Norwegian University of
Science and Technology

Multiple Escrow Agents in VoIP

Abdullah Azfar

Master in Security and Mobile Computing
Submission date: June 2010
Supervisor: Steinar Andresen, ITEM
Co-supervisor: Prof. Gerald Q. Maguire Jr., Royal Institute of Technology (KTH), Stockholm, Sweden

Norwegian University of Science and Technology
Department of Telematics

Problem Description

The introduction of a key escrow agent in Voce over IP (VoIP) communication ensures that the Law Enforcements Agencies (LEA) can retrieve the session key used to encrypt data between two users. A masters thesis by Romanidis Evripidis titled Lawful Interception and Countermeasures: In the era of Internet Telephony , Royal Institute of Technology (KTH), School of Information and Communication Technology, COS/CCS 2008-20, September 2008 http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/080922-Romanidis_Evripidis-with-cover.pdf addressed the issues of key escrow. The escrow agent stores the session key and some related data. The LEA is assumed to have recorded a communication session between two users. The escrow agent reveals the key to the LEA upon a legal request from the LEA.

However, the use of a single escrow agent can have drawbacks. First of all a fraudulent request by an evil employee from the LEA can lead to improper disclosure of a session key. After the escrow agent reveals the key the evil person could fabricate data according to his needs and encrypts it again (using the correct session key). In this situation the persons involved in the communication session can be accused of crimes that he or she or they never committed. This problem can be overcome by signing the hashes of the data with the user s private key and storing the final hash with the escrow agent. This proposed architecture is being implemented in a master thesis by Md. Sakhawat Hossen at the Department of Communication Systems (CoS) at the Royal Institute of Technology (KTH).

The problems with a single escrow agent still exists as any evil person either in the LEA or in the escrow agent can reveal the session key. Again, a failure of the escrow agent due to technical reasons can delay or even make it impossible to reveal the session key, thus the escrow agent might not be able to comply with a lawful court order or comply with their escrow agreement in the case of data being released according to this agreement (for example for disaster recovery).The idea of this master thesis is to explore the question of what happens when there are multiple escrow agents in the case of VoIP.

In the case of a VoIP session, the session key will be divided into m chunks and stored with m different escrow agents. The LEA has to retrieve n-out-of-m items to recover the session key. Utilizing multiple escrow agents enhances security in following ways. First of all, no single employee of a single escrow agent can disclose the whole key. Thus there would have to be multiple evil employees to effect disclosure of a key. Secondly, any site or company might fail any time (economically fail, experience a technical failure, or experience an accident or disaster). Using an n-of-m scheme would be robust to m-n failures.

Based on the above discussion, there are some issues that need to be addressed. This master thesis project has the following goals:

1) Implement a suitable algorithm to split the session key into m chunks.
2) Implement a means to know the list of escrow agents and the role of a user for a given session.
3) Implement a session key regeneration function to regenerate the session master key from n-out-of m chunks.
4) Measure the performance of key escrow with multiple escrow agents in a working prototype.
5) Find and evaluate out a suitable error detection and error correction method to detect and correct errors in a key chunk and/or across key chunks. A key goal in this regard is to understand how error detection and correction can be used to improve the reliability and availability of the escrow agents, potentially changing the choice of system parameters.
6) For any large organization where hundreds, thousands, or more calls are generated each hour, what could be the most suitable way to escrow the keys and data? Should it be done after each call? Or should it be done once per day by sending bulk transfer? What could be the performance bottlenecks?

Assignment given: 18. January 2010
Supervisor: Steinar Andresen, ITEM

Abstract

Using a Key escrow agent in conjunction with Voice over IP (VoIP) communication ensures that law enforcements agencies (LEAs) can retrieve the session key used to encrypt data between two users in a VoIP session. However, the use of a single escrow agent has some drawbacks. A fraudulent request by an evil employee from the LEA can lead to improper disclosure of a session key. After the escrow agent reveals the key this evil person could fabricate data according to his/her needs and encrypt it again (using the correct session key). In this situation the persons involved in the communication session can be accused of crimes that he or she or they never committed. The problems with a single escrow agent becomes even more critical as a failure of the escrow agent can delay or even make it impossible to reveal the session key, thus the escrow agent might not be able to comply with a lawful court order or comply with their escrow agreement in the case of data being released according to this agreement (for example for disaster recovery).

This thesis project focused on improving the accessibility and reliability of escrow agents, while providing good security. One such method is based on dividing the session key into m chunks and escrowing the chunks with m escrow agents. Using threshold cryptography the key can be regenerated by gathering any n-out-of-m chunks. The value of m and n may differ according to the role of the user. For a highly sophisticated session, the user might define a higher value for m and n for improved, availability, reliability, and security. For a less confidential or less important session (call), the value of m and n might be smaller. The thesis examines the increased availability and increased reliability made possible by using multiple escrow agents.

Key words: Key escrow, VoIP, Law Enforcement Agency, Multiple Escrow Agents, Threshold Cryptography, Reliability, Availability, Shamir's Secret Sharing.

Table of Contents

Abstract .... **i**

Table of Contents .... **ii**

List of Figures .... **v**

List of Tables .... **vi**

List of Listings .... **vii**

Acknowledgements .... **viii**

Abbreviations .... **ix**

Chapter 1: Introduction **1**

1.1 Motivation *1*

1.2 Thesis Overview *2*

1.3 Goals of the Thesis Project *3*

1.3.1 Splitting the Key 4
1.3.2 Where are the Keys Escrowed 5
1.3.3 Regenerating the Session Master Key 6
1.3.4 Reliability and Availability Issues 6
1.3.5 Error Detection and Error Correction 7

Chapter 2: Background **9**

2.1 Escrow Agent *9*

2.2 Key Escrow *9*

2.2.1 Vulnerabilities and Risks of Key Escrow 10
2.2.2 Multiple Escrow Agents 12

2.3 Real Time Transport Protocol (RTP) *12*

2.3.1 RTP Header Format 12

2.4 Real Time Transport Control Protocol (RTCP) *13*

2.4.1 RTCP Services 13
2.4.2 RTCP Message Types 14

2.5 Secure Real Time Transport Protocol (SRTP) *14*

2.6 Multimedia Internet Keying (MIKEY) *15*

2.7 Minisip *16*

2.8 Clipper Chip *16*

2.9 Threshold Cryptography *17*

2.10 Shamir's Secret Sharing *18*

2.11 Error Detection *20*

2.11.1 Repetition Codes 20

2.11.2 Checksum 21
2.11.3 Cyclic Redundancy Check (CRC) 21

2.12 Error Correction 22
2.12.1 Convolutional Code 22
2.12.2 Hamming Code 24
2.12.3 BCH Codes 26
2.12.4 Reed-Solomon Codes 26

Chapter 3: Related Work 27

3.1 SIP User Agent with Key Escrow 27

3.2 VoIP Lawful Interception 28

Chapter 4: Design and Implementation of Split Operation 29

4.1 Proposed Solution for Splitting the Key 29

4.2 What to Split and Escrow 30

4.3 Which Escrow Agents are Involved in a Session 31

4.4 Escrow Agents and Escrow Databases 32

4.5 Implementation Details 32
4.5.1 General Algorithm for Split and Escrow Operation 32
4.5.2 Storing the Key Chunks into Files 33
4.5.3 Dividing the TGK and Invoking the Split Function 34
4.5.4 URL Formation and Escrow Operation 34
4.5.5 How to Escrow 35
4.5.6 The Split Function 37

Chapter 5: Design and Implementation of Combine Operation 38

5.1 General Algorithm for Retrieving the Key chunks and Combining Them 38

5.2 Escrow Agent Support for LEA 39

5.3 The Combining Operation 42

Chapter 6: Performance Evaluation and Discussion 44

6.1 Experimental Setup for Escrow Operations 44

6.2 Time Required to Split the Base 64 TGK Value into Chunks 45

6.3 Escrow Time when all 5 Escrow Agents are Available 48

6.4 Escrow Time when 4 out of 5 Escrow Agents are Available 50

6.5 Escrow Time when 3 out of 5 Escrow Agents are Available 52

6.6 Escrow Time with Delay 54
6.6.1 Manipulating the Traffic Delay for a Specific Link 54
6.6.2 Escrow Time with Delay in 1 Escrow Agent 54
6.6.3 Escrow Time with Delays for Two Escrow Agents 56
6.6.4 Escrow Time with Delay for Three the Escrow Agents 57
6.6.5 Comparison of Escrow Time with Traffic Delay to Different numbers of Escrow Agents 58

6.7 Escrow Time with Different Numbers of randomly Available/Unavailable Escrow Agents *60*
6.7.1 Escrow Time with 1 Randomly Available/Unavailable Escrow Agent 60
6.7.2 Escrow Time with 2 Randomly Available/Unavailable Escrow Agents 61
6.7.3 Escrow Time with 3 Randomly Available/Unavailable Escrow Agents 62
6.7.4 Comparison of Escrow Times with 1, 2 and 3 Randomly Available/Unavailable Escrow Agents 63
6.8 Availability Measures for LEA *64*
Chapter 7: Conclusions and Future Work **67**
7.1 Summary of Achievements *67*
7.2 Future Work *68*
References **70**
Appendices **73**
A. Generating Self-Signed Certificatess on Ubuntu 9.10 .... **73**
B. SSL Enabling Script for Apache Server in OpenSuse 10.3 .... **75**
C. Shamir's Secret Sharing Algorithm Code (Common parts: To be added in both the User Agent and the LEA) **78**
D. Shamir's Secret Sharing Algorithm Code (To be added only in the User Agent) **84**
E. Shamir's Secret Sharing Algorithm Code (To be added only in the LEA) **87**
F. Configuration of the CPU Used by the User Agent **91**
G. Escrow Database Schema Definition **92**
H. CPU Clock Resolution **97**
I. SIP Express Router (SER) Configuration File **98**

List of Figures

Figure 1-1: Splitting and regenerating the session key ... 5
Figure 2-1: RTP header format ... 13
Figure 2-2: SRTP packet format ... 15
Figure 2-3: Law Enforcement Access Field (LEAF) ... 17
Figure 2-4: Shamir's secret sharing with two points (S_j and S_k; where j is not equal to k) ... 19
Figure 2-5: Example of a convolution encoder ... 23
Figure 4-1: General architecture of m Escrow agents ... 30
Figure 4-2: General Structure of the escrow databases ... 32
Figure 5-1: Login interface for LEA employee ... 39
Figure 5-2: Target information interface ... 40
Figure 6-1: Experimental setup for escrow operations ... 44
Figure 6-2: Experimental setup for split time calculation ... 45
Figure 6-3: Execution time for split function applied to the first 128 bytes of the TGK encoded in base 64 ... 46
Figure 6-4: Execution time for split function applied to the last 128 bytes of the TGK encoded in base 64 ... 47
Figure 6-5: Escrow time with 5 escrow agents working ... 49
Figure 6-6: Escrow time for 5 escrow agents shown separately ... 50
Figure 6-7: Escrow time with 4 escrow agents working ... 51
Figure 6-8: Escrow time for the 4 available escrow agents shown separately ... 52
Figure 6-9: Escrow time with 3 escrow agents working ... 53
Figure 6-10: Escrow time with 3 escrow agents working shown separately ... 54
Figure 6-11: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2 ... 55
Figure 6-12: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2 and escrow agent 4 ... 56
Figure 6-13: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2, escrow agent 4, and escrow agent 5 ... 57
Figure 6-14: Cumulative distribution of Escrow time with a delay of approximately 1 second in 1,2, and 3 escrow agents ... 58
Figure 6-15: Flow graph of the TCP packets from the user agent to an escrow agent ... 59
Figure 6-16: Success and failure state of escrow operation ... 60
Figure 6-17: Cumulative distribution of Escrow time with 1 randomly available/unavailable escrow agent ... 61
Figure 6-18: Cumulative distribution of Escrow time with 2 randomly available/unavailable escrow agents ... 62
Figure 6-19: Cumulative distribution of Escrow time with 3 randomly available/unavailable escrow agents ... 63
Figure 6-20: Escrow time with 1, 2, and 3 randomly available/unavailable escrow agents ... 64

List of Tables

Table 2-1: Next state table 24
Table 2-2: Output values 24
Table 2-3: Parity calculation in Hamming code 25
Table 2-4: Parity bits for data 1010 25
Table 2-5: Error detection in Hamming code 25
Table 6-1: Statistical data to split first half of TGK encoded in base 64 47
Table 6-2: Statistical data to split second half of TGK encoded in base 64 48
Table 6-3: Statistical data for escrowing data (all 5 escrow agents available) 49
Table 6-4: Statistical data for escrowing data (4 out of 5 escrow agents available) 51
Table 6-5: Statistical data for escrowing data (3 out of 5 escrow agents available) 53
Table 6-6: Statistics concerning the time required to escrow data with delay of roughly 1 second by one of the escrow agents 56
Table 6-7: Statistics concerning the time required to escrow data with delay of roughly 1 second by two of the escrow agents 57
Table 6-8: Statistics concerning the time required to escrow data with delay of roughly 1 second by three of the escrow agents 58
Table 6-9: Statistics concerning the time required to escrow data with 1 randomly available/unavailable escrow agent 61
Table 6-10: Statistics concerning the time required to escrow data with 2 randomly available/unavailable escrow agents 62
Table 6-11: Statistics concerning the time required to escrow data with 3 randomly available/unavailable escrow agents 63

List of Listings

Listing 4-1: Creating five different files to store the key chunks 33
Listing 4-2: Dividing the TGK into two parts and invoking split() function 34
Listing 4-3: Formation of the URL where the top gray colored area shows how the parameters are invoked and the lower yellow colored area shows the formation of the URL 35
Listing 4-4: Invocation of the libcurl method 36
Listing 5-1: PHP script to fetch information from escrow agents 41
Listing 5-2: Invocation of combine() function 43

Acknowledgements

First of all, I would like to thank my supervisor **Professor Gerald Q. Maguire Jr.** for his valuable feedback, support and suggestions throughout the thesis project. Without his personal involvement and intervention at critical stages, it would have been very challenging to complete the project. He provided critical and useful suggestions on how to approach the research problem systematically. I find myself fortunate to have him as my supervisor.

I would also like to thank my home university supervisor **Professor Steinar Hidle Andresen** in Norwegian University of Science and Technology (NTNU) for his helpful suggestions during the initial phase of the project.

Additional appreciation is extended to **B. Poettering, Erik Eliasson**, **Md. Sakhawat Hossen** and **Muhammad Sarwar Jahan Morshed** for their valuable support and constructive feedback during the project.

Last but not the least, I would like to thank my family in Bangladesh for their unconditional support throughout this journey. My always encouraging father has been a great source of inspiration throughout my life. And finally, I would like to dedicate this thesis project to my mother, whose blessings are always with me.

Abbreviations

AES	Advanced Encryption Standard
BCH	Bose – Chaudhuri – Hocquenghem
CNAME	Canonical Name
CRC	Cyclic Redundancy Check
CSB	Crypto Session Bundles
CSRC	Contributing Source
DCA	Distributed Certificate Authority
DH	Diffie-Hellman
FEC	Forward Error Correction
GF	Galois Field
GUI	Graphical User Interface
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol secure
IP	Internet Protocol
LEA	Law Enforcement Agency
LEAF	Law Enforcement Access Field
LI	Lawful Interception
MIKEY	Multimedia Internet Keying
MKI	Master Key Identifier
PKE	Public Key Encryption
PSK	Pre-Shared key
RSA	Rivest Shamir Adleman
RTCP	Real Time Transport Control Protocol
RTP	Real Time Transport Protocol
SDP	Session Description Protocol
SER	SIP Express Router
SIP	Session Initiation Protocol
SRTCP	Secure Real Time Control Protocol
SRTP	Secure Real Time Transport Protocol
SSL	Secure Socket Layer
SSRC	Synchronization Source
TEK	Traffic Encrypting Key
TGK	TEK Generation Key
TTP	Trusted Third Party
UA	User Agent
UDP	User Datagram Protocol
VoIP	Voice over Internet Protocol
XOR	Exclusive OR

Chapter 1: Introduction

1.1 Motivation

The idea of this masters thesis project is to explore the question of what happens when there are multiple escrow agents in the case of key escrowed VoIP. In the case of a VoIP session, the session key will be divided into m chunks and stored with m different escrow agents. Using threshold cryptography the Law Enforcement Agency (LEA) only has to retrieve n-out-of-m chunks to recover the session key. Utilizing multiple escrow agents enhances security in following ways. First of all, no single employee of a single escrow agent can disclose the whole key, thus there would have to be multiple evil employees to effect disclosure of a key. Secondly, any site or company might fail any time (economic failure, technical failure, or an accident or disaster). Using an n-out-of-m scheme would be robust to m-n failures.

The communication of users who are suspected of criminal activities or threat to national security can be monitored by Lawful Interception (LI). Lawful Interception has been a burning issue for 50-60 years and still the users are not positive to LI as it might raise some controversial issues such as violation of human rights and decreased confidentiality of commercial communication. However, if the session key is escrowed to an escrow agent(s) then the LEA can capture and decrypt the session by lawful interception. Some business organizations might be interested in escrowing the session key(s) as they can present a proof of agreement in future if needed.

The LEAs can retrieve the session key used to encrypt data between two users in a VoIP session by showing legal documents to the escrow agent. But the use of a single escrow agent has some drawbacks. A fraudulent request by an evil employee from the LEA can lead to improper disclosure of a session key. After the escrow agent reveals the key the evil person could fabricate data according to his/her needs and encrypt it again. In this situation an innocent person involved in the communication session can be accused of crimes that he or she or they never committed.

The problems with a single escrow agent becomes even more critical as a failure of the escrow agent can delay or even make it impossible to reveal the session key, thus the escrow agent might not be able to comply with a lawful court order or comply with their escrow agreement in the case of data being released according to this agreement (for example for disaster recovery). The failure might be due to several reasons. The escrow agent might be unavailable at the time when the request was sent. Or the escrow agent might fail as a business for some reason. Or the escrow agent might face technical difficulties in sending the key, for example, due to a network partition between the escrow agent and the LEA.

A master thesis by Romanidis Evripidis at the Royal Institute of Technology (KTH) addressed the issues of key escrow [1]. The escrow agent stores the session key and some related data. The LEA is assumed to have recorded a communication session between two users. The escrow agent reveals the key to the LEA upon a legal request from the LEA. He pointed out that there is a problem when using a single escrow agent as any evil person either in the LEA or in the escrow agent can reveal the session key.

The problem of forgery by a single escrow agent can be overcome by signing the hashes of the data with the user's private key and storing the final hash with the escrow agent. This proposed solution has been implemented in a master thesis by Md. Sakhawat Hossen also at the Royal Institute of Technology (KTH) [2].

1.2 Thesis Overview

This thesis will address the issues of using multiple escrow agents for storing a session master key. The escrow agent must potentially store this key for a long period of time. The user or the LEA can request keys any time. It is very important from both the user's and LEA's point of view that the escrow agent is able to provide them the key in a timely fashion whenever a proper request has been made. The LEA must present the proper authentication and legal notice to the escrow agent in order to retrieve a key. If for some reason an escrow agent goes out of business, then it becomes impossible for the user and the LEA to retrieve an escrowed key. For example, a LEA might need to retrieve a session key after five years, but could find that the escrow agent who had stored the session key is no longer in business. Or it might happen that the escrow agent is not available 24 hours a day and 365 days of the year. For example, an escrow agent might only be available only during some specific times of the day, for example due to schedule maintenance or limited business hours. This can create an undesirable situation. To avoid this situation the concept of multiple escrow agents has been introduced. The session key will be divided into m chunks and will be stored with m escrow agents. The key will be split in such a way that any n chunks out of m chunks will be sufficient to regenerate the key. So as long as any n of the m escrow agents supply their part of the split key to the LEA then the key can be regenerated. This overcomes the problem of one escrow agent going out of business or being unavailable for a while.

Another scenario may occur when one escrow agent has gone out of business. If the user discovers that one of its escrow agents is no longer active, then the user might want to (re-) escrow his/her key (the key which was escrowed with the previous escrow agent) with a new escrow agent. This would not be possible in the case of a single escrow agent. But an n-out-of-m key retrieval system can support this. If a user finds one escrow agent is no longer active, then he/she can retrieve the key from any other n escrow agents, then split the key again and store the part which was stored in the inactive escrow agent into a new escrow agent.

The fact that the set of escrow agents that are used can change over time makes it clear that it is important to define a means to identify the escrow agents which are being used to store a session key. With a single escrow agent, it is easy to identify user's escrow agent (logically: one only has to look at the IP address that the escrowing event sends traffic to and compare this to a list of know escrow agents). But with multiple escrow agents, the user should publish a list of his/her escrow agents. It is an important issue how to publish this list of escrow agents and when and where to publish it. The user could publish the list of escrow agents on his/her webpage. Or this information can be sent along with the Real Time Transport Protocol (RTP) media streams. The list of escrow agents could also be attached to the last signed hash value. The LEA can capture this file and if the LEA is able to retrieve the escrowed information from any one of the escrow agents then it could find the list of all the escrow agents that were used. Since the escrowed information is sent in a TLS tunnel, the escrow agent can only get this information after producing a court order to one of the escrow agents.

The issue of error detection and correction in the data sent by the escrow agents depends on the reliability of the escrow agents. A study will be done on error detection and correction based on the estimated reliability of the escrow agents. This study might be important or unimportant depending upon the results of the first phase of the project - but can allow the effective reliability of individual escrow agents to be increased by using information from multiple escrow agents. A key issue is how a user can exploit multiple escrow agents to get performance gains, similar to the performance gains that are achieved in RAID disk arrays over individual disks.

An n-out-of-m key escrowing scheme increases the reliability and availability of the key. Availability can be defined as “the ability of a system to provide a set of services at a given instant of time or at any instant within a given time interval” [3]. The asymptotic availability is considered as the most common availability measure. It is the probability of finding a system in working state at any randomly chosen time in future. The asymptotic availability is denoted by A. Formally,

$$A = \lim_{t \to 0} P(I(t))$$

where, I(t) is a function of time describing the behaviour of the system.

$$I(t) = \begin{cases} 1 \; if \; the \; system \; is \; working \; at \; time \; t, \\ 0 \; otherwise \end{cases}$$

On the other hand reliability can be defined as “the ability of a system to provide uninterrupted service” [3]. The reliability function is defined as

$$R(t) \; = P(T_{FF} > t)$$

Where, T_{FF} is the time to first failure of the system.

1.3 Goals of the Thesis Project

Based on the thesis overview above, there are some issues that need to be addressed. This master thesis project has the following goals:

- Implement a suitable algorithm to split the session key into m chunks.
- Implement a means to know the list of escrow agents and the role of a user for a given session.
- Implement a session key regeneration function to regenerate the session master key from n-out-of –m chunks.
- Measure the performance of key escrow with multiple escrow agents in a working prototype. (The prototype will build upon the earlier thesis work [2])
- Find and evaluate out a suitable error detection and error correction method to detect and correct errors in a key chunk and/or across key chunks. A key goal in this regard is to understand how error detection and correction can be used to improve the reliability and availability of the escrow agents, potentially changing the choice of system parameters.
- For any large organization where hundreds, thousands, or more calls are generated each hour, what could be the most suitable way to escrow the keys and data? Should it

be done after each call? Or should it be done once per day by sending bulk transfer? What could be the performance bottlenecks?

1.3.1 Splitting the Key

The (session) key can be split into chunks in varieties of ways. The first approach of splitting the key is to divide the key into two equal parts. Then both of these parts can be further divided into smaller parts (for some number of cycles) and each of these final parts can be sent to different escrow agents. For an m-out-of-m key scheme this approach would offer a very simple solution. Just split the key, pad each chunk with zeros to keep the relative positions of the bits unchanged and send the results to different escrow agents. To regenerate the key, retrieve all the chunks and OR them to get the key. However, this would require that all n parts be retrieved correctly. To provide higher reliability and availability we want to implement an n-out-of-m retrieval system. In this scheme if we can only retrieve n chunks, then we need to use some error correction codes to fix the errors (due to the missing chnunks). If we can successfully correct for the missing chunks, then we can derive the original key. Threshold cryptography could be another choice, but we need to consider its cost. We can use Shamir's secret sharing scheme. This algorithm is well proven one and very reliable. The choice of n and m can depend upon the user's roles. For example, whenever a user orders a pizza, it is very unlikely that this data has to be retrieved later, thus the user can use a splitting function which has low reliability. In this case the number of escrow agents might be fewer and the failure of all escrow agents might be acceptable. However, when a user orders a large number of shares (or a large value of shares) in a company, then the user may use a splitting function which escrows the key to larger number of escrow agents and is able to tolerate more failures of escrow agents, while still providing high reliability of key recovery. This is shown in Figure 1-1.

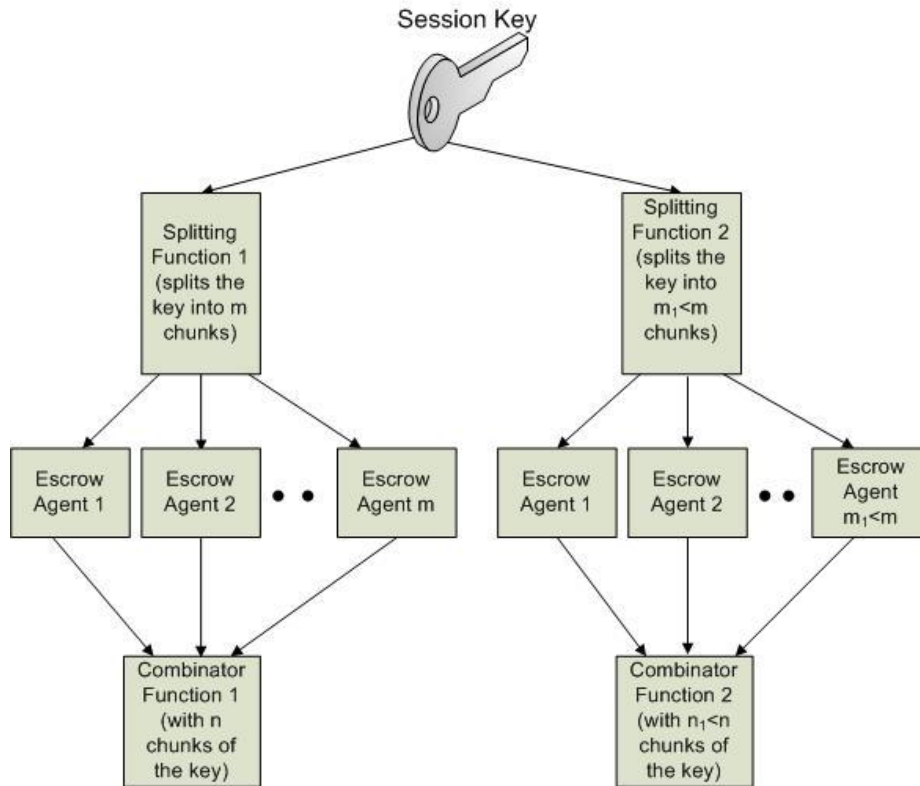

Figure 1-1: Splitting and regenerating the session key

In the libmikey library of minisip source code there is a file named Mikey.cxx. A function named escrowSessionKey() was added to the library as a public member of the Mikey class by Hossen [2]. Currently this function simply forms a URL to be used with one of the libcurl fuctions to instantiate a curl object. The function invokes a method to generate the TGK along with the pseudo-random number (Rand) and CSB ID value. After generating the parameters needed to be escrowed, we can add a splitting operation. This splitting function will split the key (and possibility the other parameters) into specified number of chunks and then send the chunks to a set of escrow agents. This raise the questions of (1) which escrow agents should chunks be sent to and (2) how does the LEA know whom these escrow agents are. These are addressed in the next subsection.

1.3.2 Where are the Keys Escrowed

A very important question to be answered in this thesis is that how can the LEA know who the escrow agents are for a user session. This question becomes even more complex if we enable the user to use different escrow agents for the different roles of a user. As from Figure 1-1, the user may use different splitting functions and different numbers of escrow agents for different sessions. How can the LEA know the role of the user, number of escrow agents for that role, and the identity of these escrow agents? This question is also valid for the user himself or herself. If the user needs to retrieve a session master key several years after the session has ended, how will they remember what role and escrow agents he/she used for that particular session?

A suitable answer to all these questions could be to send the list of escrow agents at the end of the session with the TGK, rand, CSB ID and the signed hash value of the last block to all the escrow agents. The signed hash value of the last block is already generated by Hossen's thesis [2]. The last signed hash value is sent to the escrow agent in order to detect forgery. We could just add the list of escrow agents with these parameters and send it to the escrow agents. Another possibility would be to explicitly send the role of the user and the list of escrow agents via the RTCP path. This information can be sent at any time during a session. It can be sent at the beginning, at the end, or any time during the session.

1.3.3 Regenerating the Session Master Key

The LEA captures the RTP streams and learns the user role and list of escrow agents from the captured session. Then the LEA sends requests to the escrow agents requesting them to each revealing their chunks of the session master key. The escrow agents, upon receiving legal requests from the LEA with proper authorization, sends their chunk of the key to the LEA. The LEA then regenerates the key using the received chunks. In order to regenerate the key, the LEA should receive at least n-out-of-m chunks from the escrow agents. The ratio of n/m should be low enough so that if some escrow agents fail to respond, that the key can still be regenerated. Again, the ratio of n/m should be high enough so that the key cannot be regenerated with few chunks. This ensures that even if few of the escrow agents have corrupted employees, that the key is still safe, because the algorithm requires at least n chunks to regenerate the key.

Depending on the role of the user, the number of chunks needed to regenerate the keys varies. When the user has more confidential data to communicate (e.g., when user plays the role of a chief executive officer of a company), then the number of chunks needed to regenerate the key must be high. This ensures that a larger number of escrow agents have to be corrupted to disclose their chunks to any illegal authority. Conversely, when the same user plays a private role when ordering pizzas, the number of chunks needed to regenerate the key might be lower. The reason behind this reduced security is it usually does not matter if the details of the pizza order is known to someone (however, the information might still be somewhat sensitive as the order might include information about the time and place of delivery, the ingredients that can not be used, the payment method and details, etc.). The scenario of regenerating the key with n or $n_1<n$ number of chunks was shown in Figure 1-1.

The implementation of this part of the thesis project will be done by extending the functionality of Morshed's [42] thesis to send requests to multiple escrow agents and to use this information to regenerate the session key. We have to develop the method to identify the list of escrow agents. Then the LEA should send the request for key to all the escrow agents. These escrow agents will send their chunks and if the LEA receives at least n chunks, then it will be able to regenerate the key.

1.3.4 Reliability and Availability Issues

Reliability and availability are related with each other. For any system, these two parameters carry great importance. In our multiple escrow agent system, the reliability and availability of all the escrow agents might not be the same. An escrow agent with higher availability and higher reliability is always preferable, but in real world this is not always possible or affordable. All the escrow agents will not have same reliability and availability.

Some escrow agents might have high availability but low reliability. Conversely, some of the escrow agents might have high reliability but lower availability. An example will make this statement clear. Suppose we have an escrow agent which is active all through the day (24 hours), but does not maintain a proper backup system of the database. Due to a catastrophic failure it might lose all of the data that has been stored with this escrow agent. This escrow agent has a high availability but low reliability because one disaster can make the agent completely useless (i.e., after the catastrophic failure no previously escrowed information can ever be retrieved). On the other hand, another escrow agent might only be active 8 hours per day but this agent may have a strong backup system so that any catastrophic failure cannot wipe out any data. This escrow agent has lower availability but high reliability.

The number of escrow agents where the key chunks will be stored is related with the availability and reliability of the escrow agents and the costs that the user (and agents) are willing to bear. With only one escrow agent, the reliability is low. With two escrow agents as mentioned in clipper chip in section 2.8, the reliability depends on the availability of both escrow agents. A single escrow agent failure causes an escrow retrieval request to fail. However, using n-out-of-m escrow agents, the availability and reliability are expected to increase over that of one or two escrow agents. But the value of n and m to maximize reliability has to be determined. Simply increasing the number of escrow agents may not proportionately increase reliability. There may be an inflection point where continuing to increase the number of escrow agent may degrade the performance and may seriously degrade the systems price/performance ratio. This issue should be considered while determining the value of m and n.

The cost of escrowing is a very practical issue for deploying an escrow based system. Statistical data about the number of calls made by residential and corporate users will help to determine the number of keys that have to be escrowed; as well has how an escrow agent should be dimensioned. Another question to be answered is does the cost of escrowing always increases or is there a plateau (i.e. the cost remains constant after a certain increase). The cost of escrowing refers to the cost of storage and the cost of operating an escrow service (note that we may consider this a constant fixed cost and simply consider the number of escrow agents times this fixed cost plus the cost per unit of storage for the sum of all the storage that an escrow action requires).

1.3.5 Error Detection and Error Correction

There is always some noise in the communication channel. It is very likely that an error might occur in a key chunk sent by an escrow agent to the LEA. If one chunk out-of the-n required chunks has an error then unless there is redundant information which allow the LEA to recover the missing data, then the LEA will not be able to regenerate the key. The LEA has to know which chunks may have an error in order to request these chunks again. Repeated requests for key chunks consumes time, bandwidth in both directions, and increases the load on the escrow agent(s). The escrow agent may wish to employ forward error correction to minimize the requirement for repeated requests for the same data. An efficient error detection and correction code should be selected to deal with a burst error including missing chunks. Each of the n chunks will pass through the error detection and if they are correct or can be corrected, then the key will be regenerated. The error detection and correction scheme that will be used in this thesis will be decided later.

The necessity of error detection and correction in multiple escrow agent environment depends on the availability and reliability of the escrow agents. If the escrow agents have high reliability, then due to the use of TCP connections for requesting and retrieving escrowed information there is a low probability of having an error in a key chunk. In this case the error detection and correction procedure may add extra cost to the system without adding improved reliability or increasing availability of the system. On the other hand, a system with low reliability may provide an erroneous key chunk, thus the LEA may need error detection and correction in order to ensure sufficient availability of session keys.. In order to answer these questions, a study of the available data on network reliability and availability will be made.

Chapter 2: Background

Some background information is presented in this chapter. It introduces some of the key concepts that are going to be used in this thesis. We start by presenting the basic concepts of Escrow Agent and Key Escrow. In the sections 2.3 to 2.7, we introduce the concepts of Real Time Transport Protocol, Real Time Transport Control Protocol, Secure Real Time Transport Protocol, Multimedia Internet Keying and Minisip. Finally, we briefly present the concepts of Clipper Chip, Threshold Cryptography, Shamir's Secret Sharing and Error Detection and Correction techniques.

2.1 Escrow Agent

An escrow agent is a trusted third party (TTP) with whom users will store their session master key. A set of functional requirements have been specified for an escrow agent in [4]. The main task of the escrow agent is to securely store the session key and disclose the session key only to an authenticated Law Enforcement Agency (LEA) or to another party as specified in the escrow agreement with the user. Additional security related services that an escrow agent can provide include: access control, key management, or notary (non-repudiation) servers.

The escrow agent can provide services in three ways. It can provide on-line, in-line, or off-line services [4]. However, in this thesis project we will assume that the users (or actually there SIP user agents - UAs) escrow the individual session key and other information either just after the session ends or performs a batch operation after a number of session (or after a period of time). As the UA authenticates itself to the escrow agent for any escrow operation, we can see that these are on-line operations. The LEA can make either an on-line request or make an off-line request for escrowed information.

This thesis is concerned with the key management service of an escrow agent. A simple escrow agent developed by Md. Sakhawat Hossen [2] will be the basis of the work done work during this thesis project. This thesis project focuses on using multiple escrow agents and the session master key will be split into these escrow agents.

To escrow the session key with the TTP we have used a third party application programming interface (API) named "libcurl" which is a free and easy-to-use client-side URL transfer library supporting HTTP, HTTPS, and many other protocols. We use the HTTPS protocol to securely escrow our session master key with the escrow agent. Technical details of the libcurl library can be found in [5].

2.2 Key Escrow

The term key escrow refers to storing the cryptographic key with a TTP or escrow agent [6]. The cryptographic keys which are needed to decrypt data can be escrowed. Key escrow is generally done based on an agreement with the escrow agent. Based upon this agreement the key should be revealed only to authorized parties upon proper authentication. The authorized party may be a government or law enforcement agency (LEA) representative who has the legal authority to access the content of encrypted communication.

It is very important that the keys are only disclosed to a party with proper authentication and access rights. Disclosing the key to any other entity can lead to improper (or in some cases illegal) disclosure of data. The legal authority can be a LEA or another government

organization. If the depositor of the key has lost the key, the key might also be disclosed to him/her - again only if the depositor correctly identifies them self and is authenticated.

Key escrowing has been a burning issue for the past few decades. Illegal or corrupt bodies might acquire the key from the escrow agent in order to learn personal data or even forge (fake) data. A person might be accused of crimes based upon forged data. The United States government introduced the Clipper chip with an escrow mechanism (discussed in detail in section 2.8) to escrow session keys in such a way that they could be made available to LEAs. However, this proposed solution failed due to some reasons. The Clipper Chip was designed and manufactured, and then tested by a number of users, so its failure was an expensive one.

2.2.1 Vulnerabilities and Risks of Key Escrow

Whenever a key is escrowed, there arises a question of the security of this escrowed key. Whether the escrow agent is trustworthy? What are the chances of the key being reveled to unauthorized person/organization? Can an intruder obtain the key from the escrow agent? And so on. It is clear that key escrow introduces new vulnerabilities and risks. No matter what actions the escrow agent takes there might always be a vulnerable path to unauthorized recovery of data. Some of the vulnerabilities and risks of key escrow according to H. Abelson et al. [7] are discussed below.

Recovery of plaintext If the key is escrowed then there is always a possibility of recovering the encrypted data. This gives a new point of attack for the intruders. As the key is being escrowed, intruders can try to intercept this key in order to decrypt the data. As long as this key is available somewhere there is a risk that it can be obtained and used to decrypt the data.

Insider abuse The escrow agent is expected to be a trusted third party. However, these escrow agents are designed, implemented and maintained by human beings. Due to human nature, an employee of the escrow agent could be corrupted and reveal the key to an evil person. On the other hand, the authorized agency (LEA) might itself have some corrupt employee(s) who would issue fake orders to the escrow agent to reveal a key. Once the escrow agent reveals the key the corrupted government employee could decrypt data and/or forge data using this key. The employee might fabricate data in order to accuse an innocent person of a crime based upon fabricated evidence.

New targets for attack If there is only a single escrow agent and this agent stores the keys in a single database, then attackers will target this database. If they can gain access to this database then they might gain access to all the keys. This problem can be mitigated by distributing the keys among multiple escrow agents. Unfortunately, the introduction of multiple escrow agents will introduce additional cost.

Removal of forward secrecy

The introduction of key escrowing removes the forward secrecy of data. Forward secrecy refers to the inability to recover the session data after the session ends even though the key of the next session is disclosed. It means that for each session there will be a unique key and that unique key will be destroyed at the end of the session. This makes it impossible to retrieve the session data subsequently. The session data can be recovered during the session (as the key is available during the session). Forward secrecy improves secrecy and reduces cost of a system, because the key does not need to be stored and if the key is not stored it can not be disclosed later! But with key escrow, the session key is stored with the escrow agent, thus the session data might be decrypted any time in the future using this key. This reduces secrecy and also adds the additional cost of storing the key.

Scaling

One issue that needs to be mentioned here is scaling. There will be billions of users of VoIP in future. With key escrowing, the session key for each call for each user has to be stored. It will require enormous storage capability to store the keys. The main issue is that the keys will be stored for some lifetime. Depending upon local regulations and user desires each key might be stored for thirty, forty or even fifty years. Thus the amount of data stored will be huge , but fortunately the disk storage capacity is inexpensive - but the reliable storage of this data while avoiding improper disclosure is not a simple matter.

Distinguishing different keys

Not all keys are important to the users. Some keys are relatively less speaking important, thus the user might want to destroy the key after a short while. For example, a user orders ten pizzas and gets the delivery in time. The user might not need this data any more. However, if a user orders hundred thousand dollars worth of shares in a company, then he or she might want to store information about this transaction for a longer period. A problem for key escrow is that how to differentiate these user roles and their corresponding requirements. One approach is that the user may login with a different user identity and perform tasks with this identity. In this case the escrow agent can identify the user identity and thus could identify that the user is using. This many cause problems when the user does not know what role they should have when they start a session. For example, when answering a call the user might not know the purpose of the call and hence not know the role that they are going to have until some point later in the call. The user might even change

roles during a session, for example, exchanging familiar greetings before getting down to the business purpose of a call.

2.2.2 Multiple Escrow Agents

The problems with a single escrow agent becomes critical as a failure of the escrow agent due to technical or business reasons can delay or even make it impossible to reveal the session key when a valid and lawful request is made. The single escrow agent could have a corrupted employee who might reveal the key to any evil person. All these problems lead to the idea of utilizing more than one escrow agents to overcome the shortcomings of a single escrow agent.

In this thesis project the session key will be divided into m chunks and will be stored with m escrow agents. The key will be split in such a way that any n chunks out of m chunks can be used to regenerate the key. As a result, if any n of the m of escrow agents supply their part of the split key to the LEA then the key can be regenerated. This overcomes the problem of one escrow agent going out of business or being temporarily unavailable for some time. A corrupted employee from any single escrow agent cannot disclose the whole key because at least n chunks are needed to regenerate the key; thus raising the threshold for corruption of employees of the key escrow agents to at least one employee at each of n different escrow agents. Additionally, if any escrow agent is unavailable, delay or failure to retrieve the key are avoided as long as n-out-of-m escrow agents are available.

2.3 Real Time Transport Protocol (RTP)

The real time transport protocol (RTP) provides end-to-end network transport functions for real time data, such as audio and video over the Internet [8]. RTP is widely used for providing VoIP services. RTP supports unicast or multicast network services. RTP provides the services of payload type identification, sequence numbering, timestamping and delivery monitoring. RTP runs on top of User Datagram Protocol (UDP) to make use of the multiplexing and checksum services provided by UDP.

RTP does not guarantee ordered delivery of the packets. A sequence number is used with the packets. The receiver uses this sequence number to reconstruct the sequence. For each multimedia stream, a separate RTP session is established. For example, separate sessions are established for audio and video streams.

RTP supports a range of multimedia formats such as H.264, MPEG-4, MJPEG, MPEG, etc and allows new formats to be added without revising the RTP standard. The generic RTP header does not include the information required by specific applications. It only includes the RTP profiles and payload formats for different application types.

2.3.1 RTP Header Format

The first twelve octets in the RTP header are present in every RTP packet. Optional headers may be present after that. Then it is followed by RTP payload. The fields in the RTP header are shown in Figure 2-1. The 2 bit version (Ver) field indicates the version of the protocol. The padding (P) bit is used to indicate the extra padding bytes at the end of the RTP packet. The extension bit (X) indicates presence of an extension header. The 4 bit CSRC count (CC) field indicates the number of contributing source (CSRC) identifiers. The

contributing sources are the sources who participate in a media stream. The marker (M) bit is used to indicate special relevance for the application. The 7 bit payload type (PT) field indicates the format of the payload. The 16 bit sequence number field is incremented by one for each RTP data packet sent. The receiver uses this sequence number to identify missing or out of sequence packets. The 32 bit timestamp field is used to enable the receiver to play back the received samples at appropriate intervals. The 32 bit synchronization source (SSRC) identifier indicates the source of a stream. The 32 bit contributing source identifier indicates the sources of a stream which is generated by multiple sources.

0-1	2	3	4-7	8	9-15	16-31
Ver	P	X	CC	M	PT	Sequence number
Timestamp						
SSRC identifier						
CSRC identifiers (optional)						

Figure 2-1: RTP header format

2.4 Real Time Transport Control Protocol (RTCP)

RTCP provides control information for an RTP flow. RTCP is a sister protocol of RTP. RTCP carries control information for an RTP session, but it does not transfer any media stream. RTCP messages are sent over one port number higher than RTP packets. RTCP periodically sends statistical information to the participants of a multimedia session.

The fixed part of the RTCP packet format is similar to RTP packet format as shown in Figure 2-1. The fixed part is followed by a variable length structured elements. The variable length part should end in a 32 bit boundary in order to align the packets. A compound RTCP packet can be created by concatenating multiple RTCP packets without any prevailing separators. The compound RTCP packet can be sent as a single packet to the lower layer.

2.4.1 RTCP Services

RTCP provides four types of services [8]. They are:

- RTCP gathers quality related information during a session and sends this data to the participants of the session. An example of quality related information can be the flow control and congestion control information. The participants can adaptively change the data sending and receiving rate based on this information.
- RTCP assigns a canonical name (CNAME) for the sources participating in a session. The SSRC identifier of a participant might change due to a conflict. The receivers need to keep track the source identification for uninterrupted media stream. The CNAME provides the receivers the facility of keeping track of the sources.
- RTCP information is sent by all the participants of the session. The RTCP traffic increases with the increase of participants in a session. In a multicast session, thousands of participants might be involved and network congestion may arise due to excessive number of RTCP traffic. Each participant can observe the number of RTCP packets. This observation can be used to avoid network congestion. The participants can dynamically control the frequency of report transmissions based on the existing

RTCP traffic. A general rule is that RTCP traffic should not consume more than 5% of the total bandwidth.

- The fourth service provided by RTCP is an optional service. RTCP can provide some session control information as displaying the participant information in the user interface.

2.4.2 RTCP Message Types

RTCP carries a variety of control information. For this reason, RTCP messages can be divided into several types.

- Sender Report (SR): The sender report is sent periodically by the active senders of a session to report transmission and reception statistics. The statistics includes the information of all RTP packets sent during the interval. A timestamp is included in the sender report for the receiver to synchronize the RTP messages.
- Receiver Report (RR): The receiver report includes statistics and information of the passive participants of the session. This report is sent to the active participants.
- Source Description (SDES): The source description message includes information about the sources of the session. It includes the CNAME of the session participants. It may also include additional information as the name, e-mail address, telephone number, etc.
- BYE: The BYE message indicates the end of participation. A source sends a BYE message to shut down a stream.
- Application Specific Function (APP): RTCP can provide additional services for specific applications. The application specific message provides mechanism to design services for various applications.

2.5 Secure Real Time Transport Protocol (SRTP)

The Secure Real Time Transport Protocol (SRTP) provides security enhancements to RTP. SRTP adds authentication, integrity, confidentiality and replay protection to the RTP and RTCP traffic for both unicast and multicast applications [9]. SRTP defines cryptographic transforms and key management schemes. SRTP has a sister protocol named Secure Real Time Transmission Control Protocol (SRTCP) that provides the same security to RTCP as SRTP provides to RTP. The security services provided by SRTP are optional and independent from each other. But SRTCP message authentication is mandatory because it keeps track of the session participants, maintains packet sequence number and provides feedback to the RTP senders. SRTP is independent of the underlying transport protocol. Thus SRTP can protect RTP transported over UDP, TCP, or any other transport protocol. SRTP is defined as a profile of RTP. SRTP profile is an extension to the RTP audio/video profile.

The packet format of SRTP is shown in Figure 2-2. The packet format is same as RTP with two additional fields. The first field is a variable length optional Master Key Identifier (MKI) field. The MKI field is used by the key management protocol to determine the master key used to derive the session keys to encrypt the packets. The other field is an authentication tag which is optional but recommended. The authentication tag has a configurable length and provides authentication of both the RTP header and payload. This field also indirectly provides replay protection by authenticating the sequence number of the packets.

The encrypted portion of the SRTP packet consists of the encryption of the RTP payload and the RTP padding (if present). The authenticated portion of the SRTP packet consists of

the RTP header and the encrypted portion. SRTP uses the Advanced Encryption Standard (AES) for encryption/decryption of the RTP packet's payload to provide confidentiality. A 128 bit block is encrypted with a 128 bit key. The authentication of the RTP packet is based upon a key that is derived from the same master key that is used to encrypt the RTP payload.

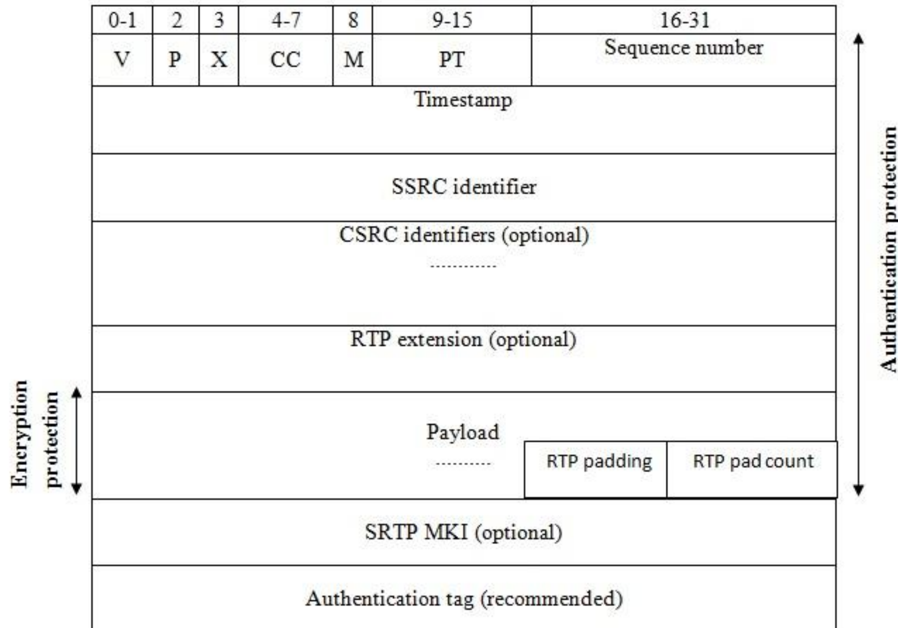

Figure 2-2: SRTP packet format

2.6 Multimedia Internet Keying (MIKEY)

MIKEY is a key management protocol used for real time applications. It supports both peer-to-peer communication and group communication. MIKEY is mainly used to support SRTP. An RTP session consists of different type of media data. The different types of media data may need different keying mechanisms. MIKEY is designed to minimize the delay of creating different keying sessions for different media data. The goal of MIKEY is to provide a single keying session for a number of multimedia sessions. This helps reduce the cost of time and storage.

A collection of one or more cryptographic session is known as crypto session bundles (CSB) [10]. The CSBs have TEK Generation Key (TGK) which is a bit-string agreed upon by two or more parties. From the TGK, Traffic-encrypting Keys (TEK) can be generated. To set up a CSB at first, a set of security parameters and TGKs are agreed upon for the CSB. Then the TGK is used to derive a TEK for each Crypto Session. This TEK acts as the session master key for SRTP.

MIKEY provides three different variants of key agreements: pre-shared key (PSK), public key encryption (PKE), and Diffie-Hellman (DH) exchange. The PSK method uses symmetric encryption. An individual key has to be shared with every single peer. This is the most efficient way to handle the transport of the common secret because only a small amount of data has to be exchanged. The PKE method is similar to the pre-shared method. Each peer requires a pair of public/private keys for encryption and signature. In DH method, both peers need to have public/private key pairs for signatures in order to authenticate each other. The public/private key pairs also protect against a man-in-middle attack. This method is more

costly in terms of time and storage due to increased number of public key operations. But the advantage of this method is that it provides both greater flexibility and perfect forward secrecy.

2.7 Minisip

Minisip is an open source Voice over Internet Protocol (VoIP) user agent (UA) [11]. It is based on the Session Initiation protocol (SIP) and special security features are included in it. Minisip was initially developed in KTH. As it is open source software, a community from both universities and companies continue working on it.

Minisip is divided into four independent subsystems. The graphical user interface (GUI) subsystem, policy subsystem, media subsystem and sip subsystem. The GUI subsystem is responsible for interacting with the user (initiating and answering calls). The policy subsystem is responsible for making decision about whether an incoming call should make the user phone generate an alert or if it should be ignored. The media subsystem is responsible for sending and playing media streams during a call. The SIP signalling logic is implemented by sip subsystem. Details about these subsystems can be found in the licentiate thesis of Erik Eliasson [12]. The most important feature of minisip is that it supports security for VoIP. Minisip provided the first public implementation of Secured Real Time Transport Protocol (SRTP) [13] and the first public implementation of Multimedia Internet Keying (MIKEY) [14]. Minisip consists of five libraries. The libminisip library implements media, policy and sip subsystems using the other four libraries. The minisip library implements a command line based and a graphical user interface. The libmsip library implements the Session Initiation Protocol (SIP) according to [15]. The libmikey library Implements MIKEY for authenticated key exchange. The libmutil and libmnetutil library implements cross-platform support for threads, mutexes, semaphores, network related functions, and other utility classes.

2.8 Clipper Chip

The Clipper chip was developed with the intention of protecting private communication as well as providing a means for the government for key escrowing [16]. The key for each device is divided into two parts held by two escrow agents. Clipper chip works based on a cryptographic algorithm named Skipjack [17] for transmitting information and Diffie-Hellman key exchange algorithm [18] to distribute session keys. The Clipper chip generates an 80 bit session key [19]. The session key is encrypted with the device's secret key and set in a LEA field (LEAF) sent at the start of a session. Whenever a LEA needs access to a session key, the LEA provides the device's serial number and a court order to both the escrow agents. After both escrow agents have provided their responses, the LEA generates the master key for the device by XORing the two parts together. Using this master key the LEA can decrypt the encrypted session key carried in the LEAF. Note that once the master key has been disclosed it can decrypt ***any*** session that this device has participated in..

At the start of every Clipper session, a Clipper chip sends a 128 bit string called the Law Enforcement Access Field (LEAF) to the receiver of the call. The Clipper chip encrypts the session key with the device’s unique encryption key. It then appends the serial number of the sender Clipper chip and a checksum. The government holds a master key known as family key. The data to be placed in the LEAF is then encrypted with the family key. The relationships between these different values and the LEAF are shown in Figure 2-3.

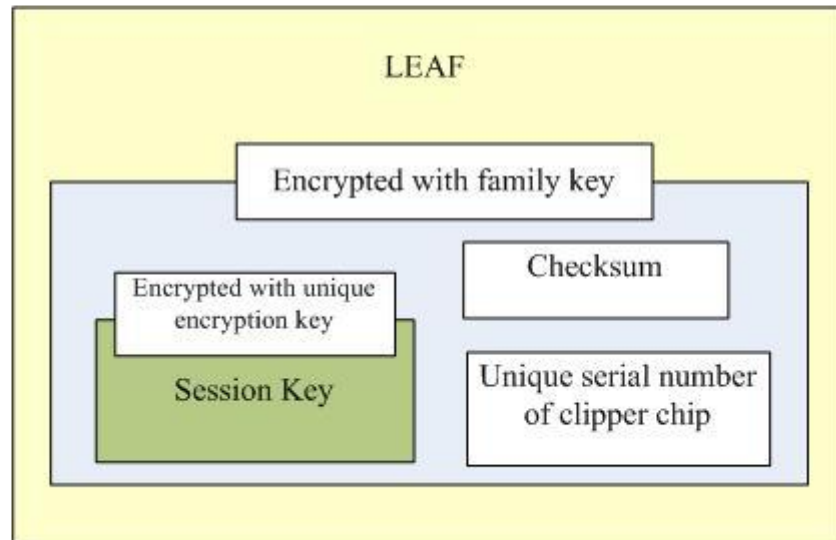

Figure 2-3: Law Enforcement Access Field (LEAF)

Whenever an LEA intercepts a conversation, it first records the session including the LEAF for this session. As the LEA has the family key from the government t can decrypt the LEAF using the family key, this reveals the serial number of the specific Clipper Chip and the encrypted session key that was encrypted with the device's unique encryption key. The LEA now contacts the two escrow agents with the serial number and their court order, then asks the escrow agents to provide their shares of the encryption key. When the escrow agents provide the LEA with the shared parts of the encryption key, it puts them together to reconstruct the device's unique key which it uses to decrypt the session key. With the decrypted session key, the LEA can decrypt now the conversation.

The Clipper chip escrow system has a serious vulnerability [21]. The LEAF field contains information necessary to recover the session key. A 16 bit hash is included to prevent forgery of the LEAF. However, this 16 bit hash is too short to protect against misuse of the Clipper Chip (as the user can generate a fake LEAF field that appears to have a valid LEAF field, but in fact if the LEA were to use this LEAF field they could not recover the session key). Another problem with the Clipper chip is that, both the escrow agents should reveal their parts of the key to the LEA. If one escrow agent fails to reveal its part or sends an erroneous part to the LEA then the key cannot be reconstituted. Additionally, all sessions with this device can now be decrypted, hence revealing much more information than a lawful intercept order might specify and compromising all future sessions with this device. As a result of criticisms of the many problems that the Clipper Chip showed, it was abandoned in 1996.

2.9 Threshold Cryptography

Threshold cryptography has been used in a Distributed Certificate Authority (DCA) scheme to protect the key against compromise [22]. The key is divided into several parts and each part is stored in different servers. For (n, m) threshold cryptography, there will be m servers with each server storing their own secret part. When a client needs to sign a message, it sends the message to these servers and the servers each partially sign the message with their keys. If at least n servers sign the message, then a valid certificate can be generated. The resulting certificate can be verified with a public key.

The secret shares in threshold cryptography do not have any explicit relation with each other. If one server is compromised by an attacker, only the secret portion stored by that server is revealed. The complete key can be generated only if n servers' keys are compromised. Additionally, if one or more servers are not functioning, the key can still be generated as long as n servers are functioning. This gives extra reliability for key retrieval. The key shares are also refreshed periodically. The shares before the refresh and after the refresh are completely different. For this reason, if an attacker reveals the secret portion stored by a server before refresh, it becomes totally useless for the attacker if a refresh operation is done.

Threshold cryptography can be applied to RSA. The RSA decryption key *d* can be split into n parts, $d= d_1+d_2+d_3+\ldots\ldots+d_n$. Signing can be done on message *m* by $s=m^{d1}.m^{d2}.m^{d3}\ldots\ldots m^{dn}$ *Mod (N),* where N is the product of two large primes. The problem with this scheme is Mod(N) has to be generated by a trusted third party [23]. The third party can easily forge the signature on a message. Boneh and Franklin [24] provided a solution where the need for a trusted third party was overcome. Unfortunately, their solution has a drawback, as it provides only m-out-of-m threshold decryption. This means that all the servers need to sign into a message to generate a valid certificate.

Rabin [25] proposed a new protocol based on Shamir secret sharing. Here the n servers can sign to create a valid signature, but they need to interact with each other. The problem with this scheme is that n servers will know additional information about each other which can cause information leaking. Shoup [26] proposed another scheme where various servers need to interact with each other only once, during the initial generation of the key. After this the servers can work separately. But the problem with this scheme is that a trusted dealer is needed which does not comply with the requirements proposed by Boneh and Franklin [24]. Nguyen [27] proposed a threshold signing scheme for RSA which does not require a trusted third party and it can provide decryption with only n servers providing their secret share. Nguyen also claims that there is no security information leak, the time and storage complexity of the protocol is linear in the number of parties, and no restriction is placed on the RSA moduli. The problem with this scheme is that the subset of servers which will provide their secret share to generate the key should be predefined.

2.10 Shamir's Secret Sharing

Shamir's secret sharing is an n-out-of-m threshold scheme based on polynomial interpolation 0. At least n participants must provide their shares in order to decrypt the secret. Any n-1 compromise of the shared secret is insufficient to decrypt the secret. Shamir's secret sharing is based on the idea that a polynomial of degree n-1 is defined by n points. For example, to define a line of degree one, it takes two points and to define a parabola of degree two, it takes three points. In order to decrypt a secret from n-out-of-m shares, a polynomial can be defined over the known finite field GF(q) (Galois field with q elements) as $f(x)=a_0+a_1x^1+a_2x^2+a_3x^3+\ldots\ldots\ldots\ldots+a_{m-1}x^{m-1}$ where the coefficient a_0 is the secret and all other coefficients are random elements in the field. Each of the m shares is a point (x_i, y_i) on the curve defined by the polynomial, where $_{xi}$ is not equal to 0. For any of the n shares the polynomial is uniquely identified and a_0 can be calculated by identifying the polynomial. But for fewer shares than n for example, n-1 shares the secret could be any element in the field.

Let us consider a simple example where two shares are required to decrypt the secret (n=2) [29]. The polynomial is a degree one polynomial which represents a line in Figure 2-4. Each

share is a point on the line and the secret is the point where the line intercepts the y axis. Thus with n=2, two points of the line can be found and the line can be interpolated to find where it intercepts the y axis. In this way the secret can be calculated.

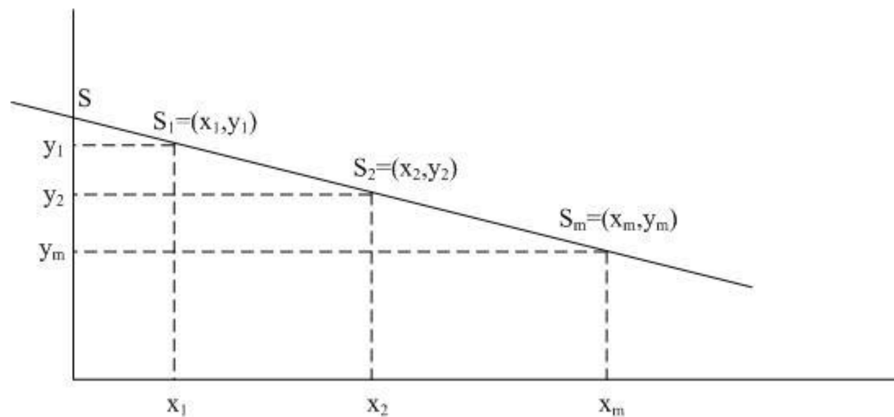

Figure 2-4: Shamir's secret sharing with two points (S_j and S_k; where j is not equal to k)

The goal is to divide some data K into m pieces K_1, K_2,……,K_m in such a way that:

a) Knowledge of any n or more K_i pieces makes K easily computable.
b) Knowledge of any n-1 or fewer K_i pieces leaves K completely undetermined.

This scheme is called (m,n) threshold scheme. To use the (m,n) threshold scheme to share a secret K, without loss of generality assumed to be an element in a finite field F, we choose at random (m-1) coefficients a_1, a_2,….,a_{m-1} in F, and let a_0=K. We build the polynomial $f(x)=a_0+a_1x^1+a_2x^2+a_3x^3+\ldots\ldots\ldots+a_{m-1}x^{m-1}$ and construct any n points out of it. For instance we set i=1,…..,n to retrieve (i, f(i)). Every participant is given a point (a pair of input to the polynomial and output). Given any subset of m of these pairs, we can find the coefficients of the polynomial using interpolation and the secret is the constant term a_0.

The following example illustrates the basic idea. Suppose that our secret is 1984 (i.e. K=1984). We wish to divide the secret into 5 parts (m=5), where any subset of 3 parts (n=3) is sufficient to reconstruct the secret. At random we obtain 2 numbers: 149, 57 (a_1=149, a_2=57). Our polynomial to produce secret shares is therefore $f(x)= 1984+149x+57x^2$.

Now, we construct 5 points from this polynomial: (1, 2190); (2, 2510); (3, 2944); (4, 3492); (5, 4154).

In order to reconstruct the secret any 3 points will be enough.
Let us consider (x_0, y_0)= (2, 2510); (x_1, y_1)= (3, 2944); (x_2, y_2)= (5, 4154).
We compute Lagrange interpolation polynomials for these three points:

$$l_0 = \frac{x-x_1}{x_0-x_1} . \frac{x-x_2}{x_0-x_2} = \frac{x-3}{2-3} . \frac{x-5}{2-5} = \frac{1}{3}x^2 - \frac{8}{3}x + 5$$

$$l_1 = \frac{x-x_0}{x_1-x_0} . \frac{x-x_2}{x_1-x_2} = \frac{x-2}{3-2} . \frac{x-5}{3-5} = -\frac{1}{2}x^2 + \frac{7}{2}x - 5$$

$$l_2 = \frac{x-x_0}{x_2-x_0} . \frac{x-x_1}{x_2-x_1} = \frac{x-2}{5-2} . \frac{x-3}{5-3} = \frac{1}{6}x^2 - \frac{5}{6}x + 1$$

Therefore

$$f(x) = \sum_{j=0}^{2} y_j \cdot \ell_j(x)$$

$$= 2510\left(\tfrac{1}{3}x^2 - \tfrac{8}{3}x + 5\right) + 2944\left(-\tfrac{1}{2}x^2 + \tfrac{7}{2}x + 5\right) + 4154\left(\tfrac{1}{6}x^2 - \tfrac{5}{6}x + 1\right)$$
$$= 1984 + 150x + 56x^2$$

The secret is the free coefficient, which means that K=1984.

2.11 Error Detection

In a good cryptographic system, a change in a single bit in the ciphertext changes more than one bit in corresponding plaintext. Most of communication channels contain some noise. This noise can interfere with data and flip the bits. As a goal of this thesis is to implement an n-out-of-m decryption of the session key, the first challenge is to detect errors in the retrieved key, since any errors in the values received from the escrow agent could lead to an incorrect regeneration of the session key. This error might occur due to a transmission channel failure, noise, software fault, memory fault, etc. An error in one share means the key cannot be regenerated. So the first task of the LEA is to detect any error in the shared parts sent by the n escrow agents. Then it should correct the error and try to regenerate the key

Error detection is simply the detection of any error in a message after being transmitted from the sender to the receiver. A lot of research and techniques have been developed over the years to detect errors in the received signals. Most of the techniques are based on adding some redundancy to the original message to detect errors. Some of these methods are discussed below.

2.11.1 Repetition Codes

This approach divides the message into blocks and repeatedly sends blocks, i.e., each block is sent more than once [30]. A comparison is made of all the repeated blocks and if any mismatch is found, then the data which is carried by the majority of the copies of a given block is considered to be as the correct data for this block. This approach is very simple, but cannot guarantee completely error free correction or detection. If an error occurs at the same position of each block, then this method fails.

For example, we want to send a bit stream 00011011. We divide the bit stream into blocks of two bits as 00, 01, 10, 11. Now we send each block by repeating each block two times. So the bit stream sent would be 000000, 010101, 101010, 111111. Suppose an error occurs in the first block and the receiver receives the first block as 000100, then the receiver can detect an error has occurred because the block does not represent a single value. As the most frequent value is 00, then the bit that is currently 1 is replaced by 0 to correct the error and the redundancy is removed to yield a received value for the first two bits of "00". It would be difficult to correct the error if the error occurs three times such as 010011. Although the receiver knows there are errors (since all three sets of two bits should be the same) but frequency count is equal for both 1 and 0. Additionally, it is impossible to detect an error when the error occurs in same positions of the block. For example, if the receiver receives

010101 in place of 000000, then it cannot detect any error. In this case the method fails. Additionally, if the receiver were to receive 000101, then it would "correct" the received value to yield "01", since the number of errors was greater than the simple redundancy could handle.

2.11.2 Checksum

A checksum refers to adding some redundancy along with the data to enable error detection. There are a number of checksum functions that can be used to calculate the checksum [31]. The performance of a checksum depends on the quality of checksum function. A good checksum function provides a high probability of error detection. Note that the checksum is used to detect an error, following this some higher layer protocol is generally utilized to decide what to do when an error is detected - for example, a retransmission might be requested.

The simplest checksum function is a parity check function. A bit stream is divided into blocks of n bits and an XOR operation is performed on each bit of a block. The result is appended to the block and sent to the receiver. The receiver receives the block (including the result of the XOR operation) and performs an XOR operation again on the whole block. If the result is 0, then the block is considered to be error free. Otherwise, an error is detected. This is also known as even parity check. For example, consider a bit stream with a block size seven consisting of the bits 1011000. After performing XOR operation in each bit we get a result 1. This 1 is appended to the block and sent to the receiver as 10110001. The receiver again performs an XOR operation on each bit and gets a result 0. The receiver believes that the block as error free. Now, assume an error occurs in sixth bit and the receiver receives 10110101. The result of XOR operation is now 1 and an error is detected. The problem with this approach is that even number of errors **cannot** be detected. It can detect only odd numbers of errors, but cannot ensure the receiver of the number of errors or where they occur. TCP uses a 16 bit checksum.

2.11.3 Cyclic Redundancy Check (CRC)

A cyclic redundancy check is produced by a non secure hash function. This method is frequently implemented in hardware. A CRC works on block of data and calculates a CRC value for each block of data [32]. When the block is received, the CRC is again calculated and matched with the previously calculated CRC value. If there is an error, then the CRC will not match. CRC generally works by using a division operation where the remainder is taken as the result. A good n bit CRC can detect an error of n bits in a burst. The problem with a CRC is that any attacker can intercept the message and modify the message and recalculate the CRC to yield a correct value for the modified message.

Computing a CRC is based on polynomial arithmetic that computes the remainder of dividing one polynomial in GF(2) by another where A polynomial in GF(2) is a polynomial in a single variable x whose coefficients are 0 or 1. For example, the message 10011011, where the order of transmission is from left to right is represented by the polynomial x7+x4+x3+x+1. To utilize a CRC to protect this data, the sender and receiver agree on a certain fixed polynomial called the generator polynomial. The generator polynomial must be of degree n to compute an n bit CRC checksum. The algorithm starts by the sender adding n 0 bits at the end of the m bit message. The sender divides the resulting polynomial of degree n+m-1 by the generator polynomial. The result is a polynomial of degree n-1 or less. The

remainder is a polynomial with n coefficients. This remainder is the CRC value. The sender transmits the m bit message and the n bit CRC value. The receiver can detect errors in two ways. The receiver can compute the CRC over the m bit message and compare this newly computed with the received n bit CRC value. If the CRC checksums are same, then no error has occurred. Alternatively, the receiver can divide all the received bits (including the n bit CRC value) by the generator polynomial and check that the n bit remainder is 0. If the remainder is 0, then the receiver believes that no error has occurred.

Choosing a suitable generator polynomial is the key to generating reliable CRC value [33][34]. All single bit errors are detected if the generator polynomial contains two or more terms. If the generator polynomial is not divisible by x (i.e., if the last term of the generator polynomial is 1), and e is the least positive integer such that the generator polynomial evenly divides xe+1, then all double bit errors within a frame of e bits are detected. If x+1 is a factor of the generator polynomial, then all errors consisting of an odd number of bits are detected. As a result, an n bit CRC detects all burst errors of length upto n bits. Ethernet uses the 32 bit CRC-32 (100000100110000010001110110110111).

2.12 Error Correction

The error correction codes are used to correct the errors in a transmitted bit stream. The errors are corrected by the receiver. This can be regarded as forward error correction (FEC), since the sender adds this redundancy in advance - allowing the receiver to correct the received message without making any additional request(s) or the sender. The error correction can be done according to the capability of the error correcting code used during transmission or storage. A number of error correcting codes have been proposed and implemented over past few decades. Several of them are discussed below.

2.12.1 Convolutional Code

Convolutional codes are used in numerous fields such as mobile communication, satellite communication and digital video for error correction. A binary convolutional code is denoted by a three-tuple (n, k, m) [35]. Each m bit string is transformed into an n bit string where m/n is the code rate and the transformation depends on the last k bits of the m bit string. In this formulation k is known as the constraint length. The choice of constraint length of a convolutional code is made in several ways. The most popular choice is m + 1. To compute the convolution code k memory registers are used to convolutionally encode the data. Each of the k registers makes use of n modulo 2 adders and n generator polynomials.

The first bit m_1 is fed into the leftmost register. The encoder outputs n bits by using the generator polynomials and the values in the other registers. By default, the values in the other registers at the beginning are assumed to be 0. The second bit m_2 is fed into the leftmost register by shifting the previous value one register right. This continues until all the bits have been fed into the registers and all registers contain 0. The encoder is shown in Figure 2-5.

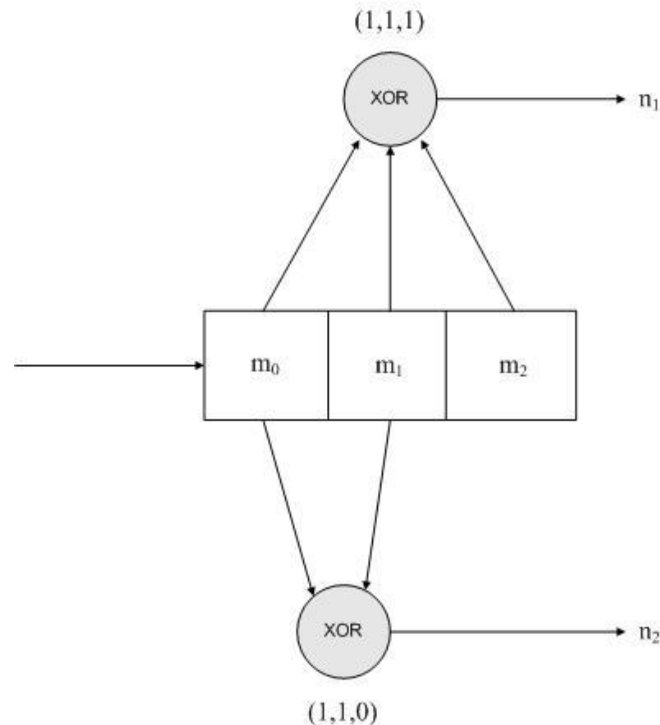

Figure 2-5: Example of a convolution encoder

For example, let us consider m=1, n=2 and k=3 [36]. So each bit will be converted into a two bit code and there will be three delay elements. Let the generator polynomials be (1,1,1) and (1,1,0). The output from the (1,1,1) polynomial uses the XOR of the current input, previous input, and the previous to previous input. The output from the polynomial (1,1,0) uses the XOR of the current input and previous input.

Let the input sequence be 0101. The first clock cycle makes the first input bit 0 available to the encoder. The inputs to the modulo-two adders are all zeroes, so the output of the encoder is 00.

The second clock cycle makes the second input bit available to the encoder. The leftmost register clocks in the previous bit, which was a 0, and the rightmost register clocks in 0 output by the leftmost register. The inputs to the top first adder are 100, so the output is 1. The inputs to the second adder are 10, so the output is also a one. So the encoder outputs 11 for the channel symbols.

The third clock cycle makes the third input bit 0 available to the encoder. The leftmost register clocks in the previous bit, which was 1, and the rightmost register clocks in 0 from two bit-times ago. The inputs to the first adder are 010, so the output is 1. The inputs to the second adder are 01, so the output is 1. So the encoder outputs 11 for the channel symbols.

The output sequence after all inputs would be 00 11 11 01.

In order to flush the registers we need to input two more 0s. So the final output becomes 00 11 11 01 11 10.

The encoder can be represented as a simple state machine. The example encoder has two bits of memory, so there are four possible states. Initially the encoder is in the all 0 state. If the first input bit is a 0, the encoder stays in the all 0 state at the next clock cycle. But if the

input bit is a 1, the encoder transitions to the 10 state at the next clock cycle. Then, if the next input bit is 0, the encoder transitions to the 01 state, otherwise, if the next input bit is 1, it transitions to the 11 state. Table 2-1 shows the next state of the encoder given the current state and the input.

Table 2-1: Next state table

	Next state	
Current state	**Input = 0**	**Input = 1**
00	00	10
01	00	10
10	01	11
11	01	11

Another table can be constructed by defining the outputs by the encoder for different combinations of inputs. This is shown in Table 2-2

Table 2-2: Output values

	Output	
Current state	**Input = 0**	**Input = 1**
00	00	11
01	10	01
10	11	00
11	01	10

2.12.2 Hamming Code

Hamming code can detect upto two bits of burst errors and correct a single bit. In contrast to a parity checksum, such a code has the capability of correcting the message. Parity bits are added to the original message in order to detect and correct errors. The more parity bits added to the message, the more errors it is possible to correct. For example, to send a 7 bit message with 3 parity bits added makes it possible to detect an error in any single position.

The Hamming code works by following algorithm [37].

- All the bits are numbered starting from 1.
- All bit positions that are powers of two are marked as parity bits (bits 1, 2, 4, 8, 16, 32, 64, etc.)
- All other bit positions are for the data to be encoded (bits 3, 5, 6, 7, 9, 10, 11, 12, 13, 14, 15, 17, etc.)
- Each parity bit calculates the parity for some of the bits in the code word. The position of the parity bit determines the sequence of bits that it alternately checks and skips.
    - Parity bit 1 (bit 1): check 1 bit, skip 1 bit, check 1 bit, skip 1 bit, etc. (1,3,5,7,9,11,13,15...)
    - Parity bit 2 (bit 2): check 2 bits, skip 2 bits, check 2 bits, skip 2 bits, etc. (2,3,6,7,10,11,14,15...)
    - Parity bit 3 (bit 4): check 4 bits, skip 4 bits, check 4 bits, skip 4 bits, etc. (4,5,6,7,12,13,14,15,20,21,22,23...)
    - Parity bit 4 (bit 8): check 8 bits, skip 8 bits, check 8 bits, skip 8 bits, etc. (8-15,24-31,40-47...)

- Parity bit 5 (bit 16): check 16 bits, skip 16 bits, check 16 bits, skip 16 bits, etc. (16-31,48-63,80-95...)
- Parity bit 6 (bit 32): check 32 bits, skip 32 bits, check 32 bits, skip 32 bits, etc. (32-63,96-127,160-191...)

- Set a parity bit to 1 if the total number of ones in the positions it checks is odd. Set a parity bit to 0 if the total number of ones in the positions it checks is even.

This algorithm can be used to detect and correct a single bit error. For example, let us consider a four bit message for transmission. Three bits of parity codes are added with this message. According to the algorithm, we can construct Table 2-3 defining the data bits parity bits and the bits to be checked by each parity bit.

Table 2-3: Parity calculation in Hamming code

8 bit codeword	1	2	3	4	5	6	7
Parity (P)/ Data(D)	P	P	D	P	D	D	D
Even Parity	x		x		x		x
		x	x			x	x
				x	x	x	x

The three parity bits (1,2,4) are related to four data bits (3,5,6,7). Parity bit 1 checks even parity for data bits (3,5,7), Parity bit 2 checks even parity for data bits (3,6,7), Parity bit 4 checks even parity for data bits (5,6,7). An error in bit 3 will affect parity bit 1 and 2, an error in bit 5 will affect parity bit 1 and 4, an error in bit 6 will affect parity bit 2 and 4 and an error in bit 7 will affect all three parity bits. Suppose we want to send the data bit 1001. According to table 3 the sent data can be calculated as shown in Table 2-4.

Table 2-4: Parity bits for data 1010

8 bit codeword	1	2	3	4	5	6	7
Data(D)			1		0	1	0
Even Parity	1		1		0		0
		0	1			1	0
				1	0	1	0

So the transmitted data will be 1011010. Now the receiver will receive this data and calculate the even parity. If there is an error in bit five, thus the receiver receives 1011110, then the error can be detected as shown in Table 2-5.

Table 2-5: Error detection in Hamming code

8 bit codeword	1	2	3	4	5	6	7
Data(D)	1	0	1	1	1	1	0
Parity error (1)	1		1		1		0
Parity correct (0)		0	1			1	0
Parity error (1)				1	1	1	0

From the correct and erroneous parities we find a value 101 which indicates bit five has an error. The value of bit five is changed to 0 and error correction is complete.

2.12.3 BCH Codes

Bose – Chaudhuri – Hocquenghem (BCH) codes are a class of cyclic codes [18]. They were discovered around 1959 and 1960 by R. C. Bose and D. K. Ray-Chaudhury and independently by A. Hocquenghem. BCH codes form a large class of multiple random error-correcting codes. BCH codes are important because there exists good decoding algorithms that can perform multiple error corrections. The BCH code is a polynomial code over a finite field with a particularly chosen generator polynomial. The codewords are formed by dividing a polynomial representing the message by a generator polynomial and then taking the reminder. The generator polynomial is a combination of several polynomials in GF(2^n) [39].

2.12.4 Reed-Solomon Codes

The Reed-Solomon code was invented in 1960 by Irving S. Reed and Gustave Solomon. The Reed-Solomon codes are special cases of BCH codes. In many applications, errors are not randomly distributed. Rather, the errors occur in a burst. Reed-Solomon codes are used for burst error correction. Reed–Solomon coding is widely used in mass storage systems, such as compact disks and digital video disks to correct the burst errors. Printed bar codes use Reed–Solomon error correction to allow correct reading even if a portion of the bar code is damaged. Reed-Solomon codes are also used for error correction in satellite communication. An example of this was to encode the digital pictures sent back by the Voyager spacecraft [40].

A Reed-Solomon code is specified as RS(n,k) with s-bit symbols [41]. The encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol codeword. There are n-k parity symbols of s bits each. A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where 2t = n-k. Given a symbol size s, the maximum codeword length (n) for a Reed-Solomon code is $n = 2^s – 1$.

Chapter 3: Related Work

In this section, some of the related works are presented. The Clipper chip as discussed in section 2.8 was an approach by the US government to use two escrow agents. The n-out-of-m threshold cryptography as discussed in 2.9 is an approach to divide the secret among m servers and retrieve the secret by having the shares from only n agents. A master thesis by Md. Sakhawat Hossen at KTH has addressed the issues of single escrow agents and has implemented the system [2]. Another thesis by Md. Sarwar Jahan Morshed at KTH addressed VoIP Lawful Interception from the point of view of the LEA [42]. Overviews of these two theses are given in sections 3.1 and 3.2.

3.1 SIP User Agent with Key Escrow

In his thesis, Hossen [2] implemented a very simple key escrow agent. The session keys are deposited with this escrow agent. During a session blocks of hashes are signed over the session contents and transmitted over the Real Time Transport Control Protocol (RTCP) channel parallel to the RTP media stream. The private key of the sender is used to sign the hash of sent packets. The receiver can use these signed hash values together with the sender's public key to detect modification of the sender's traffic. In fact, any party that has access to the signed hash values and the sender's public key can detect an attempt of forgery of the session's contents (Detecting this forgery is described in Morshed's thesis - see section 3.2.).

A signed hash over multiple SRTP packets is computed to prevent forgery of a recorded session. Because the hash is signed by the private key of the sender it is impossible for anyone to forge the digital signature of the hash over the Secure Real Time Transport Protocol (SRTP) packets as the private key is known only by the user. If an LEA retrieves the session key from the escrow agent by showing an appropriate legal order and tries to fabricate data into a captured media stream by generating SRTP packets and encrypting the media stream with the session key, then the authenticity of these packets can not be verified. The verifier decrypts the signed hash using the public key of the sender to produce the hash of SRTP blocks as calculated by the sender. The verifier processes the captured session and does the hashing operations to produce the hash of the captured session. If these two hashes are identical, then the captured packets have not been changed. But if the LEA has fabricated content or modified the session, then the hash of the captured session and the hash of the decrypted SRTP blocks will not be the same.

The escrow agent was implemented by Hossen using an Apache web server with MySQL database support. The escrow agent receives the key from an authenticated user and after proper validation of the received data it stores the escrowed data in a local database. The MySQL database stores the session master key(s). The session master key is the TEK Generation Key (TGK). This key is exchanged by the key agreement protocol MIKEY. This TGK along with some security parameters are used to generate the session keys for encryption and integrity protection. To allow a LEA to decode a captured session some additional information is escrowed along with the session master key. For example, the final signed hash value is also escrowed as a marker to indicate the end of a session. Morshed's thesis discusses what information in addition to the master key should be escrowed.

The web server enabled Secure Socket Layer (SSL) functionality. The user agent uses secure HTTP (HTTPS) to escrow the key with the escrow agent. The key is transferred along with the URL of the escrow agent. A key value pair is appended to the URL. The user name and password are used for authentication to the escrow agent. The SIP URI is used as the user

name thus only the users registered to a SIP proxy server can escrow their keys. The SIP URI need not all be in the same SIP domain nor do all of the subscribes in this domain need to use this escrow agent, the only limit is that the user is identified by something that looks like a SIP URI - and of course might actually be the user's SIP URI.The password is manually assigned and is established when a user is added to the database. A key value pair is appended to the URI with the user name and password to the escrow agent to escrow. A third party application programming interface (API) named “libcurl” is used to escrow the session master key. “libcurl” [38] is a free and easy-to-use client-side URL transfer library which supports HTTP, HTTPS, and many other protocols.

3.2 VoIP Lawful Interception

Md. Sarwar Jahan Morshed's thesis [42] addresses the issues of a LEA when retrieving a session key and performing decryption of a captured session. He assumes that the LEA has established the proper legal authority the key and associated information from the escrow agent. This thesis also addresses the case when an evil person acquires the key from the escrow agent in order to forge a captured session or to modify a captured session. He demonstrates that is possible to detect all attempts to modify a recorded session - provided that some specific data associated with the session is also recorded by the original user agent when it escrows the information about the session with the escrow agent.

Chapter 4: Design and Implementation of Split Operation

This chapter starts by proposing a solution for splitting the key into chunks in section 4.1. Section 4.2 discusses which parameters to split and escrow. Section 4.3 and section 4.4 describes where are the keys escrowed and the architecture of the escrow agents and escrow databases. Finally, section 4.5 describes the implementation details of the split and escrow operation.

4.1 Proposed Solution for Splitting the Key

As we are interested in splitting the key into m chunks and then retrieve the key from n-out-f-m chunks, a suitable algorithm for this would be Shamir's Secret Sharing Algorithm as discussed in section 2.10. The advantages of choosing Shamir's Secret Sharing algorithm are:

1) It is mathematically proven that the algorithm is secure.
2) The security level can be changed by changing the polynomial.
3) The algorithm is scalable because the number of chunks can be changed. It is possible to increase or decrease the value of m or n for an implementation.

In the libmikey library of minisip source code there is a file named Mikey.cxx. A function named escrowSessionKey() was added to the library as a public member of the Mikey class by Hossen [2]. Currently this function simply forms a URL to be used with one of the libcurl fuctions to instantiate a curl object. The function invokes a method to generate the TGK along with the pseudo-random number (Rand) and CSB ID value. After generating the parameters needed to be escrowed, we can add a splitting operation. This splitting function will split the key (and possibility the other parameters) into specified number of chunks and then send the chunks to a set of escrow agents.

The user agent is responsible for splitting the keys. The escrow agents will only store the escrowed information. The splitting operation is performed in the user agent and the user agent stores the values in files. Upon successful connection with the escrow agent, the user agent escrows the split values. This is shown in Figure 4-1.

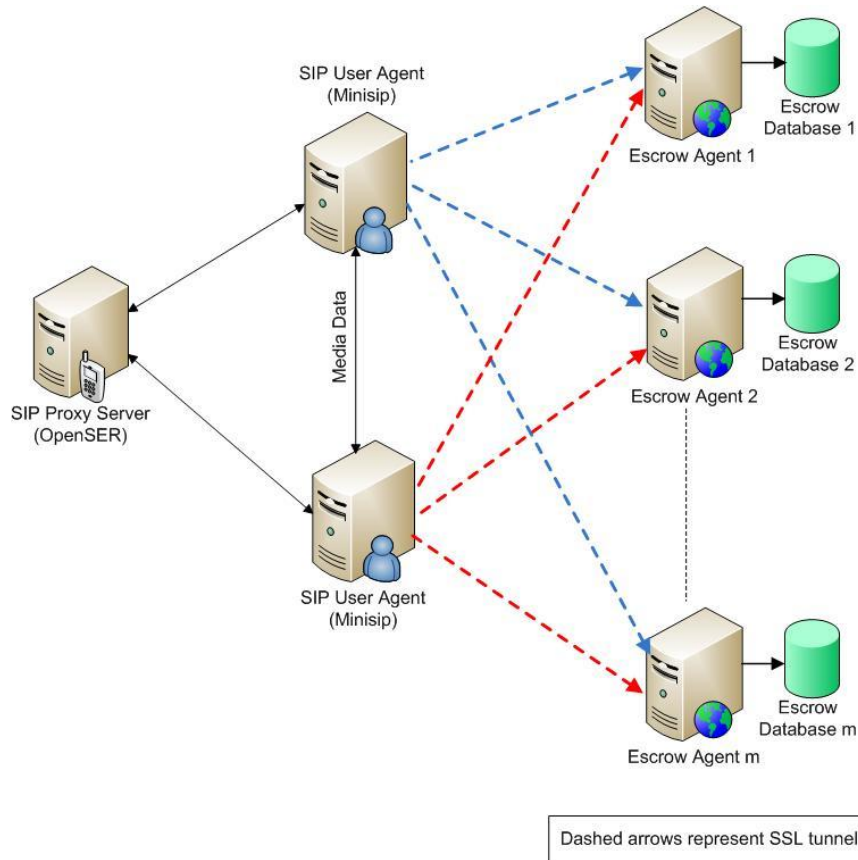

Figure 4-1: General architecture of m Escrow agents

4.2 What to Split and Escrow

According to the thesis of Hossen [2] and Morshed [42], they have escrowed the session master key, i.e., the TEK Generation Key (TGK) along with the pseudo-random number (Rand), last signed hash and CSB ID value. This key is exchanged by the key agreement protocol MIKEY. This TGK along with some security parameters are used to generate the session keys for encryption and integrity protection.

In our case, we are not modifying the escrow operation, rather enhancing the escrow operation from one escrow agent to multiple escrow agents. Thus, the parameters being escrowed will remain the same. But question arises which parameters should be split and then escrowed. For m escrow agents, we can split the TGK, Rand, last signed hash and CSB ID value into m chunks and then escrow them. But is it really necessary to split all these parameters? Well, the reason behind splitting the key is to enhance security and availability of the key. Splitting the Rand, last signed hash and CSB ID value enhances the security, but it is not really necessary to do so. If we rather only split the TGK and replicate the Rand, last signed hash and CSB ID value to m escrow agents, then our purpose is served. As this implementation is an n-out-of-m system, no one can retrieve the TGK value without having at least n chunks. This increases the security and confidentiality. Again, the availability increases as even if few escrow agents are not working, the key is retrievable from n escrow agents. So, we can conclude that we can reach our goal by splitting the TGK only into m chunks. The Rand, last signed hash and CSB ID value will be replicated with each of these m chunks into m escrow agents.

4.3 Which Escrow Agents are Involved in a Session

An important part of this thesis project is to find a way for the LEA to know who the escrow agents are for a session. Different user may use different escrow agents. Users may use different number of escrow agents for different sessions. How can the LEA know the number of escrow agents, or identity of these escrow agents? This question is also valid for the user himself or herself. If the user needs to retrieve a session master key several years after the session has ended, how will they remember the escrow agents he/she used for that particular session?

As discussed in section 1.3.2, we can solve this issue in two different ways. First we can send the list of escrow agents at the end of the session with the session master key, rand, CSB ID value and the signed hash value of the last block to all the escrow agents. The signed hash value of the last block is already generated by Hossen's thesis [2]. The last signed hash value is sent to the escrow agent in order to detect forgery. We could just add the list of escrow agents with these parameters and send it to the escrow agents. The LEA who captures the traffic can easily notice from traffic analysis that the session has ending by sending packets to some IP addresses other than the media stream has been sent to. If the LEA finds even one IP address of an escrow agent from this traffic analysis, then it can present a lawful intercept court order to this escrow agent to ask it to reveal the addresses of the other escrow agents from the list that it received along with the key chunks.

Another solution could be to explicitly send the role of the user and the list of escrow agents via the RTCP path. This information can be sent at any time during a session. It can be sent at the beginning, at the end, or any time during the session. Our proposal would be to send this information twice in the session, once at the beginning, and then at the end of the session. The reason behind this proposal is that the RTCP packets are sent over the User Datagram Protocol (UDP) and thus their transmission is unreliable. If the LEA fails to capture the RTCP packet containing the list of escrow agents due to the unreliable nature of UDP, then it will never find out the list. Additionally, it is better to send the same information at the end of the session to increase reliability. The libminisp library of the minisip code handles the media streams. The MediaStream.cxx file contains the RealtimeMediaStreamSender class which is mainly responsible for sending SRTP/RTCP packets. To send the list of escrow agents in the RTCP path we can add a function inside the RealTimeMediaStreamSender class in the MediaStream.cxx file. This function will send the user role and list of escrow agents via the RTCP path at the beginning and at the end of a session.

In this thesis project, we have implemented the first alternative. We have escrowed the names of the escrow agents involved in a session with the split TGK chunks, Rand, last signed hash and CSB ID values as discussed in section 4.2. We have added the names of the escrow agents in the escrowSessionKey() function of Mikey.cxx file in libmikey library of the minisip source code. The LEA who captures the traffic can detect from traffic analysis that the session ends by sending packets to some IP addresses other than the media stream has been sent to. The LEA would assume these IP addresses as the escrow agents. If for some reason the LEA can not detect all the escrow agents from traffic analysis, then it can present a lawful intercept court order to the escrow agent(s) it has detected from traffic analysis and ask to reveal the addresses of the other escrow agents from the list along with the TGK chunk and last signed hash value.

4.4 Escrow Agents and Escrow Databases

The escrow agents implemented for this thesis project is same as the escrow agents implemented by Hossen [2], but with a minor modification. Each escrow agent has two fields for the split TGK. In the original implementation of Minisip [11], there are 256 bytes in the base 64 value of the TGK. We have divided these 256 bytes into two equal parts (128 bytes each). This is done because we have used 128 byte security level in Shamir's secret sharing algorithm. So we can split at most 128 with a single execution of the algorithm.

The escrow agents have been implemented using Apache web server with MySQL database support. It has a database named *db_escrowAgent* consisting of two tables as presented in the Figure 4-2. The primary task of the escrow agents is to receive the key chunks from an authenticated user. After proper validation of the received data, it is stored in a secure database. Figure 4-1 shows the general architecture of our escrow agents. The web server is enabled with Secure Socket Layer (SSL) functionality so that user agent can use secure HTTP (HTTPS) to escrow the key with the escrow agent. SSL has been enabled according to the procedure shown in Appendix A for Ubuntu servers. To enable the SSL capability of the in OpenSuse, a script has been used to automate the complete process (see Appendix B).

The databases consist of two tables: one for authentication data and the other for the escrowed data. The authentication table stores the username and password of the valid users who can escrow data with the escrow agents. We have used the SIP URI as the username so that only users registered with the proxy server are able to escrow data with this escrow agent. The password is manually assigned and is established when a user is added to the authentication table of the database.

The sipmasterkey table stores the two parts of the TGK key chunks along with Rand, signed hash value, CSB ID values and the names of the escrow agents. The sipmasterkey table also contains a date field that stores the current local time as a timestamp to record when the entry in the table was made.

authentication

id	user_name	password

sipmasterkey

id	user_name	key1	key2	rand	csbid	signed_hash	date	EAnames

Figure 4-2: General Structure of the escrow databases

4.5 Implementation Details

Necessary codes have been added to escrowSessionKey() function in Mikey.cxx file to divide the base 64 value of TGK into 5 parts. The internal structure can be described as follows:

4.5.1 General Algorithm for Split and Escrow Operation

The procedure of splitting the key into chunks and escrowing them can be depicted by a general algorithm. The algorithm works according to the following steps:

1. Create five files for temporarily storing the key chunks.

2. Divide the TGK into two parts.

3. Invoke the split function for each of the parts.

4. Each part is split into five subparts. Store each of these split parts into the files created in step 1. The five subparts created from the first part will be the first entries in the five files. Five subparts created from the second part will be the next entry in the files. Separate them with a separator “%”.

5. Create a temporary string with the Rand, signed hash, CSB ID value and the names of the escrow agents. Separate them with “%” symbol.

6. Read the contents of the first file created in step 1 into a string.

7. Form a string with the IP address of the escrow agent.

8. Append the user id, password and strings created in steps 5 and 6 to the string created in step 7.

9. Escrow the parameters by creating a curl object with the string formed in step 8.

10. Repeat steps 6 to 9 four times, read values from different files each time in step 6.

4.5.2 Storing the Key Chunks into Files

As we are interested in dividing the key onto five chunks, we generated five files in order to store the TGK parts. The split operation will divide the TGK into five parts and each of the parts will be stored in these files. The number of parts can be increased or decreased by changing the total Escrow Agent number. The number of files created will be exactly same as the number of total Escrow Agents. Listing 4-1 shows how we have created multiple files to store the key chunks.

```
char filnamenumberstr[20];
char filtmp[15]="ea.out";
char eafilname[15];

strcpy(eafilname, filtmp);
for(int filnumber=1;filnumber<=totalea;filnumber++){
    strcpy(eafilname, filtmp);
    sprintf(filnamenumberstr,"%d",filnumber);
    strcat(eafilname, filnamenumberstr);
    output=fopen(eafilname,"w+");
    fclose(output);
}
```

Listing 4-1: Creating five different files to store the key chunks

4.5.3 Dividing the TGK and Invoking the Split Function

There are 256 bytes in the base 64 value of the TGK. These 256 bytes are divided into 2 equal (128 bytes each) parts. The split function is called for each of these parts and each part is divided into 5 chunks by the split function according to Shamir's Secret Share algorithm. Each of these 5 parts is written into the 5 files created earlier. At the end of the split operation, we get 5 files. Each of the files contains 2 subparts of the TGK. These 2 parts are separated by a "%" symbol. Listing 4-2 shows how the TGK is divided into two parts and passed to split function.

```
for(sub=0;sub<=128;sub=sub+128){
    if(sub==128)
       tgktemp=tgk_b_64_ecoded.substr(sub);
    else
       tgktemp=tgk_b_64_ecoded.substr(sub,128);
    ssTgk=new char[tgktemp.size()+1];
    std::copy(tgktemp.begin(), tgktemp.end(), ssTgk);
    ssTgk[tgktemp.size()] = '\0',
    filnamenumber=1;

    split(ssTgk);
    tgktemp.erase(0, tgktemp.length());
    delete[] ssTgk;
 }
```

Listing 4-2: Dividing the TGK into two parts and invoking split() function

4.5.4 URL Formation and Escrow Operation

An URL is formed with the IP address of the first escrow agent, user name and password of the user appended by the contents of the first file. The signed hash value, rand value, CSBID values are appended into the URL separated by a "%" symbol without any modification. This information is escrowed to the first escrow agent through HTTPS channel. This is repeated five times for five Escrow Agents with the contents of five different files. At the end of this step we have successfully formed an URL with necessary parameters. Listing 4-3 shows how the parameters are invoked and read from file and how the URL is formed.

```
      int decodedlength;
      unsigned char *b_64_decode
=base64_decode(tgk_b_64_ecoded,&decodedlength);
      const char *csbId = itoa((int)ka->csbId()).c_str();// get the csbid
and convert to string


      char *tempcstr;
      tempcstr=new char
[rand_b_64_ecoded.length()+signedHash_b_64_ecoded.length()+100];

      strcpy(tempcstr,rand_b_64_ecoded.c_str());
      strcat(tempcstr,"%");
      strcat(tempcstr,signedHash_b_64_ecoded.c_str());
      strcat(tempcstr,"%----");
      strcat(tempcstr,csbId);
      strcat(tempcstr,"%");
```

```
        int filnumcount=1;
        for(filnumcount=1;filnumcount<=totalea;filnumcount++){
            char eadata[1500];
            char filnumcountstr[15];
            char eafilname[20]="ea.out";
            sprintf(filnumcountstr,"%d",filnumcount);
            strcat(eafilname, filnumcountstr);
            fstream ea(eafilname, ios::in);
            while(! ea.eof()){
                 ea.getline(eadata, 1500);
            }
            ea.close();


            std::string streadata(eadata);
            string url1;

            if(filnumcount==1)
                url1="https://130.237.81.122/escrow_agent1/?user=";
            else if(filnumcount==2)
                url1="https://192.168.1.212/~ea2/escrow_agent2/?user=";
            else if(filnumcount==3)
                url1="https://130.237.81.122/escrow_agent3/?user=";
            else if(filnumcount==4)
                url1="https://192.168.1.212/~ea4/escrow_agent4/?user=";
            else if(filnumcount==5)
                url1="https://192.168.1.212/~ea5/escrow_agent5/?user=";

            cstr=new char [url1.length()+streadata.length()+2*ka-
>uri().length()+500];

            strcpy(cstr, url1.c_str());
            strcat(cstr,ka->uri().c_str());// add sip uri as userid
            strcat(cstr,"&password=");
            strcat(cstr,ka->uri().c_str());// add sip uri as password
            strcat(cstr,"&data=");
            strcat(cstr,streadata.c_str());
            strcat(cstr,tempcstr);    //This is the total string to be passed
as url
            char *eanames;
            eanames=new char[200];
            strcpy(eanames, "EA1,EA2,EA3,EA4,EA5");
            strcat(cstr,eanames);
```

Listing 4-3: Formation of the URL where the top gray colored area shows how the parameters are invoked and the lower yellow colored area shows the formation of the URL

4.5.5 How to Escrow

We have used secure HTTP (HTTPS) to escrow the session master key. The key is transferred along with the URL of the escrow agent by appending a key value pair in addition to the key value pairs used to provide the user name and password for authentication to the escrow agent. HTTPS is used to create a secure SSL tunnel between the user agent and the server so that data can not be tampered with by others and to protect our key from being intercepted. To escrow the session master key with the escrow agent from the user agent (in our case: minisip) we have used libcurl [38], as described previously in section 3.1 . We have

modified the *escrowSessionKey()* function in Mikey.cxx file in the libmikey library of the minisip source code to escrow the session master key which was written by Hossen [2].

```
     CURL *curl;
     CURLcode res;
     curl = curl_easy_init();
     if(curl) {
        curl_easy_setopt(curl, CURLOPT_URL, cstr);
        curl_easy_setopt(curl, CURLOPT_TIMEOUT,5); //wait for 5 seconds
for the escrow agent to respond

        #ifdef SKIP_PEER_VERIFICATION

/*If you want to connect to a site who isn't using a certificate that is
signed by one of the certs in the CA bundle you have, you can skip the
verification of the server's certificate. This makes the connection A LOT
LESS SECURE.If you have a CA cert for the server stored someplace else
than in the default bundle, then the CURLOPT_CAPATH option might come
handy for you.*/

            curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L);
        #endif

       #ifdef SKIP_HOSTNAME_VERFICATION

/* If the site you're connecting to uses a different host name that what
they have mentioned in their server certificate's commonName (or
subjectAltName) fields, libcurl will refuse to connect. You can skip this
check, but this will make the connection less secure.*/

           curl_easy_setopt(curl, CURLOPT_SSL_VERIFYHOST, 0L);//saki
       #endif

       res = curl_easy_perform(curl);
       cout <<"\n Time to escrow: \n" <<duration_escrow << '\n';
       ea.close();

       /* always cleanup */

       curl_easy_cleanup(curl);
       if (res == CURLE_OK)
       {
          cout<<"\ncurl has easily performed\n";
          cout << buffer << "\n";
       }
       else
       {
          cout << "Error: [" << res << "] - " << errorBuffer;
          cout<< "\ncurl objcet is not created properly\n";
       }

     }//https check ends here
     delete [] cstr;
   }
```

Listing 4-4: Invocation of the libcurl method

As we have used HTTPS with a self signed certificate, we needed to skip the verification of the server's certificate. Libcurl provides the `SKIP_PEER_VERIFICATION` macro definition. We can skip the verification of the server's certificate by defining this macro. Although it makes the connection less secure, we have used this approach as our escrow agents are using self signed certificates.

4.5.6 The Split Function

We have used third party software for splitting the key. This software was written by B. Poettering [43]. It is a free software, the code is licensed under the GNU General Public License [44]. We have modified the code according to our need to integrate with Minisip.

There was a problem with the split function. The split() function reads random values from the /dev/random file. The /dev/random file generates random values from the entropy. The file does not cache any value, so there is a substantial delay if there are no random numbers generated. This was the reason of long delay for the split() function. We changed the random number generator file to /dev/urandom. The /dev/urandom file serves the similar purpose as /dev/random. But it caches the random values. So whenever the system runs out of random numbers, it can reuse the cached values. Using /dev/urandom solved the problem of delay. Now the split() function is executes in quick time. Using the /dev/urandom file decreases security as the same random value might be repeated for encryption. But in our case this is not a big issue as we are sending the data over https channel.

Chapter 5: Design and Implementation of Combine Operation

So far in this thesis project, we have designed and implemented escrow agents and added functionalities to an existing user agent (Minisip) to split the session key. The work done so far is adequate for a user agent to initiate call and at the end of the call the session key is divided into five parts ant each of the parts is escrowed to the escrow agents with the user id, Rand, last signed hash, CSB ID value, escrow agents names and the date and time. Now our concern is to retrieve the session key from n escrow agents (where n is less than the total number of escrow agents) for a particular call. In this chapter we have discussed how we can retrieve the session key as an LEA using n-out-of-m threshold scheme.

5.1 General Algorithm for Retrieving the Key chunks and Combining Them

This section discusses about the general approach of how we have designed our system to retrieve the key chunks from the escrow agents and combine them in order to get the TGK. The general algorithm is as follows:

1. Login to the escrow agent by providing user id and password by a web based form.

2. Provide the target user id, start time and end time of target session for which the session key has to be retrieved.

3. For authenticated user, read the first escrow agent database and fetch the two parts of the split TGK, Rand, CSB ID value, last signed hash value and names of the escrow agents.

4. Write the values read in step 3 into a temporary file. Separate the values by a "%" symbol.

5. Repeat step 3 and 4 for four other escrow agents. For each escrow agent, write the retrieved values in separate files.

6. Read any three files until the first separator "%" is found.

7. Invoke the combining function with the values fetched in step 6. At the end of this step we get the first half of the TGK.

8. Read the same three files read in step 6 starting after the first "%" symbol until the next "%" symbol is found.

9. Invoke the combining function with the values fetched in step 8. At the end of this step we get the second half of the TGK.

10. Merge the two halves of the TGK to get the session key.

5.2 Escrow Agent Support for LEA

In Lawful Interception mechanism, Escrow Agent is one of the key components since it works as a Trusted Third Party for storing the TEK generation key (TGK) along with other escrowed information to derive the session keys later. So far we have discussed and implemented the escrow agents for just storing the escrowed information. But from the point of view of a LEA, the escrow agents should provide the necessary key parameters to the LEA whenever a legal order is produced. The escrow agents are web based and co-located with an apache web server. They have a database named *db_escrowAgent* consisting of two tables. MySql is used for the database operation and Apache has been used as the web server. PHP has been used for accessing the web server remotely. To support the LEA with necessary information, we need to add one more table in the escrow agents according to Sarwar's [42] thesis. This table is named as *t_lealogin* consisting two fields: l_id and l_pass. This table is used to verify the LEA authentication. Anyone with a valid LEA id and password can login and request for escrowed information. The following login prompt appears before an LEA employee.

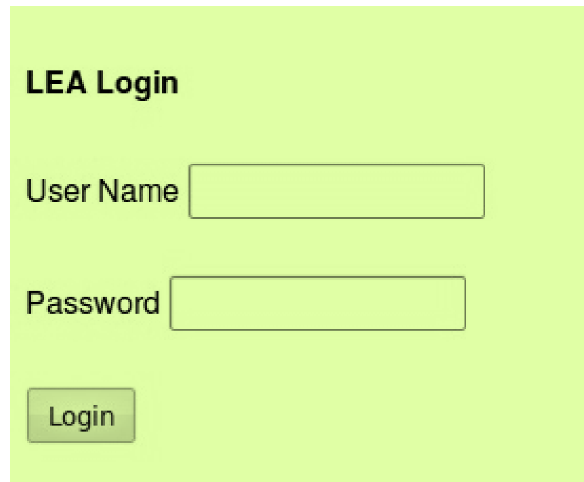

Figure 5-1: Login interface for LEA employee

After verifying the login information, a second form is provided to fill up with target information. This information includes the user id (in our case the sip user id) of a user, start time of the escrow operation and end time of the escrow operation. As we are escrowing into five escrow agents with a 5 second timeout, it is necessary to provide the end time to properly retrieve information.

Provide Target Information

User ID

Start Time

End Time

Submit

Figure 5-2: Target information interface

A php script has been written in order to fetch the information from the escrow agents. The php script runs five times and invokes the escrow agents one after another. Upon successful invocation on the escrow agents the script fetches the two split parts of the TGK, Rand, CSB ID, the signed hash value and the names of the escrow agents and writes them into files separated by a "%" symbol. After invoking five escrow agents, the LEA is provided with five text files. This text files are stored and are inputs to the combining function. Basically, at this point the LEA has retrieved everything and the role of escrow agent ends here. Now it is the job of the LEA to combine the split key parts. The php script used to fetch information and write them into files is shown in Listing 5-1.

```
<?php
$uid=$_POST['User_Id'];
$time1=$_POST['starttime'];
$time2=$_POST['endtime'];

for ($i = 1; $i <= 5; $i++) {
      if($i==1){
            $dbhost = 'localhost';  $dbuser = 'root'; $dbpass = '014409';
            $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die
('Error connecting to mysql');
            $dbname = 'escrowDatabase1';
            mysql_select_db($dbname);
            $myFile = "/home/azfar/Desktop/eadir/testFile1.txt";
            }
      else if($i==2){
            $dbhost = 'localhost';  $dbuser = 'root'; $dbpass = '014409';
            $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die
('Error connecting to mysql');
            $dbname = 'escrowDatabase2';
            mysql_select_db($dbname);
            $myFile = "/home/azfar/Desktop/eadir/testFile2.txt";
            }
      else if($i==3){
            $dbhost = 'localhost';  $dbuser = 'root'; $dbpass = '014409';
            $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die
('Error connecting to mysql');
            $dbname = 'escrowDatabase3';
            mysql_select_db($dbname);
            $myFile = "/home/azfar/Desktop/eadir/testFile3.txt";
```

```
            }
      else if($i==4){
            $dbhost = 'localhost';  $dbuser = 'root'; $dbpass = '014409';
            $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die
('Error connecting to mysql');
            $dbname = 'escrowDatabase4';
            mysql_select_db($dbname);
            $myFile = "/home/azfar/Desktop/eadir/testFile4.txt";
            }
      else if($i==5){
            $dbhost = 'localhost';  $dbuser = 'root'; $dbpass = '014409';
            $conn = mysql_connect($dbhost, $dbuser, $dbpass) or die
('Error connecting to mysql');
            $dbname = 'escrowDatabase5';
            mysql_select_db($dbname);
            $myFile = "/home/azfar/Desktop/eadir/testFile5.txt";
            }
      $select_key1 = mysql_query("select key1 from sipmasterkey where
userid ='$uid' and date BETWEEN '$time1' AND '$time2' ");
      $key1 = mysql_fetch_row($select_key1);
      $key1 = $key1[0];

      $select_key2 = mysql_query("select key2 from sipmasterkey where
userid ='$uid' and date BETWEEN '$time1' AND '$time2' ");
      $key2 = mysql_fetch_row($select_key2);
      $key2 = $key2[0];

      $select_rand = mysql_query("select rand from sipmasterkey where
userid ='$uid' and date BETWEEN '$time1' AND '$time2' ");
      $rand = mysql_fetch_row($select_rand);
      $rand = $rand[0];

      $select_signedhash = mysql_query("select signedhash from
sipmasterkey where userid ='$uid' and date BETWEEN '$time1' AND '$time2'
");
      $signedhash = mysql_fetch_row($select_signedhash);
      $signedhash = $signedhash[0];

      $select_csbid = mysql_query("select csbID from sipmasterkey where
userid ='$uid' and date BETWEEN '$time1' AND '$time2' ");
      $csbid = mysql_fetch_row($select_csbid);
      $csbid = $csbid[0];

       $select_eaname = mysql_query("select EAnames from sipmasterkey where
userid ='$uid' and date BETWEEN '$time1' AND '$time2' ");
      $eaname = mysql_fetch_row($select_eaname);
      $eaname = $eaname[0];

      $stringData = $key1.'%'.$key2.'%'.'%'.$rand.'%'.$signedhash.'%'
.$csbid.'%'.$eaname.'%';

      $b = fopen($myFile, "w") or die("can't open file");
      fwrite($b, $stringData);
      echo "your file has been saved";
      echo "</br>";
      fclose($b);
  }
?>
```

Listing 5-1: PHP script to fetch information from escrow agents

5.3 The Combining Operation

The combining operation is done by reading the split key parts from the files read by the LEA. Shamir's Secret Sharing scheme is used to combine the split chunks. The files created in section 5.2 serves as input to the combining function. As it is a 3-out-of-5 scheme, any three files are read until the first "%" symbol is found. The combining function is invoked with the values. The combining function combines the chunks to get the first half of the TGK. Now, the same three files are read beginning after the first "%" symbol until the next "%" symbol is found. The combining function is invoked with the values. The combining function combines the chunks to get the second half of the TGK. The two halves of the TGK are merged together to get the session key. Listing 6 shows this.

```
FILE *fp1;
FILE *fp2;
FILE *fp3;

static int flag=1;
static int  key1, key2, key3;
key1=0;
key2=0;
key3=0;
char c1,c2,c3;

fp1=fopen("/home/azfar/Desktop/eadir/testFile1.txt", "r");
fp2=fopen("/home/azfar/Desktop/eadir/testFile2.txt", "r");
fp3=fopen("/home/azfar/Desktop/eadir/testFile3.txt", "r");

char comb1[300], comb2[300], comb3[300], others[400];
outfp=fopen("/home/azfar/Desktop/eadir/outfile.txt", "w");
fclose(outfp);
int w;

for (w=1; w<=2; w++){
    key1=0;
    key2=0;
    key3=0;

    strcpy(comb1,"");
    strcpy(comb2,"");
    strcpy(comb3,"");

          while(1){
      c1=fgetc(fp1);
            if(c1!=EOF)     {
              if (c1!='%')  {
                comb1[key1]=c1;
                        key1++;
                        }
                 else{
                        comb1[key1]='\0';
                        break;
                        }
                        }
                }
          printf("\n");

          while(1){
            c2=fgetc(fp2);
```

```
        if(c2!=EOF)     {
           if (c2!='%') {
                comb2[key2]=c2;
                   key2++;
                            }
              else{
                comb2[key2]='\0';
                break;
                  }
                  }
          }

      printf("\n");

      while(1)    {
        c3=fgetc(fp3);
        if(c3!=EOF)     {
          if (c3!='%'){
               comb3[key3]=c3;
                  key3++;
                  }
    else{
                  comb3[key3]='\0';
                  break;
                  }
               }
           }

      printf("\n");

      printf("%s\n",comb1);
      printf("%s\n",comb2);
      printf("%s\n",comb3);

      combine(comb1,comb2,comb3);
```

Listing 5-2: Invocation of combine() function

Chapter 6: Performance Evaluation and Discussion

This chapter evaluates the performance of the proposed multiple key escrow agents based on Shamir's secret sharing. It also presents a detailed discussion and analysis of the performance evaluation results. The chapter starts with a brief description of the experimental setup. The measurements of time required to split the TGK are discussed in section 6.2. Sections 6.3 through 6.5 examine measurements on the system with different numbers of available escrow agents. Degradation of the performance of escrow agents was emulated imposing high link delay on escrow agents -see section 6.6. In section 6.7, we discuss the impact of escrow time as a function of the number of available/unavailable escrow agents. Finally, in section 6.8 we describe the availability measures as experienced by the LEA.

6.1 Experimental Setup for Escrow Operations

For the purpose of measuring the performance of the escrow operations we used the experimental setup as shown in Figure 6-1.

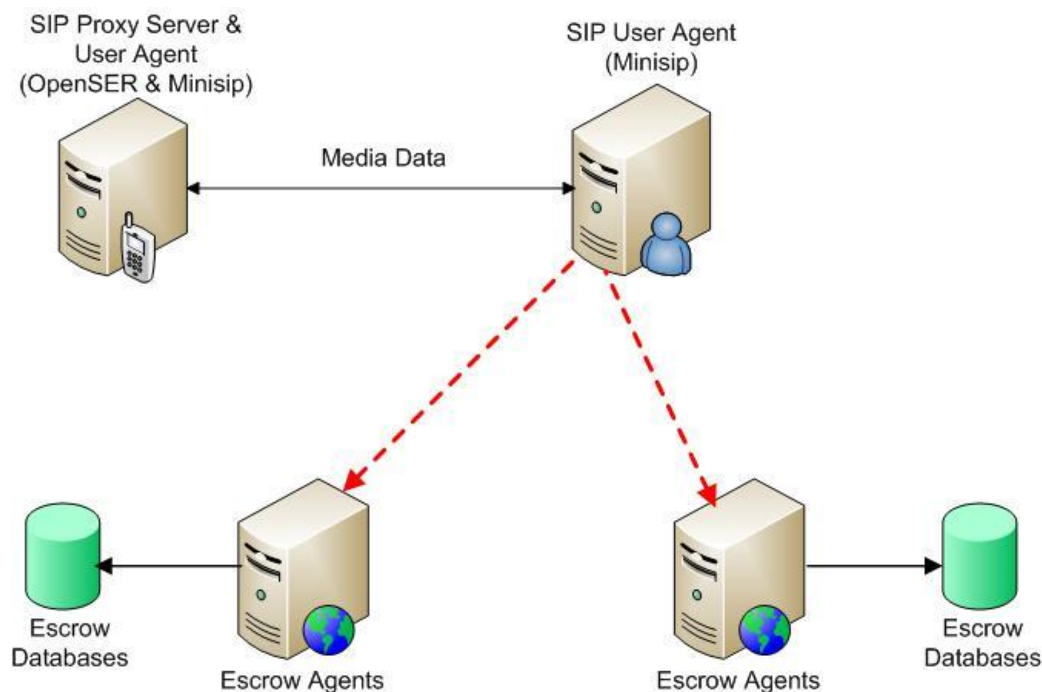

Figure 6-1: Experimental setup for escrow operations

We used one machine as SIP user agent and another machine as the SIP proxy server and other user agent. We used two machines configured as escrow agents along with escrow databases. These escrow agents were connected to the user agent via SSL tunnel. Both the escrow agent machines had five databases and five escrow agents defined in them. This allowed us to manipulate the escrow operation between the two escrow agent machines. We were able to introduce link delays to certain escrow agents and use those escrow agents in one machine. For example, if we wanted to impose link delay in escrow $agent_2$ and escrow $agent_4$,

we activated these two escrow agents in one machine and remaining three escrow agents in the other machine. Thus we could impose delay in a single link in order to manipulate the escrow operation. The other link with the three remaining escrow agents experienced no delay. Now if we wanted to add link delay to one more escrow agent then we configured that escrow agent in the same machine where the delay was being experienced. In this way we performed our experiments. We used SuSE version 10.3 of Linux, on Dell T7570 with Intel® Pentium® D CPU processor clocked at 2.80GHz and configured with 2048 MB of memory as the SIP user agents and proxy server. The CPU has a 1 microsecond clock resolution. Details about the measurement of CPU clock resolution have been discussed in appendix H.

6.2 Time Required to Split the Base 64 TGK Value into Chunks

The base 64 TGK value consists of 256 bytes. We divided this value into two equal halves. Then we executed the split function ten times for each of the halves and each time the split function was executed hundred times. This was done in order to collect samples for statistical analysis of the execution time of the split function. In a normal execution, the split function is only called twice, once for the first half of the TGK, and again for the remaining half of the TGK value. In our experiment, we executed the split function one hundred times for the same TGK value. The procedure was repeated ten times with ten different TGK values. As a result we have ten measurements for each of these hundred calls. We hope that these 1000 calls to the split function are representative of the time to compute the split and will give us insight into the processing time taken by other processes running on the same computer. Figure 6-2 shows the Experimental setup for split time calculation.

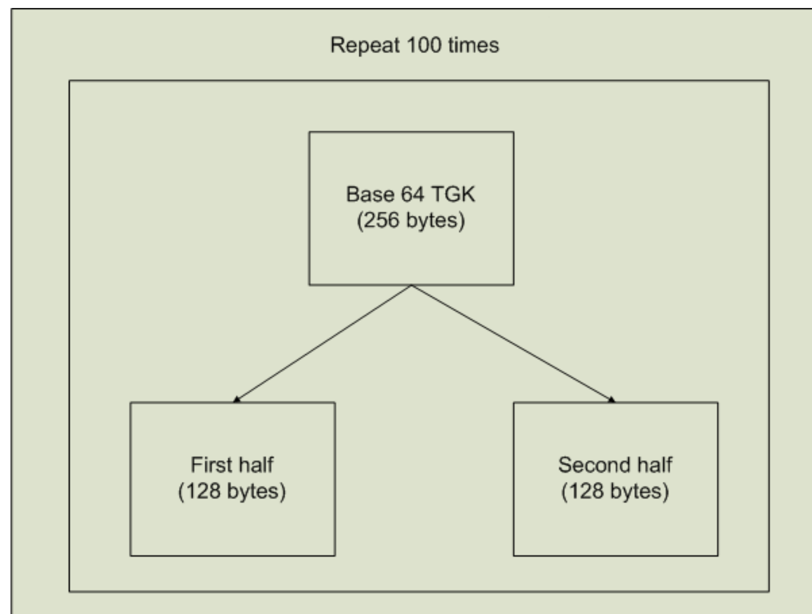

Figure 6-2: Experimental setup for split time calculation

Figure 6-3 shows a box plot of the measured execution time of the split function for the first half of the base 64 TGK value.

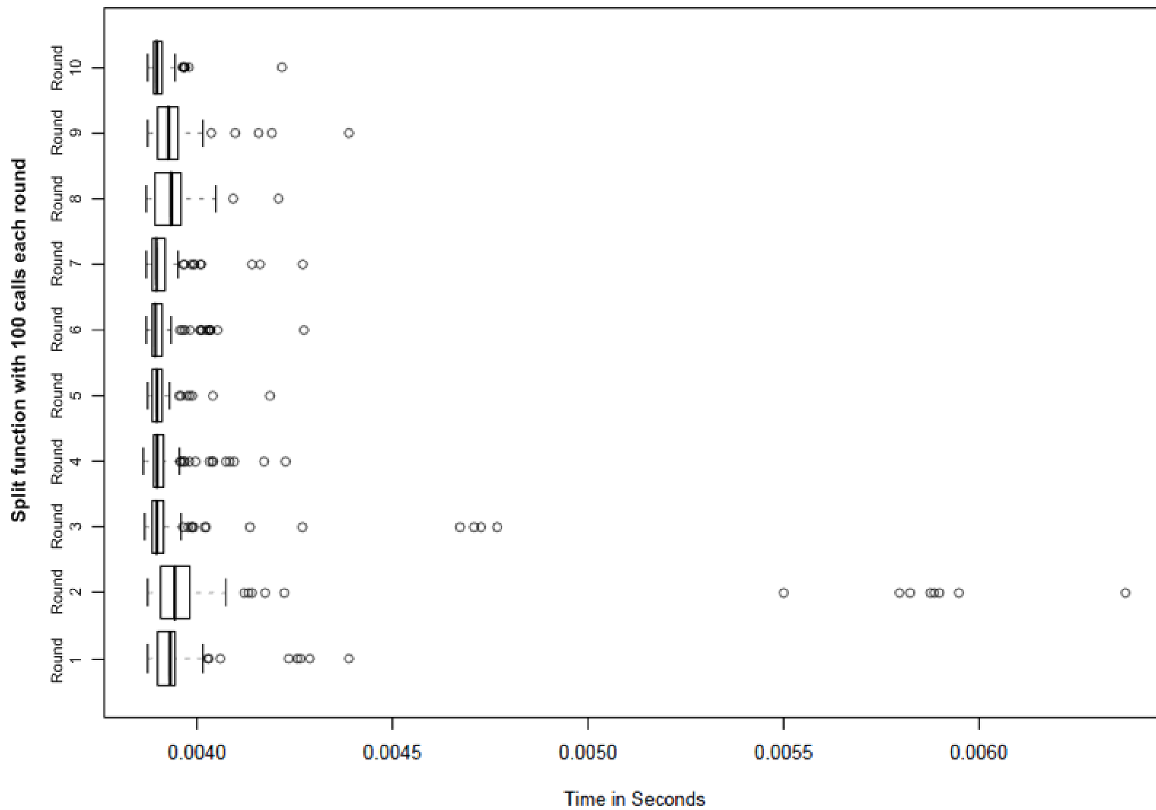

Figure 6-3: Execution time for split function applied to the first 128 bytes of the TGK encoded in base 64

From Figure 6-3, apart from round 2 there are some occasional outliers for each of the rounds. However, in the case of round 2 we can see a lot of outliers. The reason for this large number of outliers for this round is unknown. But we can make some assumptions based on the following facts.

- The time to compute the split is data dependent - i.e., for different values of a key it takes different amounts of time.
- The computer is multitasking - thus the CPU is being allocated to other processes.
- The computer is also servicing interrupts from various devices - ranging from the clock to I/O devices

Table 6-1 shows the statistical data found from the time required to split the first 128 bytes of the base 64 TGK value using 1,000 calls of the split function. Here, the unit of time is seconds.

Table 6-1: Statistical data to split first half of TGK encoded in base 64

Count	1000
Mean	0.0039
Median	0.0039
Mode	0.0038
Standard Deviation	0.00019
Sample Variance	3.72056E-08
Minimum	0.0038
Maximum	0.0063
Confidence Level (95.0%)	3.82489E-07

Figure 6-4 shows the box plot for the execution time of the split function for the second half of the TGK value encoded in base 64 (i.e., the second 128 bytes). Note that the scale for this figure is quite different for the previous figure, due to the one extreme outlier at 0.1 seconds in round 10.

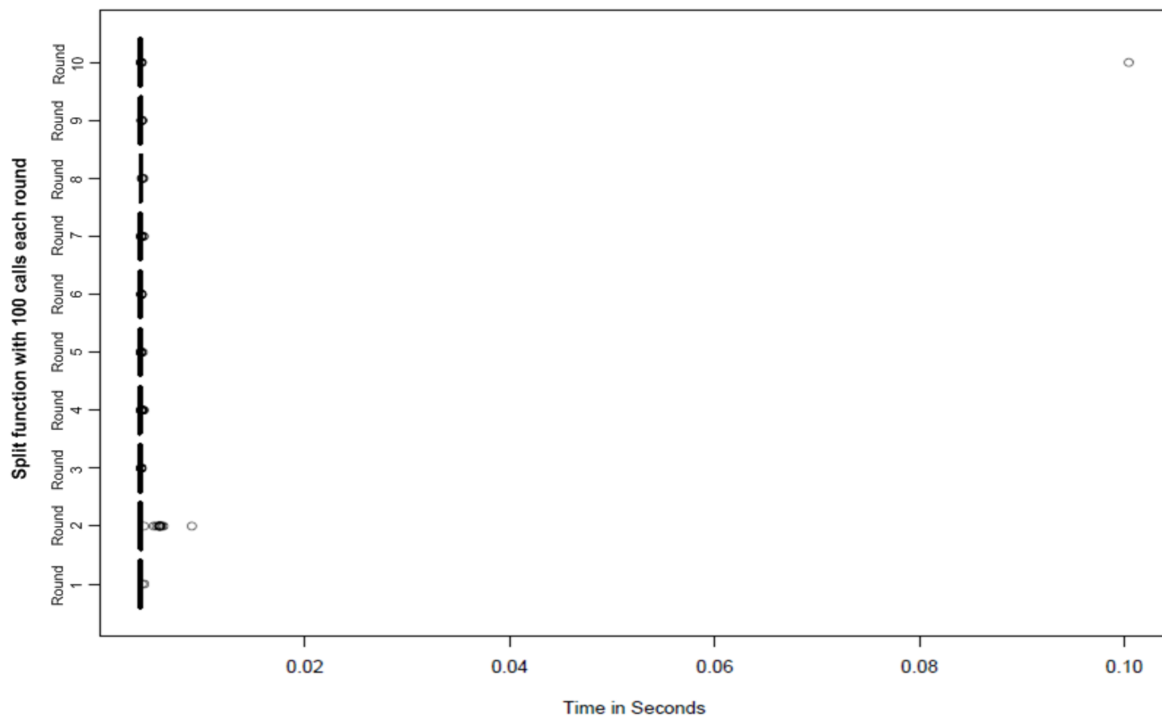

Figure 6-4: Execution time for split function applied to the last 128 bytes of the TGK encoded in base 64

Table 6-2 shows the statistical data found from the time required to split the last 128 bytes of the base 64 TGK value using 1,000 calls of the split function. The unit of time is seconds.

Table 6-2: Statistical data to split second half of TGK encoded in base 64

Count	1000
Mean	0.004
Median	0.0039
Mode	0.0038
Standard Deviation	0.003
Sample Variance	9.38E-06
Minimum	0.0038
Maximum	0.1
Confidence Level (95.0%)	6.07E-06

Some observations can be made from Table 6-1 and Table 6-2. First, the mean time to execute the split function for the first half of TGK encoded in base 64 is 0.0039 seconds and the mean time to execute the split function for the second half of the TGK encoded in base 64 is 0.004 seconds. So, the mean time to execute the split function for escrowing the TGK encoded in base 64 is the sum of these two values i.e. 0.0079 seconds or 7.9 milliseconds. The next observation is that the median values are identical in both experiments. This is expected because the split function is being executed with same security level and same number of bytes; and the median is less affected than the mean by rare outliers. The third and final observation is that the minimum time required to execute the split operation is same for both experiments. This means that the time to split the original 256 byte based 64 encoded TGK is 2 * 0.0038 seconds or 7.6 milliseconds. From this we can observe that the minimum and median time to split the original 256 byte based 64 encoded TGK are quite close (within in 0.002 milliseconds). The maximum observed time for splitting the second half of the TGK is roughly 0.1 seconds; which dominates the maximum time of the first half, thus we can observe that in this set of data the maximum time to perform the splitting of the original 256 byte based 64 encoded TGK is roughly 0.1 second.

6.3 Escrow Time when all 5 Escrow Agents are Available

The chunks have been escrowed over five escrow agents. At first, all the five escrow agents were available and 500 calls were made to escrow the split key chunks. Figure 6-5 shows a box plot for the time required to escrow the TGK when all escrow agents were working.

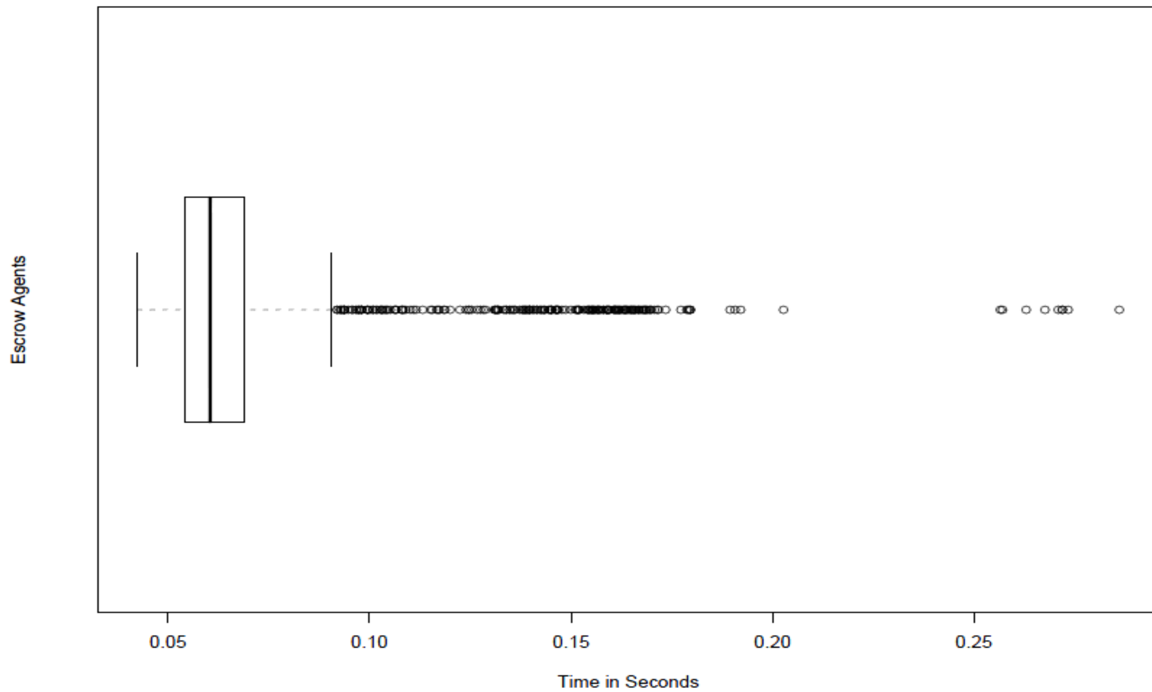

Figure 6-5: Escrow time with 5 escrow agents working

Table 6-3 shows the statistics of the time required to escrow data 500 times while all 5 escrow agents were available (i.e., with 2500 escrow operations). The unit of time is seconds.

Table 6-3: Statistical data for escrowing data (all 5 escrow agents available)

Count	2500
Mean	0.066
Median	0.060
Mode	0.043
Standard Deviation	0.027
Sample Variance	0.0007
Minimum	0.042
Maximum	0.286
Confidence Level (95.0%)	3.4945E-05

Boxplots of the time required for the escrow operation time duration for each of the five escrow agents are shown in Figure 6-6 (note that each row in the plot is based upon 500 samples).

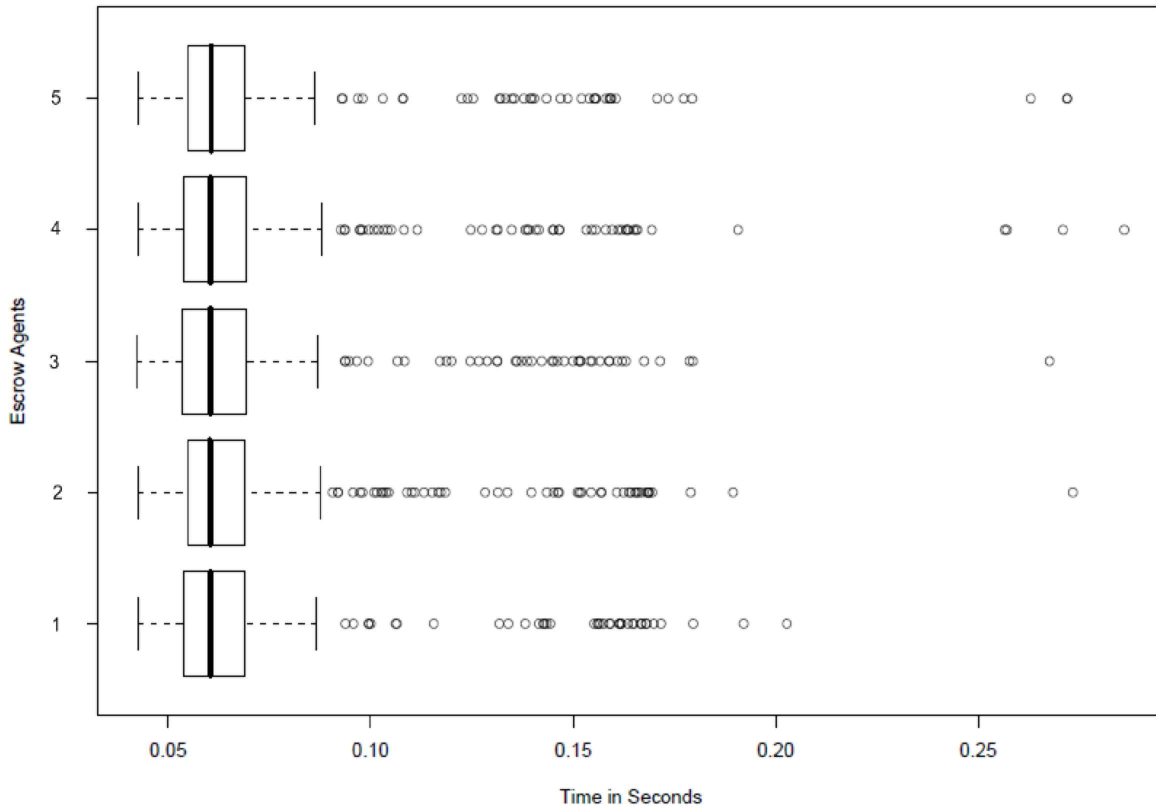

Figure 6-6: Escrow time for 5 escrow agents shown separately

These plots show the time required to escrow the split parts of the TGK with each of the 5 escrow agents. In the next section we consider the case when one of the escrow agents does not respond to an escrow request.

6.4 Escrow Time when 4 out of 5 Escrow Agents are Available

In this test, a single escrow agent was made unavailable. So, 4 out of 5 escrow agents were available. A timeout of 3 seconds were set when sending an escrow request to each escrow agent, hence the user agent waits up to 3 seconds for each escrow agent to respond. If the escrow agent does not respond within these 3 seconds, then the escrow agent is considered to be unavailable and the user agent tries to communicate with the next escrow agent. Figure 6-7 shows the box plot for the time required for the escrow operations when four escrow agents were working.

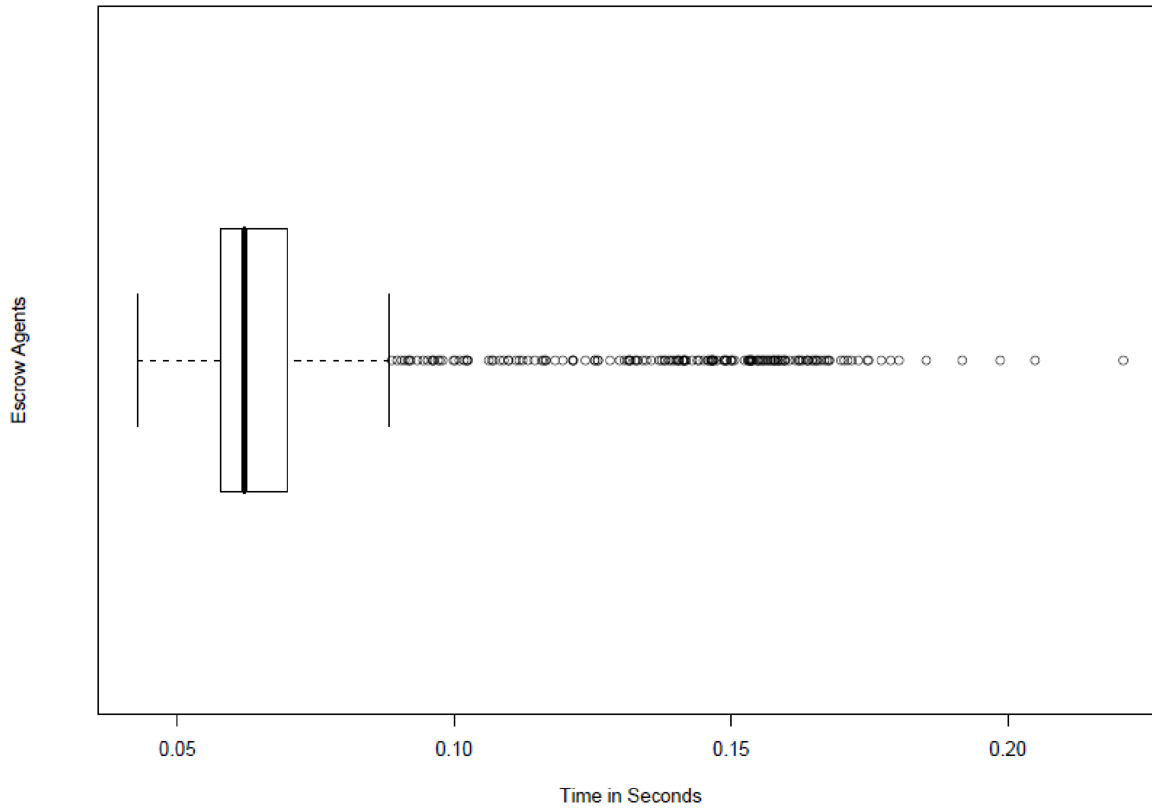

Figure 6-7: Escrow time with 4 escrow agents working

Table 6-4 shows the statistical data found from the time required to escrow data 500 times while 4 escrow agents were available (i.e., with 2000 escrow operations). The unit of time is seconds.

Table 6-4: Statistical data for escrowing data (4 out of 5 escrow agents available)

Count	2000
Mean	0.069
Median	0.062
Mode	0.057
Standard Deviation	0.024
Sample Variance	0.0006
Minimum	0.043
Maximum	0.268
Confidence Level (95.0%)	3.0958E-05

An observation made based upon Table 6-4 is that, a single escrow agent was unavailable and the user agent waited for 3 seconds before timeout. After 3 seconds the user agent tried to escrow a chunk with the next escrow agent. In our experiment, escrow $agent_2$ was made unavailable. It should be noted that we tried to escrow with each of the escrow agents sequentially. Ideally, we should try to escrow the chunks to all the available escrow agents, rather than stopping after reaching the threshold value (i.e. in our case 3). If we escrow chunks with only the threshold number of agents, then later when combining the chunks to compute the key, if one of the agents were unavailable, then the number of chunks available is below the threshold and the system can not recover the key.

Boxplots representing time for the escrow agent to escrow its chunk for each of the four available escrow agents are shown in Figure 6-8 (note that each row in the plot is based upon 500 samples).

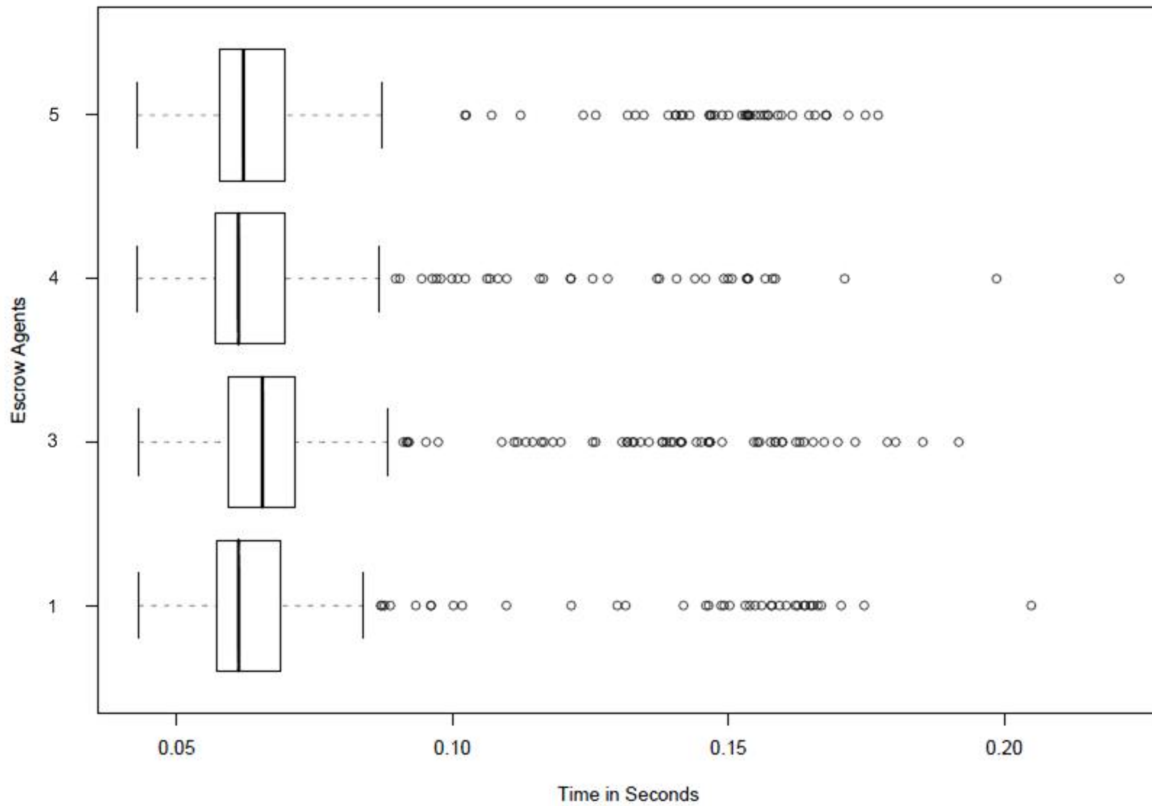

Figure 6-8: Escrow time for the 4 available escrow agents shown separately

6.5 Escrow Time when 3 out of 5 Escrow Agents are Available

Now, one more escrow agent was unavailable. Therefore, only 3 out of 5 escrow agents were available. A timeout of 3 seconds were set. Figure 6-9 shows the box plot for the time duration required to escrow the split TGK with three working escrow agents.

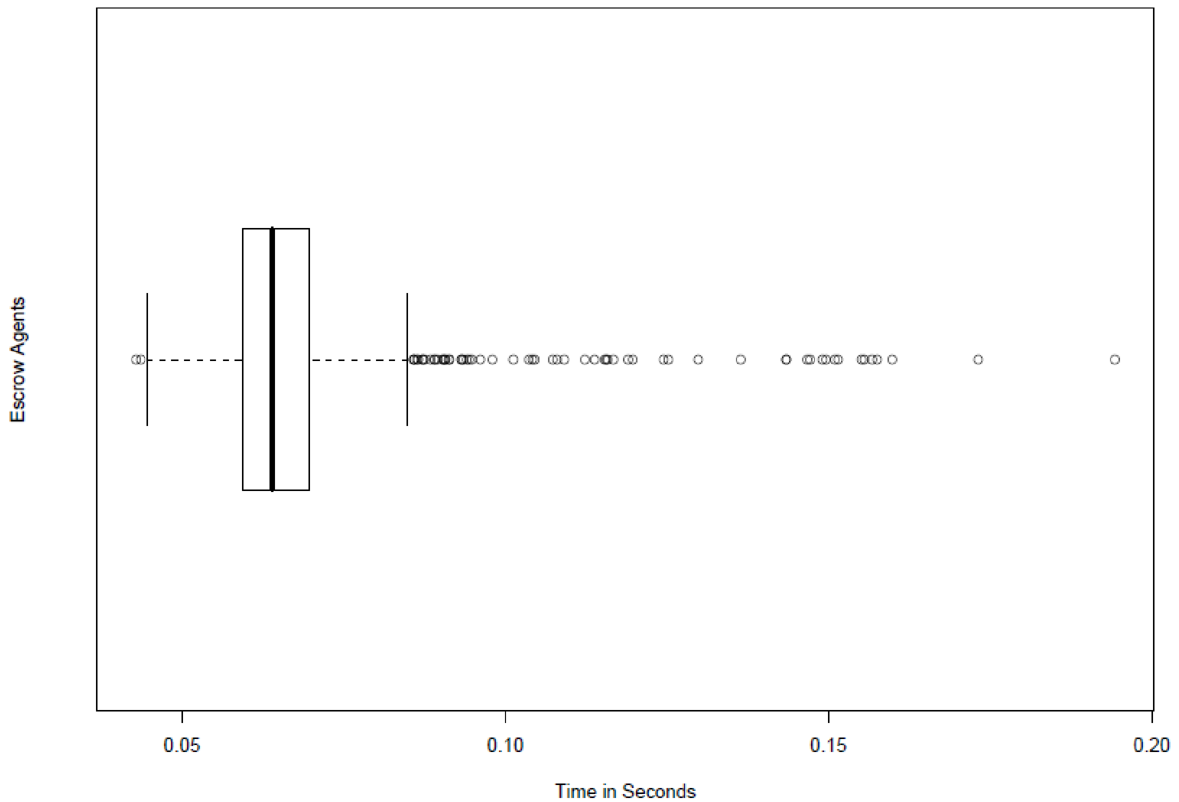

Figure 6-9: Escrow time with 3 escrow agents working

Table 6-5 shows the statistics of the time required to escrow a chunk 500 times when 3 escrow agents were available (i.e., a total of 1500 escrow operations). The unit of time is seconds.

Table 6-5: Statistical data for escrowing data (3 out of 5 escrow agents available)

Count	1500
Mean	0.066
Median	0.064
Mode	0.058
Standard Deviation	0.014
Sample Variance	0.0002
Minimum	0.042
Maximum	0.271
Confidence Level (95.0%)	1.79E-05

Boxplots representing time for escrowing a chunk with each escrow agents are shown in Figure 6-10 (note that each row in the plot is based upon 500 samples).

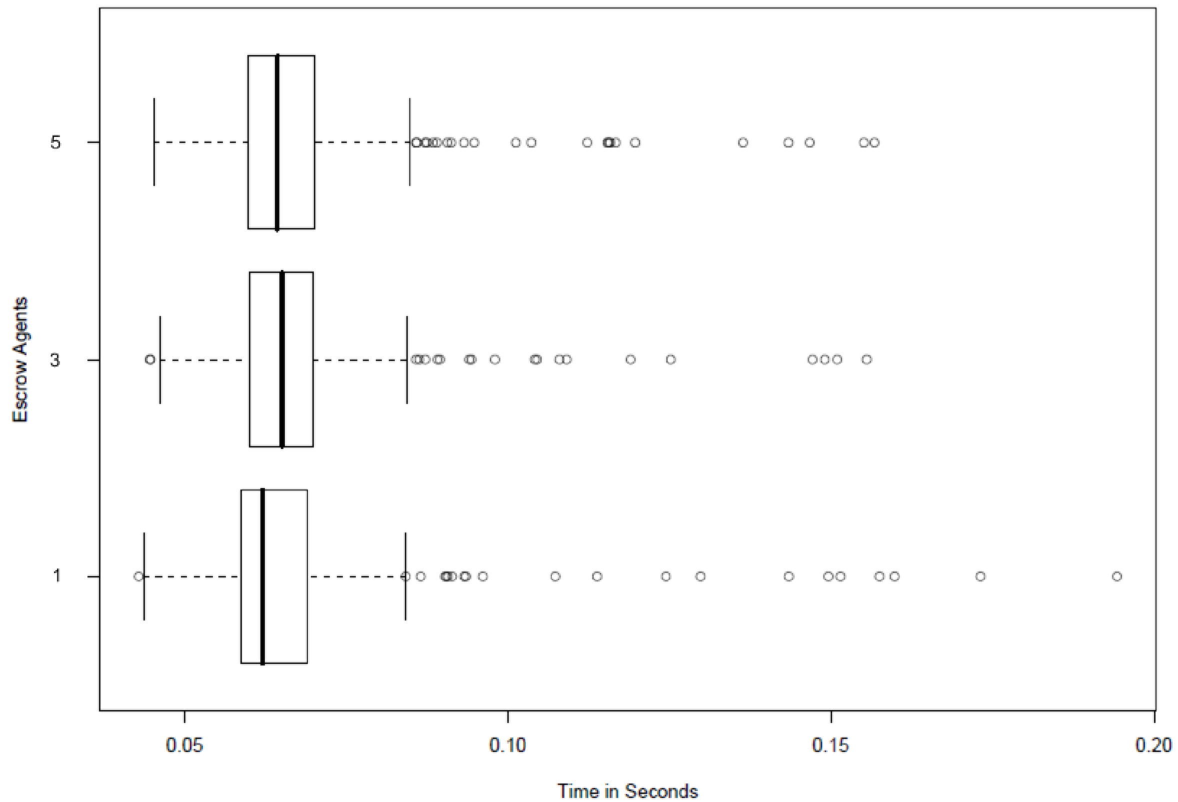

Figure 6-10: Escrow time with 3 escrow agents working shown separately

6.6 Escrow Time with Delay

For the set of experiments described in this section, we manipulated the traffic delay of a link in order to delay the escrow operation. We are interested in understanding the behavior of the escrow operation as a function of the delays in escrowing with different escrow agents.

6.6.1 Manipulating the Traffic Delay for a Specific Link

Linux offers a rich set of traffic control tools for managing and manipulating the effective delay of a link. By using the "tc" command, we can control the packet sending rate between two machines. We can control the characteristics for a specific link (e.g. eth0, eth1, wlan0, etc). For the first set of experiment, we delayed the packet being sent between the user agent and escrow agent$_2$ by 1 second with a variation of 500 milliseconds. In our testing, we used two machines as escrow agents. Both of these machines were configured with five escrow agents. For the first set of experiment we escrowed four chunks of the key with escrow agents running on one machine and the fifth chunk was escrowed to another machine. This fifth chunk will be escrowed by escrow agent$_2$. The command used to control the link in this way was:

```
tc qdisc add dev eth1 root handle 1:0 netem delay 1 sec 500 msec
```

6.6.2 Escrow Time with Delay in 1 Escrow Agent

A delay of 1 second with variation of 500 milliseconds was introduced for escrow agent2. The escrow timeout time was set to 5 seconds. As a consequence of the added link delay, the

information to be escrowed was in fact escrowed, but with a delay. For the rest of the escrow agents, the chunk was escrowed without any delay. We performed escrow operations 500 times for each of the escrow agents and plotted a cumulative distribution graph using Microsoft Excel. This plot is shown in Figure 6-11. The initial knee of the curve at ~0.01 seconds represents the round trip time delay and escrow operation time to the four escrow agents whose communication has not been delay. Since the delay is imposed on the traffic in only one direction (i.e. 1.0 seconds +/-0.5 seconds in one direction) this leads to a round trip delay of 1.0 seconds +/-0.5. Since we have to set up a TCP connection and a TLS tunnel, then due to the count of the number of roundtrips we can see that the delayed escrow operation will take place between ~3 seconds and 5 seconds. Details of the messages exchange to escrow each chunk with an escrow agent are shown in section 6.6.5.

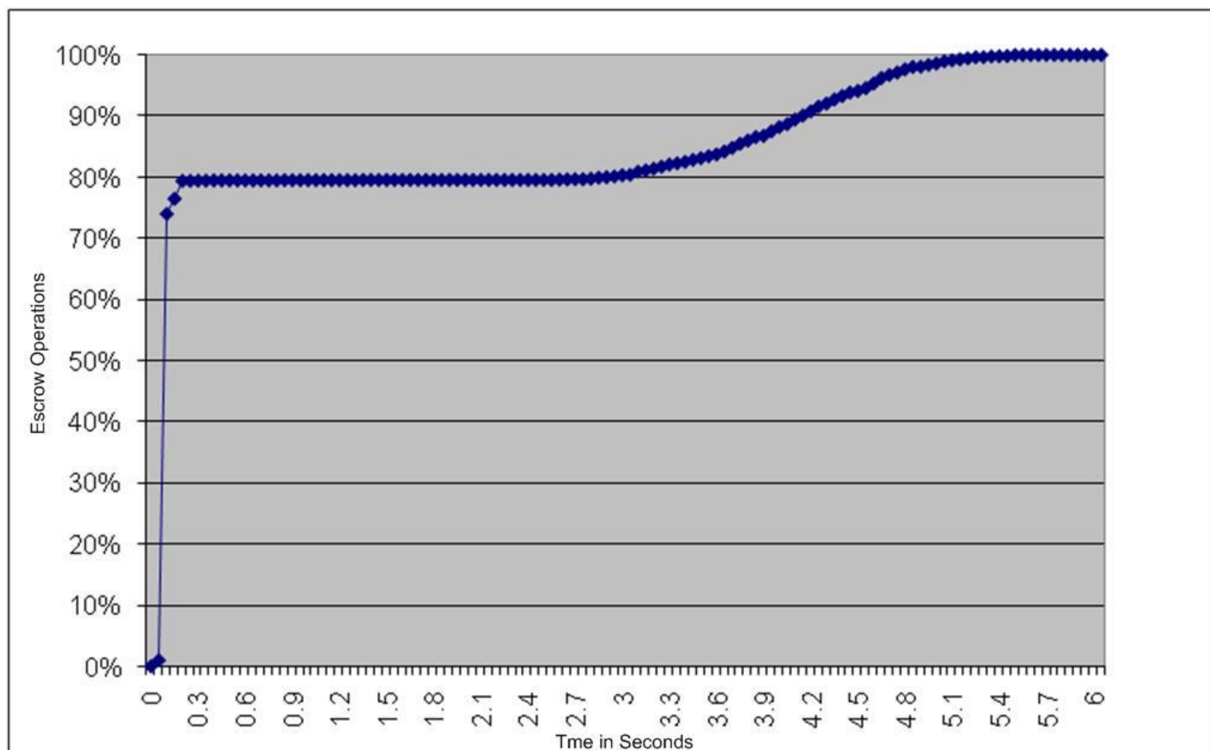

Figure 6-11: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2

Table 6-6 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with a delay of roughly 1 second by one of the escrow agents. The unit of time is seconds.

Table 6-6: Statistics concerning the time required to escrow data with delay of roughly 1 second by one of the escrow agents

Count	2500
Mean	0.896
Median	0.068
Mode	0.060
Standard Deviation	1.648
Sample Variance	2.717
Minimum	0.044
Maximum	5.488
Confidence Level (95.0%)	0.002

6.6.3 Escrow Time with Delays for Two Escrow Agents

The next experiment was very similar to the experiment described in the previous section. In this experiment we delay the traffic to two escrow agents (specifically escrow $agent_2$ and escrow $agent_4$). The amount of delay and delay variation for each link was same as the previous experiment. We performed escrow operations 500 times for each of the escrow agents and plotted a cumulative distribution, see Figure 6-12.

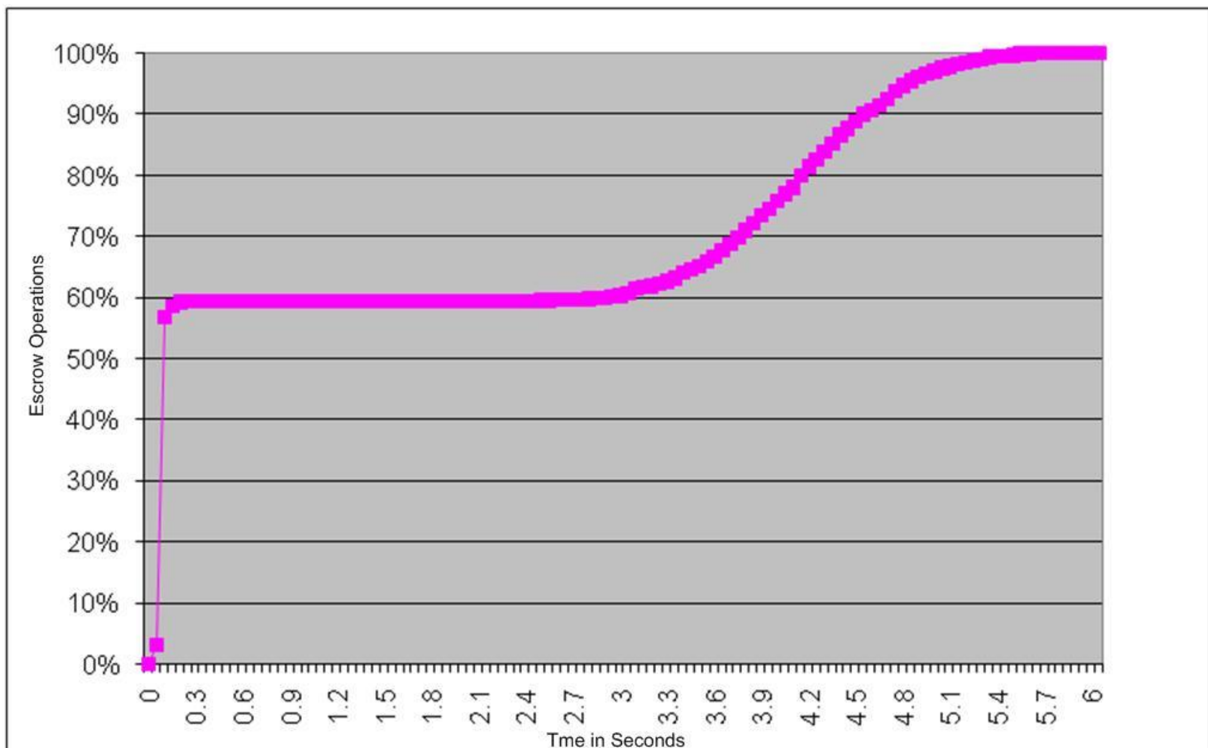

Figure 6-12: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2 and escrow agent 4

Table 6-7 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with delay of roughly 1 second by two of the escrow agents. The unit of time is seconds.

Table 6-7: Statistics concerning the time required to escrow data with delay of roughly 1 second by two of the escrow agents

Count	2500
Mean	1.722
Median	0.078
Mode	0.064
Standard Deviation	2.034
Sample Variance	4.137
Minimum	0.043
Maximum	5.695
Confidence Level (95.0%)	0.003

6.6.4 Escrow Time with Delay for Three the Escrow Agents

The final experiment involved delaying the escrow traffic for 3 escrow agents (specifically escrow $agent_2$, escrow $agent_4$, and escrow $agent_5$). The amount of delay and delay variation was same as in the previous experiments. We performed escrow operations for 500 times each of the escrow agents and plotted a cumulative distribution, see Figure 6-13.

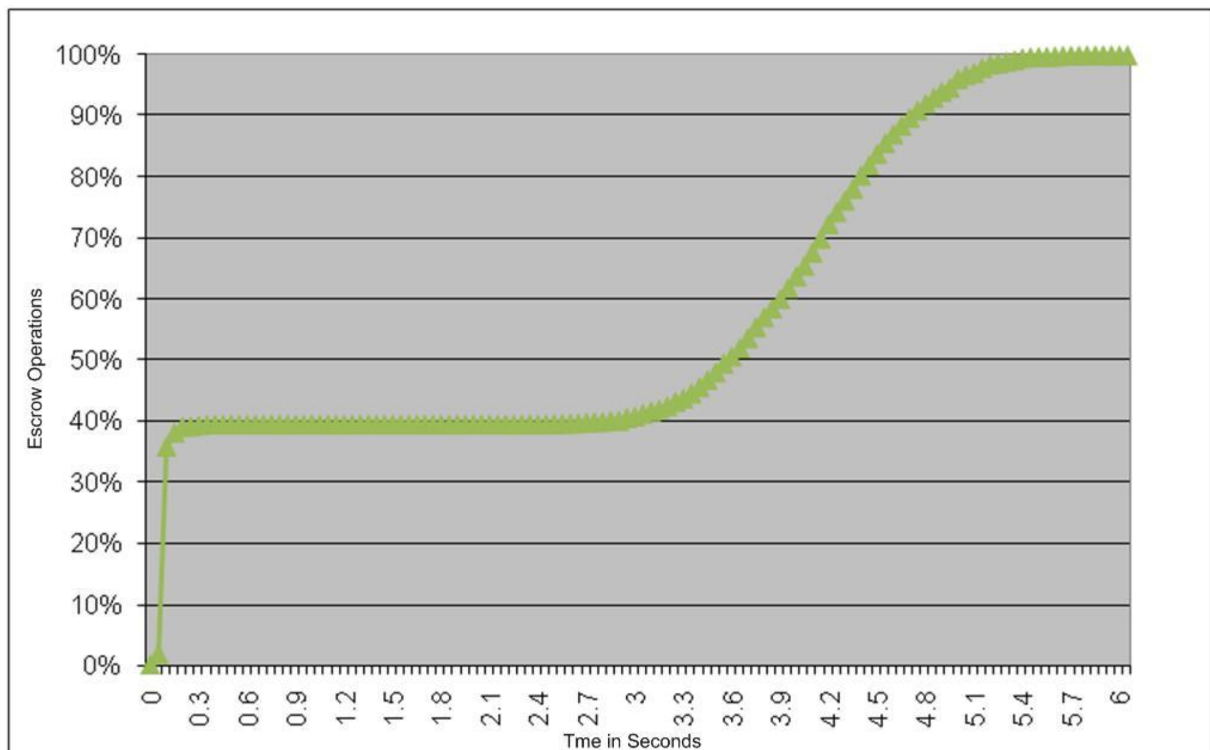

Figure 6-13: Cumulative distribution of Escrow time with a delay of approximately 1 second by escrow agent 2, escrow agent 4, and escrow agent 5

Table 6-8 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with delay of roughly 1 second by three of the escrow agents. The unit of time is seconds.

Table 6-8: Statistics concerning the time required to escrow data with delay of roughly 1 second by three of the escrow agents

Count	2500
Mean	2.539
Median	3.575
Mode	0.058
Standard Deviation	2.037
Sample Variance	4.148
Minimum	0.043
Maximum	5.760
Confidence Level (95.0%)	0.003

6.6.5 Comparison of Escrow Time with Traffic Delay to Different numbers of Escrow Agents

Figure 6-14 shows the combined cumulative distribution curves with delay imposed on the traffic to 1, 2, and 3 escrow agents. We can note that in all cases the escrow operations were successful, despite the added delays for the traffic from the client to the various escrow agents. As we can see the cumulative delay distributions for the three cases are very similar; which is as expected since each of the links was subject to the same distribution of delays.

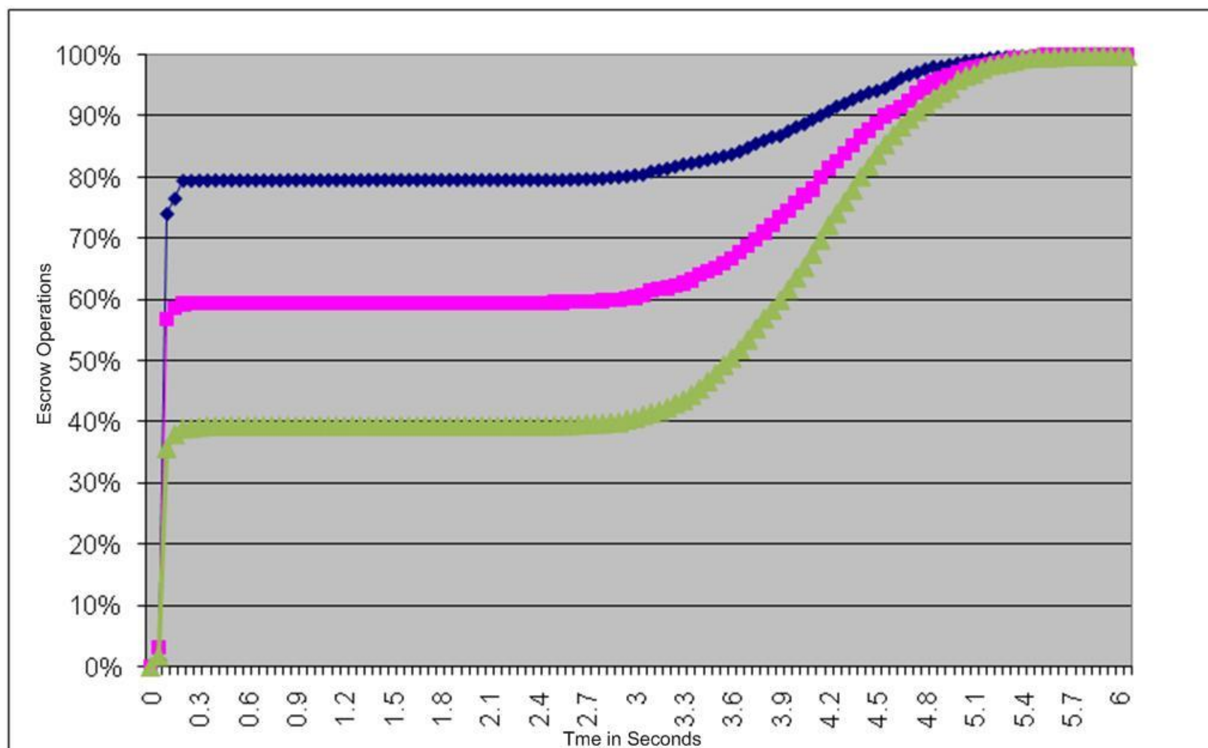

Figure 6-14: Cumulative distribution of Escrow time with a delay of approximately 1 second in 1,2, and 3 escrow agents

Figure 6-15 represents a flow graph of the TCP packets from the user agent to an escrow agent. The figure shows the initial three way TCP handshake needed to establish a TCP connection between the user agent and escrow agent. The user agent sends a SYN message to

the escrow agent. The escrow agent replies with a SYN message. Then the user agent sends an ACK message and the escrow agent replies with another ACK message. After the three way handshake, then a TLS connection needs to be established so that the data can be escrowed.

As mentioned earlier, we set a link delay of 1 second with a variation of 500 milliseconds from the user agent to the escrow agent. Prior to send the data we send 6 packets from the user agent to the escrow agent as shown in Figure 6-15. The first packet sent by the user agent is the SYN packet, the second packet is the client hello request, the third packet is the https ACK packet, the fourth packet is TLS handshake packet, the fifth packet is the TCP previous segment packet and the sixth packet is TCP out of order packet. For each of the packets being sent by the user agent there is a minimum 500 milliseconds delay. So, for the six packets there will be a minimum of 3 seconds delay before data is being escrowed. This is exactly as shown in Figure 6-14. The curve starts rising from 3 seconds of elapsed time. This is due to the increased waiting time before the escrow operation can complete. Here we have ignored the physical delay, interface delay, and processing delay because these delays are very small in comparison to the extra delay that we have introduced. However, the maximum delay to send each packet is 1.5 seconds, thus the maximum time needed to escrow the key would be 9 seconds.

Figure 6-15: Flow graph of the TCP packets from the user agent to an escrow agent

The success rate of escrowing depends on the availability of the escrow agents. The escrow operation is 100% successful when all the 5 escrow agents are available and all the parts of the key are escrowed. However, if one or two escrow agents are not available (i.e., at least 3 escrow agents are available), then we have a partial failure as although the key is still recoverable from at least 3 escrow agents. The system is considered as failure when less than 3 escrow agents are available. This is shown in Figure 6-16. The introduction of the n-out-of-m scheme leads to a partial failure state. This is a unique contribution of this thesis project. Without the redundancy introduced by the n-out-of-m scheme a single failure would lead to

system failure. The n-out-of-m scheme leads to a partial failure state, where the key is recoverable even if only 3-out-of-5 escrow agents are available.

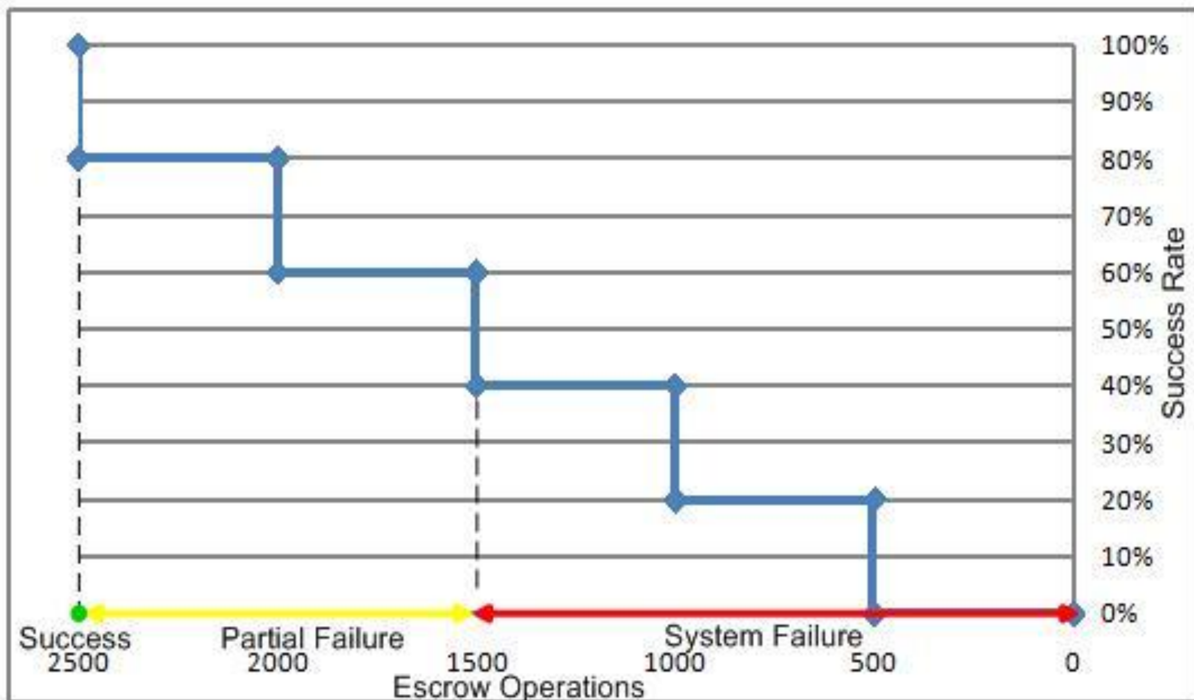

Figure 6-16: Success and failure state of escrow operation

6.7 Escrow Time with Different Numbers of randomly Available/Unavailable Escrow Agents

The experiments described in this section were performed by randomly (according to a defined distribution) making an escrow agent available or unavailable during each escrow operation. For a large number of escrow operations, an escrow agent will be available for some escrow operations, and unavailable for some escrow operations. The escrow agent is made unavailable by setting the link's delay. The link delay is manipulated in such a way that sometimes the delay exceeds the maximum timeout for escrowing (set in these experiments to be 5 seconds) and thus this escrow agent is considered unavailable. We used the TC command for setting the link delay. The delay was set to 1.5 seconds with a uniform variation of 1 second.

```
tc qdisc add dev eth1 root handle 1:0 netem delay 1500 msec 1000 msec
```

6.7.1 Escrow Time with 1 Randomly Available/Unavailable Escrow Agent

The experiment described in this section is performed by making a single escrow agent randomly available/unavailable. The escrow timeout for the user agent was set to 5 seconds and a link delay of 1.5 seconds with uniform variation of 1 second was set between the user agent and escrow agent2. All other escrow agents were available.

We performed escrow operations 500 times for each of the escrow agents and plotted a cumulative distribution, see Figure 6-17.

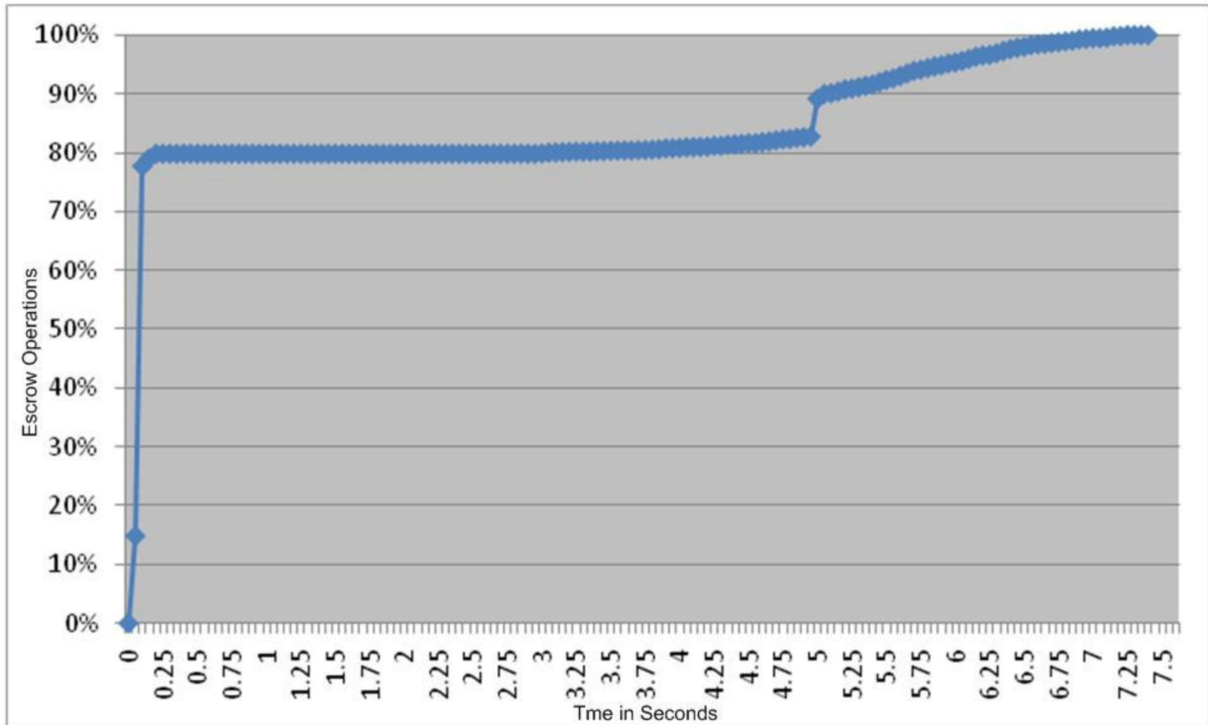

Figure 6-17: Cumulative distribution of Escrow time with 1 randomly available/unavailable escrow agent

Table 6-9 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with one randomly available/unavailable escrow agent. The unit of time is seconds.

Table 6-9: Statistics concerning the time required to escrow data with 1 randomly available/unavailable escrow agent

Count	2500
Mean	1.131
Median	0.062
Mode	0.059
Standard Deviation	2.160
Sample Variance	4.667
Minimum	0.043
Maximum	7.232
Confidence Level (95.0%)	0.003

6.7.2 Escrow Time with 2 Randomly Available/Unavailable Escrow Agents

Now we make an additionally escrow agent randomly available/unavailable and collect a series of measurements. The amount of delay and delay variation was same per agent as in the previous experiment (described in section 6.7.1). We performed escrow operations 500 times for each of the escrow agents and plotted a cumulative distribution, see Figure 6-18.

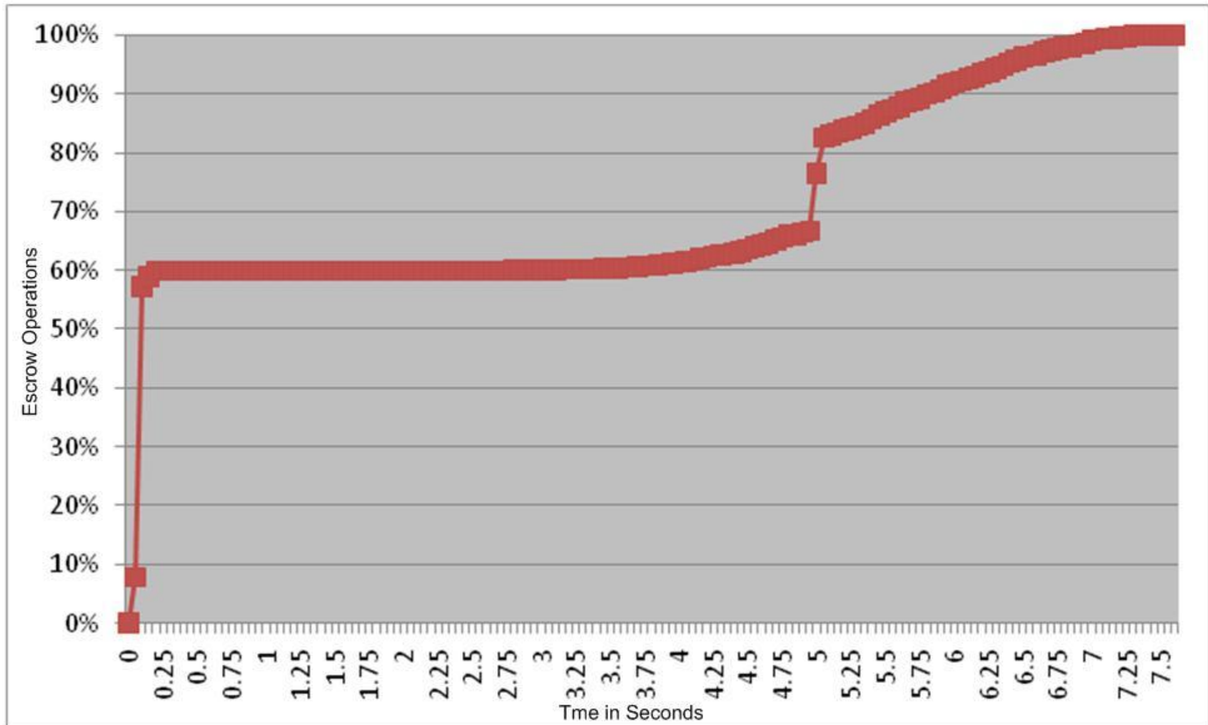

Figure 6-18: Cumulative distribution of Escrow time with 2 randomly available/unavailable escrow agents

Table 6-10 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with 2 available/unavailable escrow agents. The unit of time is seconds.

Table 6-10: Statistics concerning the time required to escrow data with 2 randomly available/unavailable escrow agents

Count	2500
Mean	2.174
Median	0.073
Mode	0.060
Standard Deviation	2.626
Sample Variance	6.895
Minimum	0.042
Maximum	7.457
Confidence Level (95.0%)	0.003

6.7.3 Escrow Time with 3 Randomly Available/Unavailable Escrow Agents

Now we make yet an additionally escrow agent randomly available/unavailable and take a series of measurements. The amount of delay and delay variation was same as the previous two experiments (described in section 6.7.1). We performed escrow operations for 500 times for each of the escrow agents and plotted a cumulative distribution graph, see Figure 6-19.

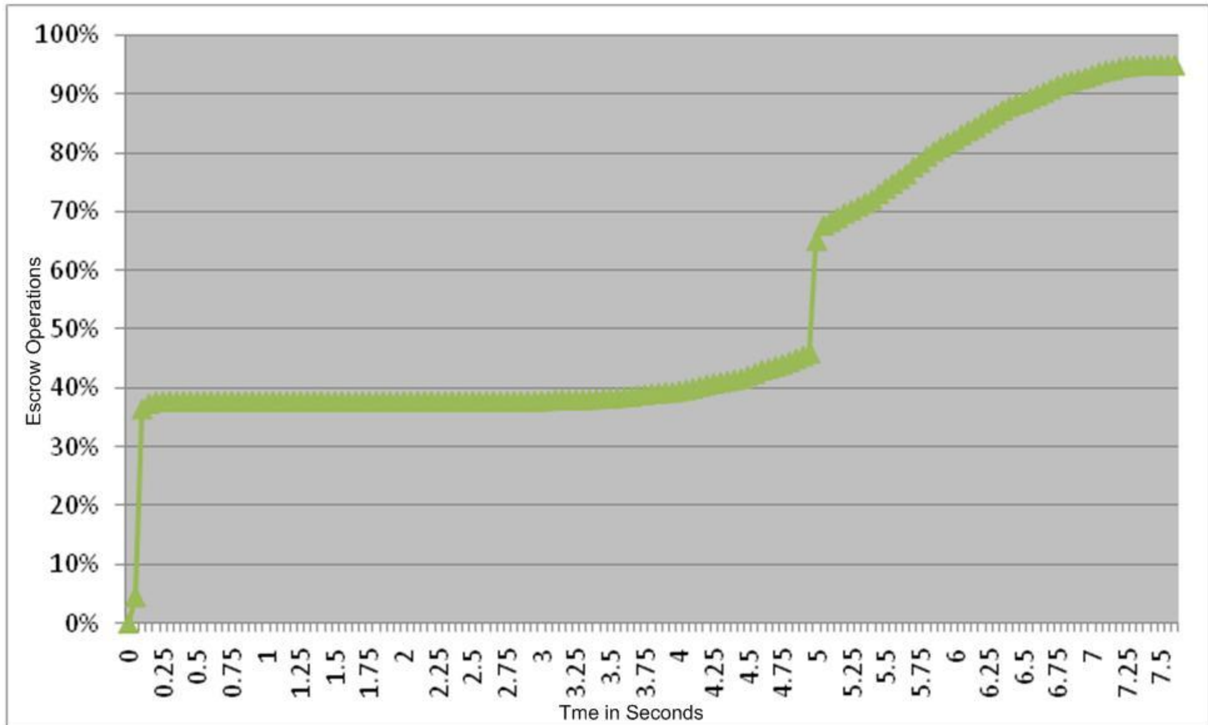

Figure 6-19: Cumulative distribution of Escrow time with 3 randomly available/unavailable escrow agents

An interesting observation from Figure 6-19 is that the curve never reached 100% mark. This was because for some of the escrow operations timeout occurred for each of the three escrow agents and thus became unavailable. Only two chunks were escrowed out of five chunks and this is considered as a failure. In our experiment, we had 25 instances out of 500 escrow operations where there was a failure to escrow in three escrow agents. This made the success rate 95% exactly as shown in Figure 6-19.

Table 6-11 shows the statistics concerning the time required to escrow data 500 times to each of the escrow agents with 3 randomly available/unavailable escrow agents. The unit of time is seconds.

Table 6-11: Statistics concerning the time required to escrow data with 3 randomly available/unavailable escrow agents

Count	2500
Mean	3.266
Median	4.994
Mode	4.997
Standard Deviation	2.670
Sample Variance	7.131
Minimum	0.043
Maximum	7.408
Confidence Level (95.0%)	0.003

6.7.4 Comparison of Escrow Times with 1, 2 and 3 Randomly Available/Unavailable Escrow Agents

Figure 6-20 shows the combined cumulative distribution curves with 1, 2, and 3 randomly available/unavailable escrow agents. The top two curves reached maximum of 100% because

at least three chunks were escrowed with each escrow operation. But failures to escrow three chunks reduced the lower curve to a maximum of less than 100%.

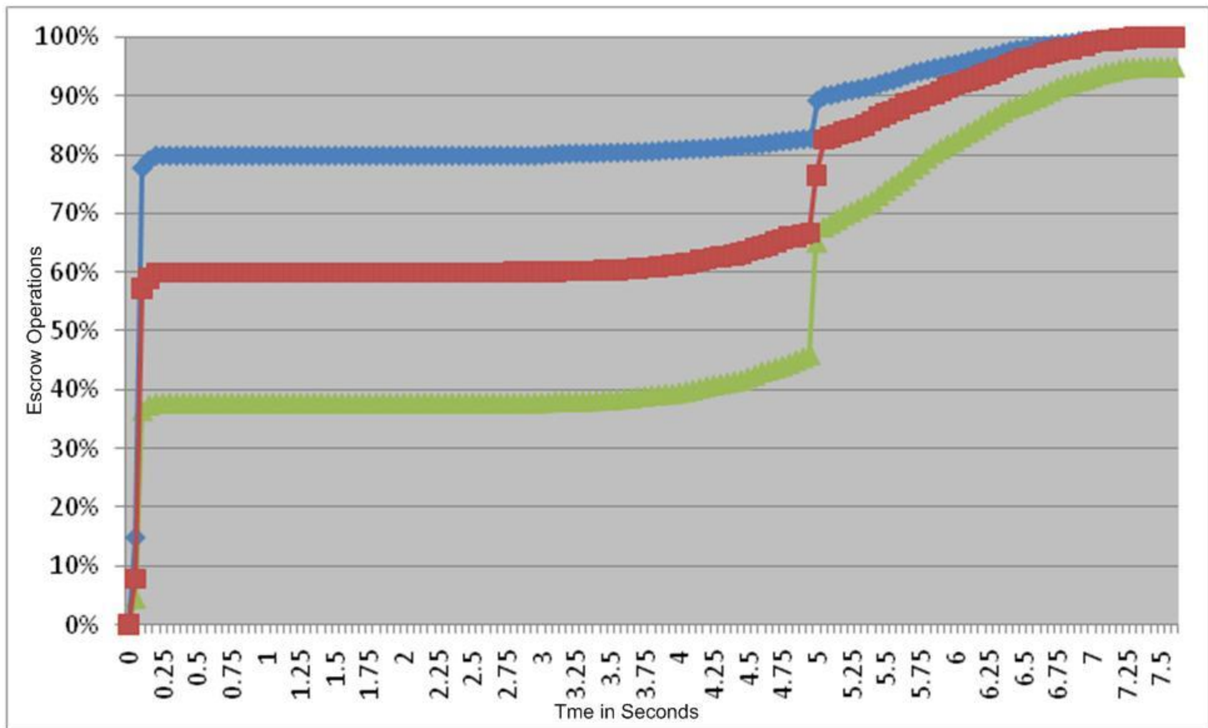

Figure 6-20: Escrow time with 1, 2, and 3 randomly available/unavailable escrow agents

6.8 Availability Measures for LEA

The LEA can request for session keys any time. It is very important from the LEA's point of view that the escrow agent is able to provide the LEA with the key in a timely fashion whenever a legal request has been made. The LEA must present the proper authentication and a legal warrant to the escrow agent in order to retrieve a key. If for some reason an escrow agent goes out of business or suffers from a catastrophic failure, then it becomes impossible for the LEA to retrieve an escrowed key. For example, a LEA might need to retrieve a session key five years after it was escrowed, but could find that the escrow agent who had stored the session key is no longer in business. Or it might happen that the escrow agent is not available 24 hours a day and 365 days of the year. For example, an escrow agent might only be available only during some specific hours of the day, for example due to schedule maintenance or limited business hours. This can create an undesirable situation for the LEA. The risk of this situation is reduced when multiple escrow agents are introduced. In this approach the session key is divided into m chunks and stored with m escrow agents. The key is split in such a way that any n chunks out of m chunks will be sufficient to regenerate the key. Therefore, as long as any n of the m escrow agents supply their part of the split key to the LEA, then the key can be regenerated. This overcomes the problem of one escrow agent going out of business or being unavailable at the time when a request is made.

$\text{The probability that } 1 \text{ or more escrow agents fail}$

$$P(\geq 1 \text{ of } 5) = P^1*(1-P)^4 + P^2*(1-P)^3 + P^3*(1-P)^2 + P^4*(1-P)^1 + P^5*(1-P)^0$$

$\text{The probability that } 2 \text{ or more escrow agents fail}$

$$P(\geq 2 \text{ of } 5) = P^2*(1-P)^3 + P^3*(1-P)^2 + P^4*(1-P)^1 + P^5*(1-P)^0$$

The probability that 3 *or more escrow agents fail*

$P(\geq 3 \text{ of } 5) = P^3*(1-P)^2 + P^4*(1-P)^1 + P^5*(1-P)^0$

where the P on the right side of the equation is the probability that an individual escrow agent fails.

In our case if we have 5 escrow agents with equal failure probability of 0.2. Then according to the above equations, the probability that 1 or more escrow agents fail is 0.11, the probability that 2 or more escrow agents fail is 0.027 and the probability that 3 or more escrow agents fail is 0.007.

If the escrow operations were not all successful, then we can replace P by an **increased** probability that the escrow agent fails. We can see this as:

$$The\ probability\ that\ the\ escrow\ agent\ fails\ (P) * (1 - probability_of_escrow_failure) + (1 - P) * (probability_of_escrow_failure)$$

Here the second term represents the agent working (i.e., not failing) but the original escrow attempt failed, while the first term represents the escrow agent's failure but with a successful escrow operation.

In our n-out-of-m scheme, we have a total of m escrow agents. As one or more escrow agents might be unavailable at any time, the number of active escrow agents will always be less than or equal to m.

In our first experiment, we escrowed the key chunks with 5 escrow agents. In the first experiment, all 5 escrow agents were available. As a result, the probability of key chunks successfully being escrowed was 100%. So, the probability of the key chunks being retrieved directly depends only on the number of active escrow agents when a LEA sends request. As long as at least 3 escrow agents are active, the key can be successfully recovered. The availability of the key chunks depends only on having at least three active escrow agents.

In our next experiment, as described in section 6.7.1, 1 escrow agent was made randomly available/unavailable. In this case some of the key chunks were not escrowed. Now, the probability of key chunks successfully being escrowed with a randomly selected escrow agent dropped to 80%. As a result of the n-of-m redundancy, the probability of a sufficient number of key chunks being retrieved decreases only if more than one escrow agent fails. When a LEA sends a request to the escrow agents; then if all 5 of them are available, then the probability of a sufficient set of chunks being retrieved is 100%, since we only need 3 chunks. However, if two escrow agents fail, then the question is if the three remaining escrow agents actually had the key successfully escrowed with them. We consider this in the next paragraph.

In the experiment described in section 6.7.2, 2 escrow agents were made randomly available/unavailable. In this case some key chunks were escrowed to only 3 escrow agents with a success rate of a randomly selected agent of 60%. As it is a 3-out-of-5 scheme, the probability of retrieving the necessary three chunks also drops. When a LEA sends requests to the escrow agents, then if all 5 of them are available then the probability of key chunks being retrieved is (1-0.027) = 0.973. If 1 escrow agent with a missing chunk is unavailable then the probability of retrieving the key remains 0.973. If 2 escrow agents with missing chunks are unavailable, then the probability of retrieving the key also remains 0.973. However, if one escrow agent with a key chunk is unavailable then the probability of escrow failure is 0.8. In

this case the probability of retrieving a sufficient number of key chunks drops to 0.78, regardless of whether the escrow agent(s) with the missing key chunk(s) are available or unavailable. In this case, the LEA gets only 2 chunks out of 5 and as a result is unable to recover the key. As a result the system is in failed state.

The reliability of key recovery by the LEA depends on how many escrow agents with successfully escrowed keys serve the LEA. The escrow agents might be inactive during some portion of the day. During this time period, any request by the user agent or the LEA will not be served by the escrow agent. As a result the availability of this escrow agent decreases. However, this escrow agent might still contribute to available of the key if it operates correctly during the remaining period of the day. Thus we can see that if there is correlation in the times when escrow agents are unavailable the availability of a sufficient number of chunks of the key rapidly decreases.

It is essential that at least n-out-of-m escrow agents remain active at any time. However, if there is a non-zero probability of the key not being successfully escrowed with all m escrow agents, then we need to have n+1 or more escrow agents operating correctly in order for the LEA to get the necessary n chunks of the key.

Chapter 7: Conclusions and Future Work

Lawful interception (LI) has always been a topic of debate in case of communication. There have been so many changes and modifications in this issue since the early days of communication. From human rights point of view, it seems awkward to record or tap a communication session as it jeopardizes privacy. But from security point of view, it is a must. Lawful interception helps to prevent and identify crimes and criminals.

But not all the employees of the LEA are honest. As human being, an employee of the LEA might be corrupted. A fraudulent request by an evil employee from the LEA can lead to improper disclosure of a session key. After the escrow agent reveals the key the evil person could fabricate data according to his/her needs and encrypt it again by using the correct session key. In this situation the persons involved in the communication session can be accused of crimes that he or she or they never committed. A solution to this problem was implemented by Hossen [2] by escrowing the last signed hash block to the escrow agent. But his solution provided the escrow operation to only a single escrow agent. It is easier for a fraudulent employee to convince a single escrow agent to reveal the session key. The problems with a single escrow agent become critical as a failure of the escrow agent can delay or even make it impossible to reveal the session key.

The basic idea of this thesis project was to implement multiple escrow agents instead of a single escrow agent. And the session key will be divided into these escrow agents in such a manner that neither the key can be retrieved from only a single escrow agent nor a failure of an escrow agent can lead to a situation where the key can not be retrieved. Threshold cryptography was the main concern of the thesis project. Then it was necessary to identify which escrow agents involved in a session. Then it was left to the performance analysis of the implemented system where it was necessary to analyse the reliability and availability of the escrow agents.

7.1 Summary of Achievements

Our first task is to split the session key into chunks. Subsequently these portions of the key are distributed to m escrow agents. In the case of a lawful interception, the LEA only needs to receive, n-out-of-m chunks in order to regenerate the key. Some of the issues we will consider while splitting and regenerating the key are the reliability and availability of the escrow agents.

This thesis project focused on a proposal, implementation, and evaluation of a multiple key escrow agent model that allows escrowing the session keys into m escrow agents. Shamir's secret sharing algorithm was used to implement threshold cryptography. The session key was divided into m chunks and each of the chunks was escrowed to the m escrow agents. An n-out-of-m key retrieval mechanism was implemented. This allowed to retrieve the key by retrieving at least n chunks out of m chunks where the value of n is always less than or equal to the value of m. Practically, the system was implemented with 5 escrow agents with a threshold value of 3. This made it possible divide the session key into 5 chunks to escrow the the chunks into 5 different escrow agents. For retrieving the key, at least 3 chunks out of those 5 chunks were needed. This increased the security as if an LEA officer wants to retrieve the key chunks for a session, he or she has to convince at least 3 escrow agents. It case of fraudulent request, it is difficult to convince all those 3 escrow agents with a fraudulent

notice. On the other hand, the reliability of the key being retrieved properly increase as if any of the 2 escrow agents are unavailable for some reason, the remaining 3 escrow agents can provide the key chunks.

The next achievement of this thesis project was to implement a way to let the LEA know which escrow agents are involved in a session. As we have used multiple escrow agents, the user might choose to escrow on different escrow agents on different sessions. This makes the job of the LEA tougher to identify the escrow agents. We solved this issue by escrowing the names of the escrow agents along with the chunks of the session key and other security parameters. The LEA, while capturing a session, can easily identify from traffic analysis that the session has ended by sending packets to some IP addresses other than the media stream has been sent to. If the LEA finds even one IP address of a escrow agent from this traffic analysis, then it can present a lawful intercept court order to this escrow agent to ask it to reveal the addresses of the other escrow agents from the list that it received key chunks.

A thorough evaluation of the implemented system has been performed as was detailed in Chapter 6. We examined the split operation time for the keys and found it to be a reasonable one. Then we performed the escrow operations with all 5 escrow agents available. The median value for this operation was 0.06 seconds. Then we measured the times needed to escrow the keys with 4 and 3 escrow agents available. The median values for these two experiments were 0.062 seconds and 0.064 seconds respectively. The median values in all three cases were very close to each other and this complied with our expectation. Next, we experimented by imposing some link delays for various escrow agents. First, we experimented by imposing a delay of 0.5 seconds to 1.5 seconds for a single escrow agent. Now the median value increased to 0.068 seconds. Then we performed the experiment with same amount of delay for 2 and 3 escrow agents. The median value for these two experiments increased to 0.078 seconds and 3.575 seconds respectively. We also calculated that the maximum time needed to escrow a key chunk would be 9 seconds. The last set of experiments we performed was escrowing the key chunks by making the escrow agents sometime available and sometime unavailable. An error in communication channel or unavailability of the escrow agents for a short period of time can create this situation in practical scenario. A timeout value of the escrow operation was set to 5 seconds for the user agent. The link delay was adjusted in a way such that the escrow operations are sometimes successful and sometimes unsuccessful. At first, a single escrow agent was made available/unavailable. The median value for the escrow operation in this case was 0.062 seconds. Next we performed the experiments by making 2 and 3 escrow agents available/unavailable and got 0.073 seconds and 4.994 seconds respectively as the median values for these two experiments. Finally, a detail analysis regarding the availability measures for the LEA was performed.

7.2 Future Work

In this project we have implemented a prototype multiple escrow agent based key escrow system with a threshold value. This prototype needs further development before commercial deployment. Some of the issues that should be addressed in future work are:

1. We have used self signed certificates for the SSL connection between the user agent, escrow agent, and the LEA. For commercial application, the certificate handling should utilize the appropriate certificates for each of these actors in order to ensure mutual identification based upon certificates

2. We have stored the split key chunks into files as plain text. These values should be securely stored into a secure database. How to ensure that the received chunk is secured and not available improperly to an employee of the escrow agent should be examined in future work.

3. We have performed the escrow operations for each session at the end of the SIP session. This escrow operation could be done at some later time, for example in a batch processing manner at the end of the day or at some other time. This should be explored in future work.

4. In the current implementation the LEA requests key chunks of a particular session and the escrow agent is expected to respond immediately. Similar to the above, these requests and responses could be done in a batch.

5. An extensive study of error detection and correction has not been done in this thesis project. As per the performance evaluation and analysis, we have found our escrow agents to be reliable up to a threshold value. However, errors may occur in worse cases than we have addressed. Improving the error detection and correction parts of this work should be done to ensure a highly reliable and available system as part of future work.

References

[1] Romanidis Evripidis, "Lawful Interception and Countermeasures: In the era of Internet Telephony", http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/080922-Romanidis_Evripidis-with-cover.pdf, School of Information and Communication Technology (COS/CCS), 2008-20, Royal Institute of Technology (KTH), September 2008.

[2] Md. Sakhawat Hossen, "A Session Initiation Protocol User Agent with Key Escrow: Providing authenticity for recordings of secure sessions", http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/100118-Md._Sakhawat_Hossen-with-cover.pdf, Department of Communication Systems (CoS), Royal Institute of Technology (KTH), TRITA-ICT-EX-2010:1, January 2010.

[3] P.J. Emstad, P.E. Heegaard, B.E. Helvik, L. Paquereau, "Dependability and Performance in Information and Communication Systems - Fundamentals", Department of Telematics, Norwegian University of Science and technology (NTNU), Tapir Akademisk Forlag, 2008.

[4] Theodor W. Schlickmann, Ensuring trust and security in electronic communication, EuroIntel '98 Proceedings, First Annual Conference & Exhibit,Brussels,Belgium, 23-26 March 1998, 1998-XE-08.. http://www.oss.net/dynamaster/file_archive/040319/e12138381ec03c1c6012940f8d0a3136/OSS1998-E1-08.pdf, last visited: January 2010.

[5] "Libcurl-the multiprotocol file transfer library|", http://curl.haxx.se/libcurl/, last visited January, 2010.

[6] Key Escrow, Wikipedia, http://en.wikipedia.org/wiki/Key_escrow, last visited 19/06/2009", last visited: January 2010.

[7] H. Abelson, R. Anderson, S. M. Bellovin, J. Benaloh, M. Blaze, W. Diffie, J. Gilmore, P. G. Neumann, R. L. Rivest, J. I. Schiller, B. Schneier, "The Risks of "Key Recovery," "Key Escrow," And "Trusted Third-Party Encryption", A report by and ad Hoc Group of Cryptographers and computer scientists, http://www.crypto.com/papers/escrowrisks98.pdf, last visited: January 2010.

[8] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", IETF RFC 3550, IETF Network Working Group, July 2003.

[9] M. Baugher, D. McGrew, M. Naslund, E. Carrara, K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", IETF RFC 3711, IETF Network Working Group, March 2004.

[10] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, K. Norrman, "MIKEY: Multimedia Internet KEYing", IETF RFC 3830, IETF Network Working Group, August 2004.

[11] MiniSIP homepage, http://www.minisip.org, last visited: January 2010.

[12] Erik Eliasson, "Secure Internet Telephony: Design, Implementation, and Performance Measurement", Licentiate thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, June 2006.

[13] Israel Abad Caballero, "Secure Mobile Voice over IP, Masters thesis, Royal Institute of Technology (KTH)", School of Information Technology and Microelectronics, June 2003, http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/030626-Israel_Abad_Caballero-final-report.pdf.

[14] Johan Bilien, "Key Agreement for Secure Voice over IP", Masters thesis, Royal Institute of Technology (KTH), School of Information Technology and Microelectronics, IMIT/LCN 2003-14, December 2003

http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/031215-Johan-Bilien-report-final-with-cover.pdf.

[15] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, E. Schooler, "SIP: Session Initiation Protocol", IETF RFC 3261, IETF Network Working Group, June 2002.

[16] Electronic Privacy information Center, "The Clipper Chip", http://epic.org/crypto/clipper/, last visited: January 2010.

[17] SKIPJACK and KEA Algorithm Specifications, Version 2, 29 May 1998, http://csrc.nist.gov/groups/ST/toolkit/documents/skipjack/skipjack.pdf, 29 May 1998.

[18] W. Trappe, L. Washington, "Introduction to Cryptography with Coding Theory", Pearson Prentice Hall, 2nd Edition, 2006.

[19] Clipper Chip Technology, http://csrc.nist.gov/keyrecovery/clip.txt, last visited: January 2010.

[20] The Metaphor is the Key: Cryptography, the Clipper Chip and the Constitution, http://osaka.law.miami.edu/~froomkin/articles/clipper1.htm, last visited: January 2010.

[21] M. Blaze, "Protocol Failure in the Escrowed Encryption Standard", in *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, 2-4 November 1994, ACM Press, pp. 59-67.

[22] H. Zhou, M.W. Mutka, L.M. Ni, "Multiple-key cryptography-based distributed certificate authority in mobile ad-hoc networks", Global Telecommunications Conference, 2005. GLOBECOM '05. IEEE, 28 November-2 December 2005, Vol. 3, ISBN: 0-7803-9414-3.

[23] A. D. Santis, Y. Desmedt, Y. Frankel, M. Yung, " How to Share a Function Securely", *Proceedings of the twenty-sixth annual ACM symposium on Theory of computing*, Montreal, Canada, 1994, pp. 522 – 533, ISBN:0-89791-663-8.

[24] D. Boneh, M. Franklin, "Efficient generation of shared RSA keys", *Journal of the ACM (JACM)*, Volume 48 , Issue 4, July 2001, pp. 702-722, ISSN:0004-5411.

[25] T. Rabin, "A Simplified Approach to Threshold and Proactive RSA", Springer Berlin / Heidelberg, Volume 1462/1998, pp. 349-369, ISBN: 978-3-540-64892-5.

[26] V. Shoup, "Practical Threshold Signatures", Springer Berlin / Heidelberg, Volume 1807/2000, pp. 207-220, ISBN: 978-3-540-67517-4.

[27] H.L. Nguyen, "RSA Threshold Cryptography", http://www.comlab.ox.ac.uk/files/269/Thesis.pdf, Dept. of Computer Science, University of Bristol, May 4, 2005.

[28] Shamirs's Secret Sharing, http://en.wikipedia.org/wiki/Shamir's_Secret_Sharing#Mathematical_definition, last visited: January 2010.

[29] RSA Laboratories, http://www.rsa.com/RSALABS/node.asp?id=2259, last visited: January 2010.

[30] Repetition codes, http://www.mdstud.chalmers.se/~md7sharo/coding/main/node15.html, last visited January 2010.

[31] A Checksum Algorithm, http://www.flounder.com/checksum.htm, last visited: January 2010.

[32] Cyclic Redundancy Check, http://www.hackersdelight.org/crc.pdf, last visited: January 2010.

[33] W.W. Peterson and D.T. Brown, "Cyclic Codes for Error Detection", *Proceedings of the Institute of Radio Engineers (IRE)*, January 1961, Volume: 49, Issue: 1, pp. 228–235, ISSN: 0096-8390.

[34] Andrew S. Tanenbaum, “Computer Networks”, Third Edition. Prentice Hall, 1996, ISBN: 0-13-394248-1.

[35] Introduction to Binary Convolutional codes, http://www.csie.ncnu.edu.tw/~yshan/convolutional_codes.pdf, last visited: January 2010.

[36] Tutorial on Convolutional Coding with Viterbi Decoding, http://home.netcom.com/~chip.f/Viterbi.html, last visited: January 2010.

[37] Calculating the Hamming Code, http://users.cis.fiu.edu/~downeyt/cop3402/hamming.html, last visited: January 2010.

[38] Libcurl-the Multiprotocol File Transfer Library, http://curl.haxx.se/libcurl/, last visited: January 2010.

[39] H. Wallace, “Error Detection and Correction using BCH the Codes”, 2001, http://www.aqdi.com/bch.pdf, last visited: February 2010.

[40] Wikipedia, the free encyclopedia, “Reed-Solomon Error Correction”, http://en.wikipedia.org/wiki/Reed–Solomon_error_correction, last visited: February 2010.

[41] An introduction to Reed-Solomon codes: principles, architecture and implementation, http://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/www/reedsolomon/reed_solomon_codes.html, last visited: February 2010.

[42] Muhammad Sarwar Jahan Morshed , “VoIP Lawful Intercept: Good Cop/Bad Cop”, Master thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, *work in progress*.

[43] Shamir’s Secret Sharing Scheme, http://point-at-infinity.org/ssss/, last visited: April 2010.

[44] GNU General Public License, http://www.gnu.org/licenses/gpl.html, last visited: April 2010.

[45] Madelon F. Zady, “Probability and the Standard Normal Distribution”, http://www.westgard.com/lesson36.htm#tableunder, last visited: May 2010.

[46] Statsoft Electronic Statistics Textbook, http://www.statsoft.com/textbook/distribution-tables/, last visited: May 2010.

Appendices

A. Generating Self-Signed Certificatess on Ubuntu 9.10

1. Tell Apache2 to enable the SSL module.

 # sudo a2enmod ssl

2. Generate our certificate...

 # cd /tmp
 # sudo openssl req -new > new.cert.csr

...when prompted for info, fill it out. Here's what I typed...

SE
Stockholm
KISTA
KTH
(enter)
Abdullah Azfar
azfar@kth.se
(enter)
(enter)

...and now we continue...

 # sudo openssl rsa -in privkey.pem -out new.cert.key
 # sudo openssl x509 -in new.cert.csr -out new.cert.cert -req -signkey new.cert.key -days 1825
 # sudo cp new.cert.cert /etc/ssl/certs/server.crt
 # sudo cp new.cert.key /etc/ssl/private/server.key

3. Now we need to tell Apache2 to use this.

 # sudo cp /etc/apache2/sites-available/default /etc/apache2/sites-available/ssl
 # sudo vi /etc/apache2/sites-available/default

Change:

Code:
```
NameVirtualHost: *
```

To:

Code:
```
NameVirtualHost: *:80
```

Change:

Code:
```
<VirtualHost *>
```

To:

Code:

```
<VirtualHost *:80>
```

 # sudo vi /etc/apache2/sites-available/ssl

Change:

Code:
```
NameVirtualHost: *
```

To:

Code:
```
NameVirtualHost: *:443
```

Change:

Code:
```
<VirtualHost *>
```

To:

Code:
```
<VirtualHost *:443>
```

After the "DocumentRoot" line, add the following:

Code:
```
SSLEngine on
SSLOptions + StrictRequire
SSLCertificateFile /etc/ssl/certs/server.crt
SSLCertificateKeyFile /etc/ssl/private/server.key
```

 # sudo cd /etc/apache2/sites-enabled
 # sudo a2ensite ssl

4. Now we need to adjust /etc/hosts if necessary, using the vi command:

Note this might already be done for you -- just doublecheck.

 # sudo vi /etc/hosts

Code:
```
127.0.0.1 localhost localhost.localdomain {your system name}
127.0.0.1 {your system name}
{static IP if you have one} {Fully qualified domain name if you have one}
```

5. Now we restart our Apache2 service.

 # sudo /etc/init.d/apache2 restart

6. Test your server. You should be able to reach your pages on both http and https. Remember, this goal here was only to get your pages to work on https for doing things like web development testing, such as testing some eCommerce pages. However, you don't want people reaching a secured page on http when they should be on https, so remember that you'll want to trap for that in your .htaccess file in your website folder and redirect users back to the page under https.

B. SSL Enabling Script for Apache Server in OpenSuse 10.3

```
#!/bin/bash
#
## OS: openSuSE 10.3 (may apply to 10.2, but not tested)
#
## This script will build the SSL server keys, csr and crt, install
them, and copy vhosts-ssl.conf
## to the appropriate directory in /etc/apache2 to provide basic
https:// functionality on
## opensuse 10.3
#
## General Functions and Colors
#
green='\e[0;32m'
red='\e[0;31m'
lightred='\e[1;31m'
lightblue='\e[1;34m'
lightgray='\e[0;37m'
nc='\e[0m'

check_root () {

ROOT_UID=0
E_NOTROOT=67

if [ "$UID" -ne "$ROOT_UID" ]; then
echo -e "\n${lightblue}You must be ${lightred}root${lightblue} to
run this script.\nUser: ${lightgray}$USER${lightblue}, UID:
${lightgray}$UID${lightblue} can't!${nc}\n"
exit $E_NOTROOT
# return $E_NOTROOT
else
return $ROOT_UID
fi
}
#
#check for root
#
check_root
#
## Intro Line
#
echo -e "\n\tThis will create apache2 SSL server.key, .csr and .crt
and install them for basic\n https:// functionality on openSuSE
10.3. It will aslo set the apache2 SSL sysconfig flag. \nIn your
key, your common name CN must be a FQDN. You must edit vhost-
ssl.conf when done.\n"
read -p " Continue (y/n)? " key
if [ $key == "y" ] || [ $key == "Y" ]; then
echo -e "${green}\n\tLet's begin!${nc}\n"
else
```

```
echo -e "\n\t${lightgray}key = $key${lightblue} pressed, Apache2 SSL
Config - ${red}Canceled${nc}\n"
exit 1
fi
echo -e "${nc}"
#
## Set SSL Flag
#
if a2enflag SSL; then
echo -e "\n\t${lightblue}Server SSL Flag Successfully Set\n${nc}"
else
echo -e "\n\t${lightblue}Server SSL Flag ${red}NOT
${lightblue}Set\nEdit /etc/sysconfig/apache2 manually\n${nc}"
fi
#
## Create Temp Directory
#
echo -en "\n\t${lightblue}Creating Directory for New SSL KeySet"

if mkdir -p new_sslkeyset && cd new_sslkeyset; then
echo -e " - ${green}OK${nc}\n"
else
echo -e " - ${red}FAILED. Exiting...${nc}\n"
exit 1
fi
#
## Generate Private Server Key
#

echo -e "\n\t${lightblue}Generating Private Server Key\n${nc}"
openssl genrsa -des3 -out server.key 1024

#
## Generate Certificate Signing Request (CSR)
#

echo -e "\n\t${lightblue}Generating Certificate Signing Request
(CSR)\n${nc}"
openssl req -new -key server.key -out server.csr

#
## Remove Passphrase from Key
#
echo -e "\n\t${lightblue}Removing Passphrase From Key To Eliminate
PW Request On Server Start\n${nc}"
cp server.key server.key.protected
openssl rsa -in server.key.protected -out server.key

#
## Generating a Self-Signed Certificate
#

echo -e "\n\t${lightblue}Generating Self-Signed Certificate\n${nc}"
openssl x509 -req -days 3650 -in server.csr -signkey server.key -out
server.crt
```

```
#
## Installing the Private Key and Certificates
#

echo -e "\n\t${lightblue}Installing server.crt, server.key and
server.csr in /etc/apache2/<dir>${nc}\n"

if cp server.crt /etc/apache2/ssl.crt && cp server.key
/etc/apache2/ssl.key && cp server.csr /etc/apache2/ssl.csr; then
echo -e "\n\t${lightblue}Key, CSR and Certificate install
${green}Succeeded${nc}\n"
else
echo -e "\n\t${lightblue}Key, CSR and Certificate install
${red}Failed${nc}\n"
fi
#
## Config Reminder
#
echo -e "${lightblue}\n\tDon't forget to create
/etc/apache2/vhosts.d/vhost-ssl.conf by copying
\n/etc/apache2/vhosts.d/vhost-ssl.template to
/etc/apache2/vhosts.d/vhost-ssl.conf and editing as \nnecessary. You
can check this script for the comments that contain a working
example of a \nvhost-ssl.conf${green}\n"
read -p " Would you like to copy /etc/apache2/vhosts.d/vhost-
ssl.template to vhost-ssl.conf now (y/n)? " key

if [ $key == "y" ] || [ $key == "Y" ]; then
cp /etc/apache2/vhosts.d/vhost-ssl.template
/etc/apache2/vhosts.d/vhost-ssl.conf
fi

echo -e "\n\t${green}All Done! ${lightblue}Remember to edit
${red}vhost-ssl.conf ${lightblue}as required and restart
apache2\n\n${nc}"
read -p " Would you like to see the example vhost-ssl.conf? " key
if [ $key == "y" ] || [ $key == "Y" ]; then
echo '
#
## Virtual Host Configuration (/etc/apache2/vhosts.d/vhost-ssl.conf)
#
<IfDefine SSL>
<IfDefine !NOSSL>
<VirtualHost _default_:443>
DocumentRoot "/srv/www/htdocs"
fix -> #ServerName www.yourhost.com:443
-> #ServerAdmin youremail@xxxxxxxxxxxx
ErrorLog /var/log/apache2/error_log
TransferLog /var/log/apache2/access_log
SSLEngine on
SSLCipherSuite
ALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SSLv2:+EXP:+eNULL
SSLCertificateFile /etc/apache2/ssl.crt/server.crt
SSLCertificateKeyFile /etc/apache2/ssl.key/server.key
SSLOptions +FakeBasicAuth +ExportCertData +StrictRequire
<Files ~ "\.(cgi|shtml|phtml|php3?)$">
```

```
SSLOptions +StdEnvVars
</Files>
<Directory "/srv/www/cgi-bin">
SSLOptions +StdEnvVars
</Directory>
SetEnvIf User-Agent ".*MSIE.*" \
nokeepalive ssl-unclean-shutdown \
downgrade-1.0 force-response-1.0
CustomLog /var/log/apache2/ssl_request_log ssl_combined
</VirtualHost>
</IfDefine>
</IfDefine>'
fi
exit 0
```

C. Shamir's Secret Sharing Algorithm Code (Common parts: To be added in both the User Agent and the LEA)

```
/*
 *  ssss version 0.5  -  Copyright 2005,2006 B. Poettering
 *  http://point-at-infinity.org/ssss/
 *  Modifications made by Abdullah Azfar in 2010
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  http://www.gnu.org/licenses/gpl.html
 *  as published by the Free Software Foundation; either version 2
 *  of the License, or (at your option) any later version.
*/

#include <stdlib.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>
#include <stdio.h>
#include <stdint.h>
#include <assert.h>
#include <termios.h>
#include <sys/mman.h>
#include <fstream.h>
#include "/usr/include/gmp.h"
#include<boost/date_time/posix_time/posix_time.hpp>
using namespace boost::posix_time;

#define mpz_lshift(A, B, l) mpz_mul_2exp(A, B, l)
#define mpz_sizeinbits(A) (mpz_cmp_ui(A, 0) ? mpz_sizeinbase(A, 2) :
0)
#define RANDOM_SOURCE "/dev/urandom"
#define MAXDEGREE 1024
#define MAXTOKENLEN 128
```

```
#define MAXLINELEN (MAXTOKENLEN + 1 + 10 + 1 + MAXDEGREE / 4 + 10)

static const unsigned char irred_coeff[] = {
4,3,1,5,3,1,4,3,1,7,3,2,5,4,3,5,3,2,7,4,2,4,3,1,10,9,3,9,4,2,7,6,2,1
0,9,6,4,3,1,5,4,3,4,3,1,7,2,1,5,3,2,7,4,2,6,3,2,5,3,2,15,3,2,11,3,2,
9,8,7,7,2,1,5,3,2,9,3,1,7,3,1,9,8,3,9,4,2,8,5,3,15,14,10,10,5,2,9,6,
2,9,3,2,9,5,2,11,10,1,7,3,2,11,2,1,9,7,4,4,3,1,8,3,1,7,4,1,7,2,1,13,
11,6,5,3,2,7,3,2,8,7,5,12,3,2,13,10,6,5,3,2,5,3,2,9,5,2,9,7,2,13,4,3
,4,3,1,11,6,4,18,9,6,19,18,13,11,3,2,15,9,6,4,3,1,16,5,2,15,14,6,8,5
,2,15,11,2,11,6,2,7,5,3,8,3,1,19,16,9,11,9,6,15,7,6,13,4,3,14,13,3,1
3,6,3,9,5,2,19,13,6,19,10,3,11,6,5,9,2,1,14,3,2,13,3,1,7,5,4,11,9,8,
11,6,5,23,16,9,19,14,6,23,10,2,8,3,2,5,4,3,9,6,4,4,3,2,13,8,6,13,11,
1,13,10,3,11,6,5,19,17,4,15,14,7,13,9,6,9,7,3,9,7,1,14,3,2,11,8,2,11
,6,4,13,5,2,11,5,1,11,4,1,19,10,3,21,10,6,13,3,1,15,7,5,19,18,10,7,5
,3,12,7,2,7,5,1,14,9,6,10,3,2,15,13,12,12,11,9,16,9,7,12,9,3,9,5,2,1
7,10,6,24,9,3,17,15,13,5,4,3,19,17,8,15,6,3,19,6,1 };

int opt_showversion = 0;
int opt_help = 0;
int opt_quiet = 0;
int opt_QUIET = 0;
int opt_hex = 0;
int opt_diffusion = 1;
int opt_security = 0;
int opt_threshold = -1;
int opt_number = -1;
char *opt_token = NULL;

int totalea=5;
int minea=3;

unsigned int degree;
mpz_t poly;
int cprng;
struct termios echo_orig, echo_off;

FILE *output;
FILE *output2;
int filnamenumber;

void split(char *buf);

/* emergency abort and warning functions */

void fatal(char *msg)
{
  tcsetattr(0, TCSANOW, &echo_orig);
  fprintf(stderr, "%sFATAL: %s.\n", isatty(2) ? "\a" : "", msg);
  exit(1);
}

void warning(char *msg)
{
  if (! opt_QUIET)
    fprintf(stderr, "%sWARNING: %s.\n", isatty(2) ? "\a" : "", msg);
}
```

```
/* field arithmetic routines */

int field_size_valid(int deg)
{
  return (deg >= 8) && (deg <= MAXDEGREE) && (deg % 8 == 0);
}

/* initialize 'poly' to a bitfield representing the coefficients of
an
   irreducible polynomial of degree 'deg' */

void field_init(int deg)
{
  assert(field_size_valid(deg));
  mpz_init_set_ui(poly, 0);
  mpz_setbit(poly, deg);
  mpz_setbit(poly, irred_coeff[3 * (deg / 8 - 1) + 0]);
  mpz_setbit(poly, irred_coeff[3 * (deg / 8 - 1) + 1]);
  mpz_setbit(poly, irred_coeff[3 * (deg / 8 - 1) + 2]);
  mpz_setbit(poly, 0);
  degree = deg;
}

void field_deinit(void)
{
  mpz_clear(poly);
}

/* I/O routines for GF(2^deg) field elements */

void field_import(mpz_t x, const char *s, int hexmode)
{
  if (hexmode) {
    if (strlen(s) > degree / 4)
      fatal("input string too long");
    if (strlen(s) < degree / 4)
      warning("input string too short, adding null padding on the
left");
    if (mpz_set_str(x, s, 16) || (mpz_cmp_ui(x, 0) < 0))
      fatal("invalid syntax");
  }
  else {
    int i;
    int warn = 0;
    if (strlen(s) > degree / 8)
      fatal("input string too long");
    for(i = strlen(s) - 1; i >= 0; i--)
      warn = warn || (s[i] < 32) || (s[i] >= 127);
    if (warn)
      warning("binary data detected, use -x mode instead");
    mpz_import(x, strlen(s), 1, 1, 0, 0, s);
  }
}
```

```
/* basic field arithmetic in GF(2^deg) */

void field_add(mpz_t z, const mpz_t x, const mpz_t y)
{
  mpz_xor(z, x, y);
}

void field_mult(mpz_t z, const mpz_t x, const mpz_t y)
{
  mpz_t b;
  unsigned int i;
  assert(z != y);
  mpz_init_set(b, x);
  if (mpz_tstbit(y, 0))
    mpz_set(z, b);
  else
    mpz_set_ui(z, 0);
  for(i = 1; i < degree; i++) {
    mpz_lshift(b, b, 1);
    if (mpz_tstbit(b, degree))
      mpz_xor(b, b, poly);
    if (mpz_tstbit(y, i))
      mpz_xor(z, z, b);
  }
  mpz_clear(b);
}

void field_invert(mpz_t z, const mpz_t x)
{
  mpz_t u, v, g, h;
  int i;
  assert(mpz_cmp_ui(x, 0));
  mpz_init_set(u, x);
  mpz_init_set(v, poly);
  mpz_init_set_ui(g, 0);
  mpz_set_ui(z, 1);
  mpz_init(h);
  while (mpz_cmp_ui(u, 1)) {
    i = mpz_sizeinbits(u) - mpz_sizeinbits(v);
    if (i < 0) {
      mpz_swap(u, v);
      mpz_swap(z, g);
      i = -i;
    }
    mpz_lshift(h, v, i);
    mpz_xor(u, u, h);
    mpz_lshift(h, g, i);
    mpz_xor(z, z, h);
  }
  mpz_clear(u); mpz_clear(v); mpz_clear(g); mpz_clear(h);
}

/* routines for the random number generator */

void cprng_init(void)
{
```

```
  if ((cprng = open(RANDOM_SOURCE, O_RDONLY)) < 0)
    fatal("couldn't open " RANDOM_SOURCE);
}

void cprng_deinit(void)
{
  if (close(cprng) < 0)
    fatal("couldn't close " RANDOM_SOURCE);
}

void cprng_read(mpz_t x)
{
  char buf[MAXDEGREE / 8];
  unsigned int count;
  int i;
  for(count = 0; count < degree / 8; count += i)
    if ((i = read(cprng, buf + count, degree / 8 - count)) < 0) {
      close(cprng);
      fatal("couldn't read from " RANDOM_SOURCE);
    }
  mpz_import(x, degree / 8, 1, 1, 0, 0, buf);
}

/* a 64 bit pseudo random permutation (based on the XTEA cipher) */

void encipher_block(uint32_t *v)
{
  uint32_t sum = 0, delta = 0x9E3779B9;
  int i;
  for(i = 0; i < 32; i++) {
    v[0] += (((v[1] << 4) ^ (v[1] >> 5)) + v[1]) ^ sum;
    sum += delta;
    v[1] += (((v[0] << 4) ^ (v[0] >> 5)) + v[0]) ^ sum;
  }
}

void decipher_block(uint32_t *v)
{
  uint32_t sum = 0xC6EF3720, delta = 0x9E3779B9;
  int i;
  for(i = 0; i < 32; i++) {
    v[1] -= ((v[0] << 4 ^ v[0] >> 5) + v[0]) ^ sum;
    sum -= delta;
    v[0] -= ((v[1] << 4 ^ v[1] >> 5) + v[1]) ^ sum;
  }
}

void encode_slice(uint8_t *data, int idx, int len,
             void (*process_block)(uint32_t*))
{
  uint32_t v[2];
  int i;
  for(i = 0; i < 2; i++)
    v[i] = data[(idx + 4 * i) % len] << 24 |
      data[(idx + 4 * i + 1) % len] << 16 |
      data[(idx + 4 * i + 2) % len] << 8 | data[(idx + 4 * i + 3) %
```

```
len];
  process_block(v);
  for(i = 0; i < 2; i++) {
    data[(idx + 4 * i + 0) % len] = v[i] >> 24;
    data[(idx + 4 * i + 1) % len] = (v[i] >> 16) & 0xff;
    data[(idx + 4 * i + 2) % len] = (v[i] >> 8) & 0xff;
    data[(idx + 4 * i + 3) % len] = v[i] & 0xff;
  }
}

enum encdec {ENCODE, DECODE};

void encode_mpz(mpz_t x, enum encdec encdecmode)
{
  uint8_t v[(MAXDEGREE + 8) / 16 * 2];
  size_t t;
  int i;
  memset(v, 0, (degree + 8) / 16 * 2);
  mpz_export(v, &t, -1, 2, 1, 0, x);
  if (degree % 16 == 8)
    v[degree / 8 - 1] = v[degree / 8];
  if (encdecmode == ENCODE)             /* 40 rounds are more than
enough!*/
    for(i = 0; i < 40 * ((int)degree / 8); i += 2)
      encode_slice(v, i, degree / 8, encipher_block);
  else
    for(i = 40 * (degree / 8) - 2; i >= 0; i -= 2)
      encode_slice(v, i, degree / 8, decipher_block);
  if (degree % 16 == 8) {
    v[degree / 8] = v[degree / 8 - 1];
    v[degree / 8 - 1] = 0;
  }
  mpz_import(x, (degree + 8) / 16, -1, 2, 1, 0, v);
  assert(mpz_sizeinbits(x) <= degree);
}

/* evaluate polynomials efficiently */

void horner(int n, mpz_t y, const mpz_t x, const mpz_t coeff[])
{
  int i;
  mpz_set(y, x);
  for(i = n - 1; i; i--) {
    field_add(y, y, coeff[i]);
    field_mult(y, y, x);
  }
  field_add(y, y, coeff[0]);
}

/* calculate the secret from a set of shares solving a linear
equation system */

#define MPZ_SWAP(A, B) \
  do { mpz_set(h, A); mpz_set(A, B); mpz_set(B, h); } while(0)

//int restore_secret(int n, mpz_t (*A)[n], mpz_t b[])
```

```
int restore_secret(int n, void *A, mpz_t b[])
{
  mpz_t (*AA)[n] = (mpz_t (*)[n])A;
  int i, j, k, found;
  mpz_t h;
  mpz_init(h);
  for(i = 0; i < n; i++) {
    if (! mpz_cmp_ui(AA[i][i], 0)) {
      for(found = 0, j = i + 1; j < n; j++)
     if (mpz_cmp_ui(AA[i][j], 0)) {
       found = 1;
       break;
     }
      if (! found)
     return -1;
      for(k = i; k < n; k++)
     MPZ_SWAP(AA[k][i], AA[k][j]);
      MPZ_SWAP(b[i], b[j]);
    }
    for(j = i + 1; j < n; j++) {
      if (mpz_cmp_ui(AA[i][j], 0)) {
     for(k = i + 1; k < n; k++) {
       field_mult(h, AA[k][i], AA[i][j]);
       field_mult(AA[k][j], AA[k][j], AA[i][i]);
       field_add(AA[k][j], AA[k][j], h);
     }
     field_mult(h, b[i], AA[i][j]);
     field_mult(b[j], b[j], AA[i][i]);
     field_add(b[j], b[j], h);
      }
    }
  }
  field_invert(h, AA[n - 1][n - 1]);
  field_mult(b[n - 1], b[n - 1], h);
  mpz_clear(h);
  return 0;
}
```

D. Shamir's Secret Sharing Algorithm Code (To be added only in the User Agent)

```
/*
 *  ssss version 0.5  -  Copyright 2005,2006 B. Poettering
 *  http://point-at-infinity.org/ssss/
 *  Modifications made by Abdullah Azfar in 2010
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  http://www.gnu.org/licenses/gpl.html
 *  as published by the Free Software Foundation; either version 2
```

```
 *  of the License, or (at your option) any later version.
*/

void field_print(FILE* stream, const mpz_t x, int hexmode)
{
  int i;
  if (hexmode) {
    for(i = degree / 4 - mpz_sizeinbase(x, 16); i; i--)
      fprintf(output, "0");
      mpz_out_str(output, 16, x);
    fprintf(output,"%%");
    rewind(output);
  }
  else {
    char buf[MAXDEGREE / 8 + 1];
    size_t t;
    unsigned int i;
    int printable, warn = 0;
    memset(buf, degree / 8 + 1, 0);
    mpz_export(buf, &t, 1, 1, 0, 0, x);
    for(i = 0; i < t; i++) {
      printable = (buf[i] >= 32) && (buf[i] < 127);
      warn = warn || ! printable;
      fprintf(stream, "%c", printable ? buf[i] : '.');
    }
    if (warn)
      warning("binary data detected, use -x mode instead");
  }
}


/* Prompt for a secret, generate shares for it */

void split(char *buf)
{
  opt_threshold = minea;
  opt_number = totalea;
  unsigned int fmt_len;
  mpz_t x, y, coeff[opt_threshold];

  int deg, i;
  int filnum=1;
  opt_security=0;
  //printf("\nLength of buffer is %d \n", strlen(buf));

  for(fmt_len = 1, i = opt_number; i >= 10; i /= 10, fmt_len++);

  buf[strcspn(buf, "\r\n")] = '\0';

  if (! opt_security) {
    opt_security = opt_hex ? 4 * ((strlen(buf) + 1) & ~1): 8 *
strlen(buf);
    if (! field_size_valid(opt_security))
      fatal("security level invalid (secret too long?)");
    opt_security);
  }
```

```
  field_init(opt_security);

  mpz_init(coeff[0]);
  field_import(coeff[0], buf, opt_hex);

  if (opt_diffusion) {
    if (degree >= 64)
      encode_mpz(coeff[0], ENCODE);
    else
      warning("security level too small for the diffusion layer");
  }

  cprng_init();
  for(i = 1; i < opt_threshold; i++) {
    mpz_init(coeff[i]);
    cprng_read(coeff[i]);
  }
  cprng_deinit();

  mpz_init(x);
  mpz_init(y);
  for(i = 0; i < opt_number; i++) {
    mpz_set_ui(x, i + 1);
    horner(opt_threshold, y, x, (const mpz_t*)coeff);

    char filname[15];
    char filnamenumberstring[20];
    char filnametmp[15]="ea.out";

    strcpy(filname, filnametmp);
    sprintf(filnamenumberstring,"%d",filnamenumber);
    strcat(filname, filnamenumberstring);
    filnamenumber++;
    output=fopen(filname,"a+");

    if (opt_token)
      fprintf(stdout,"%0*d-", fmt_len, i + 1);
    fprintf(output,"%0*d-", fmt_len, i + 1);
    field_print(stdout, y, 1);
    filnum++;
  }
  mpz_clear(x);
  mpz_clear(y);

  for(i = 0; i < opt_threshold; i++)
    mpz_clear(coeff[i]);
  field_deinit();
}
```

E. Shamir's Secret Sharing Algorithm Code (To be added only in the LEA)

```
/*
 *  ssss version 0.5  -  Copyright 2005,2006 B. Poettering
 *  http://point-at-infinity.org/ssss/
 *  Modifications made by Abdullah Azfar in 2010
 *  This program is free software; you can redistribute it and/or
 *  modify it under the terms of the GNU General Public License
 *  http://www.gnu.org/licenses/gpl.html
 *  as published by the Free Software Foundation; either version 2
 *  of the License, or (at your option) any later version.
*/

void field_print(FILE* stream, const mpz_t x, int hexmode)
{
  int i;
  outfp=fopen("/home/azfar/Desktop/eadir/outfile.txt", "a+");
  if (hexmode) {
    for(i = degree / 4 - mpz_sizeinbase(x, 16); i; i--)
      fprintf(stream, "0");
    mpz_out_str(stream, 16, x);
    fprintf(stream, "\n");
  }
  else {
    char buf[MAXDEGREE / 8 + 1];
    size_t t;
    unsigned int i;
    int printable, warn = 0;
    memset(buf, degree / 8 + 1, 0);
    mpz_export(buf, &t, 1, 1, 0, 0, x);
    for(i = 0; i < t; i++) {
      printable = (buf[i] >= 32) && (buf[i] < 127);
      warn = warn || ! printable;
      fprintf(stream, "%c", printable ? buf[i] : '.');
      fprintf(outfp, "%c", printable ? buf[i] : '.');
    }
    fprintf(stream, "\n");
    if (warn)
      warning("binary data detected, use -x mode instead");
  }
fclose(outfp);
}

/* Prompt for shares, calculate the secret */

void combine(char tempbuf1[], char tempbuf2[], char tempbuf3[])
{
  opt_threshold = 3;
  opt_number = 5;
  mpz_t A[opt_threshold][opt_threshold], y[opt_threshold], x;
  char buf[MAXLINELEN];
  char *a, *b;
```

```
  int i, j;
  unsigned s = 0;

  mpz_init(x);

  for (i = 0; i < opt_threshold; i++) {
    if (! opt_quiet)
      printf("");

   if(i==0){
      strcpy(buf,tempbuf1);
         //printf("%s",tempbuf1);
      }
    else if(i==1){
      strcpy(buf,tempbuf2);
    }
    else if(i==2){
      strcpy(buf,tempbuf3);
    }

    buf[strcspn(buf, "\r\n")] = '\0';
    if (! (a = strchr(buf, '-')))
      fatal("invalid syntax");
    *a++ = 0;
    if ((b = strchr(a, '-')))
      *b++ = 0;
    else
      b = a, a = buf;

    if (! s) {
      s = 4 * strlen(b);
      if (! field_size_valid(s))
     fatal("share has illegal length");
      field_init(s);
    }
    else
      if (s != 4 * strlen(b))
     fatal("shares have different security levels");

    if (! (j = atoi(a)))
      fatal("invalid share");
    mpz_set_ui(x, j);
    mpz_init_set_ui(A[opt_threshold - 1][i], 1);
    for(j = opt_threshold - 2; j >= 0; j--) {
      mpz_init(A[j][i]);
      field_mult(A[j][i], A[j + 1][i], x);
    }
    mpz_init(y[i]);
    field_import(y[i], b, 1);
    field_mult(x, x, A[0][i]);
    field_add(y[i], y[i], x);
  }
  mpz_clear(x);
  if (restore_secret(opt_threshold, A, y))
    fatal("shares inconsistent. Perhaps a single share was used
twice");
```

```
  if (opt_diffusion) {
    if (degree >= 64)
      encode_mpz(y[opt_threshold - 1], DECODE);
    else
      warning("security level too small for the diffusion layer");
  }

  if (! opt_quiet)
    fprintf(stderr, "Resulting secret: ");
  field_print(stderr, y[opt_threshold - 1], opt_hex);

  for (i = 0; i < opt_threshold; i++) {
    for (j = 0; j < opt_threshold; j++)
      mpz_clear(A[i][j]);
    mpz_clear(y[i]);
  }
  field_deinit();
}

int main ()
{
  char *name;
  int i;;
  #if ! NOMLOCK
  if (mlockall(MCL_CURRENT | MCL_FUTURE) < 0)
    switch(errno) {
    case ENOMEM:
      warning("couldn't get memory lock (ENOMEM, try to adjust
RLIMIT_MEMLOCK!)");
      break;
    case EPERM:
      warning("couldn't get memory lock (EPERM, try UID 0!)");
      break;
    case ENOSYS:
      warning("couldn't get memory lock (ENOSYS, kernel doesn't
allow page locking)");
      break;
    default:
      warning("couldn't get memory lock");
      break;
    }
#endif

  if (getuid() != geteuid())
    seteuid(getuid());

  tcgetattr(0, &echo_orig);
  echo_off = echo_orig;
  echo_off.c_lflag &= ~ECHO;

  FILE *fp1;
  FILE *fp2;
  FILE *fp3;

  static int flag=1;
```

```
static int  key1, key2, key3;
static int keynum1, keynum2,keynum3;
key1=0;
key2=0;
key3=0;
keynum1=0;
keynum2=0;
keynum3=0;
char c1,c2,c3;

fp1=fopen("/home/azfar/Desktop/eadir/testFile1.txt", "r");
fp2=fopen("/home/azfar/Desktop/eadir/testFile2.txt", "r");
fp3=fopen("/home/azfar/Desktop/eadir/testFile3.txt", "r");

char comb1[300], comb2[300], comb3[300], others[400];
outfp=fopen("/home/azfar/Desktop/eadir/outfile.txt", "w");
fclose(outfp);
int w;

for (w=1; w<=2; w++){
    key1=0;
    key2=0;
    key3=0;

     strcpy(comb1,"");
     strcpy(comb2,"");
     strcpy(comb3,"");


    while(1){
      c1=fgetc(fp1);
      if(c1!=EOF){
        if (c1!='%'){
          comb1[key1]=c1;
          key1++;
        }
        else{
          comb1[key1]='\0';
          break;
        }
      }
    }
    printf("\n");
    while(1){
      c2=fgetc(fp2);
      if(c2!=EOF){
        if (c2!='%'){
          comb2[key2]=c2;
          key2++;
        }
        else{
          comb2[key2]='\0';
          break;
        }
      }
    }
```

```
      printf("\n");
      while(1){
        c3=fgetc(fp3);
        if(c3!=EOF){
          if (c3!='%'){
            comb3[key3]=c3;
            key3++;
           }
          else{
            comb3[key3]='\0';
            break;
           }
        }
      }

      printf("\n");
      printf("%s\n",comb1);
      printf("%s\n",comb2);
      printf("%s\n",comb3);

      combine(comb1,comb2,comb3);
   }
   outfp=fopen("/home/azfar/Desktop/eadir/outfile.txt", "a");
   while(1){
     c3=fgetc(fp3);
     if(c3!=EOF){
       others[key3]=c3;
       key3++;
      fprintf(outfp, "%c",c3);
     }
     else break;
   }
   fclose(outfp);
   return 0;
}
```

F. Configuration of the CPU Used by the User Agent

```
processor     : 0
vendor_id     : GenuineIntel
cpu family    : 15
model         : 4
model name    : Intel® Pentium® D CPU 2.80GHz
stepping      : 7
cpu MHz       : 2793.144
cache size    : 1024 KB
physical id   : 0
siblings      : 2
```

```
core id          : 0
cpu cores        : 2
fdiv_bug         : no
hlt_bug          : no
f00f_bug         : no
coma_bug         : no
fpu              : yes
fpu_exception : yes
cpuid level      : 5
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc pni monitor ds_cpl cid
cx16 xtpr lahf_lm
bogomips         : 5591.15
clflush size     : 64

processor        : 1
vendor_id        : GenuineIntel
cpu family       : 15
model            : 4
model name       : Intel® Pentium® D CPU 2.80GHz
stepping         : 7
cpu MHz          : 2793.144
cache size       : 1024 KB
physical id      : 0
siblings         : 2
core id          : 1
cpu cores        : 2
fdiv_bug         : no
hlt_bug          : no
f00f_bug         : no
coma_bug         : no
fpu              : yes
fpu_exception    : yes
cpuid level      : 5
wp               : yes
flags            : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36
clflush dts acpi mmx fxsr sse sse2 ss ht tm pbe nx lm constant_tsc pni monitor ds_cpl cid
cx16 xtpr lahf_lm
bogomips         : 5586.14
clflush size     : 64
```

G. Escrow Database Schema Definition

```
-- phpMyAdmin SQL Dump
-- version 3.2.2.1deb1
-- http://www.phpmyadmin.net
--
```

```
-- Host: localhost
-- Generation Time: May 09, 2010 at 12:12 PM
-- Server version: 5.1.37
-- PHP Version: 5.2.10-2ubuntu6.4

SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

--
-- Database: `escrowDatabase1`
--
CREATE DATABASE `escrowDatabase1` DEFAULT CHARACTER SET latin1
COLLATE latin1_swedish_ci;
USE `escrowDatabase1`;

-- --------------------------------------------------------

--
-- Table structure for table `authentication`
--

CREATE TABLE IF NOT EXISTS `authentication` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

-- --------------------------------------------------------

--
-- Table structure for table `sipmasterkey`
--

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key1` text NOT NULL,
  `key2` text NOT NULL,
  `rand` text NOT NULL,
  `signedhash` text NOT NULL,
  `csbID` text NOT NULL,
  `date` datetime DEFAULT NULL,
  `EAnames` text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=517 ;

-- --------------------------------------------------------

--
-- Table structure for table `t_lealogin`
--

CREATE TABLE IF NOT EXISTS `t_lealogin` (
  `l_Id` text NOT NULL,
  `l_pass` text NOT NULL
```

```
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Database: `escrowDatabase2`
--
CREATE DATABASE `escrowDatabase2` DEFAULT CHARACTER SET latin1
COLLATE latin1_swedish_ci;
USE `escrowDatabase2`;

-- --------------------------------------------------------

--
-- Table structure for table `authentication`
--

CREATE TABLE IF NOT EXISTS `authentication` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

-- --------------------------------------------------------

--
-- Table structure for table `sipmasterkey`
--

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key1` text NOT NULL,
  `key2` text NOT NULL,
  `rand` text NOT NULL,
  `signedhash` text NOT NULL,
  `csbID` text NOT NULL,
  `date` datetime DEFAULT NULL,
  `EAnames` text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

-- --------------------------------------------------------

--
-- Table structure for table `t_lealogin`
--

CREATE TABLE IF NOT EXISTS `t_lealogin` (
  `l_Id` text NOT NULL,
  `l_pass` text NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Database: `escrowDatabase3`
--
CREATE DATABASE `escrowDatabase3` DEFAULT CHARACTER SET latin1
COLLATE latin1_swedish_ci;
```

```
USE `escrowDatabase3`;

-- --------------------------------------------------------

--
-- Table structure for table `authentication`
--

CREATE TABLE IF NOT EXISTS `authentication` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

-- --------------------------------------------------------

--
-- Table structure for table `sipmasterkey`
--

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key1` text NOT NULL,
  `key2` text NOT NULL,
  `rand` text NOT NULL,
  `signedhash` text NOT NULL,
  `csbID` text NOT NULL,
  `date` datetime DEFAULT NULL,
  `EAnames` text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=516 ;

-- --------------------------------------------------------

--
-- Table structure for table `t_lealogin`
--

CREATE TABLE IF NOT EXISTS `t_lealogin` (
  `l_Id` text NOT NULL,
  `l_pass` text NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Database: `escrowDatabase4`
--
CREATE DATABASE `escrowDatabase4` DEFAULT CHARACTER SET latin1
COLLATE latin1_swedish_ci;
USE `escrowDatabase4`;

-- --------------------------------------------------------

--
-- Table structure for table `authentication`
```

```
--

CREATE TABLE IF NOT EXISTS `authentication` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

-- --------------------------------------------------------

--
-- Table structure for table `sipmasterkey`
--

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key1` text NOT NULL,
  `key2` text NOT NULL,
  `rand` text NOT NULL,
  `signedhash` text NOT NULL,
  `csbID` text NOT NULL,
  `date` datetime DEFAULT NULL,
  `EAnames` text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

-- --------------------------------------------------------

--
-- Table structure for table `t_lealogin`
--

CREATE TABLE IF NOT EXISTS `t_lealogin` (
  `l_Id` text NOT NULL,
  `l_pass` text NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
--
-- Database: `escrowDatabase5`
--
CREATE DATABASE `escrowDatabase5` DEFAULT CHARACTER SET latin1
COLLATE latin1_swedish_ci;
USE `escrowDatabase5`;

-- --------------------------------------------------------

--
-- Table structure for table `authentication`
--

CREATE TABLE IF NOT EXISTS `authentication` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `user_name` varchar(100) NOT NULL,
  `password` varchar(100) NOT NULL,
```

```
  PRIMARY KEY (`id`),
  UNIQUE KEY `user_name` (`user_name`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=4 ;

-- --------------------------------------------------------

--
-- Table structure for table `sipmasterkey`
--

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key1` text NOT NULL,
  `key2` text NOT NULL,
  `rand` text NOT NULL,
  `signedhash` text NOT NULL,
  `csbID` text NOT NULL,
  `date` datetime DEFAULT NULL,
  `EAnames` text NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1 AUTO_INCREMENT=1 ;

-- --------------------------------------------------------

--
-- Table structure for table `t_lealogin`
--

CREATE TABLE IF NOT EXISTS `t_lealogin` (
  `l_Id` text NOT NULL,
  `l_pass` text NOT NULL
) ENGINE=MyISAM DEFAULT CHARSET=latin1;
```

H. CPU Clock Resolution

We have used ptime microsec_clock::local_time() function of the Boost Posix time library to calculate the time duration. This function gets the local time using a sub second resolution clock. On Unix systems this is implemented using GetTimeOfDay with microsecond resolution. We calculated the actual resolution of GetTimeOfDay system call with the following piece of code.

```
#include <time.h>
#include <sys/time.h>
#include <stdio.h>
int main(int argc, char**argv)
{
  struct timeval tv1, tv2;

  gettimeofday(&tv1, NULL);
```

```
  do {
    gettimeofday(&tv2, NULL);
  }
  while (tv1.tv_usec == tv2.tv_usec);
  printf("Difference: %ld us\n", tv2.tv_usec - tv1.tv_usec + 1000000 *
(tv2.tv_sec - tv1.tv_sec));
  return 0;
}
```

After executing the code we got the following result.

```
ccsmoto:/home/azfar/Desktop/measurment/resolution # gcc -o gettime
gettime.c
ccsmoto:/home/azfar/Desktop/measurment/resolution # ./gettime
Difference: 1 us
```

I. SIP Express Router (SER) Configuration File

```
debug=3
fork=yes
log_stderror=yes

listen=130.237.209.238          # put your server IP address here
listen=192.168.2.238
port=5060
children=4

dns=no
rev_dns=no

loadmodule "/usr/local/lib/ser/modules/mysql.so"
loadmodule "/usr/local/lib/ser/modules/sl.so"
loadmodule "/usr/local/lib/ser/modules/tm.so"
loadmodule "/usr/local/lib/ser/modules/rr.so"
loadmodule "/usr/local/lib/ser/modules/maxfwd.so"
loadmodule "/usr/local/lib/ser/modules/usrloc.so"
loadmodule "/usr/local/lib/ser/modules/registrar.so"
loadmodule "/usr/local/lib/ser/modules/uri_db.so"
loadmodule "/usr/local/lib/ser/modules/auth.so"
loadmodule "/usr/local/lib/ser/modules/auth_db.so"

#Presence related modules
loadmodule "/usr/local/lib/ser/modules/dialog.so"
loadmodule "/usr/local/lib/ser/modules/pa.so"
loadmodule "/usr/local/lib/ser/modules/presence_b2b.so"
loadmodule "/usr/local/lib/ser/modules/xlog.so"
```

```
# ----------------- setting module-specific parameters -------------
--
modparam("auth_db|uri_db|usrloc", "db_url",
"mysql://ser:heslo@localhost/ser")
modparam("auth_db", "calculate_ha1", 1)
modparam("auth_db", "password_column", "password")
modparam("usrloc", "db_mode", 2)
modparam("rr", "enable_full_lr", 1)

#presence module related params
modparam("pa", "use_db", 1)
modparam("pa", "db_url", "mysql://ser:heslo@localhost/ser")
modparam("pa", "offline_winfo_timer", 3600)
modparam("pa", "offline_winfo_expiration", 259200)

modparam("pa", "auth", "none")
modparam("pa", "winfo_auth", "none")
modparam("pa", "use_callbacks", 0)
modparam("pa", "accept_internal_subscriptions", 0)
modparam("pa", "max_subscription_expiration", 3600)
modparam("pa", "timer_interval", 1)
modparam("presence_b2b", "on_error_retry_time", 60)
modparam("presence_b2b", "wait_for_term_notify", 33)
modparam("presence_b2b", "resubscribe_delta", 30)
modparam("presence_b2b", "min_resubscribe_time", 60)
modparam("presence_b2b", "default_expiration", 3600)
#modparam("presence_b2b", "handle_presence_subscriptions", 1)

#----Main routing logic--------
route {

    # -----------------------------------------------------------------
---
    # Sanity Check Section
    # -----------------------------------------------------------------
---
    if (!mf_process_maxfwd_header("10")) {
        sl_send_reply("483", "Too Many Hops");
        break;
    };

    if (msg:len > max_len) {
        sl_send_reply("513", "Message Overflow");
        break;
    };

    # -----------------------------------------------------------------
---
    # Record Route Section
    # -----------------------------------------------------------------
---
    if (method!="REGISTER") {
        record_route();
    };
```

```
    # ---------------------------------------------------------------
---
    # Loose Route Section
    # ---------------------------------------------------------------
---
    if (loose_route()) {
        route(1);
        break;
    };

    # ---------------------------------------------------------------
---
    # Call Type Processing Section
    # ---------------------------------------------------------------
---
    if (uri!=myself) {
        route(1);
        break;
    };

    if (method=="ACK") {
        route(1);
        break;
    } else if (method=="INVITE") {
        route(3);
        break;
    } else     if (method=="REGISTER") {
        route(2);
        break;
    } else if(method =="SUBSCRIBE") {
        route(4);
        break;
    } else if(method =="PUBLISH"){
        route(5);
        break;
    };


    /*lookup("aliases2");*/
    if (uri!=myself) {
        route(1);
        break;
    };

    if (!lookup("location")) {
        sl_send_reply("404", "User Not Found");
        break;
    };

    route(1);
}

route[1] {

    # ---------------------------------------------------------------
---
```

```
    # Default Message Handler
    # -------------------------------------------------------------
---
    if (!t_relay()) {
        sl_reply_error();
    };
}

route[2] {

    # -------------------------------------------------------------
---
    # REGISTER Message Handler
    # -------------------------------------------------------------
--
    sl_send_reply("100", "Trying");

    /*if (!www_authorize("","subscriber")) {
        www_challenge("","0");
        break;
    };

    if (!check_to()) {
        sl_send_reply("401", "Unauthorized");
        break;
    };*/

    /*consume_credentials();*/

    if (!save("location")) {
        sl_reply_error();
    };
}

route[3] {
    # -------------------------------------------------------------
---
    # INVITE Message Handler
    # -------------------------------------------------------------
---
    /*if (!proxy_authorize("","subscriber")) {
        proxy_challenge("","0");
        break;
    } else if (!check_from()) {
        sl_send_reply("403", "Use From=ID");
        break;
    };*/

     /*consume_credentials();

    lookup("aliases2");*/
    if (uri!=myself) {

        route(1);
        break;
    };
```

```
    if (!lookup("location")) {
        sl_send_reply("404", "User Not Found");
        break;
    };

    route(1);
}

route[4] {
    # ------------------------------------------------------------------
---
    # SUBSCRIBE Message Handler
    # ------------------------------------------------------------------
---
    if (!t_newtran()) {
         sl_reply_error();
            break;
    };

    xlog("L_ERR", "PA: handling subscription: %tu from: %fu\n");
    handle_subscription("registrar");
    break;
}

route[5] {
    # ------------------------------------------------------------------
---
    # PUBLISH Message Handler
    # ------------------------------------------------------------------
---
    if (!t_newtran()) {
         sl_reply_error();
            break;
    };

    xlog("L_ERR", "PA: handling publish: %tu from: %fu\n");
    handle_publish("registrar");
    break;
}
```