A Session Initiation Protocol User Agent with Key Escrow

Providing authenticity for recordings of secure sessions

MD. SAKHAWAT HOSSEN

KTH Information and Communication Technology

Master of Science Thesis
Stockholm, Sweden 2010

TRITA-ICT-EX-2010:1

A Session Initiation Protocol User Agent with Key Escrow

Providing authenticity for recordings of secure sessions

Md. Sakhawat Hossen

hossen@kth.se

Masters Thesis

2010.01.18

This thesis is submitted in partial fulfilment of the requirements for a Masters of Science degree in Internetworking.

School of Information and Communication Technology
Royal Institute of Technology (KTH)
Stockhom,Sweden

Supervisor and Examiner: Professor Gerald Q. Maguire Jr.

Abstract

Voice over Internet Protocol (VoIP), also called IP telephony is rapidly becoming a familiar term and as a technology it is invading the enterprise, private usage, and educational and government organizations. Exploiting advanced voice coding & compression techniques and bandwidth sharing over packet switched networks, VoIP can dramatically improve bandwidth efficiency. Moreover enhanced security features, mobility support, and cost reduction features of VoIP are making it a popular choice for personal communication. Due to its rapid growth in popularity VoIP is rapidly becoming the next generation phone system.

Lawful interception is a mean of monitoring private communication of users that are suspected of criminal activities or to be a threat to national security. However, government regulatory bodies and law enforcement agencies are becoming conscious of the difficulty of lawful interception of public communication due to the mobility support and advanced security features implemented in some implementations of VoIP technology. There has been continuous pressure from the government upon the operators and vendors to find a solution that would make lawful interception feasible and successful. Key escrow was proposed as a solution by the U. S. National Security Agency. In key escrow the key(s) for a session are entrusted to a trusted third party and upon proper authorization law enforcement agencies can receive the session key(s) from this trusted third party However, key escrow adds some security vulnerabilities and potential risks as an unethical employee of the key escrow agent (or a law enforcement agency that has received the session key(s)) can misuse the key(s) to forge content of a communication session -- as he or she possesses the same key(s) as the user used for this session. This thesis addresses the issue of forged session content, by proposing, implementing, and evaluating a cryptographic model which allows key escrow *without* the possibility of **undetectable** fabrication of session content. The implementation utilizes an existing implementation of a Session Initiation Protocol (SIP) user agent 'minisip' developed at KTH. The performance evaluation results suggest that the proposed model can support key escrow while protecting the user communication from being forged with the cost of minimal computational resource and negligible overhead.

Key words: Lawful Interception, Key escrow, SRTP, MIKEY, PKI, Digital Signature

Sammanfattning

Röst över Internet Protokoll (VoIP), även kallad IP-telefoni är snabbt bli en välkänd term och som teknik är det invadera företaget, privat bruk, och utbildning och statliga organisationer. Utnyttja avancerad talkodning & tekniker kompression och bandbredd utbyte över paket-nät kan VoIP dramatiskt förbättra bandbredd effektivitet. Dessutom förbättrade säkerhetsfunktioner, stöd till rörlighet och kostnader minskning funktioner VoIP gör det till ett populärt val för personlig kommunikation. Grund av sin snabba tillväxt i popularitet VoIP är snabbt på att bli nästa generation telefonsystemet.

Avlyssning är ett medelvärde av övervakning privat kommunikation för användare som är misstänkta för brottslig verksamhet eller att vara ett hot mot den nationella säkerhet. Regeringens tillsynsorgan och brottsbekämpande myndigheter blir medvetna om svårigheten för avlyssning av allmänheten meddelande på grund av stöd till rörlighet och avancerade säkerhetsfunktioner genomförts i vissa implementationer av VoIP-teknologi. Det har ständig press från regeringen på operatörer och leverantörer för att hitta en lösning som skulle göra avlyssning möjlig och framgångsrik. Nyckeldeposition föreslogs som en lösning av US National Security Agency. In nyckeldeposition nyckeln (er) för en session anförtros en betrodd tredje part och ändamålsenliga tillstånd brottsbekämpande myndigheterna kan få sessionen nyckel (s) från denna betrodd tredje part dock tillagt nyckeldeposition något trygghet sårbarheter och risker som en oetisk anställd av nyckeldeposition agent (eller en brottsbekämpande myndighet som har fått sessionsnyckeln (s)) kan missbruk kilen (s) att skapa innehållet i ett meddelande session - som han eller hon besitter samma nyckel (n) som användaren använde för denna session. Denna avhandling behandlar frågan om förfalskade session innehåll, genom att föreslå, genomföra och utvärdera ett kryptografiskt modell som tillåter nyckeldeposition utan möjligheten omätbara tillverkning av sessionen innehåll. Genomförandet använder en befintlig genomföra en Session Initiation Protocol (SIP) användaragent "minisip" utvecklats på KTH. Utvärderingen av prestanda resultat tyder på att den föreslagna modellen kan stödja nyckeldeposition och samtidigt skydda användar meddelande inte smidda med kostnaden för minimal computational resurs och försumbar omkostnader.

Table of Contents

List of Figures

List of Tables

List of Listings

Acknowledgements

It is an auspicious occasion for me as a student to express my deep feelings of gratitude to the department; especially to my supervisor, teachers, and also to the departmental staffs.

I am immensely indebted to my supervisor and examiner, **Professor Gerald Q. Maguire Jr.** for his wonderful guidance, inspiration, and continuous encouragement. I would like to extend my gratitude for his sincere review and correction of this thesis project as it could not have been realized without his astute supervision. I consider myself fortunate to have such a wonderful person as my supervisor and the time that I spent with him during the project will remain as an enjoyable experience for a long time.

I give profound thanks to **Erik Eliasson** for his very valuable direction and special attention. I also acknowledge my friends who, through their interest and work, are my constant source of inspiration.

Finally, I would like to acknowledge my parents in Bangladesh who always believed in me. Their unconditional support and continuous inspiration kept me alive in this frozen land. My only brother and sister have truly been a source of inspiration. I hope that I have fulfilled their aspirations and I dedicate this thesis to my family members.

Abbreviation and Acronyms

AES	Advanced Encryption Standard
API	Application Programming Interface
B2BUA	Back-to-back User Agent
CA	Certificate Authority
CALEA	(U.S.) Communications Assistance for Law Enforcement Act
CODEC	Encoder/decoder
CSR	Certificate Signing Request
DES	Data Encryption Standard
DNS	Domain Name System
FCC	(U.S.) Federal Communications Commission
FQDN	Fully Qualified Domain Name
FRA	(Sweden’s) Försvarets radioanstalt
FTPS	File Transfer Protocol Secure
HDR	Header Payload
HMAC	Keyed Hash Message Authentication Code
HTTP	Hypertext Transfer protocol
HTTPS	Hypertext Transfer Protocol Secure
IETF	Internet Engineering Task Force
ISP	Internet service provider
IV	Initialisation Vector
LDAP	Lightweight Directory Access Protocol
LEAF	Law Enforcement Access Field
LEA	Law Enforcement Agency
LI	Lawful Intercept
MAC	Message Authentication Code
MD	Message Digest
MD5	Message Digest algorithm 5
MKI	Master Key Identifier
MIKEY	Multimedia Internet Keying
MIME	Multipurpose Internet Mail Extension
MKI	Master Key Identifier
NSA	(U.S.) National Security Agency
PKE	Public key Encryption
PKI	Public Key Infrastructure
PSK	Pre Shared Key
PRF	Pseudo-random function
PSTN	Public Switch Telephony Network
QoS	Quality of Service
RA	Registration Authority
RFC	Request for Comments
ROC	Rollover Counter
RSA	Rivest Shamir Adleman
RTCP	Real Time Transport Control Protocol
RTP	Real Time Transport Protocol
SBC	Session Border Controller
SCP	Secure Copy
SDP	Session Description Protocol
SER	SIP Express Router

SHA	Secure Hash Algorithm
SIP	Session Initiation Protocol
S/MIME	Secure/Multipurpose Internet Mail Extensions
SRTP	Secured Real Time Transport Protocol
SRTCP	Secured Real Time Transport Control Protocol
SSL	Secure Socket Layer
SSRC	Synchronization source
TCP	Transmission Control Protocol
TEK	Traffic Encryption Key
TFTP	Trivial File Transfer Protocol
TGK	TEK Generation Key
TLS	Transport Layer Security
TTP	Trusted Third Party
VoIP	Voice over Internet Protocol
UDP	User Datagram Protocol
UID	User Identification
UML	Unified Modelling Language
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
US	United States
VIPSec	Voice Interactive Personalized Security
VoIP	Voice over Internet Protocol

Chapter 1: Introduction

1.1 Motivation

Voice over Internet protocol (VoIP), also known as IP telephony is a familiar term and killer application in the area of personal communication. As a technology it is invading enterprise, educational and government organizations. This technology is gaining popularity day by day due to its many attractive features. From a technical point of view VoIP can dramatically improve the bandwidth efficiency by exploiting advanced voice coding and compression techniques and can share bandwidth with data on packet switched networks. As the packets are processed at the end-points it can incorporate advanced security features. Additionally, VoIP supports user, session, and device mobility. Moreover users like this technology because it can reduce their voice (and conferencing) costs. Due to the rapid growth in popularity VoIP is in a hurry to be the next generation phone system.

Lawful interception (LI) is a mean of monitoring private communication of users that are suspected of criminal activities or to be a threat to national security. Lawful Intercept (LI) is not a new requirement in the area of public telephony. LI was conceived 50 to 60 years ago. Users have not been positive to LI as it raises a number of controversial issues such as violation of human rights and decreased confidentiality of commercial communication. However in recent years, government regulatory bodies and law enforcement agencies (LEAs) are becoming conscious of the difficulty of lawful interception of public communication due to the mobility support and advanced security features implemented in some implementations of VoIP technology [1] [2]. There has been continuous pressure from the government upon the operators to find a solution that would make lawful interception feasible and successful. Key escrow was proposed as a solution by the U. S. National Security Agency. In key escrow the key(s) for a session are entrusted to a trusted third party and upon proper authorization law enforcement agencies can receive the session key(s) from this trusted third party However, key escrow adds some security vulnerabilities and potential risks as an unethical employee of the key escrow agent (or a law enforcement agency that has received the session key(s)) can misuse the key(s) to forge content of a communication session -- as he or she possesses the same key(s) as were used for this session.

Currently, LI in both the fixed and mobile networks is relatively easy due to the network architecture; specifically the intelligent core with dumb end terminals. As a result of this architecture it has been possible to require that the telecommunication switch vendors build in mechanisms for LI. Increasingly, LI is not always successful due countermeasure taken by users to prevent or reduce the ease of monitoring private communications. Moreover these countermeasures can result in misleading information.

Due to the Internet’s architecture of smart end devices and dumb core network it has become more technically difficult to lawfully intercept private communications. One of the major reasons for this is that smart end devices can implement sophisticated encryption techniques that make it very difficult to retrieve the actual communication contents. To facilitate LI ‘key escrow’ was first proposed during the early 1990s. The main idea underlying key escrow is that the keys needed to decrypt an encrypted communication session will be deposited with a trusted third party (TTP) as an escrow agent. The LEA can get the session key from the TTP after showing proper authorization. This method enables the LEA to perform LI of VoIP users' encrypted communication. However, key escrow raises some security vulnerabilities and potential risks. Moreover a large-scale key escrow system has not yet been implemented due to its high cost and complexity. Details of key escrow systems are described in section 2.5.

The main concern regarding key escrow systems is the trustworthiness of the TTP. Since the session keys are stored at the escrow agent an unethical employee of the TTP could misuse this information. Such an employee could both divulge the contents of a session or could forge contents of a communication session (for example, in order to blackmail the user by fabricating evidence of criminal activity that could be presented in court). This (dual) weakness of key escrow has caused many people (such as cryptographers, human right workers, and individuals) to reject key escrow as a viable solution for facilitating lawful interception. Therefore, some mechanism is required that could make key escrow feasible while preventing tampering with the communication session's content. At the same time there is a need to make key escrow desirable, i.e., there needs to be a reason for the users to want to use key escrow. However, this later issue is outside the scope of this thesis. Making key escrow feasible, while restoring a balance between users and LI, is the main motivation that leads us to propose, implement, and evaluate a model that allows key escrow *without* the possibility of **undetectable** fabrication of session content.

The implementation of the proposed solution utilizes an existing implementation of a Session Initiation Protocol (SIP) [3] user agent 'minisip' developed at KTH. The existence of a working implementation could have very high impact on businesses that for regulatory and other legal reasons need to be able to store and retrieve encrypted sessions. Such an implementation might also be valuable to other users.

1.2 Thesis overview

In this thesis a very simple key escrow agent is implemented– with whom the session keys are deposited. Note that the session key is escrowed *after* a session is over. During a session we sign blocks of hashes over the session contents and transmit these signed hash values over the Real Time Transport Control Protocol (RTCP) channel parallel to the Real Time Transport Protocol (RTP) traffic channel that is being used. 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 to forge session contents.

This thesis work has extended the existing minisip implementation to support key escrow. Minisip is an open source SIP user agent developed in KTH (see section 2.9 on page 25). Minisip was chosen because of its extensive support for security. Minisip already implements several security protocols to protect the media and signalling information of a call. Minisip implements Secure Real Time Transport Protocol (SRTP) to protect the media data (offering privacy by using encryption and integrity protection using signed hashes), Transport Layer Security (TLS) to secure signalling, and Multimedia Internet Keying (MIKEY) as a key management protocol. (SRTP and MIKEY are described in Chapter 2:). MIKEY provides the mechanism for the parties to agree upon a session master key; from which SRTP generates separate session keys for encryption and integrity protection for each media stream. SRTP uses the session keys to protect the Real Time Transport Protocol (RTP) packets. Minisip has been extended to deposit the session master key provided by MIKEY with the escrow agent after a session is terminated (see section 2.4 on page 10).

Figure 1-1 gives an overview of how the overall system works. As noted earlier, since the escrow agent has the same session key as the sender and receiver there is a potential for interception & decryption and/or forgery of the content of the media streams of the Secure Real Time Control Protocol (SRTCP) packets. To prevent real-time interception and decryption of a media stream or its associated control stream we only deposit the session key *at the end of the session*. The authors assume that the LEA conducting an authorized interception of the communication between the parties has some means of intercepting the packets that are part of the communication session (including all of the SIP, SRTP, and SRTCP packets). The technical means that the LEA uses to do this is outside of the scope of this thesis (See the thesis of Muhammad Sarwar Jahan Morshed [4].)

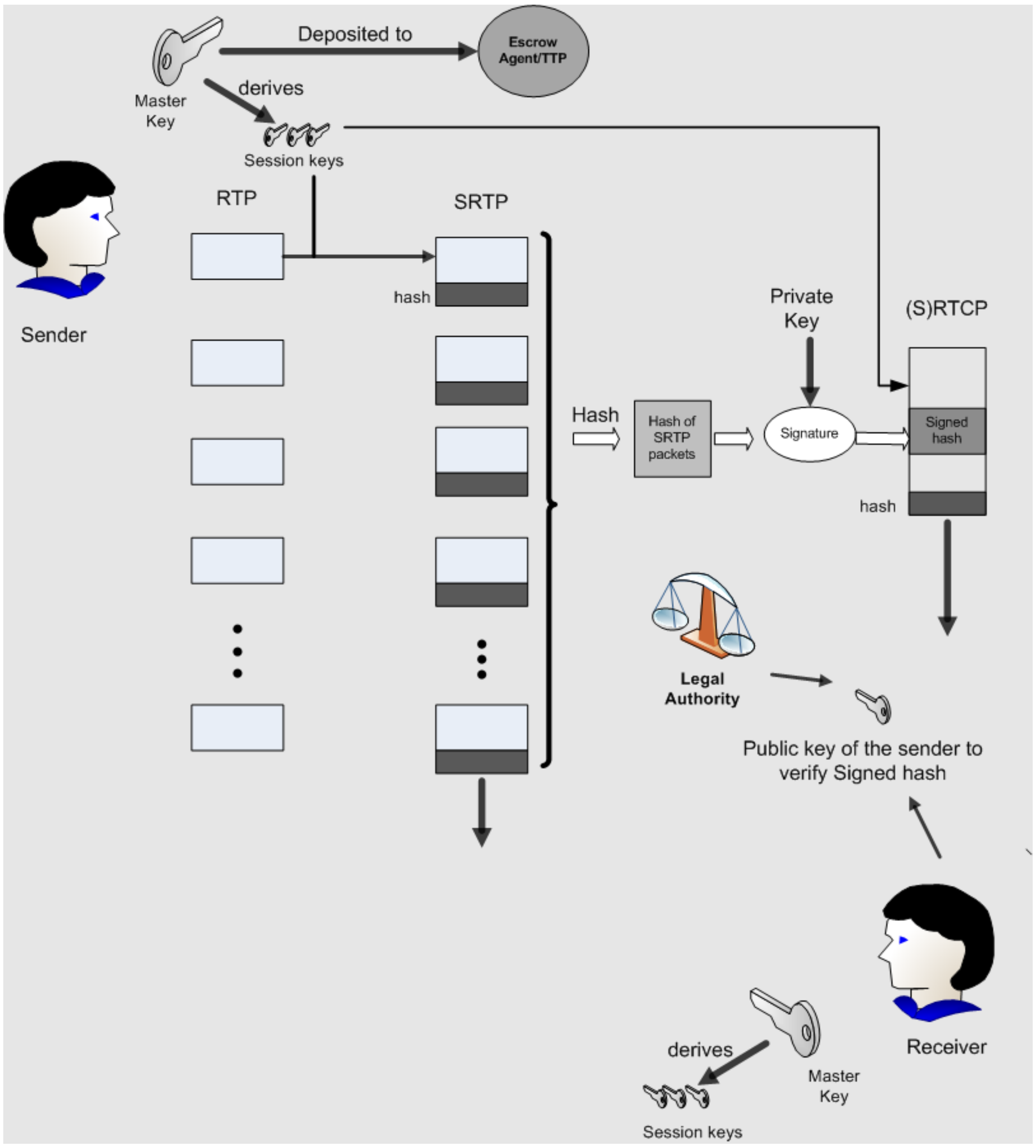


Figure 1-1: Overview of the operation of the proposed system

To prevent forgery of (or tampering with) a recorded media stream we compute a signed hash over multiple SRTP packets. It is important to note that rather than signing with a key associated with this *specific* session, we instead sign the hash using the **private key** of the sender. The resulting signed hash is sent as part of a payload in a Secure Real Time Control Protocol (SRTCP) packet. As there is no reason to deposit the private key of the sender with escrow agent it will be impossible for anyone to forge the digital signature of the hash over the SRTP packets. If someone who has obtained access to the session key(s) (for example, a LEA who has presented a lawful intercept order to the escrow agent) attempts to fabricate the contents of a (captured) media stream by generating SRTP packets and encrypting them with the correct session key, it will be possible using the sender's public key to refute the authenticity of these packets – since while they may be encrypted by the correct session encryption key, anyone can use the **public key** of the sender to verifying if media stream has the correct digital signature. This suggests that for convenience the sender may also want to deposit the final signed hash value with the TTP. The details of why this final signed hash value should be escrowed are presented in section 4.2. The final signed hash value could be escrowed at the same time as the sender deposits the session keys(s) that have been used for a session.

SRTP and SRTCP both make use of symmetric encryption in order to support low delay and high throughput for the media streams. However, there is no need for the signed hash values to be delivered with low delay – since they are only (potentially) relevant *after* the session has ended. It is the combination of signing the hash of a group of SRTP packets at the same time and the lack of any requirement for low delay that enables asymmetric public key techniques to be used for signing these hashes.

1.3 Research questions

Based on the thesis overview presented in section 1.2 there are some open research questions that need to be addressed. The questions are as follows:

Q1: How many SRTP packets should be grouped together?

Q2: What is a suitable rate for computing the signed hashes?

Q3: Should the number of packets that are group together be computed adaptively based upon the rate at which the sender can compute and sign the hashes?

Q4: Is there any minimum number of SRTP packets that should be group together?

Q5: Is there any maximum number of SRTP packets that should be group together?

Q6: Is there any problem of too frequent signing, leading to a leaking of bits of the sender's private key?

Q7: Are there any weaknesses in this system design?

Q8: Are there any weaknesses in the implementation of this system?

Some of these questions are addressed in this thesis; while some will be addressed in the companion thesis of Muhammad Sarwar Jahan Morshed [4] and other theses.

Chapter 2: Background

This chapter provides some background for the readers. It introduces some of the key concepts and protocols that are used in the thesis. We start by presenting the basic concepts of Lawful Intercept, a trusted third party, key escrow, a public key infrastructure, and a signed hash. In Section 2.6 and later, we present three important protocols for this work: SRTP, SRTCP, and MIKEY. In the final section, we briefly present an open source Session Initiation Protocol (SIP) user agent named as minisip and our motivation for selecting it as the basis for our implementation.

2.1 Lawful Intercept (LI)

Lawful Intercept (LI) is the legal monitoring of private communication. LI provides the means and mechanisms for the government and law enforcement agencies (LEAs) to conduct electronic surveillance of either circuit or packet switched communication. In most countries LI is only possible under a valid administrative or judicial order. The criterion for issuance of such a LI order is generally collecting evidence to be used in criminal proceedings or to prevent harm to the society (for instance in conjunction with national security).

Although the concept of LI was conceived more than 50 years ago when the government used technical means to tap and/or trace public telecommunication, there have been many questions raised regarding the practice of LI. Initially interception was not primarily concerned with collecting evidence for criminal prosecution; in most cases it was used for ensuring national security. Because the use of LI was typically done in secret there was little discussion of individual privacy. However, instances of politically motivated LI lead to a wider discussion of LI and the right of individuals to private communication and association. As a result, illegal monitoring is often framed in terms of being a violation of human rights. This has lead to the creation of new laws to define a proper framework for LI. (For further discussions of the framework for LI see [1]. Another potentially relevant publication is [5] where the author discusses the retention of communication data as a security measure that conflicts with the right to privacy. In her discussion she argues that *perceived privacy* is a prerequisite for making independent decisions and freely communicating with other persons while living in a participatory society. She has examined communication monitoring as a law enforcement tool with respect to interception of content, data retention, and data preservation.)

Two important requirements for successful LI are: (1) the user must **not** be aware that he or she is the subject of LI (i.e., that their communication is being intercepted) and (2) other users of the communication system must **not** be affected by the LI. The exact details of how LI is performed vary from system to system and depend on the architecture of the telecommunication system, laws, and regulatory policy. However, today in many countries all public communication service providers (operators) are generally required to provide the government and LEAs with assistance in conducting LI[6].

The technical means and requirements for LI change due to the evolution of the various communication systems. This evolution in telecommunication architecture has meant that the technical means for LI as well as the laws and policies for LI have had to adapt to the emergence of new technology. For example, in Sweden a major change in LI law occurred because of the fact that most international telecommunications is now carried via optical fibers and not via radio signals. Unfortunately, the earlier law did not provide a framework for LI of traffic carried via such fibers; but did clearly describe how and who was responsible for LI for radio communication. The new law is popularly referred to as the FRA-lagen (The FRA law) – after the initials of the Försvarets radioanstalt (FRA), the National Defence Radio Establishment – as this agency has been given the assignment of LI for international traffic under the new law. (For details see [7][8].) Similar changes in LI laws and regulations have been made in a number of countries; see for example the U.S. *Communications Assistance for Law Enforcement Act* (*CALEA*) regulations [9].

Until recently LI in fixed networks (primarily the Public Switch Telephony Network (PSTN)) and mobile networks (such as Public Land Mobile Networks) has been relatively easy to conduct due to the centralized nature of these telecommunication network architectures and the limited number of operators (until recently often only a single government owned and/or controlled operator). However, the Internet lacks centralized network architecture and there are a very large numbers of operators. Additionally, the Internet is based on packet switching; in such a network individual packets are routed – potentially over many different networks and routes between a source and destination(s). As a result LI is more challenging than for the fixed and mobile telephony architectures.

Today, Voice over Internet Protocol (VoIP) is a killer application that is both competing with and transforming the global telephony system. This revolutionary technology supports user mobility and enables a user to have multiple identities. When combined with the problems of LI in the Internet, LI for VoIP traffic is very problematic.

To further complicate the problem of LI for VoIP the modern Internet is characterized by having smart edge nodes with a dumb core (in contrast to fixed and mobile telephony networks). The presence of computationally capable nodes at the edge of the network makes it very easy to implement countermeasures against LI. Moreover, Internet users can add their own services at any time from any point in Internet *without* depending on their access operator, making LI even more challenging as there is no perfect location in the Internet to perform LI [10].

Despite the many technical difficulties of performing LI for VoIP traffic there are many interested parties that want to be able to perform LI for VoIP traffic. Thus this thesis will assume that there is a desire for LI and that legal and technical requirements have been (or will be) introduced to make the capture and storage of VoIP packets feasible (at least when applied to a small number of targeted intercept subjects).

2.2 Public Key Infrastructure (PKI)

A public key infrastructure (PKI) is a collection of components including hardware, software, people, policies, and procedure to securely distribute public keys in the form of digital certificates to achieve communication security. A PKI supports public-key cryptography.

Public key cryptography is based upon every entity that desires to communicate privately having a pair of keys: a public and a private key. This approach depends upon the assumption that data encrypted with a public key can **only** be decrypted by using the corresponding private key. The public key is publicly available – it could be printed in the newspaper, posted on a web site, printed on a user's business cards, painted on the side of a car, etc. While the private key is only known by the entity to which the pair of keys belongs.

To be sure that a certain key pair really belongs to only one person it is necessary to use a specific "document" which binds a public key to one person. Such a document or credential that contains a public key or information about the public key of a user is called a "digital certificate".

Ideally a PKI consists of a certificate authority (CA) that issues and verifies certificates, a registration authority (RA) that acts as the verifier for the certificate authority before a certificate is issued to a requestor, a repository to store and retrieve certificates, a method of revoking certificates, and a method of evaluating a chain of certificates starting with public keys that are known and trusted in advanced to reach the target.

In the following subsection the details of these digital certificates is presented; along with a brief description of how such a certificate is created. These descriptions are sufficient for the reader to understand the basic ideas utilized in the rest of the thesis, but the interested reader is referred to other sources for further details (such as [11]).

2.2.1 Why is a PKI necessary?

Internet is increasingly seen as a daily necessity in today’s personal and business worlds due to its ubiquitous nature and because of e-commerce, e-health, e-government, … representing opportunities for increased efficiency, increased flexibility, … . However, security and personal integrity are important issues that must be considered.

In the corporate world various stakeholders are expected to maintain trusted business relationships. This trusted business relationship generally requires mutual authentication of the parties, confidentiality, integrity, and non-repudiation in order to perform secure business transactions. Non-repudiation is generally required so that no party can deny that a specific transaction has occurred. Similar requirements occur in other settings, such as when a health care worker accesses and updates a patient’s medical records, electronic voting (where the voter must be determined to be a valid voter, but their actual vote can not be identified with the voter), … .

A traditional face-to-face transaction in a small community generally required only minimal interaction and normally did not necessitate the use of digital security and integrity mechanisms (for example, relying on mutual knowledge of the parties or via a human chain of trust, the ability of the community to enforce legally binding agreements, etc.). However, today face-to-face transactions are not always possible or even practical due to the physical distance between the parties. Additionally, these face to face transactions are in some cases not even desirable – for example, it may be easier to have an open electronic market for stocks, commodities, etc. where all of the transaction is captured in digital form (for example, for enforcing regulations).

To establish a trusted business relationship the two parties can use some credential (secret key or digital certificate) to securely authenticate each other. These two parties can exchange such credentials via a face-to-face meeting to exchange credentials, use postal mail or email to exchange their certificates, or can download their public key from anywhere in the Internet to a location where their stored certificate will be available to the other party (who can download it to where ever they are attached to the Internet).

Because the exchange of credentials is so important, this is often the focus of an attacker. For example, the attacker could pose as a mail transfer agent to intercept the email between the users – as a form of man-in-the-middle attack. Similarly the attacker might use DNS poisoning to induce the two parties to deposit their public key and retrieve the public key of the other party from the attacker's site. Thus enabling the attacker to replace each party's public key with their own key, thus establishing the attacker as a man-in-the-middle. In this case each of the parties will believe that they are securely communicating with each other, when in fact they are securely communicating with the attacker! As many believe that face-to-face exchange of credentials is not sufficiently scalable, there is a desire for an infrastructure to securely distribute public keys. Hence the idea of a PKI came into existence. Every PKI provides the following functionalities [12]:

Public key cryptography	the generation, distribution, administration, and control of cryptographic keys
Certificate issuance	binds a public-key to an individual, organization, or other entity, or to some other data—for example, an email or purchase order
Certificate validation	the process that verifies that a trust relationship or binding exists and that a certificate is still valid for a specific operation
Certificate revocation	the process that cancels a previously issued certificate and either publishes the cancellation to a Certificate Revocation List or enables an Online Certificate Status Protocol process

2.2.2 How does PKI work?

This subsection briefly describes the workflow of a PKI (see Figure 2-1). Initially a subject (a user) applies for a certificate to a RA. Next the RA performs verification of the subject's identity. After verification of this identity, the RA sends a certificate request to the CA on behalf of the subject. The CA checks the validity of the RA and checks the information in the forwarded certificate requests if these checks are passed, then the CA issues a certificate and stores a copy of the issued certificate in its local storage. Later the CA publishes the certificate in a certificate repository. The RA provides the user with the certificate issued by the CA. Given this certificate the subject can now digitally sign any message with the private key associated with this certificate. Upon receiving a digitally signed message the receiver first retrieves the certificate from the certificate repository, then verifies the message using the public key in the certificate. In some cases, the sender may include their public certificate in the message.

Note that the details of the creation of the certificate and the validation of a certificate lie outside the scope of this thesis (for further details see [13]). We simply assume that the various parties have valid certificates and that minisip contains the necessary code for using these certificates (the specifics of this will be described in Section 2.9).

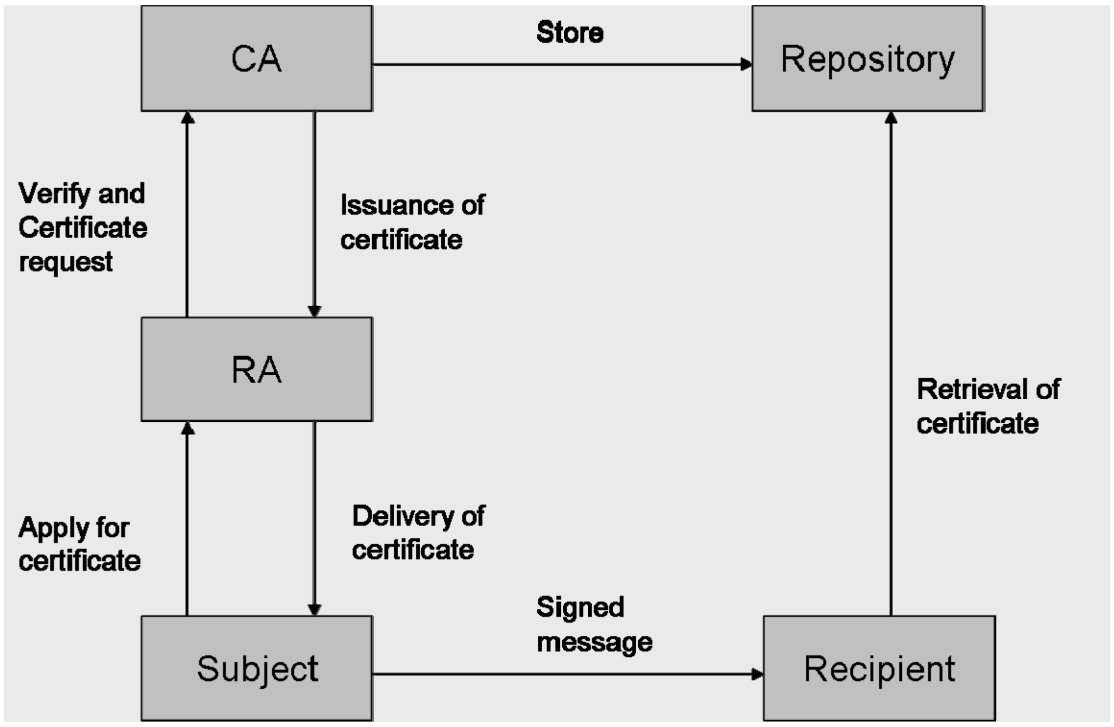


Figure 2-1: PKI workflow (adapted from [14])

2.3 Keyed-Hash Message Authentication Code

Keyed-Hash Message Authentication Code (HMAC) is one type of message authentication code (MAC) calculated using a specific cryptographic function combined with a secret key. HMAC can be used for integrity protection and authentication of a message. A message authentication code can be calculated using secret key cryptography or using a hash function; whereas HMAC can be calculated using any iterative hash function, such as Message Digest 5 (MD5) or the Secure Hash Algorithm (SHA). When an HMAC is calculated using MD5 the resulting message authentication code algorithm is referred to as HMAC-MD5 and similarly when SHA is used to calculate the HMAC the algorithm is referred to as HMAC -SHA. The security of HMAC depends on the underlying hash algorithm. All such hash algorithms (or message digest functions) should possess two properties:

- Collision resistance (i.e., it should be infeasible to find two message that produce same output); and
- Irreversible (i.e., given an output message authentication code, it will not be possible to produce the message).

All the hash algorithms work in a similar manner. The message is first padded to a multiple of some length (in practice this is generally 512 bytes) with a pad that indicates the length of the message. The shared secret key (K_{shared}) is concatenated with the message and a hash is calculated. The resulting message authentication code is Hash (K_{shared} | m), where K_{shared} is the shared secret and m is the message. However, this technique has a serious security flaw, as there is a chance of a message extension attack. In this scenario an attacker could compute a message authentication code of a longer message beginning with m, if he knows m and the correct message authentication code of m.

HMAC overcomes this shortcoming by concatenating K_{shared} to the front of the message and digesting, then prepends the key to the output and digests again. This nested digest with secret key inputs to both iterations prevents the extension attack that could be performed if we simply hash the message concatenated with the key once. Figure 2-2 shows the HMAC procedure where the HMAC function takes a variable length key and variable size message and produces a fixed size output. The output length is the same length as used by the underlying hash algorithm (128 bits for MD5 and 160 bits for SHA). As noted earlier, the digest/hash operation first pads the key to a 512-bit block length - if the key is larger than 512 bits, then HMAC first computes a digest of the key then pads again to produce a 512-bit block. The padded key is XORed with the constant $const_1$ (= 36_{16}), then this result is appended to the message and the first digest/hash is performed. The padded key is XORed with another constant $const_2$ (= $5C_{16}$) and appended to the output of the previous digest. Now a final digest is performed to produce the HMAC of the message [13].

HMAC has lower performance than the normal procedure to produce a message authentication code as it does a second digest. However, this second digest is computed over the secret and a digest, hence it does not add much cost if the original message was large (as the computational cost of this second hash is *independent* of the length of the message). For a large message HMAC's performance is negligibly worse than a single message authentication code, but its use prevents the message extension attack. As will be described later, both SRTP and MIKEY use the HMAC -SHA1 algorithm to compute a message authentication code for authentication and integrity protection.

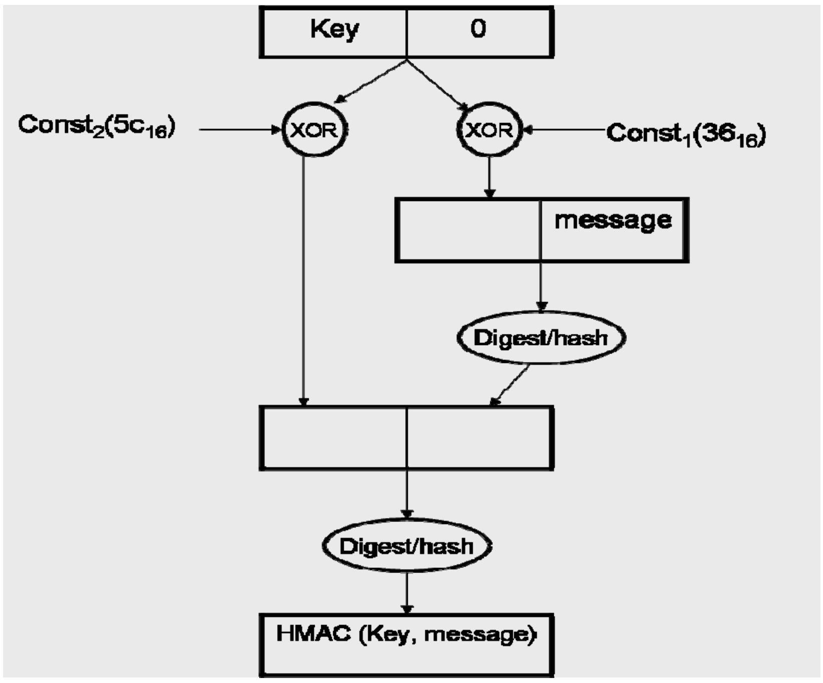


Figure 2-2: HMAC (Adapted from [13] page 143)

2.4 Trusted Third Party (TTP) or Escrow agent

A Trusted Third Party is a complementary solution to the need for a trusted service in the field of electronic communication; especially in e-commerce. The International Standards Organization defines a TTP as:

A Trusted Third Party is a security authority or its agent which is trusted by other entities for the security functions it provides. When a Trusted Third Party is the security authority for a domain, it can be trusted within that domain.[15]

A TTP must meet some functional requirements and these requirements may vary according to the scale of the TTP. The law enforcement members of Germany, England, France, and The Netherlands (known as The Security Group of G4) along with Sweden have defined fourteen functional requirements for an international TTP architecture (see section 2.3 of [15]). A TTP must be used to realize a point of trust. A TTP is mainly used to establish a secure communication channel between two parties where the TTP plays the role of a referee. There are lots of services that a TTP can provide, with an authentication service as the prominent service. Additional security related services that a TTP can provide include: access control, key management, or notary (non-repudiation) servers.

From a communication system point of view a TTP can provide either on-line, in-line, or off-line services. In case of on-line services (an authentication service) the TTP interacts in real- time with the parties who trust it. For in-line services, the TTP intercepts the path between the two communication parties if necessary by providing a translation between two encryption algorithms. When a TTP (such as a CA) provides off-line services, the TTP does **not** take part in the actual communication, but helps to enable the communication. [15]

We are concerned with the key management service of a TTP. In this thesis project, we implemented a very simple escrow agent as a TTP using an Apache web server. We will escrow the session master key after a successful secure communication session. The session master key should be stored in a secure database. Upon proper authentication the escrow agent will also provide the requested session master key to the LEA. In this case the TTP is responsible for operating the key escrow component. The TTP stores and retrieves the escrowed key and delivers the key to the LEA or government based on the specified warrant. When a TTP deals with the escrowed key it is often referred as an **escrow agent**. Denning & Branstad have described escrow agents in terms of the following characteristics [16]:

- Escrow agents can be entities in the government or private sectors. An escrow agent for the private sector is often known as a commercial or private key escrow agent.
- Escrow agents should be identified by their name and location.
- Escrow agent should be accessible during their hours of operation.
- Escrow agents should be secured against compromise, loss, or abuse of escrowed keys.
- Escrow agents must be certified and licensed with a government.

To escrow the session key with the TTP we use 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 [17].

2.5 Key escrow

Key escrow is a data security arrangement where the cryptographic keys are entrusted to a trusted third party who acts as an escrow agent. Specifically, the cryptographic keys necessary to decrypt encrypted data are stored in escrow and under normal circumstances these keys are **not** revealed to **anyone** without proper authorization[1]. When the agreement with the third party is made to escrow one or more keys the user generally specifies the terms under which the keys may be released.

The trusted third party as escrow agent will provide the keys to an entity after verifying that this entity has the proper authorization to receive the key. The authorized entity may be a government or law enforcement agency (LEA) representative who has the legal authority to access the content of encrypted communication or this entity may be an authorized corporate official that has the legal authority to access an employee’s communication due to a security concern [18]. The entity might even be the entity that deposited the key(s), in case they forget or lost their key(s). The details of how an entity establishes that they have authorization and the escrow terms are outside of the scope of this thesis project.

U.S. National Security Agency (NSA) first conceived the key escrow concept during the early 1990s. Their main motivation for introducing this concept was to enable the wide spread introduction of encrypted telephony, while preserving the ability to perform lawful interception. Their proposal was that government or LEA agents would have 24 hour availability to master keys which could be used to provide easy access to encrypted data. Another motivation for key escrow was the recovery of encrypted data by the entity that had originally encrypted the data. For example, a company could benefit from key escrow as a means of data recovery in case of an accident such as an employee’s death or a physical disaster that destroyed the key [1]. An important aspect of the proposal for key escrow was that the key escrow system should scale well (ideally there would be enough industrial or private paid use of the system that the cost to the government for the operation of the system would be zero).

2.5.1 The Clipper Chip

The most prominent and widely known key escrow implementation was “The Clipper Chip” developed and promoted by the U.S. Government in 1993. The Clipper Chip was developed as cryptographic device intended to protect private communication while at the same time permitting government agents to obtain the keys upon presentation of proper authorization [19]. An escrow agent or a Trusted Third Party (TTP) holds the keys.

The Clipper Chip was designed to be embedded in every telephony device (or added via an external “bump in the wire”). This chip would provide high quality encryption of all data passing through it. Every chip had a unique key and a unique identifier. This unique key would be stored for this identifier with an escrow agent. In operation the Clipper Chip would generate session keys to secure the session and the session key would be encrypted using the specific chip’s key and transmitted in the session along with the identifier. Therefore, once a specific chip’s key is known, then the content of *any* session encrypted by this chip can easily be recovered.

[1] Note that the sender and receiver have another means of exchanging the keys that they will use, thus in normal operation secret keys are *only* deposited for escrow.

Although the government could store the keys by themselves this would lead to controversy, thus the government decided to store the keys with one or more TTPs. To make it harder to get a key without proper authorization, every key was split into two parts that must be XORed together to produce the actual key. Each of these parts was stored by a different TTP. Thus the proper authorization must be presented to two different TTPs, who must each be convinced to reveal their part of the key. Then these two parts must be XORed to produce the original key. This has several advantages:

- If either of the TTPs refuses to reveal the part of the key that it is holding, then the full key cannot be retrieved.
- Each of the TTPs is simply storing what is effectively a random set of bits, so they can not themselves compromise the security of any of the communications encrypted with the Clipper Chip devices.

The Clipper Chip used a data encryption algorithm called Skipjack developed by NSA to transmit the data and it used the Diffie-Hellman key exchange algorithm to distribute the session keys between the pair of communicating Clipper Chips. The customized Skipjack algorithm added a 128 bit Law Enforcement Access Field (LEAF) that is sent in every session. This field contains the information necessary to decrypt the packet (i.e., it includes the identity of the chip and the encrypted session key). The Clipper Chip escrow system seemed to be very robust. However, it was abandoned in 1996 due to a serious security vulnerability discovered by Matt Blaze [20]. The vulnerability occurred because the Clipper chip used a 16 bit value in the 128 bit LEAF as a checksum to maintain the integrity of LEAF. Thus if a chip receives a packet and calculates a hash other than the received hash, then the receiving Clipper Chip would not process the packet further. Matt Blaze pointed out that a 16 bit hash was a sufficiently small field that a brute force attack could find another value for the LEAF that would result in the same hash. Thus someone could replace the valid LEAF field with a forged LEAF value, the receiving Clipper Chip would correctly process the packet - but later it would impossible to decrypt this packet using the key recovered from the escrow agent. This flaw enabled the Clipper Chip to be used as an encryption device while effectively disabling the key escrow functionality.

2.5.2 Why key escrow is problematic?

The main motivation (by the U.S. Government) for key escrow was to encourage the use of encrypted communication (particularly for official and corporate communications), while facilitating LI. The U.S. Government remains the main supporter for the implementation of a key escrow system. However, implementing a practical key escrow system is both complex and expensive. Moreover, correct implementations of such a system must avoiding both security flaws and make the abuse of such a system very difficult. In the following paragraphs we will briefly explain the technical drawbacks of a key escrow system.

Another set of problems facing key escrow is that key escrow is widely view as a potential threat to individual privacy and violation of human rights. These issues lie outside the scope of this thesis, but have been well documented in the press, see for example [21] [22].

2.5.2.1 Complexity

It is commonly believed that a perfectly secure cryptographic system is extremely difficult to create. Addition of new cryptographic parameters increases *design complexity,* as all the keys need to be stored and securely maintained. Unfortunately, key escrow adds lots of complexity to a cryptographic system. For example, the major weakness of the Clipper Chip was not in the Skipjack algorithm, but rather the design choice of a short checksum. Furthermore, a successful attack against the Skipjack algorithm was published the year after the details of the algorithm were published.

Due to the rapid growth of Internet the ability to scale to very large numbers of users and devices is vital for a successful implementation of a key escrow system. Today, there are millions of users using encrypted communication and lots of TTPs and LEAs worldwide. Establishing a key escrow system would increase *operational complexity,* as every LEA would expect and require fast response from each key escrow system. The complexity of key escrow can be mitigated to some extent by a well-designed system, well-trained staff, and proper technical control; but operational vulnerability cannot be completely avoided. In a key escrow system it is essential that only authorized entities be permitted to receive the requested key(s). Unfortunately, authentication documents such as a passport or birth certificate can easily be forged as can an authorization document -- which could lead to an unauthorized entity gaining access to a deposited key.

2.5.2.2 Cost

Today cryptography is becoming increasingly inexpensive. However, a key escrow system can add lots of cost; depending on the scale of the key escrow system. Deploying a key escrow system that extends beyond a national boundary adds lots of *operational cost* due to the cost of maintaining and controlling sensitive and valuable key information securely over a long period of time. It requires a substantial number of well-trained staff (as the facility must operate 24 hours per day – every day of the year) and high-assurance hardware and software systems to meet government requirements. In this regard new products might need to be designed which incurs substantial *product design* cost. Moreover governments and LEAs may also need to test and approve the entire key escrow system adding potentially substantial costs associated with *government oversight*.

One of the most difficult issues is the question of who is to pay for the operation of the key escrow system. This raises the related questions of when does each entity have to pay and how much do they have to pay?

2.5.2.3 Security vulnerability and risks

The major disadvantage of key escrow system is the introduction of new security vulnerabilities, which can jeopardize the proper operation, underlying confidentiality, and ultimate security of encryption system [23][1]. Some of the security vulnerabilities and potential risks are:

- **Potential inappropriate or illegal access to private data:** Every key escrow system is expected to provide the requested escrowed key(s) to a LEA after proper authorization. Moreover, the parties who have deposited keys with the TTP should **not** be aware of the fact that their key(s) have been requested by a LEA, whom has requested the key(s), or when the key(s) was/were requested.

If communicating parties knew that their keys had been obtain, then they could potentially act to prevent further communication from being compromised by discontinuing the use of these keys and they could also take other actions to make monitoring or interception harder. However, the fact that the party who has deposited a key is not aware that someone has obtain this key means that this party has no way of preventing illegal or inappropriate use of this key; thus potentially compromising the privacy of data or communication session content.

- **Insider abuse:** One of the most dangerous threats of a key escrow system arises when trusted persons misuse their position. An employee of a key escrow system may be intimidated, bribed, blackmailed, … to reveal a key. This key could be used for an illegal act (such as blackmail or extortion) against an individual or a company. An untrustworthy employee can reveal a company's confidential information. An unethical employee with access to a key could fabricate the content of the session, in order to blackmail a user. An unethical LEA agent could use a key to fabricate evidence. Unfortunately, the user cannot prove that the data has been fabricated, as the fabricated data uses the correct key. This kind of misuse can be even more dangerous than inappropriately or illegally revealing encrypted information, as it may easily be used to destroy an individual's or company's reputation and financial status.

- **New targets for attack:** If the keys are stored by a key escrow system in a central database, then this central database becomes a new target for attacks. It is a particularly rich target because if the attacker can extract keys from the database it will enable the attacker to compromise the data or communication of a company or individual. One of the worst aspects of such an attack is that it could be used to compromise *many* keys. Although distributing the databases and storing parts of each key in different databases can mitigate the risk of a successful attack, this will increase the operating costs and may also increase the response time to deliver the keys to a LEA.

- **Destruction of forward secrecy:** One of the major disadvantages of key escrow system is the destruction of forward secrecy. Forward secrecy is a security feature where by a secure session cannot be retrieved after the session is over *even if the session key for next session has been compromised*. Usually a system with forward secrecy destroys the session keys when the session is over, i.e., the communicating parties do **not** store the session key. Forward secrecy is simple to design and implement. Moreover, forward secrecy is desirable because it increases security and decreases the cost of a system, since the secrecy of the keys only needs to be maintained for the duration of the part of the session that a session key is used for. Unfortunately, key escrow destroys this property since if the master key for a session is stored with the escrow agent (TTP) at the start of a session, then the derived sessions keys are vulnerable – even if the session keys for the media streams are changed during the session (as these keys can be derived given knowledge of the master key and the earlier session keys).

- **Different kinds of keys to deposit:** Various kinds of keys are used for various kinds of communication. Some of these keys are used to provide confidentiality while others are used to provide authenticity. Some keys are used for stored data, while some others keys are used for real-time data. Keys that are used to encrypt data for storage need to be preserved for the lifetime of the data (potentially a very long period of time for documents such as deeds, sales contracts, etc.); while some keys used to secure real-time data may not be of interest to the communicating parties after the session is over. Deciding which keys need to be deposited in the TTP for the recovery of encrypted data is a critical issue. This is particularly a problem when the potential depositor has a different expectation of the lifetime of the key's usefulness than LEAs. For example, as noted above a set of communicating parties might have no interest in escrowing the master key used for a corporate videoconference, while a government regulator might want to have access to this key (potentially many years after it was deposited). Thus the expectation of LEA is that all keys would be escrowed, leading to a lot of keys needing to be deposited with the TTP. This is a challenge for key escrow system as they must implement a suitably scaled system.

A successful key escrow implementation needs to address a lot of challenges and potentially suffers from lots of vulnerabilities when deployed on a large scale. At present there are no successful implementations of a large-scale key escrow system. Eric Verheul, et al. have presented the necessary and desirable criteria for the deployment of worldwide key escrow system and also described a new concept of using a PKI as a fraud detection alternative to key escrow system that will not hamper law enforcement [24].

However, there is still pressure from governments on telecommunication operators and manufacturers to adopt key escrow in order to reduce the difficulties that LI faces. In this thesis project we will assume that there is an operational key escrow system and that registered users can use this system. Issues of the cost of becoming a registered user and the cost of retrieving a key from one or more TTPs are outside the scope of this thesis project.

In this thesis it is assumed that one or more TTPs exist and that they have implemented a suitably scaled infrastructure to receive all of the keys that their registered users wish to deposit. However, this thesis project will consider the time and communication overhead required to authenticate the registered user to the TTP and to deposit a key.

2.6 Secure Real Time Transport Protocol

The Secure Real Time Transport Protocol (SRTP) [25] is an application layer protocol that is designed to secure the Real Time Transport Protocol (RTP) traffic. SRTP defines a secure profile for RTP that provides message encryption, message authentication and integrity protection, and replay protection to every RTP packet for both unicast and multicast applications. Just as RTP is closely related with the Real Time Transmission Control Protocol (RTCP) -- which provides control functionality for an associated RTP session; SRTP has a sister protocol Secure Real Time Transmission Control Protocol (SRTCP) that provides the same security to RTCP as SRTP provides to RTP.

The security services (confidentiality, integrity and authenticity, replay protection) provided by SRTP are optional and independent from each other except that SRTCP integrity protection is *mandatory* because alternation of RTCP could disrupt the processing of the associated RTP stream[25]. Moreover, the use of SRTP is independent of the underlying transport protocol. Thus SRTP can protect RTP transported over UDP, TCP, or any other transport protocol.

SRTP provides security services to RTP on a per packet basis. It provides confidentiality to the RTP payload by encryption and provides integrity protection to both the header and payload of every packet by adding an authentication tag. Figure 2-3 shows the format of an SRTP packet. The (large) blue box shows the packet contents that are integrity protected and the (smaller) green box shows that only the actual payload of the RTP packet is encrypted.

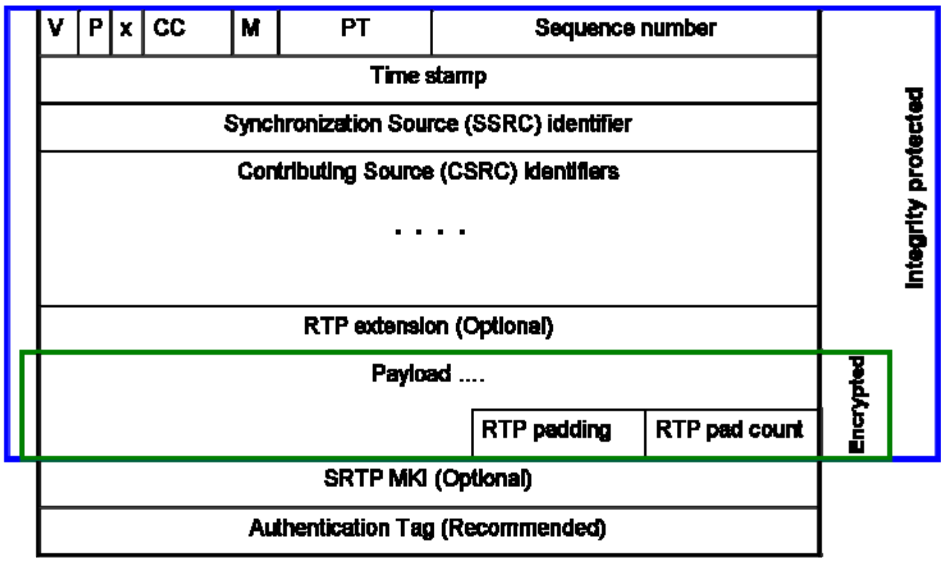


Figure 2-3: SRTP packet format

There are two additional fields that can be present in an SRTP packet. The first (optional) field is a variable length Master Key Identifier (MKI) field. The MKI field is used by the Key-Management protocol and determines which master key has been used to derive the session keys. Additionally, the MKI can also be used by the Key-Management protocol for re-keying in order to identify a particular master key within the cryptographic context. The other optional **but recommended** field is an authentication tag that has a configurable length and provides authentication of both the RTP header and payload. This field also indirectly provides replay protection by authenticating the packet's sequence number.

One of the important optimisations used in SRTP is the use the RTP sequence number rather than adding a new field in the SRTP header. A sequence number is necessary for synchronization, which in turn is a prerequisite for security processing. However, the sequence number in the RTP header is only 16 bits -- which implies that this sequence number will recur after every 2^{16} packets. This small sequence number range would require re-keying and re-keying would require the execution of a key management protocol, which is undesirable and resource consuming. SRTP solves this problem by extending the RTP sequence number with a 32 bit local counter called the Rollover Counter (ROC). This ROC is incremented when there is a wrap of the RTP sequence number. The ROC together with the RTP sequence number is known as the SRTP Index or simply Index. This index is used to generate session keys. Fortunately, there is no need to transmit the ROC in the packet, limiting the expansion of the packet, which is big advantage of SRTP over alternative protocols that do not take advantage of the existing RTP sequence number.

2.6.1 Cryptographic context and key derivation

To provide security to an RTP session the sender and receiver must keep cryptographic state information (security parameters) known as cryptographic context for each media stream. Some examples of these security parameters are: the per packet SRTP index, the key(s), an indication of the cryptographic algorithms used, key derivation rate, key lifetime, and current ROC. Some of these parameters are fixed for the duration of the entire session, while others need to be updated per packet. SRTP uses different keys for encryption and authentication. SRTP actually requires six different session keys for the protection of each RTP media stream. Three of these session keys are required for the RTP traffic and a similar triplet are used to protect the associated RTCP traffic. All of these session keys are generated from a single master key. The master key is the key that was exchanged via the key management protocol (e.g. MIKEY) (In our case we will escrow this master key with an escrow agent at the end of a session.).

SRTP uses a key derivation function in the form of a pseudo-random function (PRF) which takes the master key and some other parameters as input, then produces the six session keys as output (see Figure 2-4). The other inputs to the PRF are a master salt key provided by the key management protocol, derivation rate, and a label (the SRTP index) [26]. The master salt key is used to prevent key collision and time-memory trade-off attacks. The complete process is also known as *key splitting*.

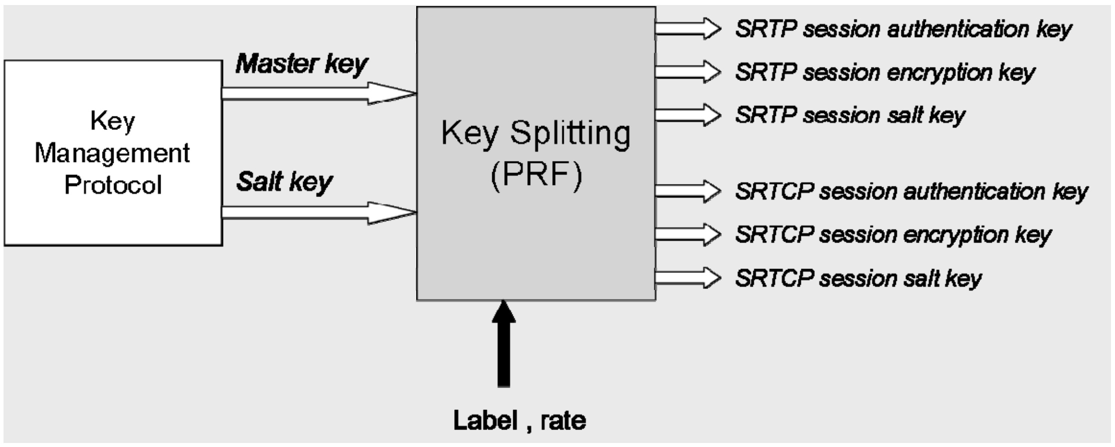

Key Management Protocol
Master key
Salt key
Key Splitting (PRF)
SRTP session authentication key
SRTP session encryption key
SRTP session salt key
SRTCP session authentication key
SRTCP session encryption key
SRTCP session salt key
Label , rate

Figure 2-4: SRTP key splitting (Adapted from [26], Figure: 24)

2.6.2 SRTP packet processing

This section briefly explains how SRTP packets are processed at both sender and receiver. The following subsection will briefly explain the cryptographic algorithm used for encryption and authentication

SRTP at the sender takes an RTP packet as input and transforms it into an SRTP packet and forwards it to a transmission layer protocol for transmission. The first task when processing an SRTP packet is to retrieve the correct cryptographic context. The next task is to derive the session keys from the master key. The RTP payload is encrypted using the appropriate session key and if message authentication is required then a message authentication code is calculated and appended to SRTP packet. Optionally a MKI field can also be added. The resulting SRTP packet is passed to the transport layer for transmission to the receiver.

Upon the arrival of the SRTP packet at the receiver the first task is to retrieve the appropriate cryptographic context to be used. The next task is key splitting to generate session keys from the master key. The next task is to check whether the packet is replayed or not. SRTP performs this check by comparing the SRTP index against a replay list. Next the receiver authenticates the packet. The packet is dropped if authentication verification is unsuccessful or the packet is a replay. Otherwise the packet's encrypted portion is decrypted and the authentication tag is removed and the RTP packet is forwarded to the next higher layer for processing.

2.6.3 How encryption and authentication is done?

By default SRTP uses the Advanced Encryption Standard (AES) for encryption/decryption of the RTP packet's payload to provide confidentiality. This algorithm was chosen due to its low computational requirements. AES can be used with various lengths of keys and block sizes. In SRTP, a 128-bit block is encrypted with a 128-bit key. SRTP supports two kinds of stream ciphers. The differences between these two different streams ciphers are the mode AES is run in: counter mode or f8 mode. The AES algorithm is used in a chain to produce a stream of keys that are used as a one time pad to encrypt the actual data by a bit wise logical XOR operation. AES in counter mode is the default method used by SRTP. When AES is run in counter mode, AES is applied to consecutive integers to build a key stream. Figure 2-5 depicts the operation of AES in counter mode. The input to the stream cipher is the session encryption key ($k_{e,}$) some synchronization data called an Initialization Vector (IV) -- formed based upon the SRTP index of the packet, the synchronization source (SSRC) field carried in the RTP packet header, and the SRTP session salt key. In case of f8 mode a similar procedure is used to create the key stream, but the difference is that when AES is run in f8 mode the IV depends on additional RTP header fields, such as the timestamp, the sequence number, the source identifier, and other flags.

The above method is used to provide confidentiality to RTP, but does not prevent an attacker from forging RTP packets. To provide authentication and integrity protection SRTP uses HMAC-SHA1 as keyed hash function. Integrity protection for RTP includes the RTP header, RTP payload, and the local ROC. The HMAC uses a session authentication key (k_a) derived from the master key. HMAC-SHA1 produces a 160-bit output, which is truncated to 80 or 32 bits to form a message authentication tag that is appended to the SRTP packet by the sender.

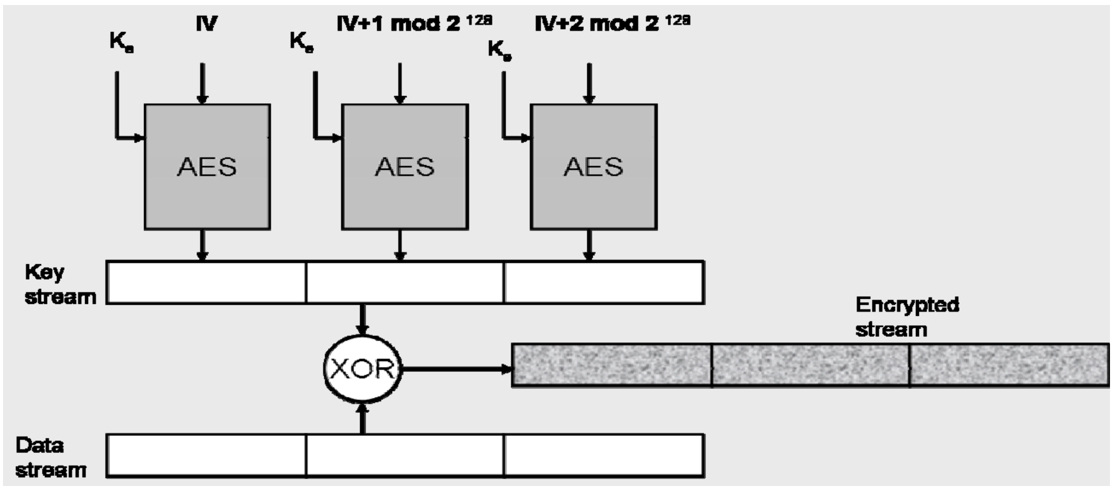


Figure 2-5: AES in counter mode

The receiver will calculate the hash similarly and check whether the locally calculated message authentication code corresponds to the received one. If both message authentication codes are equal, then the packet is accepted and sent for play out, otherwise the packet is dropped.

It is important to note here that 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. Thus if a LEA learns the master key used for this session it is possible to decrypt the encrypted content and to authenticate the packet. However, given the master key it is also possible to forge SRTP contents that would be valid, if this SRTP packet were to arrive at the receiver before the SRTP packet that the sender sent, then this forged packet would be accepted by the receiver and the actual sender’s packet would be rejected as a replayed packet! This is why it is important that the master key **not** be escrowed before the session has ended.

Further more it should be obvious that given the master key it is possible to forge packets and make them appear to be valid packets in an SRTP stream. It is this weakness that this thesis project will attempt to overcome.

2.7 Secure Real Time Transport Control Protocol

The Secure Real Time Transport Control Protocol (SRTCP) is a sister protocol of SRTP that provides security related features to RTCP. More specifically, SRTCP provides the security related features of confidentiality, authentication, and integrity protection to RTCP. SRTCP provides the confidentiality to RTCP packets by encrypting them. It provides authentication and integrity protection by adding an authentication tag in the SRTCP packet in the same way as SRTP does for RTP. Figure 2-6 shows a SRTCP packet. The shaded portion at the beginning corresponds to the SRTCP header. The fields inside the (large) blue box are integrity protected and fields inside the (smaller) green box are encrypted.

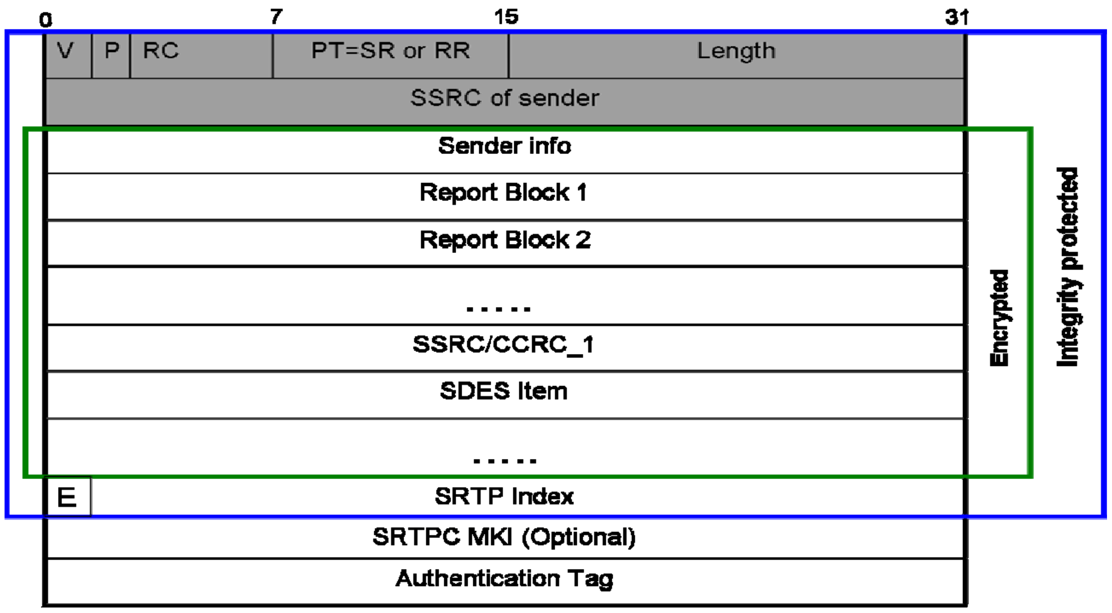


Figure 2-6: SRTCP packet format

SRTCP adds three **mandatory** fields (SRTCP index, encryption flag ‘E’, and authentication tag) and an *optional* MKI field. The encryption flag ‘E’ indicates whether the SRTCP packet is encrypted or not. The SRTCP index is a 31-bit field, which is an *explicit* index in contrast to the *implicit* index utilized by SRTP. The index value is set to zero before the first packet is sent. For every consecutive packet this value is incremented by one. If re-keying is required, then this index value must not be set to zero again and the situation is same for SRTP. For SRTP as the rollover counter is 32 bits long and the sequence number is 16 bits long, the maximum number of packets belonging to a given SRTP stream that can be secured with the same key is 2^{48} using the predefined transforms. After that number of SRTP packets have been sent with a given (master or session) key, the sender **must not** send any more packets with that key. However, since SRTCP uses an *explicit* index of 31 bits the number of packets that can be secured with SRTCP is 2^{31}.These limitations provide an upper bound on the amount of traffic that can pass before the cryptographic keys are changed [25].

The last mandatory field is an authentication tag, which is a 32-bit field by default, but whose length is configurable. This field contains authentication data similar to that for SRTP. The main difference is that in case of SRTP integrity protection was optional, but in the case of SRTCP it is mandatory. The motivation behind this is that RTCP is a control protocol, i.e., it can perform critical operations (including terminating the session), hence it is important to ensure the integrity of each SRTCP packet.

SRTCP shares most of the security parameters of SRTP, including the master key and the kinds of protection that are offered. However, a separate protection suite can also be specified for the RTCP traffic; the optional MKI field can be used to indicate this alternative suite. By using an alternative suite it is possible to expose the SRTCP traffic to an operator, for example for network management and quality of service (QoS) purposes; while preventing the operator from being able to decrypt the SRTP traffic.

2.8 Multimedia Internet KEYing (MIKEY)

MIKEY is a key management protocol that provides an efficient key agreement mechanism for peer-to-peer and (small to medium sized) group communication. MIKEY was designed to provide key management functionality for IP multimedia communication in heterogeneous networks. A multimedia session may include several media sessions, such as a bi-directional audio stream, a bi-directional video stream, and/or and a HTTP session [26]. To protect these separate media sessions different security protocols may be required (e.g., SRTP for audio and video sessions and TLS for HTTP sessions). A separate key management protocol may be required for separate security protocols to exchange security parameters and keys. However, running several different key management protocols for a single multimedia session is not a satisfactory solution, as every key management protocol adds delay due to extensive cryptographic operations and due to the impact of the roundtrip time of messages that must be exchanged. Minimizing the delay for multimedia key exchange was one of the design considerations of MIKEY. A novelty of MIKEY is its ability to instantiate the security of all media sessions within a single multimedia session in a minimum amount of time.

The data stream protected by a single instance of a security protocol (i.e., a secure session) is known as a *crypto session* and the multimedia session for which MIKEY is negotiating security parameters is called a *crypto session bundle*. A crypto session bundle is a collection of several crypto sessions. MIKEY can exchange separate Traffic Encryption Keys (TEKs) for each crypto session or alternatively it can agreed upon on a TEK Generation Key (TGK) for the whole crypto session bundle from which separate TEKs can be generated for each crypto session in a secure way. In the case of SRTP, this TEK acts as session master key. Figure 2-7 shows the basic key agreement of MIKEY for a data security protocol such as SRTP.

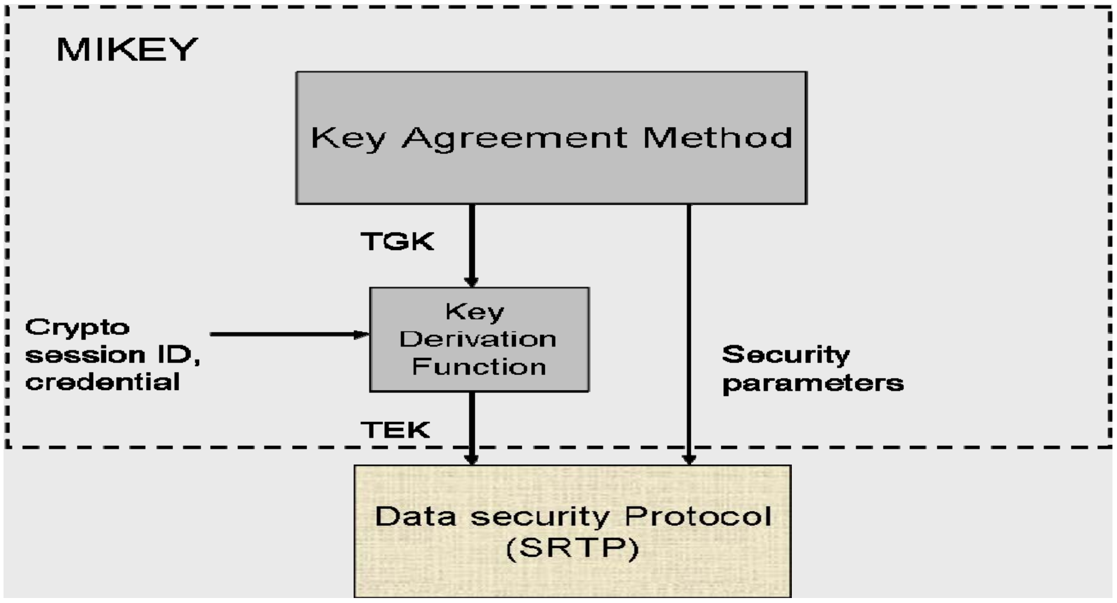


Figure 2-7: MIKEY key agreement procedure (Adapted from [27], page 33)

Figure 2-7 shows that MIKEY generates keys for a data security protocol. However, the MIKEY message itself encrypted and integrity protected in order to provide end-to-end security between communicating peers. Thus MIKEY generates keys for itself, in order to encrypt the message and the security parameters that will be signalled in-line. The cryptographic context used to encrypt the TGK/TEK depends on the method used, e.g., a pre- shared or public key infrastructure (i.e., Certificate based). A MIKEY message (in an Initiator or Responder message) may contain several payloads, each containing different fields for carrying the relevant information and signalling. An example payload is shown in Figure 2-8. Here the encrypted data field contains the actual encrypted key material and other relevant fields containing the necessary information for this encryption scheme, such as encryption algorithm, length of the key, etc.

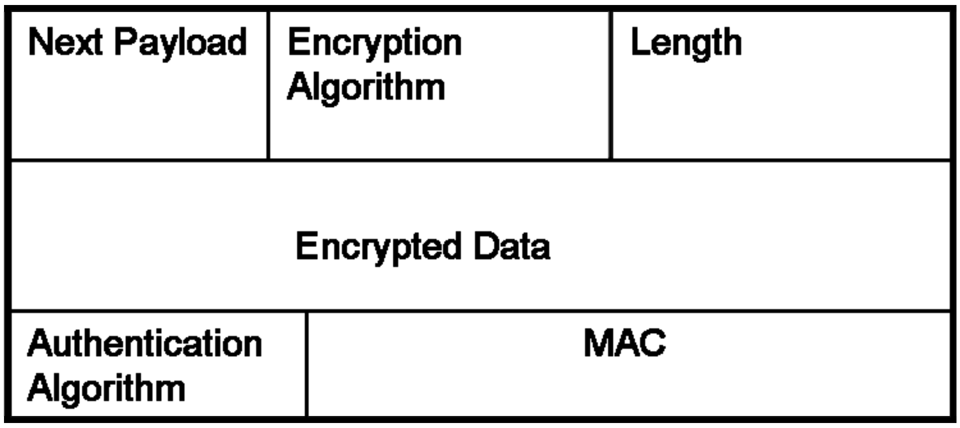


Figure 2-8: MIKEY message payload (Adapted from [26], page 40)

Another design goal of MIKEY was to minimize the number of round trips required for negotiating security parameters or cryptographic context. All other key management protocols require at least three round trips for successful key agreement. In contrast, MIKEY requires only one or a half roundtrip depending on the specific method used. A challenge-response method is frequently used by a key management protocol for authentication and replay protection; however, this requires at least three messages. On the other hand MIKEY adopts a two-way handshake (one round trip) method instead of a challenge-response (i.e, a three way handshake) method that uses a timestamp as challenge. Actually, MIKEY incorporates the key agreement process into the media negotiation process. The media negotiation uses only two messages and is usually performed using an offer/answer model via the Session Description Protocol (SDP). In this process the initiator makes an offer based upon its own media processing capabilities and the responder choose among the proposed media streams (each with their own encoder/decoder (CODEC) and other parameters). Next subsections briefly explain three different methods that can be used by MIKEY.

2.8.1 MIKEY Methods

MIKEY provides three different variants of key agreements: pre-shared key, Public Key Encryption, and Diffie-Hellman [28] exchange. In the first two methods the keys are pushed to the recipient in a secure way and the key agreement procedure can be completed within one half or a single roundtrip; but in case of the Diffie-Hellman exchange method both communicating parties must contribute to form the key and a single full roundtrip is always required to complete the key agreement procedure.

2.8.1.1 Pre-Shared Key method

In the pre-shared key (PSK) method both peers posses a shared key (K) prior to communication between them. This pre-shared key (K) is used to derive an encryption key (K_e) and an authentication key (K_a) through a key derivation function. Figure 2-9 shows the pre-shared method.

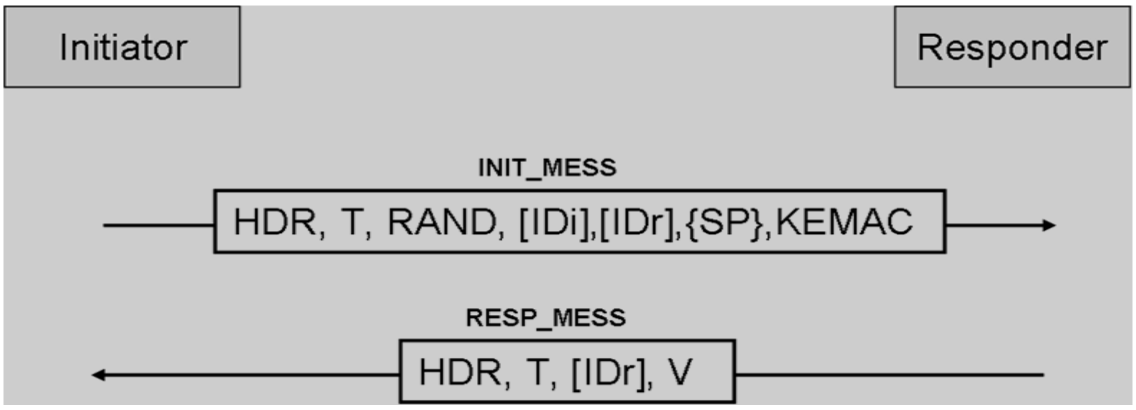


Figure 2-9: Pre-shared method of MIKEY
Where, [ ] indicates optional parameters and { } indicates zero or more occurrences of a parameter, HDR=Header Payload, T= Time, RAND=Random value, IDi=Initiator identification, IDr=Responder Identification, SP=Secure Policy Payload, KEMAC=Encrypted sub payload containing TGK and MAC, V= Verification Payload carrying MAC of the entire message (Adapted from Figure 14 of [26])

The INIT_MESS is created by the initiator. This message includes several fields as shown in Figure 2-9. The Header Payload field (HDR) contain the identifier of the crypto session bundle, number of crypto sessions, and a method for mapping the crypto sessions to the data security protocol (currently only supports SRTP) for which MIKEY is exchanging the parameters. The Secure Policy Payload (SP) field contain the security parameters for setting up the data security protocol. The most important field is KEMAC; this includes the encrypted sub payloads (see Figure 2-8) carrying the TGK/TEK and a message authentication code. Upon receiving the INIT_MESS, the responder first checks the authenticity of the message based on the message authentication code value. Next the receiver retrieves the SP and TGK/TEK and if requested by the initiator, then the responder sends back a RESP_MESS.

2.8.1.2 Public Key Encryption method

Public Key Encryption (PKE) method is similar to the pre-shared method. However, each peer requires a pair of public/private keys for encryption and signature. Figure 2-10 illustrates the PKE method used by MIKEY. It differs from the pre-shared method in that instead of using a pre-shared key to generate the encryption key (K_e) and authentication key (K_a,) a random *envelope key* is first generated – this envelope key is in turn used to generate K_e and K_a. This envelope key is encrypted by the initiator using the responder’s public key and sent to the responder in the PKE field.

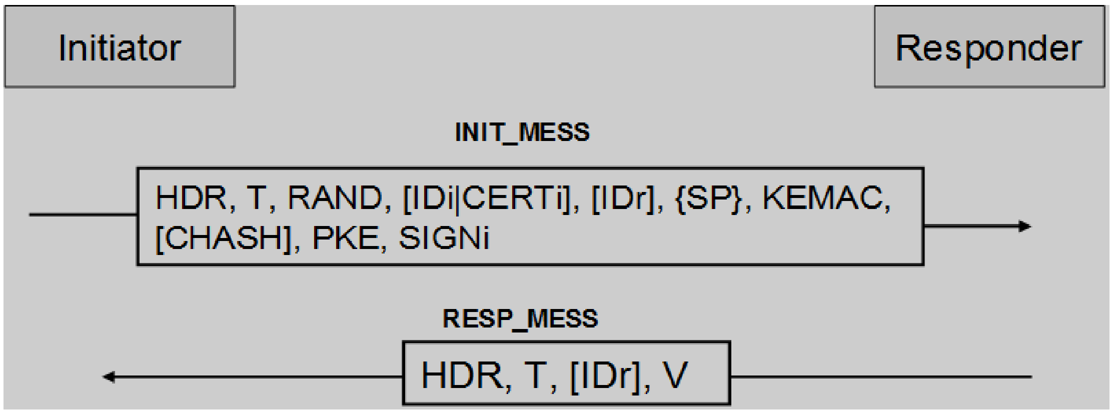

Initiator
Responder
INIT_MESS
HDR, T, RAND, [IDi|CERTi], [IDr], {SP}, KEMAC, [CHASH], PKE, SIGNi
RESP_MESS
HDR, T, [IDr], V

Figure 2-10: Public Key Encryption (PKE) method of MIKEY Where, CERTi= Initiator Certificate , PKE field Contain envelope key encrypted with responder's public key SIGNi field contain the signed value of the message by Initiator's private key CHASH field contain an indication of which public key of the responder has been used (Adapted from Figure 15 of [26])

2.8.1.3 Diffie-Hellman method

The Diffie-Hellman (DH) method is optional and requires a full round trip to complete the key agreement. This method differs from the first two methods as the key is not pushed to the peers, but rather each peer contributes to form the key.

In this method, both peers need to have public/private key pairs for signatures in order to authenticate each other and to protect against a man-in-middle attack. This scheme is the most computationally intensive due to the increased number of public key operations, but provides both greater flexibility and perfect forward secrecy. Figure 2-11 illustrates the Diffie-Hellman method of MIKEY key agreement.

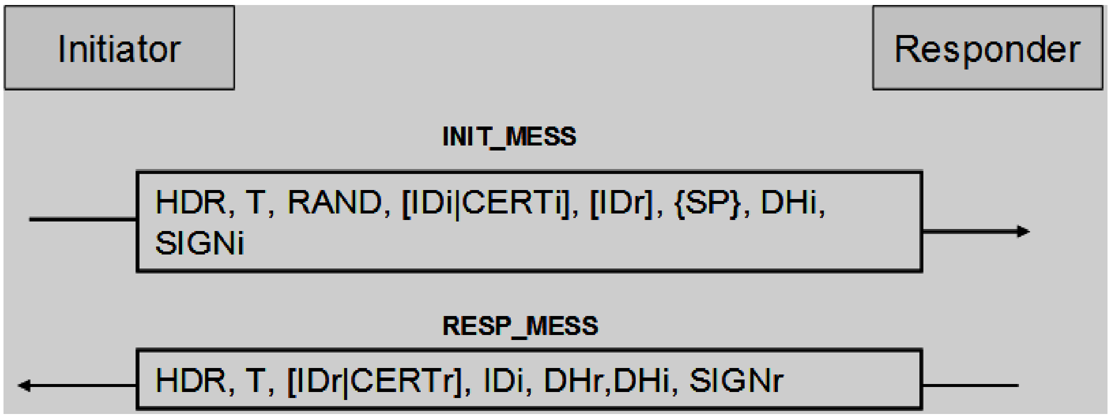

Initiator
Responder
INIT_MESS
HDR, T, RAND, [IDi|CERTi], [IDr], {SP}, DHi, SIGNi
RESP_MESS
HDR, T, [IDr|CERTr], IDi, DHr,DHi, SIGNr

Figure 2-11: Diffie-Hellman method of MIKEY Where, DHi field contain the Diffie-Hellman public value calculated by Initiator DHr field contain the Diffie-Hellman public value calculate by responder (Adapted from Figure 16 of [26])

2.9 Minisip

Minisip [29] is a SIP user agent that has been developed by students at KTH and others. It has been used as the platform for a number of master's thesis projects, including the first public implementation of SRTP [27] and the first public implementation of MIKEY [30]. Details of the design of minisip and its implementation decisions and performance are given in the licentiate thesis of Erik Eliasson [31].

The functions of minisip that have been utilized are shown in Table 1. Details of the modifications to minisip to implement the proposed design are described in detail later in the thesis.

Table 1: Some important functions of minsip that have been utilized

Function Name	Library	Description
getSignature( )	Libmcrypto	To implement the signing operation we will use the *getSignature( )* function of the SipSim class of libmcrypto library. This function is used to compute a digital signature of fixed size data provided the private key of signeer. The *getSignature( )* function is a virtual function defined in the SipSim class and implemented in the SipSimSoft class.
hmac_sha1( )	Libmcrypto	This function provides the hashing operation. The *hmac_sha1( )* function takes variable size data and produces a fixed size hash using the HMAC_SHA protocol. We will use this function to calculate the hash of SRTP blocks.
genAuth( )	Libmikey	This function generates the authentication key from the master key. This function is defined inside the KeyAgreement class. The authentication key is used as one of the parameters for *hmac_sha1( ).*
tgk( )	Libmikey	This function returns the session master key for the current session. This master key is used to generate several session keys. We will escrow this master key with our escrow agent. This function is defined inside the KeyAgreement class.

Chapter 3: Related Work

In this chapter some related work concerning detection or prevention of the forgery of RTP content and how to obtain the session keys for a secure VoIP session are presented. Section 3.1 describes research dealing with ensuring the non-repudiation of a VoIP conversation by using asymmetric cryptography. Relevant work dealing with obtaining the session encryption key for a CALEA compliant network is presented in section 3.2. Finally a secure protocol to establish a session's symmetric key is described in section 3.3.

3.1 Security and non-repudiation for a Voice-over-IP conversation

Hett et al. [32] [33] presented a way to provide non-repudiation for a VoIP conversation. They mainly focused on ensuring the integrity of a voice conversation; authentication of the speakers; and the ability of the speakers to have non-reputability after a call has completed. To achieve these goals they computed a digital signature over the whole conversation in both directions. They used public key cryptography to perform the signing of data and they assumed a PKI structure was available.

The main scenario for this work is a bi-directional interactive conversation between two parties where one party signs the conversation and sends it to the other party as a declaration of his or her commitment to this content. Both parties sign the complete conversation including both channels (comprising everything that each party has said). Both parties also store the signed conversation in a secure archive. As a result, either party can later prove to third parties or a court that the call occurred or the call consisted of the claimed contents. If either party fails to store the conversation in an archive or deletes it, then the other party can deny that the call ever occurred.

Instead of signing individual RTP packets the authors introduced a new concept of intervals and interval signatures. The complete session is divided into intervals and all of the packets in an interval are collected together and a hash is computed over these packets. For the sake of simplicity the authors have used timer-based events. Every second the collected packets are sorted by sequence number and their hashes are assembled in a data-structure with additional meta-information, such as direction, sequence numbers, and time. This small data-structure is then signed with a conventional signing algorithm (such as RSA) using the private key of the sender. These signed values are then sent to the other party who then stores them together with the collected RTP packets he or she actually received. Note that the RTP packets containing the content of the session are transported only once as in a normal RTP stream.

In this approach signatures and hashes are interleaved to ensure that there is a continuous stream of signatures building an unbreakable chain. The reason behind interleaving the signatures and hashes is that if they were separated an attacker could replace some part of communication or could cut the signatures out. Additionally, these researchers suggested using biometric data contained in the natural language content for speaker dependent identification in order to detect forgery of call contents.

3.2 A CALEA compliant network to obtain session encryption key

In [34], Stephen D. Guhl has examined the impact of SIP signalling messages and media stream encryption and a proposed architecture for a key management system that would obtain session encryption keys used in a VoIP session. The author also claimed that the architecture will provide law enforcement with a more timely ability to obtain and decrypt signalling and media data without reducing the security of the Internet or users and that the architecture would also be applicable to Communication Assistance for Law Enforcement Act (CALEA) compliant networks. For proper functioning of the architecture some requirements must be pre-established. One such requirement is the availability of session keys to the ISP. If an end user attempts to establish a secure session (media and signalling) over SIP using keys not available to the ISP, then the session set-up will be rejected and a corresponding error message will be sent to user as a response message. The user agent will then be offered a negotiation process to establish credentials (i.e., session encryption keys) in an authenticated secure manner. If LI is required for the current session, then keying information will be stored with a timestamp correlating it with the media and signalling session; otherwise the keys will be retained until the current session ends – when they will be discarded. Another requirement for the proposed architecture is the use of RSA digital signature cryptography and its use of a public key infrastructure.

Two approaches have been introduced to resolve the security issues concerning how to obtain the session encryption keys. In both cases, the key issue is that as the architecture utilizes public key cryptography, hence some means is necessary to obtain the corresponding private key. In either of the approaches the SIP proxy server needs to be modified to adapt to the new architecture. As an alternative the author proposed the use of a Session Border Controller (SBC) as an intermediary to establish a SIP session. The SBC acts as a user agent server to establish a border between the public and private VoIP network. To accomplish this, the SBC utilizes a back-to-back user agent (B2BUA) that responds to SIP requests from any user agent in the public network, then apply policies and finally forwards the modified request to the target user agent in the private network. When a call is initiated from the private network, then the B2BUA functions in reverse. In the next subsections we briefly describe two approaches proposed by the author.

3.2.1 The LI mediation device initiating the acquisition of the private key

In this approach the mediation device initiates the process to obtain the session keys. When a LEA requests a LI the mediation device will determine the relevant the certificate authority (CA) and obtain the private keys of the appropriate parties in an secure manner. If a user agent now initiates SIP signalling to set-up a call the SBC will query the mediation device to verify whether the private keys have been obtained or not. If SBC gets a positive response, then it informs the mediation device that a call has been established and sends interception related information (including the session key) to the interception point and allows the call to be established. If the SBC gets a negative response this means the call is not subject to LI, thus the SBC will wait for an appropriate amount of time before allowing the call to continue. The reason for waiting is to ensure that all calls have a similar call set-up delay - whether subject to intercept or not. Without this additional waiting, it would be possible for the subject to detect that their calls were subject to intercept by monitoring the call set-up delay.

The major drawback of this approach is the exposure of the session's private keys; hence the Internet community is very reluctant to implement such a scheme.

3.2.2 Session border controller intermediary security negotiation

In this approach the user agent negotiates with the local SBC for security purposes using the public key of the SBC for the S/MIME encryption of messages to be sent to the SBC. The SBC will use its private key for encryption when establishing S/MIME or other security requirements. There will be SBC to SBC negotiation (as necessary) to complete the security negotiation process. The initiating user agent establishes a SIP signalling session with the local SBC. This communication may utilize specific pre-established security suites for SDP authentication, integrity, and privacy. If the user agent is subject to LI, then the SBC obtains the required intercept related information and forwards this to the mediation device. Before sending the data the SBC establishes a mutually authenticated secure connection to the mediation device using a defined cipher suite for encryption and integrity protection. In addition to sending the interception related information the SBC will also send the negotiated keys for the SRTP session to the mediation device at the end of the key negotiation process. If the message contents are required for LI, then SBC will capture individual packets and forward them to the mediation device after getting the proper instructions from the Mediation Device. Finally the mediation device will forward all relevant intercept related information and call content (over a secure channel) to the LEA.

This approach has some advantages over the former one. In this approach no new security infrastructure is created with the ISP and the user agent has the freedom to choose any set of standard cipher suite to secure a conversation.

The author has performed simulations and the results of this modelling have demonstrated that utilizing a SBC as an intermediary device in a LI process is a reasonable solution to provide the desired security to the user agent and at the same time providing accessibility of call contents and intercept related information to a LEA. The simulation results also suggest that the number of LI processed on a single SBC should be kept under a threshold, otherwise the processing will exceed the available resources of the SBC.

3.3 VIPSec

Zisiadis et al. [35] have presented a voice interactive personalized security (VIPSec) protocol for media path key exchange to securely establish a symmetric session key for ensuring end-to-end secure communication. Although not directly related to our work, VIPSec is presented here as it is deals with voice communication security.

VIPSec is a symmetric key exchange protocol that uses the *media path* to exchange the symmetric key during the call set-up, i.e., before any voice communication starts. This symmetric key is used to encrypt/decrypt any media exchanged by the application layer for this session. The communicating peers commit to a challenge/signature token exchange before the voice communication takes place and the integrity of these signatures are confirmed verbally when the voice communicates starts. The main idea of VIPSec is that the users initially exchange random numbers encrypted with their private keys, then they exchange their respective public keys. The reason behind the exchange of public keys is that VIPSec does not rely on a Public Key Infrastructure (PKI). Next the initiator of the session creates a symmetric key, encrypts it with the public key of the responder and sends it to the responder. At this point a symmetrically encrypted communication channel is established. Finally, both the initiator and responder verbally confirm the exchanged numbers as a proof of integrity protection.

VIPSec is based on some predefined assumptions and the author claims some unique cryptographic features. Table 2 shows some of the important cryptographic features claimed by the authors. Experimental analysis of VIPSec suggests that typical end user terminals easily meet its requirements. Moreover, the authors also suggest that by optimising the application's performance and considering the low requirement upon the end device that its global use is feasible.

Table 2: Some Cryptographic features of VIPSec protocol [35]

- VIPSec does not rely on a public key infrastructure (PKI).
- VIPSec uses one-time keys instead of using permanent keys.
- It uses one time signature commitment and per session random numbers as the exchanged object. Alternatively, it can also use one time biometric user data such as photos, voice recordings, videos, etc.
- VIPSec can detect a man-in-the-middle attack by having the users verbally compare the received objects.
- It has perfect forward secrecy; as the keys are destroyed at the end of every call.
- It does not rely on SIP signalling for key management. It performs the key management and key agreement in a purely peer-to-peer fashion.
- VIPSec provides two verification levels: medium (voice verification) and hard (video verification).

Chapter 4: Key Escrow Agent

This chapter starts by describing the design and implementation of a key escrow agent as a TTP and how to escrow a session key with the escrow agent. Security parameters required to escrow with the session master key to generate the session keys are presented in section 4.2. Section 4.3 describes the mechanism to escrow the session master key and other necessary parameters from a user agent along with the necessary implementation details. Finally, section 4.4 discusses the time and place to escrow the key with the escrow agent.

4.1 Escrow agent and escrow database

A very simple escrow agent has been implemented using the Apache web server with MySQL database support. The primary task of the escrow agent is to receive the key from an authenticated user and after proper validation of the received data to store the escrowed data in a secure database. Figure 4-1 shows the general architecture of our escrow agent. 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. To enable the SSL capability of the Apache web server a script has been used that automates the complete process (see Appendix A).

The escrow server listens on TCP port 443[§]. Upon reception of a request from a client the server first authenticates the user using the key value pair appended to the URL. The escrow agent uses an authentication table that stores the list of all valid users who can escrow keys. In a commercial escrow agent this would be a list of subscribers. When the authentication is successful it inserts the data to be escrowed (passed as the third key value pair) into the escrow database. Before inserting the value into the database a validation check is performed so that no SQL injection attack can be performed.

[§] This is the TCP port number assigned by the Internet Assigned Numbers Authority for HTTPS traffic. See http://www.iana.org/assignments/port-numbers - last updated 2009.12.08.

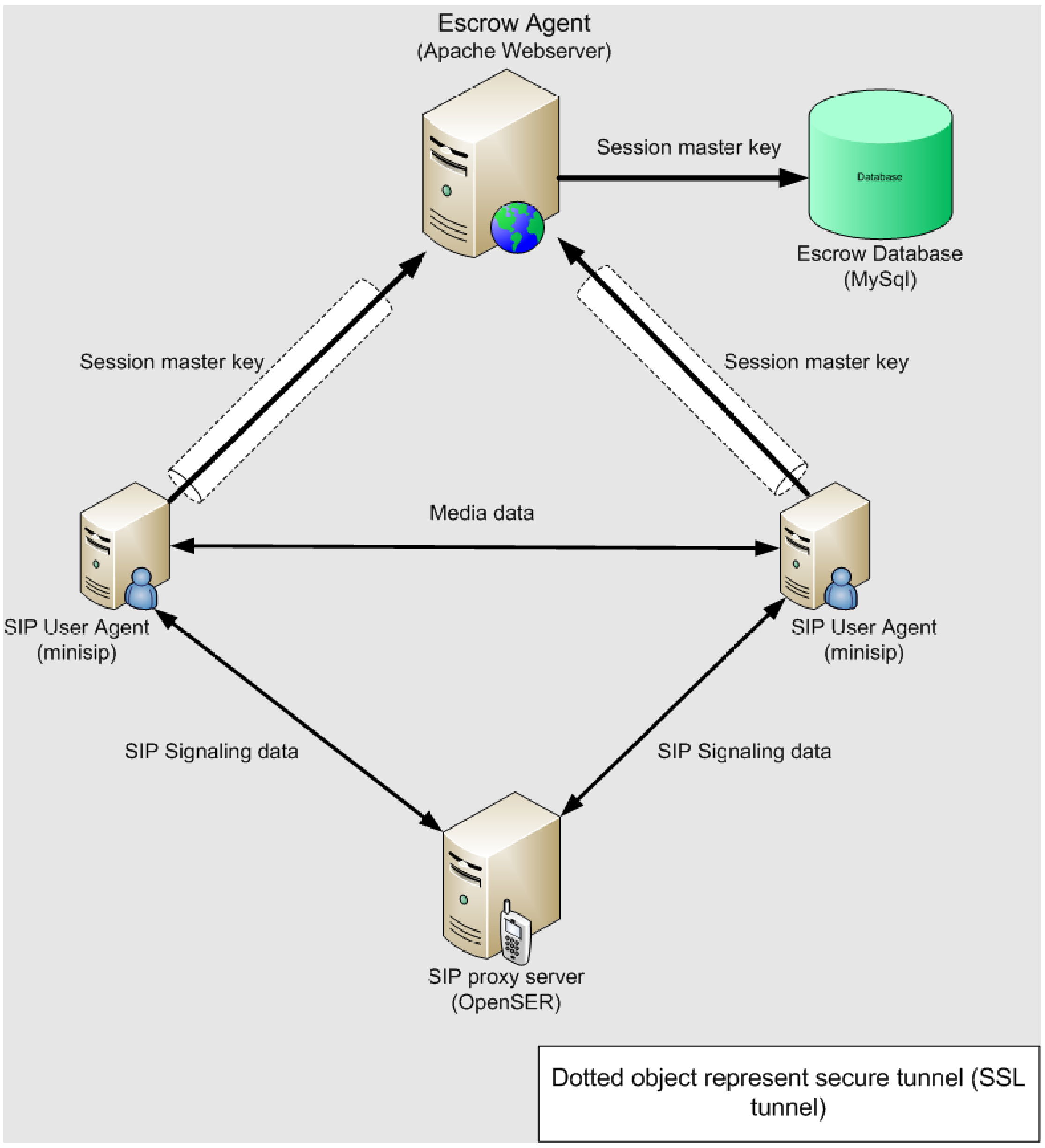


Figure 4-1: General architecture of the Escrow agent

4.1.1 Escrow database

A MySQL database has been utilized to store the session master keys that have been escrowed by SIP user agents. The database consists of two tables: one for authentication data and the other for the escrowed data. Figure 4-2 shows the general structure of the escrow database. The authentication table stores the username and password of the valid users who can escrow data with our escrow agent. In this implementation the SIP URI has been used 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. How this password is communicated to the user and their user agent is outside the scope of this thesis project. We simply assume that there is some secure off-line method of doing this.

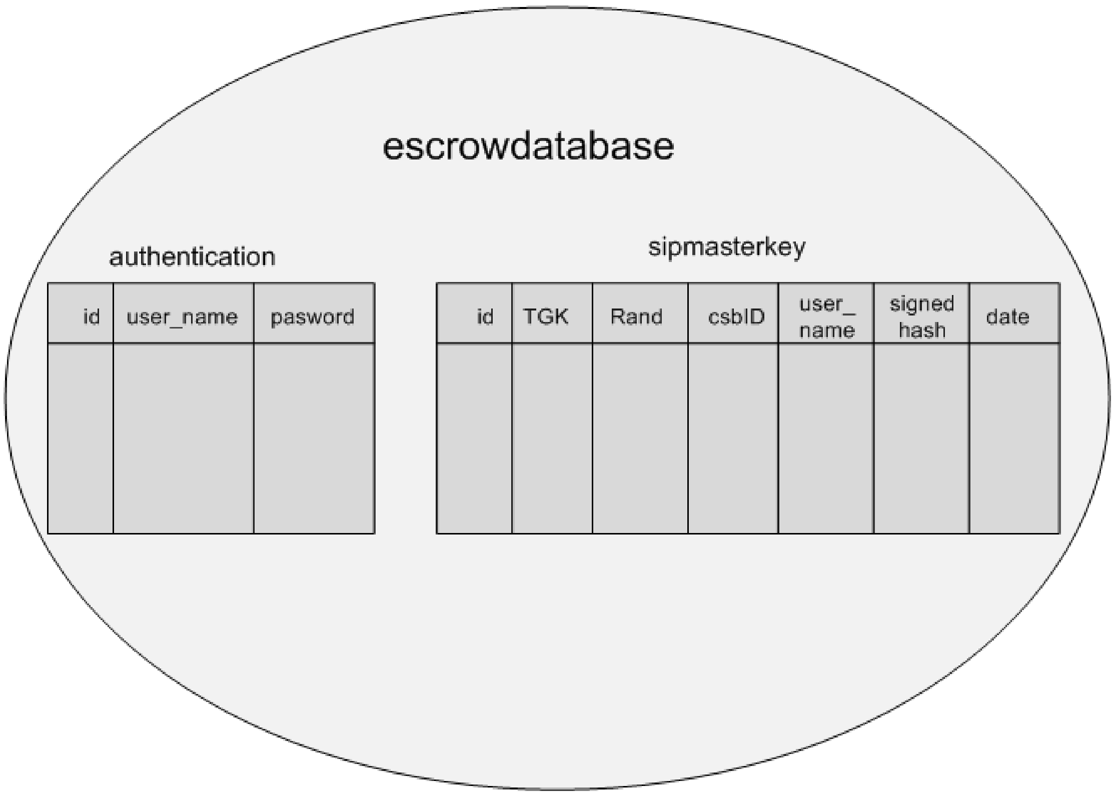


Figure 4-2: General Structure of the escrow database

The sipmaseterkey table stores the actual key along with other parameters that are needed (or useful) if the escrowed information is to be used to recover the contents of a SRTP or SRTCP stream. What parameters are necessary to escrow along with the TGK is presented in the next section. 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.

4.1.2 Implementation details

A small PHP script has been written to automate the whole process. Listing 1 shows this PHP script.

```
<?php
      include("config.php");
      $user = trim($_GET['user']);
      $pass = trim($_GET['password']);
      $data = $_GET['data'];
//Preventing SQL injection
      $user = mysql_real_escape_string($user);
      $pass = mysql_real_escape_string($pass);
      $data = mysql_real_escape_string($data);


//parse the data
     $pieces = explode("%", $data);

      $login = 0;
      if($user == "" || $pass == "" || $data == ""){
        print("malformed URL");
      }
      else{
//authenticating valid user of the system from authentication table
          $result = mysql_query("SELECT * FROM authentication where
user_name = '$user' and password  = '$pass'") or die(mysql_error());
          while($row = mysql_fetch_array($result)){
                $login = 1;
          }

// If authentication successful then insert the value to the database
          if($login == 1)
          {
mysql_query("insert into
sipmasterkey(`key`,`rand`,`signedhash`,`csbID`,`date`,`userid`)
values ('".$pieces[0]."','".$pieces[1]."','".
        $pieces[2]."','".$pieces[3]."',now(),'".$user."')") or
die(mysql_error());

          }
          else
          {
                print("<b>User id Or Password not matched</b>");

          }
      }
?>
```

Listing 1: PHP script to automate the escrow agent functionality

4.2 What to escrow?

We escrow the session master key, i.e., 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. Table 3 shows the necessary parameters that are required to generate session keys from the TGK. From the table we can see that we are escrowing the TGK along with the pseudo-random number (Rand) and csbID value. For details about these security parameters see section 2.6 and section 2.8. Interested readers are encouraged to see [25] and [36]. All these parameters are necessary for the LEA to (re-)generate the session keys. Additionally we are escrowing our final signed hash value as a marker that indicates the end of a session.

Table 3: Security parameters necessary along with TGK to generate session keys

Parameter	Need to escrow??	Remarks
Rand	**Yes**	This value is exchanged at the time of key agreement inside a MIKEY message which in turn is inside a S/MIME encoded SIP INVITE message. Unless this value is stored with the escrow agent the LEA would not have access to this value.
csbID	**Yes**	Same as above
SSRC	**No**	SSRC is in clear text in the SRTP header, hence there is no reason to escrow this value. The LEA can recover this value from captured SRTP packets
csID	**No**	For the Initiator, the sender csID value is always 1 and receiver csID is always 2; while for Responder the csID value is the reverse. The LEA can determine the csID value from the captured conversation by checking whether the subject is the initiator or responder. Alternatively the csID value can be generated from the SSRC.
ROC	**No**	The ROC can be calculated from the packet sequence number.
Policy no	**No**	For the current implementation of minsip the policy number is always 0; hence there is no reason to store it for this implementation, but this parameter might need to be stored in the future.

4.3 How to escrow?

To escrow the session master key we are using secure HTTP (HTTPS) and 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. We are using HTTPS 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 use libcurl[17], as described previously in section 2.4. We have written our own function to escrow the session master key using several functions from the libcurl library.

4.3.1 Necessary modifications to the minisip code

To successfully escrow the session master key we have modified two files (Mikey.cxx and Mikey.h) in the libmikey library of the minisip source code. In the Mikey.cxx file we added a functioned named *escrowSessionKey()* as a public member of the Mikey class. Listing 2 shows our *escrowSessionKey()* function. In this function we first form the URL that will be used by one of the libcurl fuctions to instantiate a curl object. While forming the URL of the escrow agent we have used the base_64 encoding of TGK, Rand, and csbID.

```
void Mikey::escrowSessionKey(unsigned char * signedHash, int
signedHashLength){
    static char errorBuffer[CURL_ERROR_SIZE];
    // holds the base64 value of the required parameters to escrow
    std::string tgk_b_64_ecoded;
    std::string rand_b_64_ecoded;
    std::string signedHash_b_64_ecoded;

     static string buffer;
     char * cstr;
      tgk_b_64_ecoded = base64_encode(ka->tgk(),ka->tgkLength());
      rand_b_64_ecoded = base64_encode(ka->rand(),ka->randLength());
      signedHash_b_64_ecoded =
base64_encode(signedHash,signedHashLength);
      const char *csbId = itoa((int)ka->csbId()).c_str();

     string url1("https://localhost/~saki23/escrow_agent/?user=");

     cstr=new char
[url1.length()+1+tgk_b_64_ecoded.length()+rand_b_64_ecoded.length()+s
ignedHash_b_64_ecoded.length()+2*ka->uri().length()+100 ];
     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,tgk_b_64_ecoded.c_str());
     strcat(cstr,"%");
     strcat(cstr,rand_b_64_ecoded.c_str());
     strcat(cstr,"%");
     strcat(cstr,signedHash_b_64_ecoded.c_str());
     strcat(cstr,"%");
     strcat(cstr,csbId);

// Our curl objects
     CURL *curl;
     CURLcode res;

  curl = curl_easy_init();
  if(curl) {
     curl_easy_setopt(curl, CURLOPT_URL, cstr);

#ifdef SKIP_PEER_VERIFICATION
    curl_easy_setopt(curl, CURLOPT_SSL_VERIFYPEER, 0L);
#endif
    res = curl_easy_perform(curl);
    /* always cleanup */
    curl_easy_cleanup(curl);
    if (res == CURLE_OK)
               cout<< "successfull";
    else
         cout << "Error: [" << res << "] - " << errorBuffer;
  }
     delete [] cstr;
}
```

Listing 2: escrowSessionKey() inside Mikey.cxx to escrow the session master key where the top gray coloured area shows the formation of the URL of the escrow agent with the TGK and other necessary parameters and the lower blue coloured area shows the invocation of libcurl method.

To support HTTPS with a self signed certificate libcurl provides the `SKIP_PEER_VERIFICATION` macro definition. If we want to connect to a site using a certificate that is **not** signed by one of the certificate authorities in the CA bundle we have, we can skip the verification of the server's certificate by defining this macro. Although it makes the connection less secure, we are using this approach as our escrow agent is using a self signed certificate. (Note that when the users receive their password to use with the escrow agent the user could insert the escrow agent's certificate into the configuration of their user agent.)

In the Mikey.h file we have added some header files to support the libcurl functions that we use to escrow the session master key. Listing 3 shows the necessary modification to the Mikey.h file.

```
//Defined to skip the peer verification of self signed certificate
#ifndef SKIP_PEER_VERIFICATION
#define SKIP_PEER_VERIFICATION
/*
.
Existing header files
.
*/
#include "curl/curl.h"
#include "curl/easy.h"
#include "curl/types.h"
//Added to take base_64 value
#include<libmcrypto/base64.h>

class LIBMIKEY_API IMikeyConfig: public virtual MObject{

};
class LIBMIKEY_API Mikey: public MObject{
      public:
             /*
              .
              Existing public members
              .
              */
       //definition of escrowSessionKey method
       void escrowSessionKey(unsigned char * , int);

      protected:
             /*
              .
              Existing protected members
              .
              */
      private:
              /*
              .
              Existing private members
              .
              */
};
#endif
```

Listing 3: Partial listing of modified Mikey.h file

4.4 When and from where to escrow?

It is a very important to decide when is the most appropriate time to escrow the session key. Since the escrow agent will learn the same session key as the sender and receiver, there is a potential for interception & decryption and/or forgery of the content of the media streams of the Secure Real Time Control Protocol (SRTCP) packets. To prevent real-time interception and decryption of a media stream or its associated control stream we only deposit the session key *at the end* of a successful session. Here we need to mention that either initiator or receiver may end a successful session. Figure 4-3 and Figure 4-4 show the possible ending of a successful session, along with the correct time to escrow the session master key. (Note that in this figure we have shown only the time when the Initiator performs the escrow operation, the Responder must also perform its own escrow operation when it ends the session. In our implementation and testing we have used the same escrow agent for both parties, but there is **no** requirement for this.)

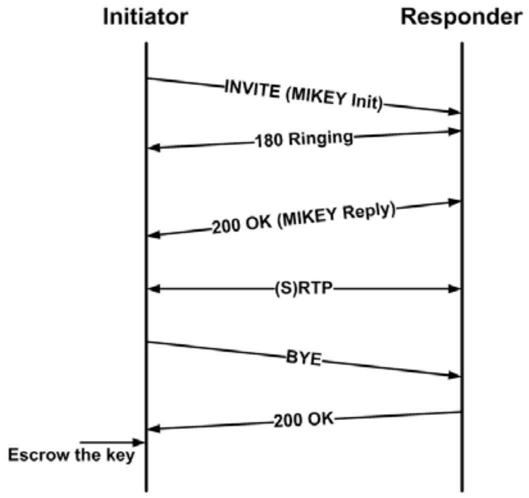


Figure 4-3: Initiator ending the session

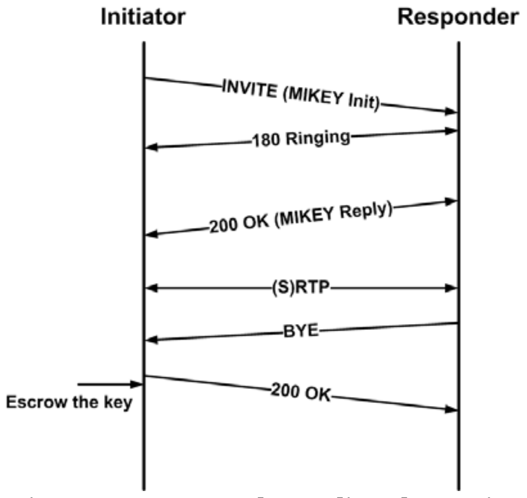


Figure 4-4: Responder ending the session

It is important to note here that 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. Thus if a LEA learns the master key used for this session, it can both decrypt the encrypted content and authenticate the packets. However, given the master key it is also possible to forge SRTP content that would be valid. As noted previously in section 2.6.3, if this forged SRTP packet were to arrive at the receiver before the SRTP packet that the sender sent, then this forged packet would be accepted by the receiver and the actual sender’s packet would be rejected as a replayed packet! This is why it is important that the master key **not** be escrowed before the session has ended.

A special case considered in the design is the *abnormal* termination of a user agent in the middle of an active session. If an ongoing call is terminated due to an abnormal termination of the user agent (For example by closing the soft phone window by clicking the ‘close’ button or by pressing Alt+F4.), then the key associated with the current call(s) must be escrowed with the escrow agent.

4.4.1 Necessary modifications to the minisip code

As mentioned in the previous section, we need to escrow our session key at the end of a successful session, thus we set a flag when a session successful starts. At the end of a session we check the flag and invoke the *escrowSessionKey( )* method to escrow our session master key along with other security parameters. When a successful session starts the *start( )* method of the Session class is invoked. A variable named escrowFlag is set inside the *start( )* method of the Session class and this flag's value will subsequently be checked inside the *stop( )* method of the Session class. Listing 4 shows the invocation of our *escrowSessionKey( )* method. Actually we increment the escrowFlag inside the *start( )* method. This flag is initialized inside the constructor of the Session class with the value zero (0) and decremented inside the *stop()* method. If the escrowFlag is zero (0), then the *escrowSessionKey()* method is invoked. Instead of defining the escrowFlag as a Boolean type we declare it as an integer. The reason for this design decision is that there may be multiple media sessions (audio and video streams) going on at time, in such a situation we escrow the key when all sessions have ended. Since every media session will invoke the *start( )* method; each invocation will increment the escrowFlag and the end of each session will invoke the *stop ()* method resulting in the escrowFlag being decremented. When the last session ends the escrowflag will have the value zero (0), hence it is time to escrow the session key.

```
//decrement the escrowFlag
--escrowFlag;

//check the escrowFlag whether to escrow the key or not

      if(escrowFlag == 0){


       getMikey()->escrowSessionKey(signature,signatureLength);


      }
```

Listing 4: Code snipped from Session::stop() of libminisip library showing the invocation of escrowSessionKey() method after checking the escroFlag

The default signal handler of the minisip code is designed such that for any kind of interruption during an ongoing call it sends a formal SIP 'BYE' message to its peer before terminating. In the course of doing this minisip invokes the Session::*stop( )* function. This is another advantage for checking the escrowFlag inside the Session::*stop( )*, as it provides the escrow functionality even in case of abnormal termination of the user agent during on-going session.

Chapter 5: Design and Implementation of a Solution

This chapter presents the cryptographic operations performed to implement the proposed system in order to support key escrow *while enabling the users to detect forgery of call content* by a LEA and/or escrow agent. A design overview is presented in section 5.1. The sections following this overview present the cryptographic operations that need to be implemented to realize the proposed model; along with the relevant implementation details. How the proposed model can detect forgery is presented in section 5.6.

5.1 Design overview

In the previous chapter we described the escrow agent and how session keys are escrowed by the user agent after the end of a session. The main concern regarding key escrow systems is the trustworthiness of the TTP. Since the session keys are stored at the escrow agent an unethical employee of the TTP could misuse this information. Therefore, some mechanism is required that could make key escrow feasible and desirable - while preventing tampering with the communication session’s content. To prevent the undetectable fabrication of session content we compute a series of digital signatures of the media stream and send them along with the media stream (by sending them in the associated SRTCP stream). Figure 5-1 shows the overall cryptographic operations performed to facilitate detection of forgery of call content.

As a first step we create a block of SRTP packets. Determination of the appropriate block size is one of the open questions we seek to answer. The reason behind creating blocks is that we want to compute the digital signature over a collection of packets to reduce the overhead of these expensive cryptographic operations. After creating the block we calculate a hash of the block. This hashing operation produces a *fixed length* output. Finally we perform the signing operation on the hash value, producing a fixed length signed hash (i.e., a digital signature of the hash). To compute the digital signature we are use the private key of the user. Note that the private key of the user is **not** escrowed – nor does any party other than the user need to know this private key. While we escrow the session keys used to encrypt the media stream, thus the TTP or a LEA who has access to the escrowed session key can forge media content, they cannot compute the correct signed hash, as this requires the user’s private key. Anyone with access to the recorded conversation, the session key, and the user’s public key can readily detect fabrication of call contents, by verify whether the hashes computed over the media stream have the correct digital signature or not.

Because we use asymmetric cryptography to perform the digital signature and because asymmetric cryptography is much more computationally expensive than symmetric cryptography we need to examine (1) if it is feasible to compute such signatures during the session and (2) what is a suitable block size (to balance computational and communication overhead with the desired granularity of forgery detection and the available resources).

SRTP and SRTCP both make use of symmetric encryption in order to support low delay and high throughput for the media streams. However, there is no need for the signed hash values to be delivered with low delay – since they are only (potentially) relevant after the session has ended. It is the combination of signing the hash of a block of SRTP packet at the same time and lack of any requirement for low delay that enables asymmetric public key techniques to be used for signing these hashes.

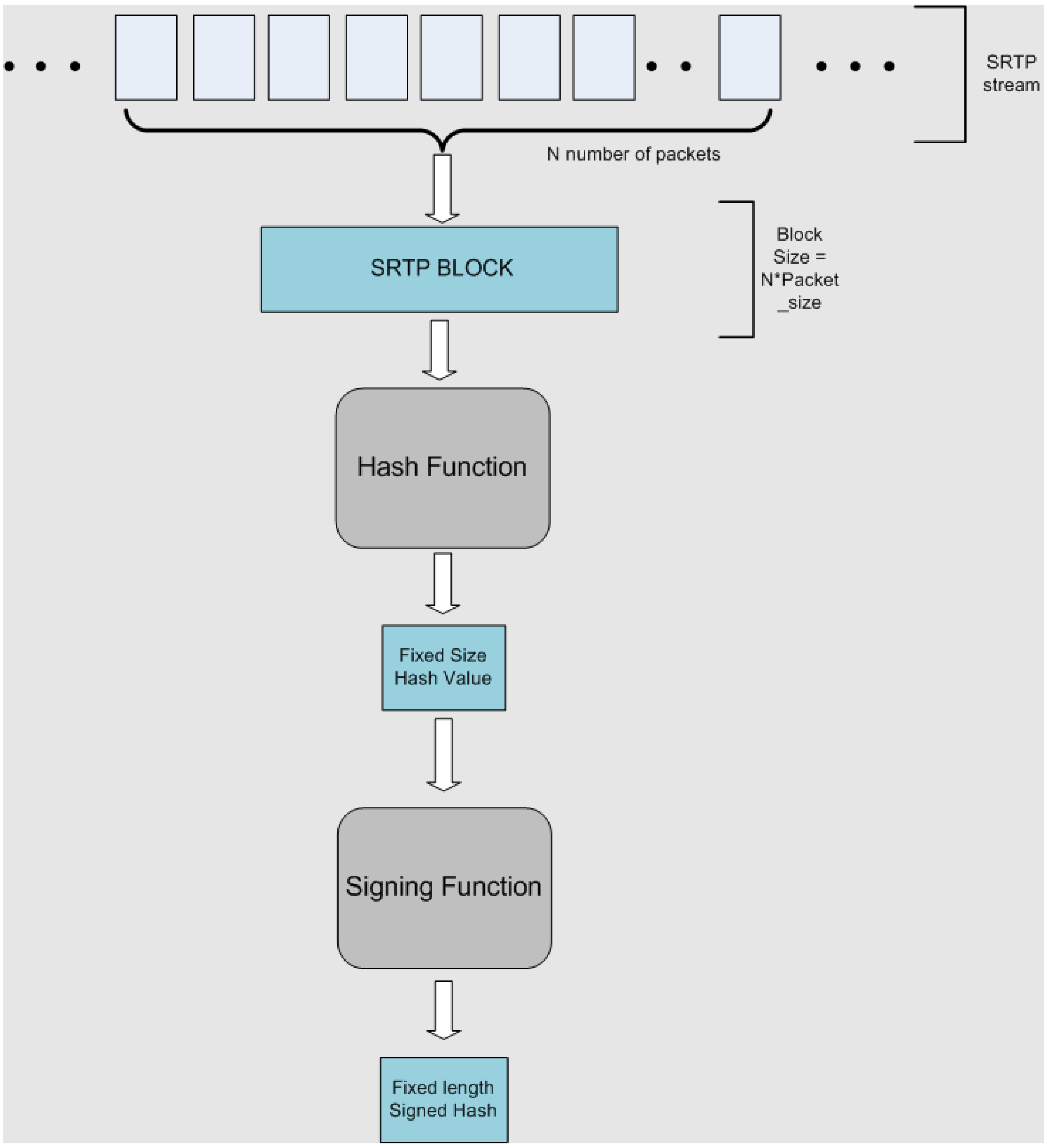


Figure 5-1: Cryptographic overview of the proposed model

5.2 Creating SRTP blocks

We create the SRTP blocks in real-time while sending the SRTP packets. We create a block based upon a pre-defined block size. After a successful session establishment when the media stream starts we collect RTP packets and place them in a buffer. When the number of packets in the buffer exceeds the block size (BLOCK_SIZE), then we perform the necessary cryptographic operations over this block. Figure 5-2 shows a flowchart of these operations.

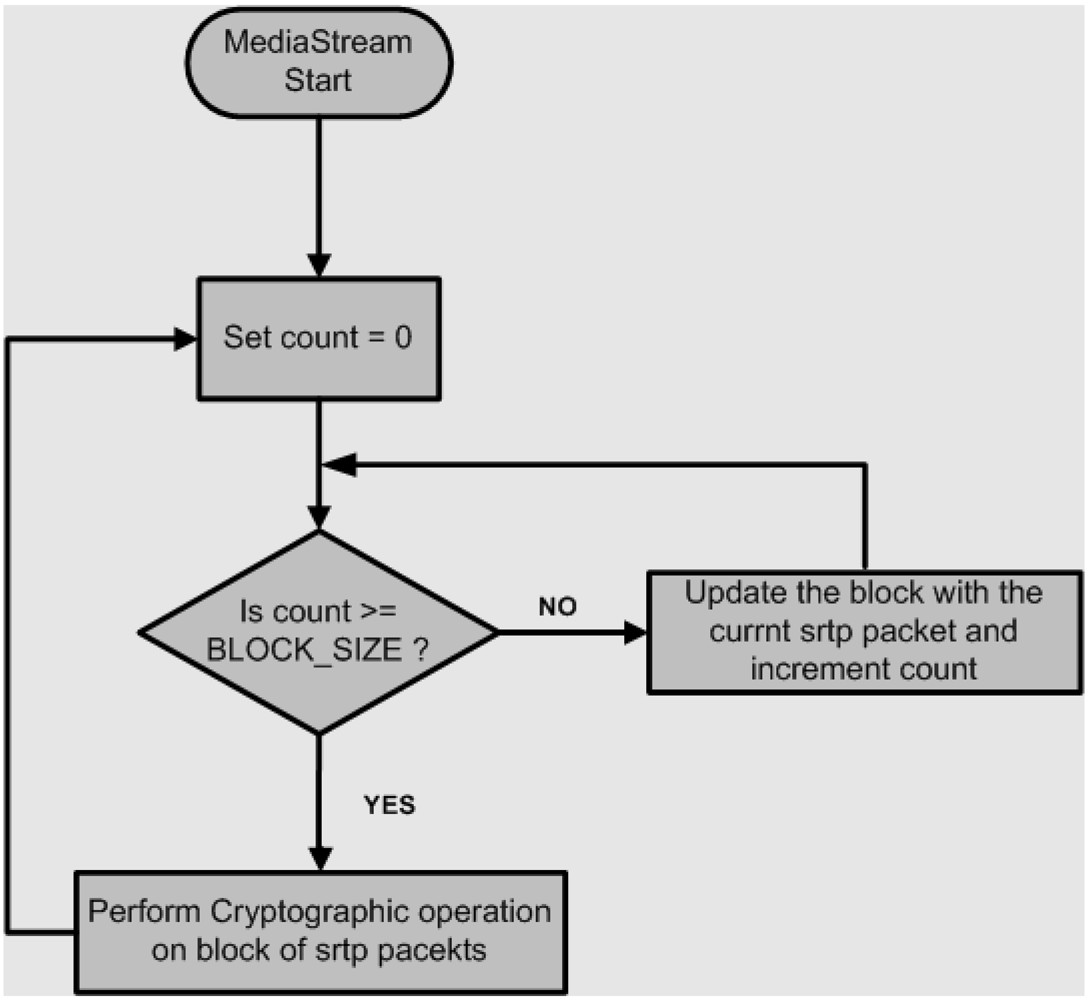

MediaStream Start
Set count = 0
Is count >= BLOCK_SIZE ?
NO
Update the block with the currnt srtp packet and increment count
YES
Perform Cryptographic operation on block of srtp pacekts

Figure 5-2: Flowchart of SRTP block creation

A special case occurs when the session ends. For the last block the number of packets can range between 1 and the BLOCK_SIZE. We process the buffer (which may not be full) and send the final signed hash through the SRTCP/RTCP path. When a LEA captures a conversation it should always capture the associated RTCP packets, especially the RTCP packet containing the last signed hash value as verifying this signed hash can be used to show that there is no additional media content sent following the last SRTP/RTP packet.

The most appropriate block size is a design choice. Increasing the number of packets that are processed together in a block will reduce the overhead (in terms of additional traffic that needs to be sent and the computation resources used). However, larger blocks will increase the delay between the content and the hash over this content and will also increase the granularity of any detected forgery. In the next chapter, we will discuss the performance for different block size when we evaluate our proposed model.

5.2.1 Necessary modifications to the minisip code

The libminisp library of the minisip code handles the media streams. Inside the MediaStream.cxx file the RealtimeMediaStreamSender class is mainly responsible for sending SRTP packets. To create the SRTP block we have added a function *updateBlock( )* as a member function of RealtimeMediaStreamSender class. Listing 5 shows the *updateBlock( )* function that takes SRTP packet data and packet size as parameters and adds the data to the current buffer (i.e., the current block). This *updateBlock( )* function is called from the *send( )* function of the RealtimeMediaStreamSender class.

```
void RealtimeMediaStreamSender::updateBlock(char *packetData, unsigned int
pSize){
   memcpy(& rawhashdata[blockSize], packetData, pSize);
   blockSize += pSize;
}
```

Listing 5: updateBlock () function to incrementally update SRTP block

The yellow coloured area in Listing 6 shows the invocation of our *updateBlock ()* function.

```
if (count == 0)
         rawhashdata = new char [BLOCK_SIZE*packet->size()];

                                                                        (a)
updateBlock(packet->getBytes(),packet->size());

count++;                                                                (b)
      if (count >= BLOCK_SIZE) {

                                                                        (c)
     signature = hashAndSignTheBlock();

                                                                        (d)
     sendSignedHash(&signature);

         // reset the counters
         count = 0;
         blockSize = 0;

         delete [] signature;
         delete [] rawhashdata;
             }
```

Listing 6: Code snippet from RealtimeMediaStreamSender::send () to deal with (a) creating SRTP block (yellow coloured), (b) checking it it time to send the signed hash, (c) hashing and signing the block (blue coloured), and (d) sending the Signed hash (orange coloured).

5.3 Hashing SRTP blocks

Instead of taking the digital signature of the raw SRTP block we first compute the hash of the SRTP block, then sign – thus reducing the amount of data that has to be processed using the asymmetric key. We use HMAC_SHA [37] as the hashing algorithm to calculate this hash. The HMAC_SHA algorithm takes as input an authentication key along with a variable size array of data and produces a *fixed* size hash. Figure 5-3 shows the block diagram of HMAC_SHA hash function. The authentication key we are using to calculate the hash value is generated from the session master key (TGK) that we are escrowing with the escrow agent. Figure 5-4 shows the block diagram of authentication key generation for HMAC_SHA. When the LEA wishes to verify the signed hash it can generate the same authentication key from the TGK to calculate the hash of a recorded session.

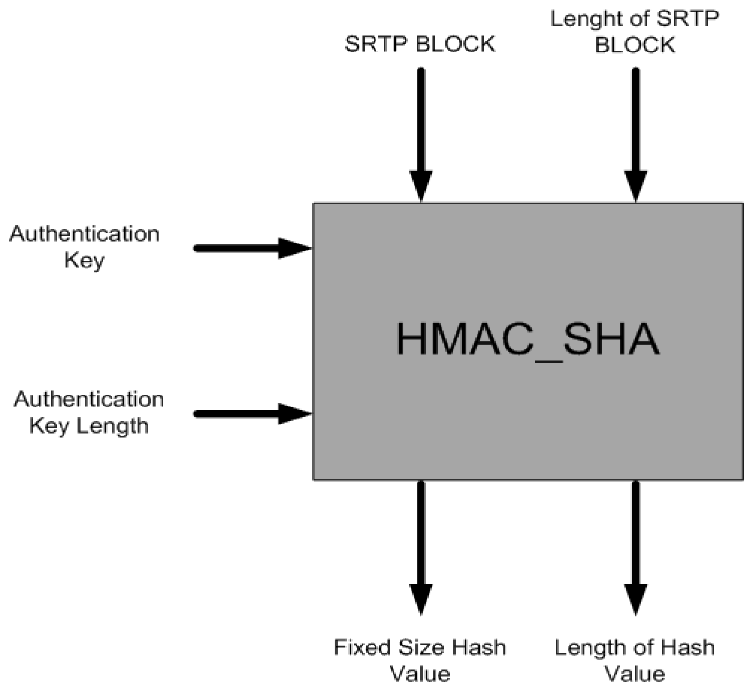


Figure 5-3: Block diagram of HMAC_SHA hash function

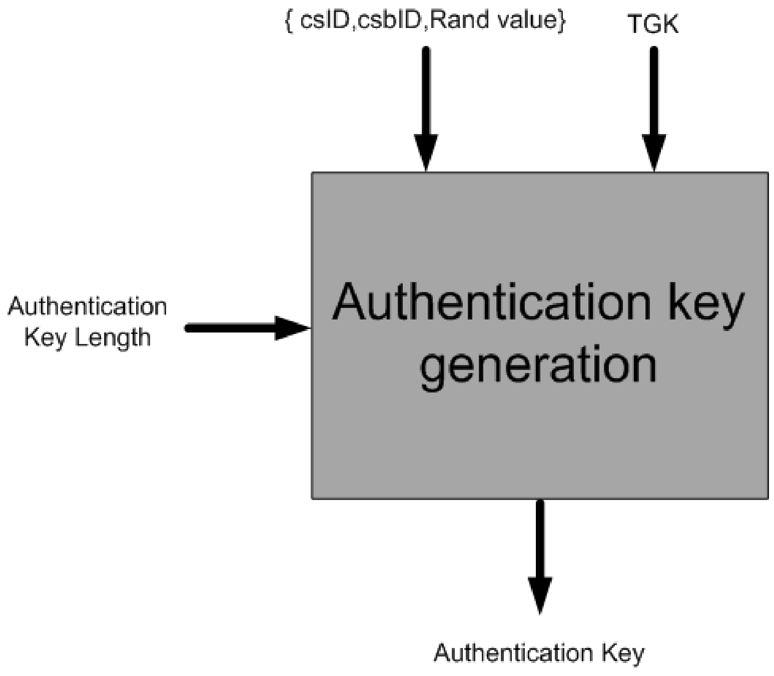


Figure 5-4: Block diagram of authentication key generation for HMAC_SHA

5.3.1 Necessary modifications to the minisip code

To implement the hashing operation we have added a function named *hashAndSignTheBlock( )* as a member function of the RealtimeMediaStreamSender class inside the MediaStream.cxx file of the minisip libminisip library. Inside this function we invoke the *hmac_sha1( )* function from the libmcrypto library. This *hamc_sha1( )* function in turn invokes the OpenSSL [38] library function that performs the actual hash operation. Listing 7 shows our *hashAndSignTheBlock( )* function where blue coloured area shows the invocation of the *hmac_sha1( )* function.

```
unsigned char* RealtimeMediaStreamSender::hashAndSignTheBlock() {

        unsigned char *signature = new unsigned char[200];
        unsigned char* hashValue = new  unsigned char [20];
        unsigned int hashLength;

// perform hash                                                            (a)
        hmac_sha1( hmacAuthKey, 32,
                (unsigned char *)rawhashdata,   /*data*/
                blockSize,                   /* data length*/
                hashValue,                     /*tag*/
                &hashLength );

// perform signature                                                       (b)
        if (!ka->getSim()->getSignature(hashValue,
                (int)hashLength, signature, signatureLength, true)) {
            cout << "\n ERROR : Could not perform digital signature of the
message";
        }
        else {
            cout << " INFO : Signature Successful ";
        }
delete [] hashValue;

   return signature;
}
```

Listing 7: hashAndSignTheBlock () function to perform (a) the hash (blue coloured area) and (b) the signature (orange coloured are) of SRTP block.

The first argument of the *hmac_sha1( )* function is the authentication key that we generate from the TGK using the *genAuth( )* function of the KeyAgreement class. Listing 8 shows the generation of the authentication key for hmac_sha1 inside the RealtimeMediaStream::*initCrypto( )* function .The blue coloured area in Listing 7 (a) shows the invocation of our *hashAndSignTheBlock( )* function.

```
uint8_t  csId = ka->getSrtpCsId( ssrc );
ka->genAuth(csId, hmacAuthKey, AUTH_KEY_SIZE);
```

Listing 8: Code snipped from RealtimeMediaStream::initCrypto () showing the generation of the authentication key for use by the hmac_sha1 function.

5.4 Signing the hashed blocks

After computing the hash of the SRTP block we digitally signing the hash value producing a fixed length signed hash. For signing we use the RSA [39] algorithm. The signing operation uses the private key of the sender; hence the corresponding public key can be used to verify the signature. Figure 5-5 shows the block diagram of signing operation.

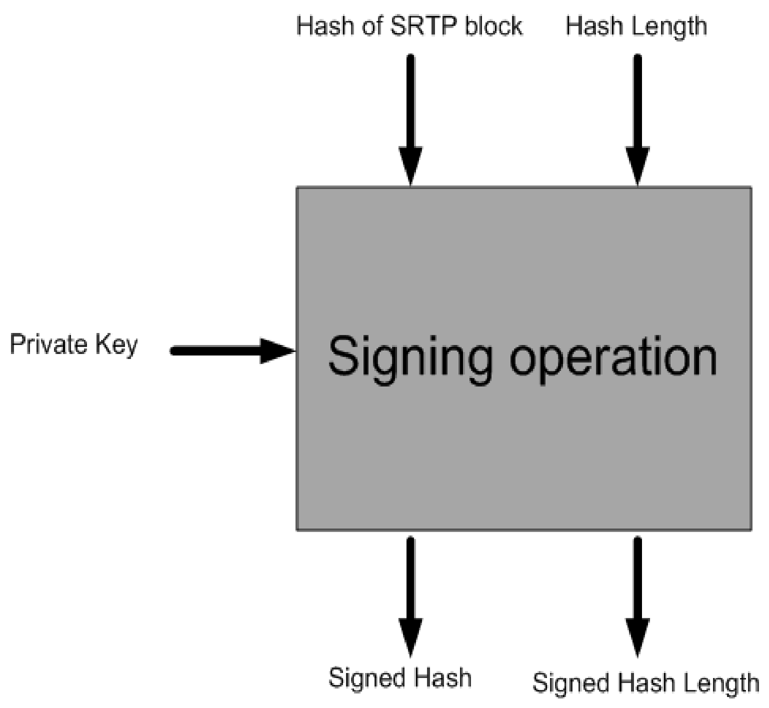


Figure 5-5: Block diagram of signing operation

5.4.1 Necessary modifications to the minisip code

To implement the signing operation we are using the *getSignature( )* function of the SipSim class from the libmcrypto library. The orange coloured area in Listing 7(b) shows the invocation of the *getSignature( )* function to compute the digital signature of the fixed size hash value of SRTP block. The *getSignature( )* function is a virtual function defined in the SipSim class. Figure 5-6 shows the UML of the call to the OpenSSL library function to compute a digital signature. The implementation of *getSignature( )* inside the SipSimSoft class is actually invoked by our *hashAndSignTheBlock( )* function which in turn calls the *signData( )* function of Certificate and OsslPrivateKey class. The OpenSSL library function is finally called from the *signData( )* function of the OsslPrivateKey class to compute the digital signature and produce the fixed length signed hash.

5.5 Sending the signed hash

The signed hashes are sent via the SRTCP/RTCP path after they are calculated. When the LEA conducts a LI, they need to capture both the SRTP/RTP and SRTCP/RTCP packets sent by the subjects. As we transmit the signed hashes in the RTCP stream, then too will be captured.

Figure 5-7 depicts the logical view of sending these signed hashes. The frequency of sending a signed hash depends on the SRTP block size. As the signed hash value is potentially relevant only when the session is over, it does not need to be sent with the low delay requirement of the SRTP packets. In our implementation we send the signed hashes as soon as they are calculated, but they could be grouped together and could be sent in a single SRTCP/RTCP packet which might reduce the amount of overhead (this will be discussed in next chapter).

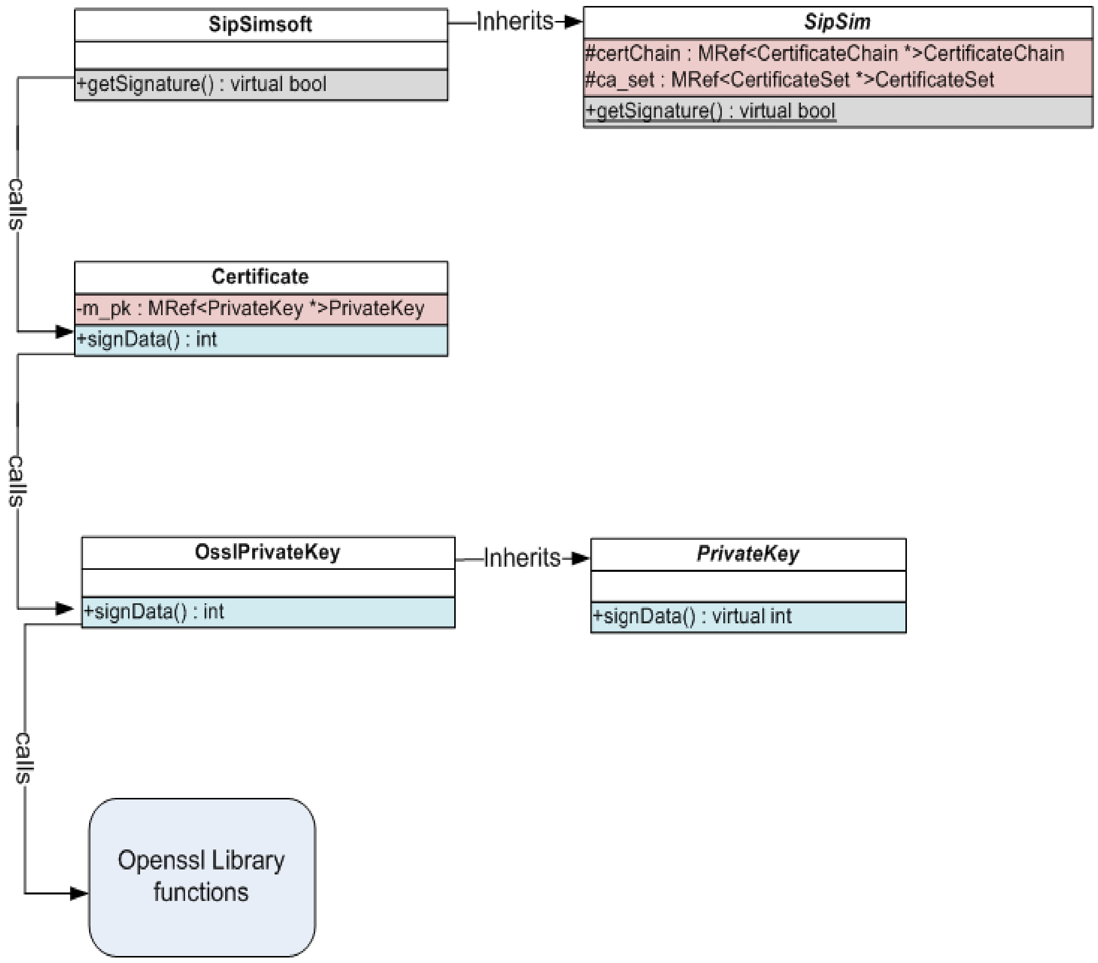

SipSimsoft
+getSignature() : virtual bool
Inherits
SipSim
#certChain : MRef<CertificateChain *>CertificateChain
#ca_set : MRef<CertificateSet *>CertificateSet
+getSignature() : virtual bool
calls
Certificate
-m_pk : MRef<PrivateKey *>PrivateKey
+signData() : int
calls
OsslPrivateKey
+signData() : int
Inherits
PrivateKey
+signData() : virtual int
calls
Openssl Library functions

Figure 5-6: UML diagram showing the invocation of OpenSSL library functions

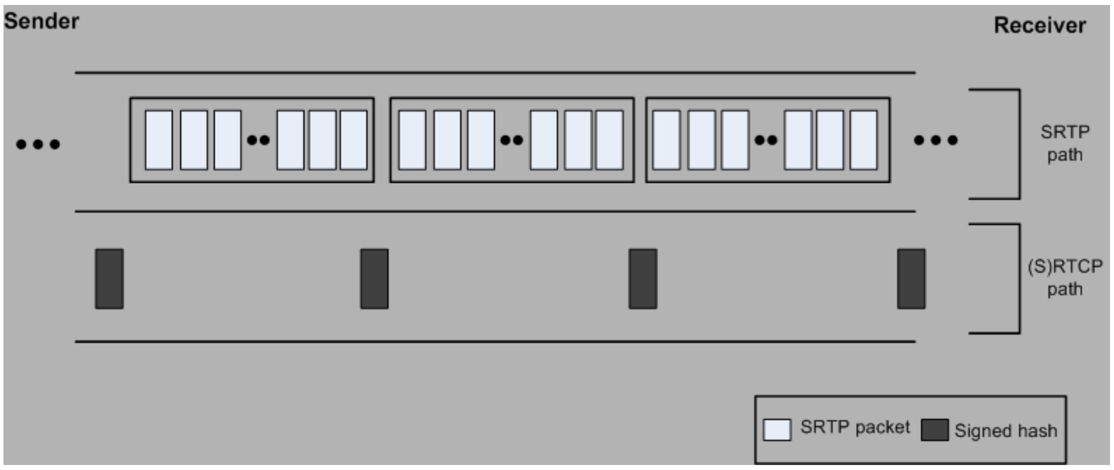


Figure 5-7: Sending of Signed hash via SRTCP/RTCP path

5.5.1 Necessary modifications to the minisip code

To send the signed hash in SRTCP/RTCP path we have added a simple function named *sendSignedHash( )* as a member function of the RealTimeMediaStreamSender class inside the MediaStream.cxx file of libminisip library. Listing 9 shows the *sendSignedHash( )* function. We send the signed hash sequence number along with the signed hash value. The orange coloured area of Listing 7(b) shows the invocation of our *sendSignedHash( )* function from the *Send( )* function of the RealTimeMediaStreamSender class.

```
/**
 * Send the signed hash in rtcp path at remote port +1
 * First 128 bytes is the signed hash!  4 bytes signed Hash sequence Number
 */
void RealtimeMediaStreamSender::sendSignedHash(unsigned char **signature) {

        // convert the seq number to a string
        const char *seq = itoa(signedHashSeqNum++).c_str();

            unsigned char *custom_packet = new unsigned
char[signatureLength+sizeof(seq)/sizeof(char)] ;



        memcpy(custom_packet, *signature,signatureLength);
        memcpy(&custom_packet[signatureLength],seq,sizeof(seq)/sizeof(char)-1);

        rtcp_sock->sendTo(**remoteAddress,getPort()+1,custom_packet,
signatureLength+sizeof(seq)/sizeof(char));

                delete []custom_packet;
}
```

Listing 9: sendSignedHash( ) function sends the signed hash via the SRTCP/RTCP path

5.6 Detection of forgery by the proposed model

In our proposed model we escrow the session key with the escrow agent, who acts as a trusted third party (TTP). However, due to the potential for insider misuse we are using asymmetric cryptographic operation to detect attempts at forgery of contents. In the previous sections we have discussed how we computed the digital signature (a signed hash) of the SRTP blocks and send these signed hashes along with the control traffic of the associated media stream. In this section we describe how these signed hash values of SRTP blocks provide authenticity for recordings of secure sessions and enable anyone to detect fabrication of media content.

As we are sending the signed hash value along with the control traffic of the associated media stream anyone who has the public key of the sender can verify whether the received session is actually what the sender has sent. This verification is shown in Figure 5-8. In this process the verifier (in this case shown as the LEA) decrypts the signed hash using the public key of the sender to produce the hash of SRTP blocks as calculated by the sender. Next the verifier processes the captured session and does the hashing operations (in our case HMAC_SHA) to produce the hash of the captured session. If these two hashes are identical, then the captured packets have not been changed. Thus the digitally signed hash provides integrity protection – using the user's public/private key pair.

Note that this integrity protection is in addition to the integrity protection provided to the SRTP and SRTCP traffic, but uses a key that has **not** been disclosed to the escrow agent (and by implication not disclosed to the LEA). This prevents either the escrow agent or LEA from being able to successfully forge captured contents.

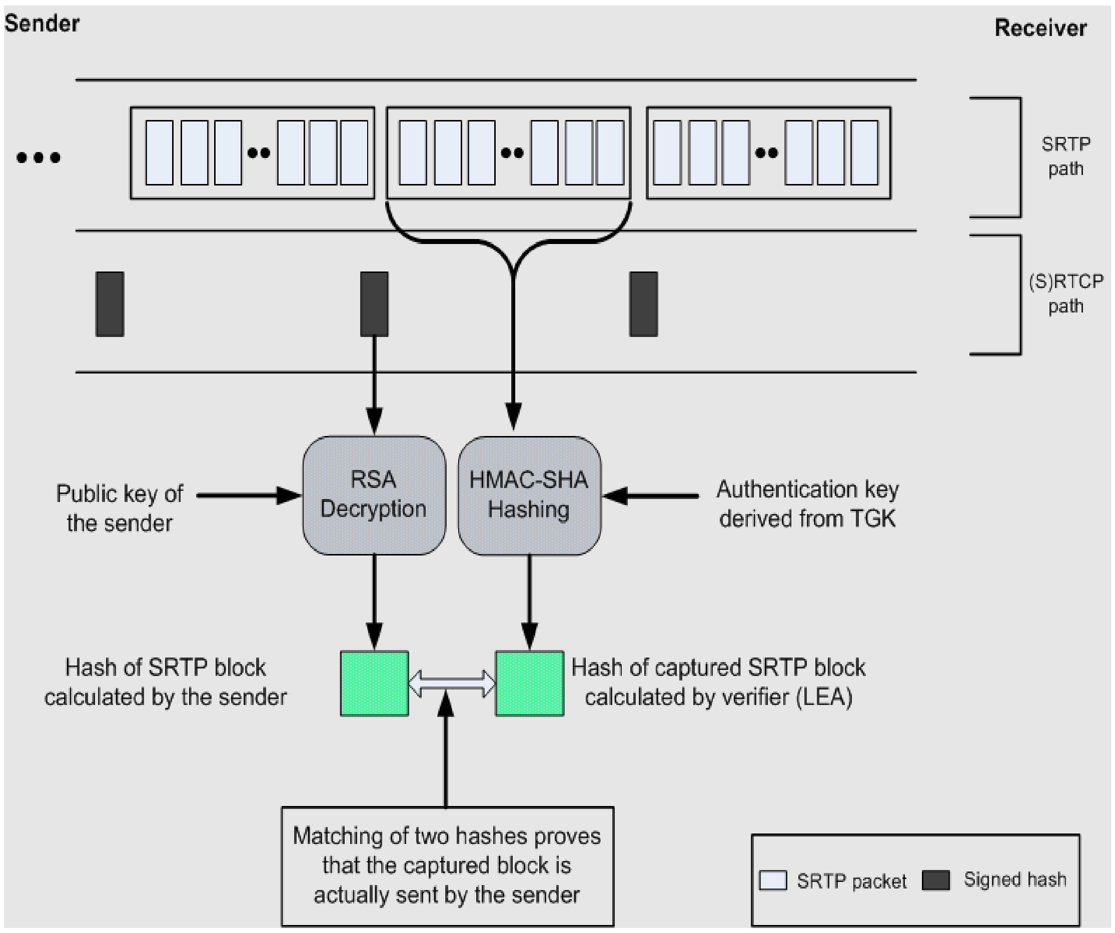

Sender
Receiver
SRTP path
(S)RTCP path
Public key of the sender
RSA Decryption
HMAC-SHA Hashing
Authentication key derived from TGK
Hash of SRTP block calculated by the sender
Hash of captured SRTP block calculated by verifier (LEA)
Matching of two hashes proves that the captured block is actually sent by the sender
SRTP packet
Signed hash

Figure 5-8: Signed hash verification by the proposed model

When the LEA captures a session for LI it should capture both the SRTP and SRTPC/RTCP packets. The SRTCP/RTCP path carries the signed hash values over the SRTP blocks; along with the usual RTCP information. For successful decryption of the session the LEA needs the session keys which it obtains from the escrow agent after showing proper authorization. Without the digitally signed hashes a dishonest employee of the LEA could modify packets in the captured session in order to present fabricated evidence in court and the subjects of this LI would have no evidence to prove this content was a forgery.

The LEA employee can forge SRTP packets as he or she has the session keys from the escrow agent, but he or she cannot compute the signed hash value as the signed hash value is calculated using the private key of the sender. However, the signed hash values could be used by the subject (or the court) to detect fabrication by the LEA of call contents. In Figure 5-9 we can see that the hash produced from the forged SRTP block by HMAC_SHA differs from the hash value produced from the signed hash value by RSA decryption operation. Note that if some SRTP packets were lost, this too would also produce a different hash value; but we can detect the loss of SRTP packets by using the sequence number and will ignore the failed verification of the block.

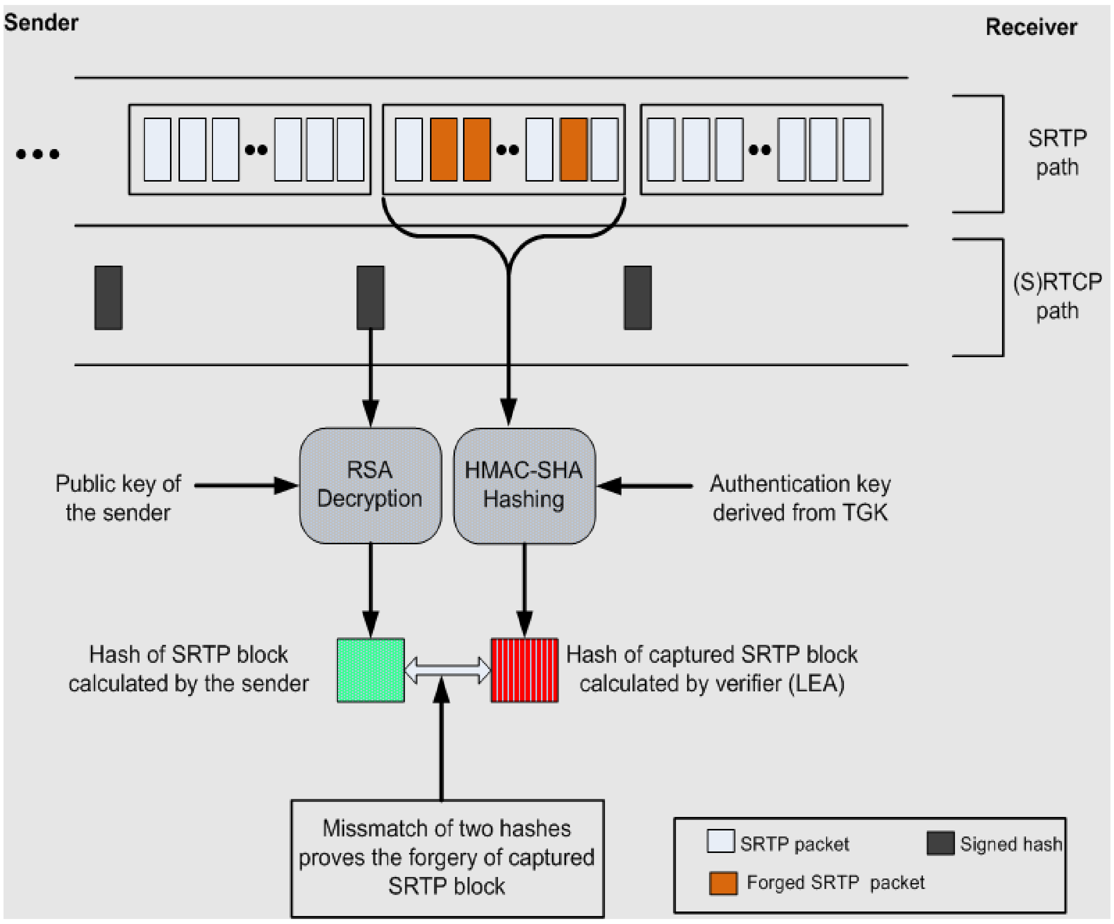

Sender
Receiver
SRTP path
(S)RTCP path
Public key of the sender
RSA Decryption
HMAC-SHA Hashing
Authentication key derived from TGK
Hash of SRTP block calculated by the sender
Hash of captured SRTP block calculated by verifier (LEA)
Missmatch of two hashes proves the forgery of captured SRTP block
SRTP packet
Signed hash
Forged SRTP packet

Figure 5-9: Detection of forgery by the proposed model

The proposed model can detect forgery on a block basis, but not on a per packet basis (unless the block size in one). Thus we can detect that a block that has been forged, but we cannot determine which packets inside a block have been forged. The granularity of our forgery detection depends on the block size (i.e., how many SRTP packets are grouped together). The smaller the block's size the finer the granularity of forgery detection; however, a smaller block size produces greater computational overhead due to frequent signing and hashing operations and will increase the amount of information that has to be sent via SRTCP/RCTP. This trade-off will be discussed in the next chapter where we will evaluate our proposed model.

Chapter 6: Performance Evaluation and Discussion

This chapter evaluates the performance of the proposed forgery detection model. It also presents a detailed discussion and analysis of the performance evaluation results. Two aspects of this evaluation are presented in this chapter. The first set of evaluations focus on the overhead introduced by our cryptographic operations to detect forgery. The second set of evaluations deal with the performance of escrowing the master key with the escrow agent.

6.1 Evaluation criteria

Quality of Service (QoS) is a central issue to the operation of VoIP. If the QoS of a VoIP system is unacceptable, then most of the attractive features (low cost, network convergence, increased security, etc.) of VoIP cannot be realized. The QoS that a VoIP user experiences can be degraded by the addition and/or poor implementation of security measures [40]. For example, the existence of a firewall or NAT can increase call set up delay or even block a call[41]; while use of encryption adds delay which can produce unacceptable latency and jitter (i.e., delay variation). As we are using asymmetric cryptographic operations that take significant CPU resources, there could be excessive latency. Therefore, the first criterion of our evaluation is that the delay introduced by our cryptographic operations to protect against forgery must not add significantly to the delay of the RTP traffic. We first measured the CPU time taken by the cryptographic operations, and then we measured the delay of the SRTP/RTP traffic.

To measure the CPU time we have used the Boost c++ library [42]. The class boost::posix_time::ptime is the primary interface for computations concerning time. The class boost::posix_time::time_duration is the base type for representing the length of a period of time. This duration can be either positive or negative. The general time_duration class provides a constructor that properly deals with hours, minutes, seconds, and fractional seconds. These functions can be used as follows:

```
ptime time_start(microsec_clock::local_time());
//do something
ptime time_end(microsec_clock::local_time());
time_duration duration(time_end - time_start);
cout << duration << '\n';
```

The second evaluation criterion is the amount of extra traffic generated due to the implementation of our model; as if there is too much additional traffic our model might interfere with the session content's transmission. This extra traffic consumes bandwidth (that in some cases may be a scarce resource). A discussion about how many extra bytes of traffic are sent and a possible way to minimize this communication overhead is also presented.

6.2 Evaluation of the forgery detection model

In our implementation we create SRTP blocks by grouping together SRTP packets and performing cryptographic operation over the SRTP blocks in order to be able to detect forgery of call content later (as elaborated in Chapter 5). How many packets should be grouped together is an important decision - as our model detects forgery on a per block basis rather than on a per packet basis (unless the block size is one SRTP packet). As a result the forgery detection granularity of our model is a function of the block size. If the block size is small, then our model will more accurately locate the data that has been forged. On the other hand if the block size is too large, the granularity of a detected forgery will be large, i.e., we cannot indicate which of many packets have been forged. However, a small block size means that the expensive cryptographic operation need to be performed more frequently, which requires more CPU resources and could lead to increased delay **and** extra traffic.

6.2.1 Delay introduced by the cryptographic operations

In our experiments we measured the overhead as a function of different sized SRTP blocks. Specifically we create blocks of 1, 8, 16, 32, 64, 128, 256, 512, and 1024 SRTP packets to perform the cryptographic operations according to our proposed model and measured the CPU time for each block size. We have examined the delay, both in terms of the total delay and in terms of its components.

Figure 6-1 shows a logical representation of the delay produced by our proposed cryptographic model. Note that the label “delay” in the figure represents the maximum delay that we can allow without affecting the inter-arrival times of the SRTP packets. In the case of G.711, the inter-arrival times of the SRTP packets is 20ms – minus the time required to process the underlying RTP packet (coding and placing the content into the RTP packet) and turning this into an SRTP packet. Note that this model assumes that there is a single processor that it doing all the computation and that the cryptographic operations for signing have to be completed before the next RTP packet can be processed (i.e., that the processing is done sequentially and not in parallel) – in practice this need *not* be true, but it represents the worst case effects on the RTP delay of performing these additional computations and it represents the current implementation. We will use this assumption of *serial* computation in the discussion that follows.

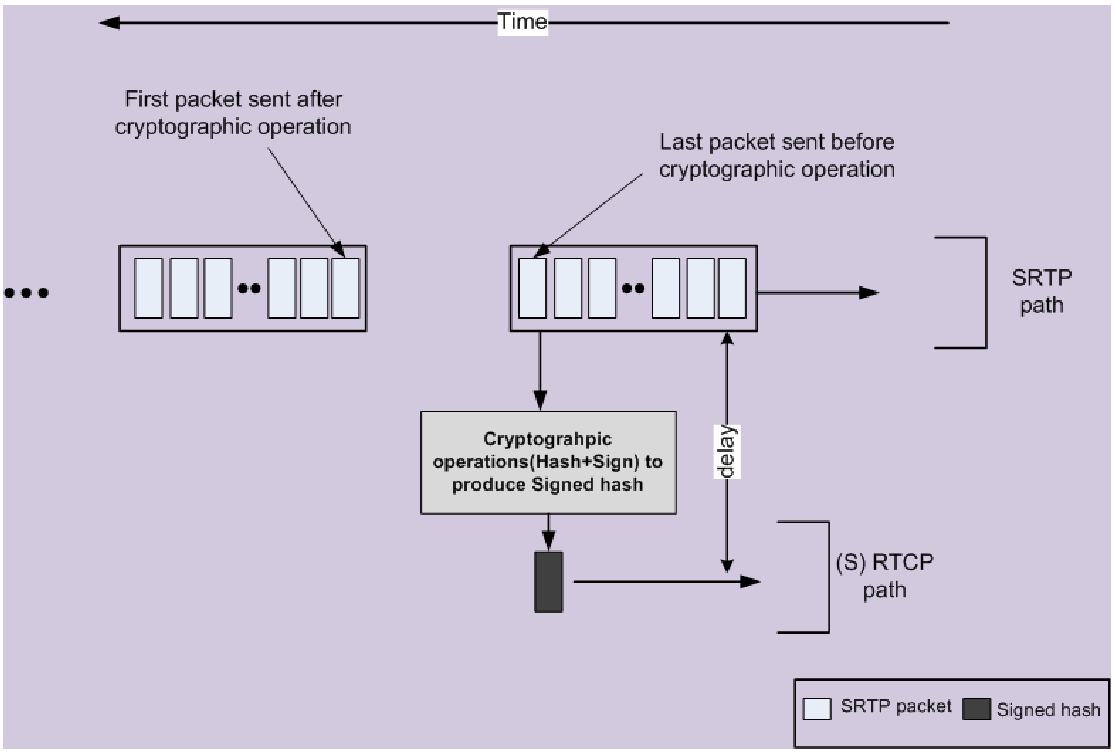

Time
First packet sent after cryptographic operation
Last packet sent before cryptographic operation
SRTP path
Cryptograhpic operations(Hash+Sign) to produce Signed hash
delay
(S) RTCP path
SRTP packet
Signed hash

Figure 6-1: Delay produced by the cryptographic operations (Hash+Sign)

The total delay introduced by our model is the sum of CPU time taken by all of the different operations performed. We have broken down the total delay as the delay due to the hashing operation, the delay due to the signing operation, and a fixed delay due to sending the signed hash value. The measurements were made on a Dell OptiPlex GX620 computer with an Intel Pentium D dual core processor running at 2.8GHz with 2 GB of memory. For detailed information about this CPU see Appendix F. In next two subsections we present our test results concerning hashing and signing delay for different block sizes.

6.2.1.1 Hashing delay

We have calculated the hash time based upon 50 test runs with each different block size using the posix::ptime class of the Boost c++ library. Table 4 shows the statistics over these measurements for 50 test runs for each different block size. The data for each of the individual runs are included in Appendix B.

Table 4: Statistical data of HMAC_SHA delay measurement in microsecond for different block size. These statistical values are calculated for 50 test runs.

Block Size	Average (µs)	Median (µs)	Minimum (µs)	Maximum (µs)	Standard deviation (µs)
1	8.94	9	8	13	1.202209
8	17	15	13	34	5.656854
16	25.08	21	19	37	6.638908
32	35.82	30	29	49	7.598845
64	52.78	50	49	66	5.207099
128	90.76	89	89	109	4.345112
256	168.58	167.5	167	182	3.35693
512	325.26	324	323	342	3.355227
1024	648.38	648	638	675	7.298001

For better understanding of the results we have plotted the data using both R[§] and Microsoft's Excel. Figure 6-2 shows an Excel plot of the hashing time for each block in microseconds for the 50 test runs; while Figure 6-3 shows an R box plot of the hashing time for different block size. From these two graphs we can see that the hashing time increases with increasing block size. Figure 6-4 shows the average hashing time for different block sizes.

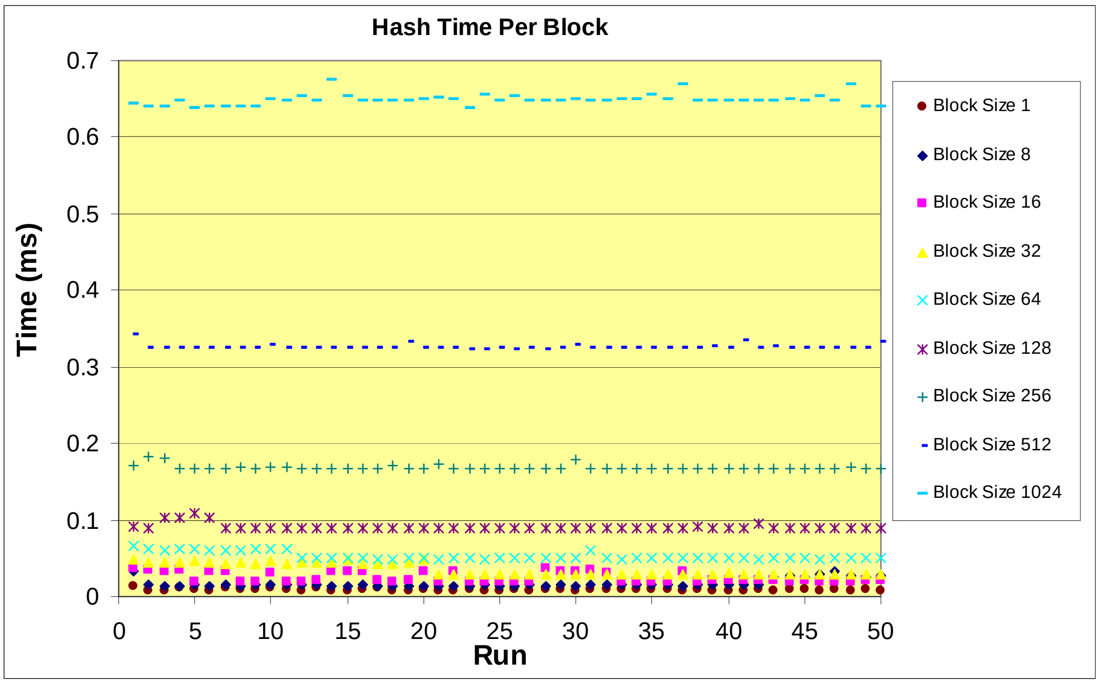


Figure 6-2: HMAC_SHA hashing time for 50 test runs of different block size Where, X axis represents the Run number and Y axis corresponds to time in millisecond

§ http://*www.r-project.org*

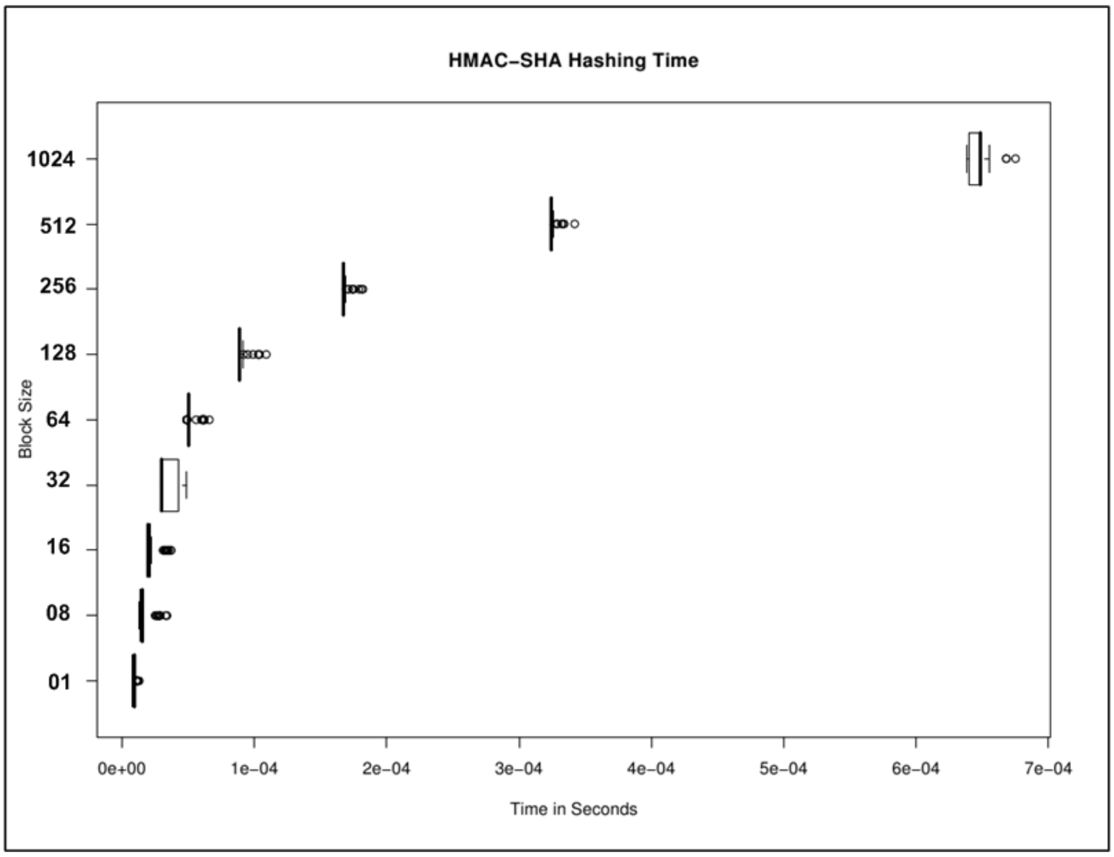


Figure 6-3: R boxplot showing the HMAC_SHA hashing time Each box represents individual block size indicated on Y axis and the Time in second is plotted on the X axis.

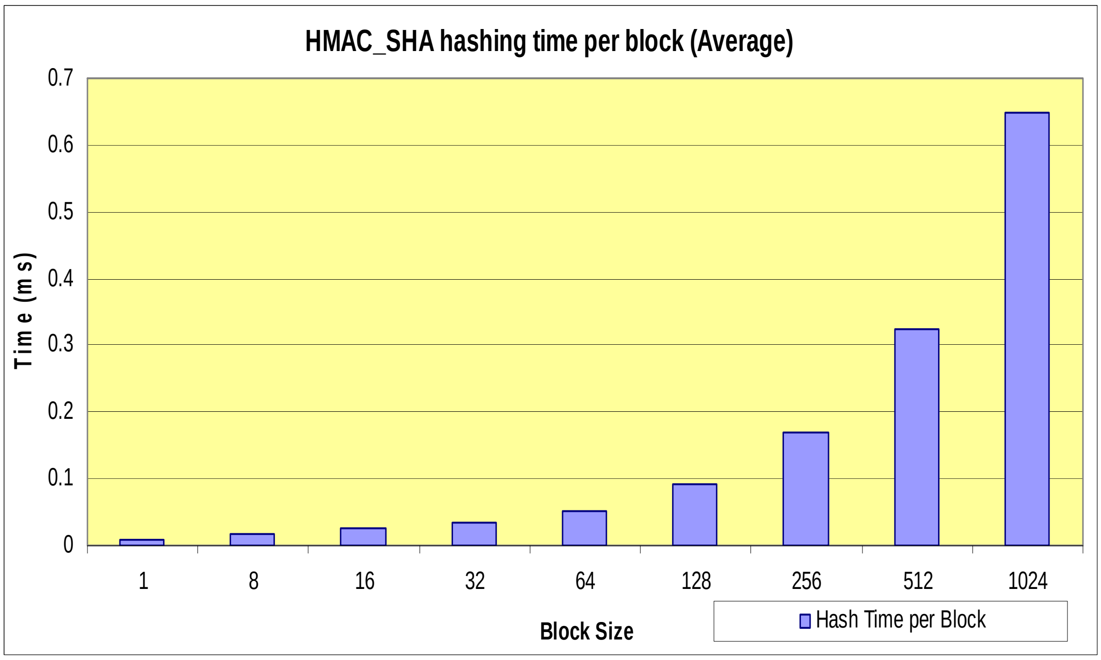


Figure 6-4: Averaged hashing time for different block sizes Where, X axis shows the block size and Y axis represents time in milliseconds (ms)

6.2.1.2 Signing delay

We have calculated the signing time for different block sizes based upon 50 test runs for each block size; in the same way as we calculated the hashing time. The statistical result for the signing time measurement are shown in Table 5. The actual measured data for individual run is included in Appendix C. Some plots of the data are shown are Figure 6-5, Figure 6-6, and Figure 6-7. From all these graphs it is notable that the signing time is almost constant for different block size and that this signing time is close to 3.5 milliseconds. The reason behind this nearly constant signing time is that we are always computing a signature of a fixed size hash value.

When the block size is 1 it takes 3.5 milliseconds to perform the signing operation. This 3.5 millisecond delay will occur every 20 milliseconds, as each SRTP packet contains 20 milliseconds worth of audio samples. On the other hand a block size of 128 requires the same overhead of approximately 3.5 milliseconds for signing, but this computation will only occur every 2560 milliseconds (i.e., every 2.56 seconds).

From Figure 6-6 we see that there is a set of outliers that follow roughly the same delay curve as the median values, but with an additional delay of ~1 ms. This is most likely due to multitasking, since the Linux scheduler is running with a HZ value of 1000 (i.e., 1 second divided by 1000 is 1 ms).

Table 5: Statistical results of signing time delay measurement in millisecond for different block size. These statistical values are calculated for 50 test runs.

Block Size	Average (ms)	Median (ms)	Minimum (ms)	Maximum (ms)	Standard deviation (ms)
1	3.4466	3.4305	3.376	4.243	0.118869
8	3.47552	3.461	3.417	4.298	0.120857
16	3.49208	3.4705	3.434	4.351	0.126334
32	3.49148	3.475	3.417	4.325	0.122656
64	3.49654	3.477	3.435	4.327	0.124445
128	3.47526	3.459	3.425	4.324	0.123611
256	3.50686	3.4835	3.426	4.354	0.130582
512	3.4977	3.474	3.445	4.358	0.126379
1024	3.50232	3.4765	3.448	4.583	0.157692

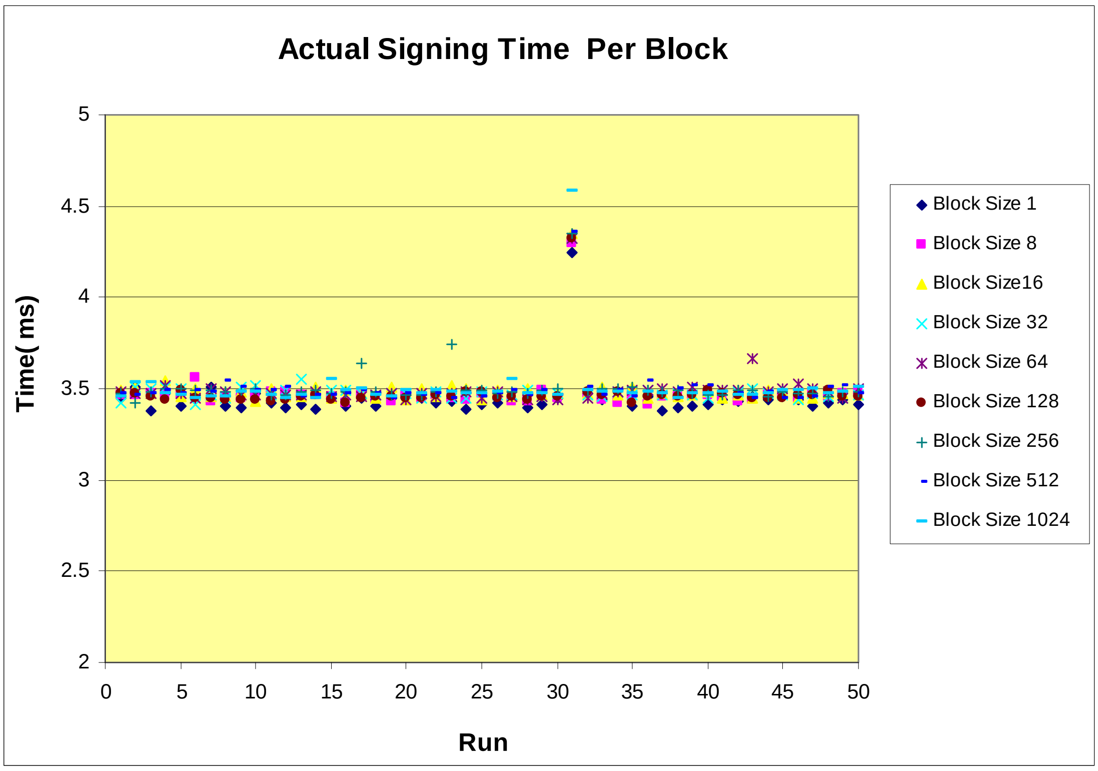


Figure 6-5: RSA signing time for 50 test runs with different block sizes Where X axis represents the run number and Y axis is the signing time in milliseconds.

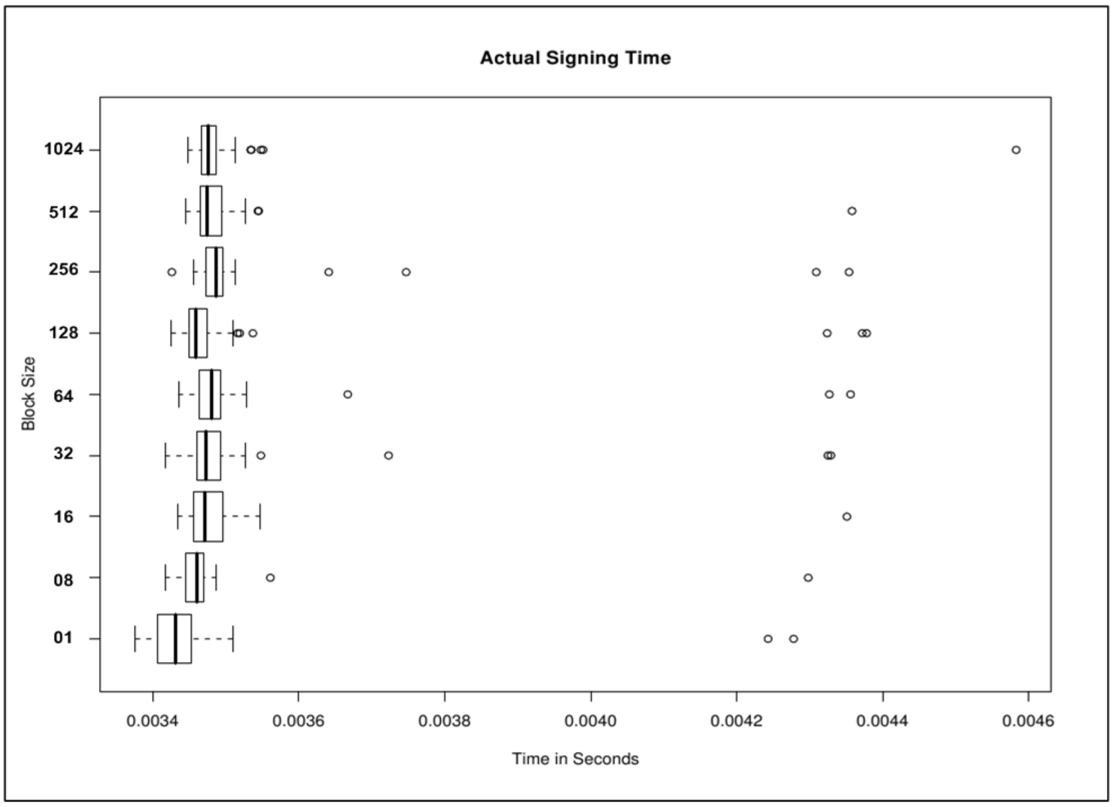


Figure 6-6: R boxplot showing the RSA signing time Each box represents the signing time for the block size indicated on Y axis and time in seconds is shown on X axis.

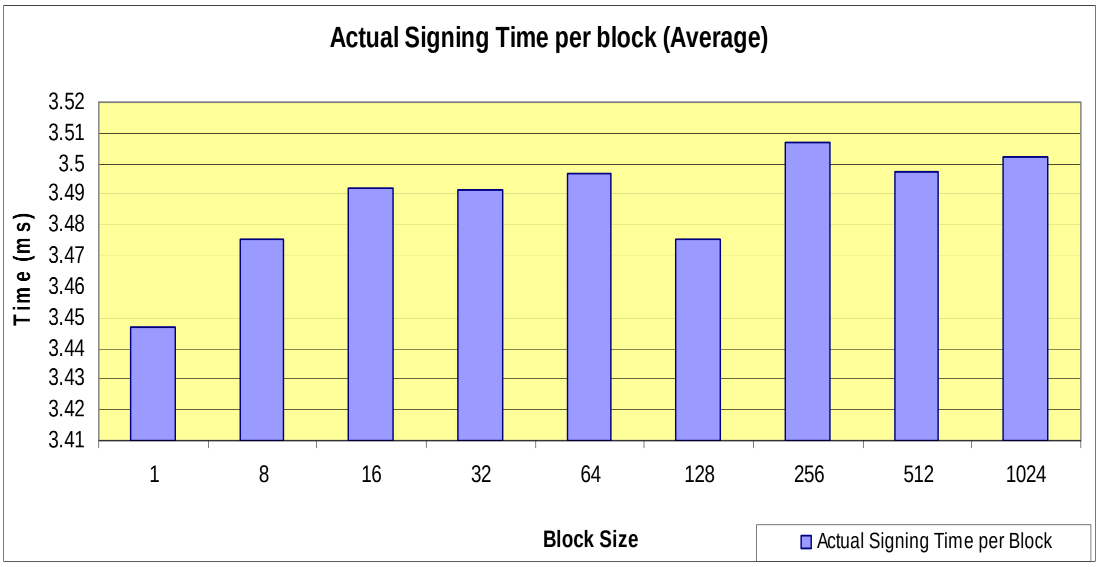


Figure 6-7: Averaged signing time for individual block size Where, X axis represent block size and Y axis represent time in millisecond (ms)

There are two curious aspects of these results. One is that for the 32nd run for all block sizes, something happens that causes the time to be longer. Second the processing time for 128 packets is comparable to the time for 8 packets and shorter than the time for all other block sizes (except for 1 and 8 packets).

Measurements were taken with a slightly different block size than the 128 which produces a local minimum. These measurement results are shown in Table 6. The measured data was plotted to see where the inflection point is. From Figure 6-8 it is notable that block size of 129 and 131 also require less signing time compared to others block sizes, as the median value of the box plots suggest. A similar result is also found in Figure 6-9 where the X axis represents the block size and the Y axis shows the average signing time for 50 test runs. The data for each of the individual runs are included in Appendix D.

Table 6: Statistical results of signing time delay measurement in milliseconds for different block sizes close to 128 to find the local minima. These statistical values are calculated for 50 test runs.

Block Size	Average (ms)	Median (ms)	Minimum (ms)	Maximum (ms)	Standard deviation (ms)
120	3.49344	3.475	3.429	4.346	0.125309765
124	3.49516	3.477	3.44	4.342	0.12402639
127	3.50306	3.4885	3.435	4.344	0.122551515
128	3.46642	3.447	3.426	4.378	0.132443148
129	3.48032	3.4605	3.414	4.32	0.125023172
130	3.49246	3.479	3.42	4.334	0.122959959
131	3.48856	3.476	3.441	4.35	0.137902829
132	3.5061	3.4605	3.455	4.471	0.139973212
136	3.49774	3.4725	3.433	4.345	0.126292794

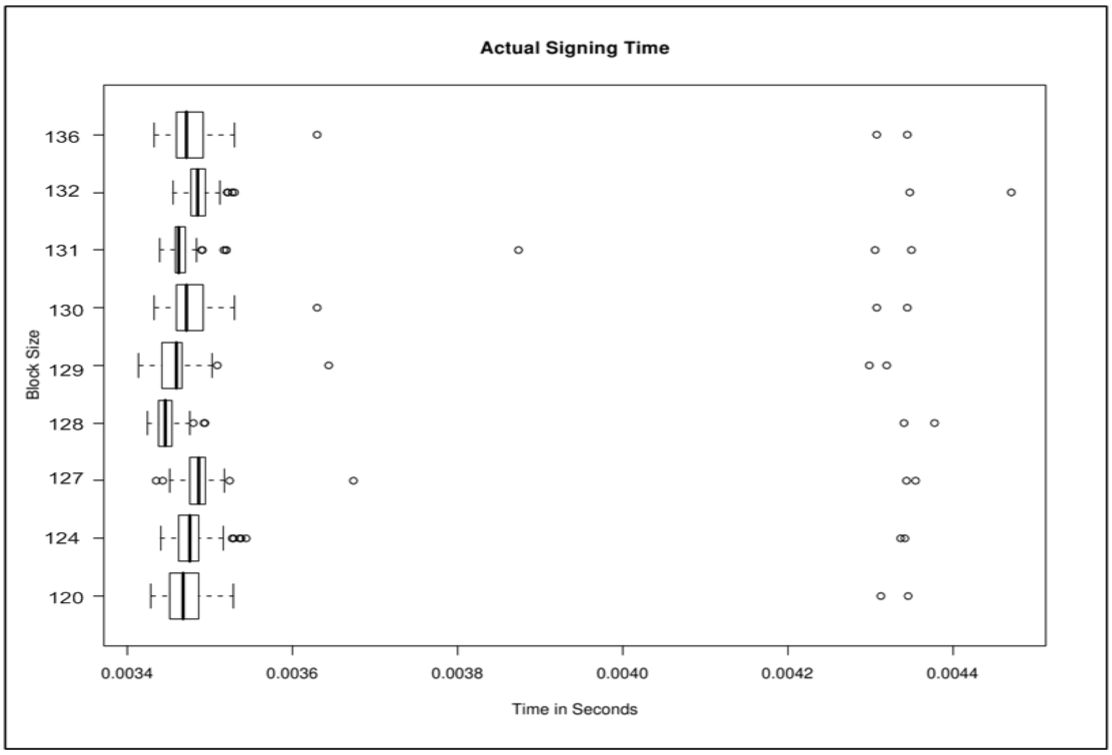


Figure 6-8: R boxplot showing the RSA signing time
Each box in Y axis represents the signing time for the block size close to 128 to find the local minima and time in seconds is shown on the X axis.

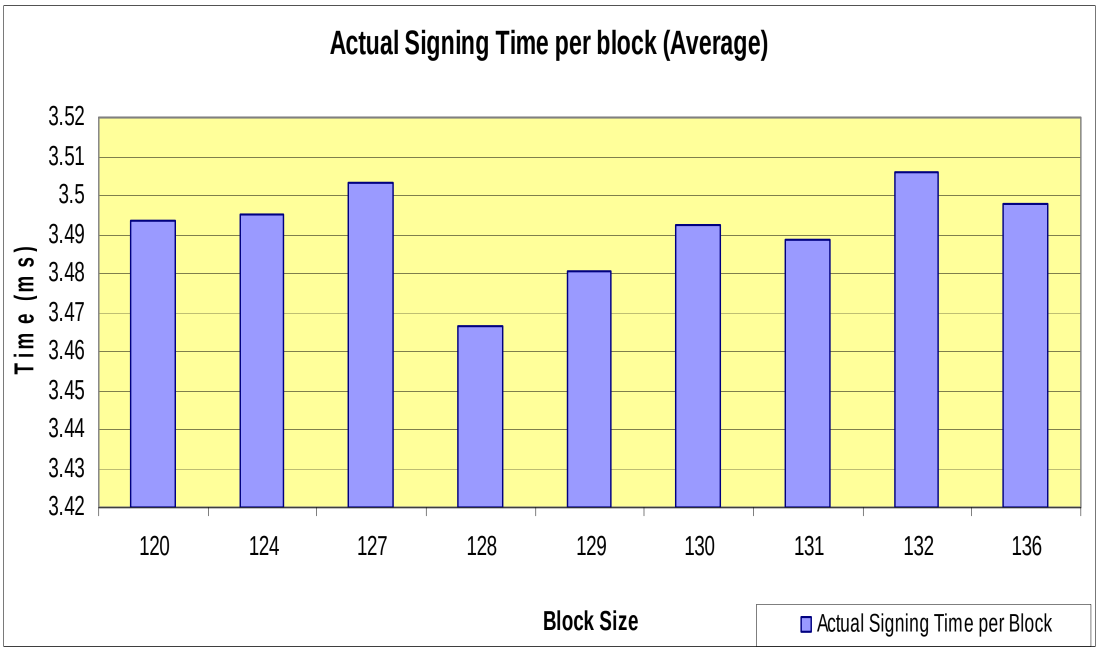


Figure 6-9: Averaged signing time for individual block size closer to 128
Where, X axis represent block size and Y axis represent time in millisecond (ms)

6.2.1.3 Total delay measurement

The total processing time (as noted in worst case this corresponds to added delay) is calculated in a similar fashion as used for calculating the hashing and signing time. The total delay includes the signing and hashing time plus the time required for sending the signed hash value through SRTP/RTCP path. The statistical analysis of these test results are presented in Table 7. The measured data for individual runs are included in Appendix E. Plots of the measured data are presented in Figure 6-10, Figure 6-11, Figure 6-12, and Figure 6-13, where the first two figures show the total delay produced by different size blocks for 50 test runs.

Figure 6-12 shows the average of total delay introduced by our cryptographic processing for different size blocks. From this figure it is clear that the total delay increases slowly with increasing block size. In Figure 6-13, the bars show the average total delay, signing delay, and hashing delay for different size blocks. This figure suggests that the lion's share of total delay comes from the signing operation and this is nearly a constant value. Actually the total delay increases with increasing block size due to additional time required for computing a hash over a larger amount of data. We can fit a linear curve to the total delay as follows:

$$Y = m*X + c$$

Where,

Y= Total Delay
X= Block Size (1, 8, 16, 32, ...)
m = Time per packet to hash (constant)
c = Signing time (constant)

From our experimental measurements we see that m=0.07 ms per packet and c=3.41 ms.

The total delay for these computations is directly related to clock speed of computer (used as the user agent). A user agent with the same type of processor but with a faster CPU will experience less delay; while a machine with a slower CPU will experience greater delay. At the same time increasing the block size means less frequent production of signed hashes, as the signing occurs on a per block basis. However, for smaller granularity of forgery detection we would like to use smaller block sizes. So a machine with a faster CPU can utilize a smaller block size and offer finer granularity of forgery detection *without* introducing too much delay.

Table 7: Statistical results of total delay measurement in milliseconds for different block sizes. These statistical values are calculated for 50 test runs.

Block Size	Average (ms)	Median (ms)	Minimum (ms)	Maximum (ms)	Standard deviation (ms)
1	3.55114	3.513	3.459	4.466	0.183084
8	3.6507	3.626	3.582	4.471	0.12418
16	3.62252	3.5985	3.548	4.447	0.121999
32	3.63178	3.613	3.53	4.462	0.121999
64	3.68958	3.6695	3.636	4.52	0.12314
128	3.7219	3.6995	3.66	4.574	0.125689
256	3.78964	3.775	3.739	4.64	0.124207
512	3.9421	3.9255	3.891	4.759	0.119316
1024	4.27708	4.26	4.214	5.074	0.116868

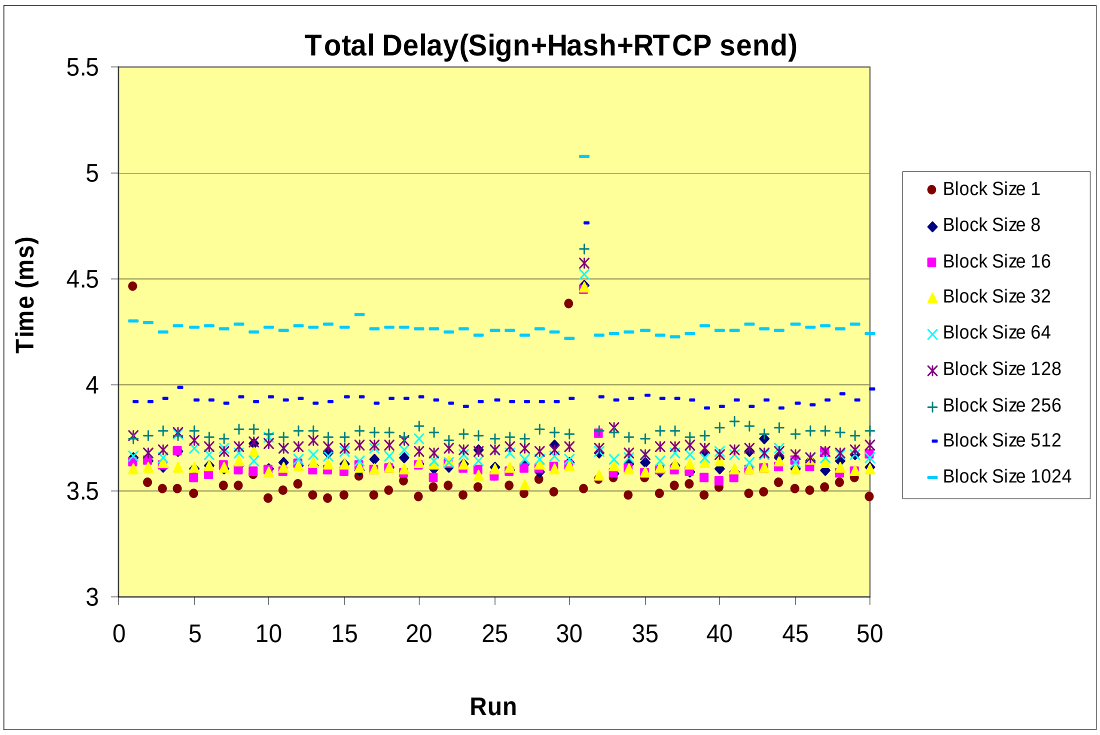


Figure 6-10: Total delay for 50 test runs of different block size
Where, X axis represents the Run number and Y axis corresponds to time in microsecond

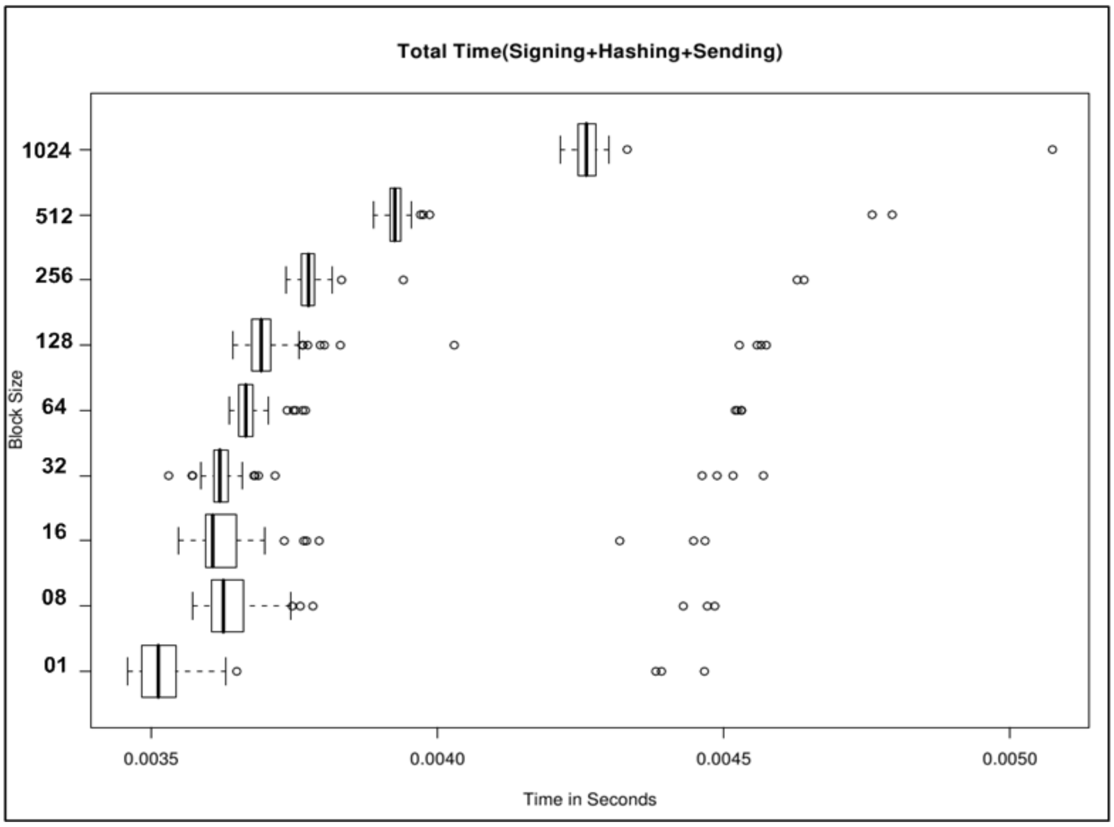


Figure 6-11: R plot showing the total delay
Where, every box represents individual block size indicated in Y axis and Time in second is indicated in X axis.

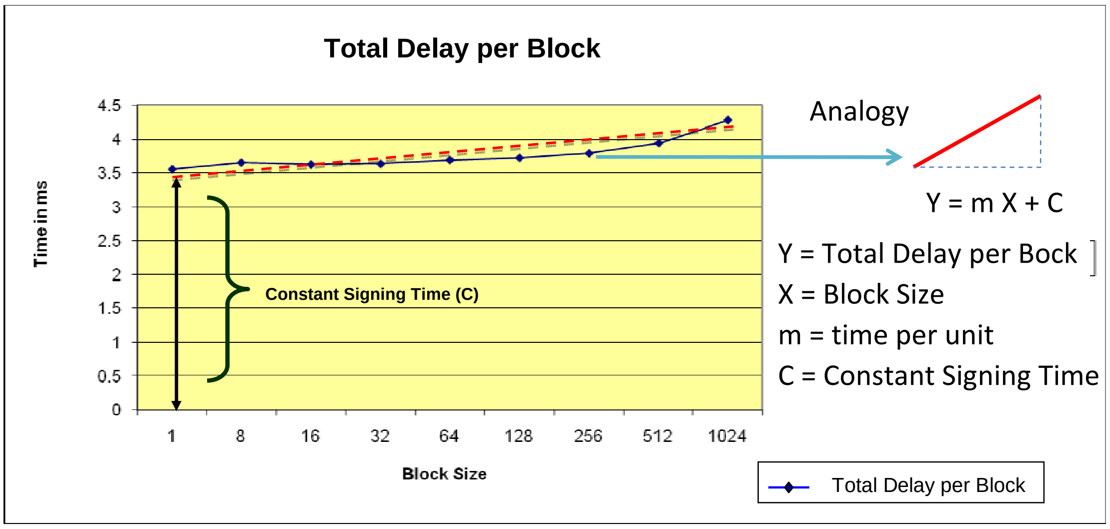


Figure 6-12: Average total delay for individual block size
Where, X axis represent block size and Y axis represent time in millisecond (ms)

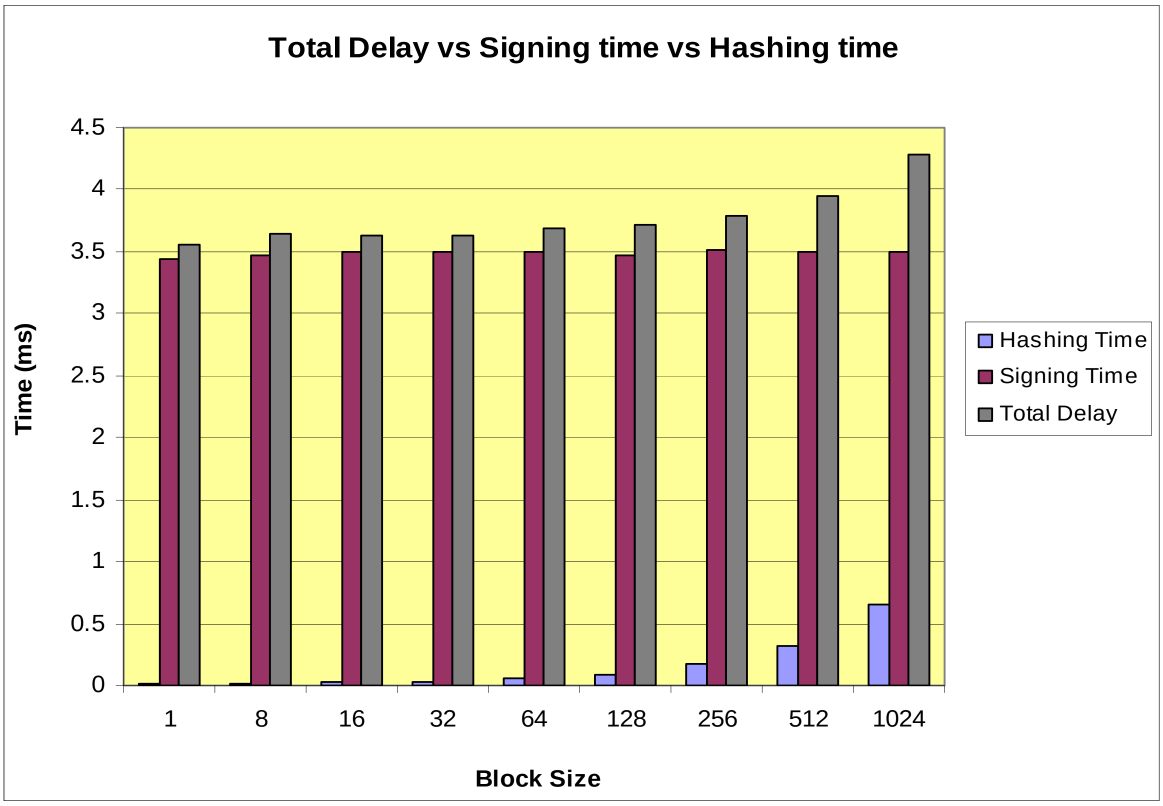


Figure 6-13: Total delay, signing time and hashing time for different size of block

6.2.2 Extra traffic generated by the signed hashes

To detect the forgery of the call content we send the signed hashes of SRTP blocks via the SRTCP/RTCP path. In our current implementation we are sending a signed hash (128 bytes) along with a signed hash sequence number (4 bytes) as an UDP packet. When sent over an Ethernet there will be an additional 42 bytes of extra protocol headers. Figure 6-14 shows the structure of a UDP packet containing a signed hash value. For every SRTP block we send only 174 bytes of extra traffic out the network interface. Of these 132 bytes are the application data and 42 bytes are overhead for the different protocol headers.

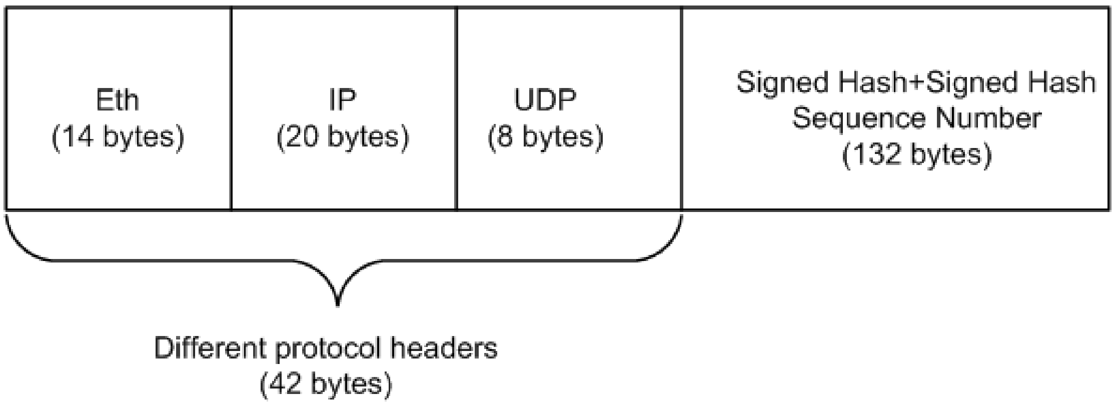


Figure 6-14: Signed hash value inside a UDP packet

The extra traffic generated by our model directly depends on the block size. If the block size is small then signed hashes are produced and sent across the network more frequently.

Table 8 shows the intervals between signed hashes for different size blocks and their logarithm value. As the table shows with a block size of 64 these 174 byte frames would only be sent every 1.3 seconds. In comparison to the 50 frames per seconds that are sent for G.711 encoded audio, this is 2% additional traffic in term of the number of packets and 1.2% additional bytes. (Note that RTCP alone would typically generate ~5% more traffic.)

In Figure 6-15, the histogram shows the signed hashes' inter-arrival time as a function of block size. For a better understanding a log-log plot with error bars is presented in Figure 6-16. Fitting an equation to this data allows us to estimate the inter-arrival times for other values of block size. From the log-log graph it can be clearly understood that signed hash inter-arrival time maintains an exact linear relationship as the R (correlation coefficient) squared value is 1. From the trend line and the equation shown in the Figure 6-16 it can be observed that the curve intercepts the Y axis at the value 4.33, this represents the constant portion. This constant value is the logarithm of the signed hash inter-arrival time when the block size is 1.

Table 8: Signed hash interval for different size of block. These average values are calculated for 50 test runs.

Block size	Average Inter Arrival Time (ms)	Log2(Block size)	Log2(Average Inter Arrival time)
1	20.27826	0	4.34186196
8	159.89016	3	7.320937345
16	319.89166	4	8.32143957
32	639.8549	5	9.321600972
64	1290.14344	6	10.33331576
128	2559.78514	7	11.321807
256	5119.71994	8	12.32184918
512	10239.58178	9	13.32186917
1024	20498.5962	10	14.32323749

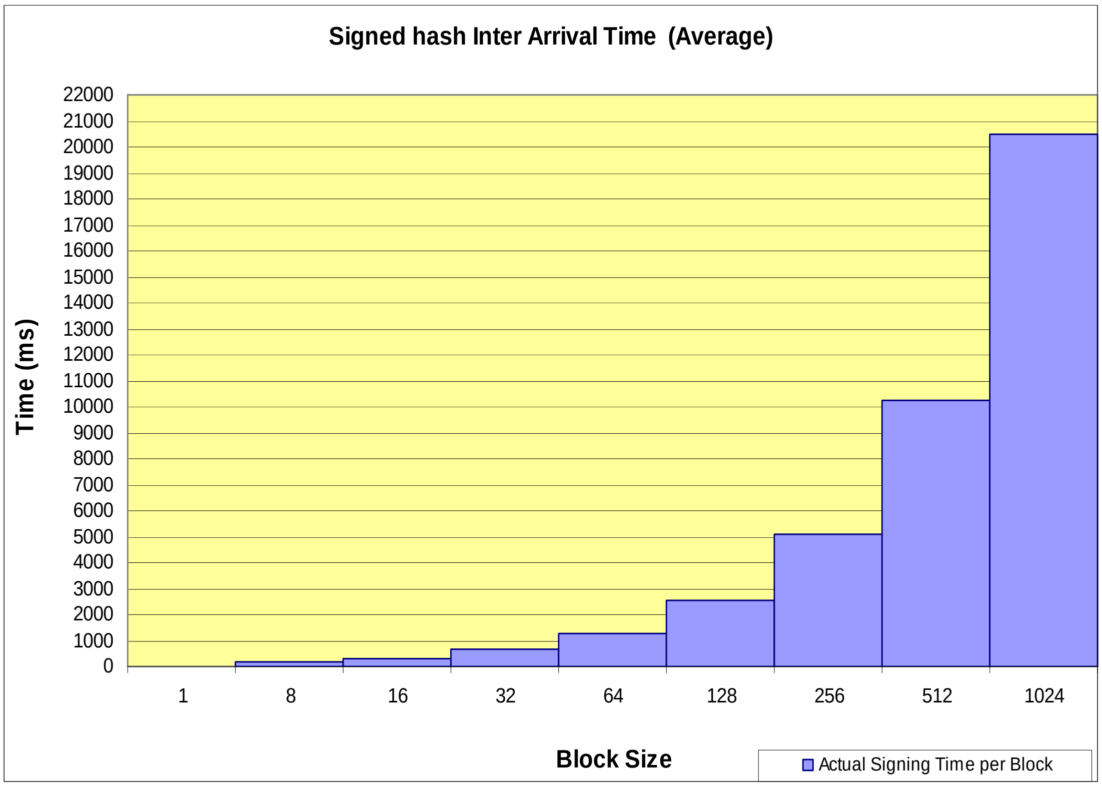


Figure 6-15: Signed Hash Inter Arrival time for different block size Where, X axis represent block size and Y axis represent time in milliseconds

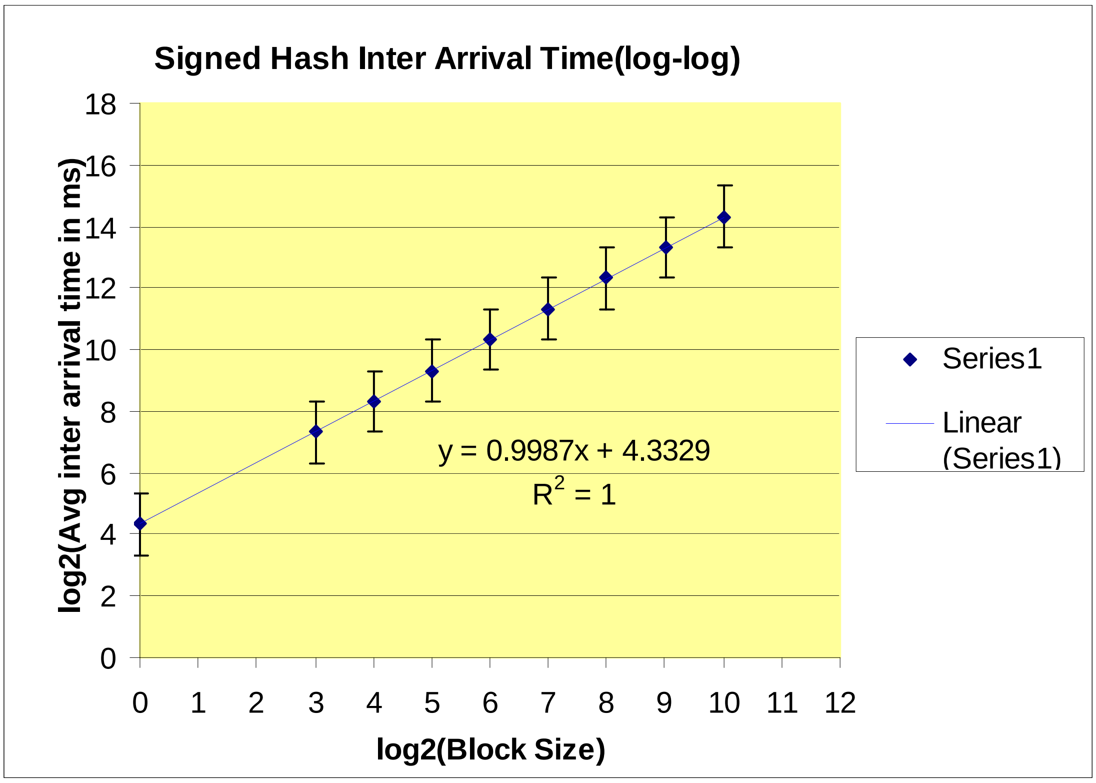


Figure 6-16: log-log plotwith error bars showing the signed hash inter arrival time Where, X axis represent logarithm of block size and Y axis shows the logarithm of average inter arrival time

The 42 bytes of overhead due to the different protocol headers could be reduced by sending the signed hash together with a RTCP Sender Report (SR) or Receiver Report (RR); as these reports need to be sent by the user agent (see Figure 6-17). The default interval between sender/receiver reports in a RTP/RTCP implementation is five seconds (5000 milliseconds).

From Table 8 we can see that for a block size of 256 or more the signed hash interval is greater than the RTCP SR/RR interval. Therefore we can place the signed hashes inside a RTCP SR/RR for a block size of 256 or larger, hence avoiding the 42 bytes of extra overhead due to the lower layers. For a block size of 128 we can either (1) place half of our signed hashes in RTCP SR/RR hence reducing the extra overhead by 50% or (2) we could delay sending half the signed hashes and place two signed hashes together with the RTCP SR/RR. Similarly 25% of extra overhead could be avoided by placing the signed hash inside the next RTCP SR/RR for blocks of 64 SRTP packets or four signed hashes could be put in each RTCP SR/RR.

Note that since there is no need for low delay for the delivery of the signed hashes, there is no reason not to delay sending the signed hashes until an RTCP SR/RR needs to be sent. Hence the cost is 132 bytes of signed hash times the numbers of hashes that have been done since the last RTCP SR/RR. Table 9 shows a simple estimate of the additional number of bytes due to the signed hashes that need to be sent as a function of the block size when piggy-backing the signed hashes on the RTCP SR/RR packets. Unfortunately, we see that for small block sizes we quickly regain some of the overhead due to the lower layers, since we will exceed the PATH MTU size and will need to fragment the packets. Thus for a block size of 256, 128, 64, and perhaps even for 32 it may make sense to delay sending the signed hash and exploit the decrease in overhead due to piggy-backing, but this gain is small for smaller block sizes.

Table 9: Increased size of the RTCP SR/RR report to carry the signed hash

Block Size	Number of bytes
1	8448
8	4224
16	2112
32	1056
64	528
128	264
256	132

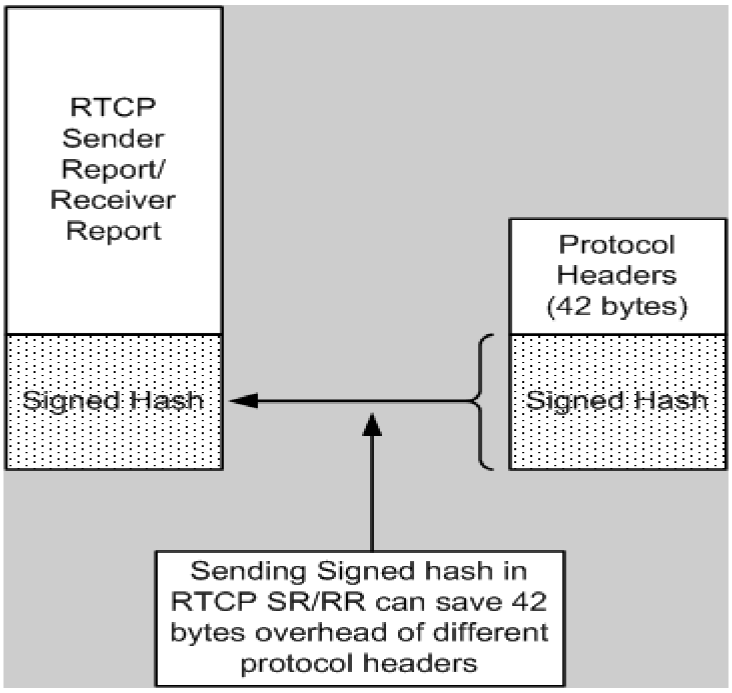

RTCP Sender Report/ Receiver Report
Signed Hash
Protocol Headers (42 bytes)
Signed Hash
Sending Signed hash in RTCP SR/RR can save 42 bytes overhead of different protocol headers

Figure 6-17: Placing the signed hash inside RTCP SR/RR

6.3 Escrowing overhead measurement

One of the major contributions of this thesis is to build an escrow agent and to escrow the session master key with this escrow agent. This section discusses the performance of the escrow operation. The method of escrowing the session master key was presented in Chapter 4:. However, we expect some performance problems because we protect the escrowing of a session master key using an SSL/TLS tunnel. While this tunnel protects the mastery key and information that is being escrowed we have to set-up a tunnel before we can send any information to the escrow agent. For simplicity we examined the case where this tunnel must be set up for each successful call that has terminated.

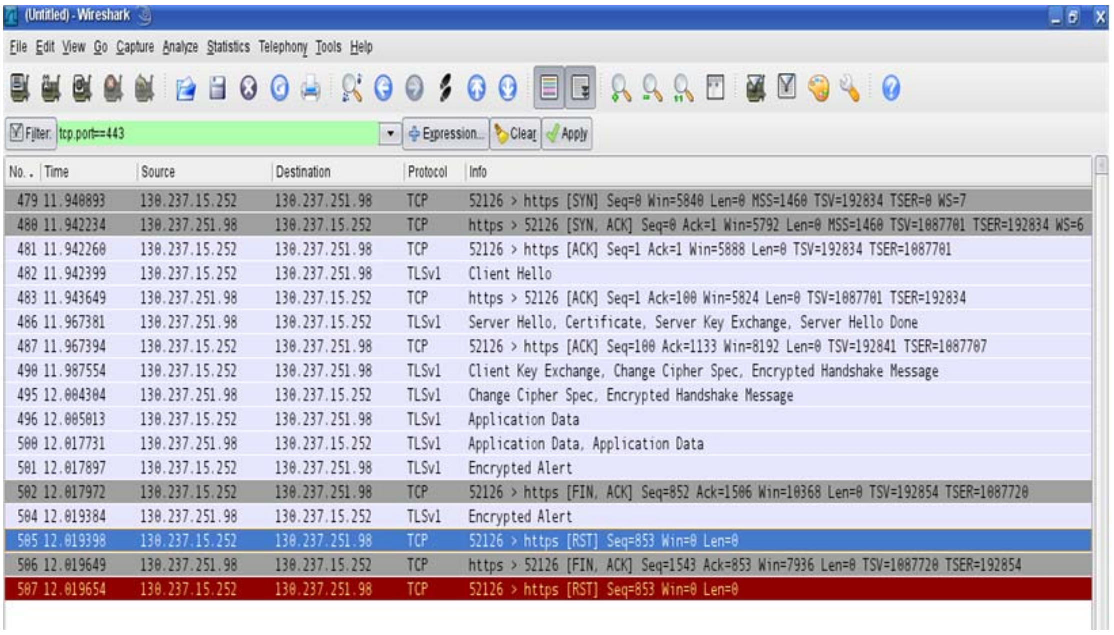
The time required for escrowing a session's master keys with the escrow agent was measured using the same Boost c++ library function described earlier. For testing the escrow agent was connected to (1) the same LAN as the user agent or (2) a different LAN from the user agent. Table 10 shows the measured time for 20 test runs of each of these two configurations. Note that in the second configuration the escrow agent has been moved to another subnet within the KTH campus network. From these measurements we can see that the time to escrow a key (with an escrow agent located in the same LAN) is very small, i.e., below 100 ms in most cases. (Note that in the first case there are some extra costs as we have to resolve the IP address of the escrow agent into a MAC address.)

Table 10: Time required escrowing a session master key with the escrow agent

Run	Escrow agent is in the same LAN (ms)	Escrow agent is in different LAN(ms)
1	112.799	150.966
2	57.467	99.89
3	50.889	71.607
4	80.093	127.361
5	50.505	73.824
6	64.092	122.736
7	49.689	126.988
8	52.321	72.962
9	53.778	72.324
10	51.002	87.847
11	66.537	125.804
12	68.26	140.801
13	66.06	105.576
14	67.537	173.664
15	66.5	118.636
16	67.661	135.4
17	64.911	136.467
18	65.772	143.409
19	64.789	119.72
20	63.976	129
Average	**64.2319**	**116.7491**

While the escrow time is tens of milliseconds or more, this time may **not** be significant as key escrow is performed *at the end* of a successful session. Therefore the time required to escrow material has no effect on the media latency, but *might* add some delay before a new session could be established.

However, the amount of communication required to escrow the desired information may be significant –as this information has to be sent across what ever network interface the user agent uses to communication with the escrow agent. Since the SSL/TLS tunnel relies on TCP connection, we begin by considering the number of packets required to establish a TCP connection and to tear down this connection. To this we have to add the SSL/TLS handshake for keys and cipher suite negotiation. Figure 6-18 shows the packets captured for a single call’s escrow operation. Figure 6-19 shows the flow of packets between the user agent (130.237.15.252) and the escrow agent (130.237.251.98) in sequential order. From these two figures it can be seen that the actual session key and other security parameters are transferred as application data.



```
No. . Time        Source           Destination      Protocol  Info
479 11.940893   130.237.15.252   130.237.251.98   TCP       52126 > https [SYN] Seq=0 Win=5840 Len=0 MSS=1460 TSV=192834 TSER=0 WS=7
480 11.942234   130.237.251.98   130.237.15.252   TCP       https > 52126 [SYN, ACK] Seq=0 Ack=1 Win=5792 Len=0 MSS=1460 TSV=1087701 TSER=192834 WS=6
481 11.942260   130.237.15.252   130.237.251.98   TCP       52126 > https [ACK] Seq=1 Ack=1 Win=5888 Len=0 TSV=192834 TSER=1087701
482 11.942399   130.237.15.252   130.237.251.98   TLSv1     Client Hello
483 11.943649   130.237.251.98   130.237.15.252   TCP       https > 52126 [ACK] Seq=1 Ack=100 Win=5824 Len=0 TSV=1087701 TSER=192834
486 11.967381   130.237.251.98   130.237.15.252   TLSv1     Server Hello, Certificate, Server Key Exchange, Server Hello Done
487 11.967394   130.237.15.252   130.237.251.98   TCP       52126 > https [ACK] Seq=100 Ack=1133 Win=8192 Len=0 TSV=192841 TSER=1087707
490 11.987554   130.237.15.252   130.237.251.98   TLSv1     Client Key Exchange, Change Cipher Spec, Encrypted Handshake Message
495 12.004304   130.237.251.98   130.237.15.252   TLSv1     Change Cipher Spec, Encrypted Handshake Message
496 12.005013   130.237.15.252   130.237.251.98   TLSv1     Application Data
500 12.017731   130.237.251.98   130.237.15.252   TLSv1     Application Data, Application Data
501 12.017897   130.237.15.252   130.237.251.98   TLSv1     Encrypted Alert
502 12.017972   130.237.15.252   130.237.251.98   TCP       52126 > https [FIN, ACK] Seq=852 Ack=1506 Win=10368 Len=0 TSV=192854 TSER=1087720
504 12.019384   130.237.251.98   130.237.15.252   TLSv1     Encrypted Alert
505 12.019398   130.237.15.252   130.237.251.98   TCP       52126 > https [RST] Seq=853 Win=0 Len=0
506 12.019649   130.237.251.98   130.237.15.252   TCP       https > 52126 [FIN, ACK] Seq=1543 Ack=853 Win=7936 Len=0 TSV=1087720 TSER=192854
507 12.019654   130.237.15.252   130.237.251.98   TCP       52126 > https [RST] Seq=853 Win=0 Len=0
```

Figure 6-18: Screenshot showing the packets involved in a single escrow operation

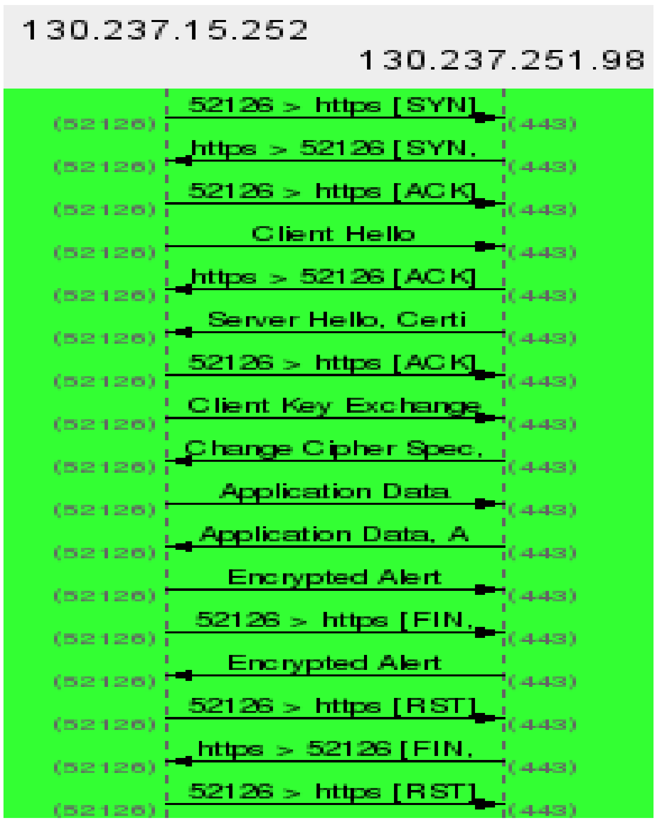


Figure 6-19: The flow of packets for a single escrow operation

Table 11 shows the number of packets and number of bytes that need to be sent for SSL/TLS tunnel establishment and closing of the tunnel. For a single escrow operation 11 packets are exchanged between the user agent and the escrow agent resulting in 2258 bytes of network traffic in addition to the application data (i.e., the master key and other security parameters). Note that the number of packets required to establish and close of a tunnel is constant, while the size of the packets may vary due to the use of different cipher suites by different servers.

Table 11: Number of packets and bytes sent as overhead in addition to the master key and other security parameters for a single escrow

Activity	Number of packets	Number of bytes
TCP connection establishment	3	76 + 76 + 68 = 220
SSL/TLS client hello	2	167 + 68 = 235
SSL/TLS server hello	2	1218 + 68 = 1286
Exchange of keys and cipher specification	2	266 + 127 = 393
TCP connection tear down	2	68 + 56 = 124
Total	11	2258

We could reduce both the delay and number of packets (and bytes) that need to be exchanged between the user agent and the escrow agent by keeping the SSL/TLS tunnel open for subsequent escrow operations. Doing so would enable a significant amount of overhead to be avoided. By keeping the tunnel open this constant overhead (11 packets) per escrow operation can be omitted. The total overhead savings directly depends on the frequency of calls. For N calls that can share the SSL/TLS tunnel set-up this approach can save the following amount of overhead in comparison to opening and closing the SSL/TLS connection for every escrow operation:

$$X = (N-1) * Y$$

Where,

X = total amount of overhead saved
N = Number of calls sharing a tunnel set-up & teardown
Y = Fixed overhead per escrow operation

Moreover this approach would save a significant amount of time for every escrow (except the first one) as (1) there will be no need for SSL/TLS handshake for cipher suite negotiation as this involves asymmetric cryptographic operations and (2) there are fewer round-trip packet exchanges. If the frequency of call is high, then keeping the tunnel open would be beneficial. On the other hand if the frequency of calls (and hence escrow operations) is too low, then it would be better to close the tunnel. In this regard some important questions need to be addressed:

- How long should a SSL/TLS tunnel be kept open?
- When does the tunnel need to be closed?
- What is the number of uniquely identified open connections that the escrow server can support?
- How many active connections can an escrow agent support?

Since each SSL/TLS tunnel requires a TCP connection, each tunnel can be uniquely identified by the source IP address, source port number, destination IP address, and destination port number. Note that the destination port number is likely to a fixed port, hence there will be only one such port number. Moreover, if a user agent is located inside a NAT, then the total number of uniquely identifiable tunnels could be limited based upon the number of IP addresses assigned for the external interface of the NAT (i.e., limiting the number of source IP addresses and source port numbers that can be used) and the IP address and port number resources used by other traffic passing through this NAT (since the NAT needs to assign different port number and IP address combinations for all of the TCP traffic passing through it). A complete analysis regarding of these issues is required before initiating deployment of the proposed solution. While these issues are interesting they are left for future work.

6.4 Time between BYE and escrow

The time between SIP's BYE message and the escrow operation is presented in this section. The time required to escrow a key was presented in section 6.3 and it was mentioned that escrow time may not be significant as key escrow is performed at the end of a successful session. Therefore the time required to escrow the key materials and the time between BYE and escrow have no effect on the media latency, but might be significant as these two may add some delay before a new session could be established. Table 12 shows the time between BYE and the escrow operation for 20 test runs in sorted order and a plot of the measured data is presented in Figure 6-20. The total time between the BYE and the escrow operaiton is less than 100 ms - and this is comparable to the time to send five audio frames (as they are each 20 ms long with the G.711 CODEC that is typically used).

Table 12:Time between BYE and the escrow operation for 20 test runs

Run	Time in Milliseconds(ms)
1	8.836
2	10.109
3	13.531
4	15.396
5	17.94
6	18.039
7	18.728
8	20.874
9	25.247
10	27.77
11	35.324
12	39.576
13	43.866
14	45.052
15	47.236
16	61.84
17	78.32
18	83.667
19	87.258
20	96.306
Average	39.74575

A LEA can not get the keys for a session until they have been escrowed. As we only escrow the keys for a session at the close of the session (in the normal case this is indicated by the SIP BYE message), the time between this BYE message and the key being escrowed is the minimum delay that a LEA would experience for LI of this session. Note that the effective end of the media stream could be much sooner than the BYE message if the parties in the session do not transmit the BYE immediately after they finish sending RTP packets.

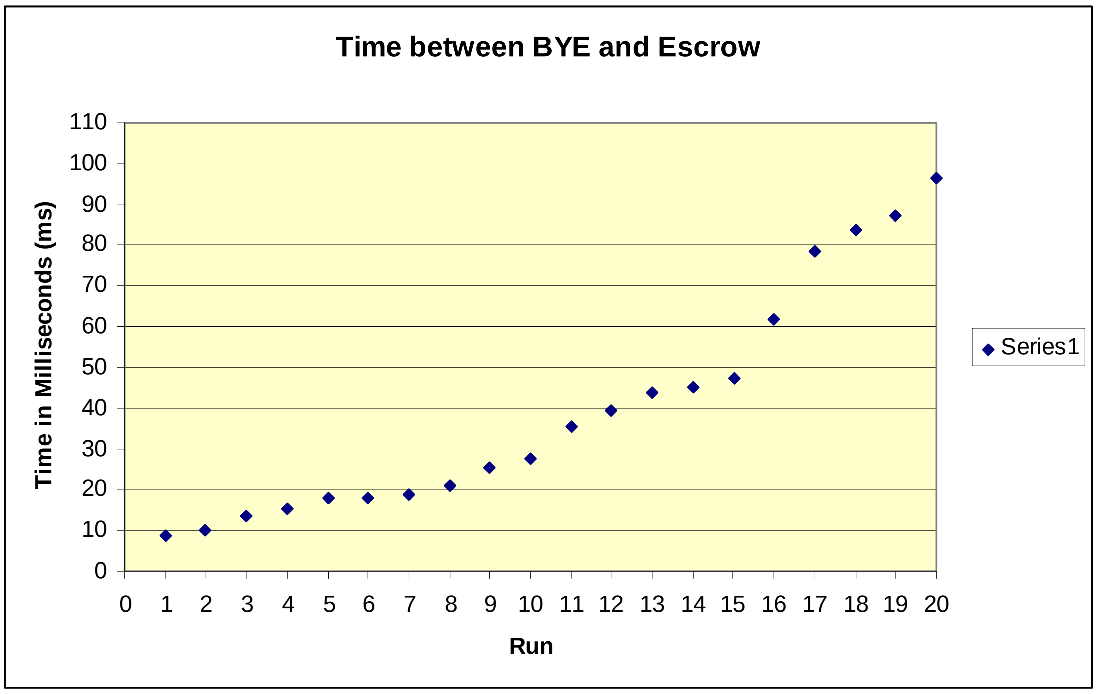


Figure 6-20:Time between BYE and escrow (sorted in increasing delay)

The major component of the delay between the BYE and the escrow operation is taken by the operations responsible for the closing of the real time media stream sender and the receiver. Separate measurements were taken to calculate the time required to close the real time media stream sender and the receiver. An interesting observation is that the time required to close the media stream sender is almost constant (3.8 ms). This time is due to the cryptographic operations performed on the last SRTP block as this operation is performed inside the RealtimeMediaStreamSender::stop (). So the proposed key escrowing model is only responsible for approximately 3.8 ms of the delay. The rest of the delay is produced by the invocation of RealtimeMediaStreamReceiver::stop () that is responsible for closing of the real time media stream receiver.

6.5 Summary

A complete performance evaluation of the proposed key escrowing model with signed hashes to detect against forgery has been presented in this chapter. The evaluation criteria were presented in section 6.1 based along with the measurements that were made. Section 6.2 discusses the overhead produced by the proposed model with relevant measurement data, charts, and graphs. The performance of the escrow agent was presented in section 6.3 with a discussion of the details of the messages required to do the key escrowing. Finally section 6.4 discussed the delay between the BYE and the escrowing of the key; thus completing the discussion of the delays for all of the processing due to the introduction of the proposed key escrow functionality.

Chapter 7: Conclusions and Future Work

Voice over Internet Protocol (VoIP) is a revolutionary application both in its effects upon the traditional telephony infrastructure and regulations, but VoIP call characteristics may be different than calls via the traditional fixed and mobile telephony system. Part of the reason for the change in VoIP call characteristics is the increasingly dynamic working habits of user and SIP's support for user, device, and session mobility.

VoIP is also a popular choice for real-time communication due to the security features it can provide. Many security protocols have been used to provide encryption and integrity protection of the real-time traffic and to provide authentication of the parties participating in a session. While the VoIP users are happy with the benefits of this additional security, governments and their Law Enforcement Agencies (LEAs) are finding Lawfully Intercept (LI) of private communication of users to be harder and harder to carry out in practice.

Key escrow was proposed as a remedy for the increasing difficulty for LI, based upon the caller escrowing the session keys needed to decrypt an encrypted communication session with a Trusted Third Party (TTP) who can provide the LEA with the necessary keys after proper authorization. However, key escrow was not accepted as a viable solution to this problem because it adds additional security vulnerabilities and due to potential risks caused by an unethical employee of the key escrow agent (or a law enforcement agency that has access to the session key(s)). Unfortunately, an employee of the TTP or the LEA can misuse the escrowed keys to forge session contents – as these are the same key(s) as the user used for this session. A viable solution to this problem is need to foster acceptance of key escrow while facilitating LI.

7.1 Summary of the thesis results

This thesis project focused on a proposal, implementation, and evaluation of a model that allows key escrow to be a viable means to facilitate lawful interception while rendering fabrication of the call content detectable. The minisip SIP user agent has been extended to escrow the session master key along with other security parameters after a successful SIP session. An extensive analysis has been performed in order to identify an optimized set of parameters that would be required by the LEA to generate the session keys from the master key; this is the set of information that is escrowed with the escrow agent. Additionally, a simple key escrow agent has been implemented using an Apache web server with a MySQL database.

In order to detect forgery of recorded session content, asymmetric cryptography has been used to create a digital signature for a block of SRTP packets. This signed hash value is sent via the SRTCP/RTCP control path as detailed in chapter 5.

A rigorous evaluation of the implemented model has been performed as was detailed in chapter 6. The evaluation result suggests that for different block sizes the overhead due to the signing operation is roughly constant and that this is the dominant factor in the additional processing that is required for the proposed model. Although there is additional total delay due to the time required to compute the hash over the SRTP packet, this time is quite small in comparison to the signing time.

While using a small block size provides finer granularity forgery detection, this occurs at the cost of more frequent asymmetric cryptographic operations and increased network traffic. The additional processing and additional data have been carefully measured and can be used to decide on what is the most appropriate block size for a given implementation and usage scenario. As a result of the analysis of the experiments that we have carried out, we can answer some of the open questions that we presented in Chapter 1:

Q1: How many SRTP packets should be grouped together?

Using a block size of 64 enables the detection of forgery of contents to a 1.28 second interval while requiring no additional SRTP/RTCP packets to be sent (but the packet that will be sent will be some what larger). The computational time is 0.3% of the total elapsed and 136 additional bytes per second of traffic. This might also be possible with a block size of 32 in some settings if the amount of SR and RR data is small and the PATH MTU is 1500 bytes.

Q2: What is a suitable rate for computing the signed hashes?

With a block size of 64, a signed hash would be calculated roughly every 1.28 seconds – hence the computation time required for signing is roughly 0.3% of this time interval. Hence the CPU resources (on the processor that was used for the measurements) are minimal.

Q3: Should the number of packets that are grouped together be computed adaptively based upon the rate at which the sender can compute and sign the hashes?

Yes, in the case of a processor with a slower CPU it may be necessary to reduce the rate at which signed hashes are computed.

Q4: Is there any minimum number of SRTP packets that should be grouped together?

No. If there is sufficient bandwidth and computational power available, then a block size of 1 is feasible with the processor used for these measurements, but the signing operation will take ~20% of the CPU's resources. With a slower CPU a block size of 1 may be infeasible if the device is to support any other tasks.

Q5: Is there any maximum number of SRTP packets that should be grouped together?

There seems to be little reason to use a block size of larger than 256, unless a very slow CPU is used to compute the signature.

7.2 Future work

This thesis considered only a single escrow agent with whom user agents escrow their session master key. However, multiple escrow agents might need to be implemented to allow a user agent to deposit parts of the session master key with separate escrow agents. This is necessary to both reduce the risk of an escrow agent misusing their knowledge of the escrowed information and to reduce the risks of an escrow agent being unavailable for key recovery (for example, due to network partitioning or financial failure of the escrow agent).

However, utilizing multiple escrow agents would require a suitable mechanism to be implemented so that the LEA could reconstruct the secret from the information provided by the escrow agents (i.e., form the whole session master key). This will be the topic of a new thesis project starting in January 2010.

There is also a need for the LEA to be able to determine which escrow agents need to be contacted to learn a session key (or in the case of multiple escrow agents with only part of the key – the set of escrow agents that need to be contacted). This will also be part of the new thesis project mentioned above.

In the current implementation the signed hash value are sent via the RTCP path as soon as they are produced, by sending a UDP packet containing the signed hash and a sequence number. The analysis presented in the previous chapter suggests delaying sending the signed hash so that it can be piggy-backed inside the RTCP sender Report (SR)/receiver Report (RR). This could reduce the number of packets that need to be sent hence avoiding some unnecessary network overhead. However, this remains to be implemented. Unfortunately, SRTCP is not currently implemented in the Minisip code, thus as part of the future work SRTCP should be implemented along with the possibility to send the signed hash values inside the SRTCP reports.

The user agent uses a SSL/TLS tunnel to escrow the session master key with the escrow agent. A theoretical analysis has been performed to examine if the tunnel should be kept open for subsequent call(s) or if each successful call should open a new tunnel and after escrowing the key the tunnel would be closed. This needs to be examined in practice and if found feasible, then some means of knowing how long the tunnel should be kept open for a given calling pattern should be determined.

The escrow system should be evaluated to measure the time and communication required to authenticate the registered user to the TTP and to deposit a key, as this may set an upper rate limit on the rate at which keys can be stored at the TTP.

We have not answered the question: Is there a problem of too frequent signing, leading to a leaking of bits of the sender’s private key? Hence this question also remains for future work.

References

[1] Romanidis Evripidis, “Lawful Interception and Countermeasures: In the era of Internet Telephony”, Masters thesis, 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

[2] F. Baker, B. Foster, and C. Sharp, “Cisco architecture for lawful intercept in IP networks”, Internet Engineering Task Force, Network Working Group, Request for Comments 3924, October 2004. http://www.ietf.org/rfc/rfc3924.txt

[3] J. Rosenberg et al., “SIP: Session Initiation Protocol”, IETF, Network Working Group, RFC 3261, June 2002, http://www.ietf.org/rfc/rfc3261.txt, Last accessed on 07-09-2009.

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

[5] L. Mitrou, Communications Data Retention: A Pandora's Box for Rights and Liberties?, Chapter 20 in *Digital Privacy: Theory, Technologies, and Practices*, edited by Alessandro Acquisti, Stefanos Gritzalis, Costos Lambrinoudakis, Sabrina di Vimercati, Auerbach Publications;, 1 edition, December 22, 2007, pages 419-434. ISBN-10: 1420052179 and ISBN-13: 978-1420052176

[6] “White Paper – Lawful Intercept Overview”, Newport Networks, http://www.newport-networks.com/cust-docs/87-Lawful-intercept.pdf , last accessed 17/06/2009

[7] Sveriges riksdag (Swedish Parlament), En anpassad försvarsunderrättelseverksamhet, Proposition 2006/07:63, Motionstid slutar: 2007-03-27, http://www.riksdagen.se/webbnav/?nid=3120&doktyp=proposition&bet=2006/07:63

[8] (Sveriges) Försvarsdepartementet (Swedish Department of Defense), Uppföljning av lagen om signalspaning I försvarsunderrättelseverksamhet, Directive 2009:10 2009-02-12. http://www.riksdagen.se/webbnav/index.aspx?nid=3260&dok_id=DIR2009:10

[9] United States Department of Justice, Federal Bureau of Investigation and Drug Enforcement Administration, Joint Petition [to US FCC] for Rulemaking to Resolve Various Outstanding Issues Concerning the Implementation of the Communications Assistance for Law Enforcement Act, 10 March, 2004 http://www.steptoe.com/publications/FBI_Petition_for_Rulemaking_on_CALEA.pdf

[10] Steven Bellovin, Matt Blaze, Ernest Brickell, Clinton Brooks, Vinton Cerf, Whitfield Diffie, Susan Landau, Jon Peterson, and John Treichler, “Security Implications of Applying the Communications Assistance to Law Enforcement Act to Voice over IP”, June 13, 2006 http://radiata.cs.columbia.edu/~smb/papers/CALEAVOIPreport.pdf

[11] Jalal Feghhi, Jalil Feghhi, Peter Williams “Digital Certificates Applied Internet Security” Addison Wesely Longman, Inc, 1998, 480 pages. ISBN 0-201-30980-7.

[12] "Public Key Infrastructure Overview", SUN Microsystems, http://www.sun.com/blueprints/0801/publickey.pdf , last visited 18/06/2009

[13] Charlie Kaufman, Radia Perlman, and Mike Speciner "Network Security-Private communication in a public world", Prentice Hall PTR, 2002, second edition

[14] Chen Jie, "Design Alternatives and Implementation of PKI Functionality for VoIP", Masters thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, 2006, http://minisip.org/publications/Thesis_Jie_jun2006.pdf

[15] 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/e12138381ec03c1c60129 40f8d0a3136/OSS1998-E1-08.pdf, Last accessed November 14, 2009.

[16] D.E. Denning and D.K. Branstad, *A taxonomy for key escrow encryption systems*. Communications of the ACM 39 No. 3 (March 1996) 34-40.

[17] "Libcurl-the multiprotocol file transfer library|" http://curl.haxx.se/libcurl/ Last accessed September 09, 2009.

[18] "Key Escrow- Wikipedia, the free encyclopaedia" http://en.wikipedia.org/wiki/Key_escrow, last visited 19/06/2009

[19] "The Clipper Chip" http://epic.org/crypto/clipper/ last visited 12/06/09

[20] "Clipper chip", Wikipedia http://en.wikipedia.org/wiki/Clipper_chip#cite_note-3 , last visited 12/06/2009

[21] Barbara Simons, Regarding S.1726, the "Promotion of Commerce On-Line in the Digital Era (Pro-CODE) Act", Testimony before the Subcommittee on Science, Technology and Space Senate Commerce, Science, and Transporation Committee of the U.S. Senate, 26 June 1996 http://usacm.acm.org/usacm/crypto/simons_senate_testimony.html

[22] Electronic Frontier Foundation, "Key Escrow, Key Recovery, Trusted Third Parties & Govt. Access to Keys", web page, last accessed 4 Aug 2009, http://w2.eff.org/Privacy/Key_escrow/

[23] "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.cdt.org/crypto/risks98/ , last visited 15/07/2009

[24] Eric Verheul, Bert-Jaap Koops, and Henk van Tilborg, "Binding cryptography — A fraud-detectible alternative to Key-Escrow proposals", Computer Law & Security Report, Volume 13, Issue 1, January-February 1997, Pages 3-14. doi:10.1016/S0267-3649(97)81186-7

[25] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, "The Secure Real-time Transport Protocol (SRTP)", Internet Engineering Task Force (IETF), Network Working Group, Request for Comments: 3711, March 2004. http://www.ietf.org/rfc/rfc3711.txt

[26] Elisabetta Carrara, "Security for IP Multimedia Applications over Heterogeneous Networks", Licentiate thesis, Royal Institute of Technology (KTH), Stockholm, Sweden, August 31 2004, http://web.it.kth.se/~carrara/licproposal.pdf

[27] 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

[28] E. Rescorla, “Diffie-Hellman Key Agreement Method”, RFC 2631, Network Working Group, http://www.ietf.org/rfc/rfc2631.txt(diffie-helmann). Last accessed on 23-09-2009.

[29] MiniSIP homepage. "http://www.minisip.org", last visited 12/07/2009

[30] 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

[31] 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.

[32] Christian Hett, Nicolai Kuntze, and Andreas U. Schmidt, "Security and Non-repudiation for Voice over IP Conversation ", ISSA 2006 From Insight to Foresight Conference, Sandton, South Africa, 5th-7th July 2006, http://arxiv.org/abs/cs.CR/0606068 , last visited 12/07/09

[33] Christian Hett, Nicolai Kuntze, and Andreas U. Schmidt, “Non-repudiation of Voice-over-IP conversations with chained digital signatures”, Poster, Fraunhofer Institute for Secure Information Technology (SIT), 30 June 2006, 8 panels http://andreas.schmidt.novalyst.de/docs/POSTER_Non-repudiation%20of%20Voice-over-IP%20conversations.pdf

[34] Stephen D. Guhl, “An architecture for obtaining VoIP session encryption keys in a CALEA compliant network”, Doctoral dissertation, Wichita State University, College of Engineering, Dept. of Electrical and Computer Engineering, May-2008. http://hdl.handle.net/10057/1950

[35] Dimitris Zisiadis, Spyros Kopsidasa, and Leandros Tassiulasa. “VIPSec defined”, Computer Networks, Volume 52, Issue 13, 17 September 2008, Pages 2518-2528. doi:10.1016/j.comnet.2008.04.020

[36] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman, “MIKEY: Multimedia Internet KEYing ”, RFC 3830, IETF, Network Working Group, August 2004, http://www.faqs.org/rfcs/rfc3830.html.

[37] M. Bellare and R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication”, Lecture Notes in Computer Science, Springer-Verlag, pages 1-15, 1996

[38] Ralf S. Engelschall. OpenSSL Project, 2002. http://www.openssl.org/, last visited September 2009.

[39] R. L. Rivest, A. Shamir, L. M. Adleman, “Cryptographic communications system and method”, US Patent number 4405829, September 1983

[40] D. Richard Kuhn, Thomas J. Walsh, and Steffen Fries, “Security Considerations for Voice Over IP Systems”, National Institute of Standards and Technology, Special Publication 800-58, January 2005 . http://www.commserv.ucsb.edu/reference/background/VOIP_security_considerations.pdf

[41] Xiao Wu, SIP on an Overlay Network, Masters thesis, Royal Institute of Technology (KTH), School of Information and Communication Technology, TRITA-ICT-EX-2009:105, September 2009 http://web.it.kth.se/~maguire/DEGREE-PROJECT-REPORTS/090915-XiaoWu-with-cover.pdf

[42] Posix time, Boost c++ library, http://www.boost.org/doc/libs/1%5F38%5F0/doc/html/date%5Ftime/posix%5Ftime.html, last visited November 2009.

Appendices

A. Script to enable Apache2 web server with SSL capability

```
#!/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
```

B. HMAC_SHA hashing time for 50 test runs

Run	HMAC_SHA hashing time in Microseconds								
	Block Size								
	01	*08*	*16*	*32*	*64*	*128*	*256*	*512*	*1024*
1	13	34	37	49	66	92	171	342	644
2	8	15	35	44	62	89	182	324	639
3	8	14	33	44	61	103	181	324	639
4	12	14	35	45	62	104	168	324	648
5	9	15	20	46	62	109	168	325	638
6	8	13	34	44	61	103	168	325	639
7	11	15	33	43	61	89	168	324	639
8	9	14	19	45	61	89	169	324	639
9	9	15	20	43	62	89	168	324	639
10	11	15	32	47	62	90	169	329	649
11	9	15	19	43	62	90	169	324	648
12	8	15	20	45	50	89	168	324	654
13	11	15	21	45	50	89	167	324	648
14	8	14	34	45	50	89	167	324	675
15	8	14	33	49	50	90	167	324	653
16	9	15	33	43	50	89	168	324	648
17	12	13	21	43	49	90	167	324	648
18	8	14	20	43	49	89	171	324	648
19	8	13	21	44	50	90	167	333	648
20	10	13	33	48	50	90	167	324	649
21	8	14	20	30	49	89	174	324	651
22	8	13	33	29	50	89	167	324	650
23	9	14	20	30	50	90	167	323	638
24	8	15	20	30	49	89	167	323	655
25	8	14	20	29	50	89	167	324	648
26	10	13	19	30	51	90	168	323	654
27	8	15	19	29	51	89	168	324	648
28	9	14	37	30	50	89	167	323	648
29	9	16	33	29	50	89	168	324	647
30	8	14	33	30	50	90	179	328	649
31	9	15	35	30	60	90	168	324	648
32	9	15	31	30	50	89	168	324	648
33	9	15	20	30	49	89	167	325	649
34	9	15	20	30	50	89	168	325	649
35	9	15	19	30	50	89	168	325	655
36	10	15	20	30	50	89	167	324	649
37	8	14	33	30	51	90	167	325	668
38	9	15	20	29	50	91	167	325	648
39	8	15	21	29	51	89	167	326	648

40	8	15	21	31	50	90	167	324	648
41	8	16	21	30	50	90	167	334	648
42	9	15	21	30	49	95	167	325	648
43	8	26	21	30	50	89	167	326	648
44	9	28	20	30	50	89	167	325	649
45	9	26	21	29	50	89	168	324	648
46	8	29	20	30	49	90	167	324	653
47	9	33	20	29	50	89	167	324	648
48	8	28	20	30	50	89	169	324	668
49	9	25	22	30	50	89	167	324	640
50	8	28	21	30	50	89	167	332	639

C. RSA signing time for 50 test runs

Run	Actual Signing time in Milliseconds								
	Block Size								
	01	***08***	***16***	***32***	***64***	***128***	***256***	***512***	***1024***
1	3.45	3.468	3.495	3.426	3.486	3.475	3.473	3.475	3.454
2	3.509	3.469	3.511	3.509	3.477	3.477	3.426	3.462	3.535
3	3.38	3.472	3.477	3.504	3.477	3.458	3.479	3.471	3.534
4	3.476	3.468	3.547	3.512	3.515	3.442	3.502	3.487	3.472
5	3.407	3.469	3.456	3.504	3.474	3.492	3.502	3.472	3.468
6	3.461	3.561	3.499	3.417	3.449	3.446	3.493	3.492	3.448
7	3.506	3.433	3.46	3.468	3.498	3.446	3.494	3.486	3.453
8	3.405	3.46	3.455	3.451	3.484	3.443	3.483	3.544	3.458
9	3.398	3.444	3.481	3.507	3.464	3.438	3.488	3.507	3.482
10	3.478	3.478	3.434	3.521	3.461	3.436	3.5	3.495	3.483
11	3.426	3.483	3.498	3.467	3.476	3.432	3.467	3.493	3.465
12	3.398	3.479	3.456	3.494	3.469	3.442	3.461	3.507	3.449
13	3.411	3.461	3.458	3.548	3.463	3.459	3.486	3.453	3.466
14	3.39	3.469	3.505	3.48	3.48	3.463	3.487	3.467	3.452
15	3.45	3.454	3.444	3.493	3.459	3.44	3.473	3.465	3.548
16	3.404	3.42	3.488	3.493	3.464	3.425	3.482	3.474	3.492
17	3.449	3.466	3.471	3.475	3.457	3.452	3.641	3.491	3.499
18	3.402	3.44	3.446	3.463	3.468	3.453	3.484	3.47	3.462
19	3.446	3.432	3.51	3.475	3.476	3.458	3.461	3.45	3.459
20	3.464	3.468	3.443	3.473	3.441	3.449	3.467	3.47	3.488
21	3.452	3.45	3.504	3.45	3.472	3.444	3.463	3.461	3.473
22	3.424	3.466	3.458	3.482	3.455	3.463	3.479	3.474	3.493
23	3.428	3.448	3.518	3.447	3.473	3.453	3.747	3.448	3.48
24	3.383	3.44	3.497	3.449	3.485	3.482	3.491	3.472	3.475
25	3.41	3.47	3.436	3.482	3.451	3.479	3.488	3.457	3.47
26	3.424	3.458	3.49	3.464	3.483	3.447	3.474	3.47	3.479
27	3.438	3.428	3.453	3.467	3.454	3.459	3.48	3.495	3.551
28	3.398	3.433	3.496	3.487	3.441	3.439	3.472	3.464	3.472
29	3.411	3.487	3.468	3.459	3.457	3.454	3.494	3.489	3.477
30	3.445	3.449	3.476	3.476	3.435	3.46	3.5	3.462	3.461
31	4.243	4.298	4.351	4.325	4.327	4.324	4.354	4.358	4.583
32	3.475	3.473	3.47	3.464	3.448	3.485	3.492	3.509	3.492
33	3.443	3.439	3.507	3.459	3.472	3.468	3.496	3.466	3.486
34	3.453	3.422	3.469	3.48	3.482	3.482	3.496	3.488	3.485
35	3.407	3.464	3.505	3.463	3.493	3.425	3.507	3.459	3.476
36	3.453	3.417	3.445	3.492	3.492	3.459	3.491	3.545	3.48
37	3.376	3.457	3.47	3.461	3.499	3.466	3.481	3.474	3.47
38	3.399	3.458	3.455	3.476	3.48	3.46	3.488	3.499	3.452
39	3.406	3.461	3.462	3.467	3.51	3.466	3.471	3.519	3.472
40	3.41	3.469	3.48	3.447	3.487	3.49	3.466	3.516	3.477

41	3.435	3.46	3.451	3.489	3.49	3.478	3.465	3.486	3.484
42	3.433	3.433	3.463	3.485	3.493	3.467	3.495	3.47	3.468
43	3.456	3.441	3.448	3.496	3.667	3.449	3.487	3.446	3.461
44	3.443	3.464	3.491	3.478	3.483	3.446	3.459	3.48	3.474
45	3.458	3.469	3.485	3.462	3.501	3.446	3.484	3.445	3.492
46	3.437	3.445	3.451	3.438	3.528	3.469	3.473	3.449	3.495
47	3.407	3.472	3.45	3.475	3.498	3.465	3.477	3.456	3.501
48	3.423	3.454	3.478	3.454	3.485	3.493	3.458	3.508	3.471
49	3.436	3.477	3.477	3.452	3.451	3.459	3.483	3.518	3.486
50	3.414	3.48	3.472	3.468	3.497	3.46	3.483	3.471	3.513

D.RSA signing time for block size closer to 128 to find local minima

Run	Actual Signing time in Milliseconds								
	Block Size								
	120	*124*	*127*	*128*	*129*	*130*	*131*	*132*	*136*
1	3.45	3.44	3.443	3.463	3.456	3.461	3.453	3.47	3.458
2	3.453	3.481	3.485	3.493	3.449	3.516	3.444	3.481	3.494
3	3.463	3.465	3.506	3.453	3.451	3.475	3.463	3.527	3.465
4	3.445	3.495	3.483	3.443	3.499	3.493	3.517	3.485	3.463
5	3.464	3.473	3.518	3.473	3.45	3.486	3.461	3.483	3.516
6	3.478	3.462	3.524	3.442	3.459	3.471	3.491	3.492	3.472
7	3.462	3.516	3.482	3.437	3.466	3.468	3.459	3.483	3.472
8	3.487	3.494	3.491	3.451	3.484	3.455	3.459	3.53	3.482
9	3.492	3.484	3.492	3.447	3.464	3.452	3.448	3.484	3.473
10	3.485	3.499	3.485	3.431	3.459	3.485	3.458	3.521	3.49
11	3.509	3.487	3.501	3.445	3.464	3.451	3.478	3.49	3.493
12	3.528	3.46	3.479	3.439	3.467	3.425	3.49	3.478	3.63
13	3.429	3.509	3.5	3.426	3.509	3.44	3.455	3.487	3.444
14	3.503	3.465	3.477	3.448	3.469	3.451	3.468	3.487	3.459
15	3.521	3.46	3.475	3.452	3.463	3.482	3.467	3.489	3.491
16	3.529	3.484	3.468	3.449	3.47	3.478	3.465	3.48	3.455
17	3.451	3.484	3.467	3.476	3.466	3.463	3.462	3.488	3.46
18	3.463	3.483	3.476	3.445	3.473	3.493	3.458	3.472	3.461
19	3.505	3.544	3.478	3.434	3.461	3.481	3.465	3.486	3.448
20	3.443	3.473	3.477	3.456	3.459	3.478	3.458	3.495	3.491
21	3.487	3.46	3.485	3.428	3.471	3.479	3.462	3.489	3.473
22	3.471	3.527	3.47	3.494	3.466	3.489	3.477	3.48	3.467
23	3.464	3.476	3.509	3.454	3.46	3.479	3.456	3.488	3.496
24	3.476	3.445	3.504	3.435	3.477	3.483	3.449	3.482	3.46
25	3.497	3.5	3.49	3.48	3.472	3.479	3.874	3.484	3.471
26	3.462	3.475	3.492	3.446	3.461	3.473	3.479	3.512	3.478
27	3.514	3.46	3.498	3.435	3.489	3.494	3.475	3.498	3.433
28	3.502	3.498	3.496	3.429	3.462	3.478	3.463	3.485	3.472
29	3.481	3.466	3.493	3.443	3.465	3.42	3.445	3.493	3.467
30	4.346	3.455	3.48	3.448	3.46	3.486	3.441	3.489	3.47
31	3.482	4.342	4.344	4.378	4.32	4.334	4.35	4.471	4.345
32	3.44	3.476	3.504	3.449	3.459	3.495	3.456	3.485	3.453
33	3.5	3.44	3.491	3.455	3.457	3.475	3.483	3.495	3.499
34	3.451	3.482	3.508	3.446	3.462	3.479	3.469	3.471	3.434
35	3.484	3.49	3.489	3.45	3.644	3.506	3.457	3.49	3.458
36	3.454	3.472	3.502	3.47	3.459	3.501	3.458	3.455	3.476
37	3.468	3.457	3.486	3.446	3.453	3.491	3.454	3.489	3.462
38	3.476	3.487	3.492	3.452	3.468	3.478	3.461	3.478	3.475
39	3.462	3.508	3.49	3.428	3.432	3.483	3.457	3.497	3.46

40	3.474	3.451	3.477	3.449	3.439	3.484	3.461	3.469	3.53
41	3.482	3.488	3.435	3.454	3.414	3.481	3.46	3.496	3.511
42	3.478	3.478	3.469	3.449	3.435	3.503	3.467	3.469	3.524
43	3.469	3.452	3.461	3.436	3.442	3.482	3.457	3.495	3.507
44	3.454	3.487	3.471	3.448	3.448	3.482	3.463	3.469	3.501
45	3.455	3.468	3.459	3.438	3.451	3.479	3.458	3.492	3.499
46	3.52	3.472	3.488	3.44	3.442	3.491	3.445	3.468	3.518
47	3.453	3.482	3.491	3.432	3.447	3.454	3.457	3.48	3.526
48	3.484	3.474	3.493	3.431	3.435	3.456	3.473	3.474	3.483
49	3.465	3.451	3.507	3.428	3.448	3.457	3.458	3.488	3.456
50	3.461	3.481	3.472	3.447	3.44	3.448	3.444	3.466	3.466

E. Total Delay (Signing +hashing + RTCP sending) time for 50 test runs

Run	Total time(Signing + Hashing + Sending) in Milliseconds								
	Block Size								
	01	*08*	*16*	*32*	*64*	*128*	*256*	*512*	*1024*
1	4.466	3.653	3.633	3.606	3.666	3.763	3.743	3.919	4.299
2	3.534	3.641	3.644	3.611	3.676	3.676	3.758	3.918	4.293
3	3.506	3.612	3.616	3.631	3.659	3.692	3.784	3.936	4.246
4	3.508	3.69	3.683	3.615	3.769	3.773	3.763	3.986	4.276
5	3.486	3.605	3.556	3.613	3.702	3.741	3.783	3.924	4.272
6	3.592	3.616	3.575	3.618	3.669	3.709	3.754	3.926	4.275
7	3.523	3.606	3.616	3.61	3.7	3.69	3.749	3.908	4.259
8	3.521	3.618	3.598	3.639	3.679	3.707	3.789	3.938	4.281
9	3.574	3.724	3.591	3.687	3.645	3.731	3.793	3.919	4.25
10	3.462	3.603	3.597	3.587	3.752	3.726	3.772	3.938	4.268
11	3.498	3.636	3.586	3.611	3.698	3.699	3.754	3.929	4.256
12	3.531	3.621	3.626	3.62	3.651	3.71	3.783	3.935	4.276
13	3.478	3.62	3.599	3.637	3.673	3.737	3.784	3.908	4.268
14	3.459	3.68	3.595	3.624	3.665	3.712	3.752	3.921	4.281
15	3.474	3.63	3.591	3.626	3.686	3.704	3.756	3.94	4.265
16	3.564	3.595	3.619	3.612	3.645	3.72	3.787	3.937	4.331
17	3.479	3.652	3.598	3.604	3.703	3.717	3.778	3.912	4.259
18	3.503	3.614	3.604	3.609	3.662	3.714	3.778	3.935	4.269
19	3.542	3.657	3.579	3.602	3.696	3.741	3.752	3.936	4.269
20	3.471	3.627	3.618	3.633	3.748	3.685	3.805	3.939	4.26
21	3.512	3.605	3.556	3.628	3.645	3.679	3.773	3.923	4.264
22	3.521	3.613	3.627	3.642	3.636	3.698	3.739	3.914	4.244
23	3.478	3.629	3.608	3.628	3.67	3.694	3.772	3.897	4.259
24	3.517	3.693	3.597	3.571	3.644	3.688	3.762	3.916	4.234
25	3.568	3.61	3.566	3.607	3.693	3.697	3.744	3.923	4.254
26	3.526	3.599	3.589	3.611	3.678	3.711	3.751	3.917	4.253
27	3.483	3.63	3.607	3.53	3.649	3.698	3.75	3.921	4.232
28	3.555	3.591	3.604	3.625	3.652	3.689	3.788	3.916	4.26
29	3.491	3.713	3.611	3.601	3.666	3.694	3.777	3.918	4.246
30	3.51	3.631	3.623	3.619	3.649	3.707	3.77	3.935	4.214
31	4.381	4.471	4.447	4.462	4.52	4.574	4.64	4.759	5.074
32	3.552	3.682	3.766	3.572	3.69	3.705	3.782	3.941	4.229
33	3.56	3.582	3.58	3.616	3.652	3.795	3.779	3.925	4.24
34	3.476	3.625	3.606	3.601	3.653	3.679	3.751	3.932	4.245
35	3.561	3.638	3.58	3.588	3.673	3.673	3.746	3.951	4.257
36	3.484	3.588	3.594	3.61	3.644	3.711	3.781	3.935	4.235
37	3.525	3.623	3.597	3.624	3.679	3.709	3.78	3.934	4.222
38	3.533	3.593	3.593	3.63	3.67	3.713	3.755	3.929	4.241
39	3.481	3.678	3.562	3.635	3.654	3.7	3.763	3.891	4.279

40	3.514	3.602	3.548	3.64	3.69	3.671	3.802	3.893	4.255
41	3.565	3.588	3.557	3.603	3.673	3.694	3.832	3.927	4.251
42	3.483	3.686	3.596	3.604	3.638	3.698	3.804	3.898	4.285
43	3.491	3.746	3.601	3.613	3.672	3.68	3.765	3.926	4.26
44	3.534	3.653	3.609	3.643	3.7	3.684	3.799	3.891	4.251
45	3.505	3.605	3.645	3.602	3.636	3.671	3.767	3.908	4.282
46	3.502	3.642	3.614	3.646	3.653	3.66	3.787	3.903	4.271
47	3.512	3.6	3.678	3.631	3.657	3.684	3.786	3.922	4.275
48	3.541	3.641	3.58	3.612	3.671	3.682	3.779	3.954	4.264
49	3.557	3.669	3.587	3.599	3.669	3.697	3.758	3.927	4.285
50	3.468	3.609	3.674	3.601	3.659	3.713	3.783	3.975	4.24

F. Detailed of the CPU used by our User Agent

```
processor       : 0
vendor_id       : GenuineIntel
cpu family      : 15
model           : 4
model name      : Intel(R) Pentium(R) 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(R) Pentium(R) 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. Schema definition of Escrow Database

```
-- phpMyAdmin SQL Dump
-- version 3.2.0
-- http://www.phpmyadmin.net
--
-- Host: localhost
-- Generation Time: Dec 02, 2009 at 10:06 PM
-- Server version: 5.1.36
-- PHP Version: 5.2.11

SET SQL_MODE="NO_AUTO_VALUE_ON_ZERO";

--
-- Database: `escrowdatabase`
--

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

--
-- 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=3 ;

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

CREATE TABLE IF NOT EXISTS `sipmasterkey` (
  `id` int(11) NOT NULL AUTO_INCREMENT,
  `userid` text NOT NULL,
  `key` text NOT NULL,
  `rand` text NOT NULL,
  `csbID` text NOT NULL,
  `signedhash` text NOT NULL,
  `date` datetime DEFAULT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM  DEFAULT CHARSET=latin1 AUTO_INCREMENT=621 ;
```

H. 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;
}
```

I. Important Apache configuration files (Two files)

```
##########################################################################
# /etc/apache2/httpd.conf
##########################################################################

### Global Environment #####

# run under this user/group id
Include /etc/apache2/uid.conf

# - how many server processes to start (server pool regulation)
# - usage of KeepAlive
Include /etc/apache2/server-tuning.conf

# ErrorLog: The location of the error log file.
ErrorLog  /var/log/apache2/error_log

# generated from APACHE_MODULES in /etc/sysconfig/apache2
Include /etc/apache2/sysconfig.d/loadmodule.conf

# IP addresses / ports to listen on
Include /etc/apache2/listen.conf

# predefined logging formats
Include /etc/apache2/mod_log_config.conf

# generated from global settings in /etc/sysconfig/apache2
Include /etc/apache2/sysconfig.d/global.conf

# optional mod_status, mod_info
Include /etc/apache2/mod_status.conf
Include /etc/apache2/mod_info.conf

# optional cookie-based user tracking
# read the documentation before using it!!
Include /etc/apache2/mod_usertrack.conf

# configuration of server-generated directory listings
Include /etc/apache2/mod_autoindex-defaults.conf

# associate MIME types with filename extensions
TypesConfig /etc/apache2/mime.types
DefaultType text/plain
Include /etc/apache2/mod_mime-defaults.conf

# set up (customizable) error responses
Include /etc/apache2/errors.conf
```

```
# global (server-wide) SSL configuration, that is not specific to  any virtual host
Include /etc/apache2/ssl-global.conf

# forbid access to the entire filesystem by default
<Directory />
   Options None
   AllowOverride None
   Order deny,allow
   Deny from all
</Directory>

# use .htaccess files for overriding,
AccessFileName .htaccess
# and never show them
<Files ~ "^\.ht">
   Order allow,deny
   Deny from all
</Files>

# List of resources to look for when the client requests a directory
DirectoryIndex index.html index.html.var

### 'Main' server configuration
Include /etc/apache2/default-server.conf


Include /etc/apache2/sysconfig.d/include.conf


### Virtual server configuration
Include /etc/apache2/vhosts.d/*.conf


##########################################################################
# /etc/apache2/vhost.d/vhost-ssl.conf
##########################################################################

<IfDefine SSL>
<IfDefine !NOSSL>

##
## SSL Virtual Host Context
##

<VirtualHost _default_:443>

   # General setup for the virtual host
   DocumentRoot "/srv/www/htdocs"
   ServerName ccsmoto:443
   ServerAdmin sakhawat23@gmail.com
```

```
    ErrorLog /var/log/apache2/error_log
    TransferLog /var/log/apache2/access_log

    #   SSL Engine Switch:
    #   Enable/Disable SSL for this virtual host.
    SSLEngine on

    #   SSL Cipher Suite:
    #   List the ciphers that the client is permitted to negotiate.

SSLCipherSuiteALL:!ADH:!EXPORT56:RC4+RSA:+HIGH:+MEDIUM:+LOW:+SS
Lv2:+EXP:+eNULL

    #   Server Certificate:
      SSLCertificateFile /etc/apache2/ssl.crt/server.crt

    #   Server Private Key:
      SSLCertificateKeyFile /etc/apache2/ssl.key/server.key



    #   SSL Engine Options:
      #SSLOptions +FakeBasicAuth +ExportCertData +CompatEnvVars
+StrictRequire
    <Files ~ "\.(cgi|shtml|phtml|php3?)$">
       SSLOptions +StdEnvVars
    </Files>
    <Directory "/srv/www/cgi-bin">
       SSLOptions +StdEnvVars
    </Directory>

    #   SSL Protocol Adjustments:
      SetEnvIf User-Agent ".*MSIE.*" \
       nokeepalive ssl-unclean-shutdown \
       downgrade-1.0 force-response-1.0

    #   Per-Server Logging:
      CustomLog /var/log/apache2/ssl_request_log   ssl_combined

</VirtualHost>

</IfDefine>
</IfDefine>
```

TRITA-ICT-EX-2010:1