus is to analyze the performance of our Secure Multi-party based Cloud Computing implementation. So, we start the chapter by introducing the experimental setup that we have considered and then we illustrate the various test cases and the corresponding results which aid us to define performance parameters related to our implementation.

4.1 Experimental Setup

It is important for us to carry out certain experiments to evaluate the performance of our proposed model. So, in order to achieve that goal, we have formulated an experimental setup which is illustrated in Figure 4.1. Furthermore, the structure of the setup is identical to the proposed model given in Figure 3.2. The Java program which we have constructed, executes each entity (Users, Proxy Server, Cloud Server and Analyzer) as a separate program having interfaces of their own. Each user is accessing the system as a client to the proxy server program while it sends collective data gathered from users to the cloud server in order to be processed. In our experimental setup, we are running all the user programs in one Personal Computer (PC). We have used other PCs for deploying proxy server and analyzer programs. Moreover, the cloud server program is running on a separate server at the University of Agder. We have established the communication between the entities using a socket communication approach. Once all the user data is entered into the interfaces of user programs and after processing the encrypted data at the cloud, the analyzer program interface would display the results of the computation.

Furthermore, we have generated timing values which indicate the data acquiring time and conveying time along with computational times at interfaces of each entity in order to capture the required time parameters to evaluate the performance of the model.

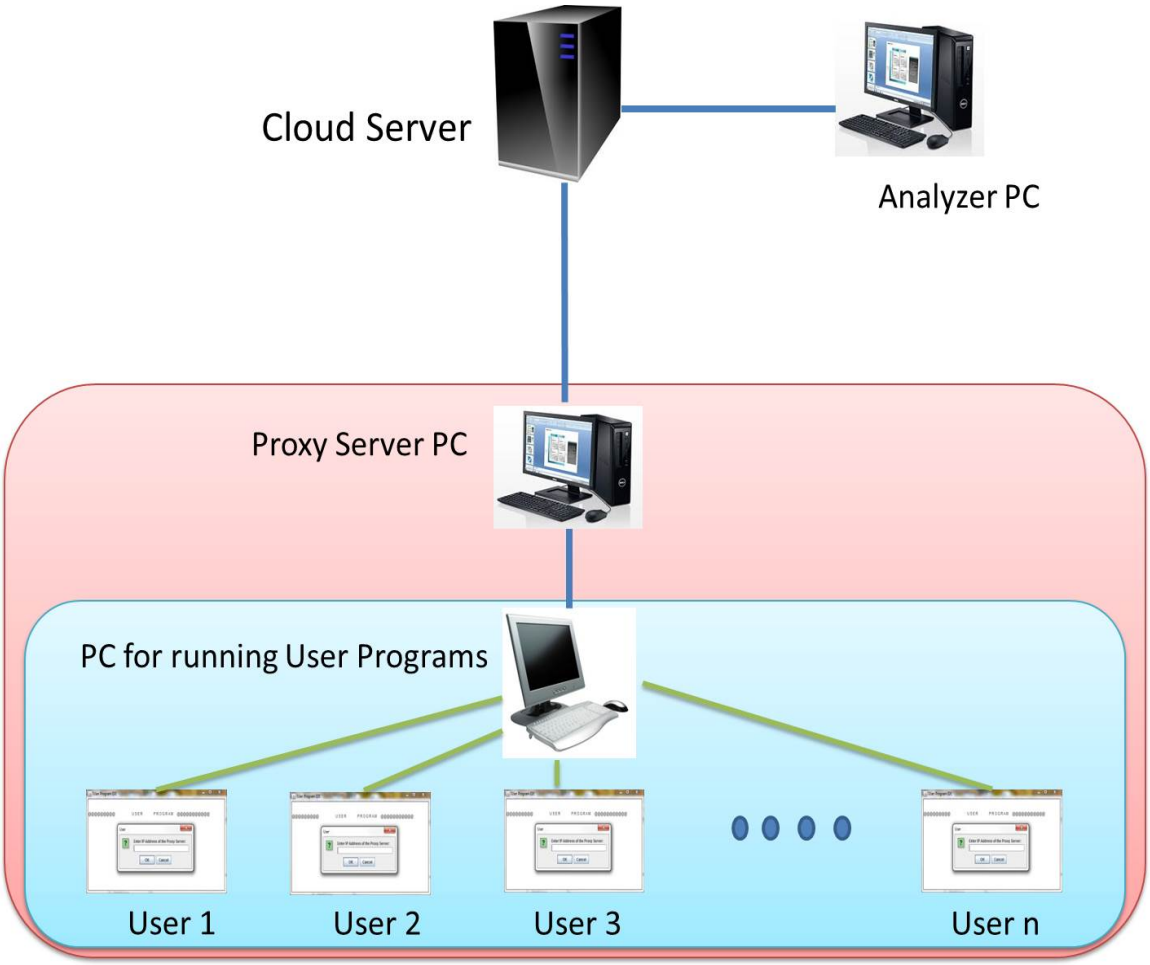

Figure 4.1: Experimental Setup

The system specifications of the PCs and servers are mentioned below.

- System specifications of the Server which runs the cloud server program:
  - Processor: Intel(R) Xeon(R) CPU X5650 @ 2.67GHz (4 CPUs), ~2.7 GHz.
  - Memory: 8190 MB RAM.
  - Operating System : Windows Server 2008 Standard (6.0, Build 6001).
  - Java Version : 1.7.

- System specifications of the PC which runs the Proxy Server program :
  - Processor:Intel(R) Core(TM) i5 CPU M460 @ 2.67GHz (4 CPUs),$\sim$ 2.7GHz.
  - Memory: : 4096 MB RAM.
  - Operating System : Windows 7 Home Premium 32-bit (6.1, Build 7600).
  - Java Version : 1.6.0_37.

- System specifications of the PCs which runs user program and Analyzer program (Both PCs have identical specifications) :
  - Processor: Intel(R) Core(TM)2 Duo CPU P8700 @ 2.53GHz (2CPUs), $\sim$2.5GHz.
  - Memory: : 3072 MB RAM.
  - Operating System : Windows 7 Home Premium 32-bit (6.1, Build 7601).
  - Java Version : 1.7.0_21.

4.2 Test Cases

In order to measure the performance of the proposed Multi-party based Cloud Computing Framework implementation, we have considered the following test cases.

- Authentication Time.
- Variation of User Data Encryption Time.
- Comparison on Computational Time for different Statistical Parameters.
- Effect of Number of Users and Size of User inserted Data on Entity Process Time and Total Process Time.
- Effect of Transmission Delay.

4.2.1 Authentication Time

In our proposed solution, as we have already mentioned in Chapter 3, it is necessary to establish a mutual authentication between cloud server and the analyzer as well as cloud server and the proxy server at the initiation.

Furthermore, we need to establish mutual authentication again, after a pre-defined number of computations where we flush the existing keys and re-generate keys to improve the level of security. So, it is important for us to first find out the associated delays for the authentication process.

As we have discussed in Section 3.3 and 3.4, we have implemented the mutual authentications by using RSA Public Key cryptosystem. Therefore, authentication time mainly depends upon the bit size of the prime numbers (p, q) which are used to generate respective RSA keys, for the intention of signing and encrypting authentication messages. Furthermore, it is important to select RSA primes according to the size of the authentication messages. So, it is necessary for us to use at least 512 bit RSA keys, since our authentication messages consists of current Timestamp (approximately 184 bits) value and ElGamal parameters required for encryptions (approximately 320 bits). Table 4.1 and Figure 4.2 shown below illustrate the variation of authentication time along with the RSA prime value size starting from 256 bits.

Table 4.1: Variation of Authentication Time with RSA Prime Size

RSA Prime Value Size (bits)	Proxy-Cloud Authentication Time (ms)	Analyzer-Cloud Authentication Time (ms)
256	40	278
384	70	318
512	114	376
640	208	450
768	327	586
896	473	736
1024	696	948
1152	971	1216
1280	1317	1541
1408	1695	1865
1536	2153	2431
1664	2716	2975
1792	3415	3742
1920	4277	4387
2048	5001	5298

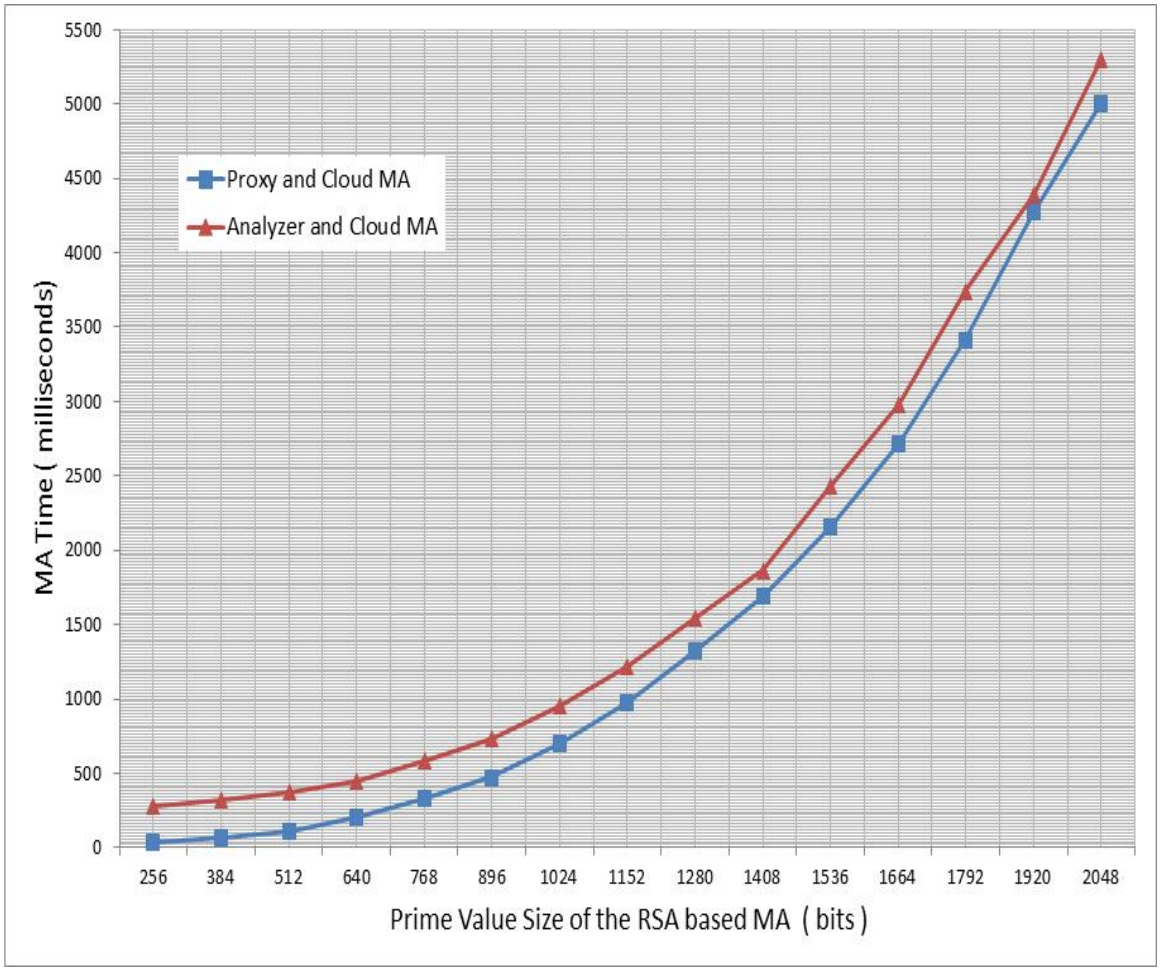

Figure 4.2: Variation of Mutual Authentication (MA) Time as a function of Size of the RSA Prime Values

Figure 4.2 depicts that both cloud-analyzer MA time and proxy-cloud MA time varies exponentially with the size of RSA prime values. Furthermore, we can also observe that cloud-analyzer authentication time is slightly higher than the proxy-cloud authentication time. We can explain the reason for such behavior as follows. When we establish the authentication, as we have explained in Chapter 3 we also share the symmetric keys shared between cloud and proxy (K_{DES_PC}) and cloud and analyzer (K_{DES_CA}). Furthermore, K_{DES_PC} is generated at the proxy before the initiation of authentication where as K_{DES_CA} is generated at the cloud server in between the authentication process. This causes the increase of analyzer-cloud authentication time with respect to proxy-cloud authentication time.

In the implemented system, we have kept RSA prime size for 512 bits since the maximum size of the authentication messages that are exchanged among the entities is approximately 504 bits (i.e. 320 + 184).

Table 4.2 illustrates the experimental authentication times that we have obtained for 10 computations by keeping RSA prime size to 512 bits.

Table 4.2: Mutual Authentication Times for RSA Prime Size of 512 bits

Authentication Session	Proxy-Cloud Authentication Time (ms)	Analyzer-Cloud Authentication Time (ms)
1	114	355
2	115	360
3	118	385
4	111	385
5	114	352
6	119	344
7	114	385
8	110	362
9	111	382
10	119	370

According to the Table 4.2;
Average cloud server-analyzer authentication time = **368 ms**
Average cloud server-proxy authentication time = **114.5 ms**

4.2.2 Variation of User Data Encryption Time

In this subsection, our focus is to analyze the encryption time variations of the extended ElGamal encryption scheme which we have used to encrypt the users' private sales data in order to induce the necessary homomorphic properties. As we have explained in the previous subsection, encryption time is mainly depending upon the size of the prime numbers (p, q and k) which are used to generate the required ElGamal keys. In Table 4.3, we have tabulated the results that we have obtained for encryption time while varying the ElGamal prime size whereas in Figure 4.3 we have graphically illustrated the results in Table 4.3.

Table 4.3: ElGamal Encryption Time Variation with Prime Size

ElGamal Prime Value Size (bits)	Encryption Time (ms)
16	12
32	17
64	20
128	22
256	34
384	49
512	70
768	110
1024	180

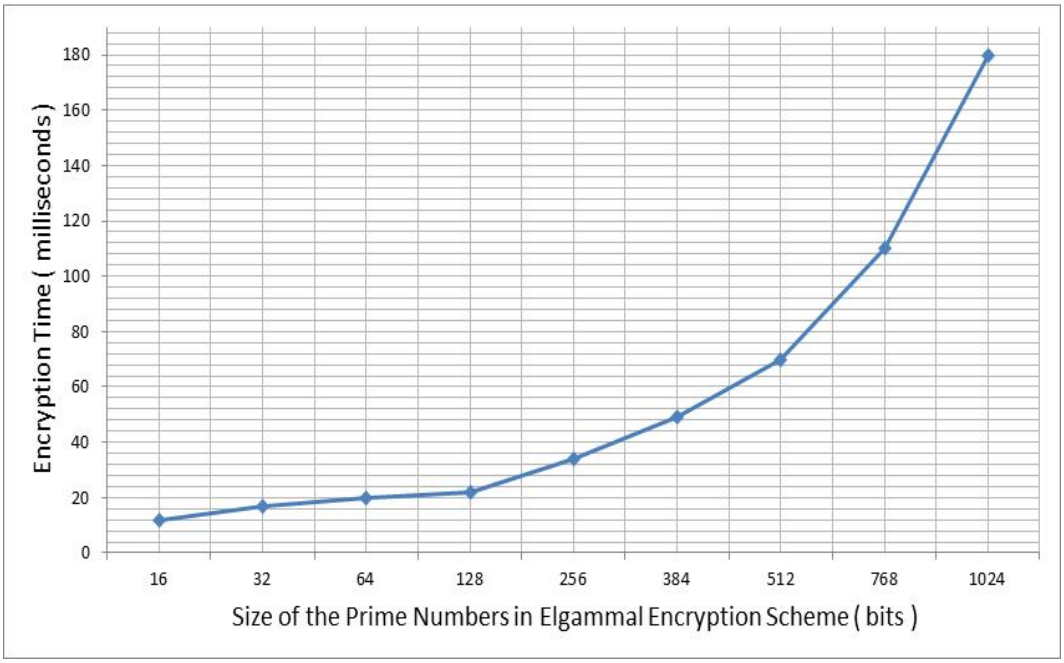

Figure 4.3: Variation of ElGamal Encryption Time as a function of Size of the ElGamal Prime Values

According to Figure 4.3, it is clear that the encryption time of extended ElGamal encryption scheme increases exponentially with the size of prime values. If we increase the size of primes it will help to strengthen the security of the cryptosystem while degrading the efficiency. Therefore, it is important to select an appropriate size for prime numbers considering the level of security, efficiency and the application. In our solution, we have kept the prime size as 64 bits.

Furthermore, prime size of 64 bits depicts that the values that can users encrypt will be limited to a maximum of $(2^{64} - 1)$ bits. However, in our application, since we are considering sales values of users as the message to be encrypted, the above mentioned range is quite adequate.

Then, it is important for us to consider how encryption time change with the size of the encrypting message while keeping the size of ElGamal prime numbers as 64 bits. Table 4.4 shown below shows the experimental data that we have gathered on encryption time while varying input message size from 4 bits to 64 bits. Furthermore, Figure 4.4 graphically represents the results in Table 4.4.

Table 4.4: ElGamal Encryption Time Variation for 64 bit Prime Size against Input Data Size

Input Message Size (bits)	Encryption Time (ms)
4	19
8	18
12	20
16	20
20	20
24	19
28	19
32	17
36	19
40	20
44	21
48	20
52	18
56	19
60	20
64	21

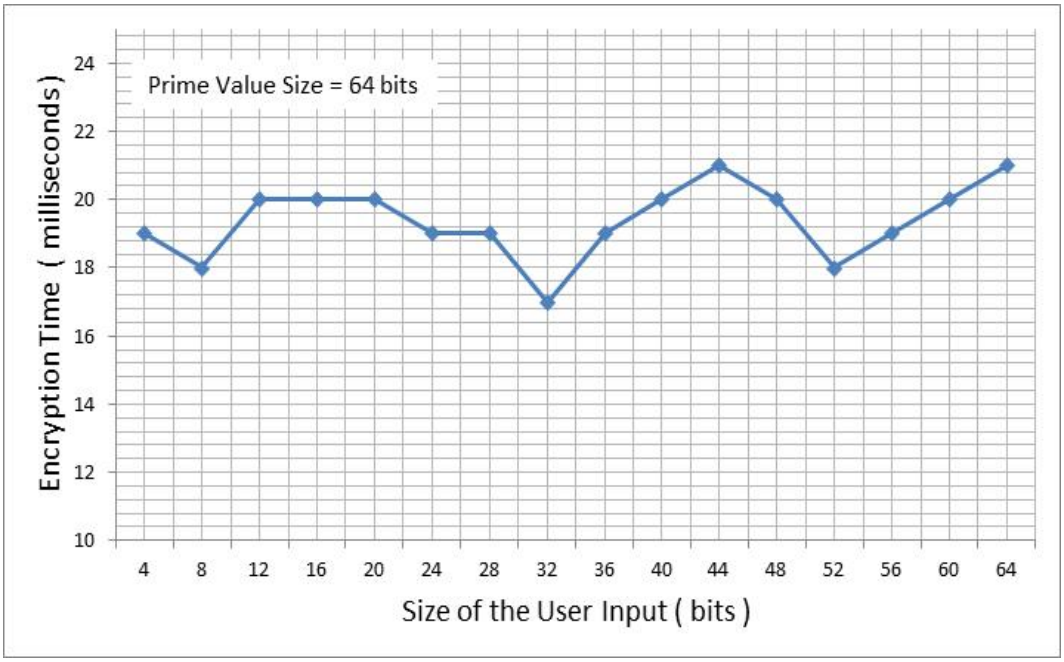

Figure 4.4: Variation of ElGamal Encryption Time for 64 bit Prime Values as a function of Input Data Size

According to Figure 4.4, we can see that the encryption time does not show any significant variation with input data size. Therefore, we can conclude that *input data size does not influence to the encryption time when the prime size is fixed.*

4.2.3 Comparison on Computational Time for Different Statistical Parameters

According to the case description presented under Section 3.1, main outcome of our proposed model is to compute statistical parameters of sales values within the organizational sales personnel. Therefore, it is important to identify the relationship between timing values for the computation of each statistical parameter that we have considered. We have tabulated the experimental results for computational time of each statistical parameter by varying the number of users in Table 4.5 while graphically representing them in Figure 4.5. These computational times are measured from the instance where computation is initiated at the cloud server to the instance where the answer is displayed in the interface of the analyzer program.

Table 4.5: Computational Times of Statistical Prameters

Number of Users	Mean Time (ms)	Variance Time (ms)	Standard Deviation Time (ms)	Skewness Time (ms)	Kurtosis Time (ms)
3	57	9	9	8	7
4	53	9	9	7	7
5	59	10	10	6	9
6	48	7	7	4	4
7	52	24	24	14	13
8	38	11	11	7	7
9	51	17	18	15	14
10	56	8	8	6	6
11	55	15	15	12	11
12	59	12	12	7	10
13	62	13	13	11	13
14	51	14	14	11	10
15	59	11	11	6	6
16	64	11	11	6	6
17	50	10	10	7	7
18	53	10	10	4	4
19	76	13	13	7	8
20	55	9	9	6	6
21	50	14	17	9	9
22	44	16	17	10	9
23	56	14	14	8	8
24	53	13	13	12	11
25	59	16	16	10	11
30	43	21	21	11	12
35	57	16	16	11	11
40	53	10	10	5	4
45	51	16	17	9	8
50	59	30	30	13	10

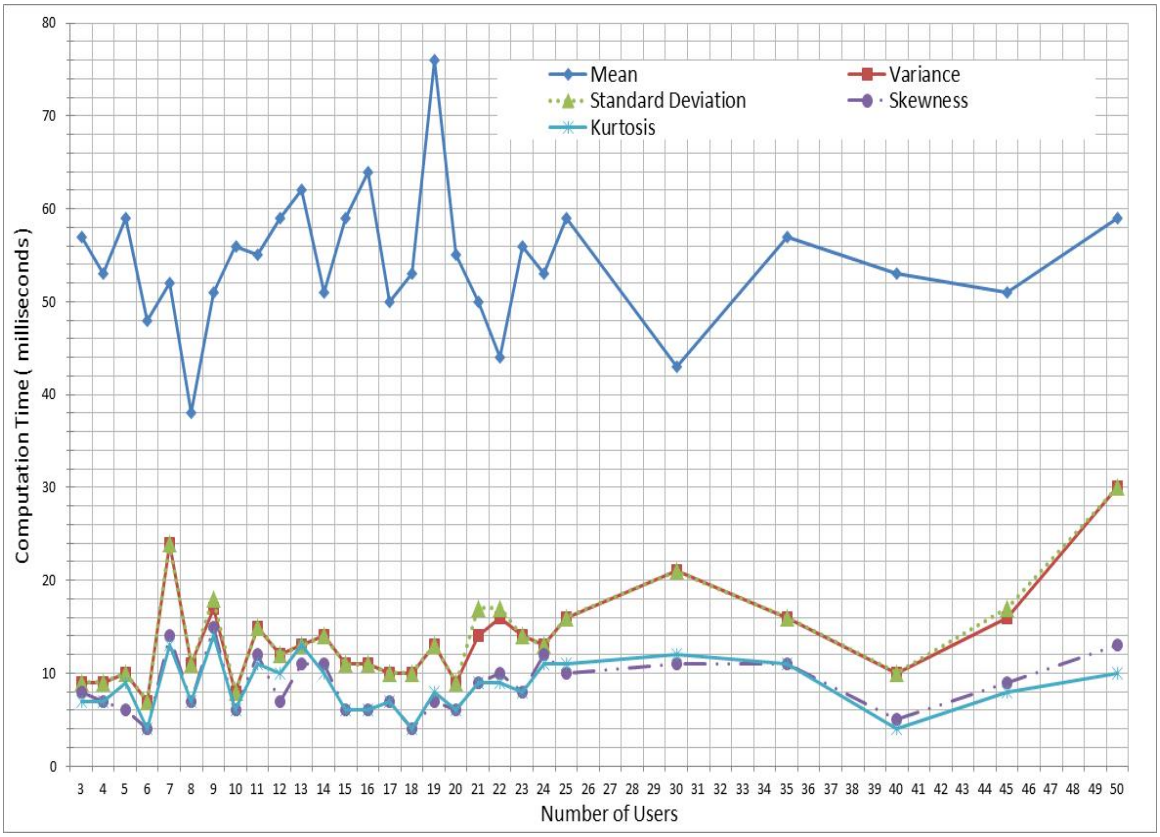

Figure 4.5: Variation of Computational Time for Statistical Parameters as a function of Number of Users

According to the graph, computational time of mean value exhibits a significant increment with regard to other computing times. In order to compute statistical data at cloud server, it is required to separate each users encrypted data component from the received message from the proxy server. Afterwards, those values should be stored in the program which requires a substantial amount of time. Hence, the mean value computation takes a higher time than the other computations purely because such a delay will not be present when computing other parameters since the users' data are already saved. However, it seems that variance and standard deviation requires higher computation time than skewness and kurtosis. The reason for this might be that, the complexity associated with the computation of skewness and kurtosis. These particular computations require division by 3^{rd} and 4^{th} power of the standard deviation, in contrast to computation of variance which only requires the division by number of users. Furthermore from the Figure 4.5, we can observe that computational time does not vary significantly with the number of users. Hence, it is conclusive that *the computational time of each statistical parameter is independent of the number of users associated in the system.*

4.2.4 Effect of Number of Users and Size of User Inserted Data on Process Time and Total Time

In this subsection we intend to find the relationship between the associated times which takes to handle encrypted data at all entities along with number of users and the size of user inserted data. First of all, it is important for us to define the two time components, entity process time and total process time for a computing session.

- **Entity Process Time:** Consider that all the users who are participating for the computation have already sent their encrypted sales data to the proxy server. Then, we define the entity process time as the complete time duration that encrypted data are handled at entities proxy server, cloud server and analyzer until the computation results are revealed at the analyzer.
- **Total Process Time:** We define the total process time as the combination of entity process time and the time it takes to encrypt private sales data of all the users. So, total process time corresponds to total time that users' data are handled at all the entities in our model from the initiation of a computing session till the end.

In order to find the effect of number of users on entity process time and total process time, we have varied the number of users from 3 to 50 while keeping all the other parameters (such as ElGamal prime size, message size, etc...) unchanged. The results that we have obtained are tabulated in Table 4.6. Furthermore in Figure 4.6 and Figure 4.7 we have illustrated the variation of entity process time and total process time on number of users respectively.

Table 4.6: Variation of Entity & Total Process Time with Number of Users

Number of Users	Entity Process Time (ms)	Total Process Time (ms)
3	165	225
4	166	250
5	160	270
6	147	261
7	171	297
8	141	309
9	167	347
10	157	357
11	158	389
12	166	418
13	167	414
14	144	452
15	159	444
16	173	493
17	166	506
18	145	505
19	196	557
20	166	546
21	162	603
22	158	598
23	171	631
24	163	643
25	179	679
30	149	749
35	176	876
40	180	1020
45	171	1026
50	194	1194

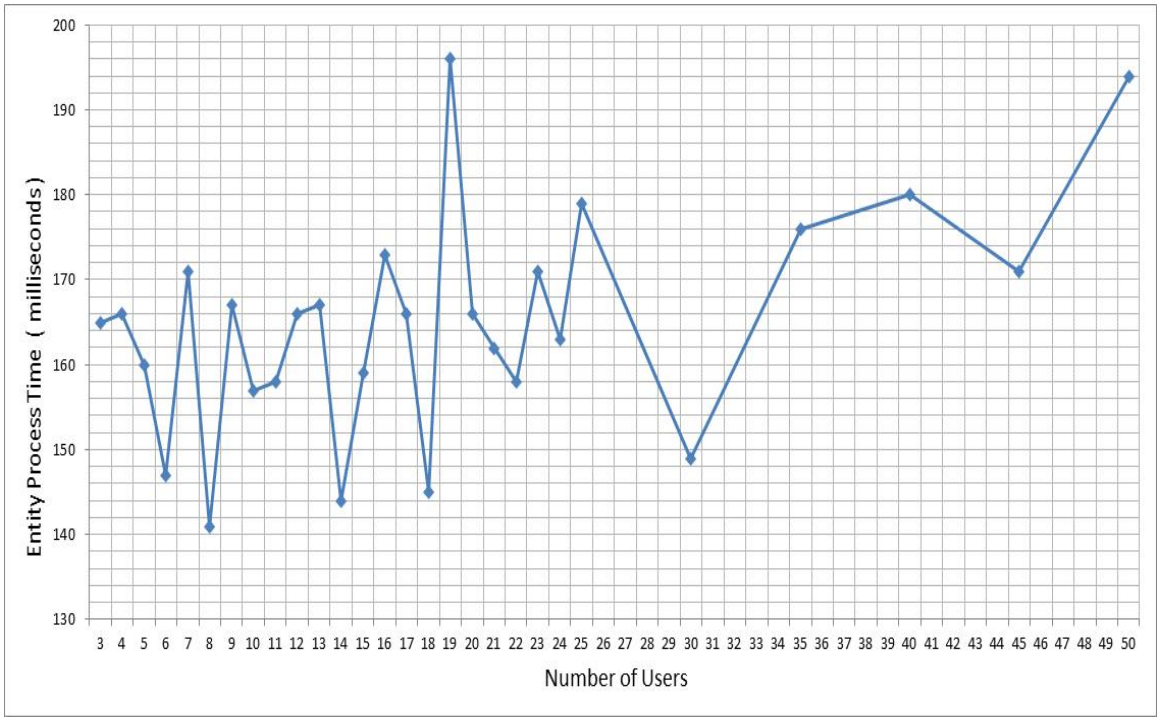

Figure 4.6: Variation of Entity Process Time as a function of Number of Users

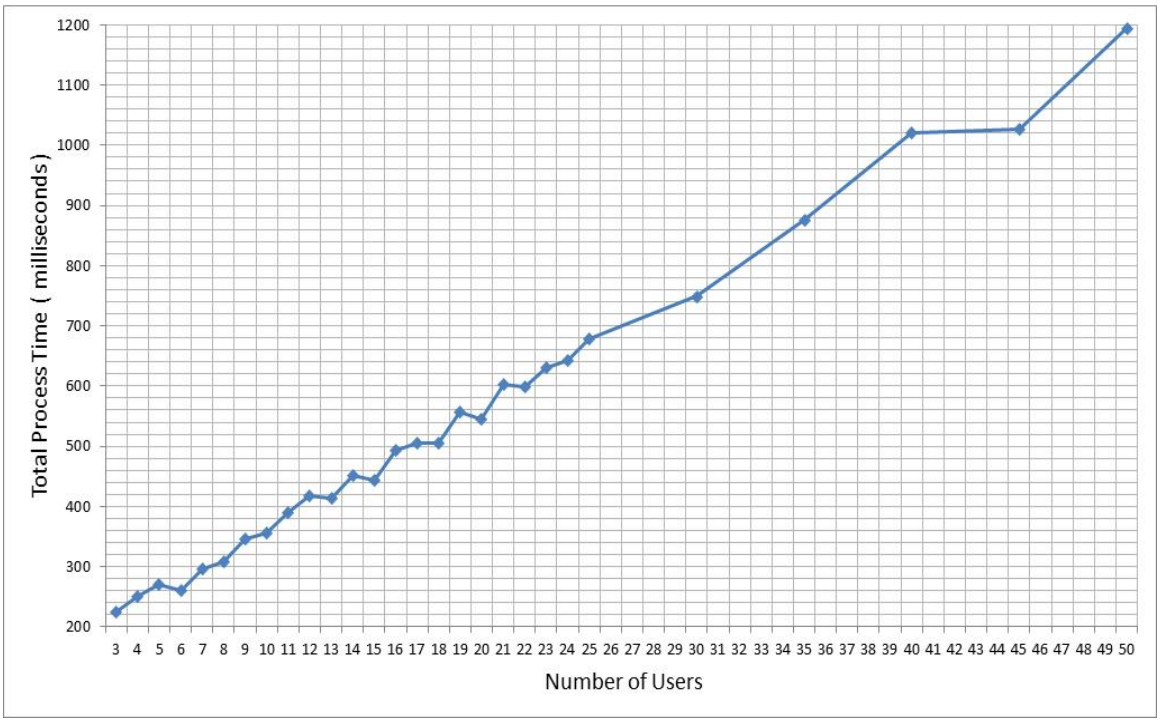

Figure 4.7: Variation of Total Process Time as a function of Number of Users

According to Figure 4.6, entity process time varies within the range of 141 ms to 196 ms with the increase of number of users in the system. Even though the introduction of more users to the system will increase the number of encrypted data segments received at the cloud, it will not cause any additional delay when processing the encrypted data at the cloud. This fact has been confirmed through the analysis given in Subsection 4.2.3. Therefore, *the entity process time is not greatly affected by the number of users who are taking part in the computation.* On the other hand, increased number of users will proportionally increase the total encryption time which causes the total process time to increase linearly with the number of users as shown in Figure 4.7.

In order to find the relationship between entity process time and total process time with the size of user input, we have acquired the results shown in Table 4.7 by varying input data size from 4 bits to 64 bits while keeping all the other parameters constant. It is important to note that we have kept the number of users to 10 and ElGamal prime size to 64 bits throughout the experiment which is the default setting for ElGamal primes. Furthermore, the results obtained are illustrated in Figure 4.8.

Table 4.7: Entity & Total Process Time Variation with Input Data Size

Input Message Size (bits)	Entity Process Time (ms)	Total Process Time (ms)
4	158	348
8	162	342
12	157	357
16	153	353
20	157	357
24	168	358
28	151	341
32	185	355
36	155	345
40	167	367
44	154	364
48	156	356
52	164	344
56	153	343
60	156	356
64	163	373

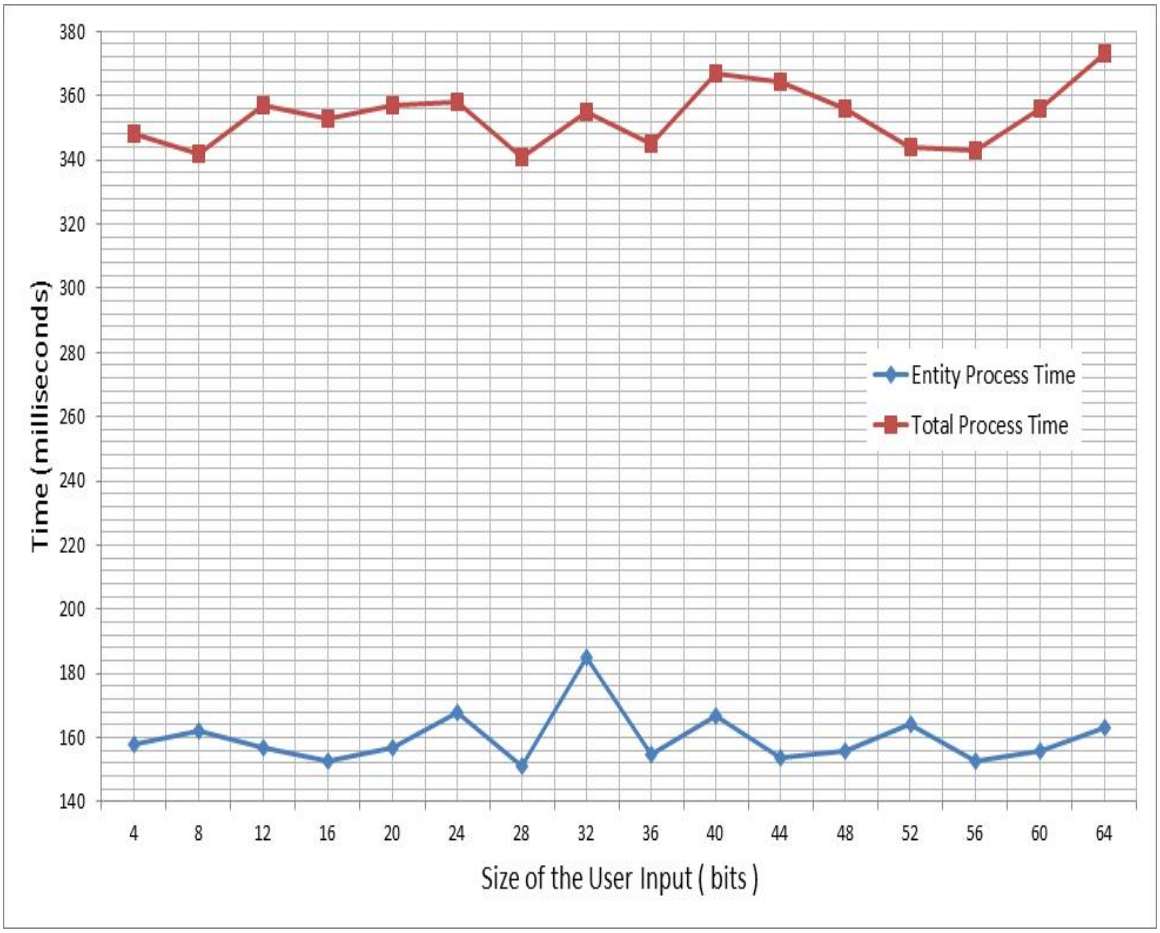

Figure 4.8: Variation of Entity Process Time & Total Process Time as a function of Input Data Size

Figure 4.8 depicts that both entity process time and total process time are independent of the input data size. Furthermore, total process time is shifted upwards approximately about 200 ms which accounts for the total encryption time of the 10 users with 64 bit ElGamal parameters. However, if the prime size is increased, then the encryption time will increase according to Figure 4.3 and the total process time would be shifted further upwards. Therefore, when the prime size is constant, we can conclude that *the variation of input data size does not have any effect on either of entity process time or total process time.*

4.2.5 Effect of Transmission Delay

In order to explain the effect of transmission delay in our system, we would like to first introduce a new parameter as total time which is defined below.

- **Total Time:** The time duration between the instance at which a computing session is initiated and the instance in which the statistical answers are displayed at the analyzer.

Therefore, we can write total time as;

$$total\ \ time = total\ \ process\ \ time + transmission\ \ delay \tag{4.1}$$

According to the communication process of our proposed model, transmission delay includes the following.

- Time required to send encrypted data of users to the proxy server (T_{UP}).
- Time required to send all encrypted user data from proxy server to the cloud server (T_{PC}).
- Time required by the cloud sever to send encrypted summation message to the analyzer (T_{CA1}).
- Time required to send encrypted mean value to cloud server by the analyzer (T_{AC}).
- Time required to finally send the encrypted answers of other statistical parameters to the analyzer (T_{CA2}).

That is;

$$transmission\ \ delay = T_{UP} + T_{PC} + T_{CA1} + T_{AC} + T_{CA2} \tag{4.2}$$

Therefore;

$$total\ \ time = total\ \ process\ \ time + (T_{UP} + T_{PC} + T_{CA1} + T_{AC} + T_{CA2}) \tag{4.3}$$

In order to find the variation of transmission delay in our system with number of users who are taking part in a computation, consider the tabulated results in Table 4.8 which shows the variation of total time and total process time with number of users. Furthermore, we have graphically illustrated the variation of total time and total process time in Figure 4.9.

Table 4.8: Variation of Total Process Time & Total Time with Number of Users

Number of Users	Total Process Time (ms)	Total Time (ms)
3	225	228
4	250	254
5	270	276
6	261	267
7	297	303
8	309	313
9	347	352
10	357	360
11	389	392
12	418	419
13	414	422
14	452	455
15	441	451
16	493	498
17	506	513
18	505	512
19	557	562
20	546	557
21	603	609
22	598	607
23	631	633
24	643	645
25	679	689
30	749	756
35	876	884
40	1020	1026
45	1026	1033
50	1194	1206

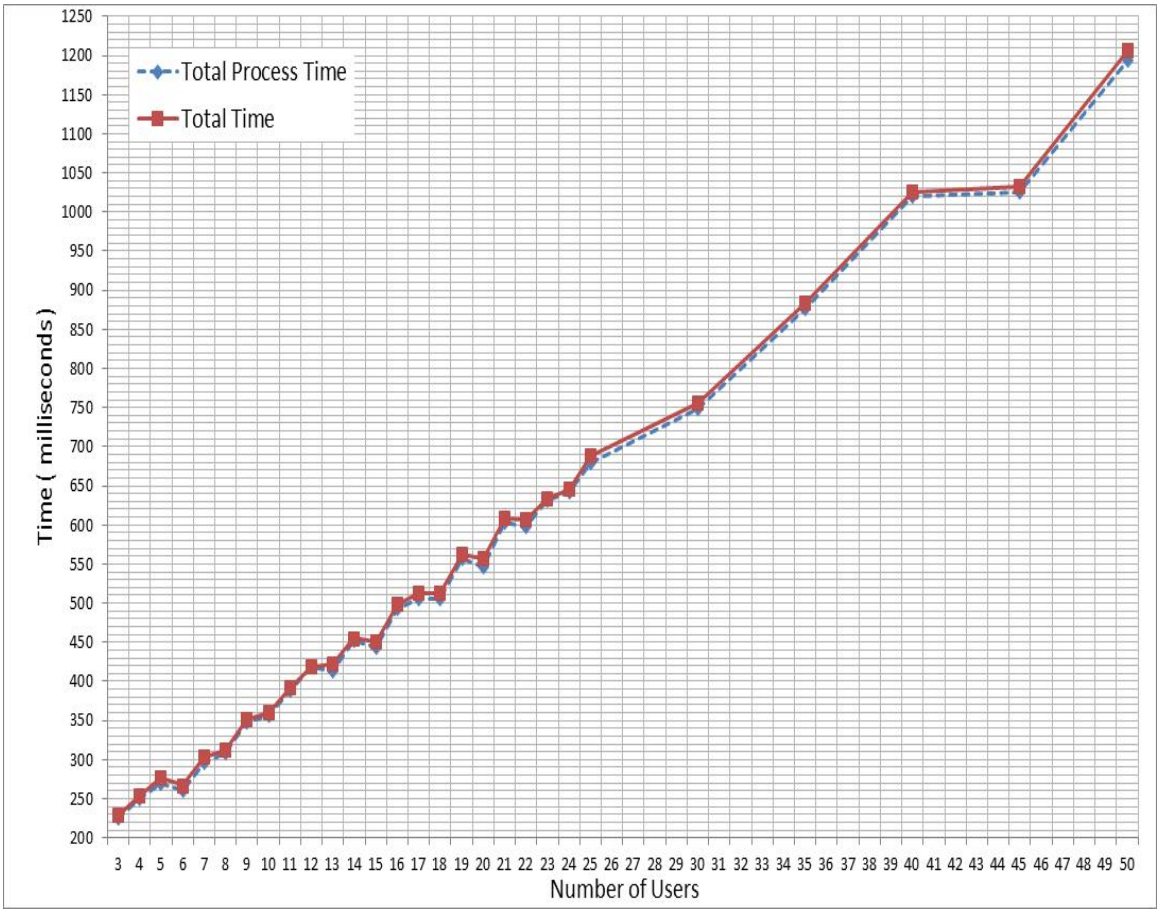

Figure 4.9: Variation of Total Process Time & Total Time as a function of Number of Users

According to Equation 4.1, the difference between total time and total process time represents the associated transmission delay in the system. Thus, Figure 4.9 shows that transmission delay is independent of the number of users. Furthermore, our experimental setup possesses a small transmission delay about 4 ms, which is mainly due to the fact that we have created the setup inside the University of Agder network. However, according to the metrics such as bandwidth, hop count, etc. of the links between the entities would result in different transmission delays.

In order to determine the behavior of total time related to the distance between the entities, we have placed a server in Sri Lanka to run the cloud server program. The Hop Count (HC) between the proxy server and the cloud server is 17 hops. Figure 4.10 shows the variation of total process time and total time with the number of users at instances when HC is 1 and 17 respectively.

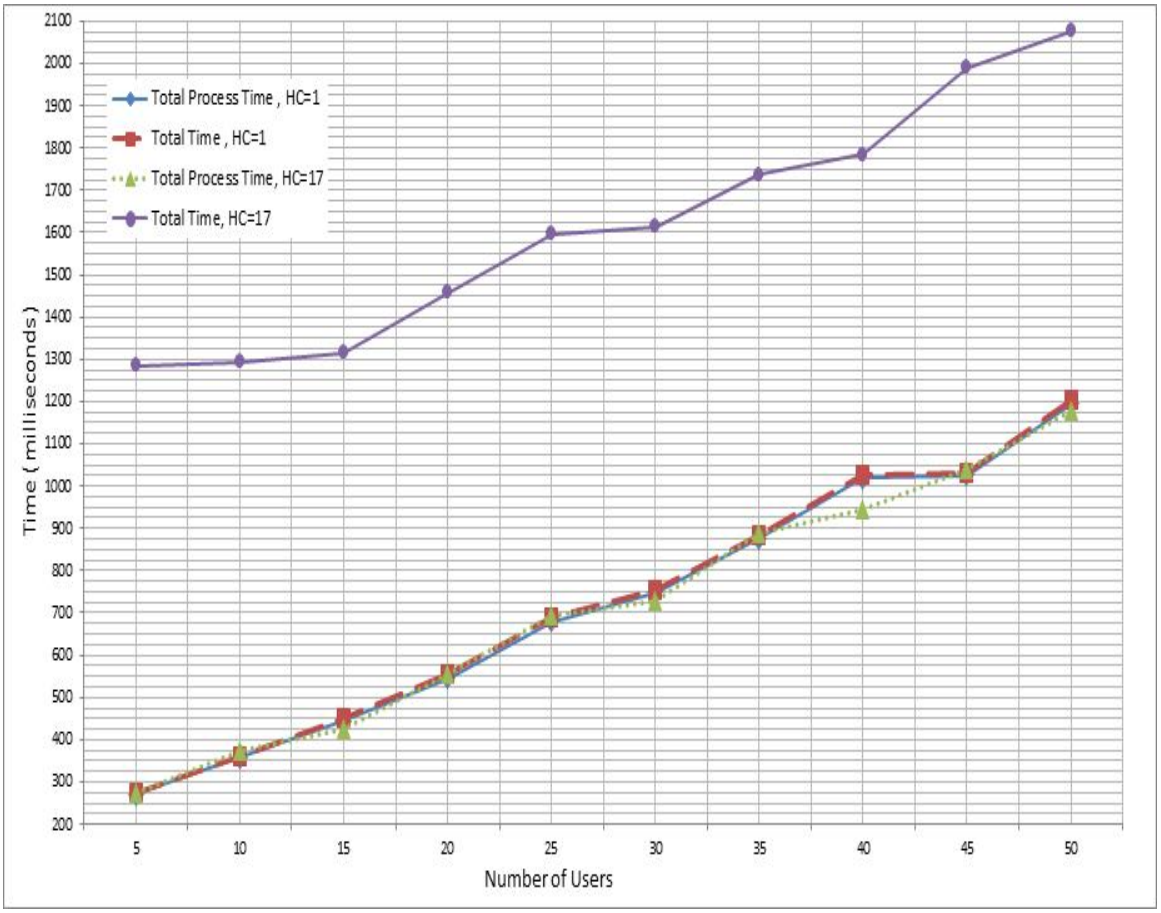

Figure 4.10: Variation of Total Process Time & Total Time as a function of Number of Users when HC =1 & HC = 17

According to Figure 4.10, behavior of the total process time when HC is 1 and 17 are both identical and their values lie in a closer range. This fact proves that total process time is independent of the distance between the entities. On the other hand, total time when HC is 17 exhibits a significant escalation compared to other timing values. The reason for this is the increment in transmission delay associated with the increased hop count. Therefore, *total time is dependent upon the distance between the entities.*

According to our analysis it is obvious that either the number of users or input data size (with constant ElGamal prime size of 64 bits) does not impose a significant impact on the entity process time of the implementation.

Hence, we can estimate average entity process time of the system from Table 4.6 and Table 4.7 as;

Average entity process time of the proposed system = **162.53 ms**

Furthermore, when we consider the total process time, we have to consider encryption time in addition to the entity process time. We have shown that the encryption time is largely affected by the bit size of ElGamal encryption parameters. Since, we have used 64 bit ElGamal parameters for the implementation we can estimate the average total process time as;

$$\text{Average total process time of the proposed system} = \text{average entity process time} + (\text{average encryption time}) \times n$$

According to Table 4.4, average encryption time with 64 bit ElGamal parameters is approximately 20 ms. Therefore;

$$\text{Average total process time of the proposed system} = \mathbf{(162.53 + 20 \times n)\ ms,}$$

where, n denotes the number of users

It is important to note that, *both entity process time and total process time are independent of the network parameters and only be depending upon the specifications of the PCs and server used in the experimental setup.*

Chapter 5

Discussion

This chapter is dedicated to discuss some important aspects about our proposed solution. We have started the chapter by discussing security attributes of existing Multi-party Cloud Computing solutions and thereafter explicating how our solution can withstand the necessary security requirements. Then we have discussed the performance characteristics of our implemented model while concluding the chapter by illustrating the associated limitations.

The literature analysis that we have put forward in Section 1.5 suggests that, even though there have been a variety of research work carried out in areas such as cloud security, multi-party computations and cloud based computations, a few of them actually looked at developing SMCs on cloud environments. When considering such efforts, we came across a solution proposed by N. Maheshwari et.al [3]; where they provided a framework for cloud computing using SMC which exhibits conceptual resemblance to our work. As we have discussed in Subsection 1.5.4 their solution capable of achieving identity anonymization and data availability. They have mentioned the idea of handling encrypted data in clouds (deep cloud confidentiality) to provide data privacy, but they have not specified a method to achieve the requirement by considering practical scenarios. Furthermore, they have not considered the possible security threats when exchanging information among users and the cloud, due to the insecurity of public networks.

In this thesis, we have proposed our Secure Multi-party based Cloud Computing Framework considering a practical scenario of outsourcing statistical computations of user inserted sales data which we have illustrated in Section 3.1.

Furthermore, our solution is capable of ensuring security, privacy and anonymity of user inserted data as well as withstanding possible replay attacks and integrity violations. Moreover, in the following section, we have broadly analyzed how our solution can cater the security requirements that we have mentioned above.

5.1 Security Analysis of the Proposed Solution

In the proposed solution, we have included a proxy server in between the user parties and the cloud server. As we have explained in Section 3.6, it authenticates each user party and creates a new data frame including private data from all the users in order to be transmitted to the cloud server. Therefore, the cloud server does not get any information regarding which data component is actually belongs to which user party. Hence, user data anonymity is guaranteed with the introduction of the proxy server.

On the other hand, all the data sent from each user party for computation is encrypted using extended version of ElGamal cryptosystem [22] and the cloud server also carry out the required statistical computations on the encrypted data. So, users do not want to worry that their private data regarding sales will be mislead at the cloud server or else will be revealed by other competitive users. This suggests that the privacy of user data is also established through the proposed solution.

When we consider the architecture of the proposed security framework which we have illustrated in Figure 3.2, the proxy server and user parties belong to the same organizational entity where as the cloud server and the analyzer are external entities. So, we can assume that data communication between user parties and proxy server is secured, since we are referring to transmission of data within an organization. However, when the data is released from the proxy server to the cloud, the communication process is not at all secured and data can be tampered by numerous ways. Therefore, it is clear that first we should make sure that the proxy is actually communicating with the cloud server before sending user data. We have achieved this by establishing a mutual authentication between the considered parties through strong RSA cryptosystem. Furthermore, we have also shared a 3DES key when establishing the mutual authentication, which we have used to encrypt the data frame when sending users' private data to the cloud server for computations. Since data frame is encrypted with such shared key, there is no possibility to decrypt the frame by any other external party except for proxy and cloud server.

When we send data from proxy to the cloud server, first we generate a data frame at the proxy that includes users' data, HMAC of users' data and the current timestamp which is ultimately be encrypted by the shared 3DES key. The introduction of HMAC allows us to detect any tampering of data on the way to the cloud server. This makes it possible, since tampering will change the bit pattern of the message and when HMAC is recalculated at the cloud server with the received message, it will be different from the HMAC that is attached to the received message. Moreover, the timestamp attached to the messages sent from proxy server allows to identify the received data frame is a replayed frame or a fresh one. The frame will be considered a fresh one, if the timestamp of the received message is within the defined clock skew of 2000 ms and otherwise it will be considered as a replayed one and it will be discarded while asking proxy server to resend the data frame again. It is also important to note that we have taken similar security measures for the communication process between cloud server and the analyzer as we have explained for proxy and cloud server.

The security of a cryptographic scheme mainly depends upon the security of associated keys. Hence, it is important to generate the relevant keys using large prime numbers as well as ensuring the secrecy of private and symmetric keys. Thus, in our solution we have used 64 bit prime values to generate ElGamal keys and 512 bit prime values to develop RSA keys. There is always a possibility of leaking keys when using the same set of keys over long periods of time. In order to overcome this, we flush the existing keys and re-generate a new set of keys after every consecutive 10 computational sessions.

We can point out that the associated security of proxy server as the major concern for the proper functionality of the whole framework. It is necessary for us to share the ElGamal parameter k between the analyzer and users which is required for the users to encrypt user data whereas analyzer needs it to encrypt the mean value. Because of that, we generate the value k at the analyzer end and forward to the proxy server by signing and encrypting by analyzer's RSA private key and proxy server's public key as discussed in Subsection 3.4.2. Since, the ElGamal parameter k is a security critical one, compromise of the proxy server will expose k which may ultimately affect the privacy of user data. However, the proxy server is considered to be a well secured entity purely because it lies inside the organization and protected through security mechanisms such as firewalls and Intrusion Detection Systems (IDS).

5.2 Analysis on Experimental Results

As explained in the previous chapter of this report, we have considered several test cases for evaluating the performance of our proposed framework. Such test cases measure the timing values for parameters such as authentication time, encryption time, entity process time, total process time and total time. According to the results we have obtained regarding authentication time, it varies exponentially with the size of the prime values used in the RSA algorithm. In order to obtain these results, we have kept all other factors constant. The distance between the servers and the analyzer might be another factor which could contribute as a dependent variable for authentication time along with system specifications of the PCs and server. Increasing the distance between the entities would ultimately increase the transmission delay of the authentication process. Hence the overall timing values would be raised.

User data encryption time also exhibits a similar behavior as authentication time when varying the size of the prime values in ElGamal encryption algorithm. Furthermore, our results indicate that size of the user input does not affect the encryption time when ElGamal prime values are unchanged. Therefore, encryption time of a certain user for a particular computational session would be independent and falls approximately in the range of 18 ms to 23 ms. This fact suggests that encryption time causes the total process time to linearly increase with the increasing number of users participating in a computational session. On the other hand, entity process time is independent of the number of users and size of the user input respectively. Therefore, entity process time is only dependent on the system specifications of the entities in the framework. Moreover, the total process time also behaves similarly, because both encryption time and entity process time are depending only upon the system specifications of the entities. These facts suggests that the average values we obtained for entity process time and total process time are universally applicable for a system with entities of similar specifications that we have mentioned in Section 4.1. However, total time is a parameter which is dependent upon the transmission delay according to the Equations 4.1 and 4.2. Therefore, the value of total time may vary depending on the positioning of the entities.

5.3 Limitations of Proposed Solution

The proposed model possesses certain limitations. Mainly, the framework which we have introduced would only be fully compatible for the application that we have stated in case description under Section 3.1. Furthermore, it might not be universally applicable for all the secure multi-party based cloud computing applications.

The extended ElGamal algorithm that we have used to encrypt user inserted data, exhibits additive and multiplicative homomorphism properties. In our system, we have developed addition, subtraction, 2^{nd} power, 3^{rd} power and 4^{th} power operations on encrypted data exploiting above mentioned properties in the algorithm. These operations are adequate to compute the statistical parameters that we have mentioned in the Subsection 3.7.2. However, complex computations require more operations to be performed on encrypted data. Since the algorithm which we have used is not fully homomorphic, complex computations cannot be performed due to the limitations of operations. Furthermore, division and square root operations cannot be performed on encrypted data since extended ElGamal algorithm is not homomorphic for such operations. Because of that, we do all the required divisions and square root operations at the analyzer end in our implementation. Furthermore, considering that the timing parameters are varying in the range of milliseconds, it is important that all the entities of the framework are properly synchronized to operate efficiently.

Chapter 6

Conclusions

In this chapter, we conclude the thesis by providing a summarization on the problem that we have addressed, main results and our contribution as well as introducing new research directions that have invoked as a result of our analysis.

The main problem associated with most of the existing multi-party computing approaches is that they incur a lot of communication overhead which ultimately affects the efficiency of the whole system. However, the use of cloud based solutions severely suffers from security and privacy issues. Therefore, our main objective of this thesis was to propose and implement a *Secure Multi-party based Cloud Computing Framework* which can ensure security, privacy and anonymity of users' private data. In order to fulfill our main objective, we considered a case where an organization requires outsourcing statistical computations of their sales data to a CSP, while ensuring privacy and security of users' private sales data.

In the proposed solution, we have used three main entities as proxy server, cloud server and an analyzer to solve the problem that we have considered. The main functionality of the proxy server is to hide identities of each user party from the cloud server's perspective to ensure identity anonymization. The cloud server carries out the required statistical computations on the encrypted sales data of users while the analyzer is the party which receives the encrypted statistical answers from the cloud server. Furthermore, analyzer is capable of decrypting the received encrypted answers to reveal statistical values (mean, variance, standard deviation, skewness and kurtosis) of users' sales data which are used to make organization's sales related decisions. Since, the cloud server is carrying out the computations on the encrypted data; our framework ensures privacy of user data.

Furthermore, we have established mutual authentication between entities at the beginning to ensure data availability of the system. Moreover, we have made the communication among entities secure by using the concepts of timestamps and HMAC while sending the data in an encrypted environment. Hence, the proposed Secure Multi-party Cloud Computing Framework ensures data security, privacy and anonymity of user inserted data. This suggests that we have fulfilled the requirements of Research Objective **RO 1** in Subsection 1.4.1.

In order to show the practicability of the framework, it is quite necessary to evaluate the performance. Therefore, we have implemented a prototype of the cloud computing framework and analyzed its' performance through performance parameters of authentication time, encryption time, entity process time, total process time and total time. Furthermore, we have illustrated how those parameters behave when changing system constraints such as number of users, size of the input and the size of prime values that are used to generate respective encryption keys. Hence, this shows that we have achieved the Research Objectives **RO 2** and **RO 3**.

The findings of this thesis provide deep insights on practicability of Multi-party based Cloud Computing Frameworks. Furthermore, from the security analysis of our framework and numerical results, we conclude that cloud environments can be successfully deployed to improve the efficiency of multi-party computations while ensuring the security requirements of user parties.

Furthermore, it is important to highlight the contributions that we have made through fulfilling the thesis objectives and several interesting research areas that are emerged.

6.1 Contribution to Knowledge

Our solution suggests a novel way to carry out SMCs using cloud environments in an efficient manner while ensuring security requirements of users such as data security, privacy and anonymity of users' private data. According to our summarization of related work in Chapter 1, it is clear that none of them actually succeeded in proposing a fully functioning secure multi-party cloud computing framework, though there have been lot of researches on the related areas such as cloud based computations, SMCs and cloud security. Therefore, our endeavor of proposing a secure multi-party based cloud computing framework would be a new knowledge for ICT research community.

Furthermore, the performance measurements that we have obtained through the implementation of our secure multi-party based cloud computing framework would be helpful for researchers to evaluate the performance of similar solutions as well as build on it to develop more secure and efficient solutions in future. Therefore, these facts will contribute to scientific community for enhancing cloud based security frameworks.

6.2 Future Work

We have used the extended ElGamal encryption scheme [22] to encrypt users' private data to induce homomorphic properties to the user data; in order to allow the required statistical computations to be carried out on the encrypted data. However, as we have mentioned in limitations of our framework in Section 5.3, we pointed out that we could not carry out divisions and square root operations at the cloud server since the extended ElGamal encryption scheme does not support homomorphism on those operations. Therefore, it is an interesting topic to be considered in the future, that how we can incorporate a method which allows such operations to be carried out on encrypted data. Then, our solution can be made more efficient, since we would be able to compute all the statistical parameters at once at the cloud server.

The privacy of user data in our framework greatly depends upon the security of the extended ElGamal encryption scheme. Furthermore, we have shown that the parameter k is a critical component in terms of security and leaking of parameter k may lead to privacy violations. Hence, it is quiet important to look at ways to make the cryptosystem more secure, which makes our approach more practicable.

Bibliography

[1] F. Shaikh, and S. Haider, "Security Threats in Cloud Computing," in *Proceedings of International Conference for Internet Technology and Secured Transactions (ICITST), IEEE*, Abu Dhabi, UAE, Dec. 2011.

[2] A. C. Yao, "Protocols for Secure Computations," *Annual Symposium on Foundations of Computer Science, IEEE*, vol. 0, pp. 160–164, Nov. 1982.

[3] N. Maheshwari, and K. Kiyawat, "Structural Framing of Protocol for Secure Multiparty Cloud Computation," in *Proceedings of* 5^{th} *Asia Modelling Symposium (AMS), IEEE*, Kuala Lampur, Malaysia, May 2011.

[4] R. Oppliger, "Contemporary Cryptography," *Artech House Computer Security Library, Norwood*, 2005.

[5] Q. Ma, L. Xiao, I.-L. Yen, M. Tu, and F. Bastani, "An Adaptive Multiparty Protocol for Secure Data Protection," in *Proceedings of* 11^{th} *International Conference on Parallel and Distributed Systems*, Fukuoka, Japan, Jul. 2005.

[6] L. Sumter, "Cloud computing: security risk," in *Proceedings of the* 48^{th} *Annual Southeast Regional Conference, ACM*, New York, USA, Apr. 2010.

[7] S. Chakraborty, S. Sehgal, and A. Pal, "Privacy Preserving E-negotiation Protocols based on Secure Multi-party Computation," in *Proceedings of SoutheastCon., IEEE*, Fort Lauderdale, USA, Apr. 2005.

[8] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson, "Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds," in *Proceedings of the ACM Workshop on Cloud Computing Security Workshop (CCSW), ACM*, New York, USA, Oct. 2010.

[9] J. Feng, Y. Chen, D. Summerville, W.-S. Ku, and Z. Su, "Enhancing Cloud Storage Security against Roll-back Attacks with a new fair Multi-party Non-repudiation Protocol," in *Proceedings of Consumer Communications and Networking Conference (CCNC), IEEE*, Las Vegas, USA, Jan. 2011.

[10] S. Pearson, Y. Shen, and M. Mowbray, "A Privacy Manager for Cloud Computing," in *Proceedings of the* 1^{st} *International Conference on Cloud Computing, Springer-Verlag*, Berlin, Germany, 2009.

[11] M. Mowbray, and S. Pearson, "A Client-based Privacy Manager for Cloud Computing," in *Proceedings of the* 4^{th} *International ICST Conference on Communication System Software and Middleware, ACM*, New York, USA, Jun. 2009.

[12] D. Mishra, and M. Chandwani, "Anonymity Enabled Secure Multi-party Computation for Indian BPO," in *Proceedings of Region 10 Conference - TENCON, IEEE*, Taipei, Republic of China, Nov. 2007.

[13] R. Mortier, A. Madhavapeddy, T. Hong, D. Murray, and M. Schwarzkopf, "Using Dust Clouds to Enhance Anonymous Communication," UK.

[14] M. Tebaa, S. El Hajji, and A. El Ghazi, "Homomorphic Encryption Method applied to Cloud Computing," in *Proceedings of National Days of Network Security and Systems (JNS2), IEEE*, Marrakech, Morocco, Apr. 2012.

[15] A.-F. Chan, "Symmetric-key Homomorphic Encryption for Encrypted Data Processing," in *IEEE International Conference on Communications (ICC), IEEE*, Dresden, Germany, Jun. 2009.

[16] B. K. Samanthula, G. Howser, Y. Elmehdwi, and S. Madria, "An efficient and Secure Data Sharing Framework using Homomorphic Encryption in the Cloud," in *Proceedings of the* 1^{st} *International Workshop on Cloud Intelligence, ACM*, New York, USA, Aug. 2012.

[17] C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient Aggregation of Encrypted Data in Wireless Sensor Networks," in *Proceedings of the* 2^{nd} *Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous), IEEE*, San Diego, USA, Jul. 2005.

[18] J. Wei, S. Guo, and Q. Xu, "Secure Homomorphic Aggregation Algorithm of Mixed Operations in Wireless Sensor Networks," in *Proceedings of International Conference on E-Business and Information System Security (EBISS), IEEE*, Wuhan, China, May 2009.

[19] W. Luo, and X. Li, "A Study of Secure Multi-party Statistical Analysis," in *Proceedings of International Conference on Computer Networks and Mobile Computing (ICCNMC), IEEE*, Shanghai, China, Oct. 2003.

[20] D. Mishra, R. Pathak, S. Joshi, and A. Ludhiyani, "Secure Multi-Party Computation for Statistical Computations using Virtual Parties on a Token Ring Network," in *Proceedings of* 7^{th} *International Conference On Wireless And Optical Communications Networks (WOCN), IEEE*, Colombo, Sri Lanka, Sep. 2010.

[21] A. Bouti and J. Keller, "Securing Cloud-based Computations against Malicious Providers," *SIGOPS Oper. Syst. Rev., ACM*, vol. 46, no. 2, pp. 38–42, July 2012.

[22] G. Xiang, B. Yu, and P. Zhu, "A Algorithm of Fully Homomorphic Encryption," in *Proceedings of* 9^{th} *International Conference on Fuzzy Systems and Knowledge Discovery (FSKD), IEEE*, Sichuan, China, May 2012.

[23] M. Stamp, *Information Security Principles and Practice*. Wiley, 2006.

Appendix A

Attached Publication

Title:	Secure Multi-party based Cloud Computing Framework for Statistical Data Analysis of Encrypted Data
Affiliation:	University of Agder, Faculty of Engineering and Science, P.O. Box 509, 4898 Grimstad, Norway
Submission status:	To be submitted to 14^{th} *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid 2014).*

Secure Multi-party based Cloud Computing Framework for Statistical Data Analysis of Encrypted Data

G. P. H. Sandaruwan, P. S. Ranaweera
Dept. of Information and Communication Technology, University of Agder (UiA), N-4898 Grimstad, Norway
Email: hsg11@student.uia.no, psr11@student.uia.no

***Abstract*—Secure multi-party computation (SMC) is a paradigm used to accomplish a common computation among multiple users while keeping the data of each party secret from others. Cloud computing is a next generation computing solution which allows its users to use high speed infrastructure and services provided by cloud service providers (CSP) in a cost effective manner. Therefore, deployment of cloud based architecture for SMCs would aid in improving its performance and efficiency. However, cloud based solutions raises concerns over security of users' private data, since data is under the control of an external entity when outsourced to cloud platforms. In this paper we have proposed a Secure Multi-party based Cloud Computing Framework which can ensure security, privacy and anonymity of users' private data. This framework is modeled by considering a scenario which requires outsourcing statistical parameter computation of private sales data of an organization's sales personnel. The results that we have obtained, provides significant evidence for the practicability of multi-party based cloud computing solutions.**

Keywords: ***secure multi-party computation, cloud computing, data security, privacy, anonymity***

I. Introduction

In general terms, a SMC can be defined as a situation where n parties who are having private inputs x_1, x_2 ..., x_n interested in computing the value of the public function $f(x_1, x_2..., x_n)$ in such a way that at the end of the computation no party is revealed any of the private inputs of other parties [1]. The concept of SMC was first introduced by Yao in 1982 through the "millionaire problem" [2] and since then SMCs have being deployed in a variety of applications such as voting systems, auctions, business related private computations and privacy-preserving data mining, etc. Theoretically, a SMC is represented with the existence of a trusted third party (TTP) which does the required user intended multi-party computation. However, this is practically infeasible due to the fact that an external entity cannot be trusted to hand over the private data of users. Therefore, SMC is all about finding appropriate cryptographic protocols that can replace the use of a TTP, to carry out a certain user intended function while ensuring data privacy of users [3].

The term cloud is analogical to the Internet. Hence, cloud computing can be visualized as computing over Internet. More precisely, it is a set of resources and facilities offered to its users economically via the Internet [4]. A cloud makes it possible for its users to access their information in the cloud from anywhere, anytime through the Internet. On the other hand users do not need to worry about the maintenance and availability of resources, due to the fact that it is the responsibility of the CSP. More importantly cloud computing is an on demand service, where users are charged only based on their resource consumption. Because of such benefits, cloud computing has become more and more popular among business entities.

The main issue with most of the traditional SMC protocols is that they incur a significant amount of communication overhead affecting the efficiency of the protocol [5]. As a solution, it is possible to outsource the computations to a CSP which would help to reduce the expenditure as well as the operational overhead. However, the difficulty that we face is how we can enforce the security requirements of a multi-party computation such as data security, privacy and anonymity; when we are dealing with an un-trusted external entity. This issue can be addressed through secure multi-party cloud computing solutions.

The rest of this paper is organized as follows. Related work is explained in Sec. II and then the case that we have formulated is introduced in Sec. III whereas Sec. IV describes the proposed solution. The performance evaluation of the proposed framework is given in Sec. V before the paper is concluded in Sec. VI.

II. Related Work

The requirement for the distributed computing systems emphasized with the advancement of Information and Communication Technology (ICT) has led to the introduction of cloud computing concepts which provides a high speed infrastructure for the users with low maintenance and high availability. Though, users are mainly concerned about the confidentiality and integrity of data in the cloud servers [6]. Therefore, the adoption of cloud computing techniques has been greatly inhibited due to the issues of data security, privacy and anonymity associated with them [7]. The researchers are indicating different approaches to overcome the drawbacks in cloud computing. A new security protocol has been introduced in [5], which assures data protection while ensuring better

performance under normal circumstances. Reducing the usage of sensitive information is another scenario which has been considered to avoid misusing and stealing of user data [8][9]. It is also important to keep the identity of the user anonymous [10].

Issues with data confidentiality of cloud users have tempted the requirement for encrypting the user data before sending it to cloud. As a solution homomorphic encryption schemes have been introduced. Homomorphic encryption schemes such as Paillier and RSA systems could perform operations like addition and multiplication on encrypted data [11]. Furthermore, two additive symmetric key homomorphic schemes called as iterated hill cipher (IHC) and modified Rivest scheme (MRS) has been suggested in [12]. Researchers have considered deploying such encryption schemes for applications concerning computations. Wireless sensor network (WSN) based applications exemplifies the requirement of encrypted data based computations due to the fact that such computations are not feasible within the sensor [13]. Method introduced in [13] aggregates the sensor data and forwards it to an entity with high computational power. Similar method with public key based scheme to ensure user privacy has been presented by [14]. Paper [15] has presented a method to carry out combination of additive and multiplicative operations on encrypted data through Paillier and RSA cryptosystems.

Though there has been a significant improvement in the areas of cloud security and encrypted data processing, multi-party based cloud computing solutions has not yet being evolved. Such solution is proposed in [1] where they have introduced a secure multi-party cloud computing framework (SMCC) which allows multiple users to perform any common computation of their interest in the cloud. Even though the SMCC method and security protocols introduced in [1] ensures data security and identity anonymization of users, it has not been validated against practical scenarios.

III. Case Description

The formulated case involves sales management in a particular organization called 'ABC'. The administrators of the organization are interested in analyzing mean, variance, standard deviation, skewness and kurtosis of the daily sales values of their employees for the purpose of sales related decision making. It is of organization's best interest to outsource the multi-party statistical computations to a CSP while outsourcing the analytical work to an external analyzer due to the lack of computational resources. Furthermore, security and privacy of user inserted sales data becomes a crucial aspect due to the engagement of external parties invoking the necessity of an efficient security protocol. We are going to consider this case for presenting the secure multi-party based cloud computing solution throughout this paper.

IV. Secure Multi-party based Cloud Computing Framework

Fig.1 illustrates the architecture of the secure multi-party based cloud computing framework that we propose to address the case which we have discussed in the previous section. Framework consists with four main entities namely proxy server, cloud server, analyzer and parties who are taking part in the computations.

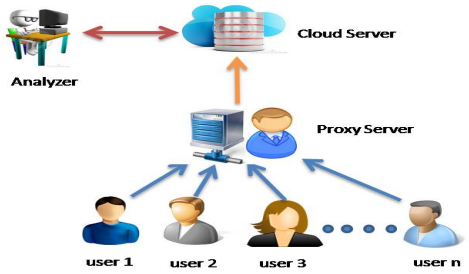

Fig. 1. Proposed Secure Multi-party based Cloud Computing Framework

The main functionality of the cloud server is to perform the required statistical computations upon the reception of user data while the function of the proxy server is to hide the identity of each of the users to provide identity anonymization. Furthermore, the analyzer is the external party which receives the statistical parameters of user sales data which is used for analytical purposes. The functionalities of each of these entities are illustrated in the following subsections while explaining how a secure computation can be achieved with the proposed framework. Moreover, we have used the following cryptographic keys in the proposed framework.

- RSA public and private key pair for proxy server, cloud server and the analyzer for authentication purposes.
- We have used extended ElGamal public key encryption scheme (EEES) [16] to encrypt private sales data of users. Therefore, we generate EEES keys at the analyzer end and send the parameters required for encrypting the data (El_para) to the users via the authentication process. El_para includes integer values p, g, N, y and k. p is a large secure prime value and $N = p.q$ where q is also a large secure prime value. Furthermore, g represents a root of $GF(p)$ and $y = g^x \ mod \ p$ where x denotes the private key of the encryption scheme. k is a positive number selected by users when encrypting the data [16].
- We have used two 3DES keys to encrypt messages conveyed between the entities during computation time.

A. Authentication and Key Exchange

In the proposed framework, cloud server and analyzer are external entities. Therefore, it is necessary to mutually authenticate cloud server and analyzer as well as cloud server and the proxy which is the exit point of the organization's network. Let us assume that RSA public keys of each entity are known to all other entities. Fig.2 represents the procedure for establishing authentication between the cloud server and the analyzer. In Fig.2 RSA signing and encryption procedures are denoted with usual notations [ ] and { } respectively. The authentication process is initiated with analyzer sending an authentication request. After receiving it, cloud server generates a response including the current timestamp (TS), the secret key phrase (KP_{DES_CA}) to generate the 3DES key to be shared

with the analyzer (K_{DES_CA}) and send it to the analyzer after signing with RSA private key ($[KP_{DES\_CA}, TS]_{Cloud}$) and then encrypting with the public key of the analyzer ($\{[KP_{DES\_CA}, TS]_{Cloud}\}_{Analyzer}$). Then, at the analyzer end, the received message is decrypted accordingly and check whether the TS is within the defined clock skew to authenticate the cloud server. If authenticated, analyzer attains KP_{DES_CA} and generates K_{DES_CA}.

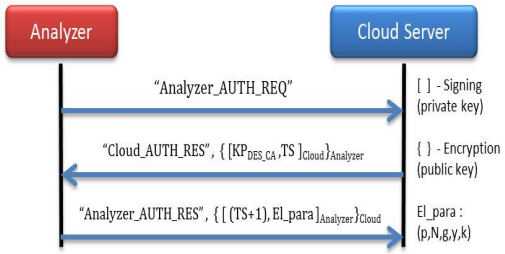

Fig. 2. Mutual Authentication between Cloud server & Analyzer

In order to authenticate itself to the cloud server, analyzer generates a message by incrementing received TS value by one and attaching El_para which is signed and encrypted by RSA private key of analyzer and public key of the proxy server respectively. Among the values in El_para; p, g, N and y are public values while k is a positive number usually selected by users when encrypting the data. Therefore, k becomes a security critical parameter for the encryption scheme. Hence, we are sending El_para in an encrypted environment. Thereafter, the complete message is signed and encrypted with RSA private key of the analyzer and RSA public key of the cloud server respectively. After receiving this message at the cloud server, it decrypts the complete message and extracts the value of TS. Finally, it checks whether the received value is equals to the value of the original TS sent to the analyzer incremented by one. If it is verified, analyzer is authenticated to the cloud server whereas the encrypted El_para are retrieved and saved to be sent to the proxy server.

We have also adopted a similar approach to authenticate proxy server to the cloud server. Hence, at the end of the authentication of proxy server and cloud server, we are able to share the 3DES symmetric key K_{DES_PC} between them while El_para will be saved at the proxy in order to be transferred to the users.

It is also important for us to authenticate users with the proxy server to make sure that only the valid users are allowed to take part in the computing process. In order to accomplish this, we have created a file in the proxy server which includes secure hash algorithm 1 (SHA1) hashes of username and password pairs of valid users. When a user logs into the system, SHA1 hashes of the entered username and password are transferred to the proxy server in order to be validated. If the login information is validated, El_para will be forwarded to the user end in order to encrypt the values that needed to be sent for the computations.

B. User Data Encryption

Each user can start sending private data for computations after being successfully authenticated into the system. When data is entered by a user party, first the data is encrypted with EEES by using received El_para. This encryption induces the required homomorphic properties into user data allowing required statistical computations to be carried out on the encrypted data at the cloud server. Furthermore, Eqn. 1 represents the encrypted result (A, B) for a plaintext message M when encrypted with EEES.

$$(A, B) = (g^k \bmod p, (y^k\ (M + r \times p) \bmod N) \bmod p) \quad (1)$$

It is important to note that (g, k, y, p, N) represents El_para and therefore common for all users whereas r is a positive integer randomly selected by each user. Eqn. 1 also depicts that component A of the ciphertext is independnt of M. After encryption, the resulting ciphertext values of user sales data is transmitted to the proxy server.

C. Proxy Server Functionality

The main idea of having a proxy server is to hide the identity of data sent from each user from the cloud server. At the beginning of a computation, proxy server waits for encrypted data from all the users. After receiving all of them, it creates a new message by including all the encrypted values of users $(A, B_1), (A, B_2), (A, B_3)..., (A, B_n)$ and the number of users (n). Then a hashed message authentication code (HMAC) of the preceding message is generated with SHA1. After that HMAC is appended to the message along with the current TS. Finally, the whole message is encrypted with the 3DES key, K_{DES_PC} and forwarded to the cloud server to begin the computational process. The structure of the message sent from proxy server is illustrated in Fig.3.

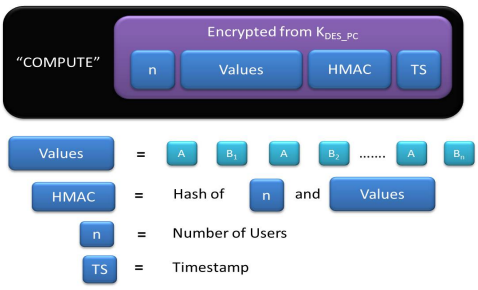

Fig. 3. Structure of the Frame sent from Proxy to Cloud Server

D. Cloud Server and Analyzer Functionality

After cloud server is authenticated to both analyzer and proxy server, it waits for encrypted user data from the proxy server. When it received such a data set, then cloud server decrypts it with 3DES key, K_{DES_PC} and separates the three components in the received message which are TS, HMAC and the original message consisting with encrypted user data and number of users. Firstly, cloud server checks the received TS is within the defined clock skew to make sure that it is not a

replayed frame. Then, if it is verified a HMAC is created with the original message part and check whether the computed and received HMACs are matching. If so, the private data of users are accepted and otherwise a retransmission request will be sent to the proxy server.

Consider that encrypted private data of users are successfully acquired by the cloud server. Then it starts the statistical computations by first computing the encrypted summation (A_{Sum}, B_{Sum}). By using the additive homomorphism of EEES, it is possible to express (A_{Sum}, B_{Sum}) as;

$$A_{Sum} = A \tag{2}$$

$$B_{Sum} = B_1 + B_2 + B_3 + ... + B_n \tag{3}$$

Then, cloud server generates a new message by including the values A_{Sum}, B_{Sum} and the number of users n. We have to send the encrypted summation with number of users since EEES does not support homomorphism for division in order to compute the encrypted mean at the cloud server. After that, HMAC of the message is created and it is appended to the tail of the message along with the current TS as shown in Fig.4. Finally, the complete frame is encrypted with 3DES key, K_{DES_CA} and forwarded to the analyzer.

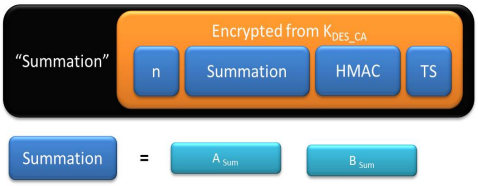

Fig. 4. Structure of the Summation Data Frame sent from Cloud to Analyzer

After receiving the data frame with encrypted summation at the analyzer end, it segments the message accordingly by decrypting with the 3DES key, K_{DES_CA}. Then, the validity of the frame is verified through TS and HMAC as explained previously. After that, the encrypted summation (A_{Sum}, B_{Sum}) is decrypted with the analyzer's private key (x) of EEES as given in Eqn.4. Finally, the decrypted result (M_1) is divided by n to obtain the mean value of users' private sales data.

$$M_1 = B_{Sum}\ ((A_{Sum})^x)^{-1}\ mod\ p \tag{4}$$

$$Mean\ value\ of\ user\ inserted\ sales\ data = \frac{M_1}{n} \tag{5}$$

In order to compute the rest of the statistical parameters variance, standard deviation, skewness and kurtosis; it is necessary to obtain the mean value which is encrypted using EEES. Therefore, after recovering the mean value by the analyzer, it re-encrypts the mean value using El_para which gives us the encrypted components A_{Mean} and B_{Mean}. Furthermore, the analyzer generates a HMAC for the new encrypted mean and appends it to the encrypted mean. Finally, the current TS is also attached to the tail of the message and forwards it to the cloud server after encrypting the complete frame with the shared 3DES symmetric key, K_{DES_CA} as shown in Fig.5.

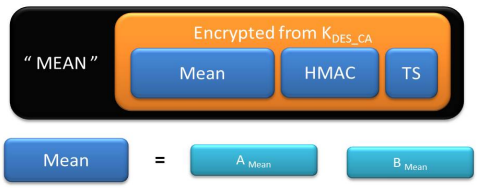

Fig. 5. Structure of the Message which carries Encrypted Mean Value to Cloud

At reception of the above message, cloud server will first decrypt it with K_{DES_CA} and acquire A_{Mean} and B_{Mean} components after validating TS and HMAC of the received data frame. In order to compute the statistical parameters variance, standard deviation, skewness and kurtosis it is necessary to obtain the deviations of the encrypted user sales data from the encrypted mean. These values can be calculated through the subtraction property which is derived from the additive homomorphism of EEES. If the encrypted mean deviation of i^{th} user's sales data is given by V_i;

$$V_i = (A_i, (B_i - B_{Mean})), \tag{6}$$

where (A_i, B_i) denotes the EEES encrypted sales data of i^{th} user. After that we need to obtain 2^{nd}, 3^{rd} and 4^{th} powers of the values V_i for all i. In order to achieve that, we use the property of homomorphism on powers of values possess by EEES [16]. Therefore, we can write the j^{th} power of V_i, (V_i^j) as;

$$V_i^j = (A_i^j, (B_i - B_{Mean})^j) \tag{7}$$

Thereafter, we can obtain $\sum_{i=1}^{n} V_i^2$, $\sum_{i=1}^{n} V_i^3$, $\sum_{i=1}^{n} V_i^4$ by using Eqn.2 and Eqn.3. Finally, all theses values are forwarded to the analyzer after attaching the HMAC, current TS and encrypting the complete message with K_{DES_CA}.

When the above mentioned data frame is received at the analyzer, it will first decrypt the data frame with K_{DES_CA} and acquire the encrypted components $\sum_{i=1}^{n} V_i^2$, $\sum_{i=1}^{n} V_i^3$, $\sum_{i=1}^{n} V_i^4$ after validating the TS and HMAC which were appended to the received message. Then, the received encrypted components are decrypted using the relation given in Eqn.4. Let us consider that decrypted components of $\sum_{i=1}^{n} V_i^2$, $\sum_{i=1}^{n} V_i^3$, $\sum_{i=1}^{n} V_i^4$ are denoted by V_1, S_1 and K_1 respectively. Then we can determine the variance, standard deviation, skewness and kurtosis of user inserted data from the equations given below.

$$Variance = \frac{V_1}{n} \tag{8}$$

$$Standard\ deviation\ (S.D.) = \sqrt{\frac{V_1}{n}} \tag{9}$$

$$Skewness = \frac{S_1}{(S.D.)^3} \tag{10}$$

$$Kurtosis = \frac{K_1}{(S.D.)^4} \tag{11}$$

After the completion of a computing session, analyzer sends a message to all the users through the proxy server to initiate the next computing session. Moreover, it is not necessary to carry out mutual authentication between entities again at the start of a new session. However, to ensure the freshness, we flush the existing keys after consecutive 10 sessions. Then, it is necessary to carry out the mutual authentication process again before starting a computing session.

V. Performance Analysis

An experimental setup has been formulated to analyze the performance of the proposed framework. Fig.6 illustrates the setup along with the system configuration information of the PCs and the server. In order to implement this setup, we have compiled a program in Java where each entity is executing as a separate program. Values to be computed are inserted through interfaces of user programs while analyzer program will display the results at the conclusion of a computing session.

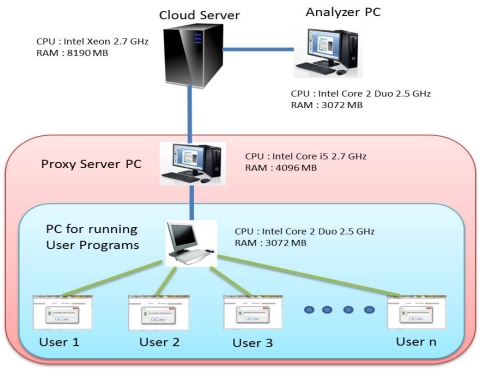

Fig. 6. Experimental Setup

A. Definitions of Measured Parameters

- Encryption Time (T_e):Time taken to encrypt user inserted value at a user program using EEES.
- Entity Process Time (T_{EP}):Time duration that encrypted data are handled at proxy server, cloud server and analyzer until the conclusion of a computing session.
- Total Process Time (T_{TP}) :Total time duration that user data is handled at all the entities user, proxy server, cloud server and analyzer.
- Transmission Delay (T_{TD}): Collective time taken for transmission of system messages between entities between initiation and conclusion of a computational session.
- Total Time (T_T): Time duration between the initiation and the conclusion of computing session.

Therefore, we can write;

$$T_{TP} = T_{EP} + T_e \times n \tag{12}$$

$$T_T = T_{TP} + T_{TD} \tag{13}$$

B. Test Results

In order to evaluate the performance of the implemented framework, we have to understand the variation of T_e, T_{EP}, T_{TP} and T_T as a function of number of users and size of the user input respectively. Tests carried out for T_e suggested that, it exhibits an exponential variation with the size of the prime numbers associated with EEES. This fact suggests that even though increasing the size of the prime values might strengthen the security, efficiency of the system will be degraded. Since we are considering sales values of users as the messages to be encrypted, prime size of 64 bits is quite adequate for our application. Hence, all the experiments were carried out with 64 bits as the prime size of the encryption scheme.

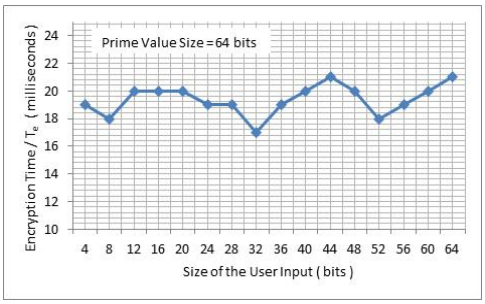

Fig. 7. Variation of Encryption Time as a function of User Input Size

Fig.7 shows the variation of T_e with the size of the user input. Even though the user input is varied from 4 bits to 64 bits, fluctuation of T_e is only limited to the range of 17 ms to 21 ms. Therefore, we can conclude that input data size does not influence to T_e when the prime size is fixed. Furthermore, average T_e taken for a single user is approximately 20 ms. Fig.8 illustrates the variation of T_{EP} and T_{TP} with n. We can clearly observe that T_{EP} is independent of n. However, T_{TP} exhibits a linear accumilation with n due to the fact that, total T_e is linearly increasing with n.

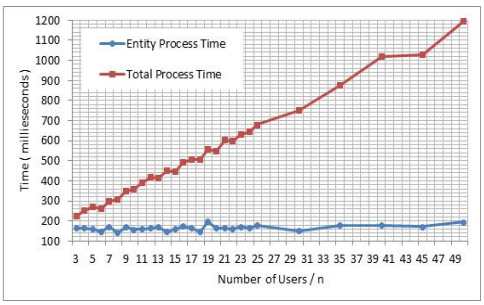

Fig. 8. Variation of T_{EP} & T_{TP} as a function of n

Fig.9 depicts that both T_{EP} and T_{TP} are independent of the size of the user input. However, T_{TP} curve is elevated approximately by 200 ms than T_{EP}. This shift accounts for the total T_e of 10 users participated for the computing session.

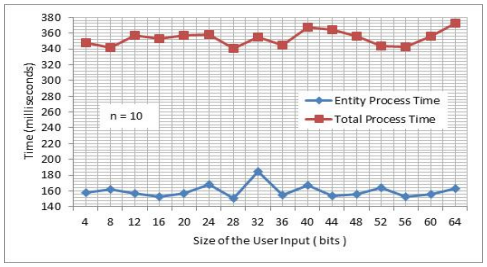

Fig. 9. Variation of T_{EP} & T_{TP} as a function of User Input Size

According to Eqn.13, T_T is dependent upon T_{TD}. T_{TD} is affected by number of factors such as bandwidth, distance or hop count between the entities and system configuration of the entities. Fig.10 shows the variation of T_{EP} and T_T when the number of users are varied from 5 to 50 at two instances where the hop count (HC) between proxy server and cloud server is 1 and 17 respectively. As we can observe form Fig.10, T_{EP} values when HC is 1 and 17 varies in the same way. T_T at HC 1 exhibits a similar behavior since the associated T_{TD} is approximately 4 ms and does not impose any significant effect on T_T. However, T_T when HC = 17 is elevated by 900 ms from T_T at HC = 1. This elevation represents the increment in T_{TD} since the distance between proxy server and cloud server is increased. This fact proves that T_T is affected by T_{TD}.

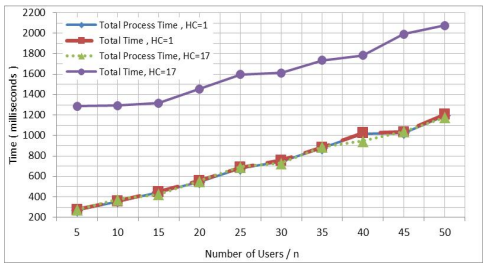

Fig. 10. Variation of T_{EP} & T_T as a function of n

The results acquired so far suggests that, both T_{EP} and T_{TP} are independent of the network parameters and only be depending upon the specifications of the PCs and server used in the experimental setup. Moreover, average T_{EP} of the system is approximately 162.53 ms. Hence, we can derive;

$$Average\ T_{TP} = Average\ T_{EP} + (Average\ T_e \times n) \quad (14)$$

$$Average\ T_{TP} = (162.53 + 20 \times n)\ ms \quad (15)$$

The maximum T_T represented in Fig.10 is slightly higher than 2s and T_{TP} is approximately 1.2s when HC is 17 and $n = 50$. Therefore, these experimental results verify that the proposed system is capable of operating efficiently under practical circumstances.

VI. Conclusions

In this paper we have proposed a secure multi-party based cloud computing framework based on outsourcing statistical parameter computations on users' private data. The inclusion of proxy server in the framework ensures that cloud server will not have any idea about the ownership of each encrypted data component received at the cloud. Hence, user data anonymization is achieved. Furthermore, data privacy is also guaranteed due to the fact that user data are encrypted with EEES while computations are also carried out on the encrypted data. We have also used concepts of TS and HMAC to make the security framework withstand against replay attacks and possible integrity violations. The performance evaluation that we have illustrated in Sec. IV provides evidence for the efficiency of the proposed framework. Therefore, we can conclude that cloud environments can be successfully deployed to improve the efficiency of multi-party computations while enforcing the security requirements of user parties.

References

[1] N. Maheshwari, and K. Kiyawat, "Structural Framing of Protocol for Secure Multiparty Cloud Computation", in *Proceedings of* 5^{th} *Asia Modelling Symposium*, IEEE, Kuala Lampur, Malaysia, May 2011.

[2] A. C. Yao, "Protocols for Secure Computations", in *Proceedings of Annual Symposium on Foundations of Computer Science*, vol.0 , pp. 160-164, Nov. 1982.

[3] R. Oppliger, "Contemporary Cryptography", *Artech House Computer Security Library*, Norwood, 2005.

[4] F. Shaikh, S. Haider "Security Threats in Cloud Computing," in *Proceedings of International Conference for Internet Technology and Secured Transactions (ICITST)*, IEEE, Abu Dhabi, UAE, Dec. 2011.

[5] Q. Ma, L. Xiao, I.-L. Yen, M. Tu, and F. Bastani, "An Adaptive Multiparty Protocol for Secure Data Protection", in *Proceedings of* 11^{th} *International Conference on Parallel and Distributed Systems*, IEEE, Fukuoka, Japan, July 2005.

[6] S. Chakraborty, S. Sehgal, and A. Pal, "Privacy Preserving E-negotiation Protocols based on Secure Multi-party Computation", in *Proceedings of SoutheastCon.*, IEEE, Fort Lauderdale, USA, April 2005.

[7] S. Bleikertz, M. Schunter, C. W. Probst, D. Pendarakis, and K. Eriksson, "Security Audits of Multi-tier Virtual Infrastructures in Public Infrastructure Clouds", in *Proceedings of the Workshop on Cloud Computing Security Workshop (CCSW)*, ACM, New York, USA, Oct. 2010.

[8] S. Pearson, Y. Shen, and M. Mowbray, "A Privacy Manager for Cloud Computing", in *Proceedings of the* 1^{st} *International Conference on Cloud Computing*, Springer-Verlag, Berlin, Germany, 2009.

[9] M. Mowbray, and S. Pearson, "A Client-based Privacy Manager for Cloud Computing", in *Proceedings of the* 4^{th} *International ICST Conference on Communication System Software and Middleware*, ACM, New York, USA, Jun. 2009.

[10] D. Mishra, and M. Chandwani, "Anonymity Enabled Secure Multi-party Computation for Indian BPO", in *Proceedings of Region 10 Conference - TENCON*, IEEE, Taipei, Republic of China, Nov. 2007.

[11] M. Tebaa, S. El Hajji, and A. El Ghazi, "Homomorphic Encryption Method applied to Cloud Computing", in *Proceedings of National Days of Network Security and Systems (JNS2)*, IEEE, Marrakech, Morocco, Apr. 2012.

[12] A.-F. Chan, "Symmetric-key Homomorphic Encryption for Encrypted Data Processing", in *Proceedings of International Conference on Communications (ICC)*, IEEE, Dresden, Germany, Jun. 2009.

[13] C. Castelluccia, E. Mykletun, and G. Tsudik, "Efficient Aggregation of Encrypted Data in Wireless Sensor Networks", in *Proceedings of the* 2^{nd} *Annual International Conference on Mobile and Ubiquitous Systems: Networking and Services (MobiQuitous)*, IEEE, San Diego, USA, Jul. 2005.

[14] J. Wei, S. Guo, and Q. Xu, "Secure Homomorphic Aggregation Algorithm of Mixed Operations in Wireless Sensor Networks", in *Proceedings of International Conference on E-Business and Information System Security (EBISS)*,IEEE, Wuhan, China, May 2009.

[15] W. Luo, and X. Li, "A Study of Secure Multi-party Statistical Analysis", in *Processdings of International Conference on Computer Networks and Mobile Computing (ICCNMC)*, IEEE, Shanghai, China, Oct. 2003.

[16] G. Xiang, B. Yu, and P. Zhu, "Algorithm of Fully Homomorphic Encryption", in *Proceedings of* 9^{th} *International Conference on Fuzzy Systems and Knowledge Discovery (FSKD)*, IEEE, Sichuan, China, May 2012.

Appendix B

JAVA Programs

In order to evaluate the performance of our proposed framework, we have implemented a prototype using java programming language. Each of the entity in the prototype is executed as a separate program as mentioned in the Section 4.1 of the report. The implementation of the entities User, Proxy Server, Cloud Server and Analyzer are coded as separate java files. Contents of those programs are given below.

B.1 User.java File

```
/**
 *      Program Name          :       User Program
 *      Version               :       FINAL
 *      Authors               :       P.S.Ranaweera
 *                            :       G.P.H.Sandaruwan
 *      Language              :       Java
 *      Date                  :       01/06/2013
 * */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.net.Socket;
import java.security.SecureRandom;
import java.util.Random;
```

```
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JPasswordField;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import java.sql.Timestamp;

public class User {
        /**
         * Variable Definitions
         */
        //Defining Prime value size for the ElGamal Encryption Scheme
        static final int bit_length = 64;
        //Defining String values for A, B encrypted components and user input
        String A,B,input;
        //Defining ElGamal public parameters
        BigInteger p,q,K,y,g;
    //Generating Random number r for ElGamal Encryption
        Random r_ran = new SecureRandom();
    BigInteger r = new BigInteger(bit_length,r_ran);
    //Definition of Input Buffered Reader and Output Print writer for
        sending and Receiving messages through the socket
    BufferedReader in;
    PrintWriter out;
    /*
     * User Interface Defining
     */
    JFrame frame = new JFrame(" U S E R Program");
    JTextArea messageArea = new JTextArea(50, 60);
    /*
     * Constructor of the User Class
     */
    public User () {
        // Defining Layout of the GUI
        messageArea.setEditable(false);
        frame.getContentPane().add(new JScrollPane(messageArea), "Center");
        frame.pack();
    }
    /**
```

```
 * Private Function Defining
 */
//Function for obtaining the Proxy Server Address from User
private String getServerAddress() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter IP Address of the Proxy Server :",
        "**** P R O X Y ***** A D D R E S S ****",
        JOptionPane.QUESTION_MESSAGE);
}
//Function for obtaining the user name of the User
private String getName() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter the username :",
        "**** U S E R ***** N A M E ****",
        JOptionPane.PLAIN_MESSAGE);
}
//Function for obtaining the User Input from User
private String getInput() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter the number to be calculated : ",
        "**** U S E R ***** I N P U T ****",
        JOptionPane.PLAIN_MESSAGE);
}
//Function for obtaining the User Password
private String getPassword(){

    int checkPwd = 0;
    JPasswordField pwd = new JPasswordField(10);
    int action = JOptionPane.showConfirmDialog(frame, pwd,"Enter the
       Password : ",JOptionPane.OK_CANCEL_OPTION);

    do {
            if((action < 0) || (pwd.getPassword().length == 0)){
                    JOptionPane.showMessageDialog(null,"You must enter a
                        password to proceed");
            }else {
```

```
                    checkPwd++;
                    break;
                }
        }while(checkPwd == 0);
        return new String(pwd.getPassword());
    }
    /*
     *  run() Function of the class User
     */
    private void run() throws IOException {
        messageArea.append("\n\n\t***********************************"
                           +"******************\n" +
                    "\t=====================================\n" +
                    "\t********** U S E R ***** P R O G R A M
                        ***********\n" +
                    "\t====================================" +
                    "\n\t*****************************************"
                           +"*************\n\n\n");
        //Getting the server address
        String serverAddress = getServerAddress();
        //Creating the Socket
        Socket socket = new Socket(serverAddress, 9001);
        //Initializing Input and Output Streams
        in = new BufferedReader(new
            InputStreamReader(socket.getInputStream()));
        out = new PrintWriter(socket.getOutputStream(), true);
        /*
         * While loop for processing messages between User program and Proxy
             Server
         */
        while (true) {
            //Reading the Input as a String
                String line = in.readLine();
            if (line.startsWith("USERNAME")) {
                //Sending User name to Proxy
                out.println(getName());
            } else if(line.startsWith("PASSWORD")){
                //Sending Password to Proxy
                out.println(getPassword());
```

```
} else if (line.startsWith("NAMEACCEPTED")) {
    //User Authentication completed with Proxy
    //Segmenting the message to assign ElGamal public values to
        variables
    String P[] = line.split(" ");
    p = new BigInteger(P[1]);
    q = new BigInteger(P[2]);
    K = new BigInteger(P[3]);
    y = new BigInteger(P[4]);
    g = new BigInteger(P[5]);
    //Obtaining User Input
    input = getInput();
    messageArea.append("\n\n@@   Input entered at
        "+getCurrentTimestamp()+"\n\n");
    messageArea.append("\nUser Input : "+input+"\n\n\n");
            //Multiplying the input by 100 and rounding it to two
                decimal points
                    Double Doubleinput = new Double(input);
                    Doubleinput = Doubleinput * 100;
                    Long Longinput = Doubleinput.longValue();
    String inputHundred = Longinput.toString();
    messageArea.append("@@ Encryption initiating at
        "+getCurrentTimestamp()+"\n\n");
    //Encryption of the User Input
    Encryption(inputHundred, messageArea);
    messageArea.append("@@ Value encrypted at
        "+getCurrentTimestamp()+"\n\n");
    //Combining A and B components of the encrypted user input
    String output = A+" "+B;
    //Sending encrypted user input to the Proxy Server
    out.println("USER"+" "+output);
    messageArea.append("Sent message : "+"USER"+"
        "+output+"\n\n");
    messageArea.append("@@       Sent time :
        "+getCurrentTimestamp()+"\n\n\n\n");
}else if(line.startsWith("Proxy_FINISH")){
    /*
     * Next Session
     */
```

```
                messageArea.append("\t@@@@@@@@@@@@@@@@@@@@@@ Welcome to a New
                    Session @@@@@@@@@@@@@@@@@@@@\n\n\n");
                input = getInput();
                messageArea.append("\nUser Input : "+input+"\n\n\n");
                        Double Doubleinput = new Double(input);
                        Doubleinput = Doubleinput * 100;
                        Long Longinput = Doubleinput.longValue();
                        String inputHundred = Longinput.toString();
                Encryption(inputHundred, messageArea);
                String output = A+" "+B;
                out.println("USER"+" "+output);
                messageArea.append("Sent message : "+"USER"+" "+output+"\n");
                messageArea.append("@@       Sent time :
                    "+getCurrentTimestamp()+"\n\n\n\n");
            }
          }
    }
/**
 * Main Method of the Program
 * */
    public static void main(String[] args) throws Exception {
        //Creating an instance of the user class an runs the constructor
        User user = new User();
        user.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        user.frame.setVisible(true);
        //Calling the run() function
        user.run();
    }
/**
 * Public Functions
 * */
    //Function for ElGamal Encryption
    public void Encryption(String SalesValue, JTextArea messageArea){
        BigInteger N;
            messageArea.append("p = "+p+"\n");
            messageArea.append("q = "+q+"\n");
            // N = p * q
            N = p.multiply(q);
            messageArea.append("N = "+N+"\n");
```

```
            messageArea.append("y = "+y+"\n");
            messageArea.append("r = "+r+"\n");
            //Encryption Step
            String m = SalesValue;
            //Converting String input into a BigInteger
            BigInteger M = new BigInteger(m);
                        // bx = M + (r * p) mod N
                BigInteger bx = M.add(r.multiply(p)).mod(N);
                // b = ( bx * ( y ^ K ) mod p ) mod p
                BigInteger b = bx.multiply(y.modPow(K, p)).mod(p);
                // a = (g ^ K) mod p
                BigInteger a = g.modPow(K, p);
            messageArea.append("Encrypted part A = "+a+"\n");
            messageArea.append("Encrypted part B = "+b+"\n\n\n");
            //Converting BigInteger A and B components into Strings
            A = a.toString();
            B = b.toString();
    }
    //Function to get the current Time Stamp
    public Timestamp getCurrentTimestamp(){
        return new Timestamp(System.currentTimeMillis());
    }

}
```

B.2 ProxyServer.java File

```
/**
*      Program Name          :      Proxy Server Program
*      Version               :      FINAL
*      Authors               :      P.S.Ranaweera
*                           :      G.P.H.Sandaruwan
*      Language              :      Java
*      Date                  :      01/06/2013
* */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.security.spec.KeySpec;
import java.sql.Timestamp;
import java.util.HashSet;
import javax.crypto.Cipher;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESedeKeySpec;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import org.apache.commons.codec.binary.Base64;

public class ProxyServer {
        /**
         * Variable Definitions
         */
    //Defining hash set for user names
    private static HashSet<String> names = new HashSet<String>();
```

```
//Defining hash set for print writers
private static HashSet<PrintWriter> writers = new
    HashSet<PrintWriter>();
    //Creating String to store hash values
    static String hash;
    //String array to store encrypted A component values of users
    static String A[] = null;
    //String array to store encrypted B component values of users
    static String B[] = null;
    //String to hold cloud server address, inputs and ElGamal parameters
    static String CloudServerAddress;
    static String input,input1,input2;
    static String ElgammalParameters;
    //Defining Port numbers
private static final int PORT1 = 9001, PORT2 = 9002;
    //prime value size of the RSA encryption scheme
    static final int RSA_bit_length = 512;
    //Maximum number of clients associated for a computational session
    static int max_clients = 50;
    static int check = 0, i = 0;
    private static int Client = 0, sessionNum = 0;
static Integer ClientNum;
    //Defining RSA public and private parameters for the Proxy
    static BigInteger e_proxy, N_proxy, d_proxy;
    //Defining RSA public parameters for Cloud
    static BigInteger e_cloud, N_cloud;
    //Time stamp variables
     static Timestamp ts, DesTs;
    //Definition of Input Buffered Reader and Output Print writer for
       sending and Receiving messages through the socket
    public static BufferedReader proxyin;
    public static PrintWriter proxyout;
    //Defining parameters for Triple DES symmetric key encryption scheme
public static final String UNICODE_FORMAT = "UTF8";
public static final String DESEDE_ENCRYPTION_SCHEME = "DESede";
public static KeySpec ks;
public static SecretKeyFactory skf;
public static Cipher cipher;
static byte[] arrayBytes;
```

```
public static String myEncryptionKey;
public static String myEncryptionScheme;
static SecretKey key;
/*
 * Interface Definition
 */
static JFrame frame = new JFrame("P R O X Y S E R V E R Program");
static JTextArea messageArea = new JTextArea(50, 60);
/**
 * Private Function Defining
 * */
//Function for obtaining the Cloud Server Address
private static String getServerAddress() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter IP Address of the Cloud Server:",
        "Cloud Server Address",
        JOptionPane.QUESTION_MESSAGE);
}
//Function for obtaining Number of clients participating in the
    computation
private static String getClientNumber() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter the number of users participating in the computation :",
        "Number of Participants",
        JOptionPane.QUESTION_MESSAGE);
}
/**
 * Main Method of the Program
 * */
public static void main(String[] args) throws Exception {
    //Defining Layout of the GUI
    messageArea.setEditable(false);
    frame.getContentPane().add(new JScrollPane(messageArea), "Center");
    frame.pack();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
    messageArea.append("\n\n\t**************************************"
```

```
                                +"****************************\n" +
                "\t================================================\n"
                    +
                "\t********** P R O X Y ***S E R V E R*** P R O G R A
                    M **********\n" +
                "\t================================================"
                    +
                "\n\t****************************************"
                                +"*************************\n\n\n");
messageArea.append("@@@@@@@@@@@@@@@@ The Proxy server is running
    @@@@@@@@@@@@@@@@@@\n\n\n");
//Initializing A and B String arrays with maximum number of user
A = new String[max_clients];
B = new String[max_clients];
//Function for generating RSA parameters
RSA();
//Triple DES parameter generation
myEncryptionKey = "ThisIsSpartaThisIsSparta";
myEncryptionScheme = DESEDE_ENCRYPTION_SCHEME;
arrayBytes = myEncryptionKey.getBytes(UNICODE_FORMAT);
ks = new DESedeKeySpec(arrayBytes);
skf = SecretKeyFactory.getInstance(myEncryptionScheme);
cipher = Cipher.getInstance(myEncryptionScheme);
key = skf.generateSecret(ks);
//Creating Server socket for users under PORT1
ServerSocket listener1 = new ServerSocket(PORT1);
//Getting Cloud Server address
        CloudServerAddress = getServerAddress();
        //Getting the number of clients
        ClientNum = new Integer(getClientNumber());
        //Creating a socket to communicate with the Cloud server
            under PORT2
        Socket socket = new Socket(CloudServerAddress, PORT2);
    //Initializing Input buffered reader and output print writer
    proxyin = new BufferedReader(new
        InputStreamReader(socket.getInputStream()));
    proxyout = new PrintWriter(socket.getOutputStream(), true);
    /*
     * Authentication with Cloud Server
```

```
*/
   //Sending Authentication request to Server
   proxyout.println("Proxy_AUTH_REQ"+" "+e_proxy+" "+N_proxy);
   messageArea.append("@@       Authentication request sent to
       cloud at           "+getCurrentTimestamp()+"\n\n");
   //Reading incoming message
   String line = proxyin.readLine();
   messageArea.append("Cloud Server : "+line+"\n\n");
   if (line.startsWith("CONNECT")){
          messageArea.append("@@       Authentication response
              received from cloud at
              "+getCurrentTimestamp()+"\n\n");
          String I[] = line.split(" ");
          //Assigning RSA public parameters of the cloud from
              received message
          e_cloud = new BigInteger(I[1]);
          N_cloud = new BigInteger(I[2]);
          messageArea.append("e : "+e_cloud+"\n"+"N :
              "+N_cloud+"\n\n");
   }
   DesTs = getCurrentTimestamp();
messageArea.append("DesTs : "+DesTs+"\n");
String auth = myEncryptionKey+" "+DesTs.toString();
//RSA double encrypting the Triple DES symmetric key phrase and
    Time Stamp
BigInteger Eauth = DoubleEncrypt(new
    BigInteger(auth.getBytes()));
//Sending Authentication response to the Cloud server
proxyout.println("Proxy_AUTH_RES"+" "+Eauth.toString());
messageArea.append("@@         DES parameters sent to cloud at
              "+getCurrentTimestamp()+" and
    waiting......\n\n");
line = proxyin.readLine();
 if (line.startsWith("Cloud_AUTH_RES")){
     String D[] = line.split(" ");
     //RSA double decrypting the received message
     BigInteger DD = DoubleDecrypt(new BigInteger(D[1]));
     String StrDESts = new String(DD.toByteArray());
   messageArea.append("StrDESts : "+StrDESts);
```

```
                String D1[] = StrDESts.split(" ");
                //Revealing the time stamp received
                String Ts = D1[0]+" "+D1[1];
                Timestamp DESts = Timestamp.valueOf(Ts);
                messageArea.append("DESts : "+DESts.toString()+"\n"+"DESts :
                    "+DESts.toString()+"\n\n");
                if (DESts.getTime() == (DesTs.getTime() + 1)){
                        messageArea.append("@@       Authentication completed
                            with Cloud at     "+getCurrentTimestamp()+"\n\n");
                        ElgammalParameters = D1[2]+" "+D1[3]+" "+D1[4]+"
                            "+D1[5]+" "+D1[6];
                }
              }
        try {
            while (true) {
                new Handler1(listener1.accept()).start();
            }
        } finally {
            listener1.close();
        }
    }
    /**
     * Handler thread for handling User programs
     */
    private static class Handler1 extends Thread {
        private String name, password, CheckPassword = "PASSWORD";
        private Socket socket;
        private BufferedReader in;
        private PrintWriter out;
        private int checkName = 0;
        //Constructor of the Handler
        public Handler1(Socket socket) {
            this.socket = socket;
        }
        /*
         * run() function of the class Proxy
         */
        public void run() {
            try {
```

```
//Initialize input and output streams for the User side
in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true);
/*
 * While loop for processing messages between Proxy Server
     and the User programs
 */
while (true) {
        if(checkName == 0){
                out.println("USERNAME");
                //Getting user name from the User program
                name = in.readLine();
                messageArea.append("User name : "+name+"\n\n");
                //Checking the user name
                if (name.isEmpty() == true) {
                        messageArea.append("Name is empty\n");
                }else{
                        synchronized (names) {
                                if (!names.contains(name)) {
                                        names.add(name);
                                        checkName++;
                                }
                        }
                }
        }
    if(checkName == 1){
        out.println("PASSWORD");
        //Getting the password
        password = in.readLine();
        messageArea.append("\tPassword Checking ......\n");
        //Checking the password
        if (password.compareTo(CheckPassword) == 0) {
                checkName++;
                messageArea.append("Password Verified\n\n");
                Client++;
                messageArea.append("Client "+Client+"
                    Authenticated\n\n");
                break;
```

```
            }
            if(checkName == 2)    break;
        }
    }
    //Sending verification message with ElGamal public parameters
    out.println("NAMEACCEPTED"+" "+ElgammalParameters);
    writers.add(out);
    /*
     * While loop for processing user inputs
     */
    while (true){
            if (check < ClientNum){
                    input = in.readLine();
            if (input == null) {
                    return;
            }
            messageArea.append(input+"\nSize of the received
                message : "+input.length()+"\n\n");
            if (input.startsWith("USER")){
                    String in[] = input.split(" ");
                    //Assigning A and B encrypted components to as
                        String array elements
                    A[i] = in[1];
                    B[i] = in[2];
                    messageArea.append("Client "+(i+1)+" values
                        recieved at "+getCurrentTimestamp()+"\n\n");
                    messageArea.append("Client "+(i+1)+" Values :
                        "+A[i]+" "+B[i]+"\n\n");
                    check++;
                    i++;
            }
            else{
                    for (PrintWriter writer: writers){
                                    writer.println("MESSAGE " + name
                                        + ": " + input);
                    }
            }
            }
            /*
```

```
 * Sending User values to Cloud Server for Computation
 */
if (check == ClientNum) {

        String values = null;
        //For loop for combining A and B array
            components
        for(int j = 0 ; j < i ; j++){
                if(j == 0){
                        values = A[j]+" "+B[j];
                }
                else {
                        values = values +" "+A[j]+"
                            "+B[j];
                }
        }
        messageArea.append("Values : "+values+"\n\n");

        try {
                //HMAC generation
                hash = Hash(i+" "+values);
        } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
        }
        messageArea.append("Hash : "+hash+"\n");
        String output = i+" "+values+" "+hash+"
            "+getCurrentTimestamp().toString();
        messageArea.append("Unencrypted message :
            "+output+"\n");
        //Symmetric key encryption with Triple DES
        String Eoutput =
            TripleDESencrypt(output,cipher);
        messageArea.append("Encrypted message :
            "+Eoutput+"\n\n\n\n");
        //Sending user values to Cloud Server
        proxyout.println("COMPUTE"+" "+Eoutput);
        messageArea.append("@@@@@@@          Values
            Sent to cloud at
            "+getCurrentTimestamp()+"\n\n");
```

```
                        check++;
                        //Gathering hash response which indicated
                            whether HMAC is correct or not
                        String HashRes = proxyin.readLine();
                        if(HashRes.startsWith("CORRECT_HASH")){
                                messageArea.append("Hash response is
                                    correct\n\n");
                        }else if(HashRes.startsWith("INCORRECT_HASH")){
                                messageArea.append("Hash
                                    Incorrect\nResending Message...\n\n
                                    ");
                                proxyout.println("COMPUTE"+" "+Eoutput);
                                messageArea.append("Sent time to cloud
                                    : "+getCurrentTimestamp()+"\n\n");
                        }
                        /*
                         * Next Session
                         */
                        while(true){
                    String Session = proxyin.readLine();
                    messageArea.append("Waiting for Next Session\n\n");
                    if(Session.startsWith("Cloud_FINISH")){
                            sessionNum++;
                            messageArea.append("Session "+sessionNum+"
                                Concluded\n\nWaiting for the initiation of
                                the next session......");
                            check = 0;
                            i = 0;
                            for(PrintWriter writer : writers){
                                    writer.println("Proxy_FINISH");
                            }
                    }
                    if(check == 0) break;
                }
            }
        }
    } catch (IOException e) {
        System.out.println(e);
    } finally {
```

```
                if (name != null) {
                    names.remove(name);
                }
                if (out != null) {
                    writers.remove(out);
                }
                try {
                    socket.close();
                } catch (IOException e) {
                }
            }
        }
    }
/**
 * Public Functions
 * */
 //Function for computing RSA public parameters for Proxy Server
 public static void RSA(){
       SecureRandom r = new SecureRandom();
       BigInteger p = new BigInteger(RSA_bit_length,100,r);
       BigInteger q = new BigInteger(RSA_bit_length,100,r);
       N_proxy = p.multiply(q);
       BigInteger n =
           (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
       e_proxy = new BigInteger("3");
       while(n.gcd(e_proxy).intValue()>1){
              e_proxy = e_proxy.add(new BigInteger("2"));
       }
       d_proxy = e_proxy.modInverse(n);
    }
 //Function for RSA Encrypting
 public static BigInteger RSAencrypt (BigInteger message,BigInteger ex,
     BigInteger Nx){
         return message.modPow(ex, Nx);
 }
 //Function for RSA Decryption
 public static BigInteger RSAdecrypt (BigInteger message){
        return message.modPow(d_proxy, N_proxy);
 }
```

```
//Function for RSA Signing
public static BigInteger RSAsign (BigInteger message){
       return message.modPow(d_proxy, N_proxy);
}
//Function for RSA Un-signing
public static BigInteger RSAunsign (BigInteger message){
       return message.modPow(e_cloud, N_cloud);
}
//Function for RSA Double Encrypting
public static BigInteger DoubleEncrypt (BigInteger message){
       return RSAencrypt((RSAsign(message)),e_cloud,N_cloud);
}
//Function for RSA Double Decrypting
public static BigInteger DoubleDecrypt (BigInteger message){
       return RSAunsign(RSAdecrypt(message));
}
//Function for generating HMAC
public static String Hash (String message) throws
    NoSuchAlgorithmException{
      MessageDigest mDigest = MessageDigest.getInstance("SHA1");
      byte[] result = mDigest.digest(message.getBytes());
      StringBuffer stringbuffer = new StringBuffer();
      for (int i = 0; i < result.length; i++) {
          stringbuffer.append(Integer.toString((result[i] & 0xff) + 0x100,
              16).substring(1));
      }
      return stringbuffer.toString();
}
//Function for Triple DES Encryption
public static String TripleDESencrypt(String unencryptedString, Cipher
    cipher) {
      String encryptedString = null;
      try {
                    cipher.init(Cipher.ENCRYPT_MODE, key);
          byte[] plainText = unencryptedString.getBytes("UTF8");
          byte[] encryptedText = cipher.doFinal(plainText);
          encryptedString = new String(Base64.encodeBase64(encryptedText));
      } catch (Exception e) {
          e.printStackTrace();
```

```
        }
        return encryptedString;
    }
    //Function for getting the current Time stamp
    public static Timestamp getCurrentTimestamp(){
           return new Timestamp(System.currentTimeMillis());
    }

}
```

B.3 CloudServer.java File

```
/**
*      Program Name          :      Cloud Server Program
*      Version               :      FINAL
*      Authors               :      P.S.Ranaweera
*                            :      G.P.H.Sandaruwan
*      Language              :      Java
*      Date                  :      01/06/2013
* */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.net.ServerSocket;
import java.net.Socket;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.security.SecureRandom;
import java.sql.Timestamp;
import java.util.HashSet;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.KeySpec;
import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESedeKeySpec;
import javax.swing.JFrame;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import org.apache.commons.codec.binary.Base64;

public class CloudServer {
       /**
        * Variable Definitions
        */
```

```
//Defining hash set for print writers
private static HashSet<PrintWriter> writers = new HashSet<PrintWriter>();
//Defining strings for holding ElGamal Parameters and encrypted output
public static String ElgammalParameters, Eoutput;
//String arrays for storing encrypted A and B components of user values
static String StrA[],StrB[];
//String for storing HMAC
static String hash;
//Defining port number
private static final int PORT = 9002;
//Definition of time limit and RSA prime value size
static final int Time_Limit = 2000, RSA_bit_length = 512;
static int check = 0, max_clients = 50;
static Integer i = 0;
//BI Arrays for storing A, B components of encrypted data and prime p
static BigInteger A[], B[],p;
//RSA public and private values for cloud
static BigInteger e_cloud,N_cloud,d_cloud;
//RSA public values for Analyzer
static BigInteger e_analyzer,N_analyzer;
//RSA public values for Proxy Server
static BigInteger e_proxy,N_proxy;
//Arrays for storing Standard, Cubic and Biquadrate deviations
static BigInteger SDA[],SDB[], CDA[], CDB[], QDA[], QDB[];
//Encrypted statistical parameter BI variables
static BigInteger MeanA,MeanB,VarianceA,VarianceB, SkewnessA, SkewnessB,
    KurtosisA, KurtosisB;
//Time stamp Variables
static Timestamp ts, DesTs;
//Triple DES secret keys for analyzer and proxy
static SecretKey key_Analyzer, key_Proxy;
//Triple DES variables
public static final String UNICODE_FORMAT = "UTF8";
public static final String DESEDE_ENCRYPTION_SCHEME = "DESede";
public static KeySpec ks;
public static SecretKeyFactory skf;
public static Cipher cipher;
static byte[] arrayBytes;
public static String myEncryptionKey_Proxy, myEncryptionKey_Analyzer;
```

```
public static String myEncryptionScheme;
 /* User Interface Defining
  */
static JFrame frame = new JFrame("C L O U D S E R V E R Program");
static JTextArea messageArea = new JTextArea(50, 60);
/**
 * Main Method of the Program
 * */
public static void main(String[] args) throws Exception {
        /*Defining Layout of the GUI
         */
        messageArea.setEditable(false);
    frame.getContentPane().add(new JScrollPane(messageArea), "Center");
    frame.pack();
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    frame.setVisible(true);
        messageArea.append("\n\n\t************************************"
                                +"*********************************\n" +
                    "\t================================================\n"
                        +
                    "\t********** C L O U D ***S E R V E R*** P R O G R A
                        M ***********\n" +
                    "\t================================================"
                        +
                    "\n\t****************************************"
                                +"************************\n\n\n");
        //Creating Server socket
    ServerSocket listener = new ServerSocket(PORT);
    /*Initializing array variables with maximum number of users
     */
        StrA = new String[max_clients];
        StrB = new String[max_clients];
        A       = new BigInteger[max_clients];
        B       = new BigInteger[max_clients];
        SDA = new BigInteger[max_clients];
        SDB = new BigInteger[max_clients];
        CDA = new BigInteger[max_clients];
        CDB = new BigInteger[max_clients];
        QDA = new BigInteger[max_clients];
```

```
        QDB = new BigInteger[max_clients];
    //Generating RSA parameters
    RSA();
    messageArea.append("@@@@@@@@@@@@@@@@ The Cloud server is running
        @@@@@@@@@@@@@@@@@@\n\n\n");
    try {
         while (true) {
             new Handler(listener.accept()).start();
         }
    } finally {
        listener.close();
    }
}
/**
 * Handler thread for handling User programs
 */
  private static class Handler extends Thread {
        private Socket socket;
    private BufferedReader in;
    private PrintWriter out;
    //Constructor of the Handler
    public Handler(Socket socket) {
        this.socket = socket;
    }
    /*
     * run() function of the class Proxy
     */
    public void run() {
        try {
                //Initialize input and output streams for the User side
            in = new BufferedReader(new
                InputStreamReader(socket.getInputStream()));
            out = new PrintWriter(socket.getOutputStream(), true);
                //Analyzer Triple DES key generation
                myEncryptionKey_Analyzer = "ThisIsSpartaThisIsSparta";
                myEncryptionScheme = DESEDE_ENCRYPTION_SCHEME;
                arrayBytes =
                    myEncryptionKey_Analyzer.getBytes(UNICODE_FORMAT);
                ks = new DESedeKeySpec(arrayBytes);
```

```
skf = SecretKeyFactory.getInstance(myEncryptionScheme);
cipher = Cipher.getInstance(myEncryptionScheme);
key_Analyzer = skf.generateSecret(ks);
writers.add(out);
String Add = null;
/*
 * While loop for processing messages
 */
while (true) {
    String input = in.readLine();
    if (input == null) {
        return;
    }
    /*
     * Proxy Authentication
     */
    else if(input.startsWith("Proxy_AUTH_REQ")){
        messageArea.append("Proxy Server is
            authenticating.....\n\n");
        String I[] = input.split(" ");
        //Extracting RSA public parameters for proxy
        e_proxy = new BigInteger(I[1]);
        N_proxy = new BigInteger(I[2]);
        check++;
        //Sending RSA public cloud parameters to Proxy
        out.println("CONNECT"+" "+e_cloud+" "+N_cloud);
        messageArea.append("Authenticating response
            sent.....\n\n");
    }
    else if(input.startsWith("Proxy_AUTH_RES")){
        messageArea.append("Authenticating response
            received.....\n\n");
        ts = getCurrentTimestamp();
        String D[] = input.split(" ");
        //RSA double decrypting
        BigInteger DD = DoubleDecrypt(new
            BigInteger(D[1]),e_proxy,N_proxy);
        String DDStr = new String(DD.toByteArray());
        String des[] = DDStr.split(" ");
```

```
//Extracting proxy TripleDES key phrase from the
    message
String myEncryptionKey_Proxy = des[0];
//Generating TripleDES symmetric key between proxy
    and cloud
myEncryptionScheme = DESEDE_ENCRYPTION_SCHEME;
arrayBytes =
    myEncryptionKey_Proxy.getBytes(UNICODE_FORMAT);
ks = new DESedeKeySpec(arrayBytes);
skf =
    SecretKeyFactory.getInstance(myEncryptionScheme);
cipher = Cipher.getInstance(myEncryptionScheme);
key_Proxy = skf.generateSecret(ks);
String Ts2 = des[1]+" "+des[2];
//Checking the time stamp for time limit
if((checkTimestamp(Ts2, ts).intValue()) < Time_Limit){
        messageArea.append("Timestamp is within the
            time limits\n\n");
        Timestamp DesTs = setTimestamp(Ts2);
        Long tsDes = DesTs.getTime();
        tsDes++;
        Timestamp TSres = new Timestamp(tsDes);
        messageArea.append("TSres :
            "+TSres.toString()+"\n");
        String authRes = TSres.toString()+"
            "+ElgammalParameters;
        //RSA double encrypting the message
        BigInteger res = DoubleEncrypt(new
            BigInteger(authRes.getBytes()),e_proxy,N_proxy);
        //Sending the message
        out.println("Cloud_AUTH_RES"+"
            "+res.toString());
        messageArea.append("@@@@@@          DES
            parameters sent and Proxy authentication is
            verified at
            "+getCurrentTimestamp()+"\n\n");
}else{
        messageArea.append("Timestamp exceeds the time
            limit\nPossibility of a REPLAY attack\n\n");
```

```
        }
    }
    /*
     * Analyzer Authentication
     */
    else if(input.startsWith("Analyzer_AUTH_REQ")){
        messageArea.append("Analyzer is
            authenticating....\n\n");
        messageArea.append(input+"\n\n");
        String E[] = input.split(" ");
        //Extracting RSA public parameters for analyzer
        e_analyzer = new BigInteger(E[1]);
        N_analyzer = new BigInteger(E[2]);
        //Sending RSA public parameters of the cloud to
            analyzer
        out.println("Cloud_Encrypt"+" "+e_cloud+" "+N_cloud);
        DesTs = getCurrentTimestamp();
        messageArea.append("DesTs : "+DesTs+"\n\n");
        String auth = myEncryptionKey_Analyzer+"
            "+DesTs.toString();
        //RSA double encrypting TripleDES symmetric key
        BigInteger Eauth = DoubleEncrypt(new
            BigInteger(auth.getBytes()),e_analyzer,N_analyzer);
        //Sending the authentication response along with
            session key
        out.println("Cloud_AUTH_RES"+" "+Eauth.toString());
    }
    else if(input.startsWith("Analyz_AUTH_RES")){
        messageArea.append("Analyzer authentication response
            received\n\n");
        String D[] = input.split(" ");
        BigInteger DD = DoubleDecrypt(new
            BigInteger(D[1]),e_analyzer,N_analyzer);
        String StrDESts = new String(DD.toByteArray());
        messageArea.append("StrDESts : "+StrDESts+"\n");
        String D1[] = StrDESts.split(" ");
        String Ts = D1[0]+" "+D1[1];
        Timestamp DESts = Timestamp.valueOf(Ts);
```

```
            messageArea.append("DESts :
                "+DESts.toString()+"\n"+"DesTs :
                "+DesTs.toString()+"\n\n");
            //Checking the time stamp with the received one
            if (DESts.getTime() == (DesTs.getTime() + 1)){
                    messageArea.append("@@@@@@          Analyzer
                        authentication completed at
                        "+getCurrentTimestamp()+"\n\n");
                    p =new BigInteger(D1[2]);
                    ElgammalParameters = D1[2]+" "+D1[3]+"
                        "+D1[4]+" "+D1[5]+" "+D1[6];
            }
        }
        /*
         * Mean Computation
         */
        else if(input.startsWith("COMPUTE")){
            messageArea.append("\n\n@@@@@@      Encrypted values
                received from proxy server at
                "+getCurrentTimestamp()+"\n\n");
            String in0[] = input.split(" ");
            //TripleDES decrypting
            String Dinput = TripleDESdecrypt(in0[1]);
            String in1[] = Dinput.split(" ");
            i = new Integer(in1[0]);
            String in2 = null;
            //Rearranging the string values
            for(int j = 1 ; j < ((2*i)+1) ; j++ ){
                    if(j == 1){
                            in2 = in1[j];
                    }else {
                            in2 = in2 +" "+in1[j];
                    }
            }
            try {
                    //Generating HMAC for the message
                    hash = Hash(in1[0]+" "+in2);
                                    } catch
                                        (NoSuchAlgorithmException e)
```

```
                                {
                                    e.printStackTrace();
                                }
                    //Checking the HMAC
                    if ((hash.compareTo(in1[((2*i)+1)])) == 0){
                            messageArea.append("Hash is correct\n");
                            //Assigning values to StrA[] StrB[] array
                                elements
                            for(int j = 1, l = 0 ; j < 2*i ; j = j+2 ,
                                l++){
                                        StrA[l] = in1[j];
                                        StrB[l] = in1[j+1];
                                        messageArea.append("StrA"+l+" :
                                            "+StrA[l]+"\n"+"StrB"+l+" :
                                            "+StrB[l]+"\n");
                            }
                            //Sending the acknowledgement for correct hash
                            out.println("CORRECT_HASH");
                            ts = getCurrentTimestamp();
                        messageArea.append("Input received time :
                            "+ts+"\n");
                        messageArea.append("Timestamp decrypted :
                            "+in1[((2*i)+2)]+" "+in1[((2*i)+3)]+"\n\n");
                        //Checking the time stamp for time limit
                        if ((checkTimestamp(in1[((2*i)+2)]+"
                            "+in1[((2*i)+3)],ts)).intValue() < Time_Limit){
                                messageArea.append("Timestamp is within
                                    the time limits\n\n");
                                //Converting String A and B components
                                    to BI
                                for(int j = 0; j < i ; j++){
                                            A[j] = new
                                                BigInteger(StrA[j]);
                                            B[j] = new
                                                BigInteger(StrB[j]);
                                }
                                //Computing Addition
                                Add = Addition(A,B,i);
```

```
                    messageArea.append("@@       Addition
                        Calculated at
                        "+getCurrentTimestamp()+"\n\n");
                    String answer1 = i.toString()+" "+Add;
                    try {
                                hash = Hash(answer1);
                            } catch
                                (NoSuchAlgorithmException e)
                                {
                                    e.printStackTrace();
                            }
                    messageArea.append("Hash : "+hash+"\n");
                    String output1 = answer1+" "+hash+"
                        "+getCurrentTimestamp().toString();
                    messageArea.append("Unencrypted message
                        : "+output1+"\n");
                //Triple DES encrypting the message
                    Eoutput =
                        TripleDESencrypt(output1,cipher);
                messageArea.append("Encrypted message :
                    "+Eoutput+"\n\n\n");
                    for(PrintWriter writer : writers){
                    writer.println("Summation"+" "+Eoutput);
                }
                messageArea.append("@@@@@@       Summation
                    sent at
                    "+getCurrentTimestamp()+"\n\n");
            }else {
                    messageArea.append("Timestamp exceeds
                        the time limit\nPossibility of a
                        REPLAY attack\n\n");
            }
        }       else    {
                    messageArea.append("Hash is
                        incorrect\nWaiting for correct
                        message....");
                        out.println("INCORRECT_HASH");
        }
    }
```

```
/*
 * Statistical Parameter Computation
 */
else if(input.startsWith("MEAN")){
    messageArea.append("\n\n@@@@        Encryped Mean
        received from Analyzer at
        "+getCurrentTimestamp()+"\n\n");
    String in0[] = input.split(" ");
    String Dinput = TripleDESdecrypt(in0[1]);
    String in1[] = Dinput.split(" ");
    try {
            hash = Hash(in1[0]+" "+in1[1]);
                        } catch
                            (NoSuchAlgorithmException e)
                            {
                                e.printStackTrace();
                            }
    //HMAC checking
    if ((hash.compareTo(in1[2])) == 0){
                    messageArea.append("Hash is correct\n");
                    ts = getCurrentTimestamp();
            messageArea.append("Mean received time :
                "+ts+"\n");
            messageArea.append("Timestamp decrypted :
                "+in1[3]+" "+in1[4]+"\n\n");
            //Checking the time stamp for time limit
            if ((checkTimestamp(in1[3]+"
                "+in1[4],ts)).intValue() < Time_Limit){
                    messageArea.append("Timestamp is within
                        the time limits\n\n");
                    MeanA = new BigInteger(in1[0]);
                    MeanB = new BigInteger(in1[1]);
                    //Calculating Deviations
                    for(int j = 0 ; j < i ; j++){
                            String Deviation =
                                Subtraction(A[j],B[j],MeanA,MeanB);
                            String Z[] = Deviation.split("
                                ");
```

```
                BigInteger DA = new
                    BigInteger(Z[1]);
                BigInteger DB = new
                    BigInteger(Z[2]);
                String SquareDeviation =
                    Square(DA,DB);
                String O[] =
                    SquareDeviation.split(" ");
                SDA[j] = new BigInteger(O[1]);
                SDB[j] = new BigInteger(O[2]);
                String CubicDeviation =
                    Cube(DA,DB);
                String O1[] =
                    CubicDeviation.split(" ");
                CDA[j] = new BigInteger(O1[1]);
                CDB[j] = new BigInteger(O1[2]);
                String BiquadrateDeviation =
                    Biquadrate(DA,DB);
                String O2[] =
                    BiquadrateDeviation.split("
                    ");
                QDA[j] = new BigInteger(O2[1]);
                QDB[j] = new BigInteger(O2[2]);
            }
        //Calculating Variance
            String Var = Addition(SDA,SDB,i);
            String V[] = Var.split(" ");
            VarianceA = new BigInteger(V[0]);
            VarianceB = new BigInteger(V[1]);
            String Variance = VarianceA+"
                "+VarianceB;
            messageArea.append("@@       Variance
                Calculated at
                "+getCurrentTimestamp()+"\n");

            //Calculating Skewness
            String Skew = Addition(CDA,CDB,i);
            String S[] = Skew.split(" ");
            SkewnessA = new BigInteger(S[0]);
```

```
                SkewnessB = new BigInteger(S[1]);
                String Skewness = SkewnessA+"
                    "+SkewnessB;
                messageArea.append("@@       Skewness
                    Calculated at
                    "+getCurrentTimestamp()+"\n");

                //Calculating Kurtosis
                String Kurt = Addition(QDA,QDB,i);
                String K[] = Kurt.split(" ");
                KurtosisA = new BigInteger(K[0]);
                KurtosisB = new BigInteger(K[1]);
                String Kurtosis = KurtosisA+"
                    "+KurtosisB;
                messageArea.append("@@       Kurtosis
                    Calculated at
                    "+getCurrentTimestamp()+"\n\n");
                /*
                 * Sending answers to the analyzer
                 */
                String answer2 = Variance+"
                    "+Skewness+" "+Kurtosis;
                try {
                            hash = Hash(answer2);
                        } catch
                            (NoSuchAlgorithmException e)
                            {
                                e.printStackTrace();
                        }
                messageArea.append("Hash : "+hash+"\n");
                String output = answer2+" "+hash+"
                    "+getCurrentTimestamp().toString();
            messageArea.append("Unencrypted message :
                "+output+"\n");
            //TripleDES encrypting the message
            Eoutput = TripleDESencrypt(output,cipher);
            messageArea.append("Encrypted message :
                "+Eoutput+"\n\n\n");
                for(PrintWriter writer : writers){
```

```
                        //Sending answers
                        writer.println("STAT_Para"+" "+Eoutput);
                        messageArea.append("@@@@@@@@@
                            Answers sent at
                            "+getCurrentTimestamp()+"\n\n\n");
                }
                }else {
                        messageArea.append("Timestamp exceeds
                            the time limit\nPossibility of a
                            REPLAY attack\n\n");
                }
            }else {
                    messageArea.append("Hash is incorrect\nWaiting
                        for correct message....");
            }
            }else if(input.startsWith("CORRECT_HASH")){
             messageArea.append("Hash response is correct \n\n");
        }else if(input.startsWith("INCORRECT_HASH")){
            messageArea.append("Hash Incorrect\nResending
                Message...\n\n ");
                if(input.contains("MEAN")){
                        out.println("Summation"+" "+Eoutput);
                }else {
                        out.println("STAT_Para"+" "+Eoutput);
                }
                messageArea.append("@@       Sent time to
                    Analyzer :
                    "+getCurrentTimestamp()+"\n\n");
        }
        /*
         * Session conclusion
         */
        else if(input.startsWith("FINISH")){
            messageArea.append("Session terminated\n\n");
            for(PrintWriter writer: writers){
                    writer.println("Cloud_FINISH");
            }
        }
        System.out.println("end\n"+input);
```

```
                }
            } catch (IOException e) {
                System.out.println(e);
            } catch (InvalidKeyException e) {
                            e.printStackTrace();
                    } catch (NoSuchAlgorithmException e) {
                            e.printStackTrace();
                    } catch (NoSuchPaddingException e) {
                            e.printStackTrace();
                    } catch (InvalidKeySpecException e) {
                            e.printStackTrace();
                    } finally {
                if (out != null) {
                    writers.remove(out);
                }
                try {
                    socket.close();
                } catch (IOException e) {
                }
            }
        }
    }
  /**
   * Public Functions
   * */
//Function for computing RSA public parameters for Cloud Server
public static void RSA(){
    SecureRandom r = new SecureRandom();
    BigInteger p = new BigInteger(RSA_bit_length,100,r);
    BigInteger q = new BigInteger(RSA_bit_length,100,r);
    N_cloud = p.multiply(q);
    BigInteger n =
        (p.subtract(BigInteger.ONE)).multiply(q.subtract(BigInteger.ONE));
    e_cloud = new BigInteger("3");
        while(n.gcd(e_cloud).intValue()>1){
                e_cloud = e_cloud.add(new BigInteger("2"));
        }
        d_cloud = e_cloud.modInverse(n);
}
```

```
//Function RSA Decryption
public static BigInteger RSAdecrypt (BigInteger message){
        return message.modPow(d_cloud, N_cloud);
}
//Function for RSA Signing
public static BigInteger RSAsign (BigInteger message){
        return message.modPow(d_cloud, N_cloud);
}
//Function for RSA Un-signing
public static BigInteger RSAunsign (BigInteger message, BigInteger ex,
    BigInteger Nx){
    return message.modPow(ex, Nx);
}
//Function for RSA encryption
public static BigInteger RSAencrypt (BigInteger message,BigInteger ex,
    BigInteger Nx){
    return message.modPow(ex, Nx);
}
//Function for RSA double encrypt
public static BigInteger DoubleEncrypt (BigInteger message,BigInteger ex,
    BigInteger Nx){
    return RSAencrypt((RSAsign(message)),ex,Nx);
}
//Function for RSA Double decryption
public static BigInteger DoubleDecrypt (BigInteger message,BigInteger ex,
    BigInteger Nx){
    return RSAunsign((RSAdecrypt(message)),ex,Nx);
}
//Function for calculating addition on encrypted data
public static String Addition(BigInteger AA[], BigInteger AB[], Integer i){
         BigInteger BI1 = AA[0];
         BigInteger BI2 = new BigInteger("0");
           for(int j = 0 ; j < i ; j++){
                  BI2 = BI2.add(AB[j]);
           }
        return (BI1.toString()+" "+BI2.toString());
}
//Function for calculating subtraction on encrypted data
```

```
public static String Subtraction(BigInteger DA1, BigInteger DB1,BigInteger
    DA2, BigInteger DB2){
    BigInteger BI1 = DA1;
        BigInteger BI2 = DB1.subtract(DB2);
               return "Subtraction"+" "+BI1.toString()+" "+BI2.toString();
}
//Function for calculating square value of encrypted data
public static String Square(BigInteger SA, BigInteger SB){
    BigInteger BI1 = SA.pow(2);
        BigInteger BI2 = SB.pow(2);
               return "Square"+" "+BI1.toString()+" "+BI2.toString();
}
//Function of calculating cubic value of encrypted data
public static String Cube(BigInteger CA, BigInteger CB){
    BigInteger BI1 = CA.modPow(new BigInteger("3"), p);
        BigInteger BI2 = CB.modPow(new BigInteger("3"), p);
               return "Cube"+" "+BI1.toString()+" "+BI2.toString();
}
//Function of calculating biquadrate value of encrypted data
public static String Biquadrate(BigInteger BQA, BigInteger BQB){
    BigInteger BI1 = BQA.modPow(new BigInteger("4"), p);
        BigInteger BI2 = BQB.modPow(new BigInteger("4"), p);
               return "Biquadrate"+" "+BI1.toString()+" "+BI2.toString();
}
//Function for generating HMAC
public static String Hash (String message) throws NoSuchAlgorithmException{
    MessageDigest mDigest = MessageDigest.getInstance("SHA1");
    byte[] result = mDigest.digest(message.getBytes());
    StringBuffer stringbuffer = new StringBuffer();
    for (int i = 0; i < result.length; i++) {
         stringbuffer.append(Integer.toString((result[i] & 0xff) + 0x100,
            16).substring(1));
    }
        return stringbuffer.toString();
}
//Function for getting current time stamp
public static Timestamp getCurrentTimestamp(){
        return new Timestamp(System.currentTimeMillis());
}
```

```
//Function for setting a time stamp
public static Timestamp setTimestamp(String ts){
    return Timestamp.valueOf(ts);
}
//Function for check time stamps
public static Long checkTimestamp(String ts0, Timestamp ts2){
    Timestamp ts1 = setTimestamp(ts0);
    Long difference = ts2.getTime()-ts1.getTime();
    messageArea.append("Timestamp Difference : "+difference+"\n\n");
    return difference;
}
//Function for TripleDES Encryption
public static String TripleDESencrypt(String unencryptedString, Cipher
    cipher) {
    String encryptedString = null;
    try {
                cipher.init(Cipher.ENCRYPT_MODE, key_Analyzer);
        byte[] plainText = unencryptedString.getBytes("UTF8");
        byte[] encryptedText = cipher.doFinal(plainText);
        encryptedString = new String(Base64.encodeBase64(encryptedText));
    } catch (Exception e) {
         e.printStackTrace();
    }
        return encryptedString;
}
//Function for TripleDES decryption
public static String TripleDESdecrypt(String encryptedString) {
    String decryptedText=null;
    try {
        cipher.init(Cipher.DECRYPT_MODE, key_Proxy);
        byte[] encryptedText = Base64.decodeBase64(encryptedString);
        byte[] plainText = cipher.doFinal(encryptedText);
        decryptedText= new String(plainText);
    } catch (Exception e) {
        e.printStackTrace();
    }
        return decryptedText;
}
}
```

B.4 Analyzer.java File

```
/**
*      Program Name          :      Analyzer Program
*      Version               :      FINAL
*      Authors               :      P.S.Ranaweera
*                            :      G.P.H.Sandaruwan
*      Language              :      Java
*      Date                  :      01/06/2013
* */
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.PrintWriter;
import java.math.BigInteger;
import java.net.Socket;
import java.security.SecureRandom;
import java.util.Random;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import java.security.InvalidKeyException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.sql.Timestamp;
import java.security.spec.InvalidKeySpecException;
import java.security.spec.KeySpec;
import javax.crypto.Cipher;
import javax.crypto.NoSuchPaddingException;
import javax.crypto.SecretKey;
import javax.crypto.SecretKeyFactory;
import javax.crypto.spec.DESedeKeySpec;
import org.apache.commons.codec.binary.Base64;

public class Analyzer {
       /**
        * Variable Definitions
        */
```

```
    //String values to hold A, B components of the encrypted Mean and
        encrypted output
    static String MeanA, MeanB, Eoutput;
    String A1,B1,input,MeanA1,MeanB1,hash, hashMean;
    //ElGamal prime value size, Time limit of Time stamps and RSA prime
        value size defining
static final int bit_length = 64, Time_Limit = 2000, RSA_bit_length =
    512;
//Double variables
static Double denomenator, Addition, Mean;
//ElGamal public parameters
static BigInteger p,q,K,g,y;
//ElGamal private parameter x
static BigInteger x = new BigInteger("152543");
//RSA public and private parameters for Analyzer
static BigInteger e_analyzer,N_analyzer, d_analyzer;
//RSA public parameters for Cloud Server
static BigInteger e_cloud,N_cloud;
//Time stamp variables
static Timestamp ts,ts_mul, ts_add, ts_mean, ts_variance;
//Input and output stream definition
BufferedReader in;
PrintWriter out;
/*
 * User Interface Defining
 */
JFrame frame = new JFrame("A N A L Y Z E R Program");
JTextArea messageArea = new JTextArea(50, 60);
//Triple DES parameters for symmetric key encryption
private static final String UNICODE_FORMAT = "UTF8";
public static final String DESEDE_ENCRYPTION_SCHEME = "DESede";
private KeySpec ks;
private SecretKeyFactory skf;
private static Cipher cipher;
byte[] arrayBytes;
private String myEncryptionKey;
private String myEncryptionScheme;
static SecretKey key;
/*
```

```
 * Constructor of the Analyzer Program
 */
public Analyzer() {
    // Defining Layout of the Analyzer GUI
    messageArea.setEditable(false);
    frame.getContentPane().add(new JScrollPane(messageArea), "Center");
    frame.pack();
}
/**
 * Private Function Defining
 * */
//Function for obtaining the Cloud Server Address
private String getServerAddress() {
    return JOptionPane.showInputDialog(
        frame,
        "Enter IP Address of the Server:",
        "Cloud Server Address",
        JOptionPane.QUESTION_MESSAGE);
}
/*
 * run() function of the class Analyzer
 */
private void run() throws IOException, InvalidKeyException,
    NoSuchAlgorithmException, NoSuchPaddingException,
    InvalidKeySpecException {
    int sessionNum = 0;
    messageArea.append("\n\n\t******************"
                       +"****************************************\n" +
                     "\t=========================================\n" +
                     "\t********** A N A L Y Z E R ***** P R O G R A M
                         **********\n" +
                     "\t=========================================" +
                     "\n\t************************"
                       +"************************************\n\n\n");
    //Generating RSA parameters
    RSA();
    //Generating ElGamal parameters
    ElGamalParameter(bit_length);
    // Getting the cloud server address
```

```
String serverAddress = getServerAddress();
//Initializing the socket under Port number 9002
Socket socket = new Socket(serverAddress, 9002);
//Initializing input and Output streams
in = new BufferedReader(new
    InputStreamReader(socket.getInputStream()));
out = new PrintWriter(socket.getOutputStream(), true);
/*
 *Authentication with Cloud Server
 */
//Sending analyzer authentication request to cloud server
out.println("Analyzer_AUTH_REQ"+" "+e_analyzer+" "+N_analyzer);
messageArea.append("@@       Authentication Request Sent at
           "+getCurrentTimestamp()+"\n\n");
/*
 * While loop for processing messages between Analyzer program and
    Cloud Server
 */
while (true) {
        //Reading the input
    String line = in.readLine();
    if (line.startsWith("Cloud_Encrypt")) {
        String P[] = line.split(" ");
        //Assigning RSA public parameters of cloud
        e_cloud = new BigInteger(P[1]);
        N_cloud = new BigInteger(P[2]);
    }
    else if(line.startsWith("Cloud_AUTH_RES")){
        messageArea.append("Authentication Initiated\n\n");
        ts = getCurrentTimestamp();
        String D[] = line.split(" ");
        BigInteger DD = DoubleDecrypt(new BigInteger(D[1]));
        String DDStr = new String(DD.toByteArray());
        String des[] = DDStr.split(" ");
        //Extracting the Triple DES key phrase from the received
            message
        myEncryptionKey = des[0];
        //Generating Triple DES symmetric key from key phrase
        myEncryptionScheme = DESEDE_ENCRYPTION_SCHEME;
```

```
            arrayBytes = myEncryptionKey.getBytes(UNICODE_FORMAT);
            ks = new DESedeKeySpec(arrayBytes);
            skf = SecretKeyFactory.getInstance(myEncryptionScheme);
            cipher = Cipher.getInstance(myEncryptionScheme);
            key = skf.generateSecret(ks);
            //Extracting the Time stamp
            String Ts1 = des[1]+" "+des[2];
            //Checking the time stamps
            if((checkTimestamp(Ts1, ts, messageArea)).intValue() <
                Time_Limit){
                    Timestamp DesTs = setTimestamp(Ts1);
                    Long tsDes = DesTs.getTime();
                    tsDes++;
                    Timestamp TSres = new Timestamp(tsDes);
                    String AuthRes = TSres.toString()+" "+p+" "+q+" "+K+"
                        "+y+" "+g;
                    //RSA double encrypting the message
                    BigInteger res = DoubleEncrypt(new
                        BigInteger(AuthRes.getBytes()));
                    //Sending Analyzer authentication request
                    out.println("Analyz_AUTH_RES"+" "+res.toString());
                    messageArea.append("@@       Authenticaition response
                        sent at            "+getCurrentTimestamp()+"\n\n");
            }else {
                    messageArea.append("Timestamp exceeds the time
                        limit\nPossibility of a REPLAY attack\n\n");
            }
        }
        else if(line.startsWith("Summation")){
            sessionNum++;
            messageArea.append("Session "+sessionNum+" Answers\n\n");
            ts = getCurrentTimestamp();
            messageArea.append("@@       Summation received at :
                        "+ts+"\n");
            String M[] = line.split(" ");
            String Answers1 = TripleDESdecrypt(M[1]);
            String Ans1[] = Answers1.split(" ");
            try {
                    //Generating HMAC for the message
```

```
                hash = Hash(Ans1[0]+" "+Ans1[1]+" "+Ans1[2]);
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }
        //Checking the HMAC
        if ((hash.compareTo(Ans1[3])) == 0){
            messageArea.append("Answer Hash is correct\n\n");
            //Sending the response mentioning correct HMAC
            out.println("CORRECT_HASH");
            //checking of timestamp for time limit
            if((checkTimestamp(Ans1[4]+" "+Ans1[5], ts,
                messageArea)).intValue() < Time_Limit){
                messageArea.append("Timestamp is within the time
                    limit\n\n");
                denomenator = new Double(Ans1[0]);
                    //Summation
                    BigInteger AdditionA = new BigInteger(Ans1[1]);
                BigInteger AdditionB = new BigInteger(Ans1[2]);
                Addition =
                    Decryption(AdditionA,AdditionB,p,x).doubleValue();
                messageArea.append("**          Addition
                    =             "+(Addition/100)+"\n\n");
                messageArea.append("@@          Addition time :
                    "            +getCurrentTimestamp()+"\n\n");
                //Mean Calculation
                    Mean = (Addition/denomenator);
                Long LongMean = Mean.longValue();
                String StrMean = LongMean.toString();
                messageArea.append("**          Mean
                    =             "+(Mean/100)+"\n\n");
                messageArea.append("@@          Mean time : "
                          +getCurrentTimestamp()+"\n\n");
                //Encrypting the Mean
                Encryption(StrMean,messageArea);
                String EMean = MeanA+" "+MeanB;
                /*
                 *Sending Mean to Cloud
                 */
                try {
```

```
                    hash = Hash(EMean);
                    } catch (NoSuchAlgorithmException e) {
                            e.printStackTrace();
                    }
                messageArea.append("Hash : "+hash+"\n");
                String output = EMean+" "+hash+"
                    "+getCurrentTimestamp().toString();
                messageArea.append("Unencrypted message :
                    "+output+"\n");
                //Encrypting the Encrypted Mean with TripleDES key
                Eoutput = TripleDESencrypt(output,cipher);
                messageArea.append("Encrypted message :
                    "+Eoutput+"\n\n");
                //Sending encrypted Mean to Cloud
                out.println("MEAN"+" "+Eoutput);
                messageArea.append("@@          Computed MEAN
                    sent at        "+
                    getCurrentTimestamp()+"\n\n");
            }else{
                    messageArea.append("Timestamp exceeds the time
                        limit\nPossibility of a REPLAY attack\n\n");
            }
        }else{
                    messageArea.append("Hash is incorrect\nWaiting
                        for Resending....");
                    out.println("INCORRECT_HASH_MEAN");
        }
    }else if(line.startsWith("STAT_Para")){
        /*
         * Computing Variance, Skewness and Kurtosis
         */
              messageArea.append("@@       Answers received at
                        "+getCurrentTimestamp()+"\n\n");
              ts = getCurrentTimestamp();
              String EAns[] = line.split(" ");
              String Ans = TripleDESdecrypt(EAns[1]);
        String Ans1[] = Ans.split(" ");
        try {
```

```
                hash = Hash(Ans1[0]+" "+Ans1[1]+" "+Ans1[2]+"
                    "+Ans1[3]+" "+Ans1[4]+" "+Ans1[5]);
            } catch (NoSuchAlgorithmException e) {
                e.printStackTrace();
            }
        //Checking for HMAC
        if ((hash.compareTo(Ans1[6])) == 0){
                messageArea.append("STAT Para Hash is
                    correct\n\n");
                out.println("CORRECT_HASH");
                //checking for time stamp time
                if((checkTimestamp(Ans1[7]+" "+Ans1[8], ts,
                    messageArea)).intValue() < Time_Limit){
                    messageArea.append("Timestamp is within
                        the time limit\n\n");
                    BigInteger VarianceA = new
                        BigInteger(Ans1[0]);
            BigInteger VarianceB = new BigInteger(Ans1[1]);
            BigInteger SkewnessA = new BigInteger(Ans1[2]);
            BigInteger SkewnessB = new BigInteger(Ans1[3]);
            BigInteger KurtosisA = new BigInteger(Ans1[4]);
            BigInteger KurtosisB = new BigInteger(Ans1[5]);
            //Computing Variance
            Double Variance =
                Decryption(VarianceA,VarianceB,p,x).doubleValue()/10000;
            Variance = (Variance / denomenator);
            /*
             * Displaying Answers
             */
            messageArea.append("**********************************
                A N S W E R S
                **********************************\n\n");
            messageArea.append("**       Addition
                =              "+(Addition/100)+"\n");
            messageArea.append("**       Mean           =
                "+(Mean/100)+"\n");
            System.out.println("Variance Answer = "+Variance);
            messageArea.append("**       Variance
                =              "+Variance+"\n");
```

```
Timestamp ts_var = getCurrentTimestamp();
//Computing Standard Deviation
Double StandardDeviation =
    Math.sqrt(Variance.doubleValue());
messageArea.append("**       Standard Deviation
    =              "+StandardDeviation+"\n");
Timestamp ts_sd = getCurrentTimestamp();
//Computing Skewness
Double Skewness = Decryption(SkewnessA, SkewnessB, p,
    x).doubleValue()/1000000;
Skewness = ( Skewness /
    Math.pow(StandardDeviation,3));
messageArea.append("**       Skewness
    =              "+Skewness+"\n");
Timestamp ts_skew = getCurrentTimestamp();
//Computing Kurtosis
Double Kurtosis = Decryption(KurtosisA, KurtosisB, p,
    x).doubleValue()/100000000;
Kurtosis = ( Kurtosis / Math.pow(StandardDeviation,
    4));
messageArea.append("**       Kurtosis
    =              "+Kurtosis+"\n\n");
Timestamp ts_kur = getCurrentTimestamp();
messageArea.append("**********************************"
+"************************************************\n\n");
//Displaying timing values
messageArea.append("@@@      Variance time  :
    "+ts_var+"\n");
messageArea.append("@@@      StandardDeviation time
    :     "+ts_sd+"\n");
messageArea.append("@@@      Skewness time  :
    "+ts_skew+"\n");
messageArea.append("@@@      Kurtosis time  :
    "+ts_kur+"\n");
messageArea.append("@@       Session "+sessionNum+"
    Concluding time :
    "+getCurrentTimestamp()+"\n\n\n\n\n");
                //Sending the concluding signal to
                    cloud server
```

```
                    out.println("FINISH");
                            }else{
                                    messageArea.append("Timestamp exceeds
                                        the time limit\nPossibility of a
                                        REPLAY attack\n\n");
                            }
                    }       else    {
                            messageArea.append("Hash is incorrect\nWaiting
                                for Resending....");
                            out.println("INCORRECT_HASH_VARIENCE");
                    }
            }
            else{
                 System.out.println(input);
            }
        }
    }
    /**
     * Main Method of the Program
     * */
    public static void main(String[] args) throws Exception {
       Analyzer analyzer = new Analyzer();
       analyzer.frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
       analyzer.frame.setVisible(true);
       analyzer.run();
    }
```