University "Alexandru Ioan Cuza" of Iaşi
Faculty of Computer Science

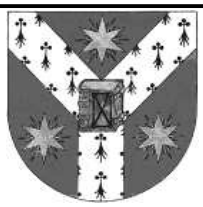
TECHNICAL REPORT

Securing the Media Stream Inside VoIP SIP Based Sessions

Emanuel Onica

TR 09-01, October 2009

ISSN 1224-9327



Universitatea "Alexandru Ioan Cuza" Iaşi
Facultatea de Informatică
Str. Berthelot 16, 6600-Iaşi, Romania
Tel. +40-32-201090, email: bibl@infoiasi.ro

Securing the Media Stream Inside VoIP SIP Based Sessions

Emanuel Onica
"Al. I. Cuza" University, Iaşi
Faculty of Computer Science
eonica@info.uaic.ro

Abstract

One of the main issues encountered in the development of VoIP applications and infrastructures relates with ensuring the security of the communication channel between peers. The discussion on this subject can be split in two principal directions: the signaling path security and the media path security. In this document we focus on the latter by overviewing the current most important available mechanisms and their way of interworking for architectures based on one of the most deployed standards in VoIP communication – Session Initiaton Protocol (SIP).

Table of Contents

1. Introduction

Since the standardization of signaling protocols such as SIP [1] and H.323 [2], the media-stream packet based communication over IP, consisting especially in VoIP services, but also video transmissions, is constantly gaining field against PSTN (Public Switched Telephony Network - the "classic" telephony system). Along with this growing there is also observed a growth in the security issues that the various communication services bring up. These issues range from media path related ones, regarding especially the protection of the media stream against interception, to a variety of signaling path related problems depending on the communication service and the network architecture which is determined by the signaling protocol.

In this document we will focus on the first category mentioned, namely the one regarding the media stream protection through its encryption. In the VoIP area of communications, in order to ensure even the basic session between two endpoints, a signaling protocol is needed. At the moment there are a variety of signaling protocols used in VoIP communications, two of the most encountered in applications being SIP [1] developed by IETF (Internet Engineering Task Force), and H.323 [2] developed by ITU (International Telecommunication Union). We will particularly refer in our discussion to SIP based sessions. The signaling protocol in VoIP communications, through its role of session management, is usually viewed as a central point of a specific VoIP architecture which includes types of endpoints and defines various specific services and extensions. As suggested above, there are several securing levels regarding a VoIP solution some of them being in direct relation with the used signaling protocol. However, our main topic of discussion targets the securing of the media stream which is situated at another layer. In most of the VoIP software regardless of the signaling solution used, the media stream is carried by RTP [3] as a protocol used for transport situated above UDP. Nevertheless, this separation between transport and signaling layer protocols[1] doesn't reflect completely also in the securing process of the protocol used for transport (RTP). The methods of securing the media stream carried by RTP vary from one signaling protocol to another and often even for the same signaling protocol. This is mostly concerning the key exchange and management solutions applied in the securing process. The effective encryption of the RTP stream is usually done through the dedicated mechanism defined in the SRTP standard (Secure RTP) [4], independent of the signaling protocol used. Our choice to focus on SIP based communications regarding the signaling layer is based on its gaining in the last years over H.323 and other signaling protocols, as specified by various statistics[2].

[1] The signaling and transport layer distinction is done here and in the following sections with respect to the VoIP architecture functionality, not to the OSI layer model or TCP/IP layer model

[2] From the 39 softphone client solutions listed in a comparison chart at [5], 28 support SIP, and from the 12 server solutions mentioned in the same source, 10 are SIP compliant.

At the present moment there is not established a strict standardization concerning key management as part of media securing in SIP based VoIP solutions, although an informational RFC [6] has been approved enumerating several dedicated protocols and describing some general requirements involved by the topic. One of our main goals in the current document is to describe the most important key management solutions, analyze them, and detail possible advantages and disadvantages of choosing one of them for a SIP VoIP based application. We do not intend to completely exhaust the topic given the number of possible solutions and the variations which might appear in different cases of practical implementations. We mainly try to offer a basic and most of all clear overview of the subject, which might be useful also for somebody interested in the theoretical aspects and also for a developer intending to add additional security features to a VoIP application.

In the next chapter we will present a brief overview of the main protocols involved by a VoIP solution. This is needed in order to offer a basic level of understanding of the functionality supposed to be secured and the flow inside a session. We will cover both the signaling and the transport layers by describing parts of the SIP and respectively RTP standards. As mentioned before a key management solution may operate in relation with both layers. However we will not get into detail in deeper SIP functionality besides basic call signaling and the main information about the relation with the media path, our topic of discussion concerning mainly the securing of the transport protocol.

The protection of RTP will be described in more detail in the third chapter of the document. We will present here the main features of SRTP [4], focusing on the parts that will be referred also by the key management solutions presented in the next sections. Also in this chapter, we will try a more detailed overview on the already briefly presented key management issue.

The fourth chapter covers probably the most important topic related to media stream securing. Here we try to resume the main functionality of several key management protocols and to discuss their features and eventual problems. Though we not intend to reproduce all the aspects described in the published versions of the protocols, we hope to give enough information in order to offer a clearer synthesized view on each solution that might help an eventual developer to apply it in a practical project. The main addition to the original individual published standards or drafts will consist in detailing, commenting and pointing out aspects like flaws, distinct features and in some cases even a verification of parts of the protocols. In our overview we will consider mainly the current solutions that are dedicated to provide the required keys for the encryption of the media stream without making use of any well-known security protocols like IPsec or TLS [7]. We consider that is more important to discuss these partly because their understanding is more difficult, and partly because these are dedicated solutions for the specific topic. Therefore our main point of interest will be centered on MIKEY [8], DTLS-SRTP [9] and ZRTP [10] as key management mechanisms for securing media stream inside VoIP SIP based sessions. However we will not take out from consideration other possible solutions briefly discussed in the final section of the chapter.

The final chapter tries to deliver some conclusions on the main subject of this document. Also here we enumerate some of the other problems concerning security in a SIP based VoIP solution or scenario. Though not related to our main topic we found it useful to offer a short description of some often encountered issues which might interest a

reader concerned also in general security aspects of SIP applications. Finally, we briefly discuss an use case as example of a secure media stream solution and we conclude with future development directions regarding the intercompatibility topic between different key exchange implementations and group communication.

2. Basic Description of VoIP SIP Based Sessions

As briefly described in our introduction the main protocols typically involved in a VoIP SIP based session are SIP – Session Initiation Protocol [1] at the signaling layer, and RTP – Real-time Transport Protocol [3] at the transport layer. From the user's point of view the main feature offered by SIP or any other signaling layer protocol used in a VoIP session is providing a higher level of abstraction for features like using an addressing format that does not require remembering the IP addresses, the possibility of registration to one server and consequently making a call with the same ID from different computers, the feature of forwarding calls and others. All these and other services are part of a so called signaling layer which may be integrated in the OSI model inside the session layer for the basic operations, and inside the presentation or the application layer for the more advanced ones. Behind these functionalities lies a protocol specification which usually has a high degree of complexity expressed also through its annexes and related protocols (in case of H.323 an "umbrella" specification is defined including more than 20 different standards). This signaling protocol acting as a central standard for a model of communication, usually defines a VoIP architecture comprising different types of communicating entities and their behavior. In our case this protocol is SIP – Session Initiation Protocol. We will briefly describe the main usage of SIP in a VoIP environment in the first part of this chapter, focusing on basic functionalities in order to establish a context to discuss our main topic of this document.

The central topic presented in our work, key management solutions used in securing the media stream, is directly related to the transport level of a VoIP session. The transport level actually can be said that identifies itself with the media stream, or more exactly this stream of audio data is carried by a transport protocol which usually, and also discussed in our case is RTP. Securing RTP payload data (the audio encoded data) involves encryption of the RTP packets. The protocol which covers these actions is SRTP which will be presented in the next chapter. Before this, we will try to describe, also briefly, in the last part of this section, the subject of the encryption – RTP.

2.1. Overview of the Signaling Layer – SIP

SIP (Session Initiation Protocol) is essentially a signaling protocol developed to set up, modify and tear down multimedia sessions over the Internet, having also presence and instant messaging delivery functionalities as described in [11]. SIP was published as a standards track IETF RFC for the first time in April 1999, being revised in the current version in June 2002 [1].

In this section we will illustrate the basic SIP signaling flow needed to establish a call between two parties directly and through a proxy. SIP is also used, regarding signaling, for more complex call flows involving forking or additional operations like registration, but for our purpose of describing the relation with the lower level transport protocol and further the potential interaction with various key management solutions, we consider the following examples to be comprehensive enough.

2.1.1. Basic Direct SIP Call Establishment

The following diagram presents the simplest case of SIP functionality concerning a call establishment between two peers:

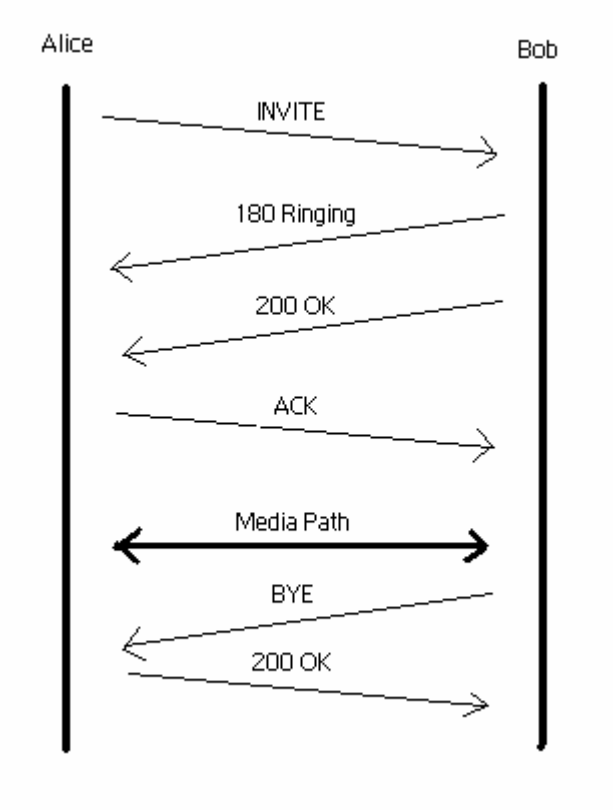


Figure 1

The peers, Bob and Alice, can be represented in an actual call by any SIP compliant endpoints connected to an IP network and knowing each other's addresses. The messages, as can be seen, mostly resemble HTTP [12] as a protocol, concerning the request/response call flow[3].

The first message sent in order to start a SIP call is the INVITE message. A common message structure for an INVITE request, as the one present in the above example is the next one:

```
INVITE sip:bob@bobdomain.org SIP/2.0
Via: SIP/2.0/UDP home.alicedomain.org:5060;branch=z9hG4bKcf782
Max-Forwards: 70
To: Bob <sip:Bob@bobdomain.org>
From: Alice <sip:Alice@alicedomain.org>;tag=99076
Call-ID: 222333444@home.alicedomain.org
CSeq: 1 INVITE
Subject: Test Call
Contact: <sip:Alice@alicedomain.org>
Content-Type: application/sdp
Content-Length: 124

…… SDP content ……
```

As it can be seen SIP messages are text based, having a structure that partly resembles SMTP [13] messages. This structure has common header fields that we will detail in the next paragraph, many of them being present also in other messages.

The request above contains on the first line the SIP URI where the INVITE is sent and the protocol version. The second line is dedicated to the *Via* field. This is a field added to the message by every host which generates or is proxying the message including its own address. Also present here is a branch number which identifies a series of messages called a *transaction*. A transaction can usually be defined as the group of messages which are in the request/response relation (with some exceptions), the branch number being therefore used to match responses with the correct requests. The third line contains the *Max-Forwards* field which represents the number of SIP servers that the message is allowed to pass, being decremented each time. In a way it is similar to the TTL field in the ICMP packets, but its usage in SIP is primarily to avoid loops of the same message. The next two lines include the *To* and *From* fields. These represent the source and destination addresses as *AoR* – Address of Record. An *AoR* logically identifies a user and can be different than its current effective device related URI. These address fields are followed by random tags added by the source and destination peers (for INVITE there is of course only one such tag because the message flow didn't yet reach the destination peer). The next line contains the *Call-ID* field which is a unique identifier of the SIP session. This is followed by *CSeq*, which is an integer number that is incremented with every new request that is sent. The *Subject* field of the header is optional simply containing a text fragment which might be used to eventually display various description data about the call. The *Contact* field holds the actual device URI of the request sender. The last two lines of the header describe the body of the message. In this case this body is present (though we skipped detailing it for the moment), is a SDP

[3] The messages lacking a numeric code are considered requests, and the one beginning with a numeric code are considered responses (like the situation found in HTTP)

body and is 124 bytes long. SDP (Session Description Protocol) [14] is a companion protocol for SIP used to describe various session parameters like codec information, port number or even security related parameters. We will detail its main features later in this chapter.

Like stated above, the fields which can be found in the INVITE request content can also be found entirely or partially in other SIP requests and also in the responses. This fact can be observed in the *180 Ringing* informational response which is sent as reply for the INVITE:

```
SIP/2.0 180 Ringing
Via: SIP/2.0/UDP home.alicedomain.org:5060;branch=z9hG4bKcf782
;received=211.101.120.45
To: Bob <sip:Bob@bobdomain.org>;tag=88659
From: Alice <sip:Alice@alicedomain.org>;tag=99076
Call-ID: 222333444@home.alicedomain.org
CSeq: 1 INVITE
Contact: <sip:Bob@bobdomain.org>
Content-Length: 0
```

This response is sent by Bob's endpoint in order to inform Alice that the INVITE has been received and the destination endpoint was announced but did not yet answer the call. As it can be seen, the most of the initial fields are also present in the response too. The order contents of the *To* and *From* fields remain the same for the duration of an entire dialog, even if the response is sent by Bob to Alice. A dialog is defined by a sequence of SIP requests and responses which have the same tags and call ID.

Following the message flow presented in Figure 1, when Bob finally answers the call a *200 OK* response is sent to Alice. The final step of establishing the SIP session is an ACK acknowledge, which as message type is a request, sent by Alice to Bob.

The depicted sequence of messages has as main purpose the preparation to start the media stream, providing capabilities exchange. Further, the media stream (whose flow takes place at the media path figured in the diagram) is carried by RTP (Real-time Transport Protocol) [3] or in a secure case scenario by SRTP [4]. In order to secure the media stream, SRTP needs a cryptographic context which mainly implies a master key that needs to be available at both of the peers. This is the moment where the key establishment process takes place (at least for the first time because rekeying is also implied by SRTP at certain intervals), and according to the mechanism used this process may interfere with the SIP signaling or with the media path.

The basic SIP session presented above is closed after the media stream transmission ends. This takes place through another request/response sequence formed by a BYE request and a *200 OK* response. Due to the fact that the SIP messages content is usually formed by a series of fields similar to the first two messages described, we will not get into more detail with respect to the message headers, unless these will be directly related with the process of securing the media stream.

2.1.2. Basic SIP Call Establishment Through a Proxy

A second, and probably the most frequent encountered case of SIP signaling in establishing calls is the one when the peers do not do it directly, not knowing each other's effective SIP device URI, and consequently making use of at least one SIP proxy.

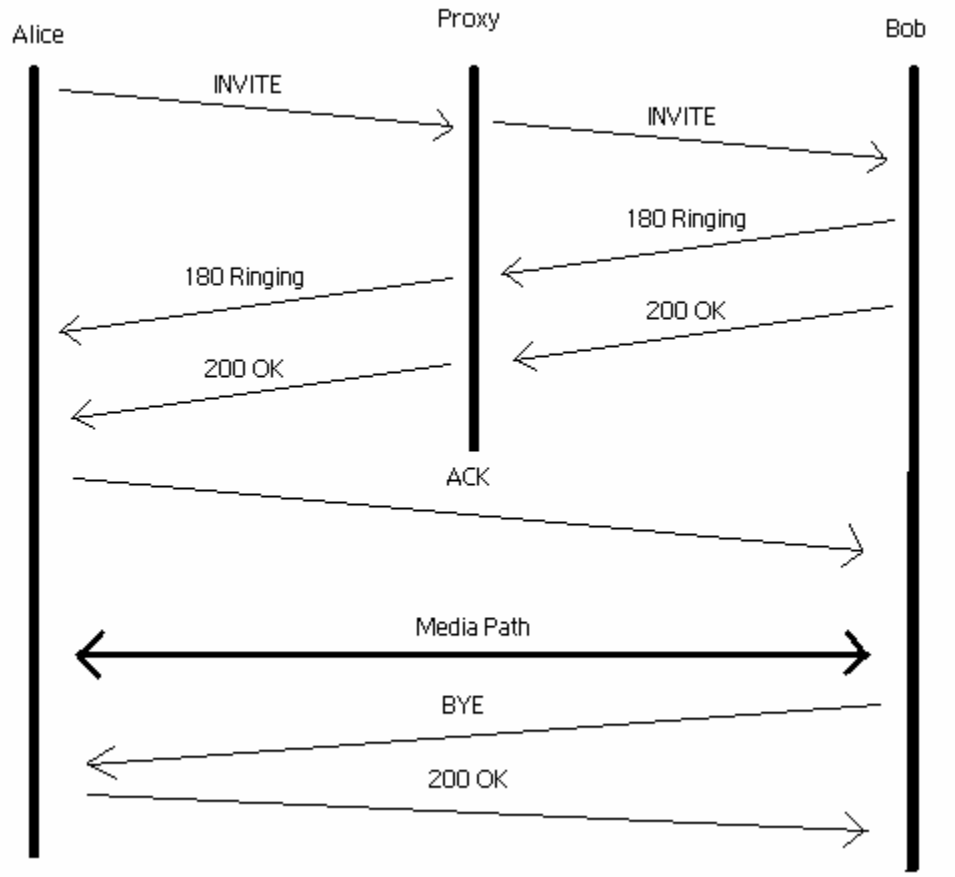


Figure 2

In the above diagram this case is pictured in its simplest form. The proxy is used to facilitate the SIP session establishment between users which do not know the exact URI of each other, therefore making use of the intermediate point lookup services (in a similar fashion with a DNS query). The message flow displayed in the above diagram resembles the one in Figure 1 in its second part after the first INVITE request and responses. This first transaction makes possible for the two peers to find out the exact URI of each other, these being added in the forwarded messages, therefore passing through the proxy isn't anymore mandatory.

2.1.3. Basic SDP Description

SDP – Source Description Protocol [14] is, as the name actually implies, a protocol used in description of media sessions. Actually SDP inside SIP communication is more like a specification of session parameters, not having an active role as a stand alone protocol (the description provided by SDP is carried inside SIP message bodies). One of main uses of SDP is in the exchange of endpoint capabilities. This is related especially with codec agreement and is performed through the INVITE request and the responses when initiating a call, or when a change of the media format is desired. However SDP bodies may be used in other purposes too. One of them is key transport in a key exchange scenario where the SIP messages are secured through an external non-dedicated solution like TLS or IPsec. A sample of a typical SDP message body is the one we omitted in our first example from the INVITE message. This may look like following:

```
v=0
o=Alice 3220855426 3220855426 IN IP4 alicedomain.org
s=Phone Call
c=IN IP4 211.101.120.45
t=0 0
m=audio 49170 RTP/AVP 0
a=rtpmap:0 PCMU/8000
```

The structure of the SDP body is a series of lines having the format: *field= parameters* where the field is defined by a single letter and the parameters are field specific and separated by a space. There is a series of mandatory fields which must be included in every SDP message body. The complete set of fields that are standard defined for SIP usage is the next:

v – protocol version number (mandatory)
o – owner and session identifier (mandatory)
s – session name (mandatory)
i – session information (optional)
u – URI (optional)
e – email address (optional)
p – phone number (optional)
c – connection information (mandatory)
b – bandwidth information (optional)
t – time when session starts and stops (mandatory)
r – repeat times (optional)
z – time zone corrections (optional)
k – encryption key (optional)
m – media information (optional)
a – media attributes (optional)

Details about the parameter list format is available in the SDP RFC [14].

2.2. Overview of the Transport Layer – RTP

According to the standard's description [3] RTP – Real-time Transport Protocol is a protocol which "provides end-to-end delivery services for data with real-time characteristics, such as interactive audio and video". At the current moment RTP is the principal mean of transportation for multimedia streams inside VoIP architectural models like SIP or H.323. RTP implementations do not usually follow some strict guidelines, the standard even suggesting at some points to adapt the specifications according to the use case context. One such adaptation, more exactly a profile defined for RTP is SRTP – Secure Real-time Transport Protocol [4], which we will describe in the next chapter.

RTP runs typically over UDP and though it does not ensure quality of service over the IP network, provides the means of detecting several issues important in the context of multimedia transmissions like packet loss, out of sequence packet arrival and variable delay in packet arrival. The protocol does not handle these problems, this being the task of higher level mechanisms like the media stream codec or the VoIP application. Besides the information provided by the RTP header, in order to aid in such purposes and also for other problems solving, a companion monitoring protocol RTCP (Real-time Transport Control Protocol) is defined in the same standard [3]. RTCP flow which consists in several types of predefined packets has a specific bandwidth allocation, the packet sending occurring at intervals dependent on this allocation. Though the RTCP companion protocol is concerned only with monitoring functions and most of the attacks targeted at the media path are intended to intercept the conversation in the stream, the RTCP packets are also subject to encryption along with the RTP ones.

Important for our discussion would be the structure of the RTP packet which is further encrypted by SRTP. The RTP packet is divided in a header and a payload including the media stream (audio or video data). The information included in the header contains among the others[4]:

- a payload type 7 bit field defining the codec used
- a 16 bit sequence number incremented for each RTP packet sent
- a 32 bit timestamp field indicating when the media stream data was sampled
- a 32 bit SSRC identifier discussed below
- a 4 bit CSRC count indicating the number of CSRC fields in the packet
- 0 to 15 CSRC identifier 32 bit fields discussed below

A RTP session represents the association of participants communicating through RTP. One participant can be involved in more than one RTP session simultaneously. Also between two participants can exist more than one session at the same time – a typical example being one for the audio stream and one for the video stream. The RTP sessions can be distinguished through the IP *address:port* pairs.

A SSRC identifier (synchronization source) represents an RTP data stream source – a participant involved in an RTP session transmitting RTP packets. The SSRC identifier is randomly chosen and is unique for the RTP session (if one RTP sender chooses an existing SSRC for the session this will be chosen again until distinct). This indicator is

[4] A detailed view of the header structure is available in Appendix 2

used by the destination for grouping the packets originated by the same source after their sequence number in order to render the stream.

A CSRC identifier (contributing source) represents also an RTP data stream source which contributed to the current media stream payload state. More exactly a CSRC can identify the endpoints from which an RTP stream results. One example would be a mixer defined below, present in case of a conference. The CSRC value is added to the CSRC list when the stream passes this type of endpoint.

The entities taking part to a RTP based transmission can be classified with respect to this type of transmission into:

- terminals: simple consumer or generators of RTP stream; the typical example consists in two peers involved in a simple call (senders are considered SSRC – synchronization sources, with respect to the RTP session)
- mixers: intermediary entities receiving RTP data stream from more sources, synchronizing and combining the different streams and sending it further to destinations; typically appears in a conference; due to the fact that the mixer modifies the media stream contained by the RTP packets the new SSRC of the packets sent forward will be the SSRC of the mixer and the SSRC from each combined streams will be added to the CSRC list as contributing sources
- translators: intermediary entities receiving RTP data stream and forwarding it without combining more streams but optionally performing other operations like re-encoding the media stream, removing or adding encryption or others; one typical scenario using this might be passing through a firewall which doesn't allow for some reason the access of a certain type of RTP packets, these needing some conversion before going through and reconversion after, performed by two translators; the translators do not modify the SSRC of the packets even if the data could be re-encoded and consequently the timestamp and sequence number of the RTP packets could be modified
- monitors: auxiliary entities which do not have active RTP transmission role but are defined by the standard for optional usage regarding RTP traffic monitoring

Note, that the types of entities defined are not by any mean directly related with SIP entities, like a SIP user agent or a SIP proxy, and are only defined with respect to the RTP flow. However a SIP proxy for example may have a role of RTP mixer, a SIP user agent (end peer involved in a conversation) typically has the role of an RTP terminal due to its operation nature, and so on. We detailed this classification of RTP entities mostly in order to clarify the meaning of SSRC and CSRC identifiers which are related with securing operations discussed in the next sections.

Like specified before the RTCP companion protocol flow consists in a series of packets with a predefined structure. The main ones are concerned with traffic reports: RR – receiver reports and SR – sender reports, and identification packets: SDES – source description. RTCP also define BYE packets indicating the departure of a member of the media session and APP for application specific information.

3. Securing the Media Stream

The effective securing of the stream transmitted through RTP is achieved by encrypting the RTP [3] packet payloads containing the encoded audio or video data. The mechanism dedicated for this task is defined in the SRTP – Secure Real-time Transmission Protocol standard [4]. In addition of protecting the media stream by encryption, SRTP also specifies an integrity control that can be optionally added to the RTP packets consisting in a Message Authentication Code (MAC). We try to overview the SRTP mechanisms in the first section of this chapter along with a brief analysis on why SRTP is preferred to be used in practice over well known solutions like IPsec.

The process of using SRTP in communication implies using an external key management mechanism. This mechanism, the central topic of our document, can relate, as stated also in the Introduction, with the protocol used at the signaling layer (therefore our need for a briefly SIP description). In our last section of this chapter we will detail this issue.

3.1. Media Stream Encryption – SRTP

SRTP – Secure Real-time Transport Protocol is “a profile of the Real-time Transport Protocol (RTP), which can provide confidentiality, message authentication, and replay protection to the RTP traffic and to the control traffic for RTP, RTCP” as defined in the IETF RFC [4].

In order to perform the mentioned functionalities the SRTP standard provides a framework for encryption and authentication, one of the goals being a low bandwidth cost needed to preserve the low latency of the media streams. This framework is placed as a filter residing between RTP and the lower transport layer (typically UDP). More exactly the RTP packets are processed by SRTP before sending them to the network and just after receiving on the other side:

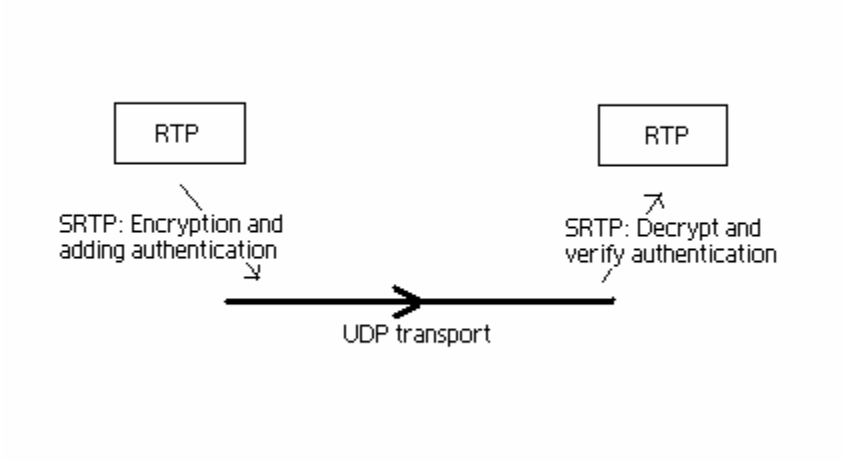


Figure 3

Following the SRTP framework processing, the RTP packet is transformed basically as depicted in the next figure:

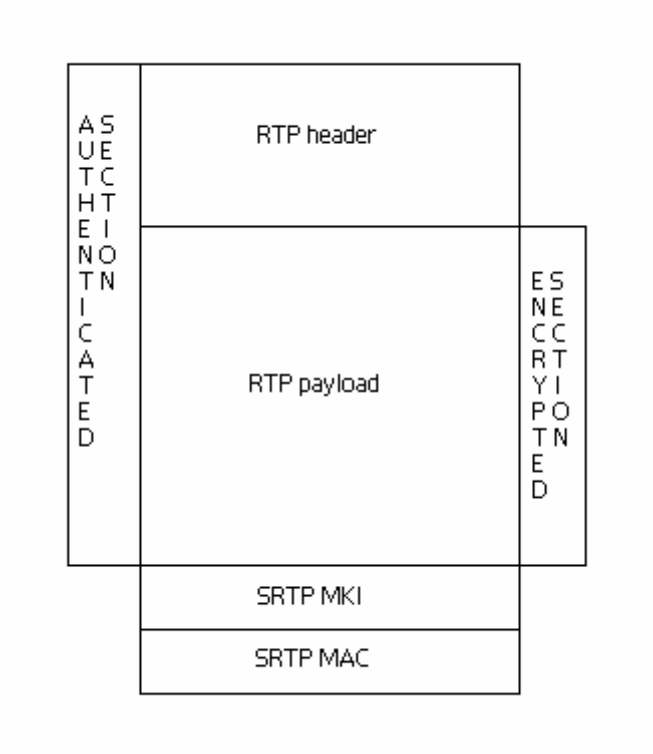


Figure 4

The SRTP MKI – Master Key Identifier is an optional value used by the receiver to choose which master key to use in decrypting the packet (useful especially in the context of more than one RTP session simultaneously). This master key is managed by a key management solution external to the SRTP standard, and is used in the derivation of session keys which are the ones effectively used in the packet encryption. Also the MKI is necessary in rekeying phases, after which delayed packets might be received still encrypted with old keys identifiable through the MKI.

The SRTP MAC – Media Authentication Code contains the value obtained after authentication of the RTP header and the encrypted RTP payload. As stated in the standard [4] the MAC provides also indirectly replay protection due to authentication of the sequence number as part of the RTP header (the sequence number being different for each packet from the previous one due to incrementing).

The RTP companion RTCP packets are usually formed in the protocol flow as compound packets from more than one simple RTCP predefined packet. Their encryption (the SRTCP packets), covers in this case all of the unencrypted packet content except the first eight bytes containing header information like the protocol version, the number of report blocks in the compound packet, the packet type, length and the SSRC of the sender. Also like the case of RTP, the authentication is performed on the entire packet content. Besides the MKI and MAC two other fields are added to the SRTCP packet. One is the E-flag which indicates if the current SRTCP packet is encrypted or not. This is included because not necessarilly all the RTCP traffic should be encrypted. According to the RTP/RTCP RFC [3] a tipically compound packet can be split prior to applying the encryption filter in order to send only some of the information crypted like the SDES blocks which identify a session member. The main reason for this separation is that other

packet types like the traffic reports may prove useful for the intermediate hosts. In addition to the mentioned E-flag in the SRTCP packets is included a specific index separate from one used in the SRTP packets described in the next section.

The cause of omission of the packet header from the encryption process in both cases of SRTP and SRTCP lies also in the need of processing at the intermediate hosts like a mixer for example, this not being possible otherwise due to the end-to-end nature of SRTP securing.

3.1.1. The SRTP Cryptographic Context

Each RTP stream has a cryptographic context associated by the SRTP framework. The external key management mechanism provides the master key for this cryptographic context, from which session keys are derived. Also the external key management mechanism should provide other parameters needed by the cryptographic context.

The cryptographic context associated with a RTP stream contains two types of parameters: transform-dependent and transform-independent.

The transform-dependent category includes parameters particulary related with an application of the encryption, authentication and key derivation processes, like the mentioned session keys for example or the Initialization Vector. We will detail aspects about these operations in the next sections.

The transform-independent parameters include among others:

- the master key (provided by key management)
- a counter of the number of packets processed with the master key (maintained by SRTP implementation) which is used in order to renew the master key when a certain limit is reached
- a random master salt used in derivation of session keys (generated by SRTP implementation)
- the session keys derivation rate (provided by key management)
- a rollover counter ROC which is used to record how many times the sequence number in the RTP packets is reset back to zero; this reset takes place after the number reaches 65535 the maximum value for the sequence number 16 bit field; the ROC is used in computing an index number for the RTP packets as:

$$i = 2^{16} * ROC + \text{sequence number}$$

  (maintained by SRTP implementation)
- the highest received RTP sequence number *s_1* used to determine the ROC value (maintained by SRTP implementation)
- a replay list containing the recently received packets holding the indexes of a certain amount of the last received and authenticated RTP packets with the purpose of replay detection (maintained by SRTP implementation)
- the MKI indicator which defines if an MKI is used or not and consequently the length of the MKI and its value in case it is used (provided by key management)
- the length of the encryption and authentication session keys (provided by key management – though session keys are derived by SRTP implementation)
- identifier of the encryption algorithm used (provided by key management; possible options will be detailed in the next sections)
- identifier of the authentication algorithm used (provided by key management, possible options are described in the next sections)

A cryptographic context is uniquely identified by an id formed from:

context id = (SSRC of the RTP stream, destination address, destination port)

The SRTP and associated SRTCP usually share the crypto context excepting parameters directly related with packet flow these being different ROC, replay list, and master key processed packets counter. We will not get into much detail about the secured SRTCP channel in our following description. Though it is important, we chose to focus on the actual RTP media stream securing, the mechanism for the associated control packets being partly similar, some points where the methods interfere being mentioned in our presentation.

3.1.2. SRTP Packet Processing

The SRTP packet processing follows a series of steps for the sender and receiver.

In case of the sender, in the usual operation conditions, these steps for a RTP session, as specified in the SRTP RFC [4], are:

- the SRTP implementation selects the appropriate cryptographic context to use based on the context id triplet obtained from the RTP packet to be encrypted
- the index of the packet is computed mainly based on the described formula in the previous section; additional issues regarding this operation that might occur are detailed in section 3.3.1 of the standard [4]
- the corresponding master key and also the master salt, initially provided by key management, are selected based on the current MKI; there is also an alternate mechanism to MKI defined in section 8.1.1 of [4] provided for simple scenarios where the MKI added field to the SRTP encrypted packet is undesirable (like low bandwidth transmission mediums)
- the session encryption and authentication keys and salt are determined based on the master key, master salt, packet index and derivation function
- the RTP payload is encrypted using the determined encryption session key and the algorithm indicated by the cryptographic context
- the MKI is appended to the packet
- a MAC is computed over the RTP packet (without the MKI field) and appended; the computation is done using the determined authentication key, ROC and the algorithm corresponding to the cryptographic context
- the ROC is updated if necessary according to section 3.3.1 of [4]

In case of the receiver, for the packet decryption, the steps are quite similar with the ones above (of course decryption and authentication tag verification are performed instead). One difference appears at computing the packet index, an initial approximation being done regarding the ROC needed in computing the value, this being caused by the unreliable nature of the underlying UDP protocol which may deliver the packets out of order. The mechanism is described in detail in section 3.3.1 of [4]. Decrypting the packet is done after verifying the MAC tag in order to authenticate it. Also before authentication

a replay check is performed checking the packet index against the replay list maintained in the cryptographic context (the mechanism is detailed in section 3.3.2 of [4]).

3.1.3. Cryptographic Algorithms Used by SRTP

In order to encrypt and compute authentication tags, the master key and salt provided by the key management mechanism must be derived to obtain session keys. Key derivation occurs at least once at the start of the session. According to the key derivation rate or to a stated limit of 2^{48} packets encrypted based on the same master key, rekeying can occur during a session (this implies exchange of new master key and salt). A pseudorandom function $PRF_{n,m}(k,x)$[5] is used to obtain the derived material. For k the provided master key will be used. In case of SRTP m (the length of x) is specified to be at least 128 bits long. x is obtained according to the next formula:

$$x = (\text{label} \,||\, (\text{packet index DIV key derivation rate})) \text{ XOR master salt}$$

where || defines concatenation, *label* is a predefined value, based on it different types of keys being derived (encryption, authentication or salt) and *DIV* operation is defined as normal division rounded down, with the convention that division by zero results in zero and the result of the division will be prefixed with the necessary amount of zeros to obtain the bit length of the packet index. The used labels and other details are defined in section 4.3.1. of the standard. The PRF used function is based on AES algorithm and is defined in section 4.3.3.

The RTP stream effective encryption is defined in section 4.1. of [4]. The encryption algorithm used is AES in stream-cipher mode. The algorithm is used in a chain to produce a stream of keys used for encryption. Quoting from the standard: "The process of encrypting a packet consists of generating the keystream segment corresponding to the packet, and then bitwise exclusive-oring that keystream segment onto the payload of the RTP packet to produce the Encrypted Portion of the SRTP packet." In order to obtain the keystream two modes of running AES are defined: counter mode and f8-mode.

A keystream obtained by AES in counter mode is essentially built from a concatenation of 128 bit output blocks of the AES cipher obtained as result of applying the AES encryption using the obtained session encryption keys on an initialization vector obtained from the next formula:

$$IV_0 = (\text{session salt key} * 2^{16}) \text{ XOR } (\text{packet SSRC} * 2^{64}) \text{ XOR } (\text{packet index} * 2^{16})$$
$$IV_i = IV_0 + i \bmod 2^{128}$$

The initialization vector will "restart" when the next RTP packet becomes the subject of encryption. The basic mechanism is depicted in the figure below:

[5] A definition of a pseudorandom function $PRF_{n,m}(k,x)$ would be a function that for a random key k, given m bit x provides an n bit string undistinguishable from a random string

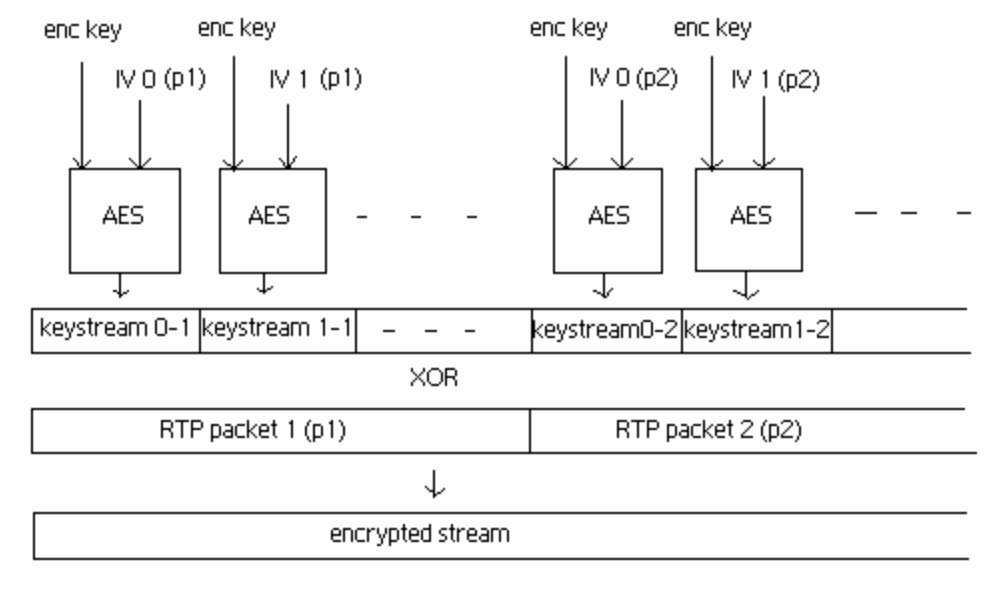


Figure 5

More details about AES in counter mode are specified in the SRTP RFC [4] section 4.1.1.

The f8-mode specified for AES is dedicated for Universal Mobile Telecommunications System data as 3G networks standardized by the 3GPP consortium. The mechanism, including a representative figure depicting it, is specified in section 4.1.2 of the standard. The main difference between f8 and counter mode consists in combining the initialization vector element for every keystream piece with the keystream piece obtained before.

The message authentication inside SRTP described in section 4.2. of [4], consists in simply applying the HMAC-SHA1 [15] keyed hash function over the RTP message header and encrypted payload using the derived session authentication key. The sender truncates the result obtained after this operation to the first 4 bytes in order to minimize the cryptographic overhead and appends it to the packet. The receiver ultimately authenticates the message by computing the truncated result in the same way and comparing with the MAC found at the message end.

3.1.4. SRTP Considerations

An alternate method to SRTP which may be used in order to secure the RTP media stream is IPsec (Internet Protocol Security). IPsec is a framework comprised of several security standards, the central one defining the architecture being [16].

The three mechanisms effectively used in securing are:

- ESP [17] – Encapsulating Security Payload – for encryption and possible authentication
- AH [18] – Authentication Header – only for authentication

- IKE [19] – Internet Key Exchange – for key and security parameters negotiation needed by the previous mechanisms

The first issue needed to be mentioned is that IPsec provides security directly at the network layer, and not end-to-end at the application level. More exactly the IPsec implementations often rely on the operating system kernel from each host passed in order to handle network traffic protection, the packets being chosen for securing based on an IPsec policy. This is part of a Security Policy Database maintained at every host implementing IPsec. According to a set of rules specified by the policy and involving various packet field values like source, destination, type of transport protocol, the packets are dropped, protected or sent in clear as mentioned in [20]. This mode of operation often adds an additional layer of complexity in the application development caused by the need to communicate with the local IPsec implementation. In case of SRTP this problem does not appear, SRTP encryption and decryption process normally taking place only at both ends of an RTP session and being independent of the hosts encountered in the path of the media stream. As a resemblance between both solutions, IPsec defines Security Associations instead of SRTP cryptographic contexts. A SA (Security Association) is associated, like the SRTP cryptographic context, to each IPsec protected stream (so different associations will be used for outgoing and incoming traffic) and includes specific information like: the mechanism used for protection – ESP or AH, an SPI – Security Parameter Index identifying the SA, the cryptographic algorithms used, source and destination address of the packets being protected and others. These Security Associations are also kept in a Security Association Database like the policies mentioned above, and selected in order to process the packets. [16] describes detailed specifications about IPsec related databases maintenance. In case of SRTP there does not exist any specification related with the use of a database, due to the ephemeral characteristic of the cryptographic context, which is different for each RTP session and includes parameters like ROC for example modified for every packet sent or received. The implementations also do not usually consider such a method of storing the cryptographic context data in a database, because it would imply longer access times which may prove critical in VoIP deployments.

IPsec makes use of IKE [19] for key exchange. The IKE operation has two main phases as described in [20]:

- setting up a secure connection and proving identity of peers
- negotiating the IPsec SA

In comparison SRTP needs an external defined key management solution not specifically defined by the standard. There are various options available, the most important being described in the next chapter.

IPsec may be used in two modes: transport and tunnel.

Transport mode is used in host-to-host transport communication and implies only the encryption of the payload inside the IP packet. In case of RTP flow this means the encryption of the entire packet: RTP header and payload and the lower level UDP header. The IPsec header is inserted between the IP header and the IP payload. The packet structure in transport mode would consequently look like the one depicted in the figure below:

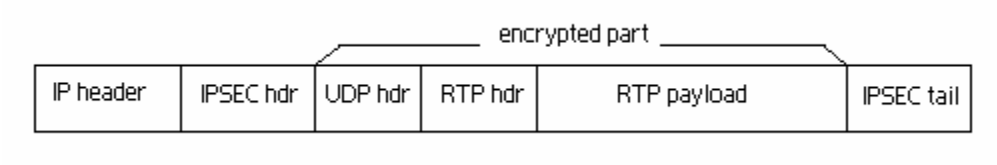


Figure 6

The tunnel mode is used in VPN communications. In this case the encrypted part consists in the entire IP packet, a new IP header being added.

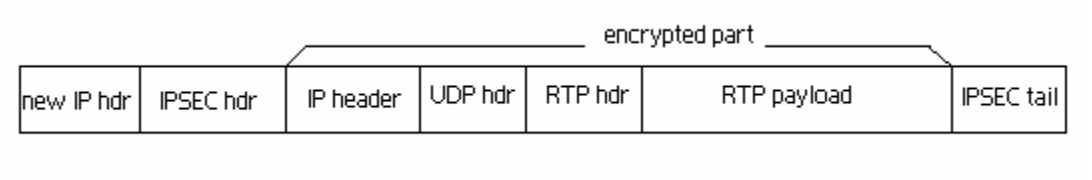


Figure 7

It can be observed that the computational overhead added by encryption and decryption process is larger in case of IPsec than SRTP where only the RTP payload is the subject of securing.

Besides the increased length of the encrypted part there also appears a problem related with NAT and firewall traversal, due to the fact the UDP header is encrypted. Therefore, taking as example the IPsec transport mode, the UDP header is doubled in practice. Consequently the resulting packet, with even more additional overhead, detailed for ESP usage presents itself like figured in [20]:

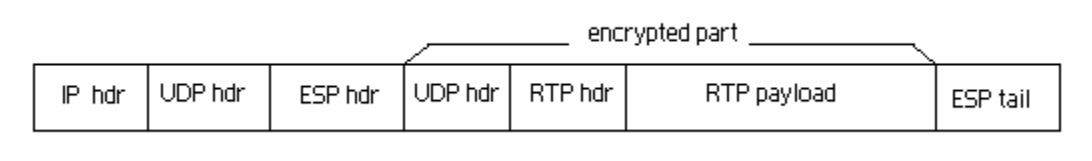


Figure 8

In comparison with the above, the SRTP packet has the already mentioned structure:

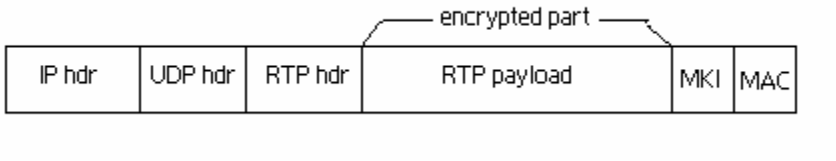


Figure 9

The ESP tail consists in padding and authentication data of variable dimension which summed results in at least 8 bytes. In case of SRTP the MKI and MAC added after the RTP payload are exactly 8 bytes. The ESP header consists in the Security Parameter

Index (SPI) and in the sequence number which added to the additional UDP header results in a minimum overhead of 16 bytes per packet as a difference from SRTP.

According to [21], depending on the cryptographic algorithms used, the overhead may vary with respect to the ESP tail and an additional initialization vector which may be included between the ESP header and the encrypted part. Two cases are considered without taking in account the additional UDP header:

- 3DES encryption: 8 bytes ESP header, 8 bytes IV, 9 bytes ESP padding, 12 bytes authentication data; a total maximum of 37 bytes per packet opposed to the 8 bytes of MKI and MAC from SRTP tail
- AES encryption: 8 bytes ESP header, 16 bytes IV, 17 bytes ESP padding, 12 bytes authentication data; a total maximum of 53 bytes per packet opposed to the 8 bytes of MKI and MAC from SRTP tail

Also, as stated in [21], the overhead becomes of more importance in relation with the codec used in actually transmitting the media stream. Due to the sampling parameters and codec characteristics the effective audio or video stream payload that is subject of encryption in each packet may vary in size. In the given example of G.711 codec the IPSec overhead has values of 30 to 50 percent of the transmitted data (which means that near half of the bandwidth allocated for audio streaming is consumed by additional cryptographic data). We would like to mention that the G.711 example is not randomly chosen, the respective codec being a mandatory requirement regarding audio format support for VoIP sessions according to ITU specifications.

Related with optimizing the bandwidth usage there are situations where the authentication tag, which is optional in SRTP packets is omitted. However, this practice is discouraged, an attacker being able to alter the ciphertext and disrupt the conversation. Also, replay protection is possible only in the presence of authentication, without this being possible to reinject a previous media packet at the media path, an example of a possible succesfull attack being a rerun of an IVR[6] session.

Though, under performance considerations, it seems that SRTP is recommended, it would be an interesting comparison to observe the delay added by this solution to normal RTP streaming. This aspect is analyzed in [22] using Opnet Modeler 10.5 simulation software [23]. The simulated environment includes two Local Area Networks connected through two routers, with 25 users on each LAN. The simulation implied on first phase, an unencrypted RTP transmission session between two endpoints, each one from one of the two networks, using as codec the mandatory supported standard G.711. In the second phase the same configuration was used but this time secured through SRTP, the keys being provided through Diffie-Hellman exchange (available in various solutions like the ones described in the next chapter). Various measurements are taken in the mentioned study [22] with respect to the call start, the key exchange, and the router passing. Even if in the key exchange phase the delay between two peers in the two different networks is 20 ms higher than in clear transmission, this being the peak achieved regarding the added latency, this is safe behind the limit of the 150 ms value stated as a critical threshold in the ITU.T recommendations [24].

[6] IVR – Interactive Voice Response, is a vocal response system, usually automated and designed to serve a specific inquiry (e.g. checking the balance for a bank account)

SRTP is generally considered a secure standard on its own, few potential flaws being reported since its approval. The main concern is about possible problems that might appear in relation with the external key management solution used. One possible issue reported by [25] relates with the consequences of using the same master key and master salt in the derivation of session encryption material for two different non-simultaneous RTP sessions. Not clearly stated in the quoted document, the assumption is based however on the granted possibility of the attacker to achieve distributing an already used key material to both peers in the external key management phase[7] and a faulty implementation or a randomness collision related with SSRC choosing by at least one peer[8]. Given these conditions the derived encryption session key which is based on the packet index computed using the sequence number starting at 0 and the ROC starting also at 0, will be the same. The initialization vector based on SSRC and packet index will be also the same. Consequently, the generated keystream will be identical with the previous exploited session. The main phase of the attack implied by [25] is based on *xor*-ing encrypted data from the previous exploited session and encrypted data from the current session, the result consisting in the *xor* result of the two unencrypted data streams (the keystream being removed). However in order to decipher one transmission the attacker should be able to reconstruct at least one of the two streams. [25] states that this should be possible due to the high redundancy encountered at least in the start of the conversations.

Though we consider the described situation as an interesting approach we find it not to be a security flaw of SRTP based on lack of randomness in key derivation as stated by [25], due to the fact that conditions based on external key management algorithm exploitation are implied. In addition, even the described attempt succeeds we find it somehow difficult to decrypt an entire conversation based only on the *xor*-ed audio data obtained after the attack. Doing that would imply after all having already a RTP session (a conversation) unencrypted, or reconstructing it based on other means. Nevertheless, we recommend as a good practice for a SRTP implementation to avoid using the same SSRC even for consequent RTP sessions. Regarding the usage of the same value for SSRC in simultaneous RTP sessions, more information can be found in section 9.1 of the SRTP RFC [4] essentially stating that the SSRC must be unique between all the RTP streams within the same RTP session that share the same master key.

[7] We will detail a case of replay when an attacker manages to distribute the same key material as in one of the previous sessions encountered for one of the key management solutions described in section 4.4

[8] According to [3], section 8, the SSRC identifier is randomized by a peer using it for every session; the standard states a SSRC identifier should be globally unique inside a SRTP session but it doesn't imply uniqueness for two consequent sessions

3.2. The Key Management Problem

Like stated in the previous sections, SRTP can not operate without an external key management mechanism. This mechanism should provide the master key, master salt and additional parameters required by the SRTP cryptographic context. The SRTP specification does not recommend a specific key management solution. However it does specify several recommendations for any chosen one.

In section 8 of the SRTP RFC [4], there is stated for example that the key management used in relation with SRTP should provide the randomized SSRC used for the protected RTP stream. In addition the initial sequence number for the sent RTP packets may also be specified by the key management solutions. The direct relation with the parameters normally maintained by RTP is caused by the fact that these are also part of the SRTP framework securing mechanism. Through the delegation of SSRC management, situations like the one described in the previous section can be avoided. Therefore concerning an implementation of an RTP stack, this should be able to manage itself the RTP operating parameters specified but also to offer the option of having them set by external means. One aspect which is not controlled by the key management and is left for the SRTP framework is triggering the rekeying. Exchange and usage of a new key should occur after a maximum of 2^{48} SRTP packets or 2^{31} SRTCP packets. These intervals are however estimated to support a continuous communication length of four months (which is implausible) and rekeying normally occurs when specifically stated through the interval set in the SRTP cryptographic context. Even if rekeying is triggered by the SRTP framework, the interaction with the key management solution is not part of the SRTP standard, this being supposed to be specified by the external solution. This is not however specified in detail in every available key exchange mechanism. The usual communication flow consists in halting the media stream after the initial SIP exchange until each of the peers has received the cryptographic context from the key exchange part. Alternate implementations permit triggering stream encryption from the key management solutions by performing key exchange even if the call is already in progress.

The key management considerations specified by SRTP state also that key sharing between streams belonging to the same session is secure due to the unique SSRC of the sources. This does suffice for protection against a keystream cancellation attempt performed by applying XOR between the two encoded media streams flowing in each direction. A typical example of sharing the key material would consist in the key management providing the same key for both inbound and outbound traffic between two peers. The different existent key management solutions have different ways of handling the specification of a possible common key. One case when the key is almost certain to be shared inside a RTP session is for the RTP stream and RTCP associated traffic, this being the default handling.

We tried to summarize in this section the main considerations related to key management as stated in the SRTP standard [4]. At the moment when [4] was approved as an IETF standards track RFC, the only key exchange dedicated standardization (in progress at that moment) for keying media stream based traffic was MIKEY [8]. In the next chapter we will detail the characteristics of this, and also other important options available today from the existing range of key management solutions.

4. Key Management Solutions Available for Media Stream Securing

From the key management solutions available at this moment, dedicated for media stream securing in SIP VoIP sessions, three can be considered of main importance, the rest being either variations or discontinued versions of drafts in process of standardization. The three mentioned solutions are MIKEY [8], DTLS-SRTP [9] and ZRTP [10].

These and other mechanisms are enumerated in [26] along with a brief description and the general characteristics and requirements for a media security management protocol. The quoted [26] classifies security mechanisms used to protect the media stream based on the robustness against the adversary capabilities. More exactly an adversary can be either active or passive with respect to the key exchange protocol type which takes places as part of the media stream securing. Being active implies that the adversary must act either at the signaling path or at the media path or both in order to obtain the key for the specific protocol. For a passive successful adversary is enough to have access at one of the media or signaling levels or both to observe information essential to learn the key. The key exchange protocol mentioned classification is done by defining seven classes based on active/passive adversary status. The easiest class to compromise is *no-signaling-passive-media* for which only passive (observe) access to the media channel is needed to reveal the content. In this class falls the unencrypted media stream. The other classes, defined in order of robustness, are:

- *no-signaling-active-media*
- *passive-signaling-passive-media*
- *passive-signaling-active-media*
- *active-signaling-passive-media*
- *active-signaling-active-media*
- *active-signaling-active-media-detect*

The last class states that an intruder must act both at the signaling and the media path and implies also that the key exchange solutions which fall into this category must be able to detect an acting intruder. Therefore the membership to this class is one of the security requirements desired for key management solutions as implied by [26]. However, the protocols analyzed by [26] are not actually classified in one of the stated classes of robustness, the authors mentioning that one protocol may fall in either one or other class, depending on the mode in which the protocol is operated.

Even if [26] clearly states that the enumerated solutions apply to SIP signaled communications, at least part of the described key management solutions can also be applied to other types of media sessions based on SRTP, and consequently many of the

considerations made, like the robustness classification previous mentioned, are also appropriate to such cases.

Despite the apparent large number of possibilities for a keying solution (more than ten mentioned only in [26]), parts of these are either expired or variations of the same key management protocol. The techniques are classified according to the level of operation, which can be the signaling path, the media path or a hybrid signaling/media solution.

The key management protocols enumerated in [26] which operate at the signaling path make use for key establishment and exchange, of SIP [1] message body. Parts of the enumerated solutions for this level of operation are variations and extensions of MIKEY [8]. MIKEY is essentially a solution which operates at the signaling level layer, being transported through SIP flow. We will detail MIKEY in the next section of this chapter. A second version of the MIKEY standard has been in development [27] (though the draft expired) this having as one of the most important additions defined, an in-band negotiation mode. This feature classified the solution as a hybrid signaling-media path one.

Other ways to provide the cryptographic context parameters to SRTP from the signaling path enumerated in [26] are Security Description with SIPS (SDES) [28] and SDP-DH [29] based on usage of SDP attributes to carry the key information aided by external ways of protection. We will refer briefly to these methods in section 4.4.

Regarding the media path keying techniques, the only solution which can exclusively operate at this level mentioned by [26] is ZRTP [10]. However, this also can make use of some signaling path SDP features for the initiation step which would classify it as a hybrid solution, but this kind of interaction is not required. We will discuss ZRTP in more detail in section 4.3.

The third category of keying techniques is the hybrid one which makes use of the signaling path and also of the media path. The most notable solution which acts this way is DTLS-SRTP [9]. The submitted standard describes a Datagram Transport Layer Security (DTLS) extension to establish keys for SRTP which operates at the media level but makes use of prior public key fingerprints exchange through SDP at the signaling path. A more extended description of this solution will be presented in section 4.2.

Another hybrid solution, also described in an IETF draft, though the document expired in 2007, is EKT: Encrypted Key Transport for Secure RTP [30]. A brief overview about this can be found in section 4.4., along with the two signaling solutions based on SDP attributes which we mentioned above.

4.1. MIKEY – Multimedia Internet Keying

From the area of key management solutions intended to secure media stream transmissions, and in particular the ones implied by a SIP VoIP session, MIKEY – Multimedia Internet KEYing is the only one which is currently approved as a standard and does not rely for its main use on complementary mechanisms needed to ensure security on hop by hop basis. MIKEY usage is not limited on SIP VoIP sessions, the standard describing it more generally as a "key management solution that addresses multimedia scenarios" [8], one example given being RTSP sessions which are a frequent part of IPTV communications. In SIP sessions MIKEY makes use of SDP for payload transport during the key exchange phase and, as presented in the layer separation description, SDP flow is integrated in the signaling path not having a direct connection with the media one. As the other cases of media stream securing dedicated key management solutions, the motivation for MIKEY usage against well-known mechanisms from the same area like IKE, lies mainly in the need of lower latency needed by the real-time streaming.

We present in the following sections a description of MIKEY functionality, resuming the essentials in [8] and commenting these aspects. In this purpose we will use part of the terminology defined in [8] with slight modifications, as it follows:

- Data Security Association: the information needed by the media stream security protocol, in our case SRTP, in order to ensure the confidentiality of the data flow, for example the encryption keys and the other parameters in the SRTP Cryptographic Context
- Crypto Session: the data streams protected by one instance of the media stream security protocol; in case of SRTP this will consist in most cases in two streams, the RTP and associated RTCP one for one direction of communication between a single source and a single destination, either incoming or outgoing; we consider that a Crypto Session includes a Data Security Association and isn't actually "viewed" like one as [8] implies, even though a Data SA might and should be unique in some cases to a Crypto Session
- TEK Traffic Encrypting Key: the key used in the effective encryption of the traffic from a crypto session, and part of a data security association; this is derived without any need of communication from the TGK described below and may also be derived into further keying material
- TGK TEK Generation Key: the value established between peers from which the TEK are generated
- Crypto Session Bundle: a collection of Crypto Sessions which may share a common TGK

The scenarios mentioned in [8] as possible targets of MIKEY application comprise of:

- simple peer-to-peer calls between two parties
- one-to-many media sessions with the sender as the peer responsible with the security insurance (the case of usual IPTV transmissions)
- many-to-many without a centralized control unit (practically like a chaining made by linking a series of simple two party peer-to-peer sessions with a peer in charge as the Initiator; this is the only one which has the right to include new members in the chain)

The MIKEY standard [8] does not cover the case of a centralized many-to-many session, one of the scenarios for a VoIP conference.

As level of abstraction MIKEY is supposed to function at a complete separate layer from the direct media securing protocol (SRTP in our discussed case). The target of MIKEY is to provide common TGK and parameters for a Crypto Session Bundle which may contain more than one Crypto Session. The standard describes also the way of deriving TEK for each Crypto Session. In our particular case, the Crypto Session is represented by a SRTP cryptographic context, and the TEK will be the Master Key Provided. A Crypto Session Bundle may be represented for example by two Crypto Sessions: the SRTP cryptographic context for the inbound traffic and the SRTP cryptographic context for the outbound traffic which takes place between two peers in a VoIP communication session.

As stated in the description of SRTP key management specifications in section 3.2 two RTP streams inside the same session may share the same key material (though it is not recommended), due to the fact that after application of the SRTP defined derivation this material will result in different keystreams (caused by different SSRC values involved in computation on each side). This is different from our MIKEY Crypto Session Bundle example above where the SRTP cryptographic contexts are associated to two different Crypto Sessions and consequently have different TEK. A possible case that will involve sharing the key material between inbound and outbound traffic would be for instance a single Crypto Session, having a single TEK and covering a bidirectional stream. This case is not excluded by the MIKEY specification. However, the standard does not clarify how a single Crypto Session could cover two SRTP cryptographic contexts. Due to the general terminology used by the standard which may raise ambiguities when applied specifically on SRTP, an appendix suggests as we also specified above the corespondence between a Crypto Session and a SRTP stream. As it will be presented in the next section two distinct Crypto Sessions from the same bundle can not share the same TEK due to the fact that in the TEK derivation process the Crypto Session IDs are involved and these are distinct.

Even if the key could be shared between cryptographic contexts along with related parameters, each stream has its own SRTP cryptographic context with different parameters like ROC, replay list and others. Viewing a Data SA as a cryptographic context, we could tell hypthetically tell that the Crypto Session could include two Data SA sharing the TEK. This is the reason we preferred above referring to a Data SA as included into a Crypto Session, and not identifying with one. Another approach would be

that we can also renounce to our convention of identifying one Data SA with one SRTP cryptographic context as suggested by the mentioned appendix and consider a Data SA as covering the common and different parameters from two SRTP cryptographic contexts in this case. Anyway, such an interpretation of the standard, though seeming to be correct, may lead to difficulties in an eventual attempt of verifying the MIKEY mechanisms implying Data SA parameters due to inconsistencies.

4.1.1. Overview of MIKEY Operation and Exchange Methods

The MIKEY standard defines three possible methods for key establishment:

- using a preshared key
- using public key encryption
- using Diffie-Hellman key exchange

All the enumerated options refer of course to TGK establishment, this being, as mentioned in the previous section the shared value for more Crypto Sessions inside a Crypto Session Bundle. The next facts need to be mentioned as general specifications applying to all three methods:

- along with the TGK or the TGK component material (in case of DH exchange) a random value - RAND – is transported
- the two values, TGK and RAND, are used for further derivation of TEK material
- a timestamp is also included in the exchange in order to avoid replay attacks
- the CSB ID and CS IDs are other fields present in any MIKEY packet exchange being used in the derivation mechanism

We will describe the basics of the three modes in the next sections, taking as example the simple case of only two peers involved in communication. First of all we consider necessary to make a brief overview of the local actions related with key derivation, which are common almost entirely for every exchange mode.

In order to obtain various key from a specified value MIKEY makes use of a pseudorandom function $PRF_{outlen,inlen}(k,x)$ where *k* is an input key, *x* is a label specific for the key to be derived which might be either an encryption, authentication or salting key used in order to protect the MIKEY exchange or might also be the TEK. *outlen* represents the output length result, and *inlen* in this case is the length of *k* and not of *x* as defined in the SRTP PRF. We have chosen to use this change from the previous definition used in section 3.1.3 in order to be consistent with the form described in the MIKEY specification. The pseudorandom function is defined in the next way:

$$PRF_{outlen,inlen}(k,x) = P(k[1], x, m) \text{ XOR } P(k[2], x, m) \text{ ..... XOR } P(k[n], x, m)$$

where:

- $n = inlen/256$
- $m = outlen/160$
- $k[i]$ = the i^{th} block of 256 bits from k

P is a concatenation function obtained from computing keyed hashes in a recursive manner as described below:

```
P (k[i], x, m) = HMAC(k[i], A[1] || x) ||
                 HMAC(k[i], A[2] || x) ||
                 ……..
                 HMAC(k[i], A[m] || x)
```

where

```
A[0] = x
A[i] = HMAC(k[i], A[i-1])
```

We have chosen to describe the entire derivation mechanism because, as we mentioned already, is part of every derivation needed by the various key exchange modes. Also, an aspect to be considered is the structure of the label used in derivation (*x* in PRF). This varies based on several constants associated with each possible usage of the pseudorandom function. Besides this label, it has in its structure the crypto bundle ID and, the crypto session ID, the RAND value. One important component of the label to note is represented by the RAND value. The RAND value is chosen randomly with the purpose to ensure the freshness of the computed derived value. For example, without integrating this value, in case of rekeying during the current RTP session, the same results as before will be obtained for the TEK. This way the lack of randomness in the SRTP derivation function mentioned as a potential problem in section 3.1.4 could be exploited. Adding this value practically ensures the needed randomness. The CSB ID and CS IDs ensure TEK uniqueness for the derivation process at a given time between any two different crypto sessions. A common example to illustrate this meaning, would be transporting in the message header two CD IDs for the same CSB, each one of the two coresponding to a Crypto Session associated with the sender's stream and respectively the receiver one. Along with the exchanged shared TGK, these CS IDs are used in the TEK derivation process as described in the next sections.

As an observation it is useful to take note that the derivation result in MIKEY is obtained mainly from a hashing sequence, this observation being useful in an eventual attempt to model a simplified version of the key exchange with the eventual purpose of verification.

We will describe these key exchange modes in the next sections without getting in more detail related with the other cryptographic transformations specified, these being standard algorithms[9] described in section 4.2 of [8].

[9] MIKEY makes use of AES in counter mode for key transport encryption, HMAC with SHA-1 for authentication, RSA PKCS#1 for public key encryption used in envelope mode exchange and OAKLEY 5 for DH groups in DH mode exchange

4.1.2. Key Exchange Using Preshared Material

This method is considered to be the most efficient with respect to both time spent and amount of data transported. It implies only symmetric cryptography usage. The main problem presented by this case consists in scalability regarding a scenario involving a large group of peers.

In this mode it is supposed that the peers already share a secret *s*. The issue mentioned above arises from the number of secrets to retain. This secret is used in the derivation process of the encryption key and of the authentication key, used to protect the key exchange messages. We can consider this the first step of this mode:

- Initiator (local action): $s \text{ --- } PRF_{outlen,inlen}(s,x) \text{---> } encr\_key, auth\_key$

where PRF is the pseudorandom function described in the previous chapter and *x* is the label identifying which output to be derived (*encr_key* or *auth_key*). *x* is composed from the next values:

$$x = output\_type\_constant \,||\, 0xFF \,||\, crypto\_bundle\_id \,||\, RAND$$

Note the RAND value chosen by the initiator in order to ensure the freshness of *x*, mentioned in the previous section. This value is also necessary for the responder to obtain the *encr_key* and *auth_key*.

The second step would be composing and sending the message in order to exchange the TGK. The Initiator's message, I_MESSAGE consists from:

- a specific MIKEY header (presented in detail in Appendix 2)
- a timestamp T to prevent replay attacks
- a random nonce RAND used as a freshness value
- the initiator ID (optional)
- the responder ID (optional)
- other security parameters if needed by the crypto session
- the shared secret KEMAC

The shared secret KEMAC is computed as the TGK encrypted with *encr_key* and is authenticated through a MAC computed using *auth_key* over the entire I_MESSAGE. The ID's are optional, the reason being based on the fact that the two peers may already know each other from the signaling session. If they don't, these fields are necessary in order to correctly select the preshared value *s*. The secrecy of the *encr_key* is guaranteed through the fact that its computation is based also on the secret preshared value and not only on RAND which is exchanged in clear.

After receiving the Initiator's message, based on the RAND value received and *s*, the Responder computes *encr_key* and *auth_key*.

- Responder (local action): s --- $PRF_{outlen,inlen}(s,x)$---> encr_key, auth_key

The last step, performed by the Responder would be to send a message, R_MESSAGE, having the role to confirm the authenticity of the exchanged data. The message consists from:

- a specific MIKEY header
- a timestamp T to prevent replay attacks (the same with the one received)
- the responder ID (optional)
- a MAC computed using *auth_key*, over R_MESSAGE, the timestamp inside the I_MESSAGE, and the identities of both parties

After the exchange takes place the Initiator and Responder use the TGK to obtain the TEK (SRTP master key) by applying again the same PRF defined in the previous section:

- Initiator (local action): TGK --- $PRF_{outlen,inlen}(TGK,x)$---> TEK
- Responder (local action): TGK --- $PRF_{outlen,inlen}(TGK,x)$---> TEK

The label in this case has as difference the addition of the Crypto Session ID:

x = output_type_constant || crypto_session_id || crypto_bundle_id || RAND

In a both way communication session a TEK is normally generated as described above for each one of the streams, coressponding with the associated Crypto Session.

According to the MIKEY specification the RAND value is common for a Crypto Session Bundle, consequently possible the same for more than one Crypto Session. Also according to the MIKEY specification the TGK might also be shared for a Crypto Session Bundle. In this particular case the difference in the TEK generated results exactly from the difference between the Crypto Session IDs.

The Responder's message is described by the standard as being optional, and to be sent only if required by the Initiator. This is done by activating a flag in the specific MIKEY header, this being actually the only importance of the transmitted header with respect to key exchange securing flow. The reason for defining the Responder message as optional results especially from the case of one to many broadcasting when the sender should verify the authenticity of a potentially large number of receivers. In case of both way media streaming the response is necessary, this way the sender obtaining the CS ID value needed in the TEK derivation process (which is of course different from the CS ID of the sender).

4.1.3. Key Exchange Using Public Key Material

This method implies also like the previous one only a two message exchange phase. In this case an "envelope key" is used in order to protect the TGK. Therefore a first local step can be considered obtaining the encryption and authentication keys from the "envelope key" derivation:

Initiator(local action):

env_key --- $PRF_{outlen,inlen}(env\_key,x)$---> $encr\_key, auth\_key$

The label *x* in this case has the same structure as in the preshared case.

The "envelope key" is communicated to the Responder in encrypted format using the Responder's public key. Along with this, the TGK is also transmitted. The Initiator's message content is composed of:

- a specific MIKEY header
- a timestamp T to prevent replay attacks
- a random nonce RAND used as a freshness value
- the initiator ID or a certificate of the initiator (optional)
- the responder ID (optional)
- other security parameters if needed by the crypto session
- the shared secret KEMAC
- the hash of the public key used for the encryption of the envelope key (optional)
- the encrypted envelope key PKE
- a signature covering the entire initiator message

In this case the initiator ID does have cryptographic importance being also a part of the KEMAC. If the message doesn't contain the initiator ID and instead a certificate is carried, this will be used to prove the correspondence with the ID information in the KEMAC.

The KEMAC is computed by encrypting the concatenation of the Initiator ID and the TGK with *encr_key*, and authenticated by adding a MAC at the end calculated using *auth_key* over the rest of the KEMAC payload.

The message of the initiator may include the hash of the responder's public key used in the encryption of the envelope key. This is specified to take place when the responder has more than one public key, in order to communicate which of them was used.

The PKE represents the envelope key used in TGK protection, encrypted on its own using the responder's public key.

After it receives the message the Responder decrypts the "envelope key" using the private key, and uses it in order to obtain *encr_key* and *auth_key* to decrypt and authenticate the TGK:

Responder(local action):

env_key --- $PRF_{outlen,inlen}(env\_key,x)$---> $encr\_key, auth\_key$

Same as in the pre-shared key case a message may be sent in reply to initiator by the responder. Its usage is confirming the authenticity of the exchanged data. Like mentioned in the preshared case, this message is also needed here in case of both way streaming in order to provide the CS ID to the initiator, this being needed in the TEK derivation. The fields are the same as the ones mentioned in the previous section:

- a specific MIKEY header
- a timestamp T to prevent replay attacks (the same as the one received)
- the responder ID (optional)
- a MAC computed using *auth_key*, over this message, the timestamp inside the initiator message, and the identities of both parties

The TEK is derived by both peer in the same way as in the previous exchange mode presented.

Regarding this method, the standard [8] mentions a hybrid possibility of application, by combining in the first phase the application of public key encryption for a crypto session, retaining the envelope key and using it for later crypto session following the preshared key approach. However, it is stated that the key cache should not be preserved indefinitely, but only for a limited period of time whose length might be defined based on a local policy.

It can be observed that the current mode doesn't differ too much from the preshared one, the difference appearing practically in transporting the encryption key and authentication key for the TGK, and not in encrypting or obtaining the TGK in a different way, which is the case of the last method.

4.1.4. Key Exchange Using Diffie Hellman Mechanism

The third method is considered optional by the standard regarding a possible MIKEY implementation, as a difference from the previous two discussed which are mandatory. One reason is that this method is feasible only for peer to peer scenarios involving only two endpoints communicating. Due to the nature of the mechanism this can not be applied to create group keys.

This method consists essentially in a normal DH parameter exchange between the two peers. In order to ensure the security of the exchange the mechanism has the next described design characteristics. We consider the DH value of the Initiator to be g^{xi}, where *g* is the generator and *xi* is the random secret parameter. The message sent by the Initiator to the Responder in the DH mode consists of:

- a specific MIKEY header
- a timestamp T to prevent replay attacks
- a random nonce RAND used as a freshness value
- the initiator ID or a certificate of the initiator (optional)
- the responder ID (optional)
- other security parameters if needed by the crypto session
- the DH parameter of the Initiator g^{xi}
- a signature covering the entire initiator message

Again it can be observed that the message has quite a similar structure with the one described in the preshared method mechanism.

The Responder computes it's DH value as g^{xr} where *g* is the generator and *xr* is the random secret parameter, and sends it to the Initiator as part of a message composed from:

- a specific MIKEY header
- a timestamp T to prevent replay attacks (the same as the one received)
- the responder ID or a certificate of the responder (optional)
- the initiator ID
- the DH parameter of the Responder g^{xr}
- the DH parameter of the Initiator g^{xi}
- a signature covering the entire responder message

Following the exchange, both peers can compute the TGK as g^{xi*xr}.

Of course in this case the Responder's answer is always mandatory being necessary to compute the TGK. Even if the RAND is not necessary any more in this case for computing an intermediate encryption key and authentication key to secure the TGK, it is mandatory to be sent in order to be used for TEK derivation. This is derived by both peers in the same way as in the previous exchange modes presented. The security of TEK is guaranteed through the fact that the TGK is not actually exchanged like the RAND, but computed at every peer based on the secret DH parameters.

4.1.5. MIKEY Transport in SIP Sessions and Other Considerations

MIKEY is specifically designed to be an independent key exchange solution, disregard of what signaling protocol is used and even without a specific relation with the media one (though like stated in a previous section a corespondence with SRTP is established in a MIKEY appendix). This is the reason for which the interaction with SRTP is not specified directly and the terminology used (TEK, Crypto Session, etc) is different. The MIKEY message transport inside SIP based VoIP sessions is described in a companion standard [31]. Quoting from the mentioned document: "the approach used is to extend the SDP description through a number of attributes that transport the key management offer/answer and also to associate it with the media sessions".

Essentially this means that [31] establishes new custom SDP attributes of the form:

```
a= key_management_attribute_field : value
```

Only two types of fields are defined, these being protocol id and key management data. The key management data field is used for effective transport of MIKEY messages encoded in base64 format.

Like MIKEY, [31] is also an independent standard mainly defining a general interaction framework which could be applied to various key exchange protocols. There are however detailed specifications dedicated for MIKEY as a possible key management solution.

One of the general requirements stated is that the key exchange should be performed in at most one request-response message in order to maintain a low complexity. All the three modes discussed above comply with this. Another general requirement is to be possible from SIP/SDP to retrieve key management data. Again MIKEY complies with this, the specifically retrieving of Data SA being discussed in section 4.4. of [8].

Other aspects related with MIKEY message transport in SDP, like support for multiple key management protocols can be found in section 4.1. of [31].

The key exchange limitation, that this should be performed in one request-response message, is also the reason for which MIKEY uses timestamps for mutual authentication and replay protection. In a classic challenge-response scenario three messages should be sent: the first one from the Initiator, the second from the Responder including a challenge (for example a hashed of a shared secret concatenated with a certain value), and the third that includes the response. In case of MIKEY the Initiator sends the response for an already known challenge, which effectively consists in the hash (part of KEMAC) computed over the timestamp and the rest of the message. The Responder as part of authentication process extracts the timestamp and verifies if it is in a allowable clock delay interval (the specification states that the peer clocks should be synchronized), afterwards verifying the hash for the message. Also the timestamp is stored in a replay cache if the message authenticates successfully in order to be used for replay detection of future messages.

As a security concern, as stated even by the official specification [8] in sections 5.4 and 9 and also mentioned in [25], MIKEY is susceptible to possible Denial of Service attacks. In order to prevent part of the possible DoS situations one recommendation is to include always the responders ID in the Initiator's message. Otherwise messages intended for other destinations could be successfully routed to an attack target which based only on the fact that the message's signature verifies the Initiator's ID, will accept the message. This applies in particular for the DH mode, in the other two the problem being detected due to the MAC authentication, however this needing additional operations normally unnecessary for a wrong addressed message. Other DoS situations may arise in relation with the replay cache maintained as specified by section 5.4 of [8] a very high load of incoming messages causing the number of maintained records to grow to unmanageable size. However, recommendations for replay cache maintenance in order to avoid this situation are available in section 5.4 of the standard.

Another susceptibility that appears, this time in relation with the MIKEY message transport through SDP consists in a potentially MitM attack. This attack implies the removal of the MIKEY fields from the SDP message not having a media stream interception effect, only disrupting the call. For this reason the SDP messages should be protected by an external mechanism functioning at the SIP signaling path.

4.2. DTLS-SRTP

DTLS-SRTP is one of the possible key management solutions mentioned as part of the securing process regarding the media stream transmissions in SIP based communications. The solution is currently in standardization process, being described in [9] accompanied by [32][33] , all mentioned documents having draft status on the IETF standards track. In the current section we will briefly describe DTLS-SRTP main operation mode, its features and some of the points which may lack security.

DTLS-SRTP is a hybrid key management solution with respect to the path at which it activates. More exactly it makes use of both the media path where the RTP and SRTP flow is present and also of the SIP signaling path. DTLS-SRTP is mainly based on a combination of the DTLS protocol [34] which is the component running at the media path and of SIP Identity [35] which is of course part of the SIP layer.

4.2.1. DTLS Overview

DTLS-SRTP, as the name suggests it, is based on DTLS (Datagram TLS), which is on its own account based on the well-known TLS [36]. Also like TLS, DTLS is meant to be placed between the application layer and the network layer with respect to the OSI model, as stated in [37], in order to provide security for the higher protocols running on top of it. TLS is not usable for datagram based applications, needing a reliable transport channel like TCP. DTLS was designed mainly to cover this lack of support, and therefore its main differences and additions to TLS are related with datagram transport, allowing applications to “replace each datagram socket with a secure datagram socket” [37] in conditions of reliable session establishment. This implies a retransmission mechanism needed for the guaranteed delivery of handshake messages. Besides the similarity with TLS, DTLS borrows also from IPSec a series of techniques. This is mentioned in [37] along with a motivation, consisting in several limitations, of why not using IPSec directly for the purpose of a secure datagram channel. We already described such reasons when we presented SRTP in chapter 3. Another hybrid solution idea using TCP for key exchange is briefly mentioned in the quoted document and considered infeasible in this alternative solution context.

The main part of the DTLS protocol flow performed in order to establish a secure channel consists in a handshake phase. This takes place in a client server fashion as depicted in the next diagram:

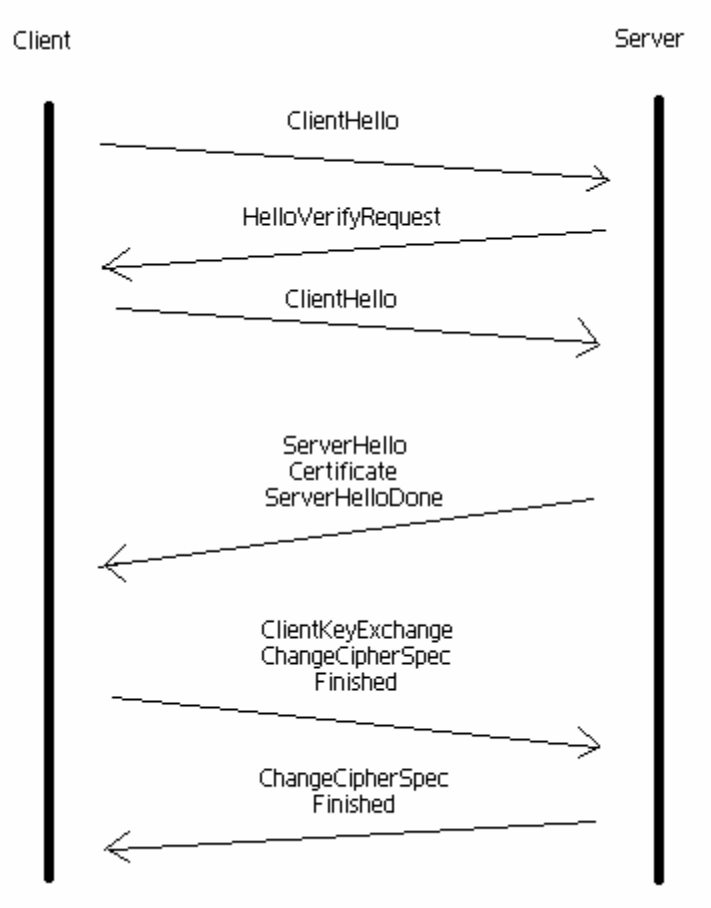


Figure 10

The handshake depicted in the above figure is almost entirely similar, as message flow, with the one that is present in case of TLS, having as difference the first part where a cookie exchange takes part. This operation is necessary due to the connectionless nature of DTLS. More exactly, the client must replay the cookie sent by the server in the *HelloVerifyRequest* in order to prove that it is able to receive further data at the initial claimed IP address. Being based on connectionless datagram transport, an attack could be tried in the absence of this step by sending the initial *ClientHello* message with the apparent source similar with the victim, and consequently using the server as a DoS amplifier. The step isn't however mandatory. The cookie field can be cached from previous protocol runs, so an older client may send it in the first *ClientHello*, in case of a successful cache check the server skipping the *HelloVerifyRequest*.

The *ClientHello* and *ServerHello* messages have as main purpose the negotiation of cryptographic algorithms. Both messages also contain a random nonce with antireplay purpose. The next message sent is *Certificate* containing the server's certificate chain. In more complicated handshakes other messages may follow. In the simplest case, the depicted one, the server sends a *ServerHelloDone* message having an end marker role for the first part of the handshake.

The *ClientKeyExchange* message carries a random secret chosen by the client and encrypted with the server's public key, which will be used in deriving the shared keying material. This is followed by the *ChangeCipherSpec* message used to indicate the switch to

the new secure context. The *Finished* messages are sent to mark the end of the handshake, each of them containing also a MAC computed on the previous exchanges.

For the protocol flow, DTLS makes use of a record format, as TLS does too:

```
struct {
            ContentType type;
            ProtocolVersion version;
            uint16 epoch;
            uint48 sequence_number;
            uint16 length;
            opaque payload[length];
} DTLSRecord;
```

The type and version fields are also present in TLS, being meant to identify the type of record in the message flow and the version of the protocol. The epoch number is added by DTLS in order to resolve problems caused by the possible data loss. The values for this field are incremented with every cipher state change, so no ambiguity may arise regarding the possible loss of a *ChangeCipherSpec* message. If the message is lost, the different epoch number in other data messages received will prove this fact, and also that the messages are secured through a different context. The sequence number field is explicitly encoded in DTLS records, as another difference from TLS, because in this case (connectionless datagram based), messages can be lost or received out of order and consequently the default TLS implicit rollover counter may prove faulty and ineffective.

To avoid the message loss in the handshake phase, where all of them are needed in order to establish the secure channel a timer state machine is used for eventual retransmissions. We reproduce its description as presented in [37]:

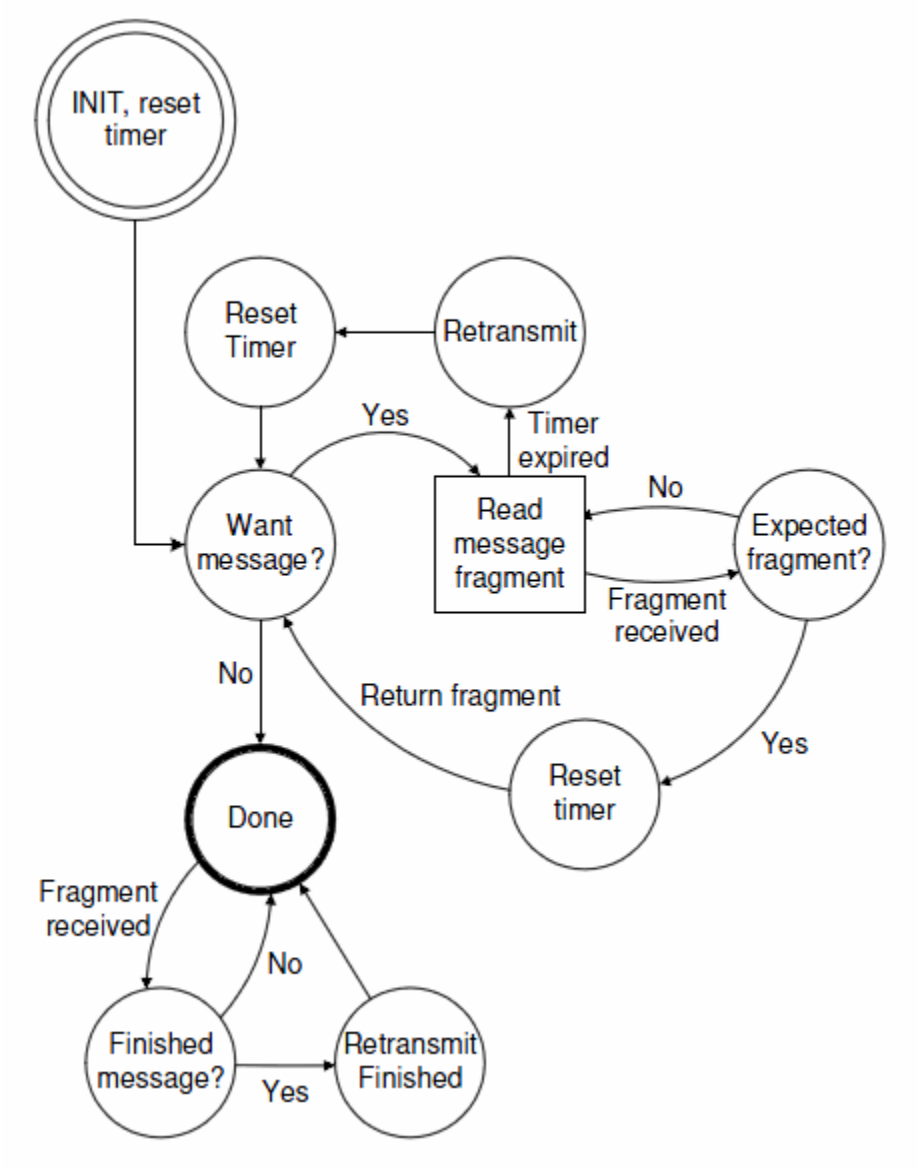


Figure 11

Due to the fact that DTLS record messages passed through this state machine could be larger than the maximum size of a datagram, these are further encapsulated in handshake records having the next specific structure:

```
struct {
          HandshakeType msg_type;
          uint24 length;
          uint16 message_seq;
          uint24 frag_offset;
          uint24 frag_length;
          HandshakeMessage msg_frag[frag_length];
} Handshake;
```

The message type field resembles the DTLS record type. The length will contain the entire message length, not only the Handshake record length, therefore as a good practice buffer space to receive the whole DTLS record (the entire Handshake message) can be reserved in advance. The sequence number field is different from the one contained inside the DTLS record and is the same for all the fragments forming a DTLS record. In case of out of order transmission the DTLS message can be consequently recomposed based on this number, along with DTLS record number and the fragments offset and length which are memorized in the next two fields.

4.2.2. DTLS-SRTP operation description

DTLS-SRTP can be viewed in two equivalent ways, according to [9]: “as a new key management method for SRTP, and a new RTP specific data format for DTLS”. Another definition of DTLS-SRTP would be an extension of SRTP, which acts as an intermediary layer between SIP and SRTP, providing the necessary key management for the latter and making use of the former for several security mechanisms needed in the process. The relation with the SIP layer is mainly linked with indicating the usage of a DTLS-SRTP session during SIP signaling and establishing also through this which peer will act as a DTLS server and which one as a DTLS client. Basically a typical DTLS-SRTP operation session negotiation is “embedded” inside the initial SIP signaling phase of a call, being triggered through the INVITE messages. The INVITE request will contain in the SDP body an attribute which expresses the intended role of the sending peer in the future DTLS negotiation and a fingerprint. The fingerprint is computed based on the sender’s certificate (that will be included in the DTLS messages) and will act like a binding between the future DTLS media path negotiation and the SIP signaling session. Likewise, after the DTLS negotiation ends, the initial receiver of the INVITE will provide its own certificate fingerprint in the 200 OK SIP response, validating and confirming this way the binding between DTLS-SRTP and the current SIP session. In order to secure the information contained in the SIP messages, SIP Identity mechanisms are required these being further described in [35]. We will focus in the next part on a basic description of a DTLS-SRTP negotiation, more detailed examples (and actual use cases) about how this interoperates with SIP being presented in [32].

One DTLS-SRTP operation session, as specified by [9], implies only one DTLS association between two peers. This is able to provide a maximum of two SRTP contexts, in case of media traffic (which is subject of protection - either RTP or RTCP) flowing on the same port for incoming and outgoing data. This particular case usually applies to symmetric RTP, where one DTLS-SRTP session is used for RTP, and one for RTCP. In the more common case of normal RTP operation, the DTLS association provides only one SRTP context dedicated to protect either inbound or outbound traffic. Therefore, a two-peer conversation will make use of four DTLS-SRTP sessions, two for RTP and two for RTCP. As a general rule a separate DTLS-SRTP session must be established for each different pair of source and destination ports. However, there exist ways of optimizing the costly operation of multiple DTLS handshakes, which might occur especially in multistream sessions (more than one media channel). One of these ways consists in using DTLS session resumption as described in the DTLS standard for the subsequent sessions after establishing the first one.

The first difference from normal DTLS operation appears in the handshake phase:

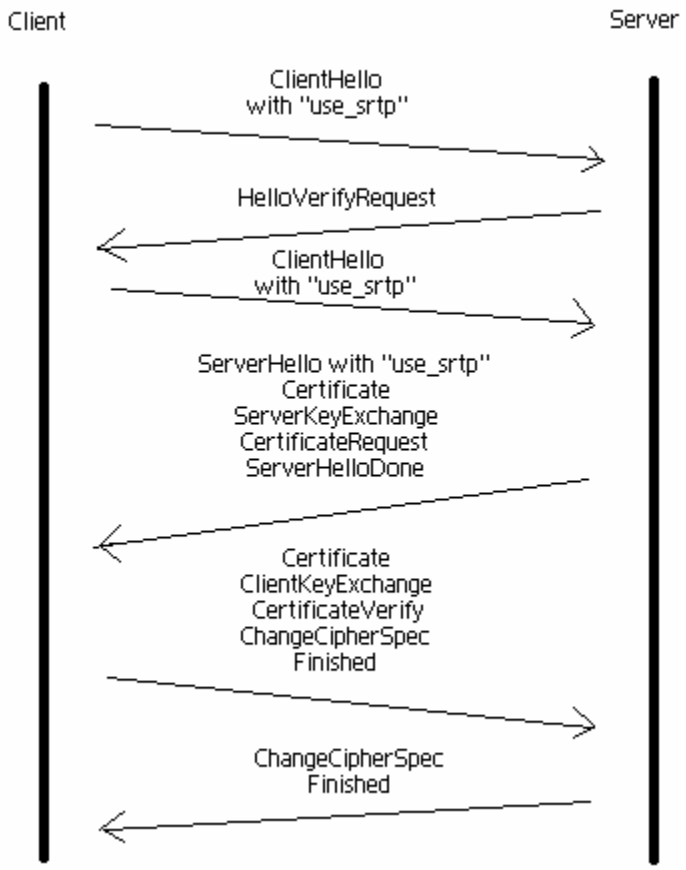


Figure 11

As it can be seen in the figure above the message flow is a bit more complicated than in figure 9. Actually besides the "*use_srtp*" extension, all the additions are part of the original DTLS standard. The reason for the more complex message flow extension lies in the nature of the application being protected. In the basic example depicted when we first described the DTLS mechanism, the sole purpose was the authentication of the server. In the current case also the client needs authenticated (actually in an ordinary SIP conversation the peers have both client and server roles). The presented message flow represents a complete message flow, some message parts being optional. For example the *HelloVerifyRequest* and the resend of *ClientHello* in most cases of DTLS-SRTP implementation will be absent in order to minimize the latency caused by the negotiation.

After the negotiation, the media stream data will be protected directly by SRTP. This means that the stream will not use the DTLS application data record encapsulation. The RTP and RTCP packets will be passed directly to the SRTP stack for transmission. Using DTLS for the data transportation in an RTP/RTCP media transmission case would imply an additional encapsulation overhead which isn't necessary due to the fact that after the initial handshake for the key establishment, the effective protection of the media

stream can be assured only by SRTP. However, any other TLS control messages that could be subsequently sent must use the DTLS framing.

The effective extension data of the "*use_srtp*" extension consists in a list of SRTP Protection Profiles that the client is offering as supported (described in the next paragraph) and the SRTP Master Key Identifier MKI which will be used in the subsequent SRTP transmission, this parameter being optional. We remind that the MKI is used to indicate to a SRTP receiver which is the key protecting the packets containing the specific MKI value. It functions like a map between the packet and the key used in the packet encryption. The MKI parameter mentioned above is carried in the "*srtp_mki*" field of the "*use_srtp*" extension, and has the role to map the association with the SRTP master keys derived from the current DTLS handshake. There is one master key in use ensuring packet protection for each SRTP session. Therefore the MKI values chosen by the client must be distinct from the previous ones and unique for the entire DTLS session length. The exception is the case of DTLS rehandshakes. We will discuss this later in this section. Like mentioned before the MKI usage is optional, as it is also described in the SRTP standard [4], therefore the peer acting as server during the DTLS handshake can either accept or reject it, but we prefer to focus on SRTP communications that make use of the MKI for key mapping because this is the most often case met in practice.

A SRTP Protection Profile is defined as a collection of parameters and options in effect for the SRTP processing. The parameters, which vary according to each profile configuration as described in section 4.1.2 of [9], are:

- the used cipher mechanism (the only one defined as supported is AES 128 in Counter Mode)
- the cipher key length
- the cipher salt length
- the maximum lifetime (a maximum of packets which can be encoded on the same key)
- the authentication function (the only one defined as supported is HMAC-SHA1)
- the authentication key length
- the authentication tag length
- a separate length for the authentication tag used in RTCP traffic securing

The last parameter is not present in all profiles defined by [9], not being a mandatory value for SRTP. Also, as a notable difference, two of the four profiles mentioned in the standard draft are dedicated only for authentication lacking the first parameters (defined as null). Other parameters needed by the SRTP cryptographic contexts are set to their default values.

In the *ServerHello* answer received from the server, only one of the SRTP Protection Profiles will be selected for usage inside the "*use_srtp*" extension. Also, the application should establish some coordination mechanism between the DTLS-SRTP sessions protecting the RTP stream and the ones concerned with the associated RTCP link. In case the peer which acts as a server can not support any profile it will not return the "use_srtp" extension. The standard [9] mentions that in such a case that the "connection falls back to the negotiated DTLS cipher suite" and that if this isn't acceptable "the server should return an appropriate DTLS alert". We understand from

this that the handshake is supposed to continue. There isn't however any clear remark in the standard related with the media stream. Without "use_srtp" exchange completed the SRTP session is not possible, so two paths could be taken at this moment. One of them would be encoding the media stream in the application data DTLS records, and the other implies leaving the RTP stream as it is.

Following a successful handshake, based on the negotiated Protection Profile parameters, key material is generated by the TLS PseudoRandom Function (part of the TLS extractor mechanism[10]), in order to be passed to the SRTP Key Derivation function. The Key Derivation Rate specified by [9] is equal to 0, therefore the keys will not be re-derived based on the SRTP sequence number. The key material generated before applying the SRTP key derivation function consists in four values associated with client master key, client master salt, server master key and server master salt. The first two are provided to the SRTP KDF which will generate the SRTP keys used to encrypt and authenticate packets sent by the client, and the latter two are used in the same purpose on the server side. Of course all keys must be available on both peers, each of them being needed by the opposite peer for decryption. Therefore the derivation procedure takes place on both sides of the DTLS session. Depending on RTP and RTCP multiplexing over the same ports or not, the key derivation is used to obtain both SRTP and SRTCP keys or not. If RTP and RTCP run on different ports separate DTLS-SRTP sessions will be used for the respective packets protection, each of them being intended to provide only the needed keys, one for RTP and one for RTCP.

In the cases of rekeying, a rehandshake is imposed with respect to the DTLS layer. This happens normally at the moment when the negotiated lifetime (included in the Protection Profile) expires. When the maximum number of RTP or RTCP packets is reached according to the mentioned parameter, a new DTLS-SRTP session must be established for the respective stream in order to obtain new keys. This is specified to take place over the already established DTLS channel, consequently the new handshake being protected by the DTLS cipher suite currently in use. In such conditions, [9] states that the DTLS-SRTP implementations should store multiple SRTP key sets in order to avoid problems caused by unordered packet arrival for example dropping delayed packets encrypted with a former key which is not any more in use. More exactly out of order delayed packets encrypted based on the older keys may arrive after rekeying. The choice related to the keys used is easily made if an MKI is used. If no MKI is used in the stream flow then both the new and older sets of keys are tried until the packet can be authenticated.

After the DTLS handshake completion, the SRTP media flow can commence.

Like said before, regarding the transmission, the media stream packets sent are not protected through the usual DTLS record framing mechanism, and instead are passed to the SRTP stack for encryption. Not making use of the DTLS framing, the media stream packets reordering will be handled based on the RTP sequence number.

The reception phase is a bit more complicated since the receiver might get multiple types of packets on the RTP dedicated port. The RTP, DTLS and STUN packets can be differentiated based on the first byte of the packet. For other different packet types, a more advanced demultiplexing mechanism should be designed.

[10] Mechanism used in order to export key material for further use after TLS specific key exchange [7]

One particular case of receiving process is the one which involves multiple DTLS-SRTP associations for the same SRTP incoming port. Such case may appear when a call is forked to multiple destinations. In [9] this case is not described in very much detail regarding the way the DTLS handshake takes place. However it is presumed that more than one DTLS associations are successfully created, being distinguished one from another based on the host and port pair. The problem appears at mapping the correspondent SRTP media streams to the DTLS associations. The media streams are identified by SSRC and not by source host and port, so a mapping between SSRC and the DTLS association must be established. The algorithm which is given as solution in [9] consists in filling an initial empty table of association maintained for each port at the current endpoint, by following the next steps for each packet received:

- if the SSRC present in the packet is already present also in the association table the corresponding DTLS-SRTP association and subsequent the corresponding keying material is used for the packet decryption and authentication
- if the SSRC present in the packet isn't already present in the association table, every DTLS-SRTP association is used in the attempt to decrypt and authenticate the packet
- if the previous step succeeds at some point the decryption attempts are stopped and the corresponding SSRC / DTLS-SRTP association entry is placed in the mapping table
- if the second step fails for every DTLS-SRTP association the packet is discarded
- when a DTLS-SRTP association is closed the related entries in the mapping table should be removed

The algorithm cost is obvious the time spent for decryption and verification times the number of DTLS-SRTP associations established on the receiving port.

There is taken into consideration the case of a source sending corrupt packets. In this situation [9] suggests the possibility of maintaining records of unmapped SSRCs which "consistently fail decryption and verification" in order to abandon attempts of processing these after some limit is reached. Though such a solution is taken into consideration we consider that a possible DoS attack might be attempted in several scenarios. Considering that the SSRC is essentially random chosen for a certain session a malicious attacker can retry the attack establishing a new audio session.

4.3. ZRTP

ZRTP [10] is a protocol based on Diffie Hellman exchange, which can operate only at the media path level in order to provide the necessary security parameters for a SRTP session establishment. In order to avoid MitM attacks the protocol makes use of a series of preshared secrets from previous sessions, and for the initial communication SAS (Short Authentication String) vocal authentication is used. The protocol acts through a RTP media session established by a signaling protocol such as SIP. The actual ZRTP messages are enclosed in a frame which is meant to distinguish the ZRTP packets from normal RTP flow, though keeping them in conformance with the RTP/AVP profile [38]. This way, the ZRTP packets can be exchanged through the media (audio/video) channel, being simply discarded in case of a client incompatible with the protocol, and continuing with a normal unencrypted media communication. As a note, this mode of operation permits also to trigger the ZRTP activation later in the media session and not necessarily at the start of the communication.

In this section we will describe the basic ZRTP operation aspects. A more extensive view on a ZRTP protocol implementation is available in chapter 5 where a use case of implementing security support in a softphone is presented. Also in that part we will get into more detail about advanced ZRTP functionalities like support for unsecuring and resecuring the stream during a call.

4.3.1. Basic ZRTP Operation

The protocol specification establishes two roles in operating the messages exchange: Initiator (we will refer to it as Bob) and Responder (we will refer to it as Alice). A basic display of the typical message flow is described in the following diagram, also figured in the original ZRTP draft:

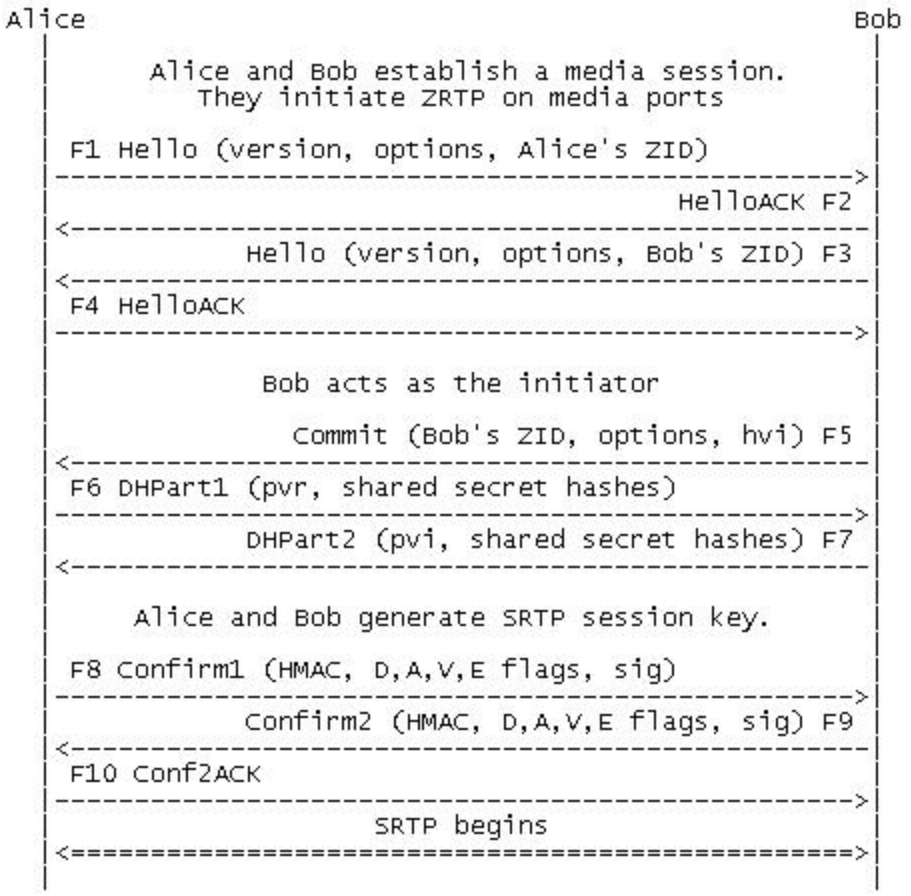


Figure 12

The message flow of the protocol in case of a basic operation can be divided into four phases: discovery, hash commitment, DH exchange and confirmation as also described in [39]. Every message contains a string identifying the message type.

The first phase of the protocol operation is the discovery phase in which the peers determine their capability to support securing through ZRTP. For this part, a *"Hello"* message is sent from each of the participants, and acknowledges are received in the form of *"HelloAck"* messages. The *"Hello"* message contains mostly information related to the capabilities of the sending endpoint such as supported hash algorithms, cipher algorithms, authentication tag types, key agreement types and SAS types.[11]

[11] The minimal set of mandatory cryptographic algorithms that must be supported by any ZRTP compatible endpoint are defined in the last draft version as: SHA-256 for hash algorithms, AES-128 for cipher algorithms, HMAC-SHA1 for authentication tags, DH3k as Diffie-Hellman key agreement type, base32 as rendering scheme for the Short Authentication Strings

Besides the information on supported algorithms, the ZRTP *"Hello"* message also includes details related to the current ZRTP version, an identifier for the vendor of the ZRTP software, and series of flags for various operation modes (such as indicating SAS authentication relaying through a trusted MitM which can usually be a type of PBX[12] software in the call/media path). Also, the majority of ZRTP messages contain hash fields from a hash chain mechanism used to prevent DoS attacks. More exactly a random value H_0 is initially selected by both peers and three hashes are computed as $H_i = hash(H_{i-1})$. Afterwards the obtained values are sent in reverse order in the ZRTP message flow ensuring that a potential attacker can not inject packets due to the impasibility of computing H_{i-1} from H_i.

The information depicted as the ZID in the ZRTP diagram above, is one significant field of the *"Hello"* message, representing a random unique identifier for the endpoint sending the message, identifier that is generated once at the installation of the ZRTP providing solution. This identifier is used in the subsequent steps of ZRTP operation for key indexing, as we will describe in the following paragraphs. The *"HelloAck"* response received for a *"Hello"* message, is nothing more than a simple string acknowledgement without any additional fields, which practically means that the other peer supports ZRTP (the peer recognized the *"Hello"* message sent through the media channel among the usual media packets). According to the standard draft, this phase can even be shortened by skipping the second *"HelloAck"* acknowledgement.

The second phase of the protocol, the hash commitment is the actual phase where the Initiator and Responder roles are established. The Initiator, displayed as Bob in the above diagram, is the peer who first sends a *"Commit"* message. The main information contained inside a *"Commit"* message consists in a selection from the common capabilities exchanged through the discovery phase, which will be used in the securing process, the ZID of the Initiator, which must be the same as the one used in the previous *"Hello"* message, and a hash - *hvi*, which is figured in the above diagram. This is used to ensure the validity of a successful SAS authentication like we will describe further below. The hash is computed on the concatenation between a precomputed *"DHPart2"* message (which can be seen in the diagram as the next message sent by the initiator) and the *"Hello"* message received in the previous phase:

hvi = hash(DHPart2 || Alice's Hello)

The creation in advance of the *"DHPart2"* message implies generating of a fresh Diffie Hellman key pair, by the Initiator, based on the DH algorithm chosen after the selection from the options received through *"Hello"*. We will further refer to the Initiator generated DH pair with: *pvi* - for the public value, *svi* - for the secret value. The generated *pvi* will be part of the *"DHPart2"* message needed for the *hvi* hash. Like stated in the beginning of this section, ZRTP makes use of previously shared secrets in order to avoid MitM attacks. We will refer to these shared secrets as *rs1* and *rs2*. These secrets are retained from a previous secured session with the same peer as the communication partner, and are obtained locally based on the peer's ZID received in *"Hello"*. The ZID

[12] Private Branch Exchange – type of software performing various call related functionalities like routing, proxying and others (similar to software telephonic central for VoIP)

is, as specified, generated once at installation, for every capable ZRTP endpoint and remains unchanged afterwards. The Initiator computes HMAC keyed hashes based on these values, used as keys, and the "Initiator" string. We will refer at the results as *rs1IDi* and *rs2IDi*:

```
rs1IDi = HMAC(rs1, "Initiator")
rs2IDi = HMAC(rs2, "Initiator")
```

These results are included together with *pvi* in the *"DHPart2"* message (also as part of the *"DHPart2"* message are included similar generated key hashes based on other two values which might be provided by the external signaling protocol and by an eventual PBX involved in communication, but these aren't related with the basic level of ZRTP operation which is the current topic).

As the last part of the hash commitment step (sending *"Commit"* containing the *hvi* hash computed over peer's *"Hello"* and the *"DHPart2"*), there must also be mentioned the case when both peers try to act as the Initiator sending *"Commit"* messages. For this case, a simple arbitration takes place by comparing the two hashes (*hvi*) contained in the own and the received *"Commit"* messages. The peer who generated the higher value will play the Initiator role in the further communication.

The third phase is the actual DH exchange. In this phase the Responder (the receiver of the *"Commit"* message) will generate a *"DHPart1"* message the same way the *"DHPart2"* in-advance creation is described above, but of course, based on the Responder's preshared secrets. The Responder, Alice, will generate her own DH pair : *pvr* - public value, *svr* - secret value. The *pvr* value is sent to the Initiator, Bob, along with the *rs1IDr* and *rs2IDr*, obtained as HMAC keyed hashes on the previously shared secrets *rs1*, *rs2* used as keys for the string "Responder":

```
rs1IDr = HMAC(rs1, "Responder")
rs2IDr = HMAC(rs2, "Responder")
```

The initiator, Bob, will respond with the precomputed *"DHPart2"* message. After receiving the DH message, each peer computes the keyed hashes, on its own version of the shared secrets, using these secrets as keys for the other's peer associated string (Bob for "Responder", Alice for "Initiator"). The peers compare the result with the received one. At this point there can result a match, a mismatch, or a match but in other order than the one received. For the last case, a sorting of the shared secrets is kept locally to maintain the same order as the other peer (note that, besides the preshared secrets, the DH messages can contain also shared secrets from other sources which also are the subject of the same verification and reordering). The order is given by the order of the Initiator's secrets. For the case of mismatch a SAS authentication is needed, which part we will detail in a paragraph further below. After the DH exchange a new shared secret will be computed as described next. First, the two peers compute the shared Diffie-Hellman value based on the public values received in the DH messages (*pvi* and *pvr*) and the secret values held (*svi* and *svr*). We will refer to this value as *dhresult*. Another hash value, *mh*,

is computed based on the concatenation of the initiator's “Hello”, “Commit”, and the two DH messages:

mh = hash(Alice’s Hello || Commit || DHPart1 || DHPart2)

Finally, the new shared secret is computed as a hash on a concatenation of *dhresult*, the two ZID values, the *mh* value and the previous shared secrets (eventually reordered) together with their lengths. This is obtained according to a NIST recommendation described in [40], which implies adding a counter (which will be 1), an algorithm ID "ZRTP-HMAC-KDF", and a key derivation context which we will refer to as *KDFcontext* and is formed as a concatenation from the already mention ZIDs and *mh*:

s0=hash(counter||dhresult||"ZRTP-HMAC-KDF"||KDFcontext||previous shared secrets)

KDFcontext = ZIDi || ZIDr || mh

A similar hashing structure including a counter and the *KDFcontext* is used also in other various keys derivation involving HMAC application. For simplicity we will describe these as simple keyed hashes mentioning only the key and the various messages to be authenticated as the main parameters of the HMAC. These are the parameters having an actual value in the protocol analysis described in the next section.

Based on this new secret, which we will name *s0*, is computed the information needed to provide to SRTP for encryption of the media stream such as SRTP master keys and salts. This is done through keyed hashes using *s0* as a key on various strings defined by the ZRTP draft.

One specific case for the DH exchange phase is the one when there aren't any preshared secrets to retrieve (like the case of the first established session for example with a certain peer, when there aren't any secrets associated with the peer's ZID). In this situation random values are used. This way an intruder can't find out from the exchanged information the number of secrets currently shared between the two peers. The random values are discarded at destination. Of course, in this case there will be a mismatch between the exchanged secrets, which situation might also occur in case of a MitM attack. To prevent this, an SAS verbal verification is required by the protocol. This is done by computing a Short Authentication String (SAS) value from the *s0* value computed before. The SAS does not depend on the equality of the shared secrets, the *s0* value which is the base for the SAS computation being calculated as hash on both DH messages - the one locally computed by a peer, and the one received. So, if the cause of the shared secrets mismatch lies in a forgery of one of the DH messages the resulting SAS value will be different due to *“DHpart1”* or *“DHpart2”* being different at the peers. The successfully vocal verification of the SAS (verbally communicating to the other peer the SAS or a value locally derived from it) ensures the MitM absence.

It should be mentioned at this point the importance of the *hvi* value transmitted in the *Commit* message and which computation as a hash between the initiator’s DH message and the responder’s “Hello” we specified above. Without this hash commitment an attempt to disrupt the DH exchange despite the SAS verification by generating a SAS collision. Due to the small length of the SAS a MitM attack could be performed by intercepting the DH messages and generating a pair of DH values to replace them which

would yield the same SAS result. With the *hvi* hash commitment this attack is infeasible as described in [41].

The last phase of the protocol can be considered to be the confirmation phase [39]. At this step, having the new *s0* shared value computed, the initiator and the responder use it to obtain a new shared secret to be stored: *rs0*, by applying *s0* as key in a keyed hash on the string "retained secret".

rs0 = HMAC(s0, "retained secret")

The confirmation phase implies sending two Confirm messages: *"Confirm1"* from the responder (Alice) and *"Confirm2"* from the initiator (Bob) and acknowledging them with two *"ConfirmAck"* messages. These contain a ciphered part composed from several flags indicating various status conditions, the secrets cache expiration interval and also an optional signature. In order to encrypt these information an AES CFB algorithm is used with a key which is obtained from HMAC keyed hashing. We will refer to this as *zrtpkeyi* for the initiator, this being computed as a keyed hash using *s0* as key on the string "Initiator ZRTP Key", and as *zrtpkeyr* for the responder, this being computed also as a keyed hash using *s0* as key, applied to the string "Responder ZRTP Key".

zrtpkeyr = HMAC(s0, "Responder ZRTP Key")
zrtpkeyi = HMAC(s0, "Initiator ZRTP Key")

Also similar keyed hashes using *s0* as key are obtained by the initiator and the responder on the strings: "Responder HMAC Key" and "Initiator HMAC Key"; we will refer to these as *hmackeyR* and respectively *hmackeyI*:

hmackeyR = HMAC(s0, "Responder HMAC Key")
hmackeyI = HMAC(s0, "Initiator HMAC Key")

The Confirm messages contain also the CFB initialization vector to be used in decryption and a keyed hash computed over the encrypted part by using *hmackeyR* and *hmackeyI* as keys. This keyed hash is the *hmac* figured in the protocol flow diagram, and it is used for authentication. The *"Conf2Ack"* confirmation message is simply a string acknowledgement of the end of the confirmation phase. The final successfully message exchange is followed by each peer with the replacement of the local shared secrets *rs1* by *rs0* and *rs2* by *rs1*, the old *rs2* being discarded.

This is, like stated in the beginning of this section, a description of the basic ZRTP operation, in case of a simple single media session based call between two endpoints. The ZRTP draft also defines other modes of operation like multistream mode (more than one media session to secure - like audio and video simultaneous communication), preshared mode for quickstart, or the characteristics needed by ZRTP to operate through a PBX (Private Branch Exchange software).

4.3.2. Analysis of ZRTP

We chose, in the previous section, to describe ZRTP mechanisms in more detail than the other key management solutions, partly for the reason that ZRTP is a protocol relatively easy to be modeled, an attempt in this direction being briefly presented in this section. Despite the impression of more complicated operation, the primitives involved by the protocol phases are simple enough and the solution does not operate at different levels implying an encapsulation layer like DTLS-SRTP. In the same time, it has a more complex message flow than MIKEY for example (there the flow is maximum two messages long), which is a strong reason for verification.

In this chapter, we refer to a verification done with the AVISPA model checker on the basic ZRTP mode of operation presented before. This verification is described in detail in [39], having as target an earlier protocol draft version than the current one, but which doesn't have any consistent differences at the basic level of operation. We reproduce in the next part, the HLPSL [42] code model from [39] which was used as input for verification:

```
role alice(
            A,B : agent,            % Agents
            SECRETS : text,         % A's Secrets
            STRINGA : text,         % A's Capabilities & IDs
            H,HM : hash_func,       % Chosen Hash and HMAC functions
            SEND,RECV : channel(dy)) % Channels

played_by A def=

local State :     nat,              % State
STRINGB :         text,             % B's Capabilities & IDs
SECRETSIDA,SECRETSIDB : message,    % Secret IDs
S0 : message,                       % New secret
COMMITSTRING : text,                % Commit message
HVB,MH : message,                   % Stuff
ZRTPKEYA,ZRTPKEYB: message,         % ZRTP keys
CIPHBLOCKA,CIPHBLOCKB : message,    % CFB
PVA,PVB : message,                  % Public keys
SVA : text                          % A's Secret key

init State := 0

transition

1.    State = 0 /\ RECV(start) =|>
      State' := 2 /\ SEND(hello.STRINGA)

2.    State = 2 /\ RECV(helloack.hello.STRINGB') =|>
      State' := 4 /\ SEND(helloack)

3.    State = 4 /\ RECV(commit.COMMITSTRING'.HVB') =|>
      State' := 6 /\ SECRETSIDA' := HM(SECRETS.responder)
                  /\ SVA':=new()
                  /\ PVA' := exp(g,SVA')
                  /\ SEND(dhpart1.PVA'.SECRETSIDA')

4.    State = 6 /\ RECV(dhpart2.PVB'.SECRETSIDB')
                  /\ SECRETSIDB' = HM(SECRETS.initiator)
                  /\ HVB = H(PVB'.hello.STRINGA) =|>
      State' :=8/\MH':=H(hello.STRINGA.commit.COMMITSTRING.dhpart1.PVA.
                     SECRETSIDA.dhpart2.PVB'.SECRETSIDB')
                  /\ S0' := H(H(exp(PVB,SVA)).SECRETS.MH')
                  /\ secret(S0',s0,{A,B})
                  /\ witness(A,B,bob_alice_s0,S0')
                  /\ ZRTPKEYA' := HM(S0'.rzrtp)
                  /\ CIPHBLOCKA' := new()
                  /\ SEND(confirm1.HM(CIPHBLOCKA').cfbivA.
                     {CIPHBLOCKA'}_ZRTPKEYA')
                  /\ secret(CIPHBLOCKA',ciphA,{A,B})

5.    State=8 /\ RECV(confirm2.HM(CIPHBLOCKB').cfbivB.
                   {CIPHBLOCKB'}_ZRTPKEYB')
               /\ ZRTPKEYB' = HM(S0.izrtp) =|>
   State' := 10 /\ SEND(conf2ack)
                  /\ request(A,B,alice_bob_s0,S0)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
role bob(
            A,B : agent,            % Agents
            SECRETS : text,         % A's Secrets
            STRINGB : text,         % A's Capabilities & IDs
            H,HM : hash_func,       % Chosen Hash and HMAC functions
            COMMITSTRING : text,    % Commit message
            SEND,RECV : channel(dy)) % Channels

played_by B def=

local State : nat,                  % State
STRINGA : text,                     % B's Capabilities & IDs
SECRETSIDA,SECRETSIDB : message,    % Secret IDs
S0 : message,                       % New secret
HVB,MH : message,                   % Stuff
ZRTPKEYA,ZRTPKEYB: message,         % ZRTP keys
CIPHBLOCKA,CIPHBLOCKB : message,    % CFB
PVA,PVB : message,                  % Public keys
SVB : text % B's Secret key
init State := 1

transition

1.    State = 1 /\ RECV(hello.STRINGA') =|>
      State' := 3 /\ SEND(helloack.hello.STRINGB)

2.    State = 3 /\ RECV(helloack) =|>
      State' := 5 /\ SVB':=new()
                  /\ PVB' := exp(g,SVB')
                  /\ HVB' := H(PVB'.hello.STRINGA)
                  /\ SEND(commit.COMMITSTRING.HVB')

3.    State = 5 /\ RECV(dhpart1.PVA'.SECRETSIDA')
                  /\ SECRETSIDA' = HM(SECRETS.responder) =|>
      State' := 7 /\ SECRETSIDB' := HM(SECRETS.initiator)
                  /\ SEND(dhpart2.PVB.SECRETSIDB')

4.    State = 7 /\ RECV(confirm1.HM(CIPHBLOCKA').cfbivA.
                    {CIPHBLOCKA'}_ZRTPKEYA')
                 /\ MH' =
            H(hello.STRINGA.commit.COMMITSTRING.dhpart1.PVA.
                  SECRETSIDA.dhpart2.PVB.SECRETSIDB')
                /\ S0' = H(H(exp(PVA,SVB)).SECRETS.MH')
                /\ ZRTPKEYA' = HM(S0'.rzrtp) =|>
      State' := 9 /\ CIPHBLOCKB' := new()
                  /\ ZRTPKEYB' := HM(S0'.izrtp)
                  /\ SEND(confirm2.HM(CIPHBLOCKB').
                        cfbivB.{CIPHBLOCKB'}_ZRTPKEYB')
                  /\ secret(CIPHBLOCKB',ciphB,{A,B})
                  /\ request(B,A,bob_alice_s0,S0)
                  /\ secret(S0',s0,{A,B})
                  /\ witness(B,A,bob_alice_s0,S0')

5.    State = 9 /\ RECV (conf2ack) =|>
      State' := 11

end role
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role session(
                  A,B : agent,
                  SECRETSA,SECRETSB : text,
                  STRINGA,STRINGB : text)

def=
local SA,SB,RA,RB : channel(dy)
const hello,helloack,commit,dhpart1,dhpart2,confirm1,confirm2,conf2ack,
responder,initiator,rzrtp,izrtp,commitstring,cfbivA,cfbivB: text,
g : nat,
h,hm : hash_func

composition

alice(A,B,SECRETSA,STRINGA,h,hm,SA,RA)
/\ bob(A,B,SECRETSB,STRINGB,h,hm,commitstring,SB,RB)
end role

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

role environment()

def=
const a,b : agent,
secretsab,secretsai,secretsib : text,
stringa,stringb,stringi : text,
s0,ciphA,ciphB,alice_bob_s0,bob_alice_s0 : protocol_id
intruder_knowledge =
{a,b,h,hm,secretsai,secretsib,stringa,stringb,stringi,
hello,helloack,commit,dhpart1,dhpart2,confirm1,
confirm2,conf2ack,responder,initiator,rzrtp,izrtp,
commitstring,cfbivA,cfbivB,g}

composition

session(a,b,secretsab,secretsab,stringa,stringb)
/\ session(a,i,secretsai,secretsai,stringa,stringi)
/\ session(i,b,secretsib,secretsib,stringi,stringb)
end role
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

goal
secrecy_of s0
secrecy_of ciphA
secrecy_of ciphB

authentication_on alice_bob_s0
authentication_on bob_alice_s0
end goal
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

environment()
```

The verification goals set for the model above were the secrecy of *S0*, which will become the new shared secret, the secrecy of the encrypted part of the confirmation messages, and the authentication based on *S0*. The result, as specified by [39] is reproduced below:

```
% OFMC
% Version of 2006/02/13

SUMMARY
SAFE

DETAILS
BOUNDED_NUMBER_OF_SESSIONS

PROTOCOL
/avispa-1.1/testsuite/results/zrtp.if

GOAL
as_specified

BACKEND
OFMC

COMMENTS

STATISTICS
parseTime: 0.00s
searchTime: 144.82s
visitedNodes: 17301 nodes
depth: 16 plies
```

As it can be observed in the modeling, the *secretsab* parameter, which represents the previously shared secrets, isn't added to the intruder knowledge, this meaning that the shared secrets never leave the local peer and consequently aren't known to the intruder. In case of new peers for example, or other case meaning no shared secrets, if we consider the shared secrets as blank, and the situation of new peers known by the intruder, we could add *secretsab* to the intruder knowledge. In that case the verification of the protocol will fail. This is why the protocol specification states that in such a case random values to be used and the mismatch condition to be differentiated from an attack through the SAS verbal verification.

A case of HLPSL modeling and verification through AVISPA on an even earlier version of the protocol which demonstrates the security goals failure in case of absence of SAS verification can be found in [25].

4.3.3. Considerations about ZRTP

We tried to offer only a brief overview of ZRTP in this document, mainly centered on the protocol's basic functionality description and on a verification done on an earlier version of the current ZRTP draft, this being somehow simpler but keeping the main characteristics of the actually more developed protocol. As a conclusion, we saw that ZRTP can be considered safe against a MitM attack, according to the verification of the model, at least at this basic level. However, much more in depth analysis should be performed also on the other ZRTP operation modes, and also on the interaction with the signaling path (which is described as an option by the standard). There already are articles like [43] which describe a possible relay attack on ZRTP and another one [44] which describe possible techniques of enhancing ZRTP through computational puzzles.

Another, more advanced topic, could be the study of running a ZRTP based security solution through a PBX type of software (which can act like a proxy for the calls), this adding some new layers of complexity to the problem. The ZRTP draft describes the rules of interaction in such cases, the protocol having integrated support for relaying through a Private Branch Exchange. Starting from this, there can be studied also the problems that appear in an environment where ZRTP is applied as a solution for securing the media streams from a conference, subject briefly discussed in the ZRTP draft.

One aspect which needs to be underlined is that ZRTP manages to achieve protection acting only at the media layer, and without needing an external solution. Despite the apparent more complicated message flow than in case of MIKEY, this fact makes it the easiest solution to implement, basically integrating it on top of SRTP which uses the same communication channel. We will describe an use case regarding ZRTP integration implemented in this way in an appendix of this document.

As another aspect related with the exclusive media path operation, referring to the classes of robustness established by [26] described in the beginning of the current chapter we consider that another class - *passive-signaling-active-media-detect*, should probably be defined. This is obvious after the ZRTP description, an intruder not having any mandatory reason to act at the signaling level, but only at the media one, and the attack can be detected through the SAS verification. However, such a classification is still debatable, taking into consideration that ZRTP can interact also with the signaling path.

4.4. Other key management solutions

Besides the three dedicated solutions for key management in multimedia based transmissions described throughout this chapter there can be mentioned also other possibilities of ensuring security of the media path.

Probably the most important of these is Security Description with SIPS (SDES)[28] which defines the usage of a cryptographic attribute for unicast media streams. The attribute, which describes a cryptographic key and other parameters that serve to configure security for the media stream, is carried at the signaling path level by SDP providing it to the lower media path. The standard doesn't include a specific description of an encryption mechanism to ensure secure attribute transport, this being provided through TLS [36] for hop-by-hop encryption or S/MIME [45] for end-to-end encryption. The use of SDES is however exposed to a possible attack that may lead to the exploit described in the end of section 3.1.4. As described in [25] the case given is of two legitimate users, Alice and Bob, who previously carried out a VoIP session, which was the subject of passively eavesdrop from an attacker. However, the attacker did not find out the session key, and consequently wasn't able to decrypt the data streams. The example displayed by [25] implies that Bob was the initiator of the session, and to in order ensure confidentiality for the SDES message, S/MIME was used to encrypt the payload. Also it is mentioned by [25] that "S/MIME, in general, is preferred over TLS for protecting SDP messages because (i) S/MIME provides end-to-end integrity and confidentiality protection, and (ii) S/MIME does not require the intermediate proxies to be trusted". However an important drawback and the main cause of the attack is that S/MIME does not provide any anti-replay protection. Because of this, after the end of the original session, the attacker "can replay Bob's original INVITE message to Alice, containing an S/MIME-encrypted SDP attachment with the SDES key transfer message" [25]. Due to the lack of anti-replay support Alice will not be able to detect the similarity of the INVITE request. Given that the key material in the SDP message is the same as in the previous invite and the SRTP derivation function doesn't imply itself a random factor, Alice will derive the same master key and master salt as in the original session, this leading to the potential exploit described by [25] and in section 3.1.4 (though considered by us not to have a very high potential threat due to the subsequent difficulty to obtain the actual stream).

Another mechanism of key management which is based on the use of a new SDP attribute is SDP-DH [26]. This is based on a Diffie Hellman mechanism and is using the same techniques as the above approach (TLS or S/MIME) for MitM protection, though as its introduction implies the "need" for these to be reduced is one of the goals achieved. However the initial standard draft of the solution expired in 2006 without any official further development so we will not get into more detail about its functionality.

EKT [30] is another key management solution. This uses SRTCP, the secure extension of the RTP's companion RTCP media path protocol, whose basic purpose is to provide various media path control and statistic information. However, this is used only for a part of the cryptographic context needed by SRTP, another signaling path key exchange protocol (like MIKEY for example) being used in completion. Given that, similar to the above case the development is currently discontinued we invite an interested reader to consult the referred draft for more information.

5. Conclusion and Future Work

We tried in the current material to offer an overview of the most important solutions involved at the present time in securing the media stream inside VoIP SIP based sessions. Due to the possible extensions, the various practical implementation cases and the continuous changes that occur in the standardization process of the existent specifications at the current time we are convinced that we did not cover the topic entirely, but we hope that we touched the most significant aspects in our presentation.

Besides the media stream protection there are also aspects considering VoIP SIP based sessions security. Some of these are related between them and also with the media path encryption. A quick enumeration following section 26.1 of the SIP standard [1] includes:

- Registration hijacking: this implies the malicious registration of an attacker to a registrar server using the AoR (address-of-record, similar concept to a domain name in DNS or an alias) of the victim; the attacker could forge its message headers to impersonate the victim, delete the current registrar records and replace them with new ones pointing to the attacker SIP devices but having the victim's AoR identifier; this type of attack is easily countered through implementation of an authentication mechanism for the originators of the request

- Server impersonation: in this type of attack the SIP server which might be either registrar or proxy impersonates an honest server and performs malicious operations related to the requests addressed to the respective server; the attack range is quite wide generated by the possible responses offered by the attacker; an example suggested by [1] is routing permanently the requests of honest users to a server designated by the attacker as the replacement for the older one; this case would imply only a redirection response from the impersonated server, the users communicating afterwards with a malicious server that is trusted based on the initial redirection; such attacks can of course be prevented by means of authenticating the server when issuing a request

- SIP message body tampering: this type of attack is included partly in the above presented ones, but also implies other situations like a direct relation with our main topic of media stream securing; more exactly if the SDP part of the message is changed by an attacker this might also imply the change of the carried key related fields, present in solutions like MIKEY or SDES; therefore, like stated actually by the respective standards an external way of protection should cover the security and integrity of the SIP message bodies; we already referred throughout our work to solutions like TLS [7] or S/MIME [45] these being two possible options

- Session tearing down: also an attack based on message forging, this implies malicious sending of BYE messages in order to disrupt a current VoIP session; though apparently this attack would have only lack of communication possibilities as a consequence for an honest target, more advanced scenarios are possible with additional losses; one case would be a combined attack where the intruder acting like a Man in the Middle continuing the session establishment and exploiting it by impersonating the victim after tearing down the session in which the victim is involved; the most effective countermeasure for this attack is the BYE message authentication

- Denial of service: we referred already to DoS attacks in relation with the MIKEY protocol in section 4.1.5; the range of attacks in this case is actually quite large and is related to various companion SIP protocols, either acting at the media path or at the signaling path; one common fact that could be stated is that regarding signaling level due to the operation nature, the Registrar servers are the most likely to be the target of such an attack type, countermeasures being possible to be defined depending on every case of the various existent attacks

In the first appendix we offer an example regarding a ZRTP based media stream securing integration inside an actual softphone available for public use. Related with the example we would like to point out our achievement consisting in the fact that we obtained an interoperable solution with the only two other ZRTP implementation existent in practice for public use, at the moment of the development. A situation of the interoperation tests completed is available online at [55]. This is a case actually not encountered as quite often as it may seem in various softphones built by using the same protocols, many of them being incompatible regarding stream communication even without implying securing it.

The discussed integration brings us to the first possible direction for future development of our current work. This regards interoperation not only limited to endpoints using same signaling protocol like the case of SIP, but also different ones. Such a target involves not only solving various problems related with the software implementations but mainly the analysis and development of possible extensions for the existing standards. Also in this direction we would like to point out that currently exist various implementations of proxying for interprotocol communication, one single example being YATE [48] which offers a SIP – H.323 proxy. However, we are not aware of such a solution that would imply also stream securing functionality. Like mentioned such an implementation would require an extended analysis of the protocols implied and the intercommunication settings. Due to the fact that, disregard of the signaling protocol used, the carrying of the media stream is done using RTP, we think that a SRTP filtering flow implementation would have high chances of success. The problem rises again at the key management level, the majority of the current solutions operating at signaling layer or both at media and signaling paths. Because the signaling protocol is different, this also implying different VoIP architectures and different interactions between the endpoint types, be believe that not all the key management solutions could be adapted. However, ZRTP for example offers the possibility of operation limited to the media path, therefore

it could be a point of start for a more detailed analysis regarding an interprotocol VoIP solution offering media stream securing.

One other interesting direction for further development is ensuring security at media path for conference scenarios. Regarding the key management issue this is currently a problem. The MIKEY standard [8] offers a brief description of two scenarios with limited functionality in section 8, without support for a many-to-many centralized group. DTLS-SRTP [9] clearly states in section 6 that it is intended for use in unicast sessions, but it might also protect group communication through RTP mixers as long as it is centralized. Decentralized mode is not supported by the main standard, being suggested however that future versions of the standard will focus on that topic too. Further development in group support is actually already under progress and described in the companion draft [33]. In case of ZRTP [10] the problem is discussed in section 10 concerning intermediary devices without having as direct target a group conference scenario. However, such an example is given, in centralized mode, the main issue being the ZRTP specific SAS verification which is not functional for a case like that. Therefore, for such scenarios a mechanism to verify the SAS with the bridge or mixer connecting the peers should be designed.

We conclude the current work with the hope that we managed to offer a useful introductive guide into the field of VoIP communication securing that might help any interested researcher and developer to proceed successfully on one or both of the above mentioned directions.

Appendix I. Use Case: Integration of ZRTP in SIP Communicator

In this appendix we will describe a practical example of ZRTP usage, as part of the securing process implemented in a VoIP softphone: SIP Communicator [46]. Our presentation follows the early ZRTP integration steps which took place during the Google Summer of Code 2008 programme as a joined effort of the SIP Communicator team.[13]

In the next section we will present briefly the SIP Communicator as an application describing its functionality. Afterwards we will overview the technologies involved in development, focusing on the ones related with the security integration implementation. In the third section of this appendix we will describe the first part of the ZRTP integration on which we contributed, based on an earlier implementation of SRTP by Su Bing and a close collaboration with Werner Dittman, the author of the single ZRTP library solution available at the integration time. We will focus mainly in this part on our contribution consisting in the library integration, functionality observations and suggestions made on possible library additions.

SIP Communicator Application Functionality Overview

SIP Communicator is a VoIP softphone and instant messaging open source client supporting various protocols such as SIP, Jabber, MSN and Yahoo! Messenger compatibility, and others. The project was initiated at the Louis Pasteur University from Strasbourg.

SIP Communicator is written in Java making use of various frameworks and being based on OSGi architecture [47]. We will get in more detail about this and other technologies involved in the next section.

From the available functionalities we will focus in the following paragraphs on the possibility of using SIP Communicator to carry on secured SIP calls. In order to use the application for communication through SIP a user should follow a few steps.

First an account should be created selecting SIP as desired transport. After this the account can be configured either to use a SIP registrar in order to be able to make calls without using the explicit IP address of the peer, or to be a registrarless account. The registrar server used can be a public available one or a solution deployed in the user's network (or even locally depending on the topology desired).[14] Also the account can be configured for various SIP options like using an additional proxy, the ports used by SIP, the transport protocol to be used (normally UDP), SIP presence options and SIP keep alive options. Also securing or not the SIP calls against interception by default is presented as an option. The only algorithm currently implemented in SIP communicator in this purpose is the ZRTP based one.

Following the account configuration starting a SIP call is fairly easy, the user entering either the address of the peer to be called or its SIP ID (if a registrar is used), and pressing

[13] This appendix is strictly based on the development process which occurred during the mentioned GsoC programme – June-September 2008, the further enhancements not being included in our description.

[14] We tested SIP communicator during development with Brekeke SIP Server[44] and OpenSER[45] as free available SIP registrar software, and Free World Dialup as public hosted solution.

a button to initiate the call. During a call flow the user can perform several actions like pausing, or muting the call, neither of this affecting the call securing support. Also, our contribution included the possibility to activate the call securing during a call and switching back to unsecure mode also during the call, this feature being optional in ZRTP specification. We will describe this addition in the next sections, though it wasn't added to the SIP Communicator functionality due to ZRTP standard inconsistencies.

These would be the most important functions available from the user perspective in order to perform a SIP call in SIP Communicator. The interface is quite simple and intuitive regarding further customization like ringing tones or codecs used. We will focus next mostly on development related issues.

Brief Description of Java Frameworks Used in SIP Communicator

As we already mentioned SIP Communicator is an OSGi [47] based application. The OSGi architecture is a set of specifications defining a "dynamic component system for Java". Essentially OSGi's main target is to offer a collaborative software environment where different components could interwork without having a-priori knowledge about each other. In this purpose OSGi defines a modular system based on bundles, represented in practice by Java JARs, which import and export used packages between them. The communication between bundles is achieved through a service framework. In few words, the bundles use a service registry in order to register services, get services and listen for other services registration or unregistration. The services registered by a bundle represent the point of access to the bundle functionality. The effective mechanism is actually more complicated. An overview of OSGi is available at [47]. What we would like to point out is the advantage that is offered through the modular system for a VoIP solution like SIP communicator. Practically the independence between the bundles and the mode of interaction offers the possibility of adding a new module without disrupting the current functionality. This ensures the easy extensibility for supporting new communication protocols or additions like adding the currently discussed support for call securing. The necessity of a modular framework for development can be observed actually also in other VoIP solutions, the YATE VoIP engine being an example [48].

The actual implementation of the OSGi architecture used in SIP communicator is represented by Apache Felix [49]. Practically an overall description of the SIP communicator code would be that it is comprised from bundles communicating between them through the Apache Felix OSGi framework in order to ensure the various functionalities. In order to obtain access to this framework a bundle must implement a *BundleActivator* interface through which it "starts" meaning essentially that it registers its services exposing the bundle functionalities for other bundles. This is done by calling a *registerService* method from the unique framework instance of a *BundleContext*, practically the service registry. Also using the *BundleContext* a bundle can listen for the registration of other bundles services by implementing the *ServiceListener* interface. Every bundle must be accompanied by a manifest inside in which is defined the *BundleActivator* class, a bundle usually containing more additional packages in which the internal module functionality is implemented. The manifest also defines the packages exported by the bundle and the packages imported. A bundle should export at least the offered services in order to be available in terms of code access from other bundles. The

offered services are usually defined through interfaces. There isn't a required type for a service interface, the bundle using it by getting it from the *BundleContext* through dynamic casting to the imported interface type.

SIP Communicator has a specific bundle for each implemented protocol. The SIP one is based on a version of NIST's JAIN SIP API [50]. More information about the available Java SIP APIs can be found in Appendix 3. Essentially JAIN SIP offers the needed abstraction for modeling the operation of the SIP protocol through classes like *SIPRequest, SIPResponse, SIPTransaction, Message* and others. However our use case of ZRTP integration concerns more the interaction with the media path. The RTP flow in SIP Communicator is ensured through the usage of the JMF RTP API [51]. The Java Media Framework is a solution offered by Sun Microsystems for "incorporating time-based media into Java applications and applets". The RTP API is the part which offers the support for media streaming, JMF offering the needed support also for applications which do not involve network traffic such as usual media players. Two of the main classes defined by the API are the *RTPManager* which is the "starting point for creating, maintaining and closing a RTP session" and the *RTPConnector* which offers the possibility to "abstract the underlying transport for RTP control and data" effectively meaning that it provides the access to the RTP and RTCP flow and the further possibility to modify the packets. We will not get into more detail related to the JMF RTP API for the moment mostly due to the fact that in SIP Communicator this is in process of replacement with a newer solution: FMJ – Freedom for Media in Java [52][15]. However we will describe how the RTP traffic is implemented flow in SIP communicator in the next section.

Before going on to present the integration of ZRTP in SIP Communicator we should mention also *BouncyCastle* [53] as an important library used in the development. The *BouncyCastle* API offers the necessary cryptographic primitives used by the securing mechanisms implied by ZRTP and SRTP, like AES in both of the modes (counter and f8), HMAC and others.

[15] The current latest JMF API version dates from 2001. Though Sun Microsystems still offers support in terms of API documentation, the code source isn't available anymore. FMJ appeared due to lack of further development of JMF, as a third party open source upgrade being compatible in most of it parts with the Sun solution.

Basic Overview of ZRTP Integration in SIP Communicator

This section describes the first phase of ZRTP integration in SIP Communicator.[16] As discussed in section 4.3. ZRTP may operate only related to media path in order to provide the necessary data for the SRTP cryptographic context. The central part of the media flow in SIP Communicator is integrated in the *media* bundle. In the SIP Communicator source tree this can be found in the *service.media* for the exported services, *impl.media* for the implementation and the various included packages. The central class for the media implementation that concerns us is *CallSessionImpl*. This class contains parameters associated with a particular call instance (implemented in the *Call* class) such as the associated audio and video *RTPManagers*.

Connecting the ZRTP Implementation with the Media Bundle

The triggering of media flow securing which in normal conditions is provided by the lower JMF implementation as normal RTP flow is done through the *initializeRtpManager* method in the previous mentioned *CallSessionImpl* class. In this method an RTP Manager responsible with sending or receiving an RTP stream is initialized. In normal conditions this does not involve more than buffer configuration and event listening. However, in order to secure the stream, direct access to the RTP packet flow is needed. As stated in the previous section this is finally achieved through the *RTPConnector* class. This is done by initializing the *RTPManager* to use different connectors than the default ones. Essentially the connectors used in our implementation have the role to trigger a filter for the normal RTP packets. This “filter” ensures the ZRTP packet exchange. This is performed by the ZRTP4J library and takes place through the same RTP channel as the stream being secured being followed, if successful, by SRTP flow instead normal RTP. The “filter” mechanism, comprised actually by several classes contained in the *media.transform* and *media.transform.srtp* packages, ensures the SRTP encoding before sending the RTP packets through the network and SRTP decoding before passing them to the upper layers when receiving.

The base classes and interfaces included in the *impl.media.transform* package (which is part of the initial SRTP implementation work done prior to our ZRTP integration) can be used as a RTP packet transform base for any protocols that will "flow" through the same channel as RTP does. Therefore we could say this is the core of our “filter” mechanism, or better said our “filter” mechanism is derived based or uses directly the classes from this package.

The generic *TransformManager* class included here has the role to create a *TransformConnector* type class specific to the used transformation protocol. This is done based on the *SessionAddress* object and a protocol specific *TransformEngine*, passed as arguments in the constructor, the *TransformEngine* being further responsible with the actual stream filtering.

[16] The ZRTP implementation in SIP communicator supported also various modifications after the first integration phase these not being described entirely in this document.

The *TransformConnector* obtains the RTP/RTCP input/output streams from the sockets got from the *SessionAdress* object.

These streams have the type of *TransformInput/OutputStream* and their creation implies also the use of a *PacketTransformer*, passed as argument in the constructor, which is obtained from the specific protocol *TransformEngine*. The effective transformation of the packet is called in the streams *read*, respectively *write* methods and implies the call of the method provided by the stream's associated implementation of the *PacketTransformer*.

As mentioned, the specific *PacketTransformer* for a certain protocol is provided by a specific *TransformEngine* and provides specific transformation routines.

In case of SRTP all these and the other SRTP specific classes are included in the *impl.media.transform.srtp* package.

Among these there is the *SRTPCryptoContext* created in the SRTP engine which holds the SRTP specific data. The context is accessible from the specific *PacketTransformer*. This calls in its *transform* and *reverseTransform* functions other implementations of transformation routines provided in the associated context class. Here, in the associated context, finally the RTP/RTCP packet is passed through the actual transformations.

The last part in the above description refers to the SRTP transformation and how is implemented based on the generic *transform* package classes. However, for the first part, the mechanism also applies in the initiation of ZRTP exchange which takes place of course before the SRTP flow. For a better understanding we reproduce the portion of code that instantiates stream security in the next lines[17]:

```
private void initializeRtpManager(RTPManager rtpManager,
                                  SessionAddress bindAddress)
        throws MediaException
    {

        /* Select a key management type from the present ones to use
         * for now using the zero - top priority solution (ZRTP);
         */
        selectedKeyProviderAlgorithm = selectKeyProviderAlgorithm(0);

        try
        {
            // Selected key management type == ZRTP branch
            if (selectedKeyProviderAlgorithm != null &&
                selectedKeyProviderAlgorithm.getProviderType()
                    == KeyProviderAlgorithm.ProviderType.ZRTP_PROVIDER
                    && rtpManager.equals(audioRtpManager))
            {
                // Set a ZRTP connector to use for communication
                TransformConnector transConnector = null;

                TransformManager.initializeProviders();

                transConnector = TransformManager.createZRTPConnector(
```

[17] The source code listings date from the early stage of ZRTP integration, consequently being possible to be different in the current SIP Communicator version

```
                                bindAddress,
                                "BouncyCastle", this);
        rtpManager.initialize(transConnector);
        this.transConnectors.put(rtpManager, transConnector);

        // ZRTP engine initialization

        ZRTPTransformEngine engine
            = (ZRTPTransformEngine)transConnector.getEngine();

        engine.setUserCallback(new SCCallback(this));

        // Case 1: user toggled secure communication prior to
        // the call
        // call is encrypted by default due to the option set
        // in the account registration wizard
        if (this.getCall().isDefaultEncrypted())
        {
            if (engine.initialize("GNUZRTP4J.zid"))
            {
                usingSRTP = true;
                engine.sendInfo(
                    ZrtpCodes.MessageSeverity.Info,
                    EnumSet.of(
                    ZRTPCustomInfoCodes.ZRTPEnabledByDefault));
            }
            else
            {
                engine.sendInfo(
                      ZrtpCodes.MessageSeverity.Info,
                      EnumSet.of(ZRTPCustomInfoCodes.
                                  ZRTPEngineInitFailure));
            }

        }
        // Case 2: user will toggle secure
        // communication during the call
        else
        {
                engine.sendInfo(
                      ZrtpCodes.MessageSeverity.Info,
                      EnumSet.of(ZRTPCustomInfoCodes.
                                  ZRTPNotEnabledByUser));
        }

        logger.trace(
            "RTP"
            + (rtpManager.equals(audioRtpManager)?" audio"
                :"video")
            + "manager initialized through connector");
    }
```

```
            else
            // Selected key management type == Dummy branch - hardcoded
keys
            if (selectedKeyProviderAlgorithm != null &&
                selectedKeyProviderAlgorithm.getProviderType() ==
                KeyProviderAlgorithm.ProviderType.DUMMY_PROVIDER
                && rtpManager.equals(audioRtpManager))
            {
                SRTPPolicy srtpPolicy = new SRTPPolicy(
                            SRTPPolicy.AESF8_ENCRYPTION, 16,
                            SRTPPolicy.HMACSHA1_AUTHENTICATION, 20, 10,
                              14);

                // Master key and master salt are hardcoded
                byte[] masterKey  =
                    {0x00, 0x01, 0x02, 0x03, 0x04, 0x05, 0x06, 0x07,
                     0x08, 0x09, 0x0a, 0x0b, 0x0c, 0x0d, 0x0e, 0x0f};

                byte[] masterSalt =
                    {0x10, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x17,
                    0x18, 0x19, 0x1a, 0x1b, 0x1c, 0x1d};

                TransformConnector transConnector = null;

                TransformManager.initializeProviders();

                // The connector is created based also on the crypto
                //services
                // provider type;
                transConnector =
                    TransformManager.createSRTPConnector(bindAddress,
                                                         masterKey,
                                                         masterSalt,
                                                         srtpPolicy,
                                                         srtpPolicy,
                                                        "BouncyCastle");

                rtpManager.initialize(transConnector);
                this.transConnectors.put(rtpManager, transConnector);

                logger.trace("RTP"+
                        (rtpManager.equals(audioRtpManager)?" audio "
                        :"video")+
                        "manager initialized through connector");

            }
            // No key management solution – unsecure communication
            // branch
            else
            {
                rtpManager.initialize(bindAddress);

                logger.trace("RTP"+
                        (rtpManager.equals(audioRtpManager)?" audio "
                        :"video")+
                        "manager initialized normally");
            }
```

```
        }
        catch (Exception exc)
        {
            logger.error("Failed to init an RTP manager.", exc);
            throw new MediaException("Failed to init an RTP manager."
                                     , MediaException.IO_ERROR
                                     , exc);
        }

.....
}
```

First a key management algorithm is selected. For the moment the only solution implemented in SIP Communicator is ZRTP. If no key management is selected SRTP with hardcoded keys (practically a test for encryption) can be used. This takes place as described, and is figured in the last part of the code above. If ZRTP key management is selected, ZRTP exchange will take place for the RTP manager corresponding to the audio stream.[18] First of all a ZRTP connector is created for the current call session. As it can be observed in the instantiation process the *BouncyCastle* instantiation is passed as an argument counting for a chosen cryptographic algorithms provider (also other possible providers could be used if following the JCE specifications [54]). Following this, the ZRTP connector is attached to the current RTP manager and memorized in a map which maintains the associations between the RTP managers and RTP connectors. The code for the described function is the next:

```
public static TransformConnector createZRTPConnector
      (SessionAddress addr,
       String cryptoProvider,
       CallSession callSession)
        throws InvalidSessionAddressException
    {//for adding multistream support the engine should be instantiated
    //once as a static variable of this class and passed to every ZRTP
    //connector as a parameter
        ZRTPTransformEngine engine = new ZRTPTransformEngine();

        Provider cp = selectProvider(cryptoProvider);

        TransformConnector connector = null;

        connector = new ZrtpTransformConnector(addr, engine);

//for adding multistream support this method should be replaced with
//an addConnector, which should add the connector to an internal engine
//connector array; supporting multistream mode by the engine implies
//the proper management of this connector array - practically every
//stream has it's own connector
        engine.setConnector(connector);

        engine.setCryptoProvider(cp);

        return connector;
       }
```

[18] In the latest version of SIP Communicator multistream securing, both audio and video was implemented.

The most important part is that from the ZRTP connector a *ZRTPTransformEngine* instance is finally obtained. The *ZRTPTransformEngine* is the class that makes the connection between the SIP Communicator and the ZRTP functionality included in ZRTP4J library. Practically from here, and trough this class, are called the ZRTP operations and the feedback is received back to SIP Communicator. We will not get into more details due to the length of the code, this being accompanied by comments. However we will refer in the next section to the ZRTP4J library and the ZRTP specific functionality.

The next important operation performed by the RTP manager initialization function is associating a user callback class instance to the engine. This user callback class has the role to make the connection between the user triggered actions and the ZRTP implementations on one direction, and in the opposite way, between the ZRTP implementation events and the triggered actions in SIP Communicator. The *SCCallback* class went through multiple changes during the development process, in the current version being replaced with other means of feedback control. However in the present version the user does not have the possibility to trigger call securing on and off during the call. We present a version of *SCCallback*, part of our work, included in the *transform.zrtp* package:

```
public class SCCallback
    extends ZrtpUserCallback
{
    private static final Logger logger
    = Logger.getLogger(SCCallback.class);

    private CallSession callSession = null;

    private SecurityGUIListener guiListener = null;

    private CallParticipant participant;

    /**
     * The class constructor.
     */
    public SCCallback(CallSession callSession)
    {

        this.callSession = callSession;

        guiListener = callSession.getCall().
                        getSecurityGUIListener("zrtp");
        Iterator<CallParticipant> participants = callSession.getCall().
                                                 getCallParticipants();
        if (participants.hasNext())
            participant = participants.next();

    }
```

```
public void init() {
    SecurityGUIEvent evt = new SecurityGUIEvent(participant,
            SecurityGUIEvent.NONE,
            SecurityGUIEvent.SECURITY_ENABLED);

    logger.info("initialize SCCallback");

    fireStateChangedEvent(evt);
}

private void fireStateChangedEvent(SecurityGUIEvent evt) {
    if (guiListener != null) {
        guiListener.securityStatusChanged(evt);
    } else {
        guiListener = callSession.getCall().
                            getSecurityGUIListener("zrtp");
        if (guiListener != null) {
            guiListener.securityStatusChanged(evt);
        }
    }

}

public void secureOn(String cipher)
{
    logger.info("Cipher: " + cipher);
    HashMap<String, Object> state = new HashMap<String, Object>(3);

    state.put(SecurityGUIEventZrtp.SESSION_TYPE,
              SecurityGUIEventZrtp.AUDIO);
    state.put(SecurityGUIEventZrtp.SECURITY_CHANGE, Boolean.TRUE);
    state.put(SecurityGUIEventZrtp.CIPHER, cipher);

    SecurityGUIEventZrtp evt = new
                      SecurityGUIEventZrtp(participant, state);
    fireStateChangedEvent(evt);
}

public void showSAS(String sas, boolean verified)
{   logger.info("SAS: " + sas);
    HashMap<String, Object> state = new HashMap<String, Object>(3);

    state.put(SecurityGUIEventZrtp.SESSION_TYPE,
              SecurityGUIEventZrtp.AUDIO);
    state.put(SecurityGUIEventZrtp.SAS, sas);

    if (verified) {
        state.put(SecurityGUIEventZrtp.SAS_VERIFY, Boolean.TRUE);
    }
    else {
        state.put(SecurityGUIEventZrtp.SAS_VERIFY, Boolean.FALSE);
    }
    SecurityGUIEventZrtp evt = new
                      SecurityGUIEventZrtp(participant, state);
    fireStateChangedEvent(evt);
}
```

```
    public void showMessage
                  (ZrtpCodes.MessageSeverity sev, EnumSet<?> subCode)
    {
        Iterator<?> ii = subCode.iterator();
        Object msgCode = ii.next();
        logger.info("Show message sub code: " + msgCode);
    }


    public void zrtpNegotiationFailed
            (ZrtpCodes.MessageSeverity severity, EnumSet<?> subCode)
    {
        Iterator<?> ii = subCode.iterator();
        Object msgCode = ii.next();
        logger.warn("Negotiation failed sub code: " + msgCode);
    }

    public void secureOff()
    {
        logger.info("Security off");

        HashMap<String, Object> state = new HashMap<String, Object>(2);

        state.put(SecurityGUIEventZrtp.SESSION_TYPE,
                  SecurityGUIEventZrtp.AUDIO);
        state.put(SecurityGUIEventZrtp.SECURITY_CHANGE, Boolean.FALSE);

        SecurityGUIEventZrtp evt = new
                         SecurityGUIEventZrtp(participant, state);
        fireStateChangedEvent(evt);
    }

    public void zrtpNotSuppOther()
    {
        logger.info("ZRTP not supported");
    }

    public void confirmGoClear()
    {
        logger.info("GoClear confirmation requested");
    }
}
```

We have chosen to present this older version of *SCCallback* because it is easy to observe the interaction between the internal ZRTP implementation and the functionalities supported at the user level. The *SCCallback* class makes use of a series of events to change the state either of the application GUI, or the call flow, or both. These events are triggered through the methods available in the class, which methods are called on their own from the mentioned before *ZRTPTransformEngine* class which makes the connection between the ZRTP4J library and the SIP Communicator code. The methods available here trigger at the next moments:

- *init* : when the ZRTP engine is initialized
- *secureOn* : when the user triggered stream security activation through ZRTP and the ZRTP initial negotiation completed successfully resulting in the start of the SRTP flow
- *showSAS* : when the SAS string is established and the peer can verify it with the call partner
- *secureOff* : when the user triggered the optional ZRTP GoClear action through which it is possible to get back to normal stream
- *zrtpNegotiationFailed* : when ZRTP exchange failed for some reason
- *zrtpNotSuppOther* : when ZRTP exchange failed due to lack of support from the other peer
- *confirmGoClear* : when the other peer decided to switch back to normal RTP stream, and the user must confirm the switch (action not implemented in this version of the callback class)

Again, we would like to point out that we described the callback class only with the reason to show the possible events and actions that might occur, the actual implementation being changed in part over time. Therefore we will not get into more detail regarding the event mechanism processing.

Getting back to the initial RTP manager initialization function, after attaching a callback class to the engine (note that different callbacks could be attached), there are two branches, one where ZRTP exchange activation is triggered by default for a call, and one where this is supposed to be triggered later during the call. In the example given only the first one is implemented. The second, though tested successfully in a development phase caused inconsistencies in ZRTP exchange, part of them being based on the lack of standard specification and consequently was removed. In the part that is implemented it can be observed the ZRTP engine initialization using the peer's ZID value. The initialization of the engine effectively starts the ZRTP exchange from the ZRTP4J library.

The ZRTP4J Library

The ZRTP4J library used in our version of ZRTP integration is based on libzrtpcpp, both being developed by the same author, Werner Dittmann, and being at the moment of our integration the only available solutions of ZRTP libraries. We tried during our integration process to enhance the functionality of the ZRTP library, and though successfully tests were performed, we chose not to integrate the respective additions due to some ZRTP standard inconsistencies. However we consider our development an important step for future extended support and we will try to discuss it in the next section after a short overview of the main components of the ZRTP4J library.

The central class of the ZRTP4J library is *ZRtp*. In this class take place all ZRTP parameter related actions like verifying the SAS, initiating message processing and others. A close companion of the *ZRtp* class, and probably the most important class in the ZRTP4J library is the *ZrtpStateClass*. This is the class that models the ZRTP protocol

state machine. Based on the received ZRTP messages or events like error state, closure or timer expiration handling done in this class the ZRTP implementation passes from one protocol state to another, the defined states being:

- *Initial* – the first state where the ZRTP state machine is placed at initialization
- *Detect* – the state where the ZRTP protocol engine sent the initial *Hello* packet and awaits to find out if the peer supports ZRTP
- *AckDetected* – the state where the *Hello* acknowledgement was received in the Detect state and the other peer's own *Hello* packet must arrive according to the ZRTP flow
- *AckSent* – the state where a *Hello* acknowledgement was sent to the other and our peer should send its own *Hello*
- *WaitCommit* – the state where a *Commit* packet is supposed to be received from the Initiator
- *CommitSent* – the state where a *Commit* has been sent and the first DH message should be received from the Responder
- *WaitDHPart2* – the state where the first DH message has been sent and the second one is expected to be received from the Initiator
- *WaitConfirm1* – the state where the Initiator waits the first *Confirm* packet
- *WaitConfirm2* – the state where the Responder waits the second *Confirm* packet
- *WaitConf2Ack* – the state where the Initiator waits the *Confirm* acknowledgement packet
- *WaitErrorAck* – the error condition state where an error packet was sent and a confirmation is expected

Note that the description for which state denotes only the very basic behavior some of the states being reachable in both Initiator and Responder roles and the receiving of other messages than the mentioned ones could cause different replies. For an extensive description of the state machine please consult the ZRTP4J source code which contains detailed comments for each state.

The ZRTP4J library also includes the next other classes and interfaces:

- the *ZrtpCallback* interface which is used to ensure the communication between an external application like SIP Communicator and the ZRTP4J library core; this is implemented by the *ZRTPTransformEngine* class which we mentioned in one of the above paragraphs
- the *ZrtpCodes* interface, containing codes for informational messages sent from the ZRTP4J library to the external application using it
- the *ZrtpConstants* class, including various constants used by the ZRTP implementation like ZRTP packet header fields for example
- the *ZrtpSrtpSecrets* class, which essentially maintains the state of the negotiated keys which are the subject of the ZRTP exchange
- the *ZrtpUserCallback* abstract class, which represents a default behavior of the final user notification and interaction part, one implementation being the *SCCallback* version presented before in this section

Besides these the ZRTP4J library includes a package containing ZRTP packet type classes for each possible ZRTP packet, a package maintaining a few utility classes used for common operations and a package for ZID management.

The GoClear – GoSecure Addition

One part of the ZRTP draft, on which we did not insist in the ZRTP presentation in section 4.3, is the *GoClear* possibility to switch back to unsecure RTP transfer mode. This could be used for example when call securing is needed only for certain duration of the call. The essential idea is that the draft offers an optional possibility to switch back and forth from secure traffic to unsecure traffic. In order to obtain this functionality we made several additions on the ZRTP4J library.

The specific packet structure classes added following the ZRTP specifications were *ZrtpPacketGoClear* and *ZrtpPacketClearAck.* This are simple classes for the GoClear and the ClearAck messages.

Several events were added to be managed by the ZRTP state machine in the *ZrtpStateClass: ZrtpGoClear* to signal GoClear send request to the state engine and *ZrtpGoSecure* for signaling switching back to secure mode after GoClear.

The main modification implied states added or, for existing ones, modified handling in the same ZRTP state machine *ZrtpStateClass*:

- *WaitClearAck* - was already added to the state enumeration but with empty events handling; This state handles the *ClearAck* message that acknowledges the successful processing of *GoClear*. Only the *GoClear* initiator can enter this state. From here the unsecured mode is switched on and the state engine can reach the next defined state *UnsecuredState*. Possible events in this state are the timeout for sent *GoClear* packet which causes a resend and the receiving of *ClearAck*. The latter means that *GoClear* was received and checked by peer, and triggers the switch to the unsecure mode

- *UnsecuredState,* the state in which the engine gets after a successful GoClear transaction; This state handles the Commit message sent by the peer after switching back to secure mode from unsecured, and also a possible GUI request sent directly by the user to initiate this back-to-secure-switch. In case of the *Commit* packet the action taken is to prepare and send a new *DHPart1* message to initiate a new ZRTP exchange and the *WaitDHPart2* state is entered. In case a "GoSecure" request is received from GUI the current peer is the one which prepares and sends the *Commit* to switch to the call secure mode and after that enters the *CommitSent* state

- *SecureState* which models mainly the GoClear related events received after call securing; This state handles either the second *Confirm* or *GoClear* messages received from peer or an eventually "GoClear" request from the GUI to initiate the GoClear procedure. If a second *Confirm* packet is sent this is the case where the *Confirm* acknowledgement wasn't received by the other peer and causes a resending of it and keeping the secure state. If a *GoClear* packet is received a *ClearAck* is sent after the user is announced by the peer's intention to unsecure the call. If the user confirms the peer's request, the switch to *UnsecuredState* occurs, else the *SecureState* is kept. If a "*GoClear*" GUI request is received directly from the user, a *GoClear* message is sent to the other peer (if supported) and the state is switched to *WaitClearAck.*

Like the previous states description this is only a basic functionality overview, more details being available in the source code comments.

Considering a secured call between peer A and peer B as an example, in order to switch to unsecure mode let us presume user A will click a padlock button (displayed as active).[19] This triggers the sending of a *SecureEvent* handled in *CallSessionImpl*, which finally is handled by the *ZRTPChangeStatus* function inside this class too. This will call *requestGoClear* from the *ZrtpTransformEngine*. The request is forwarded to the ZRTP4J *Zrtp* class function with the same name. Here a new *ZrtpGoClear* event is created and sent for processing to the *ZrtpStateClass* state engine.

The event will be passed to a switch redirecting it to a specific method according to the current state of ZRTP. In this case the *ZrtpGoClear* will reach the branch for the *SecureState.* According to the above description of the added states this will trigger the *GoClear* procedure intiation. First, after receiving the *ZrtpGoClear* request, it is checked if GoClear-GoSecure transition is accepted for the current call. This is based, according to the ZRTP draft [10] section 5.7.2 on the Allow Clear flags sent in the Confirm1 and Confirm2 messages during the first call securing. For our tests we set the flag sent to be true but this might be probably enhanced with a custom GUI setting option from the user to allow or not allow unsecuring the call. Both, the sent and received flags need to be true in order to permit GoClear - this is checked before going forward inside the *ZrtpGoClear* request processing branch, with a simple AND between the flags, performed by the *Zrtp* class *goClearAccepted* new added method. If the check returns true, *prepareGoClear* is called from the *Zrtp* class. The *prepareGoClear* method, like the other prepare methods, has the final role to return the GoClear packet, to be sent from the state engine. At this point, there might be a possible optimization of preparing the GoClear packet in advance, the preparation implying a hashing operation that could take place immediately after having the HMAC keys, or after the call is secured.

After the packet is sent, according to the ZRTP draft, the SRTP stream sending should be stopped. For this part, the *afterFirstGoClearSent* function from *Zrtp* class is called from the state engine, which forward calls *stopStreaming(true)* from *ZrtpCallback.* We added this function to the callback interface because there is needed a method at various points in the GoClear-GoSecure interface in order to start and stop the media

[19] This is actually the case of one of the interface versions implemented during the development, one padlock button being used to trigger call securing, switching back to unsecure mode and permanently displaying the call state – active for secure and inactive for unsecure.

stream. In case of the *ZrtpTransformEngine* the implementation of this method sets a *holdFlag*. This flag is checked in the *transform* method responsible with SRTP securing initiation and passing the packet further, and if found to be true a normal packet is returned instead of a transformed packet. In this case, the connector's instance of *TransformOutputStream* class which calls *transform*, returns the unmodified packet's length without sending it anymore (practically the packets are dropped). This probably is not the best solution for temporarily stopping the SRTP/RTP stream, but this permits to continue sending ZRTP packets "inside" the same stream which is needed – (the *holdFlag* check to generate the packet dropping in the *transform* method from the *ZrtpTransformEngine*, is done only in case the packet is not a ZRTP packet). After stopping the media stream, a timer is started for GoClear resend as the draft specifies, and the current engine state is switched to *WaitClearAck*.

Due to the code length we prefer to summarize the above described actions for better understanding through the next figure, instead of providing a code listing:

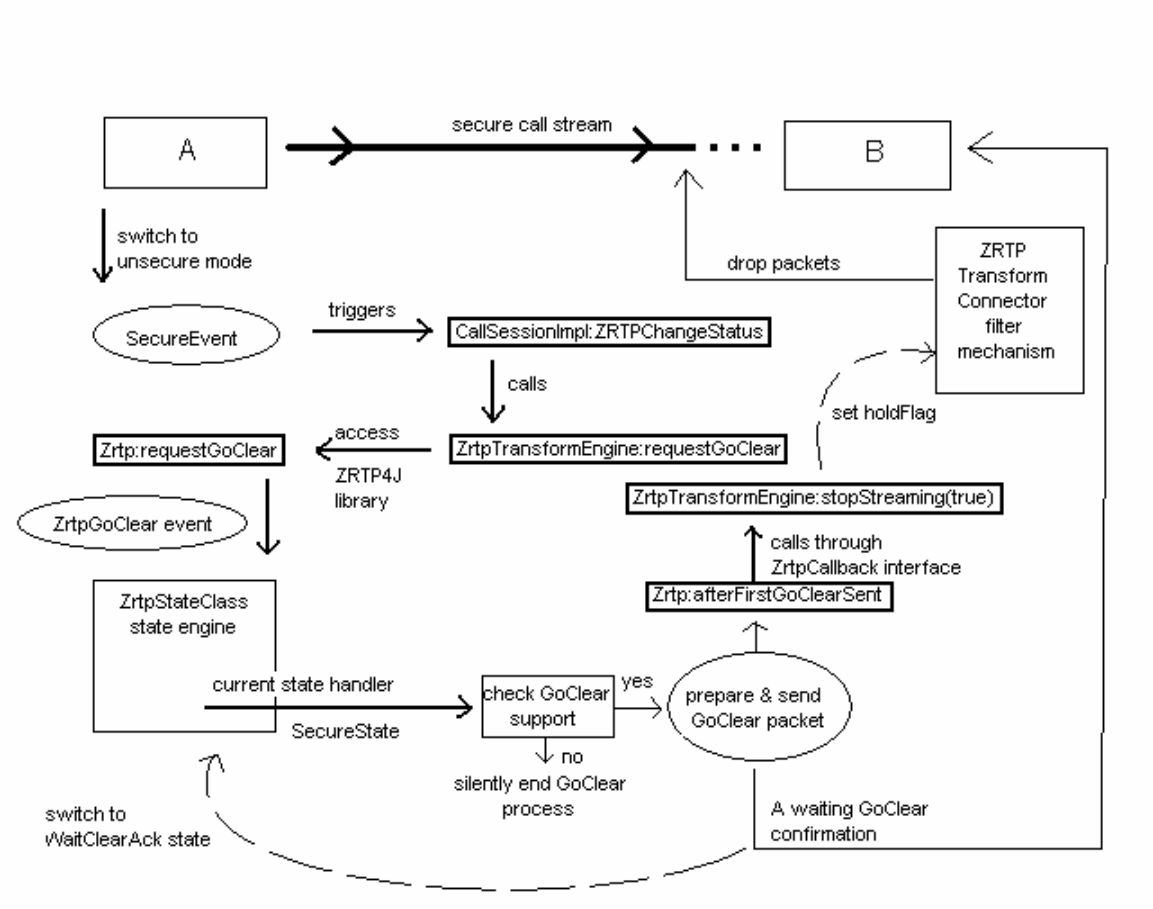


Figure 13

When the GoClear message is received by B this is signaled to the state engine as a normal *ZrtpPacket* event which is processed only in the *SecureState*, based on a packet type check, as in the case of other packets. So, after peer B receives the packet from peer A, it generates the response packet *ClearAck* by calling *prepareClearAck* from the *Zrtp*

class. This method performs several tasks. It checks the hash of the packet, stops the audio streaming as the other peer did and as the ZRTP draft specifies, clears the SRTP secrets by calling *clearSecrets*(), deletes the srtpTransformers from the *ZrtpTransformEngine* by calling *srtpSecretsOff* in order to switch to normal RTP traffic and returns a pre-created ClearAck packet to the state engine. The GoClear message receiving process is continued by the state engine with sending this ClearAck message. After this, the *handleGoClear* method is called from *Zrtp*, which forwards the call to *ZrtpCallback (ZrtpTransformEngine*) which again forwards it to the user callback which displays an announcement that the peer turned off the secure channel. The ZRTP draft says, we quote: *"The endpoint then renders to the user an indication that the media session has switched to clear mode, and waits for confirmation from the user."* We implemented this by a displaying a popup only offering an OK button. However, we are not entirely certain that "confirmation" above refers to such a simple announcement like this, or also needs providing the denial option to the user, but the draft doesn't specify exactly what should happen if the user says "no" so we opted for the announcement. After the user confirms by closing the message box and the stack of called functions start returning, the audio stream is resumed as normal RTP from the *handleGoClear* in the *Zrtp* class. Finally, switching to *UnsecuredState* in the state engine concludes this part. The described process is displayed in the next figure:

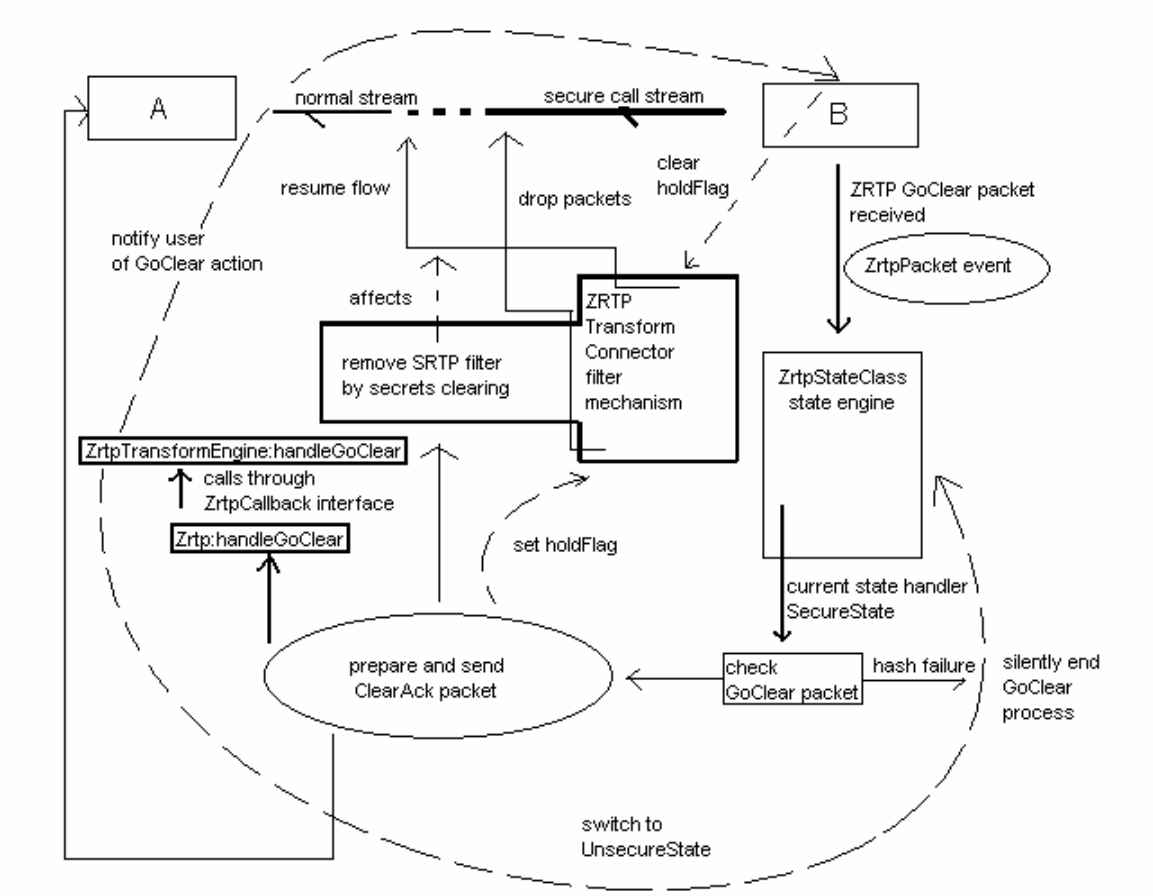


Figure 14

While the above actions take place at peer B, peer A was left in *WaitClearAck* state after sending GoClear. This state can receive two events (besides the default Close/Error branch). First of them is the response – ClearAck – sent by peer B. This is processed as a ZrtpPacket event by the state engine and goes on like this: the GoClear timer is cancelled and after this the *goClearOk* function is called from the parent *Zrtp* class. This method calls the next functions*: srtpSecretsOff* which deletes the srtpTransformer filters, *clearSecrets* described above which deletes the SRTP related secrets and *stopStreaming(false)* which resumes the media stream as RTP from being stopped when sending GoClear. Finally the state engine switches to *UnsecuredState*. The conclusion of the GoClear mechanism is depicted in the figure below:

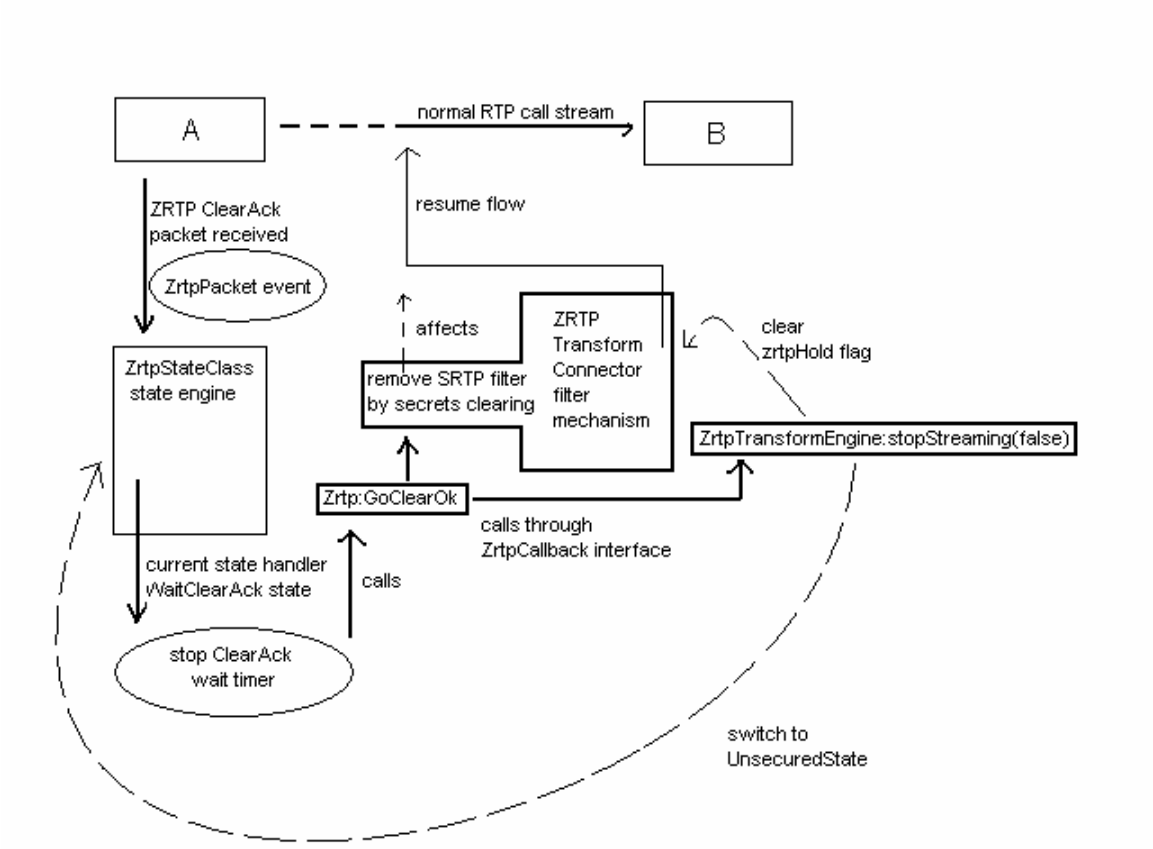


Figure 15

The other possible event in *WaitClearAck* state is the Timer event triggered by a delay in receiving ClearAck after sending GoClear. In this case the GoClear is resent. If the timer expires, the *clearSecrets* function is called to delete the SRTP secrets as specified by the ZRTP draft. Here is another point where the ZRTP draft is not very specific about the state where the peer must be left in case this happens, or if the call must be ended. At the first sight it would be reasonable if not ending the call to leave it in *UnsecuredState,* but not receiving ClearAck means probably that the other peer didn't get

our GoClear so it would continue to try decoding our packets and sending coded stream. Regarding this, the state should probably remain *SecureState* and the srtpTransformer filters kept. However taking into consideration that the secrets are deleted this also would not make much sense. Above all, there is also the fact that at this point the sending stream is stopped, and the draft doesn't say if it should be resumed or not.

We also tried to implement the reverse “GoSecure” part. The draft states secure off - on switches should be possible alternatively during a call in case GoClear operation is supported. We detail in the next paragraphs an overview of the mechanism implemented, which is practically making use of almost the same code flow suggested by the prior description and diagrams but of course in reverse order.

Both A and B peers are in *UnsecuredState* after a successful GoClear exchange. Let us consider that peer A wants to switch back to secure state after a while. In order to do this the user will press the padlock button again (this should be showing secure off at this moment). The way from GUI to the state engine is quite similar as in the GoClear case as stated before, starting with a *SecureEvent* handled in the *CallSessionImpl* and going forward as described in that case until passing a *ZrtpGoSecure* event, finally to be processed in the event handling function of the *UnsecuredState*. Here a Commit packet is prepared and sent pretty much the same as in case of sending the Commit packet when a HelloAck is received in the *AckSent* state.[20] The Commit packet is generated based on the peer's Hello which was saved when the call was first secured (in *Detect* and *AckDetected* states). The state engine goes into *CommitSent* state after sending the packet and starting a delay timer while waiting for the response.

The Commit sent above by peer A is received by peer B, which after the GoClear should also be in *UnsecuredState.* For this reason the event handling function for this state has also a *ZrtpPacket* event branch dedicated for Commit processing. An own Commit packet is generated here as in the *AckDetected* state. After this part the process flow goes again quite like in normal ZRTP securing flow, for the Commit received in the *WaitCommit* state a DHPart1 message being prepared, sent, and the state being switched to *WaitDHPart2*.

From this moment the state engine enters the normal transitions as when securing the call for the first time.

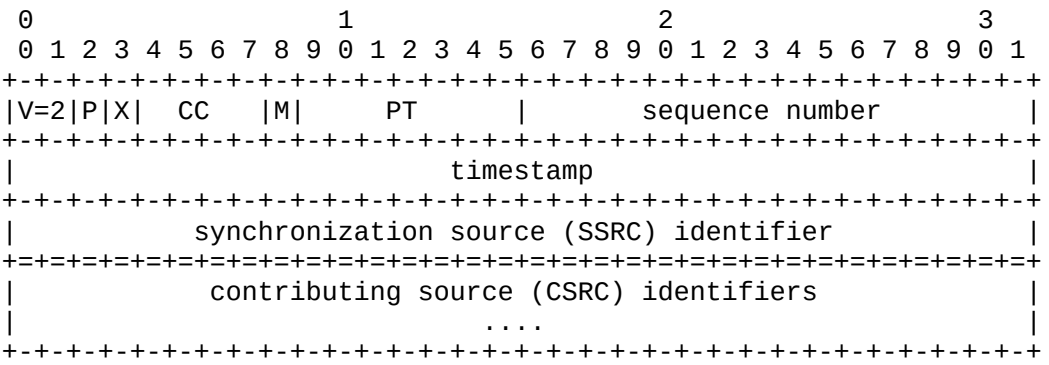
The functionality described in this section should cover the most of the GoClear-GoSecure mechanisms intended to be added to the ZRTP4J library. Like mentioned throughout the description, the ZRTP draft lacked at the development time a complete specification for several use cases, consequently various points of the implementation being based on our suppositions. Due to this fact it was considered to be better waiting for a final version of the standard before activating the optional GoClear-GoSecure functionality into the ZRTP4J library.

[20] For detailed information about the other states functionality in the state machine implementation please consult the documentation of the ZRTP4J library

Appendix II

Packet Structure for Various Protocols

- RTP packet - excerpt from IETF RFC 3550 :



version (V): 2 bits
   This field identifies the version of RTP.

padding (P): 1 bit
   If the padding bit is set, the packet contains one or more additional padding octets at the end which are not part of the payload.

extension (X): 1 bit
   If the extension bit is set, the fixed header MUST be followed by exactly one header extension

CSRC count (CC): 4 bits
   The CSRC count contains the number of CSRC identifiers that follow the fixed header.

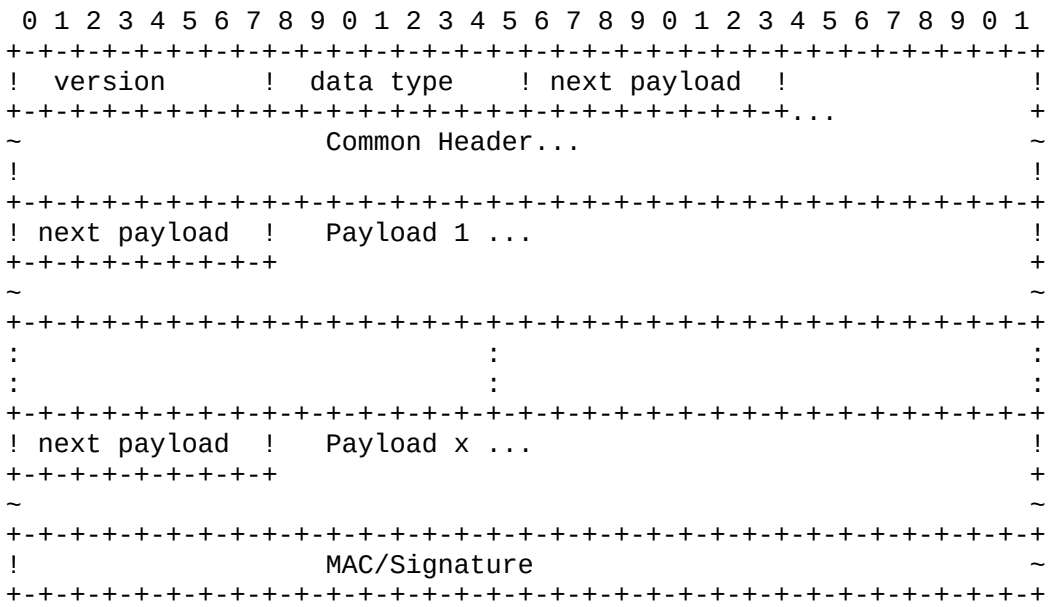
marker (M): 1 bit
   The interpretation of the marker is defined by a profile.

payload type (PT): 7 bits
   This field identifies the format of the RTP payload and determines its interpretation by the application.

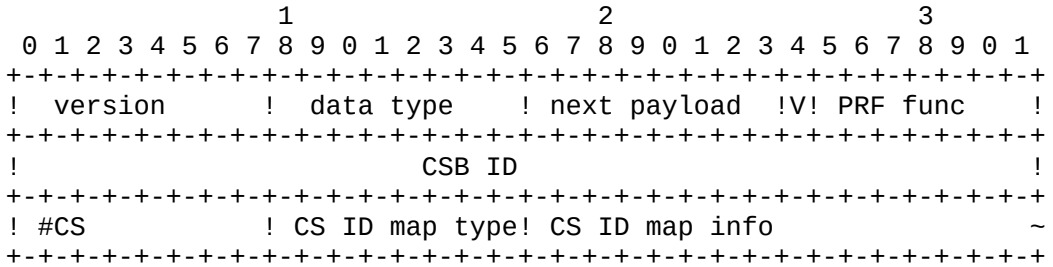
sequence number: 16 bits
   The sequence number increments by one for each RTP data packet sent

timestamp: 32 bits
   The timestamp reflects the sampling instant of the first octet in the RTP data packet

- MIKEY packet structure - excerpt from IETF RFC 3830 :



- MIKEY common header - excerpt from IETF RFC 3830 :



version (8 bits):
   the version number of MIKEY.

data type (8 bits):
   describes the type of message (e.g., public-key transport message, verification message, error message)

next payload (8 bits):
   identifies the payload that is added after this payload.

V (1 bit):
   flag to indicate whether a verification message is expected or not

PRF func (7 bits):
   indicates the PRF function that has been/will be used for key derivation.

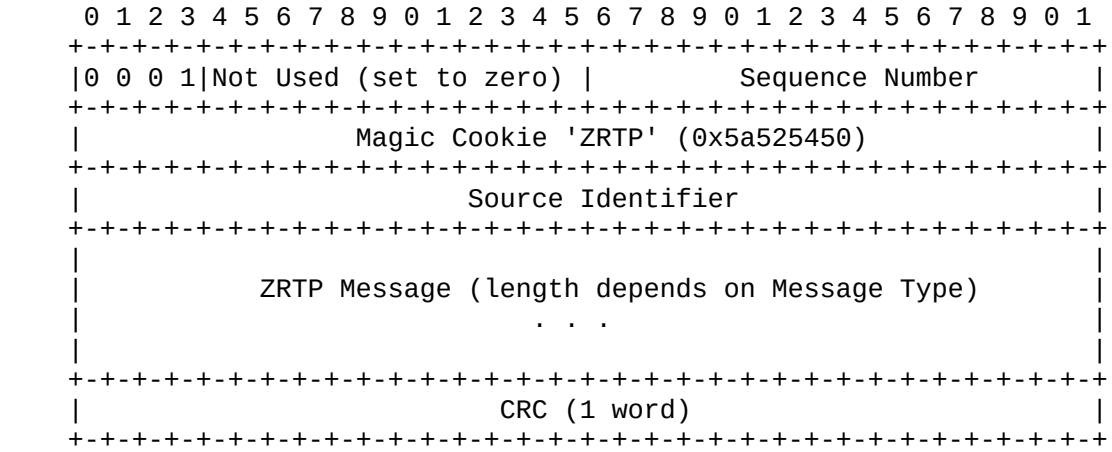
CSB ID (32 bits):
   identifies the CSB.

#CS (8 bits):
indicates the number of Crypto Sessions that will be handled within the CBS.

CS ID map type (8 bits):
specifies the method of uniquely mapping Crypto Sessions to the security protocol sessions.

CS ID map info (16 bits):
identifies the crypto session(s) for which the SA should be created. The currently defined map type is the SRTP-ID

- ZRTP packet structure - excerpt from RFC ZRTP draft version 15



Appendix III

Java SIP Frameworks

The currently available Java SIP frameworks are based on the three existent standardizations regarding SIP contained in Java Specification Requests (JSR). The three documents and the complementary APIs are addressed to the three Java platforms: Standard Edition, Enterprise Edition and Micro Edition. The associated JSR documents are respectively: JAIN SIP – JSR 32, SIP Servlet – JSR 116, SIP for J2ME – JSR 180.

- JAIN SIP -

This is the first standardized specification for a Java SIP API being part of the JAIN initiative – Java APIs for Integrated Networks. It is the closest framework to the protocol itself offering full SIP support. The main use targets are:

- SIP user agent or SIP server application development for the J2SE platform

- the base implementation for SIP Servlet containers which further allow SIP web application development

The API architecture is based on events using the Provider-Listener model. The specification is asynchronous using transactional identifiers to correlate messages. Various classes are included to create request or response SIP messages. The SIP messages' headers are defined through interfaces. This way an application can create the wanted header set which can be attached to a message. These messages are sent and received into and from the network through a Provider, which can have one or more Listeners registered. Also a Listener can be registered to more Providers. A Listener's main task is to process the received events encapsulating the messages.

Through its design JAIN SIP is extensible by defining a generic header interface which could be derived to create headers that aren't normally supported according to the standard.

The JAIN SIP API is the API used to develop SIP Communicator in its SIP related part.

- SIP Servlet -

This API is addressed to SIP network applications functioning especially in an enterprise environment. Consequently is generally used on servers supporting Java Enterprise Edition platform. Actually the API is developed following the HTTP Servlet architecture, mainly due to the similarities between SIP and HTTP, consequently making easy to develop applications permitting SIP and HTTP "interworking".

By using the SIP Servlet API, the modeling of the behavior of an Java Enterprise Edition based application is permitted in such way that this to perform the SIP signaling

functions either from an SIP user agent perspective or a SIP server perspective. As a difference from JAIN SIP, even it offers signaling support, SIP Servlet hides most of the internal SIP complexity, not being so close to the protocol specifications. From this point of view the SIP Servlet containers are the ones administering resources such as internal threads, transactions and dialogues, session state and others.

- SIP for J2ME -

We will not get into advanced details regarding this SIP framework which is addressed to the mobile devices using the Java Micro Edition platform. We only mention that the specifications are mainly maintained by Nokia and a long series of other contributors from telecommunications industry.

Besides the mentioned ones another API was also in development for a certain period, also as part of the Java Community Process under Nortel coordination. This was named JAIN SIP Lite and had as main purpose the possibility to develop SIP applications without implying extensive knowledge of the protocol. However following the departure of Nortel as the major contributor of the development process, the API was retired in 2006.

Bibliography

[1] J. Rosenberg et. all, *SIP – Session Initiation Protocol,* 2002 IETF RFC 2543

[2] H.323 - *Packet-based Multimedia Communications System,* 2006 ITU-T Standardization

[3] H. Schulzrinne et. all, *RTP: A Transport Protocol for Real-Time Applications,* 2003 IETF RFC 3550

[4] M. Baugher et. all, *The Secure Real-time Transport Protocol (SRTP),* 2004 IETF RFC 3711

[5] *** - VoIP Client Comparison
http://en.wikipedia.org/wiki/Comparison_of_VoIP_software

[6] D. Wing et. all, *Requirements and Analysis of Media Security Management Protocols*, 2009 IETF RFC 5479

[7] E. Rescorla, *Keying Material Exporters for Transport Layer Security (TLS),* 2009 IETF Standards Track draft

[8] J. Arkko et. all, *MIKEY: Multimedia Internet KEYing,* 2004 IETF RFC 3830

[9] D. McGrew, E. Rescorla, *Datagram Transport Layer Security (DTLS) Extension to Establish Keys for Secure Real-time Transport Protocol (SRTP),* 2009 IETF Standards Track draft

[10] P. Zimmerman et. all, *ZRTP: Media Path Key Agreement for Secure RTP,* 2009 IETF Informational Track draft

[11] A. B. Johnston, *SIP – Understanding the Session Initiation Protocol,* 2004 Artech House

[12] R. Fielding et. all, *Hypertext Transfer Protocol -- HTTP/1.1*, 1999 IETF RFC 2616

[13] J. Klensin, *Simple Mail Transfer Protocol,* 2008 IETF RFC 5321

[14] M. Handley, V. Jacobson, *SDP – Session Description Protocol,* 1998 IETF RFC 2327

[15] H. Krawczyk et. all, *HMAC: Keyed-Hashing for Message Authentication,* 1997 IETF RFC 2104

[16] S. Kent, K. Seo, *Security Architecture for the Internet Protocol*, IETF RFC 4301

[17] S. Kent, *IP Encapsulating Security Payload (ESP)*, IETF RFC 4303

[18] S. Kent, *IP Authentication Header*, IETF RFC 4302

[19] C. Kaufman, *Internet Key Exchange (IKEv2) Protocol*, IETF RFC 4306

[20] J. Orrblad, *Alternatives to MIKEY/SRTP to Secure VoIP*, 2005 Telecommunication System Laboratory, KTH Stockholm

[21] A. Steffen et. all, *SIP Security*, 2004 "Lecture Notes in Informatics" Bonner Köllen Verlag

[22] T. Adomkus, E. Kalvaitis – *Investigation of VoIP QoS using SRTP Protocol*, 2008 Electronics and Electrical Engineering, Kaunas: Technologija No. 4

[23] *** - Opnet Modeler http://www.opnet.com/solutions/network_rd/modeler.html

[24] G.1010 - *End-user multimedia QoS categories*, 2001 ITU-T recommandation

[25] P. Gupta, V. Shmatikov – *Security Analysis of VoIP Protocols*, http://www.cyber-ta.org/pubs/shmat_csf071.pdf

[26] D. Wing et. all, *Requirements and Analysis of Media Security Management Protocols*, 2008 IETF draft

[27] L. Dondeti, *MIKEYv2: SRTP Key Management using MIKEY*, 2007 IETF draft

[28] F. Andreasen et. all, *Session Description Protocol (SDP) - Security Descriptions for Media Streams*, 2006 IETF RFC 4568

[29] M. Baugher, D. McGrew, *Diffie-Hellman Exchanges for Multimedia Sessions*, 2006 IETF draft (expired)

[30] D. McGrew et. all, *Encrypted Key Transport for Secure RTP*, 2007 IETF draft

[31] J. Arkko et. all, *Key Management Extensions for Session Description Protocol (SDP) and Real Time Streaming Protocol (RTSP)*, 2006 RFC 4567

[32] J. Fischl et all, *Framework for Establishing an SRTP Security Context using DTLS*, 2009 IETF Standards Track draft

[33] D. Wing, *DTLS-SRTP Key Transport (KTR)*, 2009 IETF Standards Track draft

[34] E. Rescorla, N. Modagugu, *Datagram Transport Layer Security*, 2006 IETF RFC 4347

[35] J. Peterson, C. Jennings, *Enhancements for Authenticated Identity Management in the Session Initiation Protocol (SIP)*, 2006 IETF RFC 4474

[36] T. Dierks, E. Rescorla, *The Transport Layer Security (TLS) Protocol v1.2*, 2008 IETF RFC 5246

[37] N. Modagugu, E. Rescorla, *The Design and Implementation of Datagram TLS*, Proceedingd of NDSS, 2004

[38] H. Schulzrinne, S. Casner, *RTP Profile for Audio and Video Conferences with Minimal Control (IETF RFC 3551)* , 2003

[39] Riccardo Bresciani , *The ZRTP Protocol Security Considerations* , Research Report, Ecole Normale Superieure de Cachan, 2007

[40] E. Barker, D. Johnson, M. Smid, *"Recommendation for Pair-Wise Key Establishment Schemes Using Discrete Logarithm Cryptography,"* NIST Special Publication 800-56A Revision 1, March 2007.

[41] D. Sisalem, J. Floroiu, J. Kuthan, U. Abend, H. Schulzrinne, *"SIP Security"*, John Wiley & Sons Ltd, 2009

[42] The AVISPA team, *HLPSL Tutorial*, 2006

[43] M. Petrashek et. all , *Security and Usability Aspects of MitM Attacks on ZRTP*, Journal of Universal Computer Science, vol. 14, no. 5, 2008

[44] H. Hlavacs et. all , *Enhancing ZRTP by using Computational Puzzles* , Journal of Universal Computer Science, vol. 14, no. 5, 2008

[45] B. Ramsdell, *S/MIME Version 3 Message Specification*, 1999 IETF RFC 2633

[46] ***- SIP Communicator - http://sip-communicator.org/

[47] *** - The OSGi Architecture - http://www.osgi.org/About/WhatIsOSGi

[48] *** - YATE – Yet Another Telephony Engine - http://yate.null.ro/pmwiki/

[49] *** - Apache Felix - http://felix.apache.org/site/index.html

[50] *** - NIST JAIN SIP - https://jain-sip.dev.java.net/

[51] *** - *Java Media Framework API Guide*, Sun Microsystems 1999

[52] *** - Freedom for Media in Java - http://fmj-sf.net/

[53] *** - BouncyCastle cryptographic library - http://www.bouncycastle.org/

[54] *** - Java Cryptography Extension - http://java.sun.com/j2se/1.4.2/docs/guide/security/jce/JCERefGuide.html

[55] *** - ZRTP Secured Session - Client Interoperation Tests, http://spreadsheets.google.com/pub?key=pISWJIBcSpdFFBkkNNKpvC