Royal Institute of Technology (KTH)
Dept. of Microelectronics and Information Technology (IMIT)
Stockholm, Sweden

Universidad Politécnica de Madrid (UPM)
Facultad de Informática (FI)
Madrid, Spain

Secure Mobile Voice over IP

Master of Science Thesis

June 2003

Student:	Israel M. Abad Caballero
Supervisor at KTH / IT:	Professor Gerald Q. Maguire Jr.
Supervisor at FI UPM:	Professor Pedro Gómez−Vilda

Abstract

Voice over IP (VoIP) can be defined as the ability to make phone calls and to send faxes (i.e., to do everything we can do today with the Public Switched Telephone Network, PSTN) over IP−based data networks with a suitable quality of service and potentially a superior cost/benefit ratio.

There is a desire to provide (VoIP) with the suitable security without effecting the performance of this technology. This becomes even more important when VoIP utilizes wireless technologies as the data networks (such as Wireless Local Area Networks, WLAN), given the bandwidth and other constraints of wireless environments, and the data processing costs of the security mechanisms. As for many other (secure) applications, we should consider the security in Mobile VoIP as a chain, where every link, from the secure establishment to the secure termination of a call, must be secure in order to maintain the security of the entire process.

This document presents a solution to these issues, providing a secure model for Mobile VoIP that minimizes the processing costs and the bandwidth consumption. This is mainly achieved by making use of high−throughput, low packet expansion security protocols (such as the Secure Real−Time Protocol, SRTP); and high−speed encryption algorithms (such as the Advanced Encryption Standard, AES).

In the thesis I describe in detail the problem and its alternative solutions. I also describe in detail the selected solution and the protocols and mechanisms this solution utilizes, such as the Transport Layer Security (TLS) for securing the Session Initiation Protocol (SIP), the Real−Time Protocol (RTP) profile Secure Real−Time Protocol (SRTP) for securing the media data transport , and the Multimedia Internet KEYing (MIKEY) as the key−management protocol. Moreover, an implementation of SRTP, called MINIsrtp, is also provided. The oral presentation will provide an overview of these topics, with an in depth examination of those parts which were the most significant or unexpectedly difficult.

Regarding my implementation, evaluation, and testing of the model, this project in mainly focused on the security for the media stream (SRTP). However, thorough theoretical work has also been performed and will be presented, which includes other aspects, such as the establishment and termination of the call (using SIP) and the key−management protocol (MIKEY).

Sammanfattning

Voice over IP (VoIP) kan defineras som förmågan att göra ett telefonsamtal och att skicka fax (eller att göra allting som man idag kan göra över det publika telefonnätet) över ett IP−baserat nätverk med en passande kvalitet och till lägre kostnad, alternativt större nytta.

VoIP måste tillhandahållas med nödvändiga säkerhetstjänster utan att teknikens prestanta påverkas. Detta blir allt viktigare när VoIP används över trådlösa länktekniker (såsom trådlösa lokala nätverk, WLAN), givet dessa länkars begränsade bandbredd och den bearbetningkraft som krävs för att exekvera säkerhetsmekanismerna. Vi måste tänka på VoIPs säkerhet likt en kedja där inte någon länk, från säker uppkoppling till säker nedkoppling, får fallera för att erhålla en säker process.

I detta dokument presenteras en lösning på detta problem och innefattar en säker modell för Mobile VoIP som minimerar bearbetningskostnaderna och bandbreddsutnyttjandet. Detta erhålls huvudsakligen genom utnyttjande av säkerhetsprotokoll med hög genomströmning och låg paketexpansion, såsom "Secure Real−time Protocol" (SRTP), och av krypteringsprotokoll med hög hastighet, såsom "Advanced Encryption Standard" (AES).

I detta dokument beskriver jag problemet och dess alternativa lösningar. Jag beskriver också den valda lösningen och dess protokoll och mekanismer mer detaljerat, till exempel "Transport Layer Security" (TLS) för att säkra "Session Initiation Protocol" (SIP), SRTP för att skydda transporten av data och "Multimedia Internet KEYing" (MIKEY) för nyckelhantering. En implementation av SRTP, kallad MINIsrtp, finns också beskriven.

Beträffande praktiskt arbete och tester av lösningsmodellen har detta projekt fokuserats på skyddandet av datatransporten (SRTP), dess implementation och prestanda. Emellertid har en grundlig teoretisk undersökning genomförts, vilken innefattar andra aspekter såsom telefonsamtalets uppkoppling och nedkoppling (med hjälp av SIP) och valet av passande nyckelhanteringsprotokoll (MIKEY) för att stödja SRTP.

Preface

This work has been performed as a degree project at Telecommunication Systems Laboratory (TSLab), Department of Microelectronics and Information Technology (IMIT), Kungl Tekniska Högskolan (KTH), Stockholm, Sweden, during the period February 2003−June 2003.

I would like to express my sincere thanks to:

Prof. Gerald Maguire, my supervisor at KTH, for his help, his encouragement, his suggestions, and the opportunity he gave me of joining IMIT for developing my degree project;
Jon−Olov Vatn, for all his inestimable help, his patient, his continuous encouragement, his suggestions, and his kindness;
Erik Eliasson, for his help to develop MINIsrtp , his permission to use MINISIP and MIKEY source code, and for his ideas and suggestions for the project;
Prof. Pedro Gómez−Vilda, my supervisor at FI UPM, for his support and the opportunity he gave from my home university of studying at KTH;
all the friends who were near me during my stay in Stockholm, specially Álvaro, Jürgen, and Nacho.

A special thank to my parents, who gave me the opportunity of coming and studying here in Stockholm , and always support my decisions. To my family and friends, that took care of me from Spain during my stay here. To Miguel, my brother, one of my best friends and one of my strongest supports. And specially to Rocío, for all the love she gave me during this year, and without whose continuous support and infinite patient this would not have been possible.

Other acknowledgements:

Petra Rubalcaba (FI UPM); Elisabetta Carrara (Ericsson); Mark Baugher and David McGrew (Cisco Systems, Inc.); Aaron Gifford; Johan Bilien; Jürgen Prokop for his help with Figure 5.5; Professor Jean−Jacques Quisquater and the Microelectronics Laboratory Crypto Group (Université Catholique de Louvain, UCL), for their permission for using Figure 5.3.

Table of Contents

1 Introduction

The integration of Voice over IP (VoIP) into the wireless environments has become the new challenge within the telecommunications world. The limited bandwidth and other constraints present in the wireless environment limit the performance of this technology. Furthermore, the addition of suitable security mechanisms to Mobile VoIP in order to provide the process with the necessary security services limits more that performance.

This document presents a possible solution for Secure Mobile Voice over IP. Two main ideas have been kept in mind while designing this solution. First, we must carefully define the security services which the model is to be provided with. Second, suitable security mechanisms must be selected in order to implement the necessary security services without effecting the performance of the model. Our solution for this issue, provides a secure model for Mobile VoIP that minimizes the processing costs and the bandwidth consumption. This is mainly achieved by making use of high−throughput, low packet expansion protocols; and high−speed encryption algorithms.

We may consider a VoIP call as a three−phase process: establishment, conversation, and termination. The first and the third phases typically make use of a signalling protocol, such as the Session Initiation Protocol (SIP), while the second utilizes the Real−Time Protocol (RTP) to transport the media data. Therefore, this project handles the signalling protocol security and the data transport protocol security independently. The main goal of this document is the description of a suitable solution to achieve this security without effecting the performance of the model.

Mobility aspects of the model are not explicitly considered in this project, although this paper may be one of the bases for future work in this area, since the proposed solution is based on the use of mobile devices operating in wireless environments.

Regarding practical work and tests of the model, this project in mainly focused on the security for the media stream (by using the Secure Real−Time Protocol, SRTP). However, thorough theoretical work has also been performed, which includes other aspects as said above, such as the establishment and termination of the call (using the Session Initiation Protocol, SIP) and the key−management protocol to be used.

This paper is mainly divided into two parts. The first part gives a detailed description of several protocols, mechanisms, and concepts important for the context of the project, while the second part is entirely related to the problem itself, showing some requirements of the model, some alternative solutions, and finally the selected architecture and the rationale.

Regarding the first part, a short introduction to VoIP is given in section 2. The Session Initiation Protocol (SIP) and the Real−Time Protocol (RTP) are briefly described in section 3. Section 4 introduces the reader to the different Security Services we need to provide. These services are implemented by security mechanisms. Probably the most important security mechanism today is cryptography, described in detail in section 5. Furthermore, the cryptographic algorithms used in our model, such as Advanced Encryption Standard (AES) and Hash−Based Message Authentication Code based on Secure Hash Algorithm (HMAC−SHA1) are also described. Section 6 gives a brief introduction to the Public−Key Infrastructures and the use of certificates. Finally, several Security Protocols, such as Transport Layer Security (TLS) and IP Security Architecture (IPSec) are described in section 7.

The second part starts by presenting the problem to be solved in section 8. Section 9 makes an overall presentation of the Secure Mobile VoIP components and requirements. Alternative solutions to secure the model are given in section 10, while section 11 presents our solution and a rationale for it. Section 12 describes in more detail than the previous section our solution for secure SIP by establishing a Public Key Infrastructure (PKI) supporting TLS. The solution selected for securing the media data (Secure Real−Time

Protocol, SRTP) is thoroughly described in section 13 along with its implementation (MINIsrtp). SRTP is a high−throughput security profile of RTP that minimizes the packet expansion. The description of the key−management protocol chosen to support the SRTP sessions (Multimedia Internet KEYing, MIKEY) and a framework for this project are presented in section 14. MIKEY is an efficient key−management protocol specifically oriented to support secure media transport protocols, such as SRTP. Section 15 contains the analysis and evaluation of our solution.

To end with, some conclusions and a summary of future work regarding the project are given in section 16.

Appendix A presents the MINIsrtp source code, Appendix B provides a first approach to a reference implementation for MIKEY messages and payloads, while the acronyms, the notations, and the glossary of this paper are given in Appendix C, Appendix D, and Appendix E, respectively. Finally I provide an index of figures and tables, and the list of references.

2 Voice over IP Overview

Voice over IP has become a very interesting research area within the telecommunications field during the last years, given its advantages regarding low call costs. This report examines its integration into the wireless communications world given the limited bandwidth and other constraints present in this environment. The desire to provide suitable security support is the aim of many researchers nowadays. This sections briefly introduces the reader to VoIP technology.

2.1 Introduction

Voice over IP (also referred to as *Voice over Packet*, *Voice over Internet Protocol*, or simply *VoIP*) consists of several interconnected components that convert a voice signal into a stream of packets on a packet network, and viceversa. Thus, VoIP can be defined as the ability to make phone calls (i.e., to do everything we can do today with the *Public Switched Telephone Network*, PSTN) and to send faxes over data networks with a suitable quality of service and much superior cost/benefit.

A new rich set of advantages and possibilities has emerged with the VoIP technology. Since data traffic has been growing much faster during the last years than telephone traffic, there has been considerable interest in transporting voice over data networks (allowing this voice and fax traffic to travel concurrently with data traffic over a packet data network), rather than the traditional data over voice networks. This fact places the existing telephone capabilities at a significantly lower "total cost of operation". As far as the end users are concerned, a significant example would be the cost savings for long−distance telephone calls, where these users would not be imposed with additional constraints. On the other hand, the increase of their traffic volumes becomes very attractive for the *Internet Service Providers* (ISPs), and the equipment producers now have an opportunity to innovate and compete.

2.2 Components, Protocols, and Standards

Figure 2.1 depicts the infrastructure of a VoIP system. A significant component of the model is the Gateway. The Gateway is in charge of converting the media provided in one type of network to the format required for another type of network.

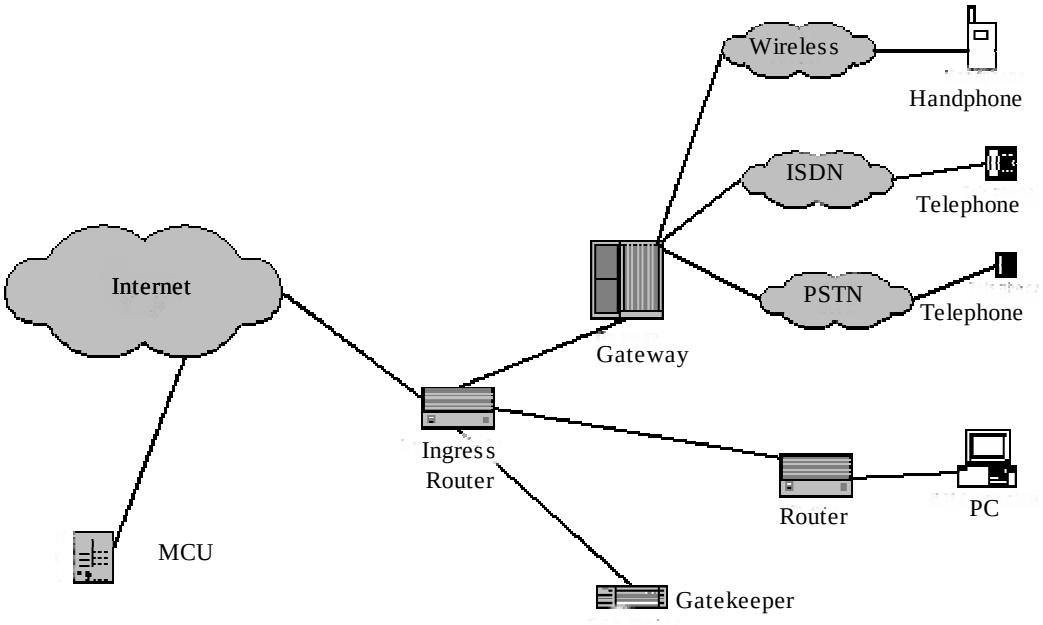

Figure 2.1 VoIP Infrastructure

The voice packets are transported using IP in compliance with a specification for transmitting multimedia (voice, fax, video, and data) across a network. There are several specifications, recommendations, and standards for performing this transmission:

- ITU−T H.323
- *Media Gateway Control Protocol* (MGCP), from level 3, Bellcore, Cisco, and Nortel
- IETF MEGACO/H.GCP
- IETF *Session Initiation Protocol* (SIP)
- ITU−T T.38
- IETF SIGTRAN
- Skinny, from Cisco

SIP is nowadays of special interest. SIP is an IETF standard specified in Request For Comments (RFC) 3261[3], and defines a signaling protocol for creating, modifying, and terminating sessions[1]. Regarding the data transport itself, the most important protocol to handle this is the *Real−Time Protocol* (RTP)[2].

As far as the quality of service (QoS) and performance are concerned, VoIP is a delay−sensitive application, so a well−engineered, end−to−end network is necessary. Issues such as delay, jitter, congestion, packet−loss, and misordered packet arrival must be carefully handled.

1. These sessions are considered exchanges of data between participants, and include Internet telephone calls, multimedia distributions, and multimedia conferences.

The following list summarizes some examples of services provided by a VoIP network according to market requirements:

- Phone−to−phone
- PC−to−phone and phone−to−PC
- fax−to−fax
- fax−to−email and email−to−fax
- Wireless Connectivity
- PC−to−PC

This study is concerned with the PC−to−PC VoIP service, assuming the users have the appropriate software and hardware installed on their PCs (*user agents*, sound card, headsets, etc.).

3 Introduction to SIP and RTP

This section provides a short introduction to the Session Initiation Protocol (SIP) and the Real−Time Protocol (RTP), widely used in VoIP technology. SIP is the most commonly used protocol to create and manage the VoIP media sessions, while RTP is the transport protocol in charge of the transmission of the data in a VoIP session.

3.1 Session Initiation Protocol (SIP)

3.1.1 Introduction

SIP is a signaling protocol used for establishing, modifying, and terminating sessions between users. SIP is defined by IETF as a standard in RFC 3261.

There are many applications of the Internet that require the creation and management of sessions, such as multimedia real−time exchanges, which this project is concerned with. There are various protocols designed to carry this real−time data (voice, video, etc.), such as *Real−Time Protocol* (RTP), and SIP works in concert with these protocols by establishing, managing, and terminating these exchanges.

As described in the SIP standard (RFC 3261), SIP is an application−layer control protocol with the ability to manage multimedia sessions, such as Internet telephone calls, which makes this protocol suitable for its use in VoIP.

SIP supports five aspects regarding the establishment and termination of communications sessions:

- **User location:** Determination of the destination end system.
- **User availability:** Determination of the willingness of the call party to accept a call to this device.
- **User capabilities:** Negotiation of the session parameters.
- **Session setup:** Establishment of the session.
- **Session management:** Modification and termination of the session.

Finally, two important ideas to keep in mind are that SIP does **not** provide services, but rather provides primitives that can be used to implement these services; and that SIP works with either IPv4 or IPv6.

SIP makes use of an *offerer/answerer* model, in which the caller represents the *offerer* and the called party represents the *answerer.*

The purpose of this section is to introduce the SIP protocol. Details such as security issues related to SIP (one of the goals of this project) are described in detail later in this document.

3.1.2 Functionality

This subsection presents a simple example of the use of SIP between two end users. This example is related to the *SIP Trapezoid* and it only shows a simple SIP message exchange. The SIP Trapezoid is depicted in figure 9.1 and described in section 9.1.

In this example, one user (the offerer, referred to as "Alice" for simplicity) calls another user (the answerer, referred to as "Bob") using his SIP identity, a type of *Uniform Resource Identifier* (URI), called SIP URI. This SIP URI is similar to an email address and it contains the user name and the host identifier (for example alice@kth.se). Alice sends a request called INVITE[2] to Bob's provider SIP server (su.se proxy) via her provider's SIP server (kth.se proxy). If Bob accepts the call, the media session is established. Figure 3.1 depicts this process. Section 3.1.3 briefly describes the requests and responses.

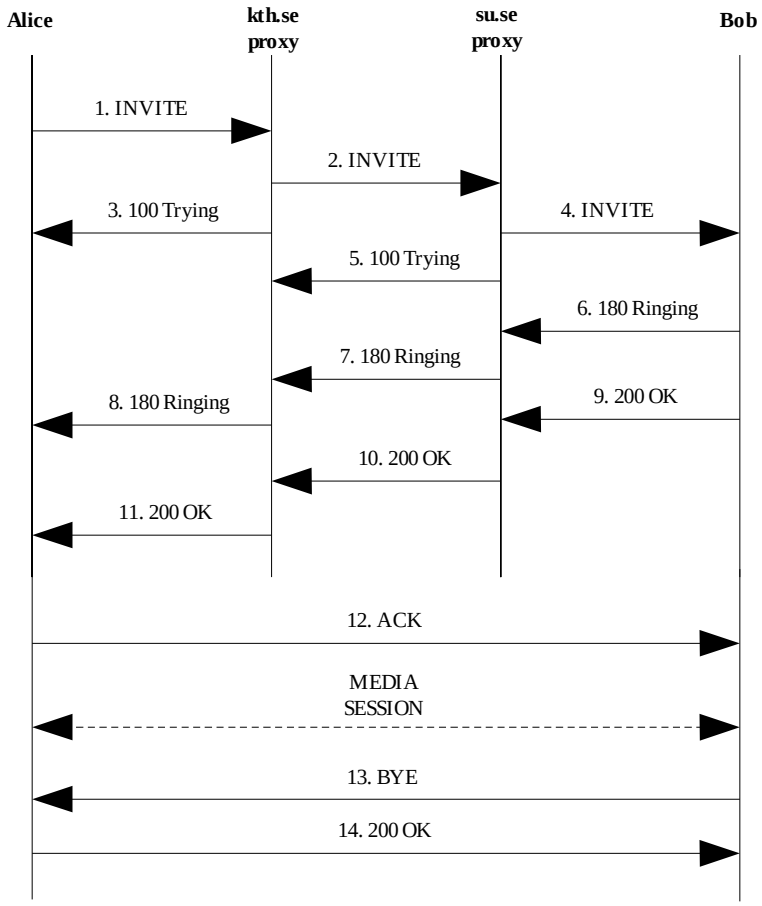

Figure 3.1 SIP setup

There are other aspects of SIP functionality besides the establishment and the termination of the call. For instance another important issue regarding SIP is the registration of the users with their provider's servers. When a SIP−based device (called *User Agent*) comes online, it first must perform registration with a SIP Registration Server (called *Registrar*). This process is handled by sending a REGISTER message. Registrations are not normally permanent, they bind the user's ID with an IP address where it can be contacted. A brief description of the REGISTER message is given in the next subsection.

The following list enumerates the main abilities SIP has in the VoIP context:

2. INVITE is an example of SIP method. These methods are described in section 3.1.3.

- Registering a user with a system
- Inviting users to join a session
- Negotiating the terms and conditions of a session
- Establishing the media stream between two or more end points
- Terminating sessions

More information and details about SIP can be found in the SIP standard (RFC3261).

3.1.3 SIP Requests and Responses

As seen in Figure 3.1, SIP is based on HTTP−like request/response (also referred to as *offer/answer*) model. The SIP specification defines a set of request messages (which in turn invoke SIP methods) and responses to those requests.

The most important method in SIP is the **INVITE** method, which is used to establish a session between participants (these participants are supposed to have previously registered with their respective provider's SIP Registrars). As an example, the following paragraph shows how the first INVITE message shown in Figure 3.1 looks:

```
INVITE sip:bob@su.se SIP/2.0
Via: SIP/2.0/UDP pc33.kth.se;branch=z9hG4bK776asdhds
Max−Forwards: 70
To: Bob <sip:bob@su.se>
From: Alice <sip:alice@kth.se>;tag=1928301774
Call−ID: a84b4c76e66710@pc33.kth.se
CSeq: 314159 INVITE
Contact: <sip:alice@pc33.kth.se>
Content−Type: application/sdp
Content−Length: 142
```

(Alice's SDP not shown)

The first line identifies the method name, and the following lines are a minimum required set of fields of the INVITE message header. These fields are briefly described below:

- **Via** contains the address at which Alice expects to receive response to her request. The branch parameter identifies the transaction.
- **To** contains a display name and the SIP URI to which the request was directed.
- **From** identifies the originator of the request by his/her display name and his/her SIP URI. The tag parameter is used for identification purposes.
- **Call−ID** is a globally unique identifier for this call.
- **CSeq** stands for Command Sequence and it is an integer used as a traditional sequence number.
- **Contact** is a SIP URI that represents a direct route to contact Alice.
- **Max−Forwards** limits the number of hops to the destination.
- **Content−Type** describes the message body (the body is not shown).

- **Content−Length** defines the length of the message body.

Section 20 in SIP standard describes the complete set of header fields.

The details of the session to be established are not explicitly described by SIP, but these details are carried in the SIP message body encoded by other protocol, typically the *Session Description Protocol* (SDP)[8].

Another important SIP method is **REGISTER**. As said above, this method is used to register a device address with a system (via SIP Registration Server or Registrar). It is necessary for a device to perform the registration in order to provide location information to permit incoming calls.

Other SIP methods are:

- **ACK:** Confirms that the client has received a final response to an INVITE request.
- **BYE:** Indicates that the user wants to terminates a session. This message may be sent by either the originator of the call or the receiver.
- **CANCEL:** Cancels a previous request message[3].

There are many different responses to these methods carried by request messages, all of them divided into six different groups [7]:

- **1xx Responses:** Informational Responses (e.g. 180 Ringing and 100 Trying).
- **2xx Responses:** Successful Responses (e.g. 200 OK).
- **3xx Responses:** Redirection Responses (e.g. 302 Moved Temporarily).
- **4xx Responses:** Request Failure Responses (e.g. 404 Not Found).
- **5xx Responses:** Server Failure Responses (e.g. 503 Service Unavailable).
- **6xx Responses:** Global Failure Responses (e.g. 600 Busy Everywhere).

The complete list and description of the SIP requests and responses can be found in the SIP standard.

3.2 Real−Time Protocol (RTP)

3.2.1 Introduction

Since 1996, the Real−Time Protocol (RTP) is an IETF standard specified in Request For Comments (RFC) 1889. RTP is a transport protocol for real−time applications which provides end−to−end network functions and services suitable for transmitting real−time data, such as audio, video, or simulation data, over unicast or multicast network services. RTP runs on top of a non−reliable transport protocol, such as UDP, to make use of the underlying multiplexing and checksum services.

RTP also provides a control protocol called *RTP Control Protocol* (RTCP), used for monitoring data delivery and to provide minimal control and identification functionality.

The services provided by RTP for the real−time data delivery include sequence numbering, payload type identification (such as audio samples or compressed video data), timestamping, and delivery monitoring. Security services for RTP and RTCP may be provided in several different ways, such as IPSec encapsulation over Virtual Private Networks (VPNs). The RTP standard also presents

3. It is important not to confuse CANCEL message with BYE message. See Chapter 7, pg. 164 in [7] for a better clarity.

some mechanisms to provide this security. However, a powerful alternative is the RTP profile *Secure Real−Time Protocol* (SRTP)[15]. RTP security issues and solutions to secure the RTP and RTCP traffic (one of the goals of this project) are described in detail later in this document.

3.2.2 Terminology and definitions

Of special interest for us is the definition of an RTP Session given in RFC 1889:

> *"**RTP session:** The association among a set of participants communicating with RTP. For each participant, the session is defined by a particular pair of destination transport addresses (one network address plus a port pair for RTP and RTCP). The destination transport address pair may be common for all participants, as in the case of IP multicast, or may be different for each, as in the case of individual unicast network addresses plus a common port pair. In a multimedia session, each medium is carried in a separate RTP session with its own RTCP packets. The multiple RTP sessions are distinguished by different port number pairs and/or different multicast addresses"*[2].

Other significant definitions are summarized as follows:

- **Synchronization Source (SSRC):** The source of a stream of RTP packets identified by a 32−bit numeric SSRC identifier carried in the RTP header, so as not to be dependent upon the network address. The RTP sender is an example of such a source. More information can be found in [2].

- **Contributing Source (CSRC):** A source of a stream of RTP packets that has contributed to the combined stream produced by the *RTP mixer*. The list of these sources is called *CSRC list* . More information about RTP mixer can be found in [2].

- **End system:** An application that generates the content to be sent in RTP packets and/or consumes the content of received RTP packets. An end system can act as one or more synchronization sources in a particular RTP session, but typically act as only one (See [2]).

3.2.3 RTP Packet Format

The RTP packet consists of a fixed header, a possibly empty list of contributing sources (unicast transmission), and a payload. The payload contains the real−time application data, such as audio or video data. Detailed information about the payload types is given in the RTP standard (RFC 1889).

The RTP header is depicted in Figure 3.2, and the fixed part has the following fields:

- **Version (V):** 2 bits. This field identifies the version of RTP. By default it is set to the value 2 for the RFC 1889 RTP specification.

- **Padding (P):** 1 bit. Set to the value 1 if padding has been applied to this packet.

- **Extension (X):** 1 bit. If the extension bit is set, the header is followed by exactly one extension field. Detailed information about the RTP extensions is given in section 5.3.1 in [2].

- **CSRC count (CC):** 4 bits. This field contains the number of CSRC identifiers that follow the RTP header.

- **Marker (M):** 1 bit. The interpretation of this field is defined by a RTP profile. See section 5.3 in [2] for further information about RTP profiles.

- **Payload Type (PT):** 7 bits. This field identifies the format of the RTP payload and determines its interpretation by the real−time application.

- **Sequence Number:** 16 bits. This field increments by one for each RTP packet sent. It may be used by the receiver to detect packet loss. The initial value of this field is random.

- **Timestamp:** 32 bits. This value reflects the sampling instant of the first octet in the RTP packet. As for the sequence number, the initial value is random.

- **SSRC:** 32 bits. This field identifies the synchronization source.

The CSRC list (zero to fifteen items, each 32 bits in length) identifies all the contributing sources for the payload of the packet. As noted above, the CC field in the fixed header contains the number of sources identified.

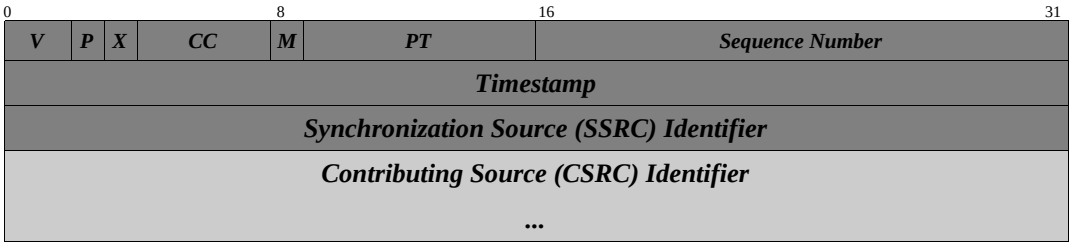

Figure 3.2 RTP Header Format

In the figure, the dark grey part corresponds to the fixed header, while the light grey part indicates the optional CSRC list.

3.2.4 RTCP Packet Format

RTP specification in defines several types of RTCP packets:

- **SR:** Sender Report. Used by a sender for transmitting statistics.
- **RR:** Receiver Report. Used by a receiver for transmitting statistics.
- **SDES:** Source Description items.
- **BYE:** Indicates end of participation.
- **APP:** Application specific functions.

Each RTCP packet begins with a fixed part similar to that of RTP data packets. This part is followed by structured elements of variable length according to the packet type, but always ending on a 32−bit boundary[2].

As an example, Figure 3.3 depicts the format of a SR RTCP packet.

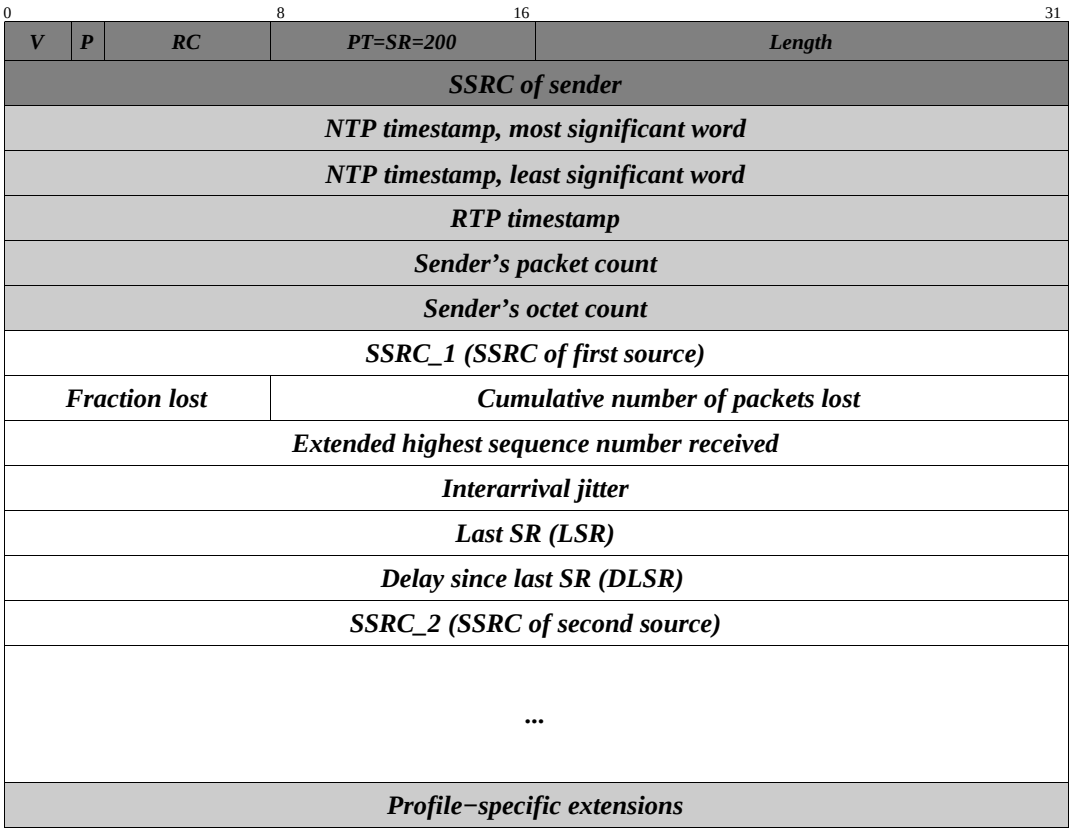

Figure 3.3 SR RTCP Packet Format

The dark gray part at the top of the figure identifies the RTCP header. This is followed by the *sender info.* The white part corresponds to the different *report blocks*. Finally certain extensions (depending on the RTP profile being used) may be added.

The RTCP header contains the following fields:

- **Version (V):** Identical to that in RTP.
- **Padding (P):** Used for the same purpose as in the RTP header.
- **Reception Report Count (RC):** 5 bits. Indicates the number of reports in this packet.
- **Packet Type (PT):** 8 bits. In this case it is set to the value 200, identifying a SR packet.
- **Length:** 16 bits. Length of the RTCP packet in 32−bit words minus 1.
- **SSRC:** 32 bits. The SSRC identifier for the RTCP packet originator.

The description of the rest of the fields is out of the scope of this document. For further information about the sender info, the report blocks, and the extensions, refer to RTP standard (RFC 1889).

4 Security Services

A Security Service is a service that enhances the security of the systems and the transfers between them, and is intended to counter Security Attacks[4]. The Security Services make use of Security Mechanisms. As a matter of fact, a Security Service implements a Security Policy, and is implemented by a Security Mechanism. We define Security Mechanism as a mechanism which is designed to detect, prevent and/or recover from a Security Attack. In the first section, a brief description of Security Attacks is given. The rest of the sections deal with each one of the main six[5] Security Services: Authentication, Access Control, Confidentiality, Integrity, Non−Repudiation, and Availability. For further information, see [41].

4.1 Security Attacks

Security Attacks are divided into two main groups: Passive Attacks and Active Attacks. A short description is given in the following paragraphs.

- **Passive Attacks:** Those whose goal is to obtain information that is being transmitted. Passive Attacks are divided into two main groups:
    - Release of message contents: Interception of the content (possibly sensitive) of a message.
    - Traffic Analysis: Interception for observing the patterns of the messages to guess the nature of a communication.
- **Active Attacks:** Those which involve some modification or alteration of the data stream, or the creation of a false stream. Active Attacks are in turn divided into four groups:
    - Masquerade: It implies one entity pretending to be a different entity.
    - Replay Attack: It consists of the capture of sensitive data, and its subsequent retransmission to produce an unauthorized effect.
    - Modification of messages: It implies the alteration, deletion, delay, or reordering of some portion of a message, producing an unauthorized effect.
    - *Denial of Service* (DoS) Attack: DoS attack prevents or inhibits the normal use or management of communication facilities by disabling or overloading them.

Passive Attacks are difficult to detect, since they do not imply alteration of the data. Thus, the solution is the **prevention** of these attacks, and the mechanism used is *encryption.*

On the other hand, Active Attacks are difficult to prevent, since that would imply the physical protection of resources and paths. Therefore, the solution is to **detect** and **recover** from these attacks.

4. A security attack is defined as an assault on system security that derives from an intelligent threat (which might exploit a vulnerability), and compromises the security of information owned by a organization.
5. In fact, OSI establishes five main Security Services, but I have added here a short definition for a sixth: Availability, since it is very related to Denial of Service (DoS) attacks, unfortunately very common nowadays.

4.2 Authentication

Authentication is the assurance that the communicating entity is who it claims to be. Two main concepts regarding Authentication are involved in an ongoing interaction:

- At the connection initiation the service assures that both entities are authentic.
- The service must assure that the connection is not interfered with in such a way that a third party can masquerade as one of the two legitimate parties for unauthorized purposes.

We must distinguish between two specific authentication services:

- **Peer Entity Authentication**: This services implies the corroboration of the identity of a peer entity, and attempts to provide confidence that an entity is not performing either a masquerade or unauthorized replay of previous connections.

- **Data Origin Authentication:** Provides for the corroboration of the source of a data unit. This specific authentication services assures, in a connectionless transfer, that the source of a received message is as claimed. Note that this specific service does not protect against duplication or modification of data units.

4.3 Access Control

Access Control service deals with the ability to limit and control the access to host systems and applications via communication links[6]. A more general definition refers to the prevention of unauthorized use of computer and network resources.

Note that it requires previous authentication to assign the correct right to each user to achieve an access control service.

4.4 Data Confidentiality

Data Confidentiality service is defined as the protection of transmitted data from passive attacks, or more generally, the protection of data from unauthorized disclosure.

Regarding the content of the data transmission, it is possible to apply Confidentiality at several levels, such as *the all messages level*, *some messages level*, *some fields of the messages level*, etc. Note that such refinements might be, in certain situations, less useful and even more complex to implement.

Another important aspect in Data Confidentiality is the *Flow characteristics Privacy*, which deals with the prevention against traffic analysis attacks mentioned in section 4.1.

The following list enumerates the different specific confidentiality services:

- Connection Confidentiality (connection protection)
- Connectionless Confidentiality (single data block protection)
- Selected−Field Confidentiality (on a connection or of a single data block)
- Traffic−Flow Confidentiality (the information from which traffic patterns can be derived)

6. In the context of network security.

4.5 Data Integrity

The Data Integrity service assures that the data received are exactly as sent by an authorized entity (i.e. no alteration, modification, insertion deletion, or replay). The integrity service, as in the confidentiality service, can be applied to a connection, single message, or selected fields of a single message.

It is necessary to distinguish between integrity services with or without *recovery*. This means that we would like **just to report** an integrity violation (service without recovery), or **to report and recover** from the violation (service with recovery).

The following list shows the different specific integrity services:

- Connection Integrity with Recovery
- Connection Integrity without Recovery
- Selected−Field Connection Integrity
- Connectionless Integrity
- Selected−Field Connectionless Integrity

4.6 Non−Repudiation

The Non−Repudiation service prevents the sender or the receiver from denying that they transmitted a message. Therefore, when a message is sent, the receiver can prove that the alleged sender sent the message. Similarly, when a message is received, the sender can prove that the receiver in fact received the message.

Thus, we distinguish between two different specific Non−Repudiation services:

- Origin Non−Repudiation
- Destination Non−Repudiation

4.7 Availability

As described in [41], Availability is "the property of a system or a system resource being accessible and usable upon demand by an authorized system entity, according to performance specifications for the system". Thus, an Availability service protects a system so as to ensure its availability. One the main purposes of an Availability service is the protection against DoS attacks, described in section 4.1.

5 Cryptography Overview

This chapter is divided into two main parts and gives a brief introduction to the basic elements and methods used by cryptography. First, some basic knowledge necessary to understand concepts as Private or Public Cryptography is given. The same subsection also presents some other concepts, such as certificates and the one−way functions. The second part is specifically oriented to this project and the cryptographic tools it utilizes (such as the algorithms used by SRTP). This part deals with some specific cryptographic algorithms and mechanisms to provide confidentiality and message integrity: Advanced Encryption Standard (AES)[34] and HMAC−SHA1[35] are described in detail here. Most of the information in this section has been obtained from [41], along with other sources.

5.1 Basic Knowledge

5.1.1 Introduction

Cryptography is the science of secret writing. We can also define it as the art of keeping messages private over an insecure medium. The generic scenario is shown in Figure 5.1, obtained from [41]. This scenario deals with two entities who want to communicate in a secure way, adding privacy to the message exchange, so that an intruder (*eavesdropper*) has no possibility to read the messages. The solution to this issue consists of coding such messages in order to avoid an unauthorized person retrieving the original text. This scheme is called *symmetric cryptography*, and it is the most simple case in cryptography. Another possible cryptographic scheme is the *asymmetric cryptography.* Further sections will give a short description of both cryptographic mechanisms.

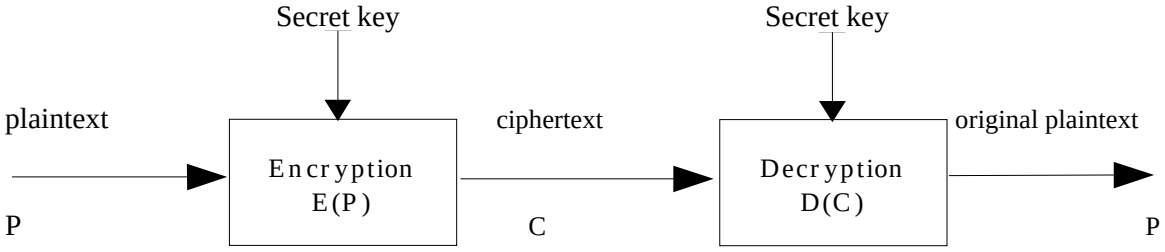

Figure 5.1 Simplified Model of Symmetric Encryption

We have 5 components in this scheme:

- **Plaintext:** This is the original message or data that is fed into the algorithms as input.
- **Encryption algorithm:** The encryption algorithm performs various substitutions and/or transformations on the plaintext. It is also called *cipher*.
- **Secret key:** The secret key is also input to the algorithm. The exact substitutions and/or transformations performed by the algorithm depend on this key.

- **Ciphertext:** This is the scrambled message produced as output. It depends on the plaintext and the secret key. Note that two different keys applied on the same plaintext, and using the same algorithm, will produce two different ciphertexts.

- **Decryption algorithm:** This is essentially the encryption algorithm run in reverse. It basically takes the ciphertext and the same secret key and produces the original message.

The following paragraphs deal with some terms and their definitions.

As said above, a *cipher* is a mathematical function used to encrypt/decrypt messages. It is also called cryptographic algorithm.

The process of coding a message is called *encryption* E(P), and its output is the ciphertext. *Decryption* D(C) is recovering the original message. Thus, we have that E(P)=C and D(C)=P, such that D(E(P))=P.

Cryptanalysis is the science of breaking ciphers. It consists of the process of attempting to discover the original message (plaintext) or the key.

Cryptology encompasses both cryptography and cryptanalysis subjects.

It is important to remark that, in most cases, the security of encryption relies on the secrecy of the key, rather than the secrecy of the cipher. By using cryptography we are mainly providing confidentiality using keys, although other security mechanisms supporting the rest of security services (such as the *digital signature* for data origin authentication) make use of cryptography. Thus, cryptography enhances computer security, but it is not a substitute for it.

5.1.2 Symmetric Cryptography

Symmetric encryption is also referred to as conventional encryption, secret−key, or single−key encryption. It remains the most widely used of the two types of encryption.

In this scheme, one only key is used by the entities to encrypt and decrypt a message, so that this shared key, which must be kept secret by both entities, is previously exchanged, or even distributed by a trusted third party to both entities. The basic scenario and its components were shown in the previous section, in the Figure 5.1.

The two main requirements for secure use of symmetric encryption are the following:

- A strong encryption algorithm is needed. We would like the algorithm to be such that an intruder with access to this algorithm and some different ciphertexts, would be unable to decrypt those ciphertexts and/or guess the secret key used. This requirement becomes stronger if we consider that the intruder should not be able to figure out the key even when having access to the ciphertext and the matching plaintext.

- Both sender and receiver must have obtained copies of the secret key in a secure and secret fashion, and must keep it secret. Of course, if someone intercepts the key and knows the algorithm being used, the exchanged data is readable.

Thus, this key exchange is one of the main challenges in symmetric cryptography, and the fact of distributing or exchanging this key in a secret and secure way becomes a problem in this scheme[7]. In

7. As a matter of fact, the principal security problem in symmetric cryptography is maintaining the secrecy of the key.

addition to this, if the number of users in a certain community is high, so it is the number of keys. This may cause an important overload to the administration system.

We have to distinguish between two different types of symmetric ciphers:

- Stream ciphers: take a data stream and a key as input, and combine each bit of plaintext with one bit of the key. These ciphers are suitable for hardware implementation.

- Block ciphers: operate on data blocks of a particular size and encrypt them with a key, and are suitable for software implementations.

Some examples of symmetric ciphers are *Data Encryption Standard* (DES)[49], *Triple DES* (3DES), *International Data Encryption Algorithm* (IDEA), *Blowfish*, *RC5*, and *Advanced Encryption Standard* (AES). The latter is also known as *Rijndael Cipher*, and it is the algorithm selected in the SRTP implementation, provided in this document, to encrypt the media stream. A wide description of AES cipher is given in section 5.2.1, and a shorter description of DES and 3DES is also given in section 5.2.2.

5.1.3 Asymmetric Cryptography

Asymmetric encryption is also known as public−key encryption. It is of equal importance to symmetric encryption, and it finds use in message authentication and key distribution.

Public−key cryptography was firstly publicly proposed by Diffie and Hellman in 1976, and it involves the use of two different keys, the private and the public key. This fact has important consequences in the areas of confidentiality, key distribution and authentication.

Asymmetric cryptography has the same components as the symmetric cryptography, except the secret key. Instead of using the same secret key to encrypt and decrypt the message, public−key cryptography makes use of two different keys, grouped in a owner's pair. One of the keys of this pair is used for encryption, and the other is used for decryption. These keys are called the private key and the public key. The former is kept secret by the owner and it is used to encrypt data, as well as decrypt data encrypted by the public key of the pair. On the other hand, the public key is made public by the owner for others to use, and it is used to encrypt data, or to decrypt data encrypted with the private key.

With this approach, all participants in a encrypted communication have access to other participant's public keys. Furthermore, as said before, private keys are generated locally by each participant and therefore never distributed.

The essential steps in public−key cryptography, shown in Figure 5.2, are the following:

- Each user generates his or her pair of keys.
- Each user places one of the two keys in an accessible public register. The companion key is kept secret by its owner.
- One user who wants to send a private message to the other, encrypts that message with the receiver's public key.
- The receiver gets the message and decrypts it with his or her private key. Note that **only** the receiver can decrypt the message, since it is assumed that he or she is the only who knowing the private key.

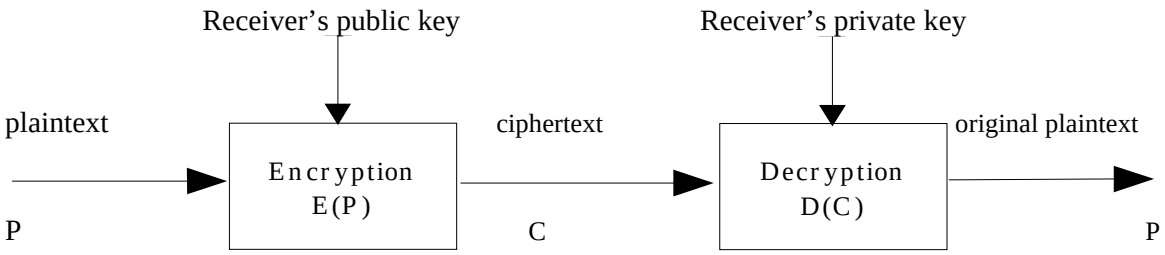

Figure 5.2 Simplified Model of Asymmetric Encryption providing Confidentiality

Another possibility in asymmetric encryption is its use as the basis for digital signatures. Let us consider the case in which one user sends a message encrypted with his or her private key. In this scheme, data origin authentication (without confidentiality) is provided, since the sender proves his or her identity by being the only possessor of the right private key.

The most important examples of public−key algorithms are RSA Public−Key Encryption Algorithm, and the Diffie−Hellman Key Exchange. Other public−key ciphers are Digital Signature Standard (DSS) and Elliptic−Curve Cryptography.

5.1.4 Symmetric Cryptography vs. Asymmetric Cryptography

Public−key cryptography is powerful, but it does not suit every situation. Furthermore, there are some common misconceptions concerning public−key cryptography (from [41]):

- **Public−key encryption is more secure from cryptanalysis than secret−key cryptography.** As a matter of fact, the security in any cryptography scheme depends on the length of the key and the computational work needed to break the cipher. In principle, there is no proof about the idea that one scheme is superior to the other from the point of view of withstanding cryptanalysis.

- **Public−key encryption has made secret−key encryption obsolete.** On the contrary, the computational overhead of current public−key cryptography makes secret−key cryptography will not be abandoned.

- **Key distribution in public−key cryptography is trivial.** In fact, it is still necessary for public−key cryptography to use some form of protocol, often involving a central and trusted agent. Furthermore, "the procedures involved are not simpler or more efficient than those used for secret−key cryptography" [41].

Thus, **asymmetric algorithms are not substitute** for symmetric algorithms. The most common solution, adopted in most of the models, consists of a hybrid cryptosystem.

5.1.5 Cryptanalysis

As said above, cryptanalysis is the process of attempting to discover the plaintext or the key. The attacker will act depending on the information available and the nature of the encryption scheme.

A encryption algorithm is generally designed to withstand a *known−plaintext attack.* In this attack, the information available for the cryptanalyst is the following:

- Encryption algorithm
- Ciphertext to be decoded
- One or more plaintext−ciphertext matching pairs formed with the secret key

If an encryption algorithm is to be proved as secure (computationally secure), the applied criteria comprises these two aspects (from [41]):

- The cost of breaking the cipher exceeds the value of the encrypted information
- The time required to break the cipher exceeds the useful time of the information

Table 5.1 (from [41]) shows how much time is involved in the key search for various key sizes.

Key Size (bits)	**Number of Alternative Keys**	**Time Required at 1 Encryption/µs**	**Time Required at 10^6 Encryption/µs**
32	$2^{32} = 4.3 \times 10^9$	2^{31} µs = 35.8 minutes	2.15 milliseconds
56	$2^{56} = 7.2 \times 10^{16}$	2^{55} µs = 1142 years	10.01 hours
128	$2^{128} = 3.4 \times 10^{38}$	2^{127} µs = 5.4×10^{24} years	5.4×10^{18} years
168	$2^{168} = 3.7 \times 10^{50}$	2^{167} µs = 5.9×10^{36} years	5.9×10^{30} years
26 characters (permutation)	$26! = 4 \times 10^{26}$	2×10^{36} µs = 6.4×10^{12} years	6.4×10^6 years

Table 5.1 Average Time Required for Exhaustive Key Search[41]

5.1.6 One−way Hash Functions and MACs

A one−way hash function accepts a variable size message *M* as input and returns a fixed−size message digest H(*M*). It is used to authenticate a message so that the hash result is sent along with the original message in such a way that the recipient can verify that the message digest is authentic. The function H() is called one−way function since it is relatively easy to compute (one way), but significantly harder to reverse. Some examples of one−way hash functions are MD5 Message Digest Algorithm and Secure Hash Algorithm (SHA−1). A detailed description of the latter is given in section 5.2.3, since it is used in the SRTP implementation.

One−way hash functions are also known as *non−keyed hash functions* or Message Description Code (MDC).

Some requirements for secure hash functions are described in [41] as follows:

- H() can be applied to a block of data of any size
- H() produces a fixed−length output
- H(x) is relatively easy to compute for any given x
- One−way property: Given a value h, it is computationally infeasible to find x such that H(x) = h
- Weak collision resistance property: Given a block x, it is computationally infeasible to find $y \neq x$ with H(y) = H(x)

- Strong collision resistance property: It is computationally infeasible to find a pair (x, y) such that $H(y) = H(x)$

On the other hand, when a shared secret is added to compute the digest, we get a Message Authentication Code (MAC). Thus, we can illustrate the MAC as follows:

$$MAC_M = F(K_{AB}, M)$$

where K_{AB} is the secret shared between the parties.

MACs are also referred to as *keyed hash functions.*

With this approach, message integrity and data origin authentication are provided. Several algorithms can be used to generate the digest, such as DES, although in recent years, there has been increased interest in developing a MAC based on a cryptographic hash code (*Hash−Based MAC, HMAC*). The reasons and approaches to this technique are described in the next section.

5.1.7 Overview of the Hash−based Message Authentication Code : HMAC

The reasons for the interest in basing MAC on hash functions are the following:

- Cryptographic hash functions execute faster in software than the conventional encryption algorithms such as DES.
- Library code for cryptographic hash functions is widely available.
- Unlike conventional encryption algorithms, there are no export restrictions for cryptographic hash functions.

The approach which has received more support is HMAC, which treats the cryptographic hash function as a *black box*[8], enhancing efficiently the use of different functions to generate the digest. A wide description of HMAC is given in section 5.2.4, since it is used (based on SHA−1 hash algorithm) for the SRTP implementation, as well as for other important protocols such as TLS and IPSec.

5.1.8 Certificates

Public−key cryptography is related to the use of certificates. The most widely used type of certificate is defined in *ITU−T X.509* standard (see [19]). The heart of the X.509 scheme is the public−key certificate associated with each user.

Certificates are assumed to be created and signed by a trusted third party, known as the *issuer* or *Certification Authority* (CA). Basically, a certificate contains the public key associated with each user, among other information, such as the user name and the certificate expiration date.

A wide description of certificates and certification infrastructures (also referred to as Public−Key Infrastructures) is given in section 6.

8. The hash function implementations can be integrated as modules when implementing HMAC, making those functions easy to modify or even replace if desired.

5.1.9 Location of encryption devices

Before using encryption mechanisms, it is necessary to decide what and where we want to encrypt. There are two fundamental alternatives:

- Link encryption
- End−to−end encryption

Link encryption refers to lower layers (physical or link layer) encryption. It is the easiest way to encrypt data, and it is often implemented by hardware encryption devices in every node in the network.

Every node traversed decrypts the incoming packet, process it, and encrypts it again before sending it out the link. The problem in Link encryption is that the data is in clear text inside each node it has to traverse to reach its destination.

The alternative to Link encryption is End−to−end encryption. It places the cryptographic equipment between the network and the transport layers, thus protecting the data from the source to the final destination. Disadvantages of this scheme are that it allows traffic analysis, and makes the key management more complex.

The encryption can also take place at the highest OSI layers (presentation and application), making it independent of the network used, but requiring interaction with the user's software.

5.2 Basic Algorithms and Methods

5.2.1 Advanced Encryption Standard (AES)

5.2.1.1 AES History

In 1997, the National Institute of Standards and Technology (NIST) issued a call for proposals for a new Advanced Encryption Standard (AES), which should have a security strength equal to or better than other algorithms such as 3DES, and provide significantly improved efficiency. In addition to these requirements, NIST specified that AES must be a symmetric block cipher with a block length of 128 bits and support for key lengths of 128, 192 and 256 bits.

The selected final standard adopted the Rijndael cipher as the proposed AES algorithm. Rijndael was developed by Dr. Joan Daemen and Dr. Vincent Rijmen, and was published as a final standard (FIPS PUB 197) in November of 2001.

5.2.1.2 Overview of the Algorithm

AES uses a block length of 128 bits and a key length which can be of 128, 192, or 256 bits, although in the description in this section we assume a length of 128 bits for the key. This 128−bit length is likely to be the most commonly implemented and the one used in the SRTP implementation described in section 13.2. The mode of operation of AES used in such an implementation is *Counter Mode* (CTR)[47].

When it comes to computational efficiency, Rijndael cipher is a **low−cost**, **high−speed** encryption algorithm. This is the main reason which makes Rijndael cipher suitable for real−time traffic

encryption, where the performance of the encryption/decryption process becomes very important. Figure 5.3 (from [48][9]) shows the time taken for the different tasks to perform by some of the AES candidates, included Rijndael.

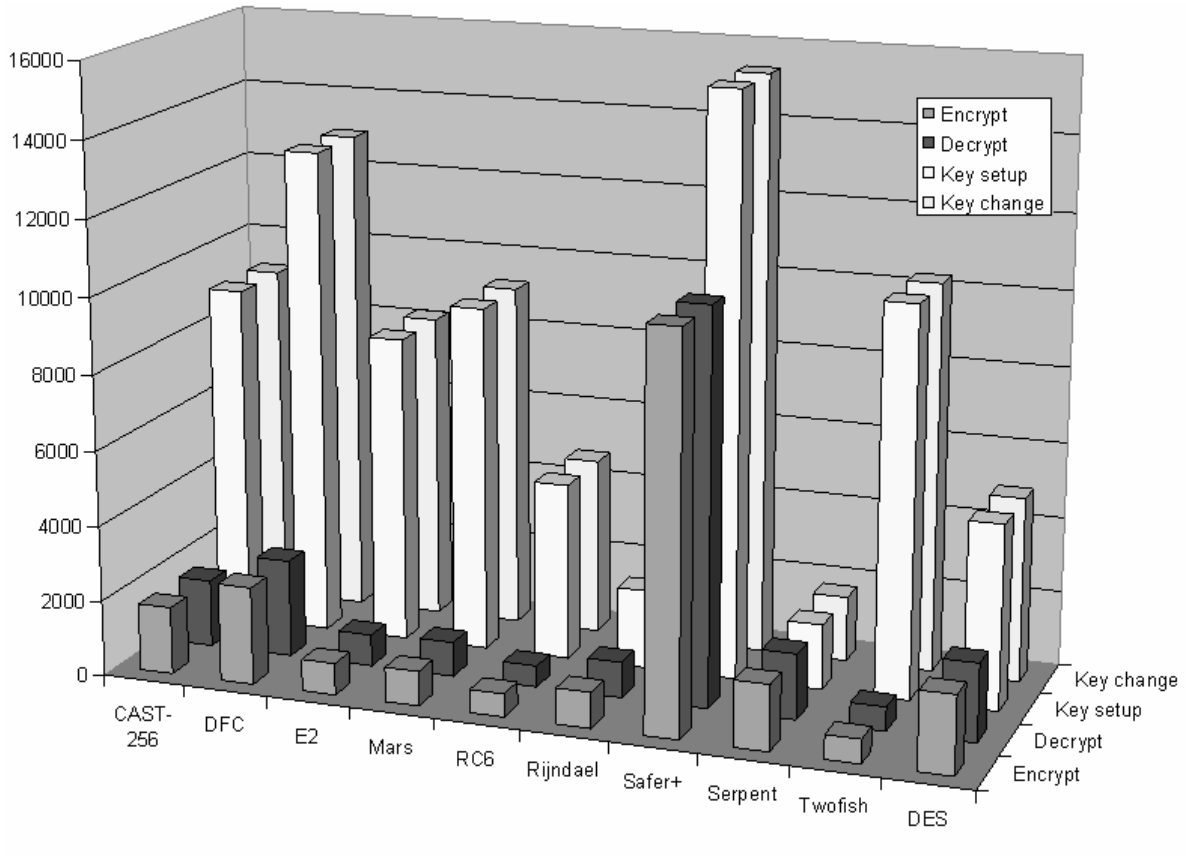

Figure 5.3 Time (clock cycles) taken by some AES candidates [48]

The overall encryption/decryption structure in CTR mode is shown in Figure 5.4, based on figure 1 in [47]. In this figure, *M* is the plaintext, *K* is the key, and *ctr* is a counter. The ciphertext is (*ctr*, *C*), or, more generally, *C* together with something adequate to recover *ctr*. Decryption is the same as encryption with the *M* and the *C* interchanged[10]. *C* is the XOR (exclusive−or) of *M* and the first |*M*| bits of the pad $E_K(ctr) \mid E_K(ctr+1) \mid E_K(ctr+2)\ldots$.

9. This figure appears in this document with the explicit permission of its owner, Professor Jean−Jacques Quisquater, Microelectronics Laboratory Crypto Group, Université Catholique de Louvain (UCL), Louvain, Belgium.
10. The ciphertext is often referred to as *C*, rather than (*C*, *ctr*).

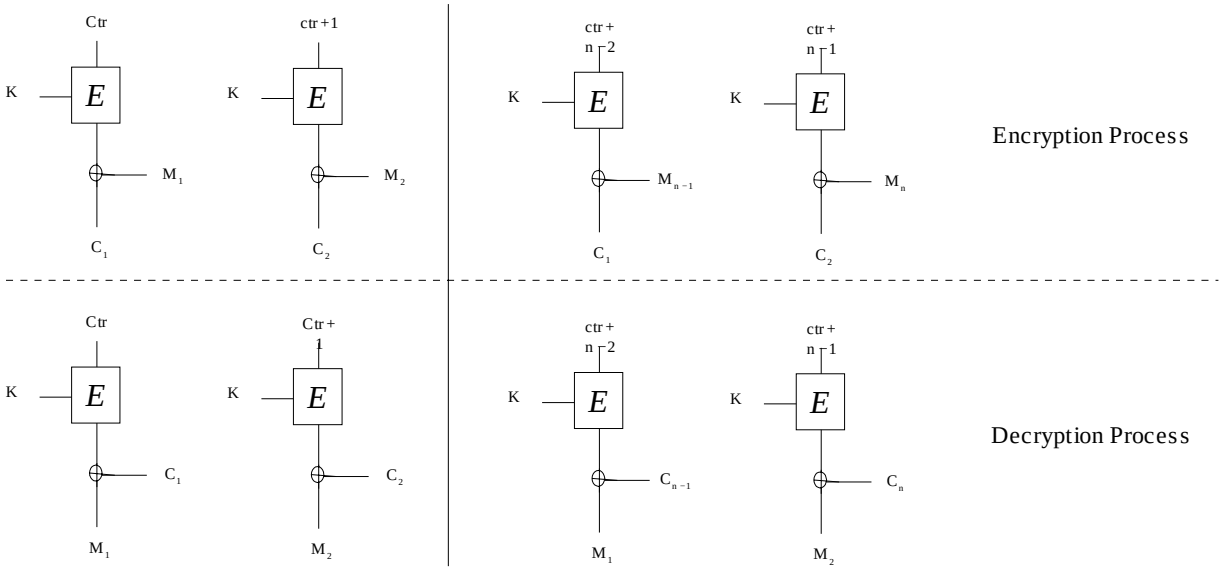

Figure 5.4 Encryption and Decryption Process in CTR Mode[47]

The structure of the algorithm is quite simple. For both encryption and decryption, the cipher starts by an Add Round Key stage, followed by nine rounds of four stages each one. These stages are the following:

- Substitution of bytes (Byte Sub)
- Shifting of rows (Shift Row)
- Mixing of columns (Mix Column)
- Add Round Key

Finally, there is a final round of three stages (all the stages indicated above, except the mixing of the columns). Figure 5.5 (based on figure 2 from [46]) depicts the structure of a full encryption round.

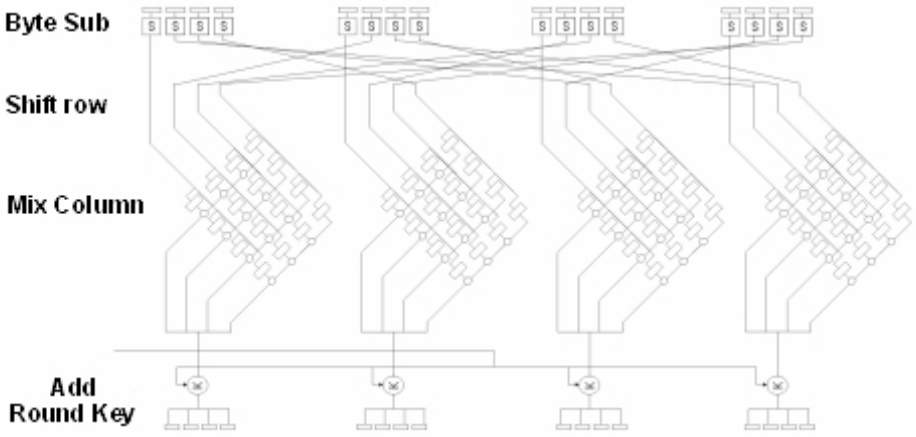

Figure 5.5 AES Encryption Round [46]

5.2.2 Data Encryption Standard (DES)

DES is the most widely used encryption algorithm. It was adopted by the NIST in 1977 (FIPS PUB 46). It is also known as the *Data Encryption Algorithm* (DEA) by ANSI and as the *DEA−1* by the ISO.

The length of the plaintext block to be processed is 64 bits, and the length of the key is 56 bits. The structure of the algorithm is a minor variation of the *Feistel Structure* (see [41], pages 32−34). There are sixteen rounds of processing, each one using a subkey generated from the 56−bit original. More details concerning the encryption/decryption process are in [49].

Nowadays, DES is a worldwide standard, and its security has been long questioned, since several studies have estimated costs and times for attacking and breaking DES. Despite numerous approaches no one has thus far succeeded in discovering a fatal weakness in the algorithm. However, a more serious concern is the key length. In July 1998 the *Electronic Frontier Foundation* (EFF) announced that it had broken a DES encryption, in less than three days, using a *cracker machine* built for less than $250,000. Assuming the EFF machine performs 10^6 decryptions/μs, the use of a key of 128 bits, very common among contemporary algorithms, it would take the EFF cracker over 10^{18} years to break the code. So a 128−bit key is guaranteed to result in an algorithm that is unbreakable by brute force with present technology.

In 1999, 3DES was incorporated as part of the Data Encryption Standard. 3DES uses three keys and three executions of the DES algorithm, so that the effective length of the key becomes 168 bits, making the brute−force attacks effectively impossible. Although 3DES is nowadays widely used, AES is intended to replace it in a number of years. Meanwhile, 3DES and AES will coexist as FIPS−approved algorithms, allowing for a gradual transition to AES.

5.2.3 Secure Hash Algorithm (SHA)

5.2.3.1 SHA History

The Secure Hash Algorithm (SHA) was also developed by the NIST, and was published as a standard (FIPS PUB 180) in 1993. A revised version was issued as FIPS PUB 180−1 in 1995, and is generally known as SHA−1.

5.2.3.2 Overview of the Algorithm

As described in [41], the algorithm takes a message with a maximum length of 2^{64} bits as input, and generates a digest of 160 bits. The input is processed in 512−bit blocks.

The overall process performed by the algorithm to produce a message digest is depicted in the figure 5.6 (based on figure 3.4 from [41]).

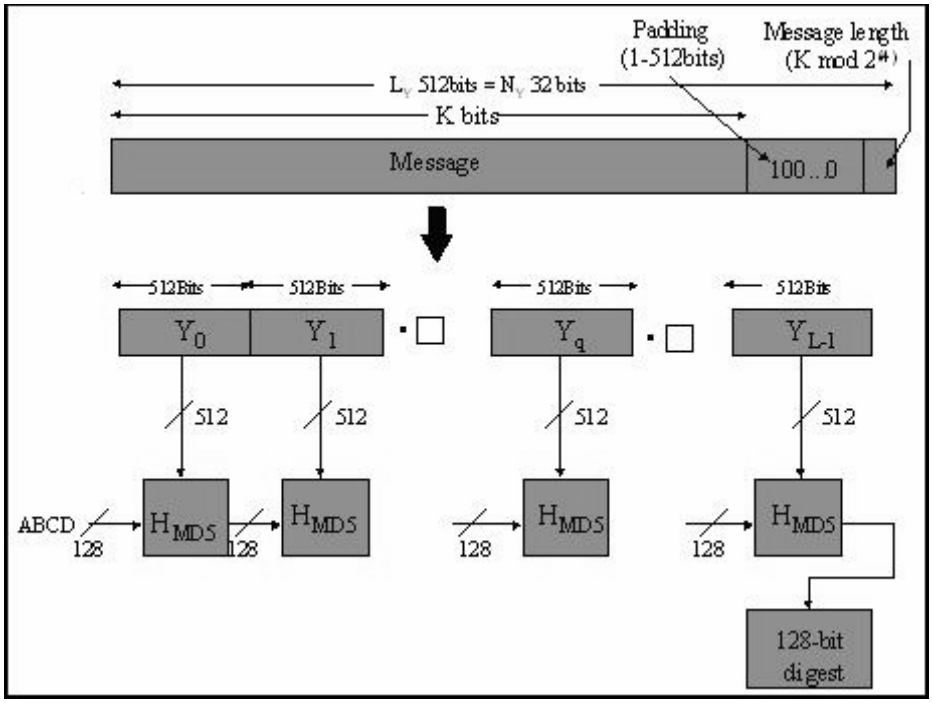

Figure 5.6 Message Digest Generation Using SHA−1[41]

The processing consists of four steps, briefly described as follows:

- **Step 1:** Append padding bits. The message must be padded so that its size is congruent to 448 modulo 512. Note that padding is **always** added, even if the message has already the desired length.

- **Step 2:** Append length. The length of the original message, in a block of 64 bits, is appended to the message.

- **Step 3:** Initialization of the MD buffer. This 160−bit length buffer is used to hold intermediate and final results of the hash function. This buffer appears as five 32−bit registers, each one initialized with certain hexadecimal values.

- **Step 4:** Process message in 512−bit blocks. This is performed by a module, known as *compression function*, which consists of four rounds of 20 stages each. Each round uses a different primitive logical function. This step is depicted in Figure 5.7 (based on figure 3.5 from [41]).

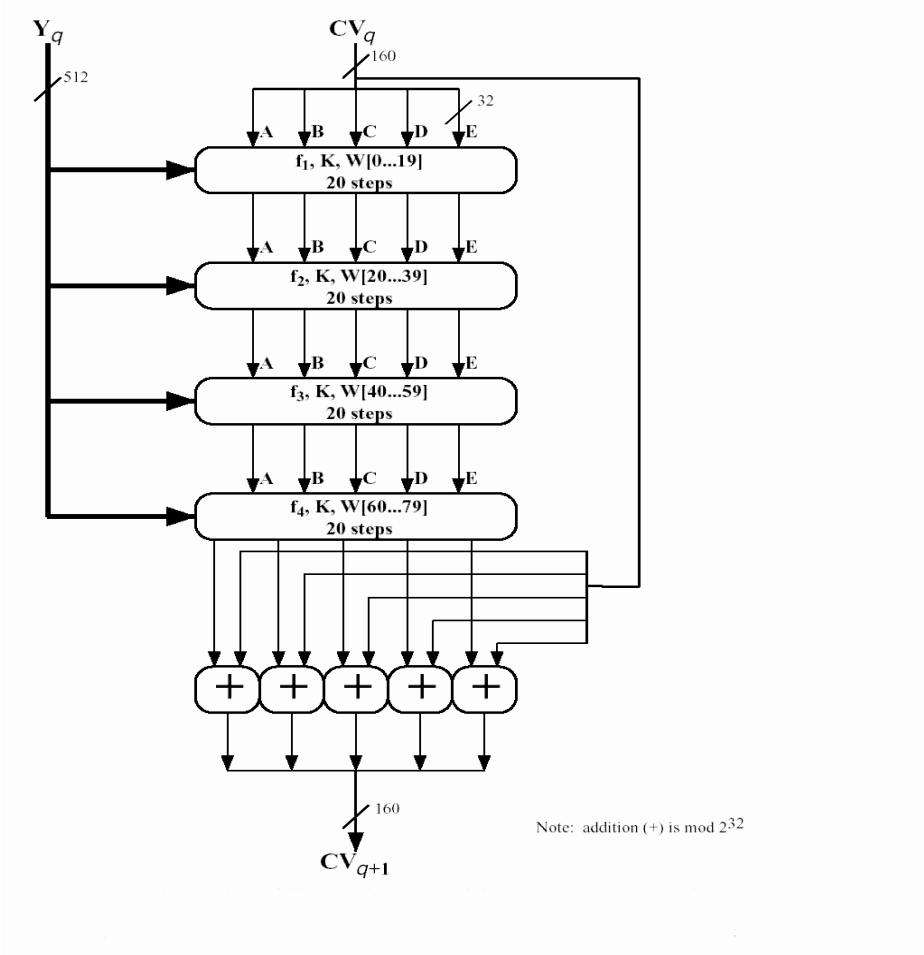

Figure 5.7 SHA−1 Processing of a Single 512−Bit Block (SHA−1 Compression Function)[41]

5.2.4 Hash−Based Message Authentication Code (HMAC)

The design objectives for HMAC are described in [35] and [41] as follows:

- To use, without modifications, available hash functions that perform well in software and for which code is freely and widely available.
- To allow for easy replaceability of the embedded hash function in case faster or more secure hash functions are found or required.
- To preserve the original performance of the hash function without incurring a significant degradation.
- To use and handle keys in a simple way.
- To have a well−understood cryptographic analysis of the strength of the authentication mechanism based on reasonable assumptions on the embedded hash function.

The HMAC can be expressed as follows:

$$\mathrm{HMAC}_K(M) = \mathrm{H}[(K^+ \oplus \mathrm{opad}) \,||\, \mathrm{H}[(K^+ \oplus \mathrm{ipad}) \,||\, M]],$$

where:

- H is the embedded hash function, such as SHA−1
- M is the message input to HMAC
- K^+ is the secret key K padded with zeros on the left
- opad is the value 01011100 repeated b/8 times, where b is the number of bits in a block (e.g. 512 bits)
- ipad is the value 00110110 repeated b/8 times, where b is the number of bits in a block (e.g. 512 bits)
- $\oplus$ is the logical XOR operation

6 Public−Key Infrastructures

Public−Key Infrastructures (PKIs) are commonly used nowadays as the base to provide different security services, such as authentication, confidentiality, message integrity, and non−repudiation; and they are the support for SSL/TLS, a good alternative for securing the SIP protocol. PKIs are strongly based on the use of certificates and, thus, on the use of public−key cryptography. For further information about Public−Key Infrastructures refer to [50, 51].

6.1 Introduction and Terminology

A Public−key Infrastructure is the backbone for large−scale use of public−key cryptography. Some concepts such as X.509 certificates and digital signature are related to the public−key infrastructures. These terms and others are described in the following list:

- **X.509 Public−Key Certificate:** The most widely used format of public−key certificate. It is the heart of the X.509 scheme, as mentioned in section 5.1.8. The public−key certificate is issued by a *Certification Authority* (CA) which certifies that the public−key belongs to the holder indicated in the certificate. Section 6.2 gives a more detailed description of the X.509 certificates.

- **Digital Signature:** It is the digital equivalent of a hand−written signature. The digital signature is created by using public−key cryptography. The sender encrypts the message to be sent (usually a hash of the message) with his or her private key, such that the receiver can ensure the incoming message was sent only by the sender.

- **Electronic Signature:** It is the electronic equivalent of the hand−written signature, through public−key cryptography (digital signature) or biometrics (*dynamic signature).*

- **Certification Authority (CA):** The Certification Authority is the organization that issues the public−key certificates. These authorities may also provide some other cryptographic services, such as certificate distribution or certificate revocation (by using *Certificate Revocation Lists*, known as CRLs*)*. The CAs are also referred to as *Trusted Third Parties* (TTPs).

- **Public−Key Infrastructure (PKI):** Infrastructure with CAs. Several examples of PKI models are shown in the section 6.3. For further information, a good reference to PKIs can be found in [51].

Therefore, a PKI is basically a coherent structure of CAs, and it is nowadays considered as a prerequisite for security in large networks and distributed systems. It can be seen as a big certificate database where users can retrieve other users' certificates, in order to have access to their public keys. For example, one user who wants to communicate with other, must previously retrieve the other user's certificate and, thus, his or her public key.

6.2 X.509 Certification Structure

The X.509[11] certificate[19] is the most common certificate format used nowadays. It is part of the scheme defined in the X.500 series of recommendations that define a *directory service*. In this scheme, a directory is a server or distributed set of servers that maintains a database of information about users.

11. The X.509 certificate is part of the ITU−T recommendation X.509, which in turn is part of the X.500 series.

These directories may serve as the repository of X.509 public−key certificates, which contain the public key of the user along with other information, and are signed by a CA.

The X.509 public−key certificate is assumed to be created by a trusted CA, and placed in the directory by the CA or even by the user itself. Thus, this directory provides the easily accessible location for the users.

Furthermore, the X.509 is an important standard, since the certificate structure and authentication protocols defined in X.509 are used in a variety contexts, such as IP Security (IPSec) and Transport Layer Security (TLS). Figure 6.1 shows the general format of a X.509 certificate. These certificates have the following features:

- The certificates are generated and signed by a CA, and any user with access to the CA's public key can verify the public key of the user who is certified in the certificate.
- No party other than the CA can modify the certificate without this being detected.
- They can be placed in the directory without protection because they are unforgeable.
- A certificate may also be directly delivered to another user.

Version
Certificate Serial Number
Signature Algorithm Identifier
Issuer X.500 Name
Period of Validity
Subject X.500 Name
Subject's Public Key Information
Issuer Unique Identifier
Subject's Unique Identifier
Extensions
CA's Signature

Figure 6.1 X.509 version 3 certificate format

The X.509 version 3 format includes the following components:

- **Version:** Differentiates among successive versions of the certificate format. The default version is 1. Version 2 adds the Issuer Unique Identifier and Subject Unique Identifier fields, and version 3 adds the Extensions field.

- **Certificate Serial Number:** A unique within the issuing CA integer value associated with this certificate.

- **Signature Algorithm Identifier:** The algorithm used by the CA to sign the certificate and its parameters

- **Issuer X.500 Name:** X.500 distinguished name of the CA which created and signed this certificate.
- **Period of Validity:** The first and last dates on which the certificate is valid.
- **Subject X.500 Name:** The X.500 distinguished name of the user to whom this certificate refers.
- **Subject's Public−Key Information:** The public key of the subject, and an identifier of the algorithm for which this key is to be used and its parameters.
- **Issuer Unique Identifier:** Optional bit string field used to uniquely identify the issuing CA in the event the X.500 name has been reused for different entities.
- **Subject Unique Identifier:** Optional bit string field used to uniquely identify the subject in the event the X.500 name has been reused for different entities.
- **Extensions:** Set of one or more extension fields, divided into three categories: Key and Policy Information, Certificate Subject and Issuer Attributes, and Certification Path Constraints.
- **Signature:** Covers all the other fields of this certificate, and contains the hash of them, encrypted with the CA's private key. This field also include the signature algorithm identifier.

6.2.1 Chaining

When all the users are subscribed to the same CA, there is a common trust of that CA. But this is not the case in today's networks. The common situation is that the users are registered to different CAs, since it is not practical for all users to subscribe to the same CA when there is a large community of parties.

Since it is the CA that signs the certificates, each user must have a copy of the signing CA's public key in order to verify the certificate signatures. Now suppose that *A* has obtained a certificate from CA *X*, and *B* has obtain a certificate from a different CA, for example *Y*. *A* can read *B*'s certificate, but cannot verify the signature, since *A* does not securely know the public key of *Y*. Let us suppose in this case that both CAs have securely exchanged their public keys. Therefore, *A* can obtain the certificate of *Y* signed by the "trusted" *X* and get *Y*'s public key verifying *X*'s signature. Now *A* can go back to the directory and get the certificate of *B*, being able to verify it by using *Y*'s public key. This process is known as *chaining*, and it is expressed as:

$$X\langle\langle Y\rangle\rangle Y\langle\langle B\rangle\rangle$$

In the same fashion, *B* can obtain *A*'s public key with the reverse chain:

$$Y\langle\langle X\rangle\rangle X\langle\langle A\rangle\rangle$$

This chaining can be expanded to *N* elements.

6.2.2 Revocation of Certificates and Certificate Revocation Lists (CRLs)

Typically, a new certificate is issued by the CA just before the expiration of the old one. However, there are three main situations in which a certificate can be revoked before its

expiration:

- The user's private key is assumed to be compromised.
- The CA's certificate is assumed to be compromised.
- The user is no longer certified by this CA.

Each CA must maintain a list consisting of the revoked (but not expired) certificates, issued by that CA, for a certain period of time. This list is the Certificate Revocation List (CRL) and contains the users', as well as the other CAs' revoked certificates, and should be posted in the directory. Other different ways to distribute and post the CRLs are via *Web/file* or by using *CRL Distribution Points* (CDPs).

Every CRL posted by a CA in the directory is signed by this CA. Figure 6.2 shows the format of the CRL. This format includes the issuer's name, the date the list was created, the day the next CRL is scheduled to be issued, and an entry for each revoked certificate, which consists of the serial number of the certificate and its revocation date.

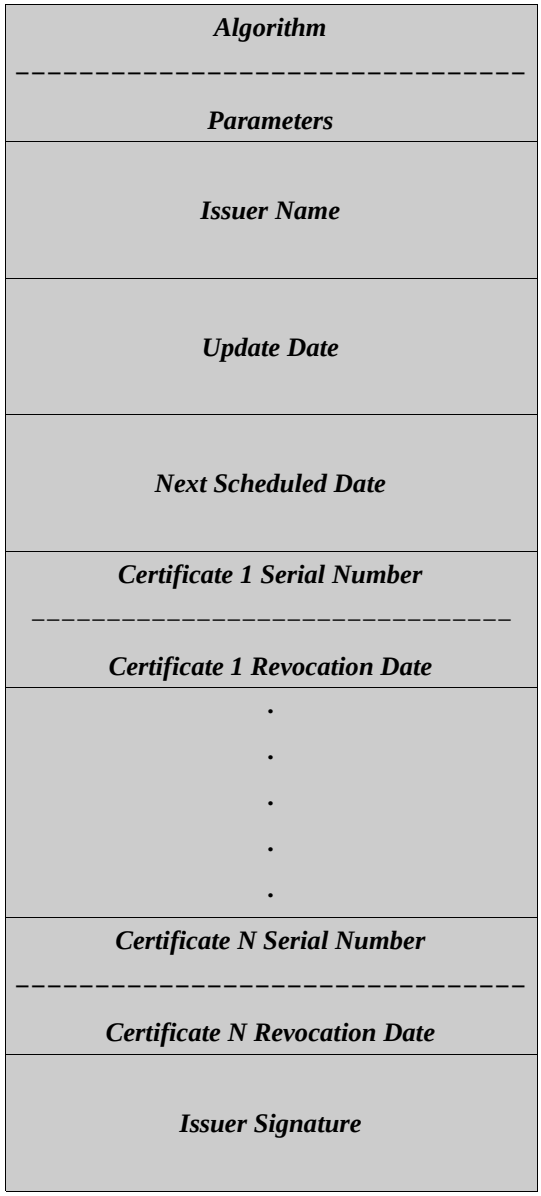

Figure 6.2 Certificate Revocation List (CRL) Format

The CA establishes the length of the period while a certain CRL is active, depending on the CA's policies. During this period, the revoked certificates, if any, are stored in the CRL, and once the period expires (new scheduled CRL), the CRL is signed by the CA and posted in the directory. A new CRL is then activated.

Every user must check the corresponding issuer's CRL when a certificate is received.

6.3 Certification Infrastructure Models

There are several models of certification infrastructures, and they are summarized in the following list:

- **Self−Signing Infrastructure:** The CA itself signs its own certificate, such that the verification

cannot be trusted. The only conclusion a user can obtain is that one key matches the other. However, this model is used by *Top Level CAs* (TCA), implicitly trusted, to sign their own certificates in order to provide certification services to other lower−level CAs.

- **Cross−Certification Model:** This model is related to the *Chaining* mechanism described in section 6.2.1, which consists of the exchange of certificates among the CAs.

- **Hierarchical Model:** The Hierarchical Model is widely used nowadays to establish infrastructures with multiple levels of CAs. Figure 6.3 depicts this model.

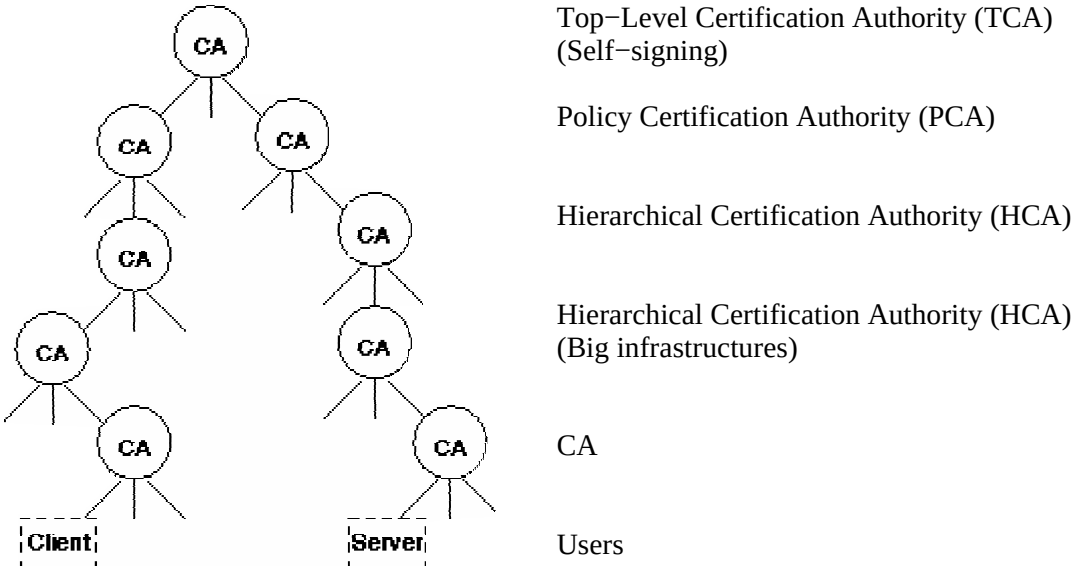

Figure 6.3 Hierarchical Infrastructure of CAs

- **Mesh Certification Infrastructure Model:** This model is based on higher−level CAs, and it is only considered a theoretical model. Figure 6.4 shows the structure of this model.

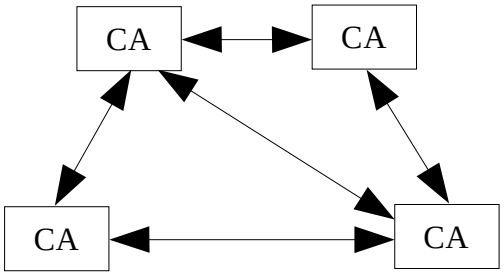

Figure 6.4 Mesh Infrastructure Model

- **Privacy Enhanced Mail (PEM) Certification Infrastructure Model:** The PEM Model is based

on the existence of an *Internet Policy Registration Authority* (IPRA). Figure 6.5 depicts the PEM Certification Model.

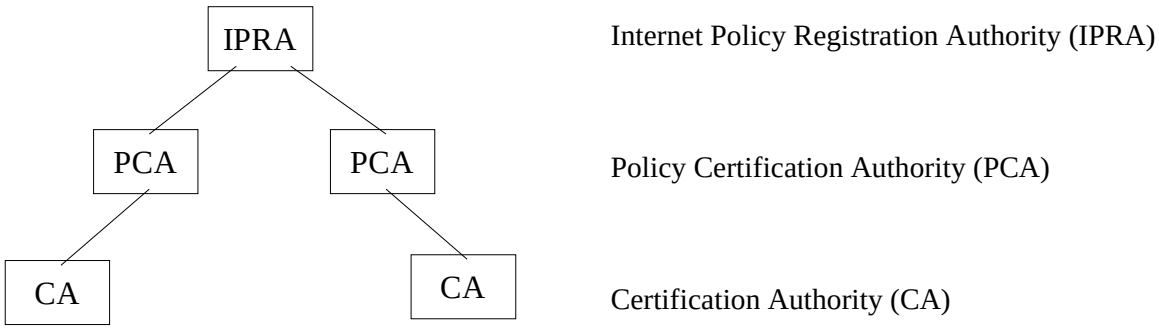

Figure 6.5 PEM Certification Infrastructure Model

The initiation of a common certification infrastructure is performed in three stages, described as follows:

1. The TCA issues its own certificate and signs it (self−signing).

2. Thus, the TCA is open to new certification requests by lower−level CAs. The TCA then issues the certificates requested and signs them.

3. The CAs that obtained their certificates in the stage 2, are now in turn able to accept more other lower−level CA’s certification requests.

Furthermore, TCA has his own certificate, second−level CAs have their own certificate and TCA’s certificate, third−level CAs have their own certificate, second−level CA’s certificate, and TCA’s certificate, and successively in the chain.

7 Introduction to Security Protocols and Related Protocols

A Security Protocol is a communications protocol that protects a message to be transmitted online, providing at the same time authentication services in some cases. In this section, some security protocols, such as Transport Layer Security (TLS)[17] and Internet Protocol Security (IPSec)[21], are briefly described, as well as several Key−Management Protocols used to negotiate the encryption and authentication schemes to be applied to the security protocol being used. Regarding the key−management schemes, this document briefly deals with Internet Key Exchange (IKE)[26], Diffie−Hellman Key Exchange as described in [41], Internet Security Association Key Management Protocol (ISAKMP) also as described in [41], or the new (and still IETF draft) Multimedia Internet KEYing (MIKEY)[29], specifically oriented to support the SRTP key management. All the protocols described in this section define a possible solution (together with SRTP and MIKEY, described later in this document) for the problem presented in this project. Hence , section 13 gives a more detailed description of the Secure Real−Time Protocol (SRTP), while section 14 deals with MIKEY, since they both become more relevant for our work.

7.1 Internet Protocol Security (IPSec)

7.1.1 Introduction, Applications and Benefits of IPSec

IPSec is a security protocol which works at the network−level layer (IP layer), and provides the capability to secure communications across a Local Area Network (LAN), across public and private Wide Area Networks (WANs), and across the Internet.

Placing the security at lower−level layers provides the following advantages:

- The security is transparent to users and applications
- Multiple connections are protected
- There is an automatic initiation
- It is globally available and interoperable

Examples and applications of the IPSec use include the following:

- Secure branch open connectivity over the Internet: by building *Virtual Private Networks* (VPNs) over the Internet or over a public WAN.
- Secure remote access over the Internet.
- Establishing extranet and intranet connectivity with partners.
- Enhancing electronic commerce security.
- Routing applications: IPSec plays a vital role in the routing architecture required for the internetworking, for instance assuring that a router advertisement comes from an authorized router.

Therefore, [41] lists the following benefits of IPSec:

- When IPSec is implemented in a firewall or a router, it provides strong security that can be applied to all traffic crossing the perimeter. Thus, traffic within a company or workgroup does not incur the overhead of security−related processing.

- IPSec in a firewall is resistant to bypass if all traffic from the outside must use IP, and the firewall is the only means of entrance from the Internet into the organization.

- IPSec is below the transport layer (TCP, UDP), and thus, as said above, is transparent to applications and users. This means that the applications must not change when IPSec is implemented in a firewall or a router, and users are not needed of a specific train on security mechanisms.

- IPSec can also provide security for individual users if needed.

7.1.2 IPSec Architecture

IPSec allows the involved parties to select the required security protocols includes in IPSec, determine the algorithms to be used, and put in place the keys needed for the cryptographic operations. The two protocols, included in IPSec, that provide security are the *Authentication Header* (AH) and the *Encapsulating Security Payload* (ESP).

The AH basically provides the data origin authentication of the packets, while the ESP provides confidentiality and, optionally, also authentication of the packets.

Table 7.1 shows the different security services provided by IPSec and which of them are specifically achieved with either the AH or the ESP, as described in [41].

Service / Protocol	***AH***	***ESP (encryption only)***	***ESP (encryption plus authentication)***
Access Control	✔	✔	✔
Connectionless integrity	✔		✔
Data origin authentication	✔		✔
Rejection of replayed packets	✔	✔	✔
Confidentiality		✔	✔
Limited traffic−flow confidentiality		✔	✔

Table 7.1 Services Provided by the AH and ESP Protocols

7.1.2.1 IPSec Transport and Tunnel Modes

Both AH and ESP support two modes of use: transport and tunnel mode. Following paragraphs briefly describe how each mode operates.

Transport mode provides protection for upper−layer protocols, and this protection extends

to the payload of the IP packet[12], such as TCP or UDP segments, or ICMP packets. Note that the payload is the data that normally follow the IP header. Transport mode is typically used for end−to−end communication between two hosts. AH in transport mode authenticates the IP payload and selected fields of the IP header, while ESP in transport mode encrypts (and optionally authenticates) the IP payload, and not the IP header.

As far as the tunnel mode is concerned, it provides protection for the entire IP packet. This is achieved by treating the entire packet plus security fields (AH or ESP fields) as the payload of new *outer* packet with a new IP header. With this encapsulation, the entire IP packet travels through a *tunnel* from one point of the IP network to another. Note that the intermediate points in this network, such as routers, are not able to inspect the *inner* packet. Tunnel mode is typically used when one or both ends of the communication are a security gateway (i.e. A firewall or a router implementing IPSec). AH in tunnel mode authenticates the entire inner IP packet and selected fields of the outer packet header, while ESP encrypts and optionally authenticates the entire inner IP packet. The use of IPSec in tunnel mode is the base for building Virtual Private Networks (VPNs).

7.1.2.2 Authentication Header (AH)

AH protocol provides support for data integrity and authentication of IP packets. Therefore, we can ensure that undetected modification of the packet content is not possible, as well as enable an end point to authenticate the source application or user, and filter traffic accordingly. We can also prevent *spoofing* attacks and provide replay protection.

AH is based on the use of MACs (described in section 5), which means that both communicating parties must share a key.

Figure 7.1 shows the format of the Authentication Header.

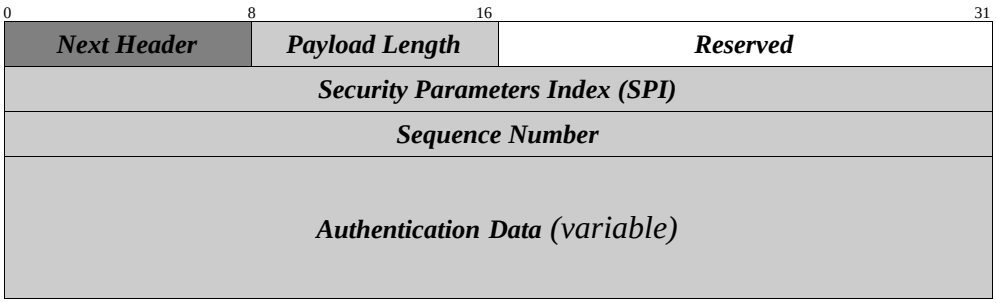

Figure 7.1 IPSec Authentication Header Format

The Sequence Number field is designed to counter replay attacks, and it is assumed that when this number reaches to $2^{32} - 1$, the sender terminates the association and another must be negotiated. Regarding the Authentication Data, it holds an *Integrity Check Value* (ICV). This is in turn a MAC or a truncated version of a MAC calculated over the payload, and IP header fields that does not change in transit (or whose change is predictable), while the other fields are set to zero for the calculation.

The AH is placed between the IP header and the payload in transport mode, and between

12. The term *packet* used in this section refers to either IPv4 or IPv6 packet.

the new outer−packet IP header and the entire inner packet in tunnel mode.

The authentication algorithm used in AH protocol is HMAC with either MD−5 or SHA−1 as the one−way hash function. See section 5.2.4 for further information about HMAC.

7.1.2.3 Encapsulating Security Payload (ESP)

ESP protocol provides two confidentiality services:

- Message contents confidentiality
- Limited traffic−flow confidentiality

Figure 7.2 shows the format of a ESP packet.

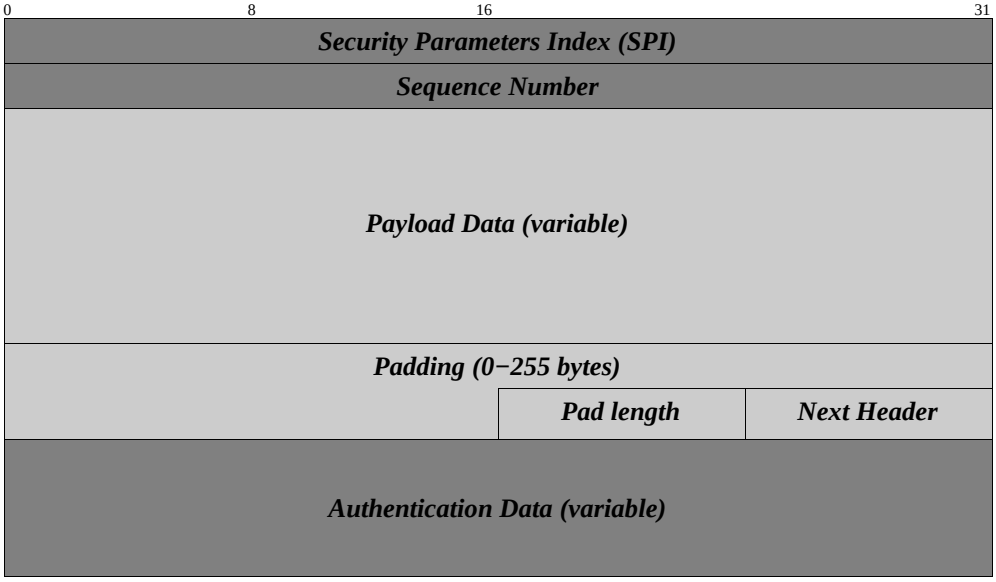

Figure 7.2 IPSec ESP Packet Format

In Figure 7.2, the light gray portion indicates the confidentiality coverage, while the authentication coverage covers the entire packet except the Authentication Data field. As occurred in AH protocol, the Sequence Number field provides anti−replay protection. The Payload Data corresponds to the transport layer segment or the inner IP packet, depending on the mode used, and the Authentication Data has the same purpose as for the AH protocol. The Padding field assures 32−bit words alignment for the encryption algorithm, and also provides partial flow−traffic confidentiality by concealing the actual length of the payload.

Algorithms, such as DES−CBC, 3DES, RC5, IDEA, 3IDEA, CAST, or Blowfish, are used for providing encryption to the packet.

7.2 Transport Layer Security (TLS)

7.2.1 Introduction, Applications and Benefits

Transport Layer Security (TLS) is the IETF standard for the *Secure Socket Layer version 3* (SSLv3), originated by Netscape mainly to protect World Wide Web traffic, so the protocol may also be referred to as SSL/TLS.

TLS sits between the application layer and TCP within the Internet Protocol stack, and was designed to provide a reliable end−to−end secure service, since as TCP, it is connection−oriented and stateful.

TLS is intended to secure end−to−end security associations, and it is very related to the use of Public Key Infrastructures (PKIs), mentioned in section 6 and thus, related to public−key cryptography. Establishing a PKI to enable TLS gives us one important advantage of TLS over IPSec: unlike IPSec, TLS may provide strong (mutual) peer authentication of the entities securely associated by TLS, as well as key management, since TLS defines a handshake protocol whereby entities agree on a cipher suite (for further data confidentiality and message integrity), establish the necessary key material, and authenticate each other. Regarding the applications, these should be slightly modified to support TLS service.

7.2.2 SSL/TLS Architecture

As said above, TLS is designed to make use of TCP to provide a reliable end−to−end secure service, and it consists of two layers of protocols: Record Protocol and Handshake Protocol, as illustrated in figure 7.3.

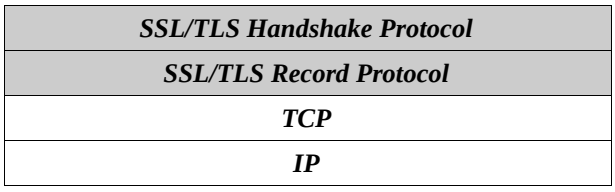

Figure 7.3 SSL/TLS Protocol Stack

The Record Protocol provides basic security services to higher−level protocols, such as the *Hypertext Transfer Protocol* (HTTP), while the Handshake Protocol provides cipher negotiation, key management, as well as mutual authentication between the entities involved.

7.2.2.1 SSL/TLS Record Protocol

The Record Protocol provides the TLS connections with two basic security services:

- Confidentiality: By using conventional encryption of the payloads, based on a shared secret key defined by the Handshake Protocol.

- Message Integrity: By using a message authentication code (MAC) based on a second shared secret key, also defined by the Handshake Protocol.

The overall operation of the SSL/TLS Record Protocol is shown in the figure 7.4 (based on figure 7.3 from [41]).

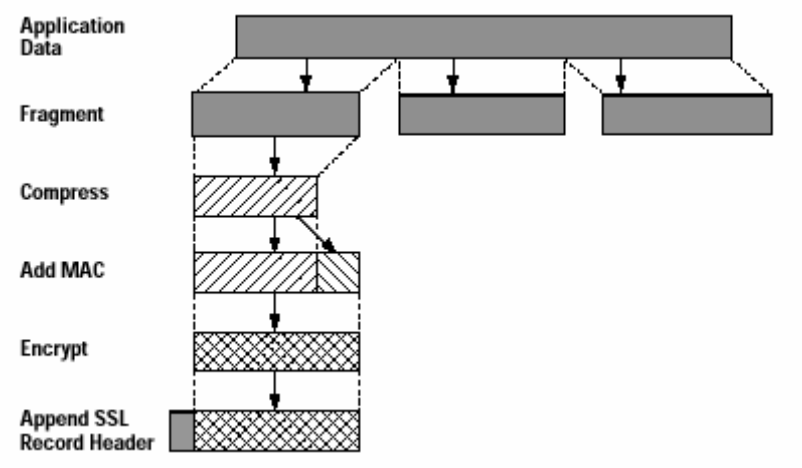

Figure 7.4 SSL/TLS Record Protocol Operation[41]

The Record Protocol takes the application data to be transmitted and fragments it into manageable blocks (2^{14} bytes), optionally compresses the block [13], computes a MAC over the compressed data (TLS standard defines the use of the HMAC, described in section 4, as the algorithm to compute the MAC), and encrypts the message plus the computed MAC using symmetric encryption (in light grey in Figure 7.5). The most important cipher algorithms defined as permitted are DES and 3DES. Finally, the Record Protocol appends a specific SSL/TLS Record Header to each block (in dark grey in Figure 7.5). This header consists of the following fields:

- Content Type (8 bits)
- Major Version (8 bits)
- Minor Version (8 bits)
- Compressed Length (16 bits)

Figure 7.5 shows the format of the SSL/TLS Record Blocks.

13. In SSLv3 and current TLS version, no compression algorithm is defined, so null compression algorithm is applied by default.

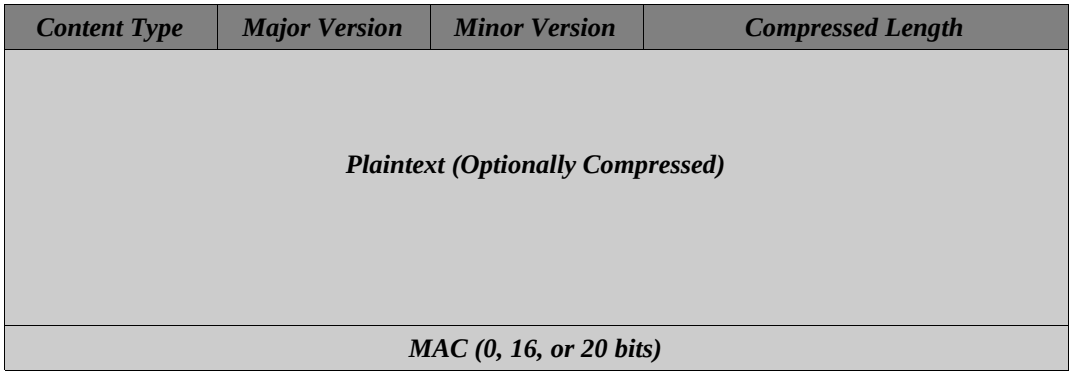

Figure 7.5 SSL/TLS Record Block Format

7.2.2.2 SSL/TLS Handshake Protocol

The Handshake Protocol is the most complex part of SSL/TLS and is used before any application data is transmitted. It allows the entities to authenticate each other, as well as negotiate an encryption and MAC algorithm and the keys to be used.

The Handshake Protocol consists of a series of messages exchanged between the entities involved. Each one of these messages has three fields:

- Type (1 byte): Identifies one of the 10 different types of messages, such as *hello_request, client_hello, certificate_request, certificate_verify, finished,* etc.
- Length (3 bytes): Length of the message in bytes.
- Content (≥ 1 byte): The parameters associated with this message.

The handshake protocol action can be illustrated as a four−phase process, as shown in the figure 7.6.

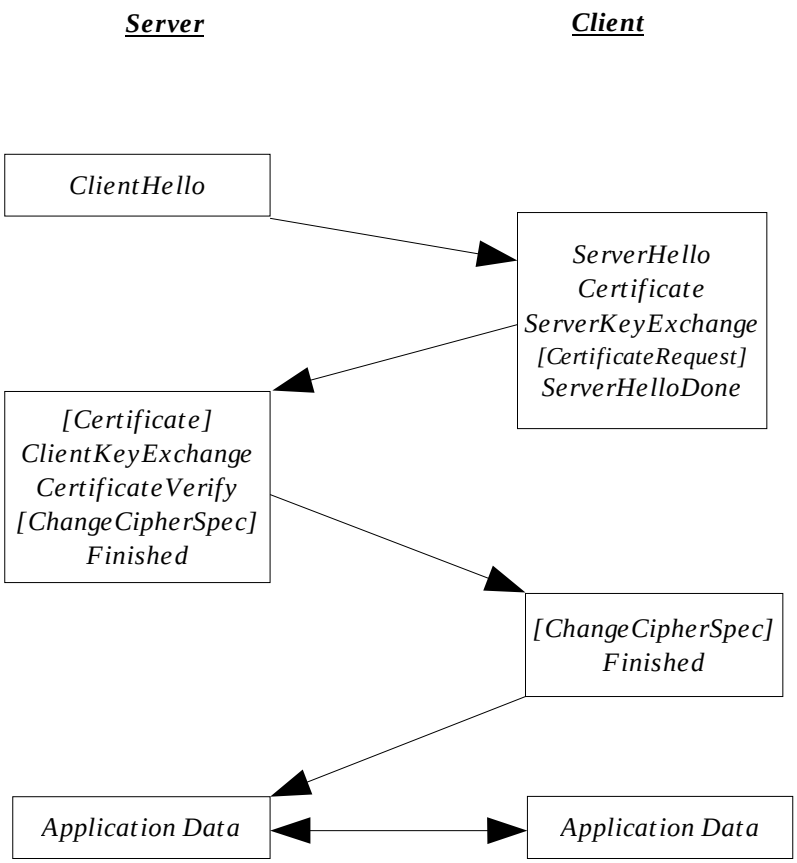

Figure 7.6 SSL/TLS Handshake Protocol Action

In Figure 7.6, the messages between "[]" are considered optional. A brief description of each phase is given in the following paragraphs.

- **Phase 1. Establish Security Capabilities.** This phase is used to initiated a logical connection and to establish the security capabilities associated with it, including protocol version, session identifier, cipher suite, compression method, and initial random numbers. This phase is initiated with a *client_hello* message containing the necessary parameters for the negotiation, such as lists of cipher suites or compression methods. The response is sent in a *server_hello* message, which selects the specific suites to be used.

- **Phase 2. Server Authentication and Key Exchange.** The entity acting as "server" sends its certificate (for instance one or a chain of X.509 certificates) to be verified by the entity acting as "client". Next, if it is required, a *server_key_exchange* message may be sent, as well as a *certificate_request* message if strong authentication is necessary. Finally, a *server_hello_done* is sent to terminate this phase.

- **Phase 3. Client Authentication and Key Exchange.** Upon receipt of the

server_hello_done message, the entity acting as "client" should verify the received certificate and check the *server_hello_done* message parameters. If the verification successes, the "client" entity is ready to send its certificate (if this was requested by the "server" entity). Next, a *client_key_exchange* and/or a *certificate_verify* message may be sent.

- **Phase 4. Finish.** This phase completes the establishment of a secure connection between entities. The most significant message exchanged is the *finished* message, after which both entities may start exchanging application layer data.

7.3 Key Management Protocols

7.3.1 Introduction

A Key Management Protocol is intended to be in charge of the distribution (and sometimes also the generation) of the necessary keys among the communicating entities, in order to provide them with the elements needed to protect the the transactions with confidentiality and/or message integrity.

There are several key management schemes, summarized in the following list:

- **Distribution of Public−Key Certificates**, with which we exchange the session keys encrypted with the receiver's public key. This method (based on RSA public encryption) is one of the many supported by SSL/TLS for the key exchange.

- **Diffie−Hellman Key Exchange**, also based on public key cryptography and indeed widely used. This method suffers the drawback that, in its simplest form, provides no authentication of the two communicating parties. However, there are more complex types of Diffie−Hellman based on the use of certificates, which are powerful alternatives. Several types of Diffie−Hellman key exchange are supported by SSL/TLS. Other protocols, such as the *Oakley Key Determination Protocol* used by IPSec, are based on this type of scheme.

- **Manual Key Management**, in which a system administrator manually configures each system with its own keys and with the keys of other communicating systems. However this method is deprecated, because of its obvious weakness.

- **Automated Distribution of Secret Keys** to be used by symmetric encryption, either by themselves or to derive session keys or one−time keys from them, such as *Internet Key Exchange* (IKE)[26], *Internet Security Association and Key Management Protocol* (ISAKMP)[53], or the still draft *Multimedia Internet KEYing* (MIKEY)[29]. IKE and ISAKMP are used together in IPSec, while MIKEY may be specifically used with SRTP to provide a *master key* from which the different session keys will be derived.

The following sections shortly describe IKE and ISAKMP protocols, as well as the Diffie−Hellman key exchange in its simplest form. The more relevant for our work MIKEY protocol is described in section 14.

7.3.2 IKE / ISAKMP

As said above, IKE is the standard key exchange protocol used together with ISAKMP in IPSec. ISAKMP provides a framework for authentication and key exchange without defining them, and is used together with IKE to provide authenticated key material to use with the IPSec protocols or to support other protocols such as SRTP.

Some benefits of IKE are:

- Avoids the manual specification of the secure session parameters
- Specifies a lifetime for the session security associations
- Provides anti−replay protection
- Assures confidentiality and authentication by different methods
- Allows the CA's support

ISAKMP defines two main phases: the establishment of the secure channel (previous agreement on methods to be used), and negotiation of security associations. Moreover, ISAKMP defines five different types of exchanges:

- Base Exchange: Key exchange and authentication material are transmitted together.
- Identity Protection Exchange: Extends the Base Exchange with protection of the identities of the communicating parties.
- Authentication Only Exchange: Performs mutual authentication without key exchange.
- Aggressive Exchange: Minimizes the number of exchanges by not providing identity protection.
- Informational Exchange: Used for the security association management.

Regarding IKE, it defines four different *modes* that can be used in either of the two ISAKMP phases:

- Main Mode: Equivalent to ISAKMP Identity Protection Exchange.
- Aggressive Mode: Equivalent to ISAKMP Aggressive Mode.
- Quick Mode: Used in the ISAKMP's second phase.
- New Group Mode: Used for define a new group for Diffie−Hellman key exchange.

7.3.3 Simple Diffie−Hellman Key Exchange

The purpose of the Diffie−Hellman Key Exchange algorithm is to enable two communicating parties to exchange a secret key securely that can be used for subsequent encryption of messages. The effectiveness of the algorithm depends on the difficulty of computing discrete logarithms. Note that, as said above, the simplest form of this algorithm is limited to the exchange of the keys, without providing authentication of the entities.

Next figure illustrates the Diffie−Hellman key exchange.

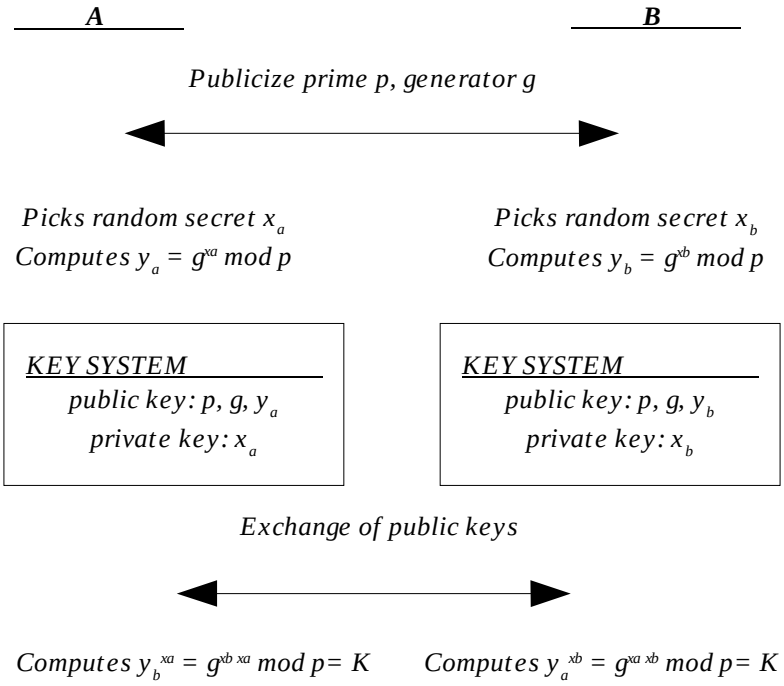

Figure 7.7 Diffie−Hellman Key Exchange

8 Objective: Enabling a Secure Mobile VoIP call

This section is an overall presentation of the problem to be solved within this project. It briefly deals with the different issues that the project consists of. Following sections will present a more detailed "picture" of the project and its requirements.

This research deals with the whole process whereby a dialogue between two mobile devices can communicate in a secure fashion, by using Voice over IP (VoIP), this is, by transporting voice (or any other type of multimedia application) over data networks. In this project I consider speech as the primary content to traverse the Internet. Basically, this process concerns a mobile phone call over IP−based networks between two end users. It does not consider, at this stage, other possibilities, such as video transmission or other multimedia applications. However, the protocol used to transport a multimedia stream between the users is Real−Time Protocol (RTP), so that the basic solution is the same regardless of the type of the media.

Adding voice to packet networks requires an understanding of how to deal with some issues and challenges, such as scalability, reliability, and of course, the two most important concerns of this project:

- Security during the whole process
- How this security affects the overall performance of the model

Voice over IP technology has become attractive, given the low−cost, flat−rate pricing of the public Internet. Needless to say, this makes sense since the intention of VoIP is to enable the users to make phone calls or, going further, to do everything they can do today with the PSTN over IP−based data networks, with a suitable quality of service and a superior benefit/cost ratio. Furthermore, the project will focus on the use of wireless technologies, particularly WLANs as the access network.

As already mentioned, the idea of a "dialogue" is similar to an ordinary mobile phone call using the PSTN. Basically, the process is considered to have three phases:

1. Call establishment
2. Conversation
3. Call termination

The first and the third phases are related, since the protocol used by the different entities involved in the communication, for both steps, is the Session Initiation Protocol (SIP). Regarding the conversation itself, as said above, it uses RTP to transmit the voice data. Given these considerations and assumptions, how to enhance security in the three different steps can be considered independently.

Other mobility aspects, such as *roaming*, or *hand−overs*, during the conversation are to be investigated further by others, and out of the scope of this paper. Despite of this, a brief description of these issues and aspects related to them is given in following sections in this paper, so as to facilitate future work.

By roaming we mean the ability to move from one access point’s coverage area to another one, without interruption in the service or loss of connectivity. As said above, the most suitable way to perform this in a secure fashion and with the best possible quality, has to be investigated further. However, I believe that the right path to follow is to assume that the support for mobility will be provided by *IPv6*[5] and *Mobile IPv6*[6]. Thus, as we will see later in this document, I have also assumed that the role of the *foreign agent* in

the remote domain will be performed by the *mobile node* itself, by making use of a *co−located care−of address*, as explained in [4].

The definitions of *roaming* and *mobility* could lead to some confusion, since some texts and researchers define mobility as "the ability of a terminal, while in motion, to access telecommunication services from different locations; and the ability of the network to identify and locate that terminal". We could define roaming as the ability to *connect*, or to get access to the Internet, or domains different than the user’s home domain or subnet.

9 Mobile Voice over IP: The Model and its Components

The scheme which this project is basically concerned with is called the Basic SIP Trapezoid, shown in the Figure 9.1. This section provides a detailed presentation of the problem to be solved in this project and describes its components.

9.1 Significant Components

The entities and components shown in Figure 9.1 are described in the following subsections.

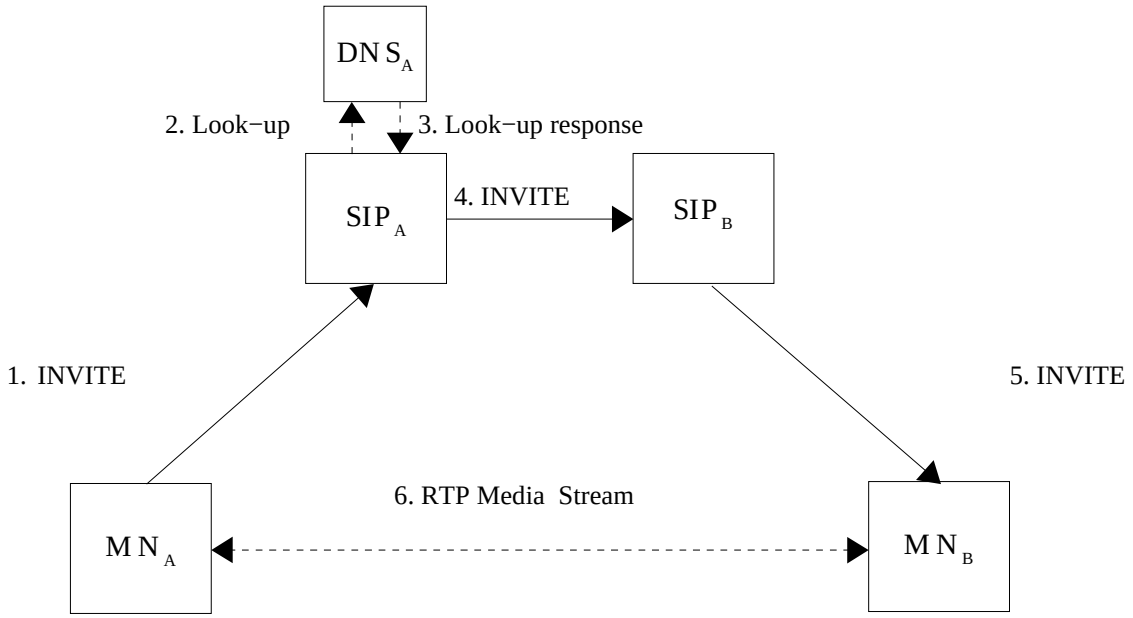

Figure 9.1. SIP trapezoid

9.1.1 Mobile Nodes

We can define a *mobile node* as a node that, as part of its normal use, changes its point of attachment to the Internet. I will consider these mobile nodes as the devices (cell phones or other wireless devices, such as laptops running the appropriate software) used by the final users to perform the call. They act as clients requiring services. In the figure they are labeled MN_A and MN_B, corresponding to users A and B. As said in the previous section, although given the implicit *mobility* of these nodes, this paper will not focus on this aspect, in particular it will not consider performing hand−overs during a conversation. Of course, these hand−overs could occur from one domain to another, possibly implying new authentication and access control policies handled by additional ISPs, different from the user's home ISP. In the case of this research paper, I will only consider the IP call from A to B, when both end−users are in their respective home domains.

Sometimes within this paper I will refer to the mobile nodes in terms of their *user agents*. The user agents are simply agent software running in the user's device.

9.1.2 SIP Servers

These are the Session Initiation Protocol servers: SIP_A in the user A's home domain, and SIP_B in the user B's domain. They are in charge of every SIP transaction, including those to establish or terminate the session. Thus, SIP is used for finding users and setting up and/or modifying multimedia sessions between them. The mobile nodes contact these SIP servers to establish and tear down their communications, using SIP protocol messages.

These servers are often called Proxy severs, since one of their tasks is to act as a proxy. There are four types of SIP servers, and they all (if necessary) can be located in the same machine in our architecture. These types are:

- **Location server:** used by a Redirect server or a Proxy server to obtain information about the called party's possible location.
- **Proxy server:** basically an intermediary that acts as both a server and a client for the purpose of making requests on behalf of other clients. These requests are served internally or transferred to other servers.
- **Redirect server:** accepts SIP requests, maps the address into zero or more new addresses and returns these to the client.
- **Registrar server:** accepts SIP REGISTER requests. Normally collocated with a Proxy or Redirect server, may also offer location services. The Registrar records in the Location server where the party can be found. Every user has to be registered with his/her own Registrar server.

9.1.3 DNS Servers

The DNS servers, such as DNS_A in Figure 9.1, are critical to finding the location of the user being called. In a first approach, this paper considers that the *Network Access Identifier* (NAI)[37] is going to be used to identify a client (e.g. bob@kth.se). In this case, a first search for that user's domain must be performed by the caller's SIP server, in order to locate the called party's SIP server and thus the user. Of course, the best way to locate the user is to use a DNS look−up in the caller domain's DNS server. It seems at this stage that the use of SRV or NAPTR records will be needed, as explained in [11].

9.2 The SIP Trapezoid

Let us consider the simple case of a mobile phone call from A to B (Figure 9.1) when both of them are in their respective domains. In this scheme, MN_A contacts its proxy (SIP_A) in order to phone user B as step 1. How SIP_A finds SIP_B is solved by a DNS look−up, in steps 2 and 3. Next, SIP_A communicates through proxy SIP_B in step 4, which in turn locates MN_B by using its Location server in step 5.

Finally, the media stream will be established between both users in the step 6. As said in previous sections, the users "dialogue" will make use of an RTP session.

Firstly, it was assumed that both users are registered in their respective domains before trying to

establish the session. This registration is handled between the user and his or her SIP server, by using a SIP REGISTER message.

Secondly, the way to initiate the session is by using a SIP INVITE message. This message, in the basic trapezoid case, will go from MN_A to SIP_A, which in turn will send the INVITE to SIP_B, and then to MN_B; as described above.

The termination of the call may be initiated by either of the users, regardless of who initiated the call. Termination is initiated by using a SIP BYE message.

Basically, this is what occurs [7]:

- Each user registers: REGISTER.
- One user invites another user to join a session: INVITE.
- The terms and conditions of a session are exchanged via *Session Description Protocol* (SDP)[8] bodies carried in the INVITE messages.
- Each user establishes their media stream (which uses RTP).
- One user terminates the session: BYE.

There are some other messages such as ACK or responses such as *Trying* involved in the process and not shown in the list above. For further information refer to section 3 in this document.

9.3 The SIP Registration

Every user must be previously registered with his or her SIP server. As described in [3], this registration allows users to update their current locations for use by proxy servers. This is performed by periodically sending REGISTER messages from the user's device to his or her SIP server (where the Registrar server is located). This action associates the user's identifier with the device he or she is currently using (this establishes a *binding*). This binding will be used by the Location server to locate the user when needed, as explained in the previous section[14].

9.4 The RTP Session

In voice calls, once the session is established, the media stream uses RTP as the transport protocol. RTP works via two separate, one−way streams between the endpoints (see [7], chapter 7).

RTP "*provides end−to−end network transport functions suitable for applications transmitting real−time data, such as audio, video or simulation data, over multicast or unicast network services*" (see [2]). A detailed description of RTP is given in section 3 in this document.

Another protocol, Real−Time Control Protocol (RTCP) is used by RTP as a control protocol to monitor data delivery, to provide minimal control, and identification functionality.

9.5 Other Components

Although out of the scope of the project, I would like to present, as briefly as possible, some studies performed at TSLab regarding mobility aspects of the architecture, as well as briefly describe some components related to this mobility (see [4]). This is done primarily to assure the reader of the feasibility of this scheme.

14. If we use Mobile IP to handle mobility, the home address of the MN will be registered with the SIP server.

Mobility introduces some new issues to deal with, regarding both security and performance of the model. This is mainly because new associations between new components appear. These new components are the *Home Agents* (HAs); *Authentication, Authorization, and Accounting servers* (AAA); and the *Access Points* (APs), as described in [9, 4].

9.5.1 Home Agents

Home Agents are nodes in the user's home network, that enable the mobile node to be reachable at its home address (which is registered with the SIP server), even when it is not attached to its home network. This node is critical when registering users[15] with their current attachment point, since every Mobile IP registration should be handled by it, considering that the first one of the registrations will be performed in coordination with the AAA server, and subsequent re−registrations by itself, according to RFC 3344[10]. The way this will be achieve is by providing the mobile node with a suitable IP address upon request.

9.5.2 AAA Servers

Authentication, Authorization, and Accounting servers are explained in [9]. If we assume session mobility from one network to another, quickly getting authorization to access the new network becomes very important. We have to ensure that the mobile node is authorized to access the network it is about to be attached to. The domain should grant access to the user only if he or she is actually authorized to use this network. This requires us to first perform an authentication which, if verified, is followed by an authorization phase. If access is granted, then the user can make use of the domain's resources. A suitable way to control this use (such as accounting) should also be provided. The AAA server in the user's home domain is called the *home AAA server*, while the one in a remote domain is called a *local AAA server*.

The local AAA servers have to contact the user's home AAA server in order to get and verify his or her credentials at the time of the first registration. Note that subsequent registrations in the same remote domain should avoid this interaction and be handled without again involving the user's home AAA server. This authorization could use RADIUS[12] or Diameter[13, 14]. Currently, the use of RADIUS seems more suitable, since Diameter is still a draft and few implementations are available. The AAA server should also be able to allocate a suitable address for the customer, this sometimes coordinated with the HA (see [9]).

Besides authenticating, authorizing, and initiating a user's accounting, AAA servers could also provide some kind of key−distribution during the initial registration, in order to provide security in subsequent transactions.

9.5.3 Access Points

Since we will use mobile technologies, such as WLAN, the access points (hardware devices or software that act as a communication hub) enable the user's wireless device to connect to a wired LAN. APs are important for providing improved wireless security and for extending the physical range of services a wireless user will have access to. APs could also act as *authenticators* and thus would be permanently in contact with the local AAA servers to enforce access control for mobile nodes within its coverage area.

15. Note here that this Mobile IP registration is different from the SIP Registration described in section 9.3.

Thus far the problem has been presented. The next sections will present the alternatives to solve it, giving some possible solutions to secure the SIP environment (including the DNS look−up), as well as to secure the RTP stream. Finally, the solution chosen is described in section 11.

10 Alternative Solutions for Secure Mobile Voice over IP

The services to be provided according to the requirements of the model are carefully defined, and there are several solutions and alternatives that can be applied to secure Mobile Voice over IP in order to provide those services.

10.1 Security Requirements of the Model

One of the main points when developing Mobile VoIP is its performance, with regard to latency, delay, voice quality, and interoperability. However, we should not overlook security issues. VoIP security needs to be handled in the overall context of data security, providing a suitably secure infrastructure to support the whole process of making, using, and terminating a call.

We should try to provide an architecture with the needed security mechanisms in order to protect each one of the phases of this communication. With these mechanisms we will try to avoid many different kinds of attacks, such as DoS attacks, packet spoofing, reply attacks, or message integrity violations, as well as eavesdropping of the media session.

As an initial approach, we believe that the most suitable mechanism for protecting the media stream would be the use of an extension of RTP called Secure Real−Time Protocol (SRTP), to encrypt the flow. For the rest of menaces, a trust hierarchy must be set up, in order to provide the process with mutual authentication, data authentication, privacy, and integrity.

Many of the potential security mechanisms are already considered as standards, for instance *HTTP Digest*[16] or *Transport Layer Security* (TLS)[17], for SIP security. Providing a design of an architecture with a complete secure infrastructure is the aim of this project. Using these mechanisms and/or adding others depends on a thorough study of them in the context of this problem.

We must consider that a lot of security associations must be established to perform a single simple call. These associations include securing all the different phases in the process, both the RTP stream and the SIP interaction between various entities. Regarding SIP security associations, they must include **all** the interactions between mobile nodes and servers, as well as between the servers themselves. This means that we must secure the registration of the users, along with the establishment, and the termination of every call. Thus, from a security point of view, all communication needs to be secured, that is, authenticated and encrypted.

As to the requirements themselves, we can divide them into functional requirements, such as total privacy of the exchanged data, and safe storage of the accounting information in the servers (call data records for the purpose of billing should be physically secured); and technical requirements. The latter are summarized in the list below:

- All the connections between elements of the architecture must be encrypted.
- Each one of the end−points of these intermediate connections must always authenticate each other (strong authentication).
- End−to−end authentication between mobile nodes must also be provided.

For this purpose, as said in preceding paragraphs, the creation of a suitable infrastructure is needed. This infrastructure could be a *Public Key Infrastructure* (PKI)[19] or a Distributed Key scheme, such as a *Kerberos*[20] model, although the latter has not been deeply studied so far. As we will see later in this paper, the PKI scheme seems to be the most suitable solution.

We will need to separately deal with each of the elements to be secured. This means that the security of the media stream (by using SRTP or another means such as IPSec) must be treated independently. Hence, the first challenge will be how to secure the SIP protocol.

10.2 Securing SIP

Securing Session Initiation Protocol is far from trivial. Furthermore, note that it is different the interaction between users and servers and that between server themselves. The main network security services required by SIP are the following:

- Preserving the confidentiality and integrity of the messages
- Preventing replay attacks or message spoofing
- Providing authentication of and privacy for the participants in a session
- Preventing DoS attacks

There are multiple alternatives to solve these problems. Of course, these alternatives may be used by themselves or in conjunction to achieve our final goal. The following paragraphs present all these alternatives. Some of them are security protocols, such as IPSec or SSL/TLS, probably combined with other security mechanisms, such as HTTP Digest.

In order to negotiate the most suitable scheme, SIP entities could make use of a negotiation agreement mechanism, as described in RFC 3329[22]. This process would perform a negotiation between a user agent and its first hop destination (SIP server), where the client selects a security mechanism (TLS, IPSec, TLS/HTTP Digest, etc.) from a list given by the server.

The SIP specification establishes HTTP Digest as the default authentication mechanism. *Basic authentication* was deprecated because of its weakness. Both mechanisms are described in [16]. *CHAP password authentication*[23] could be another alternative to provide this mutual authentication. These mechanisms may also be used in conjunction with IPSec to provide peer−to−peer mutual authentication, if the use of TLS (described in the next subsection) were finally discarded.

On the other hand, the use of SSL/TLS in a PKI is the most powerful alternative to IPSec, even if the client is not able to get a certificate when this TLS scheme was used. As a matter of fact, the usual authentication in TLS is from the server to the user in the form of one−way authentication, rather than mutual authentication[16]. The user might be authenticated to the server by utilizing some sort of challenge (i.e introducing a username and a password), as commonly established in a usual TLS connection nowadays.

As far as the security in the DNS look−up is concerned, this phase has not yet been thoroughly studied, although the use of *DNSSEC*[24] seems to be the right approach. The motivation for securing the DNS look−ups is to avoid these look−ups return wrong or malicious SIP server addresses.

16. This occurs because few clients posses a certificate nowadays.

10.2.1 Using SSL/TLS in a PKI

The use of a PKI in this architecture would enable the use of TLS. Establishing TLS connections between the involved entities (either mobile nodes or SIP servers) would provide the SIP architecture with the necessary security services during the process. These services, as [25] explains, would be:

- Strong authentication
- Cipher suite negotiation (for encryption and hashing)
- Dynamic key distribution
- Encapsulation format for the protected data stream

By using the TLS Handshake Protocol we can authenticate (with mutual authentication) each entity involved in the call (mobile nodes or SIP servers), as well as establish the parameters for the encryption and hashing mechanisms to ensure the protection of the exchanged data. We would set, in this way, the basis for data authentication, privacy, and integrity (obtaining hop−by−hop message protection between mobile nodes and SIP servers, as well as between SIP servers themselves), as well as indirectly getting a non−repudiation service in those phases of the process when the digital signature is used. Therefore, TLS could secure every security association along the process, from the registration to the termination.

With this PKI, entities would utilize certificates (with their private and public keys). An initial consideration of this approach shows us the fact that the use of the certificates for the servers would be easily achieved, although this is probably not completely true considering the users themselves, since they (the users) usually lack certificates. However, the most commonly used TLS scheme assumes this lack, and authentication is considered a one−way authentication from the server to the user. Hence, a challenge mechanism would be added to authenticate the users. In this case, some sort of username−password authentication would be used to authenticate the user to the server over the established TLS tunnel.

Computing or transmitting these certificates (given the typical constraints and limited bandwidth in wireless environments), ends up decreasing the performance. However, the ability to bypass the initial expensive public key authentication if the server has recently authenticated the client and established a shared secret key (*session resumption)* is allowed by SSL/TLS, whenever the server remembers the session secret.

By establishing a PKI, we would achieve:

- Message authentication and integrity (by generating and digitally signing *Message Authentication Codes*, MAC).

- Confidentiality (distributing previously negotiated session keys by using digital envelopes).

- Non−repudiation service (via digitally signing).

- Peer−to−peer strong authentication (by using the handshake protocol within TLS or adding a challenge authentication scheme for the client authentication).

- Enabling *Secure Multipurpose Internet Mail Extensions* (S/MIME)[28] (which can use the certificates) to secure SDP bodies within SIP messages (and probably more), as we will see in the following paragraphs.

10.2.2 Using IPSec

As described in section 7.1, IPSec is a protocol developed for transmission of sensitive information (such as VoIP traffic) over unprotected or untrusted networks. It acts as a network−layer security protocol that protects and authenticates IP packets exchanged between IPSec devices or peers. These message authentication and encryption services are independent of the key management protocol used to set up the security associations and session keys. Indeed, there are two possible ways offered by IPSec for the key management:

- Manual keying
- Automatic keying

In the manual scheme, keys are manually installed and configured by the administrator. This makes it vulnerable to attacks where an intruder gains control of the server, therefore we must deprecate the use of this scheme in our model.

On the other hand, in the automatic scheme, keys are negotiated by the entities forming a security association themselves. This negotiation may be performed by using the Internet Key Exchange (IKE)[26] protocol. These security associations may also be (are supposed to be) refreshed (*re−keying*) without requiring administrator intervention.

Some advantages of using IPSec would be:

- Provides security without changing the interface to IP.
- Within an organization, the use of IPSec in tunnel mode enables the creation of *Virtual Private Networks* (VPNs), protecting the communication between entities.
- Unlike TLS, upper layer protocols are not supposed to be changed to invoke security, and need not even be aware that their traffic is protected at the IP level, this is, it is *completely* transparent to the application. *Thus, such application needs no changes*[17].
- Provides hop−by−hop security in a really simple fashion, which would be extremely important in our architecture.

Unfortunately, there are some disadvantages as well:

- IPSec increases processing costs and communication latency, as sender and receiver perform cryptographic operations. Furthermore, as studied in [27], if IKE were finally used as a key−management protocol, it could affect in an important way the performance, having an alarming effect on it (although this fact, and whether this latency is greater than that due to a GPRS multiframe should be investigated further).
- IPSec may provide security for all upper layer protocols, but it also creates overhead for all of them.
- As IPSec does not prescribe any particular key−management protocol, although this allows different nodes to pick their favourite scheme, these have to be negotiated and agreed upon before they can protect the traffic.

17. Section 11.3.1 in this document presents a possible refutation for such assumption.

- The lack of a handshake protocol (unlike in TLS) makes IPSec incapable of providing peer−to−peer authentication by itself. This must be provided by, for instance, HTTP Digest or CHAP authentication, or even by the key management protocol if this is provided with the authentication service.

10.2.3 Securing SDP bodies and SIP headers

As the SIP messages need to be inspected in some intermediate steps of the communication, we cannot end−to−end encrypt the SIP headers. The SIP specification recommends the protection of this data by using TLS tunnels in a hop−by−hop way, although IPSec would be another suitable alternative. Note here as well, that this data would be vulnerable in the intermediate points.

Other alternatives considered in this paper assume that these intermediate points are trustful (for simplicity of the model); or consider the use of cryptographic attributes in SDP, as described in [38]. The latter has not yet been studied.

Given this, for headers needing protection, because the SDP body is of MIME type, they could all be moved, if needed, to one SDP MIME body and protected by S/MIME in order to provide header authentication and integrity. Thus, SDP MIME bodies can be protected end−to−end.

However, the problem faced is that to define what fields of the SIP header a proxy server should be allowed to modify. Furthermore, we will see later along this document that this protection is probably not needed if the key−management protocol (e.g. MIKEY) is "encapsulated" into SIP, and protects itself.

Using S/MIME to secure the SIP headers, as well as the SDP body, would require a PKI as we saw in the TLS description section, since end−points will need certificates.

However these aspects should be investigated further by others.

10.2.4 Securing the DNS Look−Up

Unfortunately, this issue has not yet been deeply studied. In spite of this, as said at the beginning of this section, it seems reasonable to use DNSSEC to protect this phase of the process. As said before, the motivation for securing the DNS look−ups is to avoid these look−ups return wrong or malicious SIP server addresses.

10.2.5 Conclusions

The PKI option of using certificates seems to be the most suitable scheme to protect this part of the architecture (registration, establishment, and termination), if used along with a DNSSEC scheme to protect the look−ups. This would enable TLS connections to be established, achieving all the security requirements as noted thus far. If the client does not support the use of certificates (the most commonly used TLS scheme), providing the architecture with a challenge mechanism in order to authenticate the client to the server, such as HTTP Digest, would enable us to continue to consider this option.

For both cases, S/MIME protection of the SIP messages is, at this stage, probably needed in order to ensure that the sensitive data is encrypted except while being processed in

intermediate points, such as untrusted proxies, although we must note that the support of certificates would again be required. However, we will see in next sections that the use of S/MIME for this purpose has been finally deprecated, since it may not be needed if we consider the intermediate points trustful (simplicity for the model).

If a previous negotiation of the security mechanism were needed, we have seen that SIP provides an agreement mechanism in order to negotiate which one of the different solutions will be finally adopted. Fortunately, this negotiation could be performed during the first steps of the SIP requests, such as the REGISTER or even the INVITE messages, as described in the RFC 3329.

10.3 Securing the Media Stream

This section provides a brief introduction to the different alternatives we have considered to secure the media stream. It is divided into two sections: *Transport Protocol* and *Key management*. The former briefly describes the possible solutions to secure the flow itself, while the latter presents the different protocols which can be used to provide the model with the key management scheme needed to support the media stream security (and may include mechanisms to authenticate the end users to each other).

10.3.1 Secure Transport Protocol

As said in preceding paragraphs, the media stream must somehow be protected. In this case, the stream would be a voice flow between the end users, A and B, and it is transported using RTP. This RTP flow must be encrypted and integrity protected, and the keys must be derived as a result of an initial strong authentication between the users (see [25]).

In order to protect this media stream, one powerful alternative is the IETF draft for Secure RTP (SRTP), which also specifies key derivation and data encapsulation. The SRTP draft defines SRTP as "a profile of the Real−time Transport Protocol (RTP), which can provide confidentiality, message authentication, and replay protection to the RTP/RTCP traffic"[15].

The use of IPSec is an alternative for protecting the data at the network−layer, and even to protect, if needed, some aspects of the stream such as the RTP headers (which are not protected by SRTP). This protection should be performed after considering users end−to−end policies to determine if doing so is required.

However, the use of SRTP (more specifically oriented to RTP traffic) seems more efficient when dealing with firewall and *Network Address Translation* (NAT) traversal (and other issues), although the presence of such components has not yet been considered, and thus, this is an assumption. Moreover, the cost of using SRTP to protect the media stream might be considerably low, given its features (such as the low packet expansion). These features are further described in section 13.

10.3.2 Key Management

As said above, the keys for protecting the traffic must be derived as a result of an initial strong authentication between the users. Negotiating a suitable key−management scheme (MIKEY, IKE, or other) or extensions to SDP are available and described in [30].

Regarding the key management protocol needed to support SRTP, a powerful alternative is the use of *Multimedia Internet KEYing* (MIKEY)[29] , since it is close to both SIP and SRTP.

What MIKEY would provide is described in more detail in sections 11, 12, and 14.

There are several other alternatives, such as for example IKE, although MIKEY seems to be more specifically oriented to supporting SRTP.

The cryptographic SDP attributes (see [38]) seem to be (thus far) an additional feature, rather than an alternative to the chosen key management protocol.

11 A secure Model for Mobile VoIP

The previous section showed us all the different solutions that can be applied to the architecture in order to provide the model with the required security services. However, there are many factors involved in the problem. The most important of these factors is the process performance. This section provides a solution for Secure Mobile VoIP. The solution presented here does not, of course, overlook the security requirements, nor the performance aspects. A rationale for the selection of each security component is also given in this section.

11.1 Overview of the Model

The model proposed in this section in intended to be an efficient solution to the problem exposed in section 9. This model is the result of a thorough investigation performed at TSLab, IMIT, KTH, Stockholm, in order to provide a single Mobile VoIP call with the necessary requirements regarding security, as well as evaluating the performance of the process.

The ideas concerning the protocols and mechanisms to use that my colleagues and I have come up with are summarized in the following list:

- The use of **SRTP** to protect the media stream between end users. Our own implementation of SRTP (*MINIsrtp)* is available and described in section 13. MINIsrtp utilizes the AES algorithm for encryption. This implementation is based on the *libsrtp* libraries[33].

- The use of **MIKEY** as the key−management protocol to support SRTP session establishment, as well as to protect the negotiation scheme. This would also provide the desired end−to−end mutual authentication.

- The use of **TLS** to protect the SIP transactions (providing the necessary hop−by−hop security). At this time, the use of S/MIME which would only be suitable to protect certain SIP payloads (hence, it is out of the current scope of our testbed) is deprecated.

- The use of **DNSSEC** (as an initial approach) to secure the correspondent look−up.

- The use of ***SIP Express Router (SER)***[31] as the software to perform the whole process in the SIP servers. The use of ***VOCAL***[7] has also been considered, although the support for DNS look−ups of SER is of special interest for us, considering that none of them supports a TLS scheme.

- The use of ***MINISIP***[32], developed at TSLab by Erik Eliasson, as the user agent software, enhanced with the necessary security aspects (SRTP, MIKEY, etc.).

Figure 11.1 depicts the overall picture of the model, while the rationale for these choices are described in the following subsections.

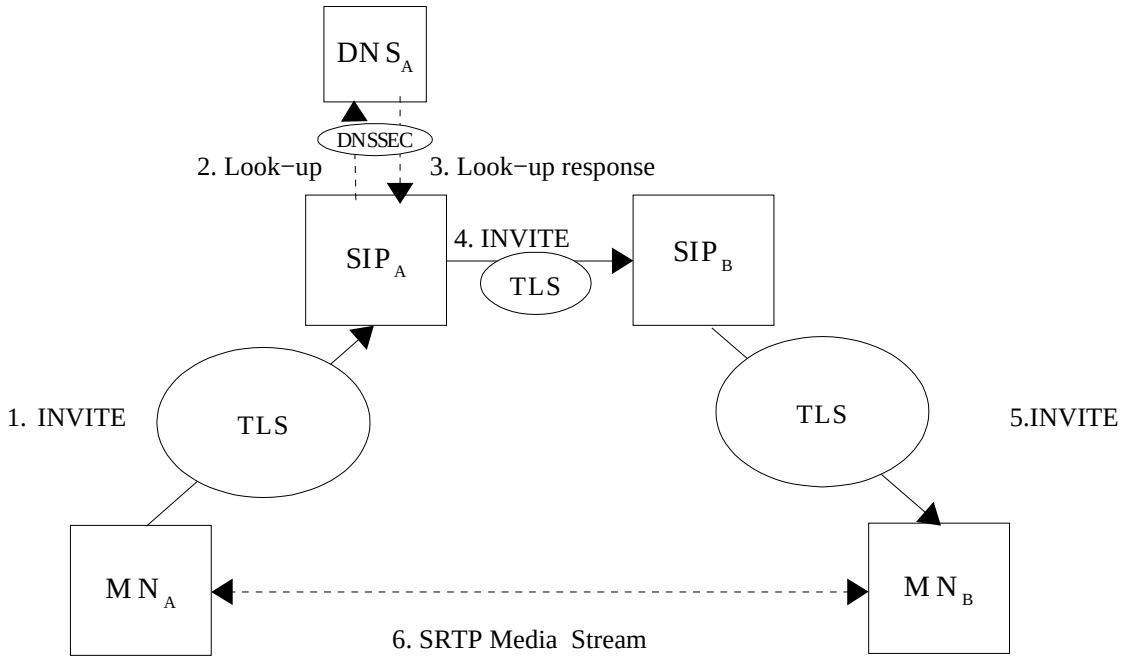

Figure 11.1 Security−Enhanced SIP Trapezoid Model[18]

11.2 Interoperation of the Components

How these components interoperate is described as follows:

- Firstly, we must set up a PKI. This allows us to protect the different security associations by using TLS. As said before, HTTP Digest challenge mechanism (or other password−based authentication mechanism) would help if the client does not support the use of certificates (the most common situation). However, I have assumed at this stage the ability of the clients to support certificates.

- We will utilize DNSSEC in order to provide the DNS look−ups with the necessary security.

- We will use our own implementation of SRTP to secure the RTP stream. This implementation, described in section 13, is based on the IETF SRTP draft, the MINISIP RTP implementation, and some C libraries from [33], called *libsrtp*[19], among others.

SRTP provides session key derivation for the encryption and authentication of the exchanged data. Hence, a master secret key is needed, and thus, a key−management mechanism is needed to provide it and to initialize the cryptographic context for using SRTP. The key−management protocol proposed would be MIKEY, which provides:

- Mutual authentication of the end users, which in turn can be set up in three different ways. These are *pre−shared key*, *public key*, or *signed Diffie−Hellman*. The latter seems to be a priori the most suitable for our purpose, and it would be supported by the presence

18. Although not shown in the figure, the INVITE SIP messages carry the MIKEY message used for the key scheme negotiation
19. *libsrtp* is copyright protected: Copyright (c) 2001, 2002 Cisco Systems, Inc. All rights reserved.

of the PKI.

- Negotiation of the cipher−suite which will be used to secure the RTP flow, and initialization of the cryptographic context.

- Establishment of a master secret key, which in turn would be used to derive the SRTP session keys.

Regarding the implementation and given all these premises we decided to follow a "bottom−up" approach. In this way, we established a simple RTP session between two MINISIP user agents. Once this simple goal was achieved, the next step consisted of establishing the same flow, but using in this case an SRTP session instead. Next steps would comprise the MIKEY addition to this scheme and finally we would consider the SIP and DNS security. As said earlier, the introduction of mobility is out of this project's scope.

A final analysis of the model and some measurements are described in section 15.

11.3 Rationale

In this section the rationale for the choices and decisions of the model is given. Furthermore, the answers to the following questions are given:

- Why use TLS to protect SIP?
- Why use DNSSEC to protect the DNS look−ups?
- Why use MINISIP as the user agent?
- Why use SRTP to protect the media stream, instead of IPSec over VPNs?
- Why use MIKEY as the key−management protocol to support SRTP sessions?

11.3.1 TLS supported by a PKI

The use of TLS supported by a PKI provides the model with the necessary security requirements, such as peer−to−peer mutual authentication of the entities (by using the SSL/TLS Handshake Protocol), hop−by−hop confidentiality and message authentication, and a dynamic negotiation of the cipher suites and key distribution. The use of certificates and public−key cryptography to establish the connections and distribute the keys is the most efficient solution for our purpose.

In our case, the placing of the security mechanisms at higher−layers of the stack, rather than at lower−layers (IPSec) is preferred since the use of IPSec implies greater protocol processing costs and higher communications latency, besides not providing a handshake protocol for authenticating the entities. The limited bandwidth and other constraints existing in wireless environments enforces this decision. Furthermore, despite the common belief, applications must change to take **full** advantage of IPSec (see Section 16.1 in [54]). Moreover, the IPSec security associations identify (trust) devices rather than sessions or applications.

11.3.2 DNSSEC

This point has not yet been deeply studied and compared to other alternatives. Hence, the reasons for choosing DNSSEC to protect the DNS look−ups is based on a straight−forward decision, rather than on a detailed study.

11.3.3 The User Agent: MINISIP

MINISIP is an implementation of a SIP user agent to be used in Mobile VoIP scenarios. It was developed at TSLab, IMIT, KTH, by Erik Eliasson. The decision of enhancing MINISIP with security mechanisms, such as SRTP and MIKEY, was based on the simplicity and good performance of the implementation.

11.3.4 SRTP vs. IPSec and VPNs

The functionality provided by SRTP and IPSec is very similar as far as our security requirements are concerned. However, the relatively high bandwidth consumption of IPSec forced us to search for other alternative. SRTP provides a suitable security functionality, but a lower bandwidth cost. Its low packet expansion (unlike VPNs over IPSec), its high throughput, and its specific orientation of SRTP to protect RTP as a profile of this, together with the use of the low−cost, high−speed AES as the SRTP encryption algorithm, made us expect a very good performance of the model when using this solution, rather than IPSec and VPNs. Besides, the use of SRTP (more specifically oriented to RTP traffic) seems more efficient when dealing with firewalls and *Network Address Translation* (NAT) traversal, although the presence of these two components must be investigated further by others. Moreover, as for the SIP security, the IPSec security associations identify (i.e. *trust*)devices rather than sessions/applications. This would be a disadvantage when having several sessions/applications running in the same device, some of which may be trusted and some others not.

A complete description of SRTP and its implementation is given in section 13, while the source code is given in Appendix A. Note that SRTP is still an IETF draft, and it has not been yet thoroughly analyzed.

11.3.5 MIKEY

MIKEY is a key−management protocol specifically oriented to support security protocols for real−time applications, such as SRTP. MIKEY provides the key needed to derive the SRTP session keys, and is compatible with SIP message transactions (it may be integrated in SIP).

MIKEY provides the following features[29]:

- End−to−end security
- Simplicity
- Efficiency: low bandwidth consumption, low computational workload, small code size, and minimal number of roundtrips
- Tunnelling: Possibility to "tunnel"/integrate MIKEY in session establishment protocols (e.g. SIP)
- Independent of any specific security functionality of the underlying transport

A complete description of MIKEY is given in section 14, along with a framework for its use in this project. As for SRTP, MIKEY is still an IETF draft, and it has not been yet thoroughly analyzed.

12 SIP Security

This model for Secure Mobile VoIP needs the establishment of a PKI which supports the SSL/TLS security associations in order to secure the SIP protocol. This section provides some hints on how to start working with a PKI supporting TLS to secure SIP. Further investigation on this issue must still be performed by others in future work. More information beyond this document can be found in the SIP RFC[3].

12.1 Background

Section 10.2.1 presented the different aspects regarding the use of a PKI supporting TLS for providing the SIP protocol with the necessary security. Moreover, section 11.3.1 gave a rationale for the selection of TLS supported by a PKI. The following list briefly summarizes those sections:

- TLS supported by a PKI provides: Strong Authentication, Dynamic Key Distribution, Cipher Suite Negotiation, Message Authentication, and Confidentiality.
- Suitable Security Associations along the process, from the establishment to the termination, by securing hop−by−hop the SIP messages, such as INVITE or REGISTER.
- Session resumption.

Furthermore, the PKI may be used to support MIKEY (providing end−to−end authentication) when the basic exchange method selected is Signed Diffie−Hellman (see section 14.2.8).

12.2 TLS within SIP

TLS can be specified as the desired transport protocol within a Via header field value or a SIP−URI. The identifier in this case will be *tls.*

The use of a SIPS−URI scheme, that signifies that each hop over which the request is forwarded, until the request reaches the SIP entity responsible for the domain portion of the Request−URI, must be secured with TLS, is described in detail in the SIP RFC.

Using TLS for the REGISTER messages should be simple: the user and its SIP provider could have certificates signed by a common CA at the time of the user signing up. The user could store the necessary certificate information in a smart card or in a file (PKCS#12), which he/she inserts or installs respectively in the SIP device.

For other SIP messages (such as the INVITE message) the situation becomes more complex, and we should distinguish between two different cases (Figure 12.1 depicts such distinction in the case user A calls user B):

1. On the hop between the sender user agent (UA) and its outgoing SIP proxy server, the already established TLS *tunnel* could be reused. The same goes for the (last) hop between other SIP proxy server and the receiver UA.
2. On the other hand, for the hop between the SIP proxy servers a PKI is needed for large scale deployment. Establishing such a PKI will take some effort.

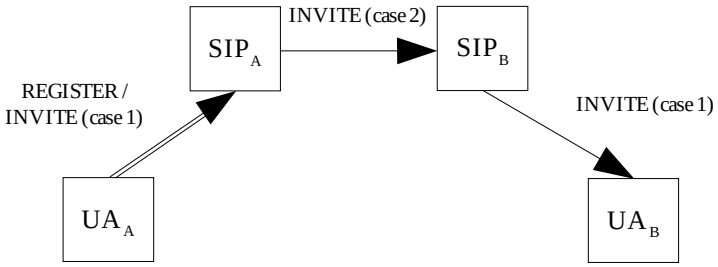

Figure 12.1 Distinction between the SIP INVITE messages regarding the establishment of a PKI

12.3 A First Approach

Although a PKI used in large scale would be needed, a good way to get started is by installing only certain certificates, since the most of your calls are usually made to a few persons. As said above, this simple PKI will support the Signed Diffie−Hellman exchange in MIKEY, providing the needed end−to−end security.

13 Secure Real−Time Protocol (SRTP)

The Secure Real−Time Protocol is still a IETF draft. The previous section gave us the rationale for using SRTP in our model. This section describes the protocol and presents a first version of an implementation for it, called MINIsrtp.

13.1 SRTP Description

SRTP provides message authentication, integrity, confidentiality, and replay−protection for unicast and multicast RTP applications, by previously using the appropriate key−management protocol[15]. In our case, we have chosen to use MIKEY as the key−management protocol. MIKEY provides the SRTP with a master key, which is used to derive the session keys needed to encrypt and authenticate the messages. MIKEY would also be in charge of the cryptographic context initialization.

SRTP has a **high throughput** while at the same time having **low packet expansion** (unlike IPSec in tunnel mode over VPNs, which expands considerably the size of the packet), and offers a **suitable protection** by using transforms based on an additive stream cipher for encryption and a keyed−hash function for message authentication. It also provides an implicit index for sequencing and synchronizing. This will be based on the RTP sequence number for SRTP and on the index number for Secure RTCP. Regarding the encryption cipher and the keyed−hash, the first implementation of SRTP will include the mandatory−to−implement algorithms (AES in Counter Mode and Null Cipher). The strong point as far as the ciphers are concerned is the use of AES, since its computing cost is very low (see section 5.2.1).

All these features seem to make the use of SRTP along with AES the most powerful alternative to protect the media stream.

SRTP resides between the RTP application and transport layer. For a better clarity, we distinguish the SRTP sender and receiver sides. On the sender’s side, SRTP intercepts an RTP packet, builds the corresponding SRTP packet and sends it to the receiver. On the receiver’s side SRTP intercepts the incoming packet (SRTP packet), extracts from it the RTP packet, and passes it up the stack.

13.1.1 SRTP Packet

The SRTP packet format is nearly the same as the RTP packet format. The SRTP packet header is identical to the RTP one, but with two new optional fields : *MKI* and *Authentication tag*. Figure 13.1 depicts the SRTP packet format. A description of the RTP/SRTP header fields has been given in section 3.2.3 in this document.

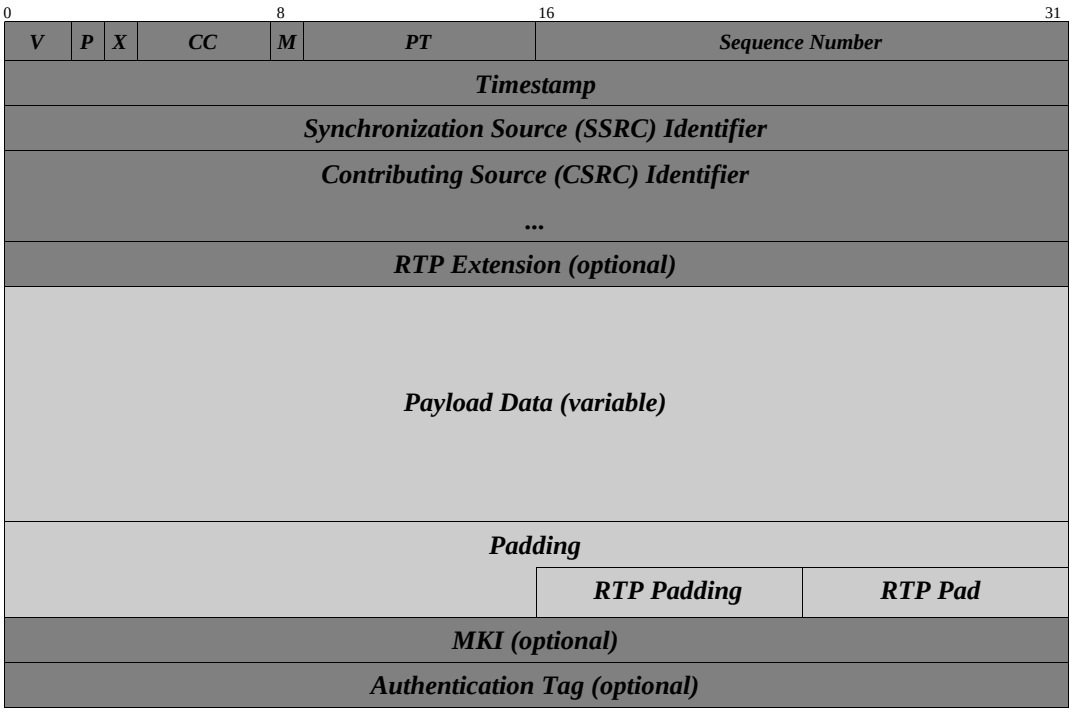

Figure 13.1 SRTP Packet Format

In the figure, the dark grey field at the top of the packet corresponds to the SRTP header (identical to the RTP header). The light gray portion of the packet is the payload, which is the part covered by the encryption operation (this portion is also referred to as the *encrypted portion)*. The message authentication operation computes the MAC over these two portions and places this MAC in the Authentication Tag.

The new optional fields are placed at the end of the packet. First, the MKI field, is used by the key−management protocol. It identifies the master key from which the session keys were derived. It may be used when re−keying for identifying a particular master−key within the cryptographic context. Initially, we will not implement this field, since the master key will be shared by both SRTP and SRTCP, and because I will only consider a single unicast session between two users. Both users share the master key, which has an unspecified lifetime.

Second, the Authentication Tag carries authentication data, if it is to be provided. As explained in section 9.5 in [15], the use of this feature could affect the bandwidth consumption in cellular and wireless environments (given the bandwidth constraints for such environments), so it has been analyzed whether this aspect was strongly necessary for our architecture. Perhaps the use of it, but reducing its size via key−management protocol (by default established as 32 bits) as much as possible without affecting the security would be another alternative. The use of a truncated size of the Authentication Tag must be evaluated with respect to the reduction of security it implies. However, considering the tests shown in section 15, the effect of this additional length of the packet (4 bytes)on the performance of the protocol is negligible.

The Authentication tag is computed in the sender and verified in the receiver, with the algorithm proposed in the cryptographic context (in our case HMAC−SHA1, described in section 5). This feature provides authentication for the RTP header and payload (called the

Authenticated Portion), as well as indirectly providing replay−protection by authenticating the packet sequence number. Note also that integrity protection is mandatory in SRTCP, so this field must appear in the SRTCP packet.

As said before, only the RTP payload will be encrypted (called the Encrypted Portion), along with possible padding, if needed, of this payload. To provide RTP header confidentiality, end−to−end policies should be considered.

13.1.2 SRTCP Packet

SRTCP adds four new fields to the RTCP packet. These are the *SRTCP index*, an *encrypt flag* (referred to as the E−flag), the *authentication tag*, and the *MKI*. Only the latter is optional. As we saw in previous paragraphs, since RTCP is a control protocol, the authentication of its messages must be ensured, and that is why the authentication tag field is mandatory.

As said in the SRTP draft, the Encrypted Portion of the SRTCP packet consists of the encrypted payload of the equivalent compound RTCP packet. The Authenticated Portion consists of the entire equivalent RTCP packet, the E−flag, and the SRTCP index, **after** any encryption has been applied to the payload. Figure 13.2 shows the SRTCP packet format. The SRTCP header is identical to that in RTCP packet. Section 3.2.4 in this document briefly described the different fields in the RTCP header.

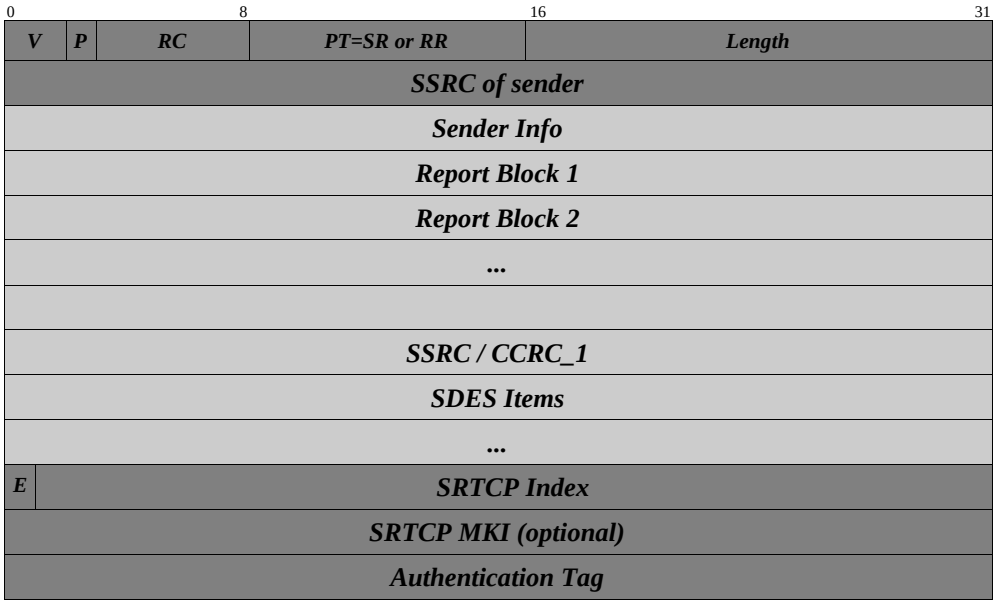

Figure 13.2 SRTCP Packet Format

The dark grey portion at the beginning of the packet corresponds to the SRTCP header. The Encrypted Portion corresponds to the light grey portion of the packet, while the authentication coverage comprises the whole packet except the SRTCP MKI and, of course the Authentication Tag itself, where the MAC is stored.

The SRTCP index is a 31−bit counter explicitly included in the SRTCP packet (note that the SRTP index was implicitly carried in the SRTP packet). Its value has to be zero before the first

packet is sent and increased by 1 modulo 2^{31} after each packet is sent. We should keep in mind that, if there would be re−keying, this index must **not** be reset to zero.

The E−flag indicates whether the current SRTCP packet is encrypted or not.

The Authentication Tag, now mandatory, is identical to the one present in the SRTP packet. Its length is variable (although set by default to be 32 bits), and carries the message authentication data, as in the SRTP packet.

As far as the MKI field is concerned, its functionality is the same as in an SRTP packet. Remember that this is the only optional new field added to the SRTCP packet, thus it will be avoided in the first implementation for the same reasons indicated in section 13.1.1.

13.1.3 Message Authentication and Integrity

Message Authentication and Integrity are ensured by the computation and verification of the Authentication tag, optional for SRTP traffic, but mandatory for SRTCP packets.

For SRTP data, the sender computes the MAC of the Authenticated Portion concatenated with the *ROC* (roll−over counter) parameter, and appends it to the packet. The receiver verifies this tag by performing a new Message Authentication and Integrity computation over the same parameters, and using the same algorithm, and compares this to the one associated with the received packet. If both are equal, the message is valid, and if not, then the receiver must discard this packet, record the event, and an audit "AUTHENTICATION FAILURE" message is returned in the receiver’s side.

This procedure is almost identical for SRTCP traffic, with one only difference: since the ROC parameter is not present for the control protocol, the tag will be computed only over the Authenticated Portion.

13.1.4 Key Derivation

As stated in the SRTP draft, the implementation will use the SRTP key derivation scheme to generate the session keys (from a master key provided by MIKEY), regardless of the authentication or encryption algorithms to be used. It is important to keep in mind that, once the key derivation rate is properly signalled at the start of the session, there is no need for extra communication between the parties that use the SRTP key derivation scheme.

Section 7.1 in [15] states that the key derivation process reduces the burden of key establishment. As many as six different keys are needed to protect the RTP/RTCP session (SRTP and SRTCP encryption keys and salts, SRTP and SRTCP authentication keys), but these are all derived from a single master key in a cryptographically secure way. Thus, the key management protocol needs to exchange only one master key (plus master salt when required), SRTP then derives all the necessary session keys.

SRTP will need, at least, one initial key derivation. Refreshment of these keys during the session is, at this stage, not considered, and the key derivation rate, and thus, the associated master key lifetime, will be fixed. Thus, I will consider an undefined lifetime in the simplest case.

By default, the SRTCP key derivation scheme will share the master key generated for the SRTP derivation. Again, details of this sharing are out of the scope of this paper. Note,

however, that the session keys must never be shared.

Further details about this derivation scheme can be found in [15].

13.1.5 Cryptographic Context

Obviously, each SRTP stream requires the sender and receiver to keep cryptographic state information, called cryptographic context. In every cryptographic context there are several common parameters (such as ROC, replay list, key derivation rate, key lifetime, etc.), independent of the encryption and authentication algorithm used; and some others (such as the block size of ciphers, the session keys, etc.) related to the specific security mechanism being used. Further details of these parameters can be found in [15].

13.1.6 Packet Processing

In this section I present an overview procedure used to create SRTP packets at the sender's side and to extract from them the corresponding RTP packet at receiver's. It is based on the RTP implementation present in the MINISIP user agent, the SRTP draft, and the *libsrtp* C libraries from [33]. At this stage, I have not considered the SRTCP case for reasons regarding the limited time, but I believe it will be similar. I assume the initialization of the cryptographic context is performed by the key management protocol (MIKEY in our case).

- Sender behavior:

  1. Determine the cryptographic context to be used.
  2. Derive the session keys from the master key from the key management protocol (MIKEY).
  3. Encrypt the RTP payload.
  4. If message authentication is required, compute the corresponding authentication tag and append it to the packet.
  5. Send the SRTP packet to the socket.

- Receiver behavior:

  6. Read the SRTP packet from the socket.
  7. Determine the cryptographic context to be used.
  8. Determine the session keys from the master key from the key management protocol (MIKEY).
  9. If message authentication and replay protection are provided, check for possible replay and, next, verify the authentication tag.
  10. Decrypt the Encrypted Portion of the packet.
  11. If present, remove the authentication tag from the packet, passing the RTP packet up the stack.

13.1.7 Predefined Algorithms

A wide set of different algorithms for encrypting and authenticating the RTP messages could be used. Despite this, there are some default mechanisms, also called *mandatory−to−implement* in [15], described here in this section.

13.1.7.1 Encryption

The default cipher for encrypting the RTP payload is the Advanced Encryption Standard (AES), and two different modes to use it are specified: Segment Integer Counter Mode (AES−CM) and f8 mode (AES−f8). I will deal only with the first mode, which is mandatory−to−implement.

We should also consider the NULL−cipher algorithm, since it is also mandatory−to−implement. This will be used when no privacy for RTP or RTCP is required. A more detailed description of these algorithms is given in section 5.2.

13.1.7.2 Message Authentication and Integrity

The predefined algorithm to use and to implement will be HMAC−SHA1, which is based on a keyed−hash function. We must also consider the *NULL Authenticator* in the implementation.

A more detailed description of this algorithm is given in section 5.2.

13.2 SRTP Implementation: MINIsrtp

13.2.1 Introduction

Our implementation of SRTP is called MINIsrtp for two reasons. First, as explained later in this section, it is not intended to be a full−functionality implementation. In addition, it is designed to be integrated into the existing MINISIP user agent. MINIsrtp is free software (see License in section 13.2.4.6) and must still be considered as work−in−progress .

MINIsrtp is based on the following:

- The IETF Secure Real−Time Protocol (SRTP) draft, which defines this protocol as a profile of RTP that provides confidentiality, message authentication, and replay protection to the RTP traffic, as well as support for packet loss and misordered packets without loosing the synchronization between sender and receiver.
- The *libsrtp* library[33], which is a work−in−progress, open−source, C implementation from David McGrew, Cisco Systems, Inc. More information about *libsrtp* can be found at http://srtp.sourceforge.net/srtp.html.
- The HMAC−SHA1 algorithm C implementation written by Aaron Gifford[20].
- The MINISIP implementation[32], developed by Erik Eliasson at TSLab, IMIT, KTH. MINISIP is the implementation of a user agent to be used in Mobile VoIP scenarios.

The idea of the MINIsrtp is to add this security profile to the MINISIP implementation. In the IETF SRTP draft, SRTP is intended to be an additional feature of RTP, rather than a substitute. MINIsrtp agrees with and respects that definition. Thus, SRTP functionality consists of intercepting RTP packets and creates from them on the sender's side the corresponding secure SRTP packets; as well as receiving SRTP packets on the receiver's side and extracting from them the corresponding RTP packet. Hence, this implementation of SRTP

20. This implementation is open code protected by the following Copyright: © by Aaron Gifford, 1998, 2000.

has been added to the RTP functionality into MINISIP, rather to substitute for it.

Note that MINIsrtp (in version 1.0) is not intended to be a *full implementation*, but an initial approach for testing the performance of the SRTP protocol in our testbed. Furthermore, the implementation has been developed to provide simplicity for its use, as well as for its easy integration into MINISIP. The MINIsrtp source code (see Appendix A) might have some bugs, some inefficient code, and even useless declarations. This is due to the limited time to implement it. Future work on this project will also include the improvement of MINIsrtp.

13.2.2 Tools

The MINIsrtp has been developed in C++ under the SuSE Linux 7.1 operating system, with the help of the KDevelop 1.3 IDE. The cryptographic engine, as well as some API functions used for providing replay protection have been taken from the *libsrtp* library (see footnote 16) and A. Gifford's HMAC−SHA1 implementation (see footnote 17).

13.2.3 Features

As said before, this MINIsrtp is designed to be integrated into MINISIP, and to provide its RTP traffic with **confidentiality**, **message authentication**, and **replay protection.** Regarding the decryption operations at the receiver's side, MINIsrtp should be able to handle up to 2^{15} misordered or lost packets without losing the synchronization between the sender and the receiver (the implementation of this feature is based on the *libsrtp* implementation).

SRTCP is not yet implemented in MINIsrtp version 1. This aspect will be included in future work on MINIsrtp.

The cryptographic engine used for the implementation makes use of the mandatory−to−implement algorithms defined in [15]:

- AES−CM (Rijndael) and, alternatively, Null−Cipher for encryption.
- HMAC−SHA1 and, alternatively, Null−authenticator for message authentication.

13.2.4 Description

This section provides a short description of the classes involved in MINIsrtp, as well as the algorithm it performs to protect the RTP packets.

Figure 13.3 depicts the MINIsrtp initial class diagram.

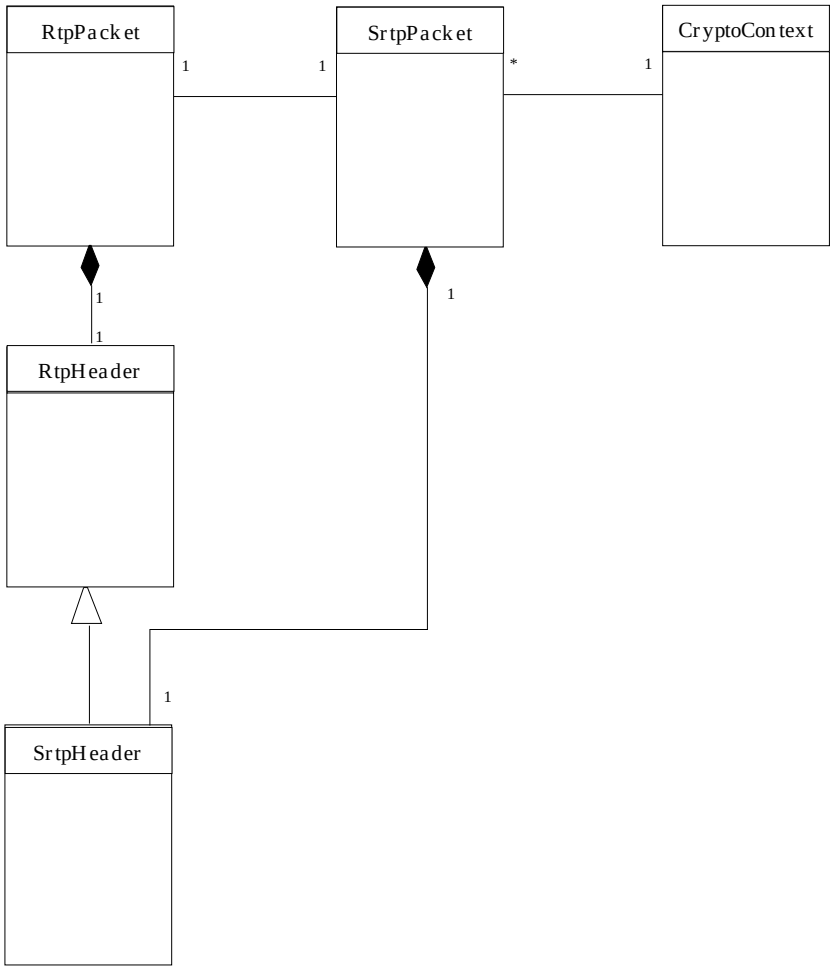

Figure 13.3 MINIsrtp Initial Class Diagram

The source code of MINIsrtp can be found in this document in *Appendix A: MINIsrtp Source Code*.

13.2.4.1 Classes

MINIsrtp adds three more classes into the MINISIP RTP implementation:

- **SRtpPacket:** The SRtpPacket class defines an SRTP packet. This class is similar to the RtpPacket class already included in MINISIP. It defines the SRTP packet attributes, such as the content, the header, the authentication tag, and the length; as well as some related methods for creating an SRTP packet from an RTP packet (protect), creating an SRTP packet from the bytes received in a socket, unprotecting a received SRTP packet to extract the corresponding RTP packet, or sending and receiving an SRTP packet through the net. These methods are described in the section 13.2.4.3.

- **SRtpHeader:** It is identical to the RtpHeader class implemented in MINISIP, so

the SRtpHeader class is inherited from RtpHeader class[21].

- **CryptoContext:** The CryptoContext class defines a cryptographic context as described in section 13.1.5. This class stores the state and attributes of the cryptographic operations associated with each SRTP flow, such as the ROC, the replay lists, the type and parameters of the cipher or authenticator to be used, the master key provided by the key−management protocol (MIKEY), etc.

  MINIsrtp associates one instance of the CryptoContext class with each SRTP flow. Thus, the creation of the a cryptographic context in the sender and receiver would be required and the negotiation of this context should be performed by the key−management protocol (MIKEY)[22]. Section 13.2.4.4 describes in more detail the methods used in the CryptoContext class.

13.2.4.2 Algorithm

The algorithm used by MINIsrtp to protect a RTP packet is based on the SRTP draft. The sender performs the following actions (the cryptographic context is assumed to be initialized by MIKEY):

1. Get the RTP packet to be protected
2. Estimate packet index and add it to the sender's replay list if the estimation was successful, otherwise, discard and log the error.
3. Encrypt the RTP payload (by invoking the *libsrtp* cryptographic engine).
4. If message authentication service is to be provided, compute the MAC over the message concatenated with the ROC (by invoking the *libsrtp* cryptographic engine in the case of the Null Authenticator is to be used, or by invoking the HMAC−SHA1 APIs in the case this algorithm is to be used), append it to the packet, and modify its length.
5. Ready to send the SRTP packet created in steps 3 and 4.

The behavior of the receiver is the following (the cryptographic context is assumed to be initialized by MIKEY):

1. Receive the bytes from the socket and build the SRTP packet according to the cryptographic context used by the sender.
2. Estimate the packet index and check the replay list to avoid replay attacks. If successful continue; otherwise discard and log the event.
3. If message authentication service is to be provided, compute the MAC of the received packet (by invoking the *libsrtp* cryptographic engine in the case of the Null Authenticator is to be used, or by invoking the HMAC−SHA1 APIs in the case this algorithm is to be used) and store it in the cryptographic context. Otherwise, go to 5.
4. Compare the computed MAC (stored in the cryptographic context) with the received in the authentication tag. If the comparison is successful, remove the authentication tag of the packet and continue, otherwise discard the packet and record the event.
5. Decrypt the encrypted portion of the packet (by invoking the *libsrtp* cryptographic engine) to get the RTP original payload.
6. Add the index of the received packet to the receiver's replay list.

21. As SRtpHeader uses RtpHeader methods,and thus the former inherits the latter's functionality, no description of this class will be provided. See *Appendix A: MINIsrtp Source Code* for further information.

22. Note that the cryptographic context negotiation and its initialization (establishment of the parameters and master key), both hardcoded into this implementation, should be provided by the key−management protocol (MIKEY) in future work.

13.2.4.3 SRtpPacket Class Methods

- **`SRtpPacket::SRtpPacket()`:** This constructor creates an empty SRTP packet.
- **`SRtpPacket::SRtpPacket(CryptoContext *scontext, RtpPacket *rtppacket)`:** This constructor is used by the sender to create a SRTP packet from a given RTP packet and a cryptographic context (via `protect`, described below). This method is also in charge of adding an authentication tag to the SRTP packet in the case the message authentication service were provided.
- **`SRtpPacket::SRtpPacket(SRtpHeader hdr, void *content, int content_length)`:** This constructor is used by the recipient (in the case that message authentication were **not** provided) to build a SRTP packet given its header, its content and its content length.
- **`SRtpPacket::SRtpPacket(SRtpHeader hdr, void *content, unsigned char *tag, int content_length)`:** This constructor is used by the recipient (in the case that message authentication were provided) to build a SRTP packet given its header, its content, its authentication tag, and its content length.
- **`SRtpPacket::~SRtpPacket()`:** Class destructor.
- **`RtpPacket *SRtpPacket::get_rtp_packet(CryptoContext *scontext, SRtpPacket *pkt)`:** This method is used by the receiver to convert a received SRTP packet into a RTP packet by unprotecting (via `unprotect`, described below) the former.
- **`void SrtpPacket::send_to(CryptoContext *scontext, UDPSocket &socket, IPAddress &to_addr)`:** This method is used to send a packet to a given IP address through a given UDP socket. The length of the SRTP packet to be sent depends on the security services provided.
- **`Static SRtpPacket *SRtpPacket::receive_from(UDPSocket &srtp_socket, CryptoContext *scontext, int timeout=-1)`:** This method is used to receive a stream of bytes from a given socket, and build a SRTP packet from those bytes, given the cryptographic context associated with the flow.
- **`char *SrtpPacket::get_bytes(CryptoContext *sctx)`:** This method returns a pointer to an array of bytes containing this SRTP packet. The length of the array depends on which services the cryptographic context is providing.
- **`SRtpHeader &SrtpPacket::get_header()`:** This method returns the SRTP header of this SRTP packet.
- **`void *SRtpPacket::get_content()`:** This method returns a pointer to the payload of this SRTP packet.
- **`unsigned char *SRtpPacket::get_tag()`:** This method returns a pointer to the authentication tag of this SRTP packet. It is used in the case that message authentication were provided.
- **`void SRtpPacket::remove_tag()`:** This method removes the authentication tag of a SRTP packet provided with message authentication service

- **`int SRtpPacket::get_content_length():`** This method returns the length of the payload of this SRTP packet.

- **`int SRtpPacket::size(CryptoContext *sctx):`** This method returns the total size of a SRTP packet. Note that the size of the packet depends on which services the cryptographic context is providing.

- **`int protect (CryptoContext *scontext, RtpPacket *srtppacket, int *len, int content_len):`** This auxiliar function is used by the sender to provide the RTP packet with confidentiality (by encrypting the RTP payload) and message authentication (by computing a MAC over the RTP packet concatenated with the ROC). Moreover, this function deals with the estimation of the index used by the cryptographic engine, and also with the replay protection (by invoking *libsrtp* functionality for this purpose).

- **`int unprotect (CryptoContext *scontext, SRtpPacket *srtppacket, int *len, int content_len):`** This auxiliar function is used by the receiver to extract the original RTP packet from the received SRTP packet (by decrypting the SRTP payload, verifying the MAC received in the SRTP packet, and removing the authentication tag if this were present). Moreover, this function deals with the estimation of the index used by the cryptographic engine, and also with the replay protection (by invoking *libsrtp* functionality for this purpose).

13.2.4.4 CryptoContext Class Methods

- **`CryptoContext::CryptoContext(string key, string salt, string service, string cipher_type, string auth_type):`** This constructor defines a cryptographic context associated to one messages flow. By invoking it, all the cryptographic attributes, such as the master key, the type of cipher, the key length, etc., are initialized. Note that is this MINIsrtp first version, this attributes are obtained form MINISIP configuration file, *minisip.conf*. This constructor also invokes the function `init_aes_128_prf()` after allocating the cipher and the authenticator (via *libsrtp* cryptographic engine) to initialize the cryptographic context and derive the SRTP keys.

- **`CryptoContext::~CryptoContext():`** The class destructor.

- **`void CryptoContext::set_key_deriv_rate(int r):`** This method may be used to set the key derivation rate in the case *re−keying* were applied.

- **`unsigned int CryptoContext::get_roc():`** This method may be used to return the rollover counter associated to this cryptographic context.

- **`void CryptoContext::update_roc():`** This method may be used to update the rollover counter associated to this cryptographic context.

- **`Void CryptoContext::reset_roc():`** This method may be used to reset the rollover counter associated to this cryptographic context.

- **`unsigned int CryptoContext::get_tag_len():`** This method returns the length of the authentication tag associated to this cryptographic context.

- **`int CryptoContext::get_serv():`** This method returns an integer which identifies

the type of service provided by this cryptographic context.

- **`short CryptoContext::get_s_l():`** This method may be used to get the `s_l` attribute associated to this cryptographic context (see [15] for further information about this attribute).

- **`cipher_t *CryptoContext::get_cipher_t():`** This method returns the cipher used by this cryptographic context. Remember that in this version it is hardcoded to use AES−CM or the Null Cipher.

- **`auth_t *CryptoContext::get_auth_t():`** This method returns the authenticator used by this cryptographic context. Remember that in this version it is hardcoded to use the HMAC−SHA1 algorithm or the Null Authenticator.

- **`err_status_t CryptoContext::init_aes_128_prf():`** This method (based on the *libsrtp* implementation) is used in the constructor to initialize the cipher, the authenticator, and to derive the encryption key and the authentication key to be used by the *libsrtp* cryptographic engine.

13.2.4.5 Bug Information

MINIsrtp uses two different interfaces to invoke either the HMAC−SHA1 algorithm or another algorithm (in our case Null Authenticator) when message authentication is to be provided. This feature seems to cause some incompatibility when allocating the message authentication algorithm context. In fact, when providing **only** confidentiality service, the message authentication algorithm selected **must be an algorithm other than HMAC−SHA1**.

The first version of MINIsrtp "patches" this problem in the SRTP cryptographic context by "hardcoding" the use of the NULL Authenticator when only confidentiality is desired. Obviously, this does not effect the process at all, but the code might be confusing for the reader.

Moreover, several tests on the implementation have shown that the **first** packet received returns a message authentication failure when using HMAC−SHA1. Thus far, this problem has not yet been solved. However, considering that we are going to send fifty packets each second, this bug has been considered negligible.

13.2.4.6 License

14 Multimedia Internet KEYing (MIKEY)

MIKEY is still an IETF draft[29]. This section provides a brief description of the protocol, as well as a framework to be used within our model. Future work in this area would provide a suitable implementation of the protocol in order to integrate it with the MINISIP and MINIsrtp implementations.

14.1 Overview

Although there is work done in IETF to develop key−management schemes, there is still a need for a scheme with low latency, suitable for demanding cases such as for the real−time data. Moreover, when dealing with wireless networks, this demand is further impacted by the bandwidth constraints present in such environments. MIKEY suits this demand well, since it is designed to have the following features[29]:

- End−to end security
- Simplicity
- Efficiency: low bandwidth consumption, low computational workload, small code size, and a minimal number of roundtrips
- Tunneling
- Independent of any specific security functionality of the underlying transport

In our case we consider a simple peer−to−peer (unicast) scenario, where a SIP−based call between two parties is to be provided with the necessary security mechanisms, and where these mechanisms and their parameters (cryptographic context) must be set up by mutual agreement.

The following concepts are interesting for a better clarity[29]:

- Crypto Session: data stream protected by a single instance of a security protocol (i.e. SRTP stream).
- TEK Generation Key (TGK): a bit−string agreed upon by two parties associated with a CryptoContext. From the TGK, Traffic Encrypting Keys (TEK) can be generated without need of further communication.

The procedure followed by MIKEY is the following:

1. Agreement of the security parameters and TGK(s) for a Crypto Sessions group.
2. TEK derivation from the TGK for a Crypto Session.
3. The TEK and the security protocol parameters are the input to the Security Protocol (i.e. SRTP).

The TEK may be used either directly by the Security Protocol, or to derive further session keys from it. The latter is the option followed by SRTP. Figure 14.1(based on figure from[29])depicts this procedure.

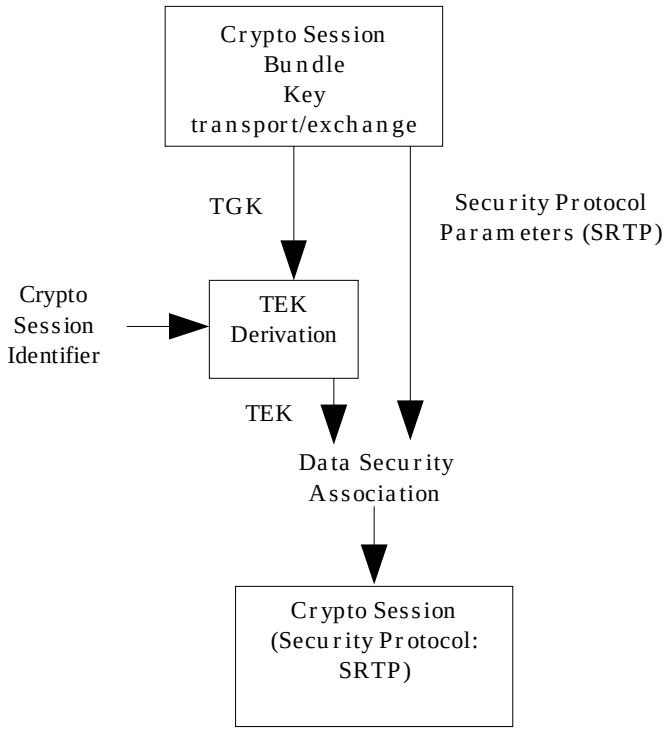

Figure 14.1 MIKEY: Overview of the Key Management Procedure[29]

Regarding the basic key transport and exchange methods, MIKEY offers three alternatives (for further information, refer to MIKEY draft[29]:

- Pre−shared Key
- Public Key Cryptography
- Signed Diffie−Hellman

The predefined (mandatory to implement) transforms in MIKEY are the same used for SRTP:

- Key data transport encryption: AES−CM
- Hash functions: SHA−1
- Pseudo random number generator: SHA−1
- MAC and verification Message function: HMAC−SHA1

Figure 14.2 shows the structure of a MIKEY message:

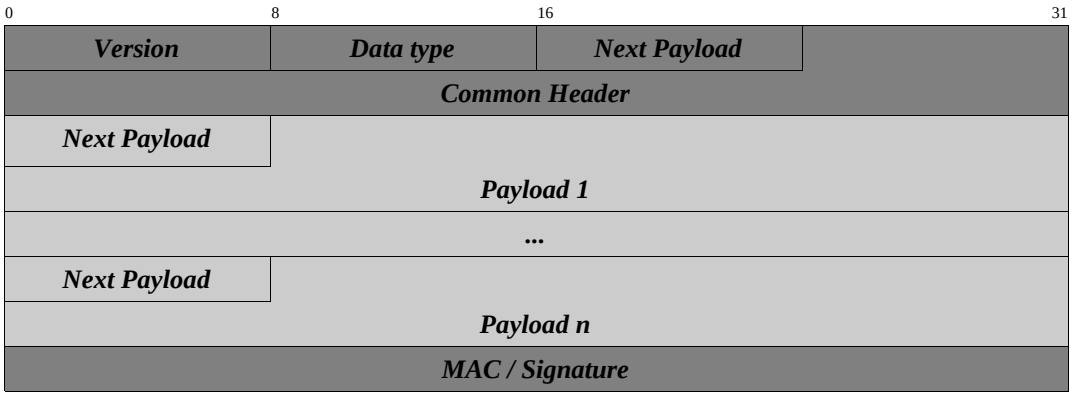

Figure 14.2 Structure of a MIKEY message

How a message is created and parsed by MIKEY is summarized below:

1. Creation of initial message starting with the Common header payload
2. Concatenation of the necessary payloads
3. Creation and concatenation of the MAC / signature payload. The MAC is calculated over the entire message except the MAC signature field

For parsing the message:

1. Extraction and verification of the Timestamp to check possible replay
2. Extraction ID and authentication algorithm
3. Verification of the MAC / signature
4. Processing of the message
5. Sending of a verification response message to the initiator

The following list summarizes the different types of MIKEY payloads:

- Key Data Transport Payload (KEMAC)
- Envelope Data Payload (PKE)
- Diffie−Hellman Data Payload (DH)
- Signature Payload (SIGN)
- Timestamp Payload (T)
- ID Payload (ID)
- Certificate Payload (CERT)
- Cert Hash Payload (CHASH)
- Message Verification Payload (V)
- Security Policy Payload (SP)
- RAND Payload (RAND)
- Error Payload (ERR)
- Key Data Sub−Payload
- General Extension Payload

14.2 MIKEY Framework for Secure Mobile VoIP

14.2.1 Terminology Relationship

We must establish a relationship between terminology in MIKEY and SRTP, since the former needs to be more general[29]. Thus, the SRTP stream is called *Crypto Session* in MIKEY, the input to the SRTP's cryptographic context is called Data Security Association (Data SA) in MIKEY, and the SRTP master key from which the session keys will be derived is referred to in MIKEY as the Traffic Encrypting Key (TEK).

Furthermore, another relationship may be established between the *offerer/answerer* model used in SIP and SDP and the *initiator/ responder* model used in MIKEY. However, this section will use both terms to refer to the end users.

14.2.2 MIKEY within SIP

MIKEY may work "within" SIP (i.e. MIKEY messages may be carried in SDP bodies inside the SIP messages). As a matter of fact, MIKEY is suitable in the SIP Trapezoid model (figures 9.1 and 11.1). The SIP offerer (User A) will be the MIKEY Initiator, and the SIP answerer (User B) will be the MIKEY Responder (see Figure 14.4). Thus, this implies that the MIKEY Initiator's message is included in the SIP INVITE message (in the SDP body), while the answerer's response to this INVITE will contain the MIKEY Responder's message. Section 14.2.3 describes the integration of MIKEY into SDP. Figure 14.3 illustrates the situation:

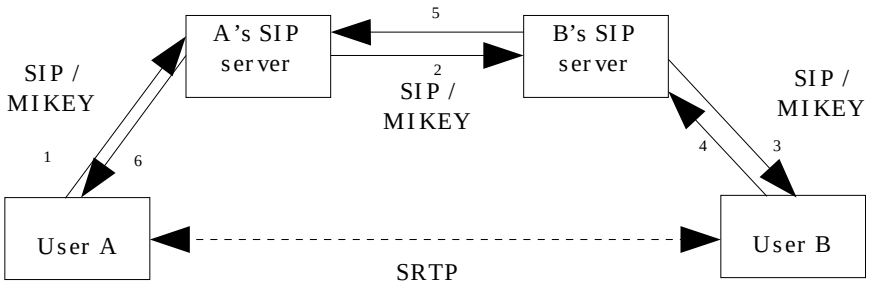

Figure 14.3 SIP−Based call example using MIKEY over SIP

If the MIKEY part of the offer is not accepted by the answerer, a MIKEY error message is included in the answer (see section 14.2.4 for MIKEY Error Handling information).

As described in the MIKEY draft, *"it may be assumed that the offerer knows the identity of the answerer. However, unless the Initiator's identity can be derived from SIP, the Initiator must provide its identity to the Responder. It is* ***recommended*** *to use the same identity for both SIP and MIKEY"*[29].

14.2.3 MIKEY Integration into SDP

SIP makes use of SDP descriptions to carry information about the session. Therefore, it is also convenient to integrate the key management procedure into the session description it is going to protect. This provides low latency, since the number of roundtrips for setup may be reduced.

The SDP key management attribute (*key−mgmt*)takes two arguments: the keying scheme to be used (*mikey*) and the attributes of this scheme (MIKEY packet encoded in base64). Further information can be found in section 7.1 in [29].

The following example obtained from [30] illustrates this situation:

```
v=0
o=alice 2891092738 2891092738 IN IP4 lost.somewhere.com
s=Cool stuff
e=alice@w−land.org
t=0 0
c=IN IP4 lost.somewhere.com
a=key−mgmt:mikey uiSDF9sdhs727ghsd/dhsoKkdOokdo7eWsnDSJD...
m=audio 49000 RTP/SAVP 98
a=rtpmap:98 AMR/8000
m=video 52230 RTP/SAVP 31
a=rtpmap:31 H261/90000
```

14.2.4 Error Handling

Any error regarding the key−management protocol should be reported to the peers by a MIKEY error message. The error messages are formed by a MIKEY header, a timestamp, the error payload(s) (ERR), and the signed MAC computed over the entire message.

If the answerer (Responder) does not support the set of parameters offered by the offerer (Initiator), the returned error message will include the supported parameters.

Further security considerations related to the implementation and the local policy are described in [29].

14.2.5 MIKEY Over an Unreliable Transport Protocol

As described in section 15.1.3 in this document, the first implementation of this model will be built on UDP, rather than TCP. If we are going to use MIKEY over an unreliable transport protocol, such as UDP, a basic process must be performed to ensure MIKEY reliability[29]:

- The entities must set a timer and initialize a retry counter
- If the timer expires, the message will be resent and the retry counter will be decreased
- If the retry counter reaches zero, the event will be logged

14.2.6 MIKEY Payloads

All the different MIKEY payloads are described in detail in the MIKEY draft . However, a first approach to a suitable C++ implementation of MIKEY massages and payloads was made by Erik Eliasson at TSLab, IMIT, KTH. This source code is included in this document in

Appendix E: A First Approach to MIKEY Messages Implementation[23].

14.2.7 MIKEY Interface

If the MIKEY implementation is separate from the SIP and SRTP implementation, a suitable API between those protocols must be defined by the programmer. Some hints and advices regarding this interface for the implementers of MIKEY are given in section 7.4 in the MIKEY draft[29].

14.2.8 MIKEY Exchange Method: Signed Diffie−Hellman

Regarding the basic key transport and exchange methods, of special interest for us is the Signed Diffie−Hellman method. If supported by the use of certificates and a PKI, we can ensure peer−to−peer mutual authentication and it is a good option in the special peer−to−peer case[2]. Furthermore, although the resource consumption is higher than in the other alternatives, this method provides *Perfect Forward Secrecy* (PFS) and it does not require the possession of the responder's certificate by the initiator before the setup. It would be sufficient that the responder includes its signing certificate in the response. Figure 14.4 illustrates this exchange.

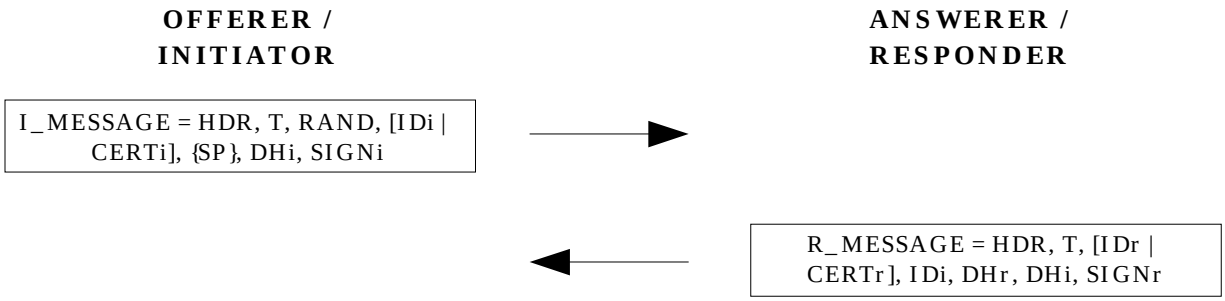

Figure 14.4 Signed Diffie−Hellman exchange in MIKEY

23. This first approach to the MIKEY messages implementation (© by E. Eliasson, 2003) is intended to be a start reference implementation, and it **may** be used for future work on the final implementation.

15 Description of the Implementation of the Model and its Analysis

This section provides a detailed description of the practical work related to the Secure Mobile VoIP model described in section 11. Some tests and measurements regarding security and performance for this model are described and analyzed here. Since this project has been concerned with the media stream security, rather than the establishment and termination security, the tests performed are mostly related to this aspect. Section 16 presents several considerations regarding future work to be performed within the SIP security.

15.1 Implementation

As said in the previous sections, the implementation of the model was performed in a "bottom−up" way. With "bottom−up" we mean that the process was performed as follows:

- A simple RTP conversation between two end users was tested.
- Our SRTP implementation (MINIsrtp) was developed.
- MINIsrtp was integrated and tested into MINISIP user agent.
- The setting up of the SIP servers using the SER software[31] was performed at TSLab by Jon−Olov Vatn.
- MINISIP user agent was tested together with the SER SIP servers.
- SRTP performance was tested.

15.1.1 MINIsrtp Development

The first version of MINIsrtp was finished by the beginning of June. In this first version, MINIsrtp provided confidentiality, message authentication, replay protection, and handled misordered and lost packets without breaking the synchronization between sender and receiver down. SRTCP support was not yet implemented.

MINIsrtp was designed and developed, following the SRTP draft guidelines, to be a RTP profile, rather than a substitute for it; and to be integrated into MINISIP as easily as possible. Therefore, the intended behaviour builds up MINISIP RTP implementation and transformations the RTP packets into SRTP (by using security mechanisms) packets and viceversa in the receiver.

Before its integration into MINISIP, MINIsrtp was tested by a simple application which used text strings instead of audio code. The test worked without problems.

15.1.2 Integration of MINIsrtp into MINISIP User Agent

MINIsrtp files were added to MINISIP *Code Versioning System* (CVS) by the beginning of June. The first test consisted on a simple call between two MINISIP user agents running on the same machine. This call was performed in such a way that the initialization of the cryptographic context was hardcoded. This means that, assuming there was no previous key management, the security services MINIsrtp provided, the type of cryptographic algorithms to be used, and the master keys from which SRTP session keys are derived in advance before the

SRTP session started[24].

A second simple test of the implementation assumed the cryptographic parameters were located in the MINISIP configuration file (*minisip.conf*). This file includes all the MINISIP parameters and part of the cryptographic configuration. Thus far, the parameters set into this file are:

- Security services provided by SRTP: Into MINIsrtp *NONE* (just replay protection), *CONF* (replay protection and confidentiality), *AUTH* (replay protection and message authentication), or *BOTH* (replay protection, confidentiality, and message authentication).

- The type of algorithm to be used: Into MINIsrtp *AESCM* (AES in Counter Mode) or *NULLCIPHER* (the null cipher algorithm) for providing confidentiality; *HMACSHA1* (HMAC−SHA1 algorithm) or *NULLAUTH* (the null authenticator algorithm) for providing message authentication.

- The master key (16−byte length) from which session keys will be derived (for message authentication and encryption). This is supposed to be provided by MIKEY in future work (see section 14).

- The master salting key (14−byte length). This is supposed to be provided by MIKEY in future work (see [15], section 7.2 for a rationale).

MINISIP source code uses a *flag* to distinguish with type of media transport is to be used (either RTP or SRTP).

15.1.3 Setting up of the SIP Servers

At the same time MINIsrtp was being integrated into MINISIP and tested, Jon−Olov Vatn set the SIP servers[25] up at TSLab. Unfortunately, SER does not provide support for TCP nor TLS, and it only supports the exchange of UDP datagrams. This makes us unable to test the selected TLS scheme to protect the SIP protocol. Thus, future work on this project must add this TLS support to SER. The peer authentication of the SIP REGISTER is provided in our case through the HTTP Digest mechanism. The HTTP Digest mechanism to be used together with SIP is described in section 22 in [3]. The main reason for using SER is that, unlike VOCAL, it provides support for the DNS look−ups in the SIP Redirect Server, thus enabling call setups between IP−hosts in different domains.

15.2 Analysis and Validation of the Model

The validation of the model has been performed using MINIsrtp **to implement media stream security** (i.e. Extending MINISIP to use SRTP). We must consider several points when our evaluating SRTP implementation:

- We must explicitly state what our solution does with respect to SRTP
- It should be functionally correct: i.e. it emits the messages it should and nothing more
- Performance: measure the transition times for each significant interaction

24.This was made by hardcoding those values into the cryptographic context constructor, invoked just before the packets exchange started.
25. By using the SIP Express Router software (SER)[31]. Refer to www.iptel.com/ser for further information.

- Demonstrate correctness, i.e., proper behavior despite improper packets

First of all, our solution for securing the media stream is intended to provide the following additional security services beyond the existing RTP functionality of MINISIP without significantly effecting the performance (as perceived by the user):

- Confidentiality: by using encryption/decryption operations through AES algorithm.
- Message Authentication: by using a hash−based Message Authentication Code (HMAC), such as HMAC−SHA1.
- Protection against replay attacks.
- Support for packet loss and misordered packets without loosing the synchronization between sender and receiver.

15.2.1 Correctness of MINIsrtp

Given these requirements, we can start by describing the correctness of MINIsrtp:

- MINIsrtp into MINISIP sends and receives SRTP packets created from RTP packets. The SRTP packets are formatted according to the definition given in[15] and they are sent and received using the usual MINISIP mechanisms which were used for the RTP packets.

- MINIsrtp sets up a cryptographic context in the sender and receiver, as described in [15], before exchanging any packet. The initialization of this context, although supposed to be performed by the key−management mechanism , is hardcoded into our implementation. As noted earlier, the addition of key management is left as future work.

- Confidentiality service is provided by encrypting and decrypting the payload of the packets in the sender and receiver respectively, as defined in [15]. The algorithms used for this purpose are those established as mandatory−to−implement in [15]: AES in Counter Mode and the Null Cipher.

- Message Authentication service is provided by using message authentication codes, as described in [15], with one exception: [15] defines the message (*M*) over which to apply the MAC computation as the authenticated portion concatenated with the ROC (see section 13.1.3), while MINIsrtp when using the Null Authenticator, computes this MAC only over the authenticated portion. In the case HMAC−SHA1 is to be used, MINIsrtp properly computes this MAC as described in [15]. As to *libsrtp*, it seems that the MAC calculation is done only over the authenticated portion as well. In order to verify this, I posted a question on the *libsrtp Mailing List*, but I have not received an answer yet.

- The algorithms used to provide message authentication are those established as mandatory−to−implement in [15]: HMAC−SHA1 and Null Authenticator. The use of the latter is not recommended (see [15]).

- Protection against replay attacks is also provided by storing the indices of the received packets in a list, and checking this list every time a packet is received. Several tests were performed to verify the correctness of this feature (although not in detail).

- MINIsrtp should support up to 2^{15} misordered or lost packets without loosing the synchronization between sender and receiver. This feature has not been tested in detail. However, the loss of one of every ten packets has been tested and it is supported, and thus

there is no reason to think that this feature is not provided in other cases.

The following list summarizes the behaviour of MINIsrtp when faced with erroneous packets during processing:

- If a replay is detected by the receiver, the event is logged by sending a message to *syslog*, the replayed packet is dropped, and the execution continues.

- If an error occurs when computing the MAC in the sender using the Null Authenticator, the event is logged by a message sent to *syslog*, the authentication tag is established as empty, and the execution continues[26]. Note that although considered by MINIsrtp, this situation will never occur, since the Null Authenticator algorithm always returns a value indicating that the computation was properly performed (*libsrtp* feature).

- If an error occurs when computing the MAC in the receiver using the Null Authenticator, the event is logged by a message sent to *syslog*, the packet is dropped for security reasons, and the execution continues.

- The same error as above when using HMAC−SHA1 is not yet considered and it depends on the HMAC−SHA1 implementation.

- Different errors due to encryption and decryption of the data (i.e. internal cryptographic engine failures) are handled by the *libsrtp* cryptographic engine.

- If the MAC computed over a received packet does not match with that received in the authentication tag (possible violation of integrity), the event is logged by a message sent to *syslog*, the packet is dropped, and the execution continues.

- If the cipher or the authenticator's state can not be allocated before starting the session, the event is logged by sending a message to *syslog*, but the execution continues. Future work on MINIsrtp will include a mechanism to abort the execution if necessary.

- If the cipher or the authenticator context can not be initialized, the event is logged by sending a message to *syslog*, but the execution continues. Future work on MINIsrtp will include a mechanism to abort the execution if necessary.

15.2.2 Performance Measurements on MINIsrtp

Regarding the performance of MINIsrtp, some measurements have been made. These measurements are based on the processing time of an SRTP packet and are compared to the processing time of an ordinary RTP packet. Note that the performance depends on the specifics of the implementation itself, in the sense that the code effects the efficiency of the process. These measurements were taken when executing a simple SRTP and RTP packet exchange between two entities in the same machine. The intention of this test is not to analyze in detail the measured times themselves, but rather to show the relationship between RTP processing time and SRTP processing time. The method used to perform these measurements utilized the function *gettimeofday()*.

The features of the machine in which the measurements were performed are the following:

26. Note that the tag is empty in either case when using the Null Authenticator, and this could lead to misinterpretation in the receiver's side.

- Laptop ASUS 1300B with Pentium III © processor, 700 MHz.
- 112 MB RAM (having enough free memory, so that there is no swapping)
- Operating System: SuSE Linux 7.1 Personal Edition

The following list summarizes the features of MINIsrtp in the tests:

- Security Services: confidentiality and message authentication [27]
- Cryptographic Algorithms: AES in Counter Mode for the confidentiality and HMAC−SHA1 for the message authentication
- Length of the master key: 16 bytes
- Length of the salting key: 14 bytes
- Length of the encryption key: 16 bytes
- Length of the authentication key: 16 bytes
- Length of the block: 128 bytes

The code of the example application we are running for the tests on is the following:

```
* srtptest.cxx
*
* Purpose:
*      Demonstrates how the SRTP implementation can be used to transmit
*      information. This application implements both a sending and
*      receiving part and which one will be started depends on the
*      arguments to the application. Developed for MINIsrtp tests.
*
* @author Israel Abad & Erik Eliasson   israel@kth.se  eliasson@it.kth.se
/

*
* Here you specify to which host and port the SRTP traffic will be sent.
*/
define TO_HOST "localhost"
define TO_PORT 20000

include"RtpPacket.h"
include"SRtpPacket.h"
include"CryptoContext.h"
include<unistd.h>
include"../ipv6util/IP4Address.h"
include<string.h>
include"../util/ConfigFile.h"
include <sys/time.h>

* SENDER
*
* Purpose: Sends one string of 160 bytes (payload) in an SRTP packet: 50
*          packets
* to localhost:20000
*
* Alg.
*  1. Create socket that will be used for srtp traffic.
*  2. Specify receiver
*  3. Do 50 times
      3.1 Define content of packet to transmit
       3.2 Create RTP packet.
       3.2 Create SRTP packet.
*      3.3 Send SRTP packet.
*/
void sender(ConfigFile config){
 struct timeval *Tps, *Tpf;
 struct timezone *Tzp;

 Tps = (struct timeval*) malloc(sizeof(struct timeval));

 Tpf = (struct timeval*) malloc(sizeof(struct timeval));
```

27. The Replay Protection service is always provided by MINIsrtp.

```
 Tzp = 0;
 cout << "-[SRTP sender started]-"<< endl;


 string messages[1] =
{"0000000000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000000000000
00000\0"};
      cout << "-[Creating socket]-"<< endl;
      UDPSocket *udpsock = new UDPSocket(false, 10000); //false means do not
                                                //use IPv6, 10000 is the port
      IP4Address ipaddr(TO_HOST);
      ipaddr.set_port(TO_PORT);
      cout << "-[Opening Crypto Context]-"<< endl;

       CryptoContext *scontext_s = new
CryptoContext(config.get_string("master_key"),
              config.get_string("salt_key"),
              config.get_string ("sec_services"),
              config.get_string ("cipher_type"),
              config.get_string ("auth_type"));

       cout << "-[Crypto Context created!]-"<< endl;
      for (int i=0; i<50; i++){
         char sbuf[2048];
         for (unsigned int e=0;e<messages[0].length();e++)
            sbuf[e]=messages[0][e], sbuf[e+1]=0;
         string rtpcontent = string(sbuf);

         gettimeofday (Tps, Tzp);

         RtpPacket *rtppacket = new RtpPacket((void *)rtpcontent.c_str(),
                                  rtpcontent.length()+1, i, i*1000);

         SRtpPacket *srtppkt = new SRtpPacket(scontext_s, rtppacket);

         gettimeofday (Tpf, Tzp);
         printf("Total Time creating SRTP pkt(usec): %ld\n",
                          (Tpf->tv_sec-Tps->tv_sec)*1000000
                          + Tpf->tv_usec-Tps->tv_usec);

         srtppkt->send_to(scontext_s, *udpsock, ipaddr);

         delete srtppkt;
         delete rtppacket;
}



* RECEIVER
*
* Purpose: Receives SRTP packets, gets RTP packets and prints the content of
* them to standard output.
* Alg.
* 1. Create socket
* 2. Forever do
*   2.1 Read SRTP packet from socet
*   2.2 Extract RTP packet
*   2.2 Print content of RTP packet.
*/
void receiver(ConfigFile config){
 int i=1;
 struct timeval *Tps, *Tpf;
 struct timezone *Tzp;

 Tps = (struct timeval*) malloc(sizeof(struct timeval));
 Tpf = (struct timeval*) malloc(sizeof(struct timeval));
 Tzp = 0;
      cout << "-[SRTP receiver started]-"<< endl;

 UDPSocket udpsock(false, 20000);
 cout << "-[Opening Crypto Context]-"<< endl;
 CryptoContext *scontext_r = new CryptoContext
                   (config.get_string("master_key"),
                    config.get_string("salt_key"),
                    config.get_string ("sec_services"),
                    config.get_string ("cipher_type"),
                    config.get_string ("auth_type"));
```

```
 cout << "-[Crypto Context created!]-"<< endl;
 while (1){
     gettimeofday (Tps, Tzp);
     SRtpPacket *packet = packet->receive_from(udpsock, scontext_r);
     RtpPacket *rtppkt = packet->get_rtp_packet(scontext_r, packet);
     gettimeofday (Tpf, Tzp);
     printf("Total Time creating RTP pkt(usec): %ld\n",
                          (Tpf->tv_sec-Tps->tv_sec)*1000000
                           + Tpf->tv_usec-Tps->tv_usec);
     cout << "Received packet: " << (char*)rtppkt->get_content()<<endl;
     delete rtppkt;
     delete packet;
}


void usage(){
       cerr << "Usage srtptest {s|r}"<< endl;
       exit(1);


/*
* Alg.
*  o if argument is 's' then "sender()"
*  o if argument is 'r' then "receiver()"
*  o (else) usagemessage
*/
int main(int argc, char **argv){
   string configfile = "../minisip/minisip.conf";
   if (argc!=2 || strlen(argv[1])!=1 )
       usage();
   ConfigFile config(configfile);
   switch (argv[1][0]){
              case 's':
              case 'S':
                     sender(config);
                     break;
              case 'r':
              case 'R':
                     receiver(config);
                     break;
              default:
                     usage();
       }
       return 0;
}
```

MINISIP sends 50 packets of 160−byte RTP payload length per second (i.e. 64 Kbps). This means that the RTP packet has a size of 172 bytes, and the SRTP packet has a size of 176 bytes (172 + 4 for the authentication tag if message authentication is to be provided). Assuming this, we have 20 milliseconds between packets, and the SRTP performance has been compared to this aspect.

After several measurements, the results I have obtained are shown in table 15.1:

Action / Transport	***RTP***	***RTP + SRTP***
Packet Creation	3−5 µs.	76−80 µs.

Table 15.1 Time taken by RTP and SRTP to Create a Packet in MINIsrtp

The time needed to transmit/receive a packet is about 55−60 µs. Considering that our application continuously sends a bidirectional SRTP flow, this results show that the processing requires about 1% of the time between packets, i.e., it adds an additional delay corresponding to about 1% of the inter−packet delay:

$$\frac{packet\_creation + packet\_transmission/reception}{time\_between\_packets} \; X\ 100$$

We have measured some SRTP packets with a peak delay of 240 microseconds. According to our tests, this additional delay (about 2.5% of the time between packets) happens less than 1% of the time, so we consider the performance impact to be negligible. In addition, these delay variations are far smaller than the expected delay variance due to the packets flowing over the network and will in any case be imperceptible to the human listener.

The time for the transmission of the SRTP packets on each link (assuming *store−and−forward* entities)will be slightly longer than that for RTP packets, since when message authentication tag is present, 4 bytes are added. However, this additional time is negligible.

MINIsrtp's cryptographic throughput when protecting and unprotecting packets using AES and HMAC−SHA1 is about 20 Mbps. packets using AES and HMAC SHA1 is about 20 Mbps. In comparison to our software implementation, currently available high performance hardware devices performing these same operations have a throughput of about 600 Mbps[55].

16 Conclusions and Future Work

This section summarizes the main conclusions regarding the proposed Secure Mobile VoIP model. Some future work to be performed on the model is also detailed here. Note that, regarding the implementation, this document only examined to security for the media stream, rather than the SIP security. However, the document is intended to be a base for work in this area, and the proposed solution, even though not completely tested, has been thoroughly investigated.

16.1 Conclusions

We have faced the security in Mobile VoIP as a chain where no link should fail to maintain the model secure. Through this document, I have presented the problem to be solved, as well as the different alternatives to achieve this goal. Some of these alternatives are already in use (IPSec and VPNs to protect the media stream) and some others are still being investigated (SRTP and MIKEY). Thus, this document presents a solution to secure every link of this chain, from the establishment to the termination, giving the model the necessary security services and minimizing the effect of this security services and their mechanisms on the performance.

Regarding the practical work and tests on this proposed solution, this document is focused on the security of the media stream. However, a thorough investigation work has also been performed concerning the SIP security and the key management. Given this, we can state some conclusions on this work:

- Regarding SIP and the Basic SIP Trapezoid, we believe that the most suitable scheme to protect the message exchange with the needed security services (confidentiality, message authentication, and replay protection), as well as to provide a peer authentication scheme for the entities involved, and a dynamic session key distribution is the establishment of a PKI to support SSL/TLS.

- MIKEY seems to be the appropriate candidate to handle the key management, since it suits well the demands we have for real time applications.

- The features provided by SRTP (such as confidentiality, message authentication, replay protection, support for packet loss, high−throughput, fast crypto operations, and low packet expansion) make of it a suitable solution to protect the media stream. Besides, the low bandwidth consumption of SRTP (unlike IPSec) makes us expect a good performance of the model.

16.2 Future Work in this Area

To establish the suitable PKI to support a TLS scheme to secure the SIP protocol is still to be provided. This document only theoretically describes this PKI and TLS scheme, but no practical work regarding this aspect of the proposed solution has yet been performed.

This project provides a brief description of a possible framework which is intended to be the base to start working with MIKEY. Future work will include a suitable implementation of MIKEY to work together with SRTP and SIP. The start reference implementation written by E. Eliasson at TSLab, IMIT, KTH, and shown in Appendix B may be used as a base to start working on a final MIKEY

implementation.

Regarding MINIsrtp, the time to design and develop it was very limited. Thus, future work will include its **overall improvement** concerning inefficient code and possible bugs, as well as the addition of new and/or enhanced features (either to work together with MINISIP or other user agent), such as SRTCP or the use of other cryptographic algorithms (AES in f8 Mode). Its integration with a suitable MIKEY implementation must also be performed. Another improvements, such as handling the bugs reported in section 13.2.4.5 must also be considered. The concatenation of the ROC with the authenticated portion of the packet, as described in 13.1.3 to compute the MAC, is only considered in the case of HMAC−SHA1 algorithm is to be used, and is still to be done in the case of the Null Authenticator is used (although this is not explicitly necessary since the use of the Null Authenticator implies no computation at all).

This project does not **explicitly** deals with mobility aspects, such as roaming or hand−overs, but it may be useful as one of the reference documents for future work with such aspects.

Appendix A. MINIsrtp Source Code

```
/*****************************************************************************
                          CryptoContext.h  -  description
                             -------------------
    begin                : Fri Apr 4 2003
    copyright            : (C) 2003 by Israel Abad
                           TSLab, IMIT, KTH, Stockhlom, Sweden
    email                : israel@kth.se
                         : i_abad@terra.es
 *****************************************************************************/
// This class defines a cryptographic context associated to one SRTP
// flow. This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.
// This implementation makes use of the libsrtp cryptoengine.
// Version: 1.0

// libsrtp and its APIs is © by Cisco Systems, 2001, 2002.
// HMAC-SHA1 implementation is © by Aaron Gifford, 1998, 2000.
/*****************************************************************************
 *                                                                           *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                           *
 *****************************************************************************/

#ifndef CRYPTOCONTEXT_H
#define CRYPTOCONTEXT_H

#include <string.h>
#include <stdio.h>
#include <syslog.h>

extern "C" {
   #include"crypto/include/cipher.h"
}
extern "C" {
   #include"crypto/include/auth.h"
}
extern "C" {
   #include"crypto/include/rdbx.h"
}
extern "C" {
   #include"crypto/include/datatypes.h"
}
extern "C" {
   #include"crypto/include/integers.h"
}
extern "C" {
   #include"crypto/include/err.h"
}
extern "C" {
   #include"crypto/include/rijndael.h"
}
extern "C" {
   #include"crypto/include/rijndael-icm.h"
}

extern "C" {
   #include "hmac_sha1.h"
}



enum srtp_serv{
        NONE,           // No security services provides
        CONF,           // Confidentiality
        AUTH,           // Message Authentication
        BOTH            // Both Services
};
```

```
enum cipher_id{
        NULLCIPHER,
        AESCM
};

enum auth_id{
        NULLAUTH,
        HMACSHA1
};


class CryptoContext {
public:
        CryptoContext(string key, string salt, string service, string cipher_type,
                      string auth_type);
        ~CryptoContext();
        void set_key_deriv_rate(int r); // For re-keying
        unsigned int get_roc();         // NOT USED, handled by libsrtp cryptoengine
        void update_roc();              // NOT USED, handled by libsrtp cryptoengine
        void reset_roc();               // NOT USED, handled by libsrtp cryptoengine
        unsigned int get_tag_len();     // Length of the authentication tag
        int get_serv();                 // Securty services provided
        cipher_t *get_cipher_t();       // Type of cipher
        auth_t *get_auth_t();           // Type of authentication algorithm
        short get_s_l();
        err_status_t init_aes_128_prf(const unsigned char key[16],
                                      const unsigned char salt[14]);
                                   // Initializes the context and derives the keys

        rdbx_t *rl_snd;            // Sender's replay list
        rdbx_t *rl_rcv;            // Receiver's replay list
        cipher_t *encryptor;
        auth_t *authenticator;
        HMAC_SHA1_CTX hmac_ctx;    // HMAC-SHA1 context
        unsigned char *auth_tag;   // Holds temporary storage for authentication tag
                                // (used in comparison)

private:
        auth_id auth_name;
        cipher_id cipher_name;
        unsigned int MKI_ind;      // 0 if MKI is not present; otherwise 1
        unsigned int MKI_len;      //if MKI_ind=1   fixed for the context lifetime
        unsigned int MKI_value;    //if MKI_ind=1   fixed for the context lifetime
        unsigned char masterkey[16];           // Hardcoded
        unsigned char mastersalt[14];          // Hardcoded
        int key_pkt_ctr_srtp;      // SRTP packet counter
        int key_pkt_ctr_srtcp;     // SRTCP packet counter
        unsigned int n_e;          // Encryption key length
        unsigned int n_a;          // Authentication key length
        unsigned int n_s;          // Session salt key length
        unsigned int n_b;          // Bit size of the block for the block cipher
        unsigned int tag_len;      // Authentication tag length (by default 32 bits)
        unsigned int key_deriv_rate;
        unsigned int  roc;          // ROC (32 bits)  Each time the session starts, the
                                     //sender sets it to zero! Only for SRTP
        short s_l;                  // Highest received RTP seq number. For SRTCP
                                     //(instead of ROC)
        srtp_serv services;
        // From & To fields not specified
};

#endif
```

```
/***************************************************************************
                          CryptoContext.cxx  -  description
                             -------------------
    begin                : Fri Apr 4 2003
    copyright            : (C) 2003 by Israel Abad
    email                : israel@kth.se
                           : i_abad@terra.es
 ***************************************************************************/
// This class defines a cryptographic context associated to one SRTP
// flow. This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.

// Version: 1.0
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include "CryptoContext.h"
#include <errno.h>
#include <stdio.h>
#include <unistd.h>
#include <string>
#include <syslog.h>
#include <iostream>
#include"../util/ConfigFile.h"

extern "C" {
   #include"crypto/include/cipher.h"
}
extern "C" {
   #include"crypto/include/auth.h"
}
extern "C" {
   #include"crypto/include/rdbx.h"
}
extern "C" {
   #include"crypto/include/datatypes.h"
}
extern "C" {
   #include"crypto/include/integers.h"
}
extern "C" {
   #include"crypto/include/err.h"
}
extern "C" {
   #include"crypto/include/rijndael.h"
}
extern "C" {
   #include"crypto/include/rijndael-icm.h"
}
extern "C" {
   #include"hmac_sha1.h"
}
extern "C" auth_type_t hmac_sha1;
extern   "C" cipher_type_t rijndael_icm;
extern  "C" auth_type_t null_auth;
extern "C" cipher_type_t null_cipher;

typedef enum {                                                  // Used for key
derivation
  label_encryption = 0x00,
  label_message_authentication = 0x01,
  label_salt  = 0x02
} srtp_prf_label;


CryptoContext::CryptoContext(string key, string salt, string service, string cipher_type,
                 string auth_type){ // Uses AES and HMAC-SHA1 (BOTH services) by default

         if (service == "none")
            this->services = NONE;    // No Confidentiality and No Message Authentication
         else if (service == "confidentiality")
            this->services = CONF;    // Confidentiality and No Message Authentication
         else if (service == "msg_authentication")
```

```
    this->services = AUTH;    // No Confidentiality and Message Authentication
  else if (service == "both")
    this->services = BOTH;    // Confidentiality and Message Authentication
  else{
    syslog(LOG_ERR,"No service or invalid service specified. Assuming none");
    this->services = NONE;// No Confidentiality and No Message Authentication
 }

 encryptor = new cipher_t;
 authenticator = new auth_t;

 if (cipher_type == "aes"){
        this->cipher_name = AESCM;                // AES Counter Mode
        this->encryptor->type = &rijndael_icm;     // rijndael_icm
                                 //(rijndael_icm is AES in Counter Mode)
 }
 else if (cipher_type == "null"){
        this->cipher_name = NULLCIPHER;
        this->encryptor->type = &null_cipher;     // null_cipher
 }
 else{
   syslog(LOG_ERR,"No cipher type or invalid cipher type specified specified.
                  Assuming AES");
   this->cipher_name = AESCM;
   this->encryptor->type = &rijndael_icm;     // rijndael_icm     (rijndael_icm is
                                             //AES in Counter Mode)
 }

 if (auth_type == "hmac"){
    this->auth_name = HMACSHA1;
    this->authenticator->type = &hmac_sha1;
 }
 else if (auth_type == "null"){
    this->auth_name = NULLAUTH;
    this->authenticator->type = &null_auth; // null authenticator: NOT RECOMMENDED
 }
 else{
    syslog(LOG_ERR,"No authenticator type or invalid authenticator type specified
                        specified. Assuming HMAC-SHA1");
    this->auth_name = HMACSHA1;
    this->authenticator->type = &hmac_sha1;
 }

 if (services == CONF)          // Avoids interaction with HMAC-SHA1 (BUG)
        this->auth_name = NULLAUTH;
 this->MKI_ind = 0;             // MKI not present
 rl_snd =  new rdbx_t;          // Init replay list = set to zero
 rl_snd->index.roc = 0;
 rl_snd->index.seq = 0;
 rl_rcv = new rdbx_t;           // not used
 this->roc = 0;        // ROC initialized to zero. Handled by libsrtp cryptoengine
 this->s_l = 0;                          // Only for SRTCP
 this->key_pkt_ctr_srtp = 0;
 this->key_pkt_ctr_srtcp = 0;
 this->key_deriv_rate = 0;                       // No re-keying needed
 this->encryptor->key_len = 16;
 this->encryptor->salt_len = 14;
 this->authenticator->key_len = 16;
 if (services == CONF)           // Avoids interaction with HMAC-SHA1  (BUG)
        this->authenticator->type = &null_auth;
 if (services == BOTH || services == AUTH){  // If message authentication is
//provided, set length of the tag to 4 bytes (default), otherwise, no tag needed
        this->authenticator->out_len = 4;
        this->auth_tag = new unsigned char;
 }
 else{
        this->authenticator->out_len = 0;
        this->auth_tag = NULL;
 }
 this->n_e = this->encryptor->key_len;
 this->n_a = this->authenticator->key_len;
 this->n_s = this->encryptor->salt_len;
 this->n_b = 128;
 this->tag_len = this->authenticator->out_len;
 // Generation of master keys (Harcoded)      <----- Now in config file
 //for (unsigned int i=0; i<this->n_e; i++)
 //        this->masterkey[i]='M';
 //for (unsigned int i=0; i<this->n_s; i++)
 //        this->mastersalt[i]='S';
```

```
        //string key = config.get_string("master_key");


        for (unsigned int i=0; i<this->n_e; i++)
                this->masterkey[i]=key[i];

        for (unsigned int i=0; i<this->n_s; i++)
                this->mastersalt[i]=salt[i];

        // Cipher and Authenticator Allocation
        err_status_t status;
        status = cipher_type_alloc(this->encryptor->type, &this->encryptor, this->n_e);
        if (status)
                syslog(LOG_ERR,"ERROR: CIPHER NOT ALLOCATED!");

        if (this->authenticator->type != &hmac_sha1){
                status = auth_type_alloc(this->authenticator->type, &this->authenticator,
                                         this->n_a, this->tag_len);
                if (status){
                        syslog(LOG_ERR,"ERROR: AUTHENTICATOR NOT ALLOCATED!");
                        cipher_dealloc(this->encryptor);
                }
        }

        status = init_aes_128_prf(this->masterkey, this->mastersalt);
        if (status)
                cout << "ERROR: INIT PRF!" <<endl;
}

CryptoContext::~CryptoContext(){
}

void CryptoContext::set_key_deriv_rate(int r){
        key_deriv_rate = r;
}

unsigned int CryptoContext::get_roc(){
        return roc;
}

void CryptoContext::update_roc(){
        this->roc += 1;             //Wrapping not considered: handled by libsrtp engine
}

void CryptoContext::reset_roc(){
        this->roc = 0;
}

unsigned int CryptoContext::get_tag_len(){
        return tag_len;
}

int CryptoContext::get_serv(){
        switch (services){
                case NONE:
                        return 1;
                case CONF:
                        return 2;
                case AUTH:
                        return 3;
                case BOTH:
                        return 4;
                default:
                        return 0;
        }
}

cipher_t *CryptoContext::get_cipher_t(){
        return encryptor;
}

auth_t *CryptoContext::get_auth_t(){
        return authenticator;
}

short CryptoContext::get_s_l(){
        return s_l;
}
```

```
// Key derivation for SRTP session and Cipher and Authenticator Initialization (function
// based on libsrtp, from David McGrew)

err_status_t CryptoContext::init_aes_128_prf(const unsigned char key[16],
                                             const unsigned char salt[14]) {
        err_status_t stat;
        rijndael_icm_context c;
        xtd_seq_num_t idx = { 0, 0 };
// For setting icm to zero-index
        unsigned char *buffer = new unsigned char[this->n_e + this->n_s + this->n_a];
// Temporary storage for keystream
        unsigned char *enc_key_buf, *enc_salt_buf, *auth_key_buf;
        unsigned char initial_counter[16];
        // Set intitial_counter
        initial_counter[0]  = salt[0];
        initial_counter[1]  = salt[1];
        initial_counter[2]  = salt[2];
        initial_counter[3]  = salt[3];
        initial_counter[4]  = salt[4];
        initial_counter[5]  = salt[5];
        initial_counter[6]  = salt[6];
        initial_counter[7]  = salt[7];
        initial_counter[8]  = salt[8];
        initial_counter[9]  = salt[9];
        initial_counter[10] = salt[10];
        initial_counter[11] = salt[11];
        initial_counter[12] = salt[12];
        initial_counter[13] = salt[13];
        initial_counter[14] = 0x00;
        initial_counter[15] = 0x00;
        // Set pointers
        enc_key_buf = buffer;
        enc_salt_buf = buffer + this->n_e;
        auth_key_buf = enc_salt_buf + this->n_s;
        // Generate encryption key, putting it into enc_key_buf
        // Note that we assume that index DIV t == 0 in this implementation
        initial_counter[7] = salt[7] ^ label_encryption;
        rijndael_icm_context_init(&c, key, initial_counter);
        rijndael_icm_set_segment(&c, idx);
        rijndael_icm_encrypt(&c, enc_key_buf, this->n_e);
        // Generate encryption salt, putting it into enc_salt_buf
        initial_counter[7] = salt[7] ^ label_salt;
        rijndael_icm_context_init(&c, key, initial_counter);
        rijndael_icm_set_segment(&c, idx);
        rijndael_icm_encrypt(&c, enc_salt_buf, this->n_s);
        // We don't yet know the ssrc of the sender, so we don't exor the
        // ssrc value into the enc_salt_buf
        // Initialize cipher
        stat = cipher_init(encryptor, enc_key_buf, enc_salt_buf);
        if (stat) {
                delete [] buffer;
                return err_status_alloc_fail;
                syslog(LOG_ERR,"ERROR: CIPHER INIT!");
        }
        // Generate authentication key, putting it into auth_key_buf
        initial_counter[7] = salt[7] ^ label_message_authentication;
        rijndael_icm_context_init(&c, key, initial_counter);
        rijndael_icm_set_segment(&c, idx);
        rijndael_icm_encrypt(&c, auth_key_buf, this->n_a);
        // Initialize authenticator

       if (authenticator->type != &hmac_sha1){
                auth_init(authenticator, auth_key_buf, this->n_a);
                if (stat) {
                        delete [] buffer;
                        syslog(LOG_ERR,"ERROR: AUTHENTICATOR INIT!");
                        return err_status_init_fail;
                }
       }
       else{
                HMAC_SHA1_Init(&hmac_ctx);
                HMAC_SHA1_UpdateKey(&hmac_ctx, auth_key_buf, this->n_a);
                HMAC_SHA1_EndKey(&hmac_ctx);
       }

        // Free memory then return
        delete [] buffer;
        return err_status_ok;
}
```

```
/***************************************************************************
                          SRtpHeader.h  -  description
                             -------------------
    begin                : Wed Mar 26 2003
    copyright            : (C) 2003 by Israel Abad
                           TSLab, IMIT, KTH, Stockhlom, Sweden
    email                : israel@kth.se
                         : i_abad@terra.es
 ***************************************************************************/
// This class defines a SRTP Header associated to one SRTP
// packet. This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.
// This implementation makes use of the libsrtp cryptoengine.
// Version: 1.0

// libsrtp and its APIs is © by Cisco Systems, 2001, 2002.
// HMAC-SHA1 implementation is © by Aaron Gifford, 1998, 2000.
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/
#ifndef SRTPHEADER_H
#define SRTPHEADER_H

#include"vector"

#include"RtpHeader.h"

class SRtpHeader : public RtpHeader{
        public:
                SRtpHeader();
                void operator=(RtpHeader &rtph);
        // private:
};

#endif
```

```
/***************************************************************************
                          SRtpHeader.cxx  -  description
                             -------------------
    begin                : Wed Mar 26 2003
    copyright            : (C) 2003 by Israel Abad
                           TSLab, IMIT, KTH, Stockhlom, Sweden
    email                : israel@kth.se
                         : i_abad@terra.es

 ***************************************************************************/
// This class defines a SRTP Header associated to one SRTP
// packet. This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.
// This implementation makes use of the libsrtp cryptoengine.
// Version: 1.0

// libsrtp and its APIs is © by Cisco Systems, 2001, 2002.
// HMAC-SHA1 implementation is © by Aaron Gifford, 1998, 2000.
/***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include"SRtpHeader.h"
#include<netinet/in.h>
#include"SRtpPacket.h"

SRtpHeader::SRtpHeader(): RtpHeader(){
}

void SRtpHeader::operator=(RtpHeader &h){
        this-> version =h.version;
        this->extension = h.extension;
        this->marker = h.marker;
        this->payload_type = h.payload_type;
        this->sequence_number = h.sequence_number;
        this->timestamp = h.timestamp;
        this->SSRC = h.SSRC;
        this->CSRC = h.CSRC;
}
```

```
/*
************************************************
RTP Header declaration
Author: Erik Eliasson. TSLab, IMIT KTH Stockhlom
         eliasson@it.kth.se
copyright: (C) 2003 by Erik Eliasson
NOTE: This is not the original Erik Eliasson's
       header file. This has been modified for this
       MINIsrtp version.
************************************************
*/

#ifndef RTPHEADER_H
#define RTPHEADER_H

#include"vector"

class RtpHeader{

       public:
              RtpHeader();
              void set_version(int v);
              void set_extension(int x);
              void set_CSRC_count(int cc);
              int get_CSRC_count();
              void set_marker(int m);
              void set_payload_type(int pt);
              int get_payload_type();
              void set_seq_no(int seq_no);
              int get_seq_no();
              void set_timestamp(int timestamp);
              void set_SSRC(int ssrc);
              void add_CSRC(int csrc);
              void print_debug();
              int size();
              char *get_bytes();
              int CSRC_count;
              int version;
              int extension;
              int marker;
              int payload_type;
              int sequence_number;
              int timestamp;
              int SSRC;
              vector<int> CSRC;
};

#endif
```

```
/*
************************************************
RTP Header definition
Author: Erik Eliasson. TSLab, IMIT KTH Stockhlom.
        eliasson@it.kth.se
Copyright: (C) 2003 by Erik Eliasson
************************************************
*/


#include"RtpHeader.h"
#include<netinet/in.h>
#include"RtpPacket.h"

RtpHeader::RtpHeader(){
       version=0;
       extension=0;
       CSRC_count=0;
       marker=0;
       payload_type=0;
       sequence_number=0;
       timestamp=0;
       SSRC=0;
}

void RtpHeader::set_version(int v){
       this->version = v;
}

void RtpHeader::set_extension(int x){
       this->extension =  x;
}

void RtpHeader::set_CSRC_count(int cc){
       this->CSRC_count = cc;
}

int RtpHeader::get_CSRC_count(){
  return this->CSRC_count;
}

void RtpHeader::set_marker(int m){
       this->marker = m;
}

void RtpHeader::set_payload_type(int pt){
       this->payload_type=pt;
}

int RtpHeader::get_payload_type(){
       return payload_type;
}

void RtpHeader::set_seq_no(int seq_no){
       this->sequence_number=seq_no;
}

int RtpHeader::get_seq_no(){
       return sequence_number;
}

void RtpHeader::set_timestamp(int timestamp){
       this->timestamp = timestamp;
}

void RtpHeader::set_SSRC(int s){
       this->SSRC = s;
}

void RtpHeader::add_CSRC(int c){
       CSRC.push_back(c);
}

int RtpHeader::size(){
       return 12+4*CSRC.size();
}

char *RtpHeader::get_bytes(){
       char *ret = new char[size()+4*CSRC.size()];
```

```
	struct rtpheader *hdrptr = (struct rtpheader *)ret;
	hdrptr->v=version;
	hdrptr->x=extension;
	hdrptr->cc=CSRC_count;
	hdrptr->m=marker;
	hdrptr->pt=payload_type;
	hdrptr->seq_no=htons(sequence_number);
	hdrptr->timestamp=htonl(timestamp);
	hdrptr->ssrc=htonl(SSRC);

	for (unsigned i=0; i<CSRC.size(); i++)
		((int *)ret)[3+i]=htonl(CSRC[i]);
	return ret;
}

void RtpHeader::print_debug(){
	cerr << "\tversion: "<< version<<"\n\textension: "<< extension <<
	"\n\tCSRC count: "<< CSRC_count << "\n\tmarker: "<< marker <<
	"\n\tpayload type: "<<payload_type <<"\n\tsequence number: "<<sequence_number <<
	"\n\ttimestamp: "<<timestamp <<"\n\tSSRC: "<< SSRC << "\n"<< endl;

	for (int i=0; i< CSRC_count; i++)
		cerr << "\tCSRC "<<i+1 << ": "<<CSRC[i]<< endl;
}
```

```
/***************************************************************************
                          SRtpPacket.h  -  description
                             -------------------
    begin                : Wed Mar 26 2003
    copyright            : (C) 2003 by Israel Abad
                           TSLab, IMIT, KTH, Stockhlom, Sweden
    email                : israel@kth.se
                         : i_abad@terra.es
 ***************************************************************************/
// This class defines a SRTP packet SRTP packet.
// This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.
// This implementation makes use of the libsrtp cryptoengine.
// Version: 1.0

// libsrtp and its APIs is © by Cisco Systems, 2001, 2002.
// HMAC-SHA1 implementation is © by Aaron Gifford, 1998, 2000.
 /***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#ifndef SRTPPACKET_H
#define SRTPPACKET_H

#include "RtpPacket.h"
#include "SRtpHeader.h"
#include "../../minisip/ipv6util/UDPSocket.h"
#include "../../minisip/ipv6util/IPAddress.h"
#include "CryptoContext.h"
#include <syslog.h>

extern "C" {
    #include "crypto/include/cipher.h"
    #include "crypto/include/auth.h"
    #include "crypto/include/rdbx.h"
    #include "crypto/include/datatypes.h"
    #include "crypto/include/integers.h"
    #include "crypto/include/err.h"
    #include "crypto/include/rijndael.h"
    #include "crypto/include/rijndael-icm.h"
}
struct srtpheader{
        unsigned cc:4;
        unsigned x:1;
        unsigned p:1;
        unsigned v:2;
        unsigned pt:7;
        unsigned m:1;
        unsigned seq_no:16;
        unsigned timestamp:32;
        unsigned ssrc:32;
};

class SRtpPacket{
  public:
    SRtpPacket();
    SRtpPacket(CryptoContext *scontext, RtpPacket *rtppacket);
                    // Used by the sender

    SRtpPacket(SRtpHeader hdr, void *content, int content_length);
                    // Used by the receiver

    SRtpPacket(SRtpHeader hdr, void *content, unsigned char *tag,
                int content_length);
                    // Used in the case message auth were provided

    ~SRtpPacket();


    void send_to(CryptoContext *scontext, UDPSocket &udp_sock, IPAddress &to_addr);
                  // Sends a packet via socket

    static SRtpPacket *receive_from(UDPSocket &udp_sock, CryptoContext *scontext,
                             int timeout=-1);
```

```
                    // Receives a packet via socket

    RtpPacket *get_rtp_packet(CryptoContext *scontext, SRtpPacket *pkt);
                    // Gets a RTP packet from a SRTP packet via "unprotect"

    SRtpHeader &get_header();           // returns de header

    void *get_content();                // returns a pointer to the content

    unsigned char *get_tag();           // returns a pointer to the authentication tag

    void remove_tag();                  // removes the authentication tag

    void set_tag(unsigned char *tag);   // sets an authentication tag

    int get_content_length();           // returns the length of the payload

    char *get_bytes(CryptoContext *sctx);
                    // returns a pointer to an array containing the SRTP packet

    int size(CryptoContext *sctx);      // returns the size of the packet

    SRtpHeader header;                  // public for simplicity


  private:
    int content_length;
    void *content;
    unsigned char *tag;
    CryptoContext *scontext;
};

#endif
```

```
/***************************************************************************
                          SRtpPacket.cxx  -  description
                             -------------------
    begin                : Wed Mar 26 2003
    copyright            : (C) 2003 by Israel Abad
                           TSLab, IMIT, KTH, Stockhlom, Sweden
    email                : israel@kth.se
                           : i_abad@terra.es
 ***************************************************************************/
// This class defines a cryptographic context associated to one SRTP
// flow. This class belongs to the MINIsrtp implementation, developed
// at TSLab IMIT KTH, Stockholm.
// This implementation makes use of the libsrtp cryptoengine.
// Version: 1.0

// libsrtp and its APIs is © by Cisco Systems, 2001, 2002.
// HMAC-SHA1 implementation is © by Aaron Gifford, 1998, 2000.
 /***************************************************************************
 *                                                                         *
 *   This program is free software; you can redistribute it and/or modify  *
 *   it under the terms of the GNU General Public License as published by  *
 *   the Free Software Foundation; either version 2 of the License, or     *
 *   (at your option) any later version.                                   *
 *                                                                         *
 ***************************************************************************/

#include<sys/poll.h>
#include<errno.h>
#include<netinet/in.h>
#include<sys/socket.h>
#include<stdio.h>
#include<unistd.h>
#include<string.h>
#include"SRtpPacket.h"
#include"SRtpHeader.h"
#include"CryptoContext.h"
#include <err.h>
#include <syslog.h>

extern "C" {
    #include "crypto/include/cipher.h"
    #include "crypto/include/auth.h"
    #include "crypto/include/rdbx.h"
    #include "crypto/include/rdb.h"
    #include "crypto/include/datatypes.h"
    #include "crypto/include/integers.h"
    #include "crypto/include/err.h"
    #include "crypto/include/rijndael.h"
    #include "crypto/include/rijndael-icm.h"
    #include "hmac_sha1.h"
}
extern "C" cipher_type_t rijndael_icm;
extern "C" auth_type_t hmac_sha1;

// Some auxiliar functions to protect and unprotect the data

unsigned char  *protect (CryptoContext *scontext, RtpPacket *srtppacket, int *len, int
content_len){
        xtd_seq_num_t index;
        int status;
        int delta;
        void *encr_port = srtppacket->get_content();
        unsigned char *ptr_empty = new unsigned char;



       cout << "Displaying SRTP payload before protection: " << (char *)encr_port <<endl;

       scontext->set_key_deriv_rate(0);               // No re-keying provided

        // Index estimation and index addition

       delta = rdbx_estimate_index(scontext->rl_snd, &index,
                                   ntohs(srtppacket->header.get_seq_no()));
        if (delta > 0)
                rdbx_add_index(scontext->rl_snd, delta);
        else {
                syslog(LOG_ERR,"Something unexpected occurred when estimating the index.
                        MAC not calculated. Packet might be corrupted!");
```

```
                return ptr_empty;  //NULL
        }

        // dealing with packet loss: set the cipher to the proper keystream segment

        status = cipher_set_segment(scontext->encryptor, index);
        if (status);


        // Providing encryption

       if (scontext->get_serv() == 2 || scontext->get_serv() == 4)
                status = cipher_encrypt(scontext->get_cipher_t(),
                                        (unsigned char*)encr_port, content_len);

          cout << "Proving encrypted portion: " << (char *)encr_port <<endl;
          cout << "Displaying SRTP payload after encryption: " <<
           (char*)srtppacket->get_content() <<endl;

        // Providing message authentication

         if (scontext->get_serv() == 3 || scontext->get_serv() == 4){
                unsigned char *auth_tag = new unsigned char[scontext->get_tag_len()];
            for (unsigned int i=0; i<scontext->get_tag_len(); i++)
                  auth_tag[i]=0;
                unsigned char *aux_tag;
                // NO HMAC-SHA1 (Harcoded NULL Authenticator
                if (scontext->authenticator->type != &hmac_sha1){
                        cout <<"Using NULL Authenticator" <<endl;
                        status = auth_compute(scontext->get_auth_t(),
                                              (octet_t*)srtppacket, *len, auth_tag);
                        if (status){
                         syslog(LOG_ERR,"ERROR:Null Authentication computation failed!");
                          return ptr_empty;  //NULL
                        }
                }

                //HMAC-SHA1
                else{
                   cout <<"Using HMAC-SHA1" <<endl;
                   aux_tag = new unsigned char[srtppacket->header.size()+content_len+4];
                                    //auxiliar storage

                    for (int i=0; i<srtppacket->header.size()+content_len+4; i++)
                        aux_tag[i]=0;
                        memcpy(aux_tag, srtppacket, srtppacket->header.size() +
                                 content_len);
                         memcpy(&aux_tag[srtppacket->header.size()+content_len],
                                 &index.roc,4)
                        // Computation
                        HMAC_SHA1_StartMessage(&(scontext->hmac_ctx));
                        HMAC_SHA1_UpdateMessage(&(scontext->hmac_ctx), aux_tag,
                                        srtppacket->header.size()+content_len+4);
                        HMAC_SHA1_EndMessage(aux_tag, &(scontext->hmac_ctx));
                        cout << "Displaying Auxiliar tag code: " << aux_tag << endl;
                        // Storing the authentication tag
                        memcpy(auth_tag, aux_tag, scontext->get_tag_len());// Truncated
                        cout << "Displaying Auth tag code: " << auth_tag << endl;
                        delete [] aux_tag;
                }
                *len += scontext->get_tag_len();       // Modify length of the packet
                return auth_tag;
        }
        else
                return NULL;     // No Authentication tag needed

}

int unprotect (CryptoContext *scontext, SRtpPacket *srtppacket, int *len,
               int content_len){
        xtd_seq_num_t index;
        void *encr_port = srtppacket->get_content();
        unsigned char *aux_tag;
        int status;
        int delta;

        scontext->set_key_deriv_rate(0);               // No re-keying provided

        // Determination of the index and replay check
```

```
        delta = rdbx_estimate_index(scontext->rl_snd, &index,
                                    ntohs(srtppacket->header.get_seq_no()));
         if (rdbx_check(scontext->rl_snd, delta) != replay_check_ok){
                syslog(LOG_ERR,"ERROR: Replay detected!!!");
                return err_status_replay_fail;
         }

          // dealing with packet loss: set the cipher to the proper keystream segment */
          status = cipher_set_segment(scontext->encryptor, index);
          if (status);


         // Message authetication verification

        if (scontext->get_serv() == 3 || scontext->get_serv() == 4){
                if (scontext->authenticator->type != &hmac_sha1){
                        // NO HMAC-SHA1 (Harcoded NULL Authenticator)
                        cout <<"Using NULL Authenticator, verifying..." <<endl;
                        status = auth_compute(scontext->get_auth_t(),
                                 (octet_t*)srtppacket, *len - scontext->get_tag_len(),
                                  scontext->auth_tag); // Store into CryptoContext
                        if (status){
                                syslog(LOG_ERR,"Authentication computation fail");
                                return err_status_auth_fail;
                        }
                }

                // HMAC-SHA1
                else{
                    cout <<"Using HMAC-SHA1, verifying..." <<endl;
                    aux_tag = new unsigned char[srtppacket->header.size() +
                           content_len+4];   // Auxiliar storage for computation
                    for (int i=0; i<srtppacket->header.size()+content_len+4; i++)
                         aux_tag[i]=0;
                    memcpy(aux_tag, srtppacket, srtppacket->header.size()+content_len);
                    memcpy(&aux_tag[srtppacket->header.size()+content_len],
                           &index.roc,4)
                    // Computation. A. Gifford implementation of HMAC-SHA1
                    HMAC_SHA1_StartMessage(&(scontext->hmac_ctx));
                    HMAC_SHA1_UpdateMessage(&(scontext->hmac_ctx), aux_tag,
                           srtppacket->header.size() + content_len + 4);
                    HMAC_SHA1_EndMessage(aux_tag, &(scontext->hmac_ctx));
                    memcpy(scontext->auth_tag, aux_tag, scontext->get_tag_len());
                           // Truncated: Only 32 out of 160 stored into CryptoContext
                    delete [] aux_tag;
                }
                Verifying message authentication
                unsigned char *pkt_tag = srtppacket->get_tag();

                // Compare computed auth tag with that in the packet
                if (octet_string_is_eq(scontext->auth_tag, pkt_tag,
                    scontext->get_tag_len())){

                    syslog(LOG_ERR,"ERROR: Message Authentication, verification
                                failed!!!!");
                    delete [] scontext->auth_tag;
                    srtppacket->remove_tag();
                    *len -= scontext->get_tag_len();
                    return err_status_auth_fail;
                }
                //cout << "Message Authentication successful" << endl;
                // Modify the length of the packet and removing tag
                *len -= scontext->get_tag_len();
                delete [] scontext->auth_tag;
                srtppacket->remove_tag();
        }

        // Decryption

        if (scontext->get_serv() == 2 || scontext->get_serv() == 4)
                status = cipher_encrypt(scontext->get_cipher_t(),
                         (unsigned char*)encr_port, content_len);

        rdbx_add_index(scontext->rl_snd, delta);    // Add index to the list
        return 0;
}
```

```
// End of auxiliar functions

SRtpPacket::SRtpPacket(){
        content_length=0;
        content=NULL;
}

SRtpPacket::SRtpPacket(CryptoContext *scontext, RtpPacket *rtppacket):scontext(scontext){
        int len = rtppacket->size();
        int content_len = rtppacket->get_content_length();

        tag = new unsigned char;
        content_length = rtppacket->get_content_length();
        header = rtppacket->get_header();
        content = rtppacket->get_content();

        // Packet protection

       cout << "Protecting RTP packet...: "<< endl;
        this->tag = protect(scontext, rtppacket, &len, content_len);

        content = rtppacket->get_content();

         // Payload Length
         if (scontext->get_serv() == 1 || scontext->get_serv() == 2)
                this->content_length = len - header.size();
        else
                this->content_length = len - header.size() - scontext->get_tag_len();
}

SRtpPacket::SRtpPacket(SRtpHeader hdr, void *content, int content_length): header(hdr),
                        content(content){
        this->content_length = content_length;
        header.set_version(2);
}

SRtpPacket::SRtpPacket(SRtpHeader hdr, void *content, unsigned char * tag, int
                        content_length): header(hdr), content(content), tag(tag){
        this->content_length = content_length;
        header.set_version(2);
}

SRtpPacket::~SRtpPacket(){
}

RtpPacket *SRtpPacket::get_rtp_packet(CryptoContext *scontext, SRtpPacket *pkt){
        int len;
        RtpPacket *rtppacket;

        if (scontext->get_serv() == 1 || scontext->get_serv() == 2)
                len = 12 + pkt->get_content_length() + 4*pkt->header.CSRC_count;
        else
                len = 12 + pkt->get_content_length() + 4*pkt->header.CSRC_count +
                      scontext->get_tag_len();

        // Unprotecting SRTP packet

       if (unprotect (scontext, pkt, &len, pkt->get_content_length()) !=
                        err_status_auth_fail){
              // RTP packet creation

                rtppacket = new RtpPacket((RtpHeader &)pkt->header, pkt->get_content(),
                                          pkt->get_content_length());
        }
        else
                rtppacket = new RtpPacket();      // Empty packet to be dropped
        return rtppacket;
}

void SRtpPacket::send_to(CryptoContext *scontext, UDPSocket &socket, IPAddress &to_addr){
        char *bytes = get_bytes(scontext);
        socket.sendto(to_addr, to_addr.get_port(), bytes, size(scontext));
                 // The packet size depends on the services provided
        delete [] bytes;
}


SRtpPacket *SRtpPacket::receive_from(UDPSocket &srtp_socket, CryptoContext *scontext, int
                                     timeout=-1){
```

```
        int i;
        char buf[2048];
        for (i=0; i<2048; i++)
                buf[i]=0;

        struct pollfd p;
        p.fd = srtp_socket.get_fd();
        p.events = POLLIN;
        int avail;

        do{
                avail = poll(&p,1,timeout);
                if (avail==0){
                return NULL;
                }
                if (avail<0){
                        if (errno!=EINTR){
                                syslog(LOG_ERR,"Error when using poll");
                                exit(1);
                        }
                        else{
                                syslog(LOG_WARNING,"Signal occured in wait_packet");
                        }
                }
        }while(avail < 0);

        i = recvfrom(srtp_socket.get_fd(), buf, 2048, 0, /*(struct sockaddr *)
                        &from*/NULL, /*(socklen_t *)fromlen*/NULL);
        if (i<0){
        syslog(LOG_ERR,"recvfrom");
        }

        // Creating SRTP packet...

        srtpheader *hdrptr=(srtpheader *)&buf[0];
        SRtpHeader hdr;
        hdr.set_version(hdrptr->v);
        hdr.set_extension(hdrptr->x);
        hdr.set_CSRC_count(hdrptr->cc);
        hdr.set_marker(hdrptr->m);
        hdr.set_payload_type(hdrptr->pt);
        hdr.set_seq_no(ntohs(hdrptr->seq_no));
        hdr.set_timestamp(ntohl(hdrptr->timestamp));
        hdr.set_SSRC(ntohl(hdrptr->ssrc));
        for (unsigned j=0; j<hdrptr->cc; j++)
                hdr.add_CSRC(ntohl( ((int *)&buf[12])[j] ));
        int datalen = i - 12 - hdrptr->cc*4-scontext->get_tag_len();
        int hdrctr = 12;
        void *data=new char[datalen];
        memcpy(data,&buf[hdrctr+4*hdrptr->cc],datalen);

        // If message authentication is provided
        if (scontext->get_serv() == 3 || scontext->get_serv() == 4){
                unsigned char *tag=new unsigned char[scontext->get_tag_len()];
                          //Creating auth tag
                memcpy(tag,&buf[hdrctr+4*hdrptr->cc+datalen],scontext->get_tag_len());
                SRtpPacket *srtp = new SRtpPacket(hdr, data, tag, datalen);
                return srtp;
        }

        // If no message authentication is provided
        SRtpPacket *srtp = new SRtpPacket(hdr, data, datalen);
        return srtp;
}

char *SRtpPacket::get_bytes(CryptoContext *sctx){
        char *ret = new char[header.size()+content_length+sctx->get_tag_len()];
        char *hdr = header.get_bytes();

        memcpy(ret, hdr, header.size());
        delete [] hdr;

        memcpy(&ret[header.size()], content, content_length);

        // If message authentication is provided
        if (sctx->get_serv() == 3 || sctx->get_serv() == 4)
               memcpy(&ret[header.size()+content_length], tag, sctx->get_tag_len());
        return ret;
}
```

```
SRtpHeader &SRtpPacket::get_header(){
        return header;
}

void *SRtpPacket::get_content(){
        return content;
}

unsigned char *SRtpPacket::get_tag(){
        return tag;
}

void SRtpPacket::remove_tag(){
      this->tag=NULL;
       delete [] tag;
}

void SRtpPacket::set_tag(unsigned char *tag){
        this->tag=tag;
}

int SRtpPacket::get_content_length(){
        return content_length;
}

int SRtpPacket::size(CryptoContext *sctx){
        if (sctx->get_serv() == 3 || sctx->get_serv() == 4)  // Message auth provided
                return 12 + 4*header.CSRC_count + content_length + sctx->get_tag_len();
        else
                return 12 + 4*header.CSRC_count + content_length;
}
```

Appendix B. A First Approach to a MIKEY Messages Implementation

```
/*
************************************************
MIKEY Message declaration (Start reference)
Author: Erik Eliasson. TSLab, IMIT KTH Stockhlom.
        eliasson@it.kth.se
Copyright: (C) 2003 by Erik Eliasson
version: 0.01
************************************************
*/

#ifndef MIKEYMESSAGE_H
#define MIKEYMESSAGE_H

#include<list>
#include"MikeyPayload.h"

class MikeyMessage{
       public:
              MikeyMessage(unsigned char *message, int length_limit);
              ~MikeyMessage();



       private:
              list<MikeyPayload *> payloads;
};


#endif
```

```
/*
************************************************
MIKEY Message definition (Start reference)
Author: Erik Eliasson. TSLab, IMIT KTH Stockhlom.
        eliasson@it.kth.se
Copyright: (C) 2003 by Erik Eliasson
version: 0.01
************************************************
*/

#include"MikeyMessage.h"
#include"MikeyPayload.h"
#include"MikeyPayloadHDR.h"
#include"MikeyPayloadKEMAC.h"
#include"MikeyPayloadPKE.h"
#include"MikeyPayloadDH.h"
#include"MikeyPayloadSIGN.h"
#include"MikeyPayloadT.h"
#include"MikeyPayloadID.h"
#include"MikeyPayloadCERT.h"
#include"MikeyPayloadCHASH.h"
#include"MikeyPayloadV.h"
#include"MikeyPayloadSP.h"
#include"MikeyPayloadRAND.h"
#include"MikeyPayloadERR.h"
#include"MikeyPayloadKeyData.h"
#include"MikeyPayloadGeneralExtension.h"
#include"MikeyException.h"

/*
 * Alg.
 *  1. Parse HDR payload
 *  2. While not end of packet
 *    2.1 Parse payload (choose right class) and store next payload type.
 *    2.2 Add payload to list of all payloads in message.
*/
MikeyMessage::MikeyMessage(unsigned char *message, int length_limit){
       unsigned char *msgpos = message;
       int limit = length_limit;

       MikeyPayloadHDR *hdr = new MikeyPayloadHDR(message, limit);       // 1.

       limit-=  (hdr->get_end()-msgpos);
       msgpos = hdr->get_end();

       int next_payload_type = hdr->get_next_payload_type();

       while (!(msgpos >= message+length_limit) &&
              next_payload_type==MikeyPayload::LastPayload){
              MikeyPayload *payload;
              switch (next_payload_type){
                     case MIKEYPAYLOAD_KEMAC_PAYLOAD_TYPE:
                            payload = new MikeyPayloadKEMAC(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_PKE_PAYLOAD_TYPE:
                            payload = new MikeyPayloadPKE(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_DH_PAYLOAD_TYPE:
                            payload = new MikeyPayloadDH(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_SIGN_PAYLOAD_TYPE:
                            payload = new MikeyPayloadSIGN(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_T_PAYLOAD_TYPE:
                            payload = new MikeyPayloadT(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_ID_PAYLOAD_TYPE:
                            payload = new MikeyPayloadID(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_CERT_PAYLOAD_TYPE:
                            payload = new MikeyPayloadCERT(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_CHASH_PAYLOAD_TYPE:
                            payload = new MikeyPayloadCHASH(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_V_PAYLOAD_TYPE:
                            payload = new MikeyPayloadV(msgpos, limit);
                            break;
                     case MIKEYPAYLOAD_SP_PAYLOAD_TYPE:
```

```
                        payload = new MikeyPayloadSP(msgpos, limit);
                        break;
                case MIKEYPAYLOAD_RAND_PAYLOAD_TYPE:
                        payload = new MikeyPayloadRAND(msgpos, limit);
                        break;
                case MIKEYPAYLOAD_ERR_PAYLOAD_TYPE:
                        payload = new MikeyPayloadERR(msgpos, limit);
                        break;
                case MIKEYPAYLOAD_KEYDATA_PAYLOAD_TYPE:
                        payload = new MikeyPayloadKeyData(msgpos, limit);
                        break;
                case MIKEYPAYLOAD_GENERALEXTENSIONS_PAYLOAD_TYPE:
                        payload = new MikeyPayloadGeneralExtensions(msgpos, limit);
                        break;

                case MIKEYPAYLOAD_LAST_PAYLOAD:
                        break;
                default:
                        throw new MikeyExceptionMessageContent("Payload of unrecognized
                                                                type.");
                }

                next_payload_type = payload->get_next_payload_type();

                payloads.push_back(payload);                        // 2.2
                limit-=  (payload->get_end()-msgpos);
                msgpos = payload->get_end();
        }

        if (! (msgpos==message+length_limit && next_payload_type ==
                                                MIKEYPAYLOAD_LAST_PAYLOAD))
                throw new MikeyExceptionMessageLengthException("The length of the message
                                did not match the total length of payloads.");

}
```

```
/*
************************************************
Base class for all payloads in a MIKEY message
(Start reference)
Author: Erik Eliasson. TSLab, IMIT KTH Stockhlom.
        eliasson@it.kth.se
Copyright: (C) 2003 by Erik Eliasson
version: 0.01
************************************************
*/

#ifndef MIKEYPAYLOAD_H
#define MIKEYPAYLOAD_H


#define MIKEYPAYLOAD_LAST_PAYLOAD 0


class MikeyPayload{
       public:
              static const int LastPayload;


              MikeyPayload(unsigned char *start_of_message);
              virtual ~MikeyPayload();

              /**
               *
               * @returns The type of the payload that is starting
               *     at get_end().
               */
              int get_next_payload_type();

              /**
               *
               * @returns Pointer to the first memory location after
               *     this payload
               */
              unsigned char *get_end();

              virtual int get_length()=0;

       protected:
              void set_next_payload_type(int t);

              unsigned char *start;     //Points to the first memory position
                                  //within this payload.


              unsigned char *end;       //Points to the first memory position
                                        //after this payload
              int next_payload_type;

       private:
};


#endif
```

Appendix C. Acronyms

3DES	Triple DES
AAA	Authentication, Authorization, and Accounting
AES	Advanced Encryption Standard
AH	Authentication Header
ANSI	American National Standards Institute
AP	Access Point
CA	Certification Authority
CBC	Cipher Block Chaining
CDP	CRL Distribution Point
CHAP	Challenge Handshake Authentication Protocol
CM	See CTR
CRL	Certification Revocation List
CSRC	Contributing Source
CTR	Counter Mode
DEA	Data Encryption Algorithm
DES	Data Encryption Standard
DH	Diffie−Hellman
DNS	Domain Name Service
DNSSEC	DNS Security Architecture
DNS SRV	DNS Service
DoS	Denial of Service
DSS	Digital Signature Standard
EFF	Electronic Frontier Foundation
ESP	Encapsulating Security Payload
FIPS	Federal Information Processing Standards
FIPS PUBS	FIPS Publications
GPRS	General Packet Radio Service
HA	Home Agent
HCA	Hierarchical CA
HMAC	Hash−Based Message Authentication Code
HMAC−SHA1	HMAC based on SHA1
HTTP	Hyper−Text Transfer Protocol
ICV	Integrity Check Value
IDEA	International Data Encryption Algorithm
IEEE	Institute of Electrical and Electronics Engineers, Inc
IETF	Internet Engineering Task Force
IKE	Internet Key Exchange
IMIT	Department of Microelectronics and Information Technology
IP	Internet Protocol
IPRA	Internet Policy Registration Authority
IPSec	IP Security Architecture
ISAKMP	Internet Security Association and Key Management Protocol
ISO	International Organization for Standardization
ISP	Internet Service Provider
ITU	International Telecommunications Union
ITU−T	ITU Telecommunication Standardization
KTH	Kungl Tekniska Högskolan (Royal Institute of Technology)
LAN	Local Area Network

libsrtp	Library implementing SRTP (by David McGrew, Copyright © 2001, 2002 Cisco Systems, Inc. All rights reserved.)
MAC	Message Authentication Code
MD	Message Digest
MD5	Message Digest Algorithm 5
MDC	Message Description Code
MGCP	Media Gateway Control Protocol
MIKEY	Multimedia Internet KEYing
MINISIP	SIP User Agent Implementation (by Erik Eliasson)
MINIsrtp	SRTP implementation for integrating into MINISIP (by Israel Abad)
MKI	Mater Key Identifier
MN	Mobile Node
NAI	Network Access Identifier
NAPTR	Naming Authority Pointer
NIST	National Institute for Standards and Technology
OSI	Open System Interconnection
PC	Personal Computer
PCA	Policy CA
PEM	Privacy Enhanced Mail
PKI	Public Key Infrastructure
PSTN	Public Switched Telephone Network
QoS	Quality of Service
RADIUS	Remote Authentication Dial In User Service
RC5	Rivest Cipher 5
RFC	Request for Comments
ROC	Roll−Over Counter
RSA	Rivest, Shamir, and Adleman
RTCP	RTP Control Protocol
RTP	Real−Time Protocol
S/MIME	Secure Multipurpose Internet Mail Exchange
SDP	Session Description Protocol
SER	SIP Express Router
SHA1	Secure Hash Algorithm 1
SIP	Session Initiation Protocol
SRTCP	Secure RTCP
SRTP	Secure RTP
SSL	Secure Socket Layer
SSRC	Synchronization Source
TCA	Top CA
TCP	Transmission Control Protocol
TEK	Traffic Encrypting Key
TGK	TEK Generation Key
TLS	Transport Layer Security
TSLab	Telecommunication Systems Laboratory
TTP	Trusted Third Party. Also known as CA
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VOCAL	Vovida Open Communication Application Library
VoIP	Voice over Internet Protocol
VPN	Virtual Private Network
WAN	Wide Area Network
WLAN	Wireless LAN
XOR	Exclusive−or operation

Appendix D. Notation

$E_K(P)$ Encryption of the plaintext with the key *K*

$D_K(C)$ Decryption of the ciphertext with the key *K*

$H(x)$ Hash of the message *x*

$MAC_M = F(K_{AB}, M)$ MAC computation of the message *M*, by applying the function *F* with the key K_{AB}, to the message *M*

$HMAC_K(M) = H[(K^+ \oplus opad) || H[(K^+ \oplus ipad) || M]]$
HMAC computation of the message *M*, by applying the hash function *H* with the key *K* to the message *M*

X<<A>> Certification. *A*'s certificate is signed by the CA *X*

X<<Y>>Y<<B>> Chaining certification. *B*'s certificate is signed by the CA *Y*, whose certificate is in turn signed by the trusted CA *X*

I_MESSAGE = HDR, T, RAND, [IDi | CERTi], {SP}, DHi, SIGNi
Initiator's message in MIKEY's DH exchange method. The message consists of a MIKEY header (HDR), timestamp (T), a random value, the identity of the initiator and its certificate , the DH value of the initiator, the policies of the security protocol, and the signature of the initiator covering the entire message.

R_MESSAGE = HDR, T, [IDr | CERTr], IDi, DHr, DHi, SIGNr
Responder's message in MIKEY's DH exchange method. The message consists of a MIKEY header (HDR), timestamp (T), the identity of the responder and its certificate , the identity of the initiator, the DH value of the responder and the initiator, and the signature of the initiator covering the entire message.

Appendix E. Glossary

Authenticator	device which provides authentication services.
Bug	In programming, an error or unexpected behaviour of the code.
Certificate	structure for binding a user's identity to his/her public key. Issued and digitally signed by a Certification Authority.
Cracker Machine	device used to break the security provided by a cryptographic algorithm.
Cryptographic Engine	set of cryptographic operations, algorithms, and processes contained and performed in a encryption device.
Denial of Service Attack	security attack where an attacker floods the server with bogus requests, or tamper with legitimate requests. The attacker does nor benefit, but service is denied to legitimate users.
Digital Signature	method for verifying that a message was originated from an entity, and that is has not changed during the network traversing.
Encryption Device	device (hardware or software) which performs cryptographic operations.
Hand−off	see *hand−over*.
Hand−over	for a mobile device, the change from one access point coverage area to another. Also referred to as *hand−off*.
Hardcode	In programming, the fact of forcing some value or result.
Internet Service Provider	organization which provides internet services (World Wide Web, e−mail, etc.) to the users.
Message Digest	fixed−length output of a one−way hash function.
Patch	Piece of code added (not always in an elegant fashion) to an implementation in order to fix a bug.
Pre−shared Key	key shared by communicating entities which has been previously distributed to those entities by other trusted entity.
Re−keying	periodic refreshment or substitution of the key performed by the key−management protocol.

Replay Attack	security attack where an attacker captures a message and communicates later that message to an entity.
Strong Authentication	peer−to−peer authentication of both users to each other. Also referred to as Mutual Authentication.

Figures and Tables Index

FIGURES

TABLES

References

[1] J. Ryan. Voice over IP (VoIP). The Applied Technologies Group white paper. 1998.

[2] H. Schulzrinne, S. Casner, R. Frederick, and V.Jacobson. RTP: A Transport Protocol for Real−Time Applications. RFC 1889, Internet Engineering Task Force, January 1996.

[3] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261, Internet Engineering Task Force, June 2002.

[4] J−O. Vatn. A roaming Architecture for IP based Mobile Telephony in WLAN environments. Proceedings of Mobility Roundtable 2003. http://www.hhs.se/cic/roundtable2003/papers/S13−Vatn.pdf. Stockholm, 2003.

[5] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Internet Engineering Task Force, December 1998.

[6] P. McCann. Mobile IPv6 Fast Handovers for 802.11 Networks. IETF draft <draft−mccann−mobileip−80211fh−01.txt>. October 2002.

[7] L. Dang, C. Jennings, and D. Kelly. Practical VoIP using VOCAL. O'Reilly & Associates Inc. ISBN 0−596−00078−2. July 2002.

[8] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327, Internet Engineering Task Force, April 1998.

[9] S. Glass, T. Hiller, S. Jacobs, and C.Perkins. Mobile IP Authentication, Authorization and Accounting Requirements. RFC 2977, Internet Engineering Task Force, October 2000.

[10] C. Perkins. IP Mobility Support for IPv4. RFC 3344, Internet Engineering Task Force, August 2002.

[11] J. Rosenberg and H. Schulzrinne. Session Initiation Protocol (SIP): Locating SIP Servers. RFC 3263, Internet Engineering Task Force, June 2002.

[12] C. Rigney, A. Rubens, W. Simpson, and S. Willens. Remote Authentication Dial In User Service (RADIUS). RFC 2138, Internet Engineering Task Force, April 1997.

[13] P. Calhoun, J. Loughney, E. Guttman, G. Zorn, and J. Arkko. Diameter Base Protocol. IETF draft <draft−ietf−aaa−diameter−16.txt>. December 2002. Work in progress.

[14] P. Calhoun, T. Johansson, C. Perkins. Diameter Mobile IPv4 Application. IETF draft <draft−ietf−aaa−diameter−mobileip−13.txt>. October 2002. Work in progress.

[15] M. Baugher, R. Blom, E. Carrara, D. McGrew, M. Naslund, K. Norrman, and D. Oran. The Secure Real Time Transport Protocol. IETF draft <draft−ietf−avt−srtp−05.txt>. June 2002. Work in progress.

[16] J. Franks, P. Hallan−Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart. HTTP Authentication: Basic and Digest Access Authentication. RFC 2617 Internet Engineering Task Force, June 1999.

[17] T. Dierks and C. Allen. The TLS Protocol Version 1.0. RFC 2617 Internet Engineering Task Force, January 1999.

[18] S. Farrell. Outlining Wireless Public Key Infrastructure. Baltimore Technologies. www.baltimore.com. February 2003.

[19] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 2617 Internet Engineering Task Force, April 2002.

[20] J. Kohl and C. Neuman. The Kerberos Network Authentication Service (V5). RFC 1510 Internet Engineering Task Force, September 1993.

[21] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 1510 Internet Engineering Task Force, November 1998.

[22] J. Arkko, V. Torvinen, G. Camarillo, A, Niemi, and T. Haukka. Security Mechanism Agreement for the Session Initiation Protocol (SIP). RFC 3329 Internet Engineering Task Force, January 2003.

[23] W. Simpson. PPP Challenge Handshake Authentication Protocol (CHAP). RFC 3329 Internet Engineering Task Force, August 1996.

[24] D. Eastlake. Domain Name System Security Extensions. RFC 3329 Internet Engineering Task Force, March 1999.

[25] J−O. Vatn. Establishing a Secure Mobile VoIP phone call. TSLab IMIT KTH, Stockholm. February 2003. Work in progress.

[26] D. Harkins and D. Carrel. The Internet Key Exchange (IKE). RFC 2409 Internet Engineering Task Force, November 1998.

[27] M. K. Ranganathan and L. Kilmartin. Investigations into the Impact of Key Exchange Mechanisms for Security Protocols in VoIP Networks. Communication and Signal Processing Research Unit, Department of Electronic Engineering, National University of Ireland. *Consulted February 2003.*

[28] S. Dusse, P. Hoffman, B. Ramsdell, L. Lundblade, and L. Repka. S/MIME Version 2 Message Specification. RFC 2311 Internet Engineering Task Force, March 1998.

[29] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. MIKEY: Multimedia Internet KEYing. IETF draft <draft−ietf−msec−mikey−06.txt>. February 2003. Work in progress.

[30] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. Key Management Extensions for SDP and RTSP. IETF draft <draft−ietf−mmusic−kmgmt−ext−07.txt>. February 2003. Work in Progress.

[31] SER: iptel.org SIP Express Router. www.iptel.org/ser. *Consulted February 2003.*

[32] E. Eliasson. MINISIP: SIP user agent software. TSLab IMIT KTH, Stockholm. February 2003. Work in progress.

[33] D. McGrew. A library for Secure RTP. http://srtp.sourceforge.net/srtp.html. July 2002. *Consulted February 2003.* Work in progress.

[34] Advanced Encryption Standard (AES), Federal Information Processing Standard Publications (FIPS PUBS) 197, November 2001.

[35] H. Krawczyk, M. Bellare, and R. Canetti. HMAC: Keyed−Hashing for Message Authentication. RFC 2104 Internet Engineering Task Force, February 1997.

[36] Security in SIP−Based Networks. Cisco Systems Inc. white paper. *Consulted February 2003.*

[37] B. Aboba and M. Beadles. The Network Access Identifier. RFC 2486 Internet Engineering Task Force, January 1999.

[38] M. Baugher. SDP Security Descriptions for Media Streams. IETF draft <draft− baugher−mmusic− sdpmediasec−00.txt>. September 2002. Work in progress.

[39] T. Kanter, C. Olrog, and G. Maguire. VoIP over Wireless for Mobile Multimedia Applications. http://psi.verkstad.net/Publications/pcc99/VWMMA_PCC.PDF. *Consulted February 2003.*

[40] L. Blunk and J. Vollbrecht. PPP Extensible Authentication Protocol (EAP). RFC 2284 Internet Engineering Task Force, March 1998.

[41] W. Stallings. Network Security Essentials, Applications and Standards. Second Edition. Prentice Hall. ISBN 0−13−120271−5. 2003.

[42] H. M. Deitel and P. J. Deitel. C++, How to Program. Second Edition. Prentice Hall. ISBN 0− 13−528910−6. 1998.

[43] D. Gollmann. Computer Security. First Edition. John Wiley & Sons Ltd. ISBN 0−471− 97844−2. 1999.

[44] A. Mishra and W. A. Arbaugh. An Initial Security Analysis of the IEEE 802.1X Standard. Department of Computer Science, University of Maryland. February 2002.

[45] A. James. Using IEEE 802.1X to Enhance Network Security. Foundry Networks white paper. October 2002.

[46] J. Savard. The Advanced Encryption Standard (Rijndael). http://home.ecn.ab.ca/~jsavard/crypto/co040401.htm. *Consulted April 2003.*

[47] H. Lipmaa, P. Rogaway, and D. Wagner. Comments to NIST concerning AES Modes of Operations: CTR−Mode Encryption. Helsinki University of Technology. www.tcs.hut.fi/~helger/ papers/lrw00/html/. *Consulted April 2003.*

[48] Microelectronics Laboratory − Crypto Group, Université Catholique de Louvain (UCL), Louvain, Belgium. Some Figures about AES Candidates Performances. www.dice.ucl.ac.be/crypto/CAESAR/performances.html. *Consulted April 2003.*

[49] ANSI X3.106. American National Standard for Information Systems−Data Link Encryption. American National Standards Institute.

[50] B−J. Koops. Public−Key Infrastructures. http://rechten.kub.nl/koops/pki.htm. *Consulted May 2003.*

[51] U. Maurer. Modelling a Public−Key Infrastructure. Department of Computer Science. Swiss

Federal Institute of Technology (ETH), Zurich. 1996.

[52] D. Newman. PKI: Build, Buy or Bust? Network World. http://www.nwfusion.com/research/2001/1210feat.html. October 2001. *Consulted May 2003.*

[53] D. Maughan, M. Schertler, M. Schneider, and J. Turner. Internet Security Association and Key Management Protocol. RFC 2408 Internet Engineering Task Force, January 1999.

[54] C. Kaufman, R.Perlman, and M. Speciner. Network Security, Private Communication in a Public World. Second Edition. Prentice Hall PTR. ISBN 0−13−046019−2.

[55] Hifn HIPP Security Processor 7815/7855. Hifn. http://www.hifn.com/docs/HIPP−7815−7855.pdf. *Consulted June 2003.*