Security and Non-Repudiation for Voice-over-IP conversations

DIPLOMA THESIS

Christian Hett

Prof. Dr. Claudia Eckert
Fraunhofer-Institut für Sichere Informations-Technologie
Fachgebiet Sicherheit in der Informationstechnik
Fachbereich Informatik, TU-Darmstadt
Supervisor: Dr. Andreas U. Schmidt,
Dipl. Inform. Nicolai Kuntze

Darmstadt, July 2006

You can contact the author at diplomarbeit-contact(at)c-hett.de

Contents

1

Abstract

This thesis presents a concept to achieve non-repudiation for natural language conversations by electronically signing continuous, packet-based, digital voice communication (VoIP).
Signing a VoIP-based conversation means to protect the integrity and authenticity of the bidirectional data stream and its temporal sequence which together establish the security context of the communication. The solution is based on chains of hashes and continuously chained electronic signatures. The protection is provided continuously during the ongoing conversations, they are *not* processed at once like traditional digital documents which could be trivially signed.
A possible implementation and necessary protocols are described to apply these concepts to SIP/RTP-based VoIP-communication. This provides a high level of inherent security and enables PKI-based non-reputable signatures over voice as true declarations of will, without additional witnesses and in principle between unacquainted speakers.
As a demonstrator for these concepts, an efficient VoIP-archive securing the integrity of SIP-based two-party conversations was implemented.

Keywords

Electronic signature; non-repudiation; voice over IP; interval signature; cryptographic chaining; natural-language communication

2

Introduction

2.1 Motivation and scope[1]

The latest successful example for the ever ongoing convergence of information technologies is internet based telephony, transporting voice over the internet protocol (VoIP). Analysts estimate an annual growth rate in a range of 20% to 45%, expecting that VoIP will carry more than fifty percent of business voice traffic (UK) in a few years [2]. The success of VoIP will not be limited to cable networks: Convergent speech and data transmission will affect next generation mobile networks as well. The new technology raises, however, some security issues. For eavesdropping traditional, switched analogue or digital phone calls, an attacker has to get physical access to the transport medium. While he might have a harder time decoding the more complex protocols of IP networks and DSL-lines after dredging landlines of home-users, digital networks are generally more vulnerable to attacks, especially when used together with insecure wireless networks. Efforts to add security features to VoIP products are currently infrequently deployed, though proposals exist for privacy protection. Protocols like SRTP [3] can provide end-to-end security to phone calls, making them independent from the security of the transport medium and the communication provider [4]. Secure VoIP protocols, using cryptographic protection of a call, would even be at an advantage compared to traditional telephony systems. While the problem of eavesdropping is solved for digital networks - at least in theory -, no efforts to add non-repudiation can be found.
On the other hand, voice conversations provide inherent evidentiary value as they allow forensic evaluation and analysis of the contained biometric data, e.g., as an independent means of speaker identification [5, 6]. Methods for the latter are advanced [7] and provide for the recorded voice communication

[1] Parts of this section were taken from our paper submitted for [1]

a rather high probative force, for example in a court of law. In comparison to other digital media, e.g., text documents, specific features of voice communication can be viewed as contributing to security. The medium of communication here consists of a linearly time-based full duplex channel enabling inter- and transactivity [8]. In particular, interactivity enables partners to make further enquiries in case of insufficient understanding. Furthermore, digital voice communication will offer a rather high reliability and quality of service, leading generally to a higher understandability of VoIP communication in comparison with its analogue predecessors [9, 10]. The mentioned properties mitigate to some extent problems to which digital documents are usually prone, e.g., misinterpretations due to misrepresentation, lack of uniqueness of presentation, and inadvertent or malicious hiding of content.
Therefore, departed from the basic security aspects of VoIP communication, conversations will here be viewed on a transactional level between caller and callee. The top-level category of protection targets considered is non-repudiation of conversations. Three tasks of ascending complexity are addressed in this work:

1) Protection of the integrity of voice conversations. Protecting a (recorded, digital) voice conversation from falsification and tampering with is different from protecting the integrity of other digital data due to the relevance of the temporal context. In particular, packet ordering and loss have to be considered properly, and a creation time must be assigned to each conversation.
2) Authentication of speakers. An initial authentication of caller and callee together with the inherent biometric authenticity of voice is the basic approach to this problem. While it could be resolved in principle solely on the transport layer, it is advantageous to combine it with the methods of 1), to obtain proof that a (recorded) conversation was carried out completely from the authenticated devices and not taken over by an attacker. It has to be noted that each authentication of a speaker requires trust in the devices used by the communication parties.
3) Electronic signatures over voice conversations. Building on 1) and 2) it is possible to achieve, for voice conversations, the level of non-repudiation provided by electronic signatures over digital documents, i.e., an expression of will. For this, the aforementioned tasks must be complemented by a proof of possession of a trustworthy signature token and device, and the intention to sign.

Theoretical and technological concepts for each of the tasks 1) – 3) are presented and their realisation is described in a demonstration environment for secure archiving of calls. The existing VoIP infrastructures are largely un-

affected by the concepts through a seamless and efficient integration in the SIP [11] and RTP [12] protocols. On the other hand, the three tasks pose increasing technical requirements on the part of the involved devices.
Task 1) can be resolved in a stand-alone way, without any change in devices or transport methods. The solution concept rests on cryptographic secrets created at the initiation of a call and their perpetuation throughout the call by a cryptographic chaining method. As a key issue, stability, quality of service, and necessary fragility to prevent attacks must be balanced with each other. The main application of this concept is a secured archive for VoIP conversations which yields tamper-resilience and in consequence evidentiary value by far exceeding that of traditional tape archives. On the other hand, tasks 2) and 3) need additions on the used devices ranging from the inclusion of authentication software and the roll-out of pertinent data, to the fulfilment of high security requirements for signature terminals.
The combined benefits of the technology developed here amount to a new paradigm for non-repudiation of digital data. The combination of integrity of recorded conversations, security about the identity of dialogue partners, and finally expressions of will embodied in signatures enables legally binding verbal contracts between unacquainted persons.
A trivial way of providing signed conversations would be be achieved by recording the conversation and afterwards replay the recording to the participants. Such a solution raises various problems as the recording is unprotected during the recording and the replay has to be protected as well. The practicality of such an approach is also doubtful, because it requires at least twice as much time. By contrast, the presented concepts allow to use natural, fully interactive, traditional conversations to create legally binding verbal contracts. While the German law already provides this for verbal agreement, here proof is provided even without additional witnesses.

2.2 Scenarios

2.2.1 Secure Self-Signed Archive for voice conversations

In security sensitive application domains like telephony brokerage, calls need to be archived to ensure non-repudiation. A solution based on this is shown in Figure 2.1 and was implemented as a demonstrator together with a verification and playback-tool.
In this scenario, party A archives the call and signs it using his certificate or the certificate of the archiving-software, which is realised as proxy-server. A timestamping-authority secures the exact start of the call. In contrast to traditional long-term-archives, this solution is based on streaming and securing

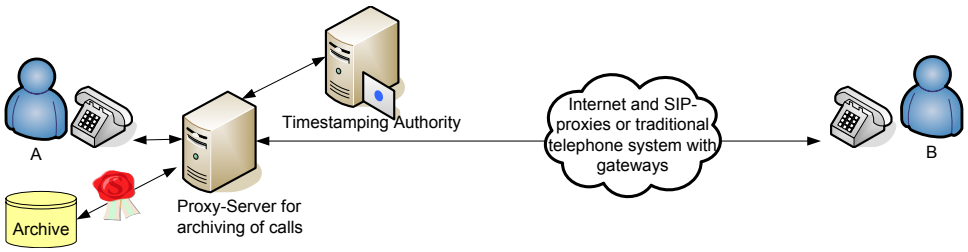


Fig. 2.1: Self signed archive for voice conversations

ongoing conversations. It uses the timestamp to pinpoint the exact starting-time of a conversation and not the moment of archiving.
Note that the archive could be placed at the site of either of the parties or anywhere in between, as long as at least some part of the communication is based on SIP/RTP. It could be under the control of any party including third parties or installed centrally in a corporate environment.

2.2.2 Signing interactive, duplex voice conversations between two parties

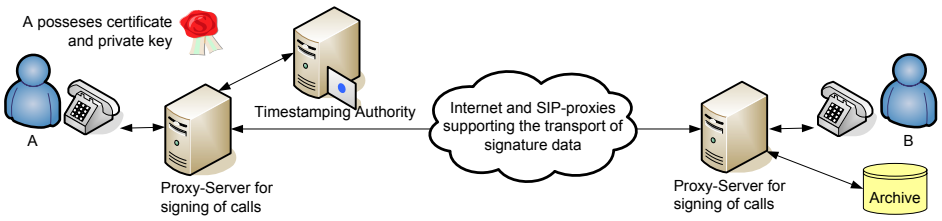


Fig. 2.2: Signing scenario: A signs the call, B archives it as proof

The goal of this scenario is to achieve legally binding contracts over VoIP without witnesses. Figure 2.2 shows a bidirectional interactive conversation between two parties A and B. A wants to sign the conversation and release it to B as a declaration of his will, i.e. a commitment in the sense of a signed offer. By signing the call A also expresses his explicit consent to the archiving of the call. In effect, A wants or is required to make sure that the conversation between A und B provides non-repudiation, in any case he expresses the explicit will to make non-reputable statements or stipulations in the call. In order to do that, A possesses a digital certificate for a public/private key pair. B and any third party to whom the signed conversation is presented as evidence, are assumed to be able to verify the certificate of A and any data signed with the associated private key, e.g. by using a PKI structure in the background.

A signs the complete call including both channels comprising everything that B says. B stores the signed conversation in a secure archive of his choice. B can later proof to third parties or court that the call happened and had the claimed contents. If B fails to store the conversation in an archive or deletes it, A can deny that the call ever happened. An important security goal of A and B is that A signs exactly what was heart by both parties.

2.3 Thesis organisation

The preceding section presented and motivated the two main scenarios for the technique. Chapter 3 presents technical details and protocols of VoIP based on the SIP-standard. It covers all important aspects and side-aspects that led to problems for the implementation of the demonstrator or are otherwise important. Chapter 4 is the main part of this thesis: Here the requirements for both scenarios are analysed and solutions are described. This finally leads to a secure data format for signed calls and a protocol to carry out signed conversations as described in the signing scenario. From the general signing protocol chapter 5 leads to the technical implementation for SIP/RTP. Here different ways of incorporating signatures in the SIP and RTP protocols are discussed and assessed. Chapter 6 explains and presents the software that was developed for this thesis: A flexible SIP-proxy to manipulate calls with a pluggable complete implementation of the self-signed-archive scenario, a tool to check and play signed calls and a statistic tools to measure packet loss. Chapter 7 goes into technical details how this was implemented and describes what every class does. The last chapter (Chapter 8) finally presents a vast field of future research- and implementation ideas related to the presented concepts.

2.4 Related work

Most basic security requirements of VoIP are perfectly well handled by the SRTP standard [3]. It is discussed in section 3.4 and provides basic transport integrity, confidentiality, replay-protection and transport authentication. But because it is based on symmetric algorithms, i.e. HMACs, it does not provide non-repudiation. The partner knows the HMAC-key for verification and thus can easily create HMAC-keys at any later point in time. Therefore recorded SRTP-packets provide no evidentiary value that the recorded conversations was unaltered by anyone but the speaker. Also protection of the exact start-time e.g. through timestamps is not provided.
It is currently not clearly specified how the symmetric keys of SRTP are to be

managed between both parties because several standards exist. But if they are embedded in the SDP-body of a SIP-INVITE-call, they can be signed using S-MIME. While S-MIME is the method of choice for non-repudiation of email-messages and attachments (documents), it cannot provide this for SRTP- conversations as the protected data only includes temporary IP-addresses, port-numbers and the symmetric key known to both parties.
Related work on securing the integrity of streamed data by signatures is scarce. The authors of [13] describe a method for stream signatures for broadcast media, where the presented methods in this thesis are concerned with interactive, duplex conversation. In [14, 15] a method to transport authentication information employing watermarks and steganography is presented. Digital signatures are not explicitly used and achievable data rates seem low.

3

Basic principles of packet based voice communication

This chapter will present the basic principles and protocols of VoIP considering SIP and RTP as example. Almost all explained aspects are necessary to understand the supplied source code for the demonstrator, where parsers for these protocols have been implemented.
First the Session Initiation Protocol (SIP) is discussed. SIP is a powerful open protocol for initiation, modification and termination of any kind of interactive user session. Here its purpose is to

- register a SIP-phone at the provider, informing him of its network address.
- initiate calls and signal incoming calls using the concept of dialogues.
- (optionally) provide presence information and enable clients to subscribe to events.
- (optionally) enable instant messaging.
- (optionally) transport events during the call like DTMF-digits, but this can also be transported in-band with the audio data [16].

Its older and much more complicated rival is the H232-protocol which is not in the scope of this work, but similar enough to apply the main concepts anyway.
Both SIP and H232 use the same protocol and codecs for the actual transportation of the voice, namely RTP, the real-time transport protocol which itself is based on UDP.
In order to transport real-time conversations over packet based networks, the audio input first needs to be digitized by sampling and quantization. Then it is chopped up into packets and encoded into bytes using a codec. After that, the packets can be optionally encrypted using the SRTP-protocol or VPN technology like IPSEC embedded into the network stack and are send out using the network.

3.1 SIP - The Session Initiation Protocol

SIP is specified by the IETF in RFC 3261 [11] and is a general protocol for establishing multimedia sessions. It is not restricted to VoIP-calls, but this is the prevalent use today.
SIP is a text-based, open and extensible protocol based on well established internet concepts like URLs. It is very similar to HTTP and uses MIME and SMIME from internet emails to optionally transport some kind of message-body with a specified MIME content-type. Here the next important VoIP sub-protocol comes into play: SDP, the session description protocol defined in RFC2327 [17], which describes codecs, parameters and port numbers of the RTP-stream used to actually transport the audio stream. If SIP creates a session and transports a SDP-message body, a multimedia conversation over RTP is established.
SIP is usually used on top of the UDP-protocol and it's well known port number is 5060. It can also be used with TCP port 5060 and although support for this is mandatory by RFC 3261, it is not widely deployed among VoIP-operators today. For transport security SIP supports TLS over port 5061, but this only secures the signaling and session establishment, not the actual audio data.

3.1.1 Request and response types

Similar to HTTP where requests can be of type GET, PUT, POST, etc., SIP-requests contain a method to execute. Some are listed in table 3.1
For every SIP-request the client answers with a SIP-response. This applies to all requests with the sole exception of **ACK**. A response message differs from a request in that its first line contains a response code that is very similar to HTTP-responses. Common values are listed in table 3.2

3.1.2 SIP trapezoid and call setup

Figure 3.1 shows the typical situation when a SIP based VoIP-call is made between A and B using the infrastructure of a VoIP-provider. A calls B and after some time B terminates the call.
This call is not carried out directly between two parties. Making direct, proxy-less calls is possible, but the callee would need to have fixed IP-addresses or at least a specific DNS-address. Further the caller's address book would have to include the direct IP-addresses or host-names of his contacts in addition to their usernames. Also such a setup would strongly interfere with lawful

Method	Description
REGISTER	Register or deregister clients with a presence server so that it knows who is online and at which IP-address they are to be found.
SUBSCRIBE	Subscribe to events like message wait indicators. This SIP-extension is defined in RFC 3265 [18].
NOTIFY	Notify about events that have been subscribed with SUBSCRIBE.
MESSAGE	Send instant message/text message. This SIP-extension is defined in RFC 3428 [19].
INVITE	Start a session (see Figure 3.1 for how this is used). Because starting a session usually involves a ringing phone that needs to be picked up by the user, the process of establishing a session itself is a dialogue. Dialogue-requests can receive multiple responses and are acknowledged with a final ACK.
CANCEL	Cancel an ongoing INVITE request before the session is established.
ACK	Acknowledges all dialogue-related responses thus making dialogues a three way handshake.
BYE	Terminates an session started with INVITE.
INFO	Used to carry session related control information. This is sometimes used to tranport DTMF dial tones. This SIP-extension is defined in RFC 2976 [20].

Tab. 3.1: SIP methods

interception and any kind of regulation from a SIP-provider.
The proxy on the left can be thought as the outgoing proxy of A, responsible for locating B. Therefore it analyzes the URL that A is calling. The host-portion of B's URL is handled by B's proxy, so A's proxy forwards the INVITE-request to B's proxy. Because B registered with his proxy server as soon as his phone had network connectivity, B's proxy will always know where to actually find B and which network address and port-number B uses. We can see that A's outgoing proxy will refuse to make calls without previous authentication. We also see that every response related to the process of establishing of a session has an additional ACK-message.
Another important detail is that the last ACK-message of the session initiation and the session termination bypass both proxy-servers and are send directly between A and B. At this point in time A already knows the specific contact (and network IP-address) of B and thus there is no need to use the proxies. This behavior is highly unfavorable for the implementation of a proxy in this thesis, but the Record-Route-Headers will solve this.

Response-Code	Meaning	Comments
1xx	Provisional, searching, ringing, queuing etc.	
100	Trying	Is sent regularly after INVITE to indicate that a proxy is contacting the next hop or the phone.
180	Ringing	Is sent regularly after INVITE to indicate that the user is notified, e.g. by a ringing phone.
2xx	Success	
3xx	Redirection, forwarding	
301	Moved Permanently	User has been assigned a new phone number.
302	Moved Temporarily	Call has been temporarily redirected.
4xx	Request failure (client mistakes)	
400	Bad Request	
401	Unauthorized	Responses like this contain a challenge so that the next request can use it for authentication. This is usually used for REGISTER-requests.
404	Not found	Call-party or subscribed event doesn't exist.
407	Proxy Authentication Required	The proxy needs authentication. This response also contains a challenge. This is usually encountered for INVITE-requests to avoid spoofing of bills.
486	Busy Here	The called party is already having a conversation.
5xx	Server failures	
6xx	Global failure	

Tab. 3.2: Important SIP Response Codes

3.1.3 Important headers and how do proxies work?

By means of Figure 3.2 some headers that are important for the implementation are discussed:

From The SIP-URL of the caller.

To The SIP-URL of the callee. From and To will not be swapped for SIP-responses to SIP-requests.

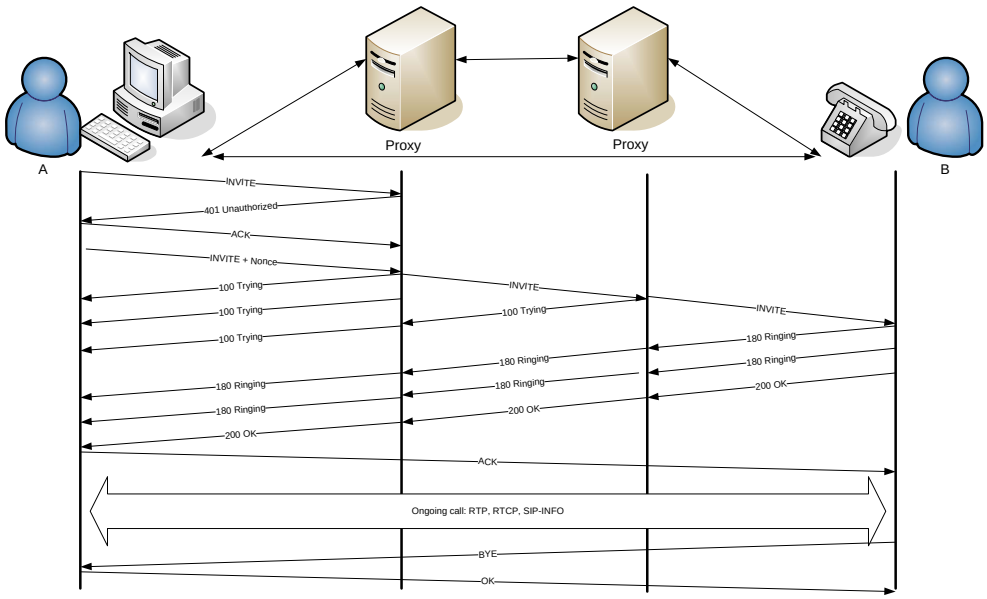


Fig. 3.1: Sequence diagram of a complete SIP-VoIP-call. As there is indirect communication between party A and B using the proxy servers as well as direct communication for ACK/BYE/RTP, this is also called "the SIP trapezoid".

Contact SIP-URL that represents a direct route to the party that sends this header. This is used for the last ACK in Figure 3.1 which bypasses the proxies. It is also used in REGISTER-SIP-requests.

CSeq Contains an increasing sequence number for every request.

Via For a request the sender and every proxy server on the way stores its contact information in the form of a SIP-URI as the top-most VIA-header line.
When the final receiver constructs his answer, he will copy all Via-headers but the first into the response and send it to the topmost VIA-header. For a response every intermediate proxy reads the topmost line, removes it and forwards the remaining packet to this new destination.
The VIA-headers ensure that every SIP-response goes exactly the same way backwards as the SIP-request was send, so that every proxy sees the answer.

Record-Route If a proxy wants to stay in the path and not be bypassed as seen in Figure 3.1, he can put his SIP-URI/contact data at the topmost position. This does not affect the processing of VIA-headers, but comes into effect when the next resulting

```
INVITE sip:496123456789@84.178.139.124:5070 SIP/2.0
Record-Route: <sip:212.227.15.225;ftag=1168841412;lr=on>
Record-Route: <sip:+496123456789@217.188.44.231;ftag=1168841412;lr=on>
Via: SIP/2.0/UDP 212.227.15.197
Via: SIP/2.0/UDP 212.227.15.225
Via: SIP/2.0/UDP 212.227.15.197
Via: SIP/2.0/UDP 217.188.44.231
Via: SIP/2.0/UDP 1und1-1.sip.mgc.voip.telefonica.de:5060
From: +491793123456 <sip:+491793123456@1und1-1.sip.mgc.voip.telefonica.de;user=phone>
To: +496123456789 <sip:+496123456789@1und1.interconnect.sip.voip.telefonica.de;user=phone>
Call-ID: 7dbde21a-6a92b0f7-73ae04b2-b982@subscriber1.interconnect.mgc.voip.telefonica.de
CSeq: 1 INVITE
Supported: timer
Session-Expires: 1800
Min-SE: 1800
Contact:  <sip:+491793123456@1und1-1.sip.mgc.voip.telefonica.de:5060>
Allow: INVITE,ACK,PRACK,SUBSCRIBE,BYE,CANCEL,NOTIFY,INFO,REFER,UPDATE
Max-Forwards: 6
Content-Type: application/sdp
Content-Length: 618

v=0
o=- 1710955 0 IN IP4 62.53.226.3
s=Cisco SDP 0
c=IN IP4 62.53.226.3
t=0 0
m=audio 18324 RTP/AVP 8 0 99 102 2 103 4 104 105 106 107 18 0 125 101
a=rtpmap:99 G726-16/8000
a=rtpmap:102 G726-24/8000
a=rtpmap:103 G7231-H/8000
a=rtpmap:104 G7231-L/8000
a=rtpmap:105 G729b/8000
a=rtpmap:106 G7231a-H/8000
a=rtpmap:107 G7231a-L/8000
a=rtpmap:125 GnX64/8000
a=rtpmap:101 telephone-event/8000
a=fmtp:101 0-15
```

Fig. 3.2: Complete content of a SIP-INVITE-*request* for initiating a call using the provider GMX from United Internet. Only tags and branches parameters were stripped from URLs and phone numbers were anonymized. In the lower part the SDP-body can be seen which defines codecs and port 18324 for the RTP-stream.

request is created (e.g. ACK-packets for a dialogue initiating a session or BYE-requests to end a session).

Max-Forwards Every proxy has to decrement this field, so that loops can be detected (similar to limiting the amount of Received-headers in SMTP-mails).

Content-Type SIP-requests and -responses can contain message bodies. This could be instant text chat messages, pictures or –in the case of VoIP calls– SDP-data which is used to negotiate codecs, parameters and RTP-port-numbers.

Content-Length If a body is present, this field should contain its length, so that several SIP-messages can be send consecutively and split by the receiver if SIP is used over TCP.

Call-ID This uniquely identifies separate calls because multiple calls can happen at the same time.

3.2 SDP - the Session Description Protocol

For VoIP-calls the body of a SIP-INVITE-request usually contains SDP-data to establish the call that is transported through RTP-streams. This includes RTP-port-numbers and codec-offerings for one or more audio- and video-streams. The response (e.g. the "200 OK"-response in Figure 3.1) contains the RTP-ports from the called partner and a list of the codecs he understands. Codecs are mapped to the 127 possible RTP-payload-numbers seen in section 3.3.2. Thus SDP performs a two-way handshake.
An example of SDP can be seen in Figure 3.2 below the blank line. SDP is a text-based format with fixed syntax and little variations. SDP is thoroughly explained in RFC 2327 [17] and contains many field with no use in the context of SIP like the email-address of the initiator. Some important fields are described in the following:
The line starting with "`o=`" describes the owner/initiator of the session and contains the IP-address of the RTP-endpoint, which could even be another computer or process and not the one where SIP-signaling is handled. The IP-address is repeated in the "`c=`"-line. The line

```
m=audio 18324 RTP/AVP 8 0 99 102 2 103 4 104 105 106 107 18 0 125 101
```

is the most interesting one: Every time a line starts with "`m=`" this marks another stream like e.g. an additional video stream. After that there is the type of stream: `audio`, `video` or `application`. Next comes the port-number and the protocol (either `RTP/AVP` or `UDP`). The remaining numbers are a list of possible RTP-payloads ordered by priority. RFC 3551 [21] defines the static range from 0–95. For each specified payload type, it states the used codec, the frequency and the number of channels. The remaining payload numbers 96–127 leave room for extensions and dynamic negotation of codecs and parameters. This is done through the "`rtpmap`"-attribute:
Lines starting with "`a=`" are used to provide additional attributes to the last stream defined with "`m=`". "`rtpmap`" is used to define a mapping of a payload-types to a codec, the number of channels and the frequency for the dynamic range of payloads. "`fmtp`" provides yet again additional details for a negotiated payload.
A SDP-line starting with "`k=`" (not shown here) would be a reserved way to transport a cryptographic master-key for the use with SRTP.

3.3 RTP - the Real-time Transport Protocol

3.3.1 Packet loss and jitter

RTP is specified by the IETF in RFC 3550 [12] and is a lightweight protocol above UDP that –unlike UDP– allows for ordering of the received packets

and provides timestamps for e.g. measurement of jitter. It also allows to detect packet loss, but –unlike TCP– there are no mechanisms to retransmit lost packets. Doing so would be impractical for interactive voice data, because the retransmit-timeouts of TCP are very large and the time needed for acknowledgments and retransmission at least triples the maximum latency of a packet. The receiver would need to stop playback for some time resulting in massive stutter or provide a large playout buffer resulting in forbidding high latency. Overall the latency of interactive voice needs to be minimized and should be kept below 150ms, according to the ITU-T document G.114 [22]. Otherwise the illusion of talking at the same time would be lost. The latency is not only the result of the pure network latency, but also from the needed time to process network packets and audio data and because of another important characteristics named jitter: Jitter is the variance in the latency of transmitted packets. A jitter buffer needs to increase the overall latency by storing/buffering received packets for a short time until all packets have arrived, maybe in another order than the sent. Packets arriving too late cannot be considered and are dropped. Therefore the size of the jitter buffer is a tradeoff between packet loss and increased latency.

3.3.2 Format of RTP-packets

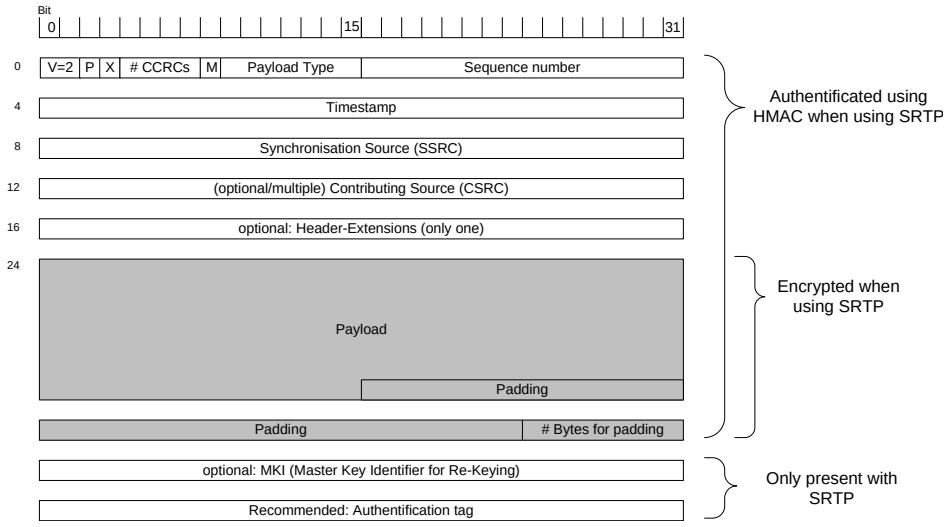


Fig. 3.3: Format of RTP- and SRTP-packets.

In Figure 3.3 the format of a RTP-packets is shown. The contributing source IDs and the synchronization source ID are not discussed. They are not impor-

tant for two-party VoIP-calls and are mainly used for broadcasts where RTP was originally developed for. Important fields are:

V Version number: Must have the value 2.

P Padding-Bit: If this bit is set, then the last byte is the amount of padding, otherwise no padding is present.

X Extension-Bit: This bit indicates whether an extension is present.

M Marker-Bit: The meaning is defined by the codec or the profile, usually it means that the stream continues after silence suppression.

Payload Type This value from 0 to 127 indicates the used codec and its parameters. Payload types 0–95 are defined in RFC 1890 [21], while the remaining range from 96-127 is dynamically negotiated, usually through the SDP-protocol (see section 3.2)

Sequence Number This is the sequence number of the packet. Every new packet increments this number by one. This value is rather short with 16 Bit and therefore wraps around after 65535. The *absolute*, long sequence number can be reconstructed from temporal context.

Note that this value does not have to start with 0. It is even recommended to start with a random number to support symmetric encryption and mitigate known plaintext attacks. This is recommended not only if SRTP is used, but in any case because the VoIP-phone might not be aware of e.g. IPSEC-based VPNs which could also benefit from reduced payload-predictability.

Timestamp This field provides a timestamp based on a regular time source for every packet. The value is not concerned at all by timezone and is not synchronous between different parties. Instead it is also recommended to start with a random value to support encryption. Also like sequence numbers, the 32-Bit values for timestamps can wrap around for long enough calls. It increases by the clock value as seen in the SDP-body of Figure 3.2, which is usually identical to the sampling rate. A common value is 8000 Hz, so that the timestamp increases by 8000 every second and by one for every sample. For PCMU and PCMA (the G.711-codec) it even increases by the length of the payload so that it could almost be used like a stream position.

Header Extension If the X-bit is set, then the first 16 Bit of the Header Extension contain a code for the type of extension and then 16

Bit specify the amount of DWORDs that follow this extension header. Only one extension is supported per RTP-packet and they are reserved for experiments. If an extension became official, a new version of RTP would be released.

Padding Only present if the P-bit is set. Padding is not really needed for plain UDP-based RTP, except if SRTP-encryption is used and mandates a certain block cipher size. The use of only one header-bit to express that no padding is needed provides a very efficient way avoiding unnecessary overhead through unneeded padding.

Please note that some codecs or phones support silence suppression, so that they don't have to send RTP-packets if nobody talks. In this case the timestamp-values are incremented as if packets were send, so there are gaps between received packets. The sequence numbers will not show gaps, because they count sent packets. So the sequence numbers can always be used to determine packet loss.

3.4 SRTP - Secure RTP

SRTP demands a short discussion in this thesis as it is a well established way to provide the security-goals integrity, confidentiality and authentication. However, SRTP does not provide non-repudiation. SRTP also does not use public key cryptography.
The format of a SRTP-packet was already shown in Figure 3.3 as it is very similar to RTP. Security keys and parameters are handled out of band and RTP already handles padding well. To support intelligent network devices and advanced routing and Qos, only the gray shaded parts, which is payload and padding, are encrypted with a symmetric algorithm. Optionally –though recommended– the whole packet is authenticated using a HMAC which is appended to the RTP-packet. A master key identifier supports re-keying. Note that HMACs are not suitable for providing non-repudiation as they are based on shared secrets.
The SRTP-standard in RFC 3711 [3] also describes how to derive IVs from the RTP-header and how to derive key material for symmetric encryption and authentication from a master key. The master key might be transported using the k-value of SDP (see section 3.2), but there a lots of ways of transporting it, e.g. SDP security description [23] or MIKEY [24].
The SRTP-standard also describes an algorithm for replay detection based on a sliding window of at least 32 Bit. Duplicate packets are to be dropped and counted.

3.5 Audio codecs and packet loss concealment

The RTP-payload itself results from applying a codec and its parameters to a piece of audio with a specific length, e.g. 20ms. There are codecs like G.711 that handle each such piece independently, e.g. by simply encoding it. Other codecs like Speex, GSM and ILBC are more complicated and based on linear predictive-coding (LPC). In general they try to encode parameters to reconstruct speech instead of storing the samples themself. This needs much less bandwidth.
The codec G.711 is explained in more detail, not only because it is simple to implement, but mainly because it is widely used: It is the ISDN-codec used in the normal telephone system and can be processed by VoIP-gateways without additional conversion or degradion of quality. It is a waveform codec which directly encodes the digitized samples, which are then transmitted as packets. It is based on the PCM (pulse code modulation) method and simply uses a logarithmic scale to encode the dynamic range of voice more efficiently and provide a better signal-to-distortion ratio than a linear quantization would provide.
In RTP it is used in two variants: Static payload type 0 (PCMU) is the μ-law[25]-variant shown in Equation 3.2 which is used in America and Japan.

$$F(x) = sgn(x) \begin{cases} \frac{87.7|x|}{1+\ln(87.7)}, & |x| < \frac{1}{87.7} \\ \frac{1+\ln(87.7|x|)}{1+\ln(87.7)}, & \frac{1}{87.7} \leq |x| \leq 1 \end{cases} \tag{3.1}$$

Static payload type 8 is a-law[26] shown in Equation 3.1 and used in Europe.

$$F(x) = sgn(x) \frac{\ln(1+\mu|x|)}{\ln(1+\mu)} \quad -1 \leq x \leq 1 \tag{3.2}$$

In each case x is the input sample in the range of $x \in [-1, 1]$ and $F(x)$ is the encoded value stored as byte.
Both static payload types specify one channel and a sampling rate of 8000Hz.
As seen in section 3.3.1, in packet-based telephony there is always the problem of dealing with packet-loss which is called **Packet Loss Concealment**.
For the G.711-codec every packet contains only a specific short piece of samples independent of other packets. Therefore only receiver-based techniques can be used, which try to create a substitution packet for the one that was lost. The simplest method would be to replace the lost packet with silence, which results in very bad results. An excellent compromise between implementation complexity and result is duplication of the last packet (See Figure 3.4 and compare the magenta line “G.711, Burst=1” with the dark-blue line “G.711plc, Burst=1”). An appendix to the G.711 standard also describes

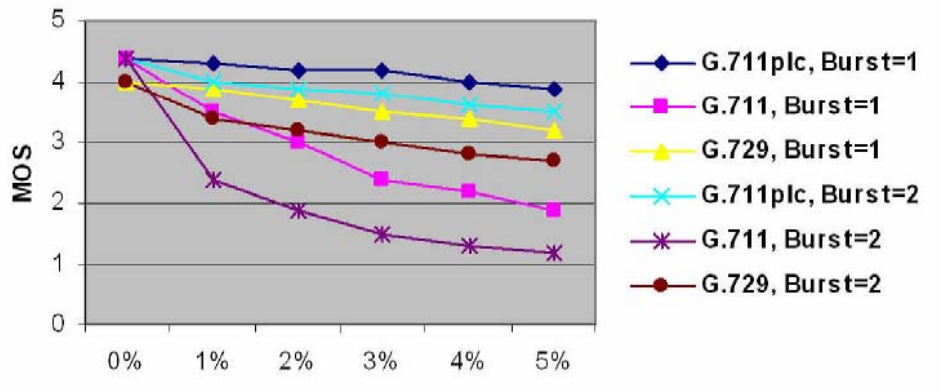


Fig. 3.4: MOS depending on packet loss for the G.711 (and G.729) codec. MOS (Mean opinion score) is expressed from 5 for excellent to 1 for bad. Burst describes how many consecutive packets were lost. The curve for G.711 uses silence for lost packets, while G.711plc uses the simple form of packet loss concealment to duplicate the last packet. Courtesy of [27]

a computationally expensive algorithm to reconstruct the missing waveform. More complicated codecs like ILBC [28] are specifically designed for graceful speech quality degradation in the case of lost packets. By using prediction and modeling speech instead of simply encoding it, they can easier regenerate missing packets. They can also use so called sender-based techniques for packet loss concealment like interweaving data from other packets. For an example of the performance of ILBC, see Figure 3.5.

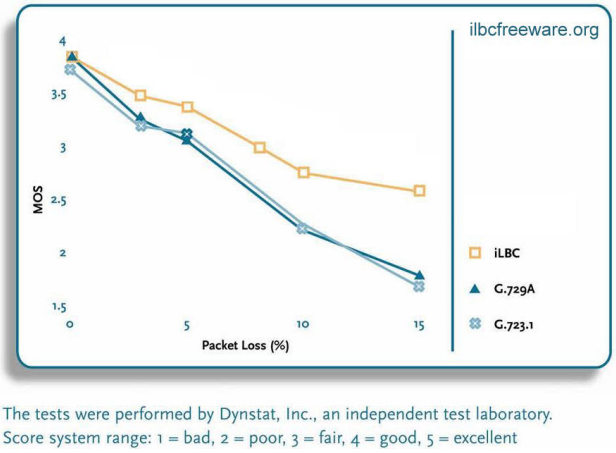


Fig. 3.5: MOS depending on packet loss for the ILBC-codec. Note that the x-axis ranges from 0%–15% unlike 0%–5% as in Figure 3.4. Courtesy of Global IP Sound [29]

3.6 STUN, NAT and Session Border Controllers

As we have seen in the last sections, there are several places where IP-addresses and port-numbers are embedded into SDP and SIP-packets, maybe even protected with TLS. Furthermore RTP doesn't use fixed port numbers, but instead dynamically assigned free ports. This, of course, raises several problems in todays home as well as corporate networks, which make heavy use of firewalls and NAT (**N**etwork **A**ddress **T**ranslation)- routers. Often the parties A and B participating in a call are both behind NAT-devices.
A NAT device in its most common form (also known as masquerading) translates port numbers and IP-address in IP, TCP and UDP-packets in a way that a whole private network (like 192.168.1.0/24) can access the internet through only one public addressable IP-address, for example assigned by a broadband-ISP. For UDP-packets –unlike TCP-packets– there is no clear definition of a connection and its termination which would be necessary for matching and rewriting incoming answer-packets to outgoing packets.
For UDP two main heuristics are applied by NAT-devices:

1. If an application or host sends UDP-packets to the internet originating from a certain port, it is assumed that it will also want to receive answers to only that port.
2. If an application or host sends an outgoing UDP-packet, the NAT-devices creates a mapping in its translations-tables and leaves it for several minutes. This creates a kind of virtual connection based on the assumption of regular packet flow.

The latter is the reason why SIP-clients usually support the feature to regularly send empty SIP-packets to keep the connection open, even if there is no phone call for hours. The SIP-proxy can then "answer" these packets with a SIP-INVITE-requests for a new incoming call.
According to RFC 3489 [30], there are several types of NAT-devices:

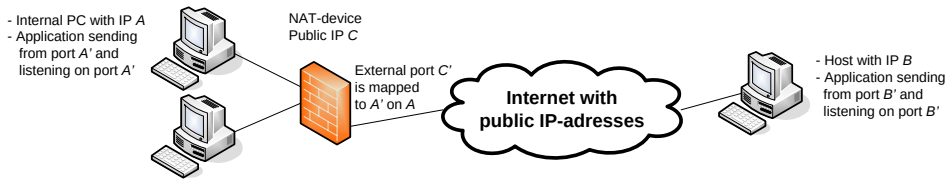


Fig. 3.6: Simple NAT-scenario where A is behind a NAT-router and B has a public IP

If an UDP-packet from internal host A and port A' is send to the outside host B, port B', then first this packet is rewritten, so that it appears to be originating

from the public IP of the NAT-device C. The source port is rewritten to C', a free, currently unused UDP-port of the NAT-device. This is recorded in the NAT-translation table. (see Figure 3.6)
If the NAT is a

- *full cone NAT*,
- *port restricted cone NAT* or a
- *restricted cone NAT*,

then a packet sent with source port B' from host B arriving on the NAT-device (IP C) on port C' will be rewritten and forwarded to A on port A'. Furthermore the same mapping of (A, A') to (C, C') is reused also for any other destination than B, B'.
But if it is a

- *symmetric NAT*

then packets coming from IP-address B on port B' are the only ones forwarded to A on port A'. Furthermore a different mapping can be used for each destination.
For the former three types of NAT-devices, a trick known as *UDP hole punching* [31] can be used to establish a direct connection between A and B despite of the NAT. For symmetric NATs this is not possible, because the mapping cannot be determined by the SIP-phone behind the NAT.
This trick is based on the STUN (Simple traversal of UDP over NATs) protocol and algorithms [30]. A STUN-server can be contacted by a STUN-client on two different ports and two different IP-addresses to learn

- whether it is located behind a NAT and which type of NAT it is, i.e. how the mapping changes for different IP-addresses and ports and from where answers can be received.
- its external IP-address, so that it can embed the correct IP-address in e.g. SDP-data or in the contact-header of SIP-packets
- the specific binding of a port-number, i.e. which outgoing internal port-number A' was mapped to which public visible port number C'.

A simple version of NAT-punching works like this after a SIP-phone has detected that it is used behind a NAT:
When an RTP-port was allocated for a call, first STUN is used originating from this port to determine the associated external IP-address and port-number. These are announced to the other party in the SDP-portion of the SIP-INVITE-request instead of the internal ones. The other party does the same with its RTP-endpoint and the SIP-response containing the SDP.
When the call starts, both parties start to send RTP-streams roughly at the same time. But depending on the specific type of NAT, the first RTP-packets

of B might be lost and dropped by the NAT of A, because it does not yet allow incoming packets from the specific host and port of B. But because A soon also starts to send packets to B and its IP and port-number, A's NAT quickly allows incoming packets from B's IP and port. Thus by using STUN a direct connection can be established in spite of NATs without any changes to the RTP, SIP and SDP-protocols.
Because this trick relies on unchanging mappings of internal ports to external ports, it does not work with symmetric NATs.

Session Border Controllers

Another common, but controversial technique to solve costly NAT-related support calls in exchange for increasing bandwidth consumption is the use of SBCs (Session Border Controllers) or similar kinds of relays. In this case, the SIP-provider rewrites all IP-addresses and ports embedded in SDP-bodies, so that all RTP-traffic is relayed through his servers which are accessible with public IPs. This also helps with lawful interception.
This technique is very reliable, but can be expensive and has problems with SMIME or –to an lesser extent– TLS-encryption of the SDP-body containing the IP-addresses and port-numbers. Also it might be that the relay is not willing to transport more than one RTP-stream, thus limiting video conferences or other innovative application specific data. Some SBC-vendors even provide monitoring and restrictions of the transmitted voice-data and used bandwidth [32], which can seriously impact novel uses of SIP and RTP to transport application specific data like signatures for non-repudiation.
Experiments in preparation for this thesis showed the interesting result, that two main SIP-providers in Germany, namely GMX from United Internet and SipGate from Indigo Networks, already redirect *all* traffic through their own relay servers, even without earning money from network-internal calls. Fortunately, these redirect servers did not apply polices or filters and forwarded any kind of RTP-traffic presented.

4

Concepts[1]

4.1 Main requirements

The central requirements for achieving non-repudiation by signing VoIP are, of course, related to security. Of the well known trinity – Confidentiality, Integrity, and Availability – of information security requirements, integrity is the central one to achieve non-repudiation for digital, packet-based, natural language communication. It needs to be assured that a communication was not changed at any point in time, be it during transmission or later. Furthermore, integrity comprises as well the integrity of any relevant meta-data created or used during a call, in particular call-time and the data that authenticates the communication partners, or at least the partner exerting the signature over the communication.
Due to the special features of voice communication, i.e., a bidirectional, full-duplex interactive conversation, only both channels together provide the necessary context to fully understand the content of the conversation and to make use of the inherent security that interweaved natural language conversations provide. To ensure that parts of the talk are not exchanged with other parts, replaced by injections, or cut out, the envisaged system needs to ensure what we call **cohesion**. This means that the temporal sequencing of the communication and its direction is data for which the integrity needs to be protected in a way that makes later tampering practically unfeasible, i.e., by sufficiently strong cryptographic methods. Cohesion as a feature related to time entails a subsidiary requirement, namely the secure **assignment of a temporal context** to a conversation. Each conversation has to be reliably associated with a certain time, which must be as close as possible to the conversation's start and the initiation of the signing (note that assignment of a signing time is a legal requirement for qualified electronic signatures according to the Euro-

[1] Parts of this section can also be found in our paper submitted for [1]

pean Signature Directive and pertinent national regulations). Drift of the time base should be mitigated during a signed conversation. Finally, cohesion also refers to qualitative aspects of the communication channel. A signatory is well advised not to sign a document which is illegible or ambiguous. In the digital domain this relates to the presentation problem for electronically signed data [33]. In analogy, the **quality of the VoIP channel** must be maintained to a level that ensures understandability to both partners during the time span in which the conversation is signed.
To ensure later **availability** of a signed conversation is a trivial requirement for the receiver to be able to make use of it as evidence. Considerations of long-term archiving aspects for signed digital data can be found in [34].
Further requirements regard the **efficiency** of the system design and implementation. First, it is highly desirable, both from a security as well as an efficiency viewpoint, to sign and secure the VoIP conversation as "close" as possible to its transmission, and conceptually close to the actual VoIP stream, which is realised i.a. by using the the original RTP-packets as part of the signed and archived data format. Simplicity of the implementation should minimise the effect on existing systems and infrastructures. For the archiving scenario (section 2.2.1), no additional client-side requirements should be necessary. For the signing scenario (section 2.2.2), obviously clients need to be modified, but for both scenarios the existing infrastructure should be left completely unchanged. This can be achieved as long as a way to transport additional signing data is available, which is the case for SIP/RTP. An efficient use of memory, bandwidth, storage space, and computational resources can be achieved by undermentioned conceptual design decisions. Furthermore, **scalability** of the concept to a large number of concurrent calls is a necessity in real business environments. This means that centralised signature creation infrastructures must be avoided. Finally, any architecture that copes with VoIP needs to appropriately take **packet loss** and **quality of service** into account, in particular in view of the cohesion requirement.

4.2 Requirement and concepts for dealing with packet loss and QoS-Policies

In general, packet loss leads to modifications of the conversation perceived by the receiver, which is a problem when providing non-repudiation. When the archived data is played back as evidence, it must be semantically identical to what both partners said: This of course includes the requirement of cohesion from the last section, i.e. it must be impossible that parts are cut and exchanged. But also the interactivity and understandability must be maintained:

If B does not understand what A said (because he mumbled, had ambient noise, expressed himself unclear, B did not listen or the transmission was distorted by jitter, delay or packet loss) he asks for clarification or repeating of what was said and vice versa. So the fact whether some part of the conversation was heart by the other party must be protected, too. And packet loss is the natural attack/problem for this requirement, because it affects understandability.
Because it is generally a good principle that A only signs what was actually heart and transmitted, the protocol will support this and handle packet loss like this (also see Figure 4.1):

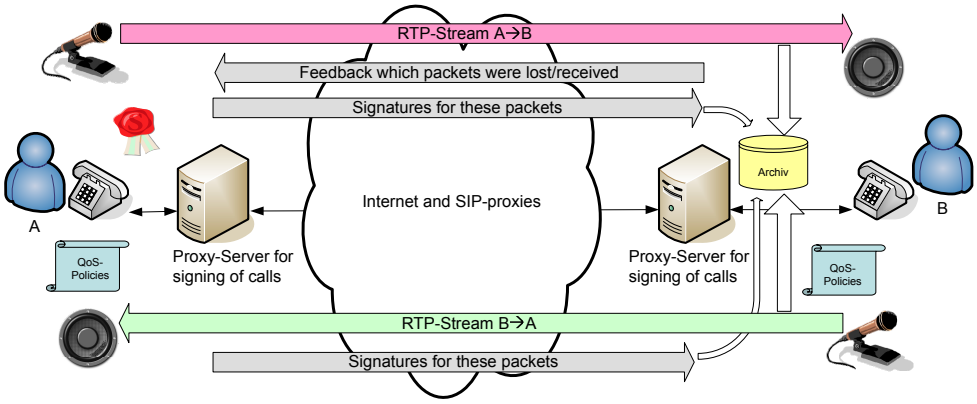


Fig. 4.1: Simplified diagram of the signing protocol: Besides the two normal RTP-streams for voice, there are two additional channels to tranport lists of packets actually received by the other party, so that only these packets are signed.

Instead of signing the packets B sends on the way to A, **only the packets that A actually received from B are signed**. And not the packets that A sends to B are signed, but instead **the packets that B actually received are signed**.
A result of this principle is of course that signing can only happen with a certain delay as seen in the next chapter.
So this design principle deals with packet loss and guarantees, that only packets that were heart by the receiver are actually signed. But this does not yet guarantee the goal to protect understandability and thus interactivity and its semantic meaning: Now A needs to trust B that B correctly reports the amount of packets received back to A and vice versa. If B forges this information, there can be a mismatch between recorded call and acutal interactivity: B can still listen to the packets and play that part of the call in real-time and thus the person B knows what A said and can pretend that he heart it. But his software could be manipulated (by B) to transmit a wrong amount of packets, so that

A will not sign this part (which might be unfavorable to B). In front of court when the recording that A signed digitally is presented, A suddenly learns that important objections and constraints he made are missing.
This also works vice versa: A can pretend that packets from B did not arrive and therefore are not signed. And yet, he heart them, understood them clearly and maybe even approves them verbally. So B is tricked because he thinks that the whole call or negotiated contract is non-reputable. Later he wants to use the call as evidence and learns that some statements of A are so distorted that they are inaudible. The whole recording might be worthless to him.
The solution to this is controlling the quality of service. Both parties agree to a level of quality (in terms of packet loss after dropping packets that are too late for dejittering) and the software will terminate the call, if this level is not met. This is called a *QoS-underrun* and the agreed quality bar is called *QoS threshold*.
As seen in chapter 3.5, some percent of packet loss will not affect the understandability of the call, especially when advanced codecs like Speex or iLBC are used. Furthermore in experiments done for this thesis, significant packet loss was seldom seen at all, especially over broadband-connections. Packet loss usually occurred together with really bad quality or congestion of the connection, which lead to complete drops of communication: Either by the software or by the caller or callee who can't hear each other and try to call again. An example for this is leaving the reception area of a wireless lan.
Therefore it seems very plausible to handle packet loss in this way: It is measured and if it reaches a to severe level, the connection is forcefully terminated by the proxy-software. Of course this is a matter of policy and therefore configurable. Here an default value of 5% is used which worked very well.
It should be noted that this implementation of a proxy-server (instead of direct inclusion in a SIP-client) can now be seen as a **reference monitor** [35]: RTP- and SIP-packets will only go through the proxy-server, he does not only provide access, but can also enforce policies, namely that specific QoS-guarantees are met.
It should also be noted that this method here for handling packet loss is in strong contrast to the technique of stream signatures presented in [13] where the authors show methods of signing unidirectional broadcast traffic in a way that makes it possible to still check a signature if packets are lost in transmission. Here signatures for lost packets are simply not provided. If packets are lost afterwards (e.g. in the archive), this is detected because it means that integrity is broken. This could invalidate the whole recorded call.

4.3 Extending SIP/RTP to transport signatures

As seen in Figure 4.1, obviously there must be a way to transport signatures and lists of actually received packets over the internet between A and B. The signing protocol extends SIP/RTP in a compatible way to transport signatures and acknowledgments of signatures. Several ways of doing this on protocol-level are explained in chapter 5. Here it should be noted, that a strength of the technique is that it does not modify or in any way delay the transported audio stream itself. Instead signatures are transported separately from the audio stream in an extra channel.
Without delaying and postponing the signing process itself, the real time requirement for interactive voice can be achieved. Of course this method wouldn't achieve non-repudiation of the call if it was possible for A to simply stop the signing process and let B believe that everything is okay. (And vice versa). Therefore time-limits are defined how long a signature (or an interval signature, see next section) may be delayed before the call is terminated. Again this is a matter of policy, different options would be to signal it to the users or to ignore it. But the preferred policy is to abort the call and signing. If this timeout is one second, then only one second of speech could be made before the call is terminated. This barely is enough to make any statement or say a complete sentence.

4.4 Building intervals to gain efficiency

As stated in section 4.1 it is important that the implementation is efficient and scalable. Therefore solutions like signing every single packet with a signature algorithm like RSA is neither efficient with respect to bandwidth and storage capacities, nor sufficient to protect the full conversations. Signing each packet alone easily uses more than 128 bytes to store a signature of a RTP packet with maybe only 44 bytes of sampled audio and is computationally expensive. Therefore the central concepts of **intervals** and **interval signatures** are introduced.
Each party collects packets in intervals of adjustable length, e.g., one second. Time based intervals could pose certain problems as it might be hard to predict the size of the required buffers for very dynamic codecs or suspension of the timer. Other solutions to determine the duration of a interval could be based on the sent and received amount of packets. But for the sake of simplicity here the amount of packets in an interval is based on timer based events.
Every second the collected packets are sorted by sequence number and their hashes are assembled in a data-structure with additional meta-information like direction, sequence numbers and time. This small data-structure is then

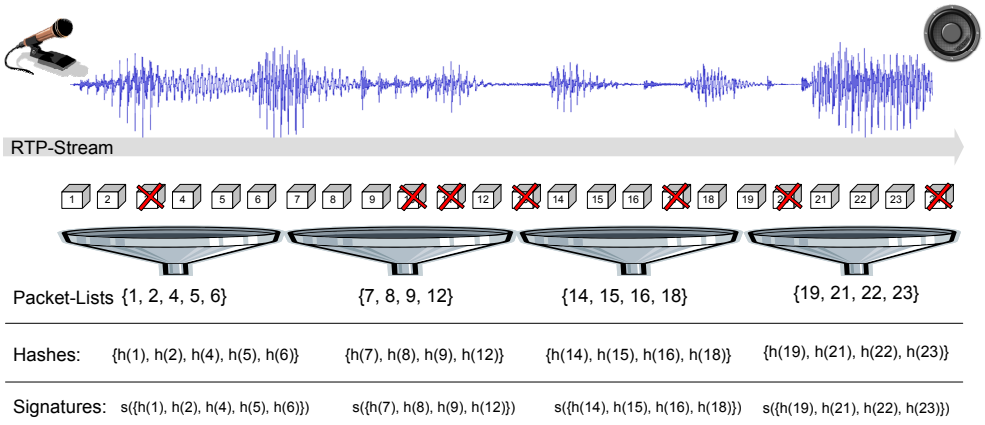


Fig. 4.2: Building intervals: After a certain time the collect-buffer (illustrated as a funnel) is flushed and processed. First the packet-list is created. Below the list of hash-values is shown. Below that the hash-values (without the packets themself) are signed. Not shown is important additional metadata which is signed together with the hashes like the date/time of each interval and the direction.

signed with a conventional signing algorithm like RSA, using the private key of party A. They are then sent to B who will store them together with the collected RTP packets he actually received. Note that the full packets are transported only once in the normal RTP stream. Therefore bandwidth as well as CPU time needs are drastically reduced making the whole method applicable in the first place (computing hashes is much less expensive compared to RSA signatures, especially on mobile phones with limited processor-speed).
As a side note, it would be possible to further reduce the bandwidth usage since the sequence number of the packet is enough for B to reconstruct the hash, rendering the transmission of hashes unnecessary. Transmitting only packet numbers would bear the cost of additional consistency checks on the part of B. This variation is not discuss further because the result would be the same but presentation would be more complex than signatures built only on actual data.
It is also important to stress that the signature for a complete interval is broken if any of the hash values becomes invalid, namely, if any bit of the signature or in the associated and stored RTP packets is changed, or if any packet is missing. This is the basic way of protecting the integrity of the speech data.
So not only is the original RTP-stream completely unaffected from the signing process, but furthermore the additional data is very small and scare: Only once a second a list of packets and a signature over their hashes is sent, while

the RTP-packets continue to be transmitted with their typical high rate without any delay (expect for making a copy of them to a buffer).

4.5 Achieving cohesion by interval chaining

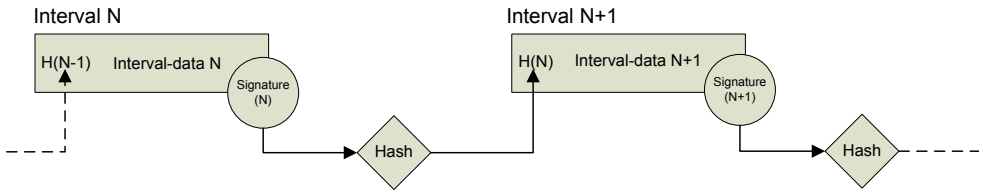


Fig. 4.3: This Figure shows how intervals are chained and how signatures and hashes are interleaved.

Signing intervals alone does not ensure cohesion for the whole call, only for the 1s long intervals. An attacker could still exchange parts of the conversation or cut them out by exchanging and omitting complete intervals. Therefore the concept of hash chains is used: Every interval contains, embedded in its metadata, a hash of the last interval including its signature. In this way signatures and hashes are interleaved ensuring that there is a continuous stream of signatures building an unbreakable chain as seen in Figure 4.3. (That is, as unbreakable as e.g. RSA and SHA1 are)

4.6 Interweaving both channels of bidirectional communication

The chaining of intervals is further extended to factor in the bidirectional nature of the call. Both channels are interwoven and the chaining applies to both channels. An interval of packets from the channel A⇒B contains a hash of the last signed interval from the channel B⇒A and so on. When lists of packets and also intervall-signatures arrive on either side, additional checks are employed that ensure that there is no time drift between both channels. This way cohesion is strongly secured by RSA signatures and by the nature of a bidirectional conversation: Mutual inquiries hold together the semantics of the conversation, while synchronous time after start of the call is guaranteed by checks and secured with hash-chains over metadata and timestamps embedded in the original RTP-packets.

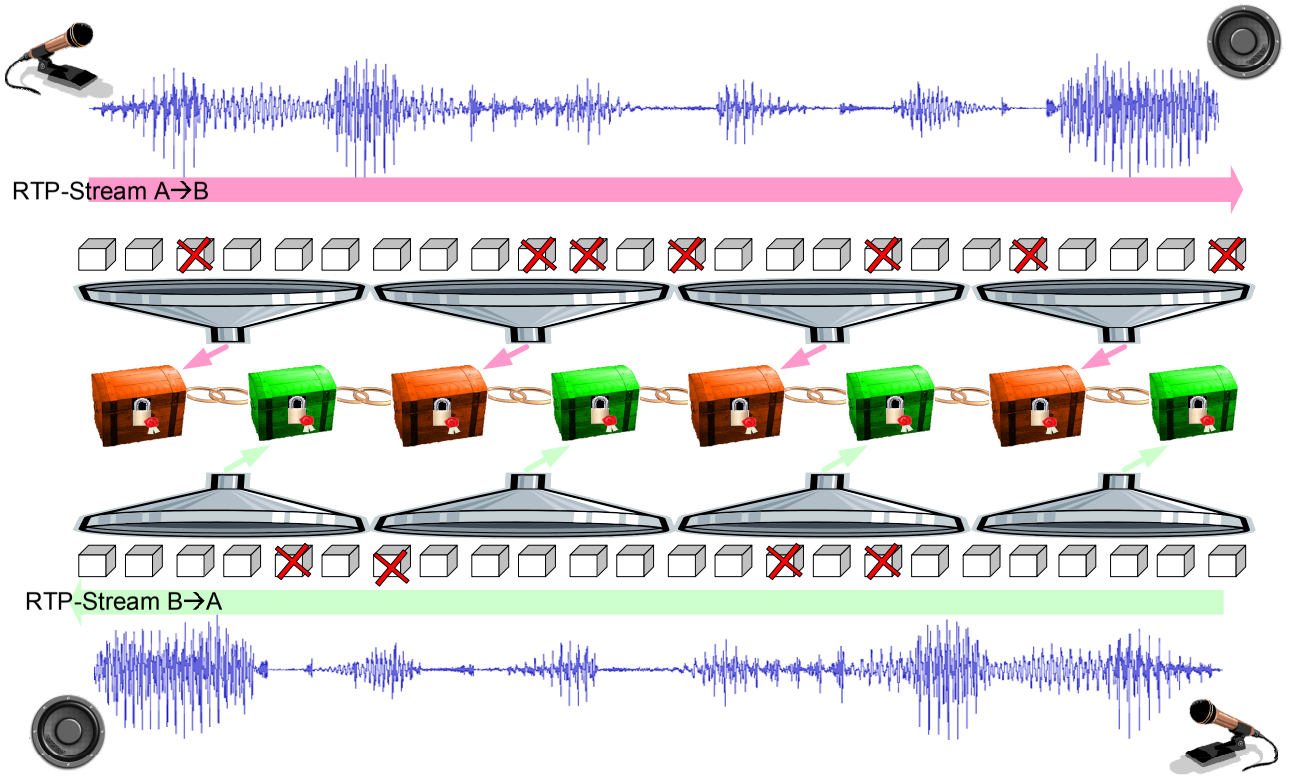


Fig. 4.4: This Figure shows how intervals from both channel A⇒B (red) and B⇒A (green) are interwoven: Each interval carries its signature (illustrated as a padlock with a seal) and intervals are hash-chained (illustrated with a golden chain). Both channels are treated independently (each channel has its own collecting-buffer on both sides, packet loss is measured, lists of packets are created, hashes are built independently) expect for the hash-chaining, where an interval A⇒B is followed by B⇒A, is followed by A⇒B, and so on ...

4.7 Signed data format and additional timestamps

As non-repudiation of calls is only meaningful if the party who is interested in using a conversation as evidence –in this case party B– can possess it as evidence, the signed conversation must be recorded. Special emphasis must be put on the format in which the calls are stored, i.e. the final outcome of the signing protocol (or alternatively the self-signed archive-scenario). All intervals of a call are simply stored continuously by the proxy software the recording party.

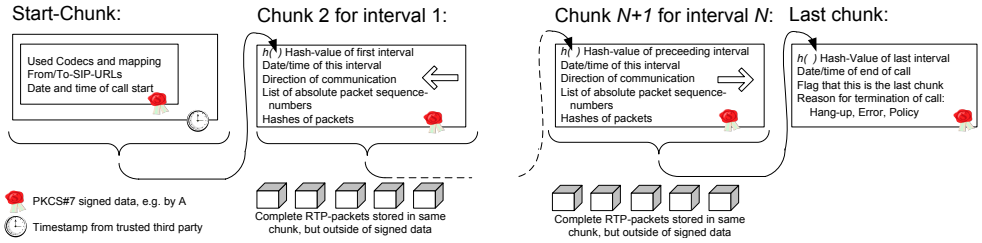


Fig. 4.5: The data format for signed conversations. This also shows the hash-chain of signed interval-data. Note that B stores the collected packets next to A's signature for each interval. The box with the clock in the first chunk symbolises the time-stamp from a trusted authority

In general additional timestamps (as can be seen in the start chunk in Figure 4.5) help pinpointing the exact start and duration of the call, thus providing additional security. This is especially important in the self-signed archive scenario, where the archiving proxy might be compromised by an attacker. Such an immense attack can be mitigated quite well, if

- the start of the call is signed by a trusted third party
- the long-term archive is separated from the archiving proxy
- the call data is immediately streamed from the archiving proxy to the long-term archive
- the long-term archive immediately verifies that all arriving data is valid, properly signed and has not too much packet loss, i.e. cohesion of time, sequence numbers and timestamps are maintained.
- (optionally) an itemised bill from the phone company can be compared to the timestamp

This architecture is highly secure, because an attacker would only have the duration of one interval (e.g. 1 second) to execute his attacks including any forgery, cuts, realigning and speech synthesis. While doing this he would

have to maintain interactivity so that no participant notices it. Also any delay in streaming an interval to the archive would be immediately noticed and the whole call cannot be delayed to an later point in time because of the time-stamp form a third party authority and optionally an itemised bill.
The format itself is shown in Figure 4.5 and is a simple chunk-based continuous data format starting with an initial chunk containing meta-data (SIP URLs of caller and callee, date and time of start of call and mapping of codecs to RTP payload fields). After that signed intervals are stored as they are produced, reducing working memory requirements to the order of magnitude of one interval. At the end of the call (either by normal hang-up or by policy termination if quality of service requirements are not met) a special chunk is added which contains date/time and the reason for the termination of the call. For each interval the associated chunk contains the collected RTP packages of this interval, and the following signed data:

- The direction/channel of the interval (from A to B or vice versa)
- The date and time of this interval
- The list of absolute package sequence numbers and
- The hashes of each considered packet.

In case of the signing-scenario, the signature is transported separately from the RTP-packets, so the signature only covers hashes of the packets. Other variations were described in section 4.4.
This signed data is embedded in a PKCS#7 signed envelope container. PKCS#7 [36] is an established standard to sign data using public key cryptography and X509-certificates [37]. It is able to carry the signing certificate and also any intermediate certificate with the signature. Therefore it enables the signature verifier to build and check the certificate chain against a trusted root. Only the first PKCS#7 envelope needs to store the whole certificate chain, all other envelopes do not need to store any certificates, because they are then already known to the verifier.

4.8 Signing protocol

This section describes the protocol that is used to transport acknowledgments, signaling, and signatures for the normal RTP packets containing multi-media frames. It builds on section 4.2 and provides the missing details. As already stated, the protocol does not affect or delay the normal transportation of any of both RTP-streams. Instead it adds a second low-bandwidth transport channel separate to the original one, e.g. a second RTP-channel. It should be noted that most ways of transporting additional data provide UDP-like, unreliable datagram-based communication and not a reliable stream like TCP. Therefore

retransmission must be provided.
Initially it is necessary to signal the fact that someone is calling who has signature capabilities and is willing to sign the conversation. This can be part of the normal negotiation at the start of a SIP-call, e.g. with the the `k`-value of the SDP-protocol which is used for securing the call using SRTP (see 3.4). Here, A transmits essentially the same data as in the first chunk in Figure 4.5, signed using his private key.
To further describe the protocol it must be stressed again that even though both channels of the duplex conversation need to be signed to provide cohesion, in the basic signing scenario still only party A needs to have a private key for signing. A has to sign both directions of the conversation. Accordingly the signing protocol differs for both channel directions which are described in the next two subsections.

4.8.1 Signing the channel from A to B

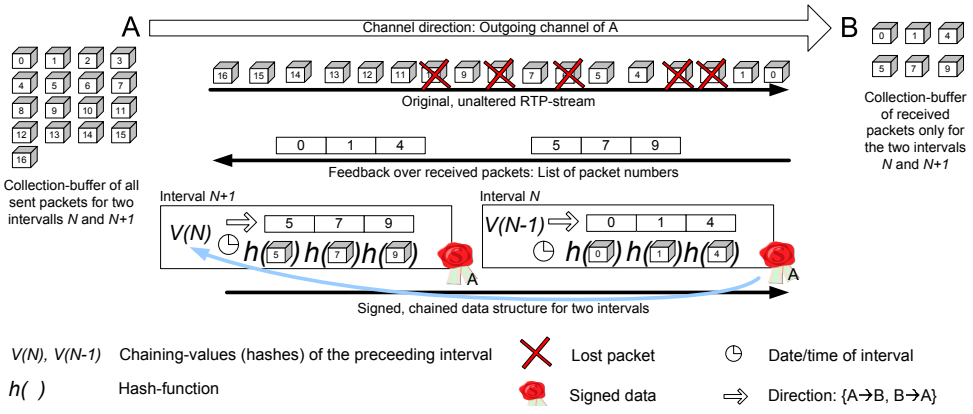


Fig. 4.6: Schematics of the signing protocol for the direction A⇒B

For the channel direction A⇒B, A could simply send (every second) an signed interval containing the hash-values of all his RTP-packets of this interval. If any packet loss occurs –which is normal in current networks– B wouldn't be able to provide the missing packets as evidence. In this case his whole archive of the call would be deprived of probative force. Therefore B transmits a list of all sequence numbers of the packets he received during an interval to A, who will then send the requested signature that covers exactly the received packets.
More precisely both A and B continuously collect all packets for the channel A⇒B in a small buffer, as indicated in the left- and rightmost parts of

Figure 4.6. Whenever B's interval timer expires he will send a list of packet sequence numbers that were collected during that interval to A and start a new interval. The transfer of the sequence numbers is performed asynchronously, while the conversation is going on. A will then create the hash values for this list of packet numbers. He uses his complete collection of packets he sent. Then he creates and signs an interval data structure as described in Section 4.7 and transmits it to B. B then checks the signature and hashes and stores this together with the collected packets as a chunk in the archive file. He can then drop the collection of packets from memory. A can drop his complete packet collection from memory one iteration later, because if the communication that is described in this paragraph fails, then B will retransmit its packet list until A has successfully transmitted the interval signature. While waiting for the response, the interval timer of B is temporarily suspended as it can only fire after an interval signature was successfully transmitted. During this suspension of the interval timer the signing process does not stop. The timer is restarted if all data of the last interval is transmitted. If the transfer of this data needs too much time (e.g. longer than the interval length) the system should consider this as a quality of service under-run and terminate the call.

4.8.2 Signing the channel from B to A

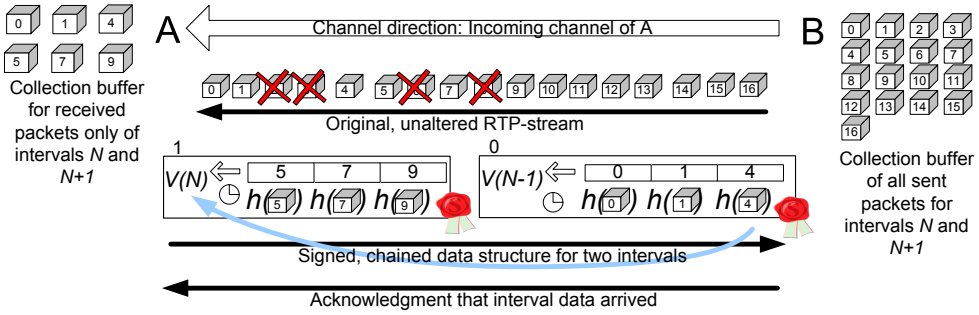


Fig. 4.7: Schematics of signing protocol for the direction B⇒A

The communication for signatures for the channel B⇒A is a little bit different. Again A and B both collect all packets for the current interval. But this time A decides on the precise point in time at which a new interval starts. Then A takes the collected and sorted packets which are exactly the packets that were not lost in the transmission from B to A, creates hashes over them, and sends them in a signed interval data structure to B. When B receives this it has to send a short acknowledgment information for this interval to A. If A does not receive this before a timeout occurs, he will resend the signature package.

Until the acknowledgment is not received, A will not start a new interval and the interval timer is temporarily suspended. This process is sketched in Figure 4.7.

4.9 Test criteria for verification of signatures

This section describes the necessary tests to verify that signatures are valid. In the signing scenario, A and B must performs tests continuously and employ timeouts to mitigate certain attacks and maintain a correctly signed, authentic, non-reputable call continuously. Additionally, a tool to verify the data file containing the archived call must be developed. This tool can be used anytime by any user or even an expert witness in front of court to verify that an archived call is valid and correctly signed. The need for such a tool is a sure requirement, because the data format is not a simple signature over a complete document like with SMIME-emails, which can be verified with many email-clients. Instead things like the hash-chain or packet loss must be checked. If these additional checks succeed, this leaves the problem of checking the embedded PKCS#7 signatures according to common x509-certificate and PKI-rules. Further discussion and algorithms to do so are discussed in RFC 3280 [37]. A common solution is to install a list of trusted root certificates on the verifiers computer and to roll out valid certificates to all signers. It can be envisioned that X509-certificates can be extended to not only secure the identity of X.501-names, internet hosts and email-addresses, but also SIP-URIs. But because SIP-URIs look like email-addresses, existing email-certificates can also be used.
In the self-signed archive scenario, no additional data is transferred between both parties. Therefore the most basic scenario of the archive contains no online verification: Only the mentioned tool will perform checks and it can be invoked at any time on an archived call file.

4.9.1 Common checks on the file format for both scenarios

These checks are performed by the verification tool and by B in the signing scenario (A is the person who signs the call):

CHK1 Checking whether the first interval with the meta data is correctly signed by the external time-stamping service. It must also be checked that the time of the timestamp is close to the start time of the call because this is the basis for cohesion and forms the temporal context. If there is an

additional third party proof of the time of communication (like an itemised bill from the phone company) this can as well be compared to this initial timestamp manually.

CHK2 Validating the PKCS#7 signature of each interval. As only the first interval contains all certificates to build the full certificate chain, these must be cached for verification of all other interval signatures.

CHK3 Verifying interval chaining. This can be easily done by storing the hash of the last interval and comparing it to the embedded hash value in the current one. If they do not match the chain was broken, the call is invalid from that point and an attacker might have to cut or rearranged sections of the call.

CHK4 Checking packet loss by checking the absolute sequence numbers in the interval structure. If the packet loss is above the QoS threshold, the call should be assumed to be invalid from this point, because understandability could not be maintained and a conforming archiving software would have terminated the call at that point. (Note that this is a matter of policy.)
A good QoS threshold is 5% packet, which still ensures good understandability.

CHK5 The borders of the intervals are checked against the respectively next interval, ensuring that there is no gap larger than few packets.

CHK6 Checking the consistency of the RTP packets: Timestamps and sequence must be strictly monotonically increasing with no overlaps.

CHK7 Checking the temporal integrity of the RTP packets, i.e., whether the time-stamps and sequence numbers stored in the RTP protocol, which can suffer from overflows and rollovers, are consistent with the time recorded in the interval. Also it must be possible to reconstruct the sequence numbers so that they match the recorded absolute sequence numbers.

CHK8 Checking against duplicate or replayed packets by using the algorithm form SRTP.

CHK9 The intervals of both channels are checked to have matching time so that they cannot drift away from each other and break cohesion.

4.9.2 Additional checks for the signing scenario

In the signing scenario the same checks are performed online by the receiving party B. If anything fails, B immediately detects this, optionally informs the user and terminates the call. This is in contrast to the archiving scenario where a conforming implementation of the archive would always result in valid calls, but where only the verification tool can verify the correct archiving of calls and conformance to policies and the rules and formats presented here.
Of course, if B uses a non-conforming implementation he could create invalid archived calls. But this would provide no advantage as the evidentiary value

would be lost.
In the signing scenario, CHK4 should be modified so that B will abort archiving if a QoS underrun is detected. With the most secure policy he would also forcefully terminated the call. A proxy-implementation could easily do this by injecting a BYE command and terminating any other SIP and RTP forwarding.
A also needs to check packet loss as discussed in section 4.2. If he detected a QoS-underrun, he should stop signing immediately.
An additional check is possible because the online scenario provides a real reference time source:

CHK10 Checking the time embedded in the intervals whether it drifted not more than two times the interval duration from the internal clock of B.

Note that all DOS-attacks emerging from this protocol can be easily mitigated by using SRTP and its HMAC-based authentication. Otherwise an attacker who is able to spoof UDP-packets could inject packets or signatures violating these checks and therefore terminate existing conversations.
With these policies a consistent approach to fulfill the requirements of integrity and cohesion was presented. In view of scalability and efficiency, by adjusting the interval duration a trade-off between required computational power and the maximum unprotected time can be adjusted. But the default time of twice the interval time of 1 second is believed to be already sufficient to make it very hard to change the meaning of a signed conversation. An attacker should not be able to forge more than two seconds at the very end of the conversation before the call is forcefully terminated. It should be noted that all described checks can as well be performed in a forensic evaluation, since all the data on which checks are performed is signed and secured.

5

Discussion of possible ways of transporting signature data

To solve the requirement of the signing scenario to provide a secondary bidirectional communication channel between A and B (see section 4.3), several methods can be used. Also the signaling of the presence of a signing service must be announced. This chapter presents several ways to accomplish this. Some of these methods were implemented in the demonstrator to make experiments with various SIP-providers, namely GMX, 1&1 and SipGate:

Using SDP for signaling As stated before, the `k`-value of SDP which is reserved for keys would be a logical match for transporting e.g. a public key. Therefore extending this to the whole starting interval, which is also timestamped, should be okay. Tests with the above mentioned SIP-providers showed that the line with the `k`-value was transported through the SIP-providers without problems and even transporting illegal SDP (like a line with `Y=`) was handled without problems.

An advantage of this solution is that signaling happens as early as possible, i.e. before the call is picked up. A disadvantage is that it can be filtered by the provider, but little can be done against that anyway. A more serious problem is the amount of data that can be transported: Tests showed that up to 14kB could be transported, but this would lead to serious UDP-fragmentation. This could be mitigated by transporting only the information that the service is present and not the actual first interval, which would be transmitted with another way.

SIP-SUBSCRIBE and NOTIFY As seen in chapter 3, this is a general way to subscribe to and signal any kind of events between SIP-clients. It is defined in RFC 3265 [18]. Currently it seems to be used mainly to control a LED for a waiting message indicator.

SUBSCRIBE and NOTIFY could be used to find the signing service and to subscribe to its events. They could even be used to transport the actual signature data, which would be forwarded by the SIP-proxies and SBCs, if present.
While this approach seems to be very standard-friendly at first, a problem is that the event-framework was only designed to transport events that happen rarely. RFC 3265 explicitly states that this mechanism is not expected to be used for rapid frequencies, because this could overload SIP-networks. Still this method would be ideal to start signing in the middle of a call.
Tests showed however that none of the tested providers transported SUBSCRIBE-messages between SIP-clients. NOTIFY was blocked by SIPGATE.
Another disadvantage of this approach is digest-authentication. To protect against SPAM and abuse of services, SIP-proxies usually force authentication for REGISTER and INVITE. It is to be expected that they enforce this for SUBSCRIBE, INFO and MESSAGE, too. In that case, the proxy would have to know the password of the client and loose its universality.

SIP-INFO SIP-INFO allows for the carrying of session related control information that is generated during a session. One example are DTMF-dialtones, which can be transported using this method. The soft phone X-Lite uses this method. This extension to SIP is defined in RFC 2976 [20].
This approach is similar to SUBSCRIBE and NOTIFY, because it could overload the SIP-network. On the other hand, it is less likely to be limited by throttling or digest authentication because it is already used for dialtones.

SIP-MESSAGE RFC 3428 [19] proposes the MESSAGE SIP-method to transport instant messages over SIP. MESSAGE requests do not themselves initiate a SIP dialog. Under normal usage each instant message stands alone, much like pager messages.
This method also has the disadvantage that it might overload SIP-networks. Because of the SPAM and SPIT or –more specific– SPIM-problem it is very likely to be affected by digest-authentication. On the other hand it is less likely to be blocked by SIP-providers because chatting has much more demand and appeal to end users than e.g. a general purpose event-framework.
The biggest disadvantage of this method is that it is a clear abuse of the intended usage of instant messages.

Custom UDP or TCP data stream Signatures could be transported by creating a completely custom protocol that works parallel to the SIP/RTP-communication. An advantage is that there would be no restrictions of size and contents of the transported data. If a signature-service is not present, such incoming connections would simply be blocked by the network stack if no process is listening to the assigned port. This would reduce potential compatibility problems to a minimum.

A strong disadvantage of this approach is the whole topic of NAT-routers and session border controllers (see section 3.6). SBCs would not help to reflect custom TCP/UDP traffic from two users each behind a NAT. Such a solution would need to deal itself with the prevalent NAT-routers and may even have to supply its own reflection servers.

In contrast, all following approaches would exploit the existing mechanisms and optional data reflection services like SBCs for the transportation of signature data, if they don't block it by detailed traffic analysis

RTP with different SSRC In general it would be favorable to use RTP instead of a completely new protocol to transport signature data. This can be done by sending the packets using the same RTP-port as the existing speech channel, but with a different value in the synchronisation source field of the RTP-packets to differentiate between signature data and speech data. An advantage of this approach is of course that it uses the same RTP port and therefore benefits from any redirection services, port forwarding or STUN-servers, which might be provided only for one port-pair. An advantage might also be that systems manipulating RTP like mixers can see that the signatures are coming from another system and leave them alone. But on the other hand the RTP-standard does not support any kind of multiplexing of different kinds of data and other systems might not understand the signature data.

In experiments it was shown that SipGate and GMX both do not modify or process RTP-streams for pure internet based calls.

Multiplexing using different codec In principle, one RTP-stream could transport data encoded with up to 127 different codecs. Therefore it would also be possible to multiplex speech and signature data with the payload-type field instead of the SSRC. This is already done with DTMF-dialtones (see RFC 2833 [16]). An advantage of this approach would be that devices not understanding the signature pseudo codec should simply ignore it. An strong disadvantage of

this approach is that the RFC standard explicitly forbids any kind of multiplexing, e.g. for video and audio and SDP is not able to represent multiplexing. Instead, multiple RTP-streams should be used. On the other hand, signatures are not a different kind of medium, but an additional security property of an existing stream.

RTP Header extension As seen in section 3.3.2, RTP supports experimental extensions. These could be used to piggyback signature information with some of the regular RTP-packets. This must also contain a flag whether the original payload should be processed or not because silence suppression could lead to one of the RTP-channels being "silent" for a period of time. An RTP-extensions would clearly be ideal if every packet was to be signed with its own signature instead of using intervals. And with intervals it would also have the advantage, that these would be piggybacked on UDP-packets that would be send anyway, so that additional UDP-overhead for signatures would be saved.

An disadvantage of this method is the incompatibility with existing RTP-implementations and the lack of a global registry to register magic numbers for RTP-extension types.

New RTP-stream and port SDP supports multiple RTP-streams per SIP-session very well. As seen in 3.2, RTP-streams even have an associated type of "audio", "video" or "application". A new RTP-stream of type "application" can be added to a normal audio and/or video conversation.

An advantage of this approach is that signature data is clearly separated from e.g. audio streams and the different ports can even be opened by different processes. While providers might be able to filter out all but one RTP-stream and not provide reflection for these services, on the other hand the existence of video conferences in SIP-clients seems to be a supported scenario desired by customers. This mandates support for more than one RTP-stream.

The OpenSer software used by SipGate supports a feature named NatHelper that redirects UDP-traffic to special public servers that can be reached from behind a NAT. NatHelper provides these reflection-services for up to eight RTP-streams per session and therefore works with this approach. Then again, other providers might handle this differently.

The solution of choice here is to use the `k`-value of SDP for signaling and a separate RTP-stream of type "application" for the signature data. Therefore the proxy implementation is not concerned with knowing the user's password or doing digest authentication.

6

The supplied software

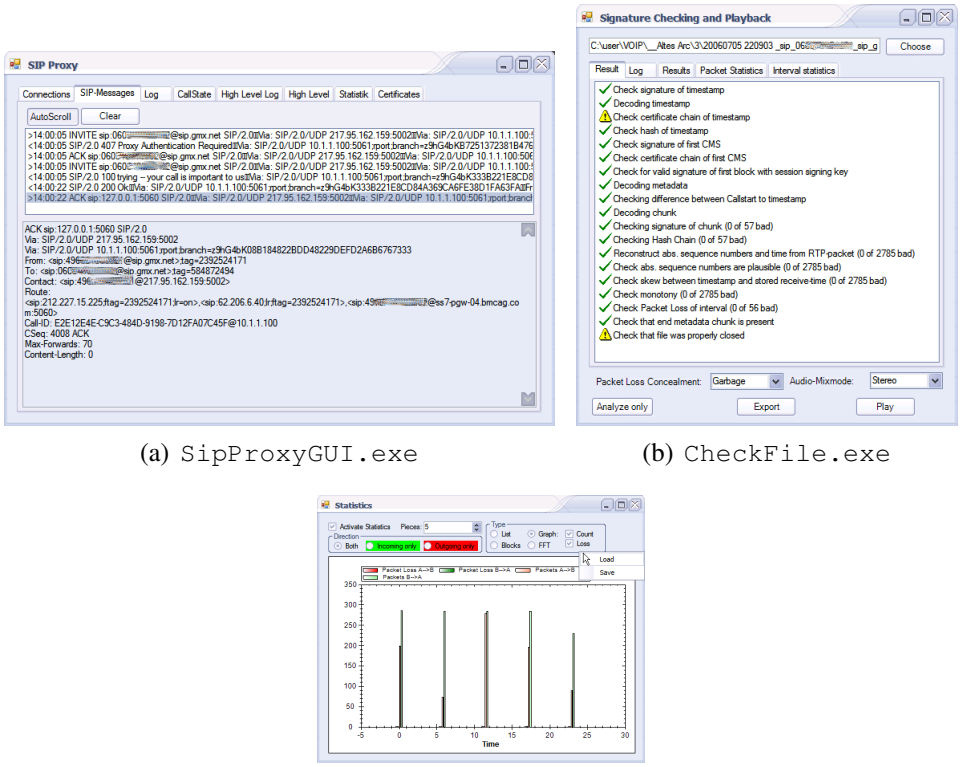
This chapter describes the software implemented for this thesis. The supplied programs are:

`SipProxyGUI.exe` This is the main program, a SIP proxy server which redirects SIP and RTP packets to itself to allow for passive archiving and active modifications.
It contains a testbed for signaling and transporting signing data in parallel to calls and implements the complete self-signed archive scenario.

`CheckFile.exe` This is the verifier that checks signed calls for integrity, trusted signatures and cohesion. It also allows playback of archived calls and export as WAV-files, e.g. for submitting calls to an forensic expert witness.
It also contains analysis and statistics tools for packet loss to encourage further, more detailed experiments and analysis for the optimal value for QoS-abort beyond the compromise of a 5%-boundary presented in this thesis.

`Statistics.exe` This is a subset of `CheckFile.exe` with statistic and analysis functions only. `CheckFile.exe` is able to export the list of packets with their jitter and contents as CSV-files. This program can import them and analyse them independently of archived calls.

Screenshots of the three programs can be seen in Figure 6.1. All programs are currently based on the Microsoft .NET-framework [38] and run under Microsoft Windows and all programs show a GUI. But this is no limitation, because GUI and implementation have been strictly separated. The proxy itself also runs under Linux using the Mono Project [39]. The GUI of the proxy is only used to monitor and debug SIP-connections and the signing and archiving process. This is not needed if the proxy is used in a server-scenario where it is installed on a headless machine, e.g. in a corporate environment.



(a) `SipProxyGUI.exe` (b) `CheckFile.exe`

(c) `Statistics.exe`

Fig. 6.1: GUIs of the three supplied programs

The following steps are necessary to run the software:

- Install the “Microsoft .NET Framework 2” from Windows Update.
- (optional) If the programs should be recompiled or modified, install the free “Visual C# 2005 Express Edition”.
- If calls should be signed, at least one valid certificate must be installed in the Windows Certificate Store. The private key must be available. For chip cards and chip card readers, compatible drivers must be installed.
  You can get your certificate from e.g. your network administrator, institution or commercial vendors like `Verisign.com`, `Geotrust.com` or `Thawte.com` or the free `CaCert.org`. Software and instructions for generating self signed certificates is out of scope for this thesis.
- If signed calls should be verified to be completely valid, trusted root certificates must be installed in the Windows Certificate Store if not already present. Otherwise verification will fail.
- If you want to archive conversations, a SIP-account must be available. By supplying your German bank info you can get one from `gmx.de`. Or

get one from Sipgate.de, which even includes a German telephone number. Both are free unless you call numbers from the old telephone system or other providers.

- If you want to archive conversations, a SIP-compatible soft phone or hardware device must be available and reconfigured to use the proxy server. This is explained below for the free soft phone "X-Lite" from xten.com.
- The timestamping service must still be in operation on my server or you must setup your own by simply copying all files from the folder "TimeStampService" in a new virtual directory of an Microsoft "Internet Information Server" (IIS) on a server where the .NET-framework 2 is installed.

6.1 Handling of certificates

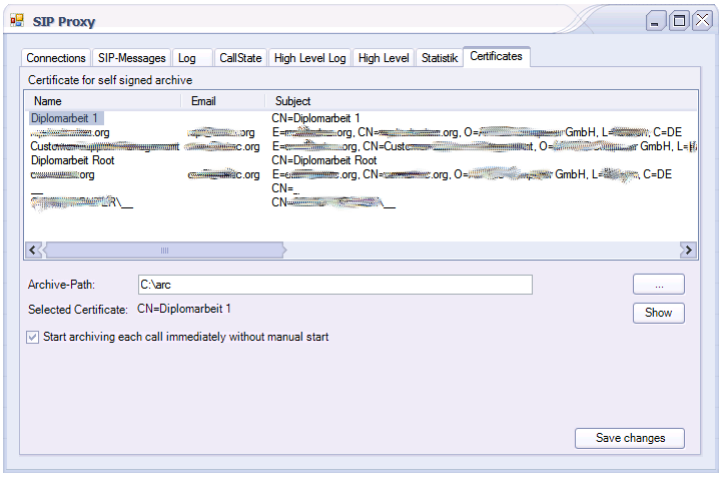


Fig. 6.2: In the proxy-server, an certificate can be selected from the Windows Certificate Store

As the .NET-framework has extensive classes for certificate handling and PKCS#7, the program can hand of the actual verification task to the underlying crypto provider. Under the Windows operating system, this is Microsofts Crypto-API. For the self-signed archive, the user can choose a certificate from the Windows Certificate Store as seen in Figure 6.2. Under Windows, verification of PKCS#7 signatures and certificates is also done using the Microsoft Windows Crypto API, so that trusted root certificates from certificate authorities should be installed using the system command "`certmgt.msc`": Click Start, click Execute and type this in to manage certificates.

6.2 The TUD-chip card as example for a secure token

(a) The front-side of the TUD card

(b) The used cardreader "Kobil KAAN Advanced"

Fig. 6.3: Equipment for using a smartcard to sign a call

Another interesting feature of the developed software is that it supports chip cards and card readers, e.g. the TUD chip card [40] shown in Figure 6.3(a), which has been rolled out and distributed to all students of the Technische Universität Darmstadt in 2006.
The recommended card reader "Kobil KAAN Advanced" [41] shown in Figure 6.3(b) comes with driver software which directly plugs into Microsoft's Crypto-API and install the certificate supplied by the TUD into the Windows Certificate Store. This of course leaves the private key on the card and blocks any attempts to access the private key directly or use it outside the card inserted into the plugged in cardreader.
In the demonstration environment the TUD card was used to sign the archived conversations. Only a tiny modification had to be done to the software: It would be very impractical for the user to enter the PIN-number every second in the cardreader to confirm the signing of every interval separately. Until a programmatic way to supply the PIN number to the cardreader is found, the following workaround is used: At the beginning of the conversation, a RSA-keypair is generated. The public key is signed together with the first interval using PKCS#7, the whole certificate chain and the chip card. The private key is kept until the termination of the call when it is wiped from memory. All subsequent intervals are not signed with the chip card, but with the RSA-private key.
It is unfortunate that the chip card – which naturally is very slow and needs about 2 seconds for every signature– could not be used to demonstrate how well the presented concept adapts to limited computational resources for signing.

6.3 Compilation

This section describes how the program can be compiled from the supplied source code.
Download the free "Visual C# 2005 Express Edition" from http://msdn.microsoft.com/vstudio/express/visualcsharp/ and install it. Then open the solution "SipProxy.sln" and click "Build Solution" in the "Build" menu. Afterwards you'll find the executable files together with all necessary DLLs next to it in the subfolders "SipProxy\bin\Release\", "\CheckFile\bin\Release\" and "Statistics\bin\Release\".

6.4 Configuration of soft phones to use the proxy

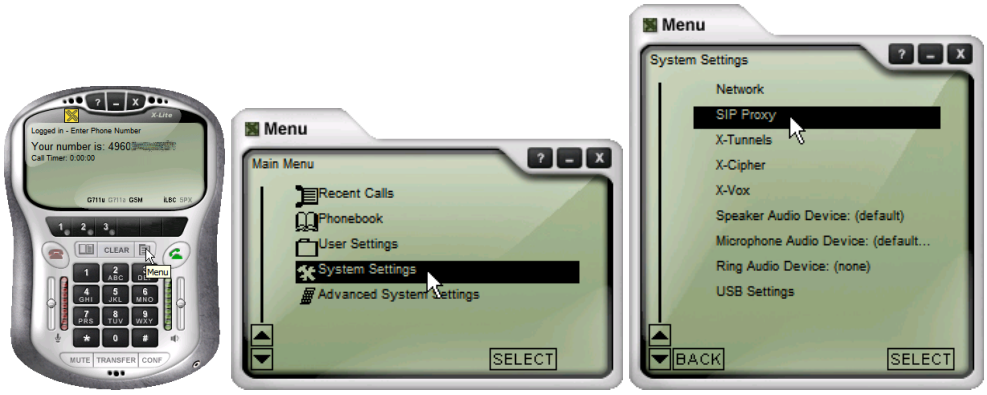
The main role of the proxy is to intercept SIP-packets and modify embedded IP-addresses to redirect the audio-stream to go through itself for signing or archiving. It supports multiple parallel calls with multiple SIP-clients/soft phones to several SIP-providers (original outgoing proxies). SIP-clients must be reconfigured to use this proxy as outgoing proxy instead of the original outgoing proxy from their SIP-provider. If they are installed on the same computer, this new proxy should be configured to localhost (127.0.0.1) and the appropriate port from Table 6.1: The main way for a SIP-client to specify which original outgoing proxy it wanted to contact and what should happen to the call (i.e. archiving or signing) is by using a different port number of the ones the proxy listens on.
In the demonstrator e.g. 127.0.0.1:5060 is a proxy that archives calls and redirects them to sip.gmx.net:5060, while 127.0.0.1:5080 archives calls using SipGate.
If both programs are used on the same computer, port conflict may happen:

Proxy Port	Action	Provider	Original outgoing SIP-proxy of the Provider
5060	Self-Signed Archive	GMX	sip.gmx.net
5081	Self-Signed Archive	SipGate	sipgate.de
5082	Self-Signed Archive	Web.de	sip.web.de
5083	Self-Signed Archive	Freenet Phone	iphone.freenet.de

Tab. 6.1: Default configuration for listening ports of the proxy.

While SIP-clients usually find a free port if the default port 5060 is already occupied, the SIP-proxy relies on the fixed, known port numbers shown in Table 6.1. Therefore it is best to start the proxy before the soft phone or to

reconfigure the table entry for GMX.
The configuration of a widely-used free soft phone named “X-Lite” is shown in Figure 6.4: “X-Lite” was setup with a normal account of GMX (domain, username, password), only the outgoing proxy was changed.
On http://www.sipgate.de/faq/index.php SipGate has an extensive list of instructions for various devices. You can use these instructions to setup your phone if you substitute the outgoing proxy with the one from Table 6.1.



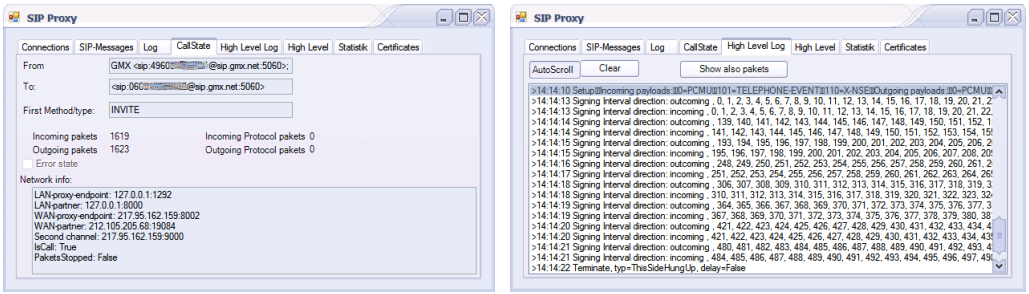
(a) Click the menu button (b) Choose system settings (c) Click SIP proxies

(d) Configure the first proxy (e) Change the outgoing proxy according to Table 6.1

Fig. 6.4: Configuration of the X-Lite soft phone

6.5 Instructions for the proxy

Figure 6.1(a) shows the first screen when the proxy is started. Here one sees a tree where each top node is one of the four default proxy servers. The nodes below these nodes are for every connected client. And for every client-node, its child-nodes show different sessions or other SIP-actions, separated by Call-ID. If a SIP-client was reconfigured to use the proxy correctly, at least an entry for REGISTER should appear where the phone announces its address to the provider. Each real phone call will result in another entry. If a failure (e.g. exception) occurred, there will be a X in front of the call. In that case, please look at tab of the main window named “log”.
Before making the first calls, you should choose a certificate as seen in Fig-



(a) Debug information from the basic proxy, e.g. which RTP-ports are used for redirecting

(b) Debug log from the highlevel component “Self Signed Archive”

Fig. 6.5: Debugging information of the proxy

ure 6.2. In the same tab there is also the setting for the archive path. Each call will be written to its own file in that directory.
After setup, you can make the first calls either with the soft phone or by calling the associated telephone number of your SIP-account with the old telephone system. Both outgoing and incoming calls will appear on the first tab as new connections. Double click on it and you will see the tab “CallState” from Figure 6.5(a). It shows counters for packets and which new RTP port numbers were allocated by the proxy to redirect RTP-packets for this call to itself.
The tab seen in Figure 6.5(b) is reserved for the higher implementations that are base on the basic proxy, in this case the self signed archive. Here it shows a log which absolute packet numbers were archived in which interval and which events got passed to this component by the basic proxy.
The last interesting tab is “Statistics” which is explained in section 6.7.

6.5.1 Configuration of the proxy

Several aspects of the proxy can be configured by modifying the XML-file "`SipProxyGUI.exe.config`" in the program directory. But the default settings should be fine.

`MaxPaketsbeforeStopHighLevelLog` This is the maximal amount of packet related log entries that are logged. After this amount is reached, logging of packet related data is suspended because RTP-streams could generate lots of log-entries.

`SelfSignedOwnCert` The SHA1-thumbprint of the archive certificate. This is better changed through the GUI. Settings changed in the GUI are stored in another XML-file in the user profile and take precedence over settings in "`SipProxyGUI.exe.config`".

`SelfSignedArchivePath` Path for storing signed, archived calls. This should also better be changed using the GUI.

`TimestampUser` Username for accessing the timestamp-service.

`TimeStampPassword` Password for accessing the timestamp-service.

`SipProxyHelper_timestampservice_Service` URL of the SOAP-endpoint of the timestamp-service. When I stop to operate the primitive timestamping-service created for this thesis, you'll have to change this setting so that it points to your installation of the timestamp-service.

`AutostartArchiving` Whether archiving should start at the beginning of each call or only after the user chooses to do so.

`UseSessionSigningKey` If this is true, then only one PKCS#7 signature is used at the start of the call as described in section 6.1. This is done to support chip cards.

`replaydetectWindow` This configures the size of the sliding window for the replay detection algorithm used to detect duplicate packets. This value can be up to 64 because the replay window is implemented as 64-Bit sliding window.

`SignIntervalMs` The length of an interval, or more precisely: The timer-value until a new collection of packets is started and the old ones are signed. This value is in milliseconds.

`MaxLookaheadWindowTime` The amount of time that packets are allowed to stay in the collector even if a new interval has started. This is because packets can arrive in a different order and the collector sorts them by sequence number. In and between intervals, the absolute packet numbers must be strictly monotonic increasing. If now a new interval takes all collected packets from the collector immediately, then late packets arriving out of order would have no chance to be sorted correctly and would need to be dropped. Therefore this value is the number of milliseconds a

packet may still stay in the collector after a new interval has started. This value should be a bit larger than the maximum length of the jitter buffer of the SIP-client.

`MaxLookAheadWindowSize` The same like `MaxLookaheadWindowTime`, but this time for the number of packets that can stay in the collector buffer. Packets must satisfy both criteria to stay, so this value can be identical to `replaydetectWindow` as later packets would be dropped by the replay window anyway.

`MaxSkew` The maximum allowed drift between internal clock and reconstructed packet time from the RTP-timestamps.

`MaxPacketLoss` The QoS-underun criteria for packet loss: A value of 1.0 means that no packet loss is allowed while a value of 0 allows all packets to be lost. The default value is 0.95 which results in a maximum of 5% packet loss before the remaining parts of the call are marked invalid and the call is terminated.

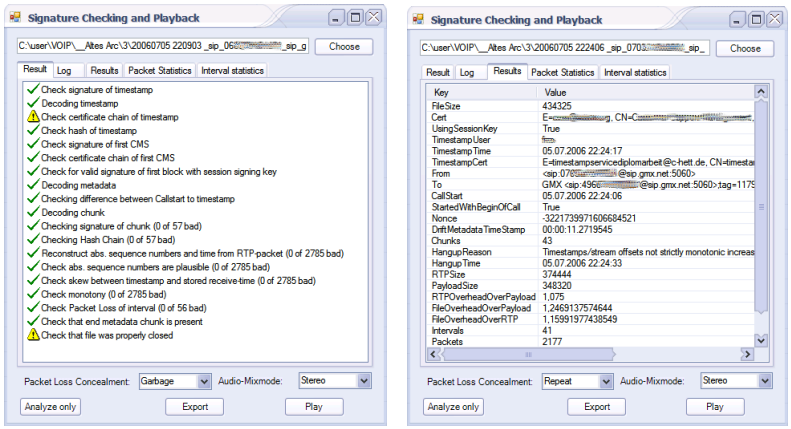
`Policy` This describes what to do when a failure occurs:

- `DropPacketsIfPossible` will silently discard any packets that raise any kind of error, e.g. to much drift from the internal clock.
- `Ignore` passes those packets along.
- `AbortArchiving` stops the process of archiving, but lets the call continue.
- `AbortCall` not only stops archiving and any forwarding of packets, but terminates the call by sending a SIP-BYE request to both parties. This is the recommended policy for maximum security.

6.6 Instructions for the call verifier

The call verifier shown in Figure 6.6(a) is a tool to open, verify and play archived conversations. In the upper part of the dialogue, the file to open must be chosen. Then you can click one of the buttons “Analyse”, “Play” or “Export”. Analyse will only verify the file and generate statistics. Verification covers signatures, the hash chain, cohesion and QoS-criteria like packet loss. The other two buttons will additionally convert the recorded conversation into the PCM format. Export will ask for a location to export the file in the WAV-format while ‘Play” will play it over your soundcard. The two listboxes above the buttons allow to configure this process. You can choose how packet loss is handled:

Silence Silence is used wherever a packet is missing. As seen in section 3.5, this yields bad results.



(a) GUI of the verification utility

(b) Results display in the verification utility

Fig. 6.6: Results of the verification process

Garbage This experimental setting adds an ugly sinus tone wherever a packet is missing to maximise the bad effects of packet loss.

Noise This experimental setting adds random noise (with low volume) instead of missing packets. This should yield better results than Silence.

Repeat This is the default value and method of choice for best results without expensive wave reconstructions: Lost packets are replaced with their preceding packet which therefore is repeated. See section 3.5 for the performance of this method.

The other box "Audio-Mixmode" allows to configure how both channels of the conversation are reconstructed:

Mix Mixes the audio samples from both channels by adding them.

AOnly Allows to listen to only the isolated speech from A to B.

BOnly Allows to listen to only the isolated speech from B to A.

Stereo Produces stereo WAV-files respectively plays the conversations using both speakers. One output channel is used for A⇒B, the other one for B⇒A. This can assist a forensic expert witness in analysing the call because the resulting WAV-file with retain as much information as possible by not mixing both channels. It is ideal or further processing or analysing in audio editors.

The main window of the first tab shows the progress and successes/failures of the verification process. A green playback symbol means that the program is processing that task. A green checkmark means that verification was okay for this entry, while a red cross indicates a failure. Yellow warning-symbols indicate problems with e.g. signatures. If you double-click the entry, you will

be able to see and verify the certificate chain or see other problems with the certificate. In the screenshot, the root certificate for the timestamp-service was not installed properly. Also the file was not properly closed because the proxy was terminated during the conversation, which leaves a still checkable file.
The next tab just shows a log with details about the verification process. More interesting is the third tab, shown in Figure 6.6(b): It shows detailed information what was in the archive file, e.g. caller and calle, which user authenticated to the timestamp service, the call-start and difference to the timestamp and the reason for termination of the call.
It also allows to measure the overhead resulting from the signing in relation to the overhead caused by the RTP-packets alone. For a short call, the RTP-format added an overhead of 7.5%. On top of that, signatures added 15%. Note that this is drastically reduced for longer calls as the initial metadata carries large X509 certificate chains from the archiver and the timestamp service.
The remaining tab contains statistics.

6.6.1 Configuration of the call verifier

The call verify also has a config file in the XML-format. It has the name "`CheckFile.exe.config`" and is also located in the program directory. It contains the following entries:

`MaxSkew` See section 6.5.1.
`replaydetectWindow` See section 6.5.1.
`MaxPacketLoss` See section 6.5.1.
`MaxMetadataDrift` The maximum timespan allowed to pass between the date and time of the timestamp from the trusted timestamp authority and the recorded start of the call.
`MixMode` The default value for packet loss concealment which can and should be easily changed using the GUI described in the preceding section.
`PacketLossConcealment` The default value for packet loss concealment which can and should be easily changed using the GUI described in the preceding section.

6.7 Instructions for the statistics module

The statistics module is actually shared between all three programs: `SipProxyGUI.exe` can show live statistics during ongoing conversations and

`CheckFile.exe` shows them whenever an archived call is processed. Both programs can export CSV-files with details about the packets. The separate `Statistics.exe` program can import and display these and is a direct wrapper around the shared module which is contained in the platform specific `GUITools.dll`.
In Figure 6.7 the available modes are displayed. To import and export the data

(a) Click the menu button

(b) Choose system settings

(c) Configure the first proxy

(d) Change asdsa

Fig. 6.7: Screenshots of the statistics module

as CSV-file, please right-click in this statistics subwindow and choose “Load” or “Save”.
Figure 6.7(a) shows the raw data on which statistics operate: For every RTP-packet it shows the RTP-sequence number, the reconstructed absolute sequence number starting from 0, the exact time when the packet was received over the network, the RTP-timestamp from the packet, the reconstructed timestamp and the drift between the receive time and the reconstructed timestamp. The colour of the line indicates the direction (A⇒B or B⇒A) of the channel the packet belongs to. The list can be easily filtered by direction.
Figure 6.7(b) shows how the packet loss can be visualised using blocks: This

allows for a quick visual overview for patterns of packet loss. One can see whether packet loss was in bursts or if only single packets were affected. The blocks are in the colour of the direction of the channel or have the mixed colour yellow if these sequence number was present in both directions. At least at the beginning of the conversation one can easily see whether packet loss in both channels is related. Later the associated times will differ because of silence suppression: This diagram only displays the sequence numbers and not associated time. For that the remaining display modes are recommended.
Figure 6.7(c) shows a diagram of packet loss and number of packets over time. What is shown can be selected by clicking “Count” and “Loss” and selecting a specific direction or both. The diagrams are drawn using the free ZedGraph library [42]. Note that you can zoom and export the diagrams with the context menu displayed after right-clicking.
The X-Axis always shows the time, while the Y-axis shows packet loss between 0 and 1 and the number of total packets in that timespan. The number of bars and therefore the granularity can be chosen my modifying the value “Pieces”.
Finally, Figure 6.7(d) shows a fourier analysis of the packet loss over time. This was done to find correlations and patterns in packet loss and to detect whether packet loss results from network outtakes or attacks. So far in the available network connections packet loss was far too low to find meaningful values, but further studies are needed.
There is also another statistics tab available, namely “Interval Statistics”. It

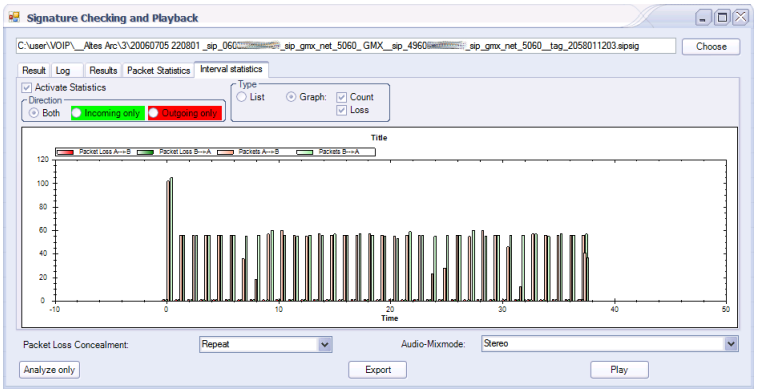


Fig. 6.8: Analysis of packet loss by interval

is shown in Figure 6.8. It does not display packet loss by linearly segmenting the call time in pieces, but by showing a bar for every interval and showing the packet loss for this interval.
The statistics module has no configuration files.

7

Implementation details and classes

This chapter describes how timestamps are created and explains all relevant classes of the supplied software. Some implementation concepts are explained in more detail.

7.1 Timestamp Service

As seen in chapter 4, the security of an archived call benefits from a trusted third party timestamp service. Because commercial timestamp services charge up to 1$ for every timestamp, are hard to find and entering and negotating a contract with them would cost an enormous amount of time, it seemed appropriate to just simply create one myself. It is currently hosted on a server I operate and is realised as a webservice based on the SOAP-protocol. The signatures use PKCS#7 signed envelope. The service is protected by a primitive user/password-based authentication and contains an user database base on a text file with one username/password-pair per line separated by "`:`" in the file "`App_Data\Users.txt`". The SOAP interface to use this service consists of this simple method:

```
byte[] GetTimeStamp(byte[] Hash,string user,string password)
```

If the user and passwords match, the class `TimeStamp` is used to encode the hash, the current date/time and the username into a byte-blob. This blob is signed using PKCS#7 and returned encoded according to the PKCS#7 standard. The private key is stored together with the timestamp service certificate in `App_Data\cert.p12` using the PKCS#12 format. Additionally, the root-certificate is stored in `App_Data\hett-cacert.pem` and also encoded into the PKCS#7 signature.

7.2 SIP-Proxy

7.2.1 Helper classes and protocol implementations

`Utils` This class contains several utility methods, e.g. to encode data into blobs. It should be noted that through the whole implementation, datastructures and protocols are usually encoded with the standard classes `BinaryReader` and `BinaryWriter`. These classes store simple datatypes in a common way, e.g. integers are simply stored with 4 bytes in big endian format. To guarantee a simple and operable data format, strings and byte arrays are not stored with proprietary serialisation frameworks. Instead methods like `WriteByteArray(BinaryWriter wr,byte[] buf)` simply store an integer with the length of the byte array encoded as 4 bytes, followed by the byte array itself.
For a more efficient implementation, the ASN.1 standard and DER encoding rules could be used.

`RTPPacket` This class stores all properties of a RTP-packet, e.g. SequenceNumber, Timestamp and the byte-array with the payload itself. It has a constructor that parses a byte-array according to the RTP-standard and it can also encode its data back into the RTP-format. Hooks are provided –although currently unused– to implement SRTP with the methods for producing and checking the authentication-data and encyption and decryption. See section 3.3 and RFC 3550 [12] for the format of RTP-packets.

`SDPParser` This class stores all properties of a SDP-body. It can be initialized with a string containing a SDP-body and can also reencode its –modified– contents back into a string.
Its public property "`Medias`" contains a list of `MediaDescriptions`, one for each defined RTP-stream. Those again each store in addition to port, media type and encryption key a dictionary of payload types. This dictionary maps values from the range $0 - 127$ to `PayloadFormat`-objects where clock, number of channels and a string with the codec name are stored. Further this class transparently handles the list of static predefined payloads from RFC 1890 [21]. A helper method to restrict the number of allowed codec-types is also provided.
See section 3.2 and RFC 2327 [17] for details about SDP.

`SIPParser` This class contains a SIP-parser, can store a list of SIP-headers, the SIP-method and destination URI or the response code and message and also reassembles modified data together with the body-data back to an SIP-packet.

Unlike the other parsers, `SIPParser` does not represent all parts of a SIP- packet, but mainly header-lines. But it allows easy replacement of headers or adding and removing headers that can appear multiple times like VIA or Record-Route. Properties for From, To, Call-ID and an enumeration for SIP-methods are provided.
SIPParser can also construct an response for a given request and create new SIP-packets. See section 3.1 for a description of the SIP-protocoll.

G711 This class contains static methods to encode and decode μ-law and a-law encoded voice data into PCM 16 Bit. It implements the formulas 3.1 and 3.2.

STUNPacket This class contains a minimal STUN-implementation, which does not carry out the full range of NAT-detection, for symmetric NATs it will simply not work. It supports detection of the outside binding of an opened RTP-port, which it returns as external IP-address and port-number stored in an `IPEndPoint`-structure.
It also omits support for authentication, because such a server could not be found. Instead a vast amount of free, unrestricted STUN-servers is currently available in the wild, e.g from United Internet, SipGate or Google Talk. See section 3.6 for details about STUN and the necessity to implement it.

UDPListener This class provides a handy wrapper around asynchronous UDP-sockets and STUN: It can either listen on a given port-number or search a free port-range for an unused port. If configured to do so it will even determine the external mapping (IP-address and port) of this port behind a possible NAT-device.
The socket is used asynchronously and whenever a packet is received, the public event `ReceiveCallback` is invoked if somebody subscribed to it. On the windows operating system `UDPListener` does not waste a permanent thread, instead a thread from the thread pool is used whenever a packet is received.

AudioChunk This class works with short pieces of audio data and can decode and encode them in the WAV-format. It can also play sounds and downsample them to half the sample rate.

FestivalClient This class can communicate by TCP-sockets with the LISP-based open-source software "‘Festival"’ which provides speech synthesis to PCM-audio.

MSSpeechAPI This class provides a wrapper around Microsofts Speech API which is only available on windows.

Injecting synthesized speech into RTP-streams for signaling to the user is not yet fully implemented.

`ConfigurationSettings` This class helps with the configuration settings of the proxy (all settings are stored in a XML-file with the extension `.config`) and is able to enumerate the existing network interfaces so that the proxy can bind to the default one. Alternatively the user can select a specific interface to bind to.

`Logger` Provides a simplistic common logging-service and a hook for GUI- or console based interfaces to subscribe to these messages.

7.2.2 The basic SIP-proxy

An important component of the demonstrator is the SIP-proxy which redirects SIP and RTP-packets to itself for further processing. It is well separated from the high level applications in section 7.2.3, because already two main application for the main technique exists which can be plugged in to the proxy. It is also possible to easily replace the SIP-proxy with a different implementation, because the main concepts could be applied to H.232 or IAX as well. In order to support this, two interfaces `ICallClient` and `ICallProvider` (see Figure 7.1 for a class diagram) are provided, which describe the contracts to be implemented by the highlevel plugin and by the proxy. This section describes the implementation of a SIP-proxy, while section 7.2.3 focuses on the implementation of the self signed archive according to the concept described in section 2.2.1.

The demonstrator does not implement a full SIP-Proxy as defined in RFC 3261 [11], because it is not concerned with e.g. finding the correct next hop by querying the DNS-system. It also does not provide call routing or registration support. Instead it is focused on proxying an existing outgoing proxy from a SIP-provider.

The main role of the proxy is to intercept SIP-packets and modify embedded IP-addresses to redirect the audio-stream to go through itself for signing or archiving. It supports multiple parallel calls with multiple SIP-clients/soft phones to several SIP-providers (original outgoing proxies). SIP-clients must be reconfigured to use it as outgoing proxy instead of the original outgoing proxy from their SIP-provider as seen in section 6.4. The main way for a SIP-client to specify which original outgoing proxy it wanted to contact and what should happen to the call (i.e. archiving or signing) is by using a different listening port number of this proxy.

`SipProxies` This class contains a list of `SIPProxy` objects, one for each local proxy that can be setup in a SIP-client. To add a new proxy a free local UDP-port, the address and port of the original outgoing

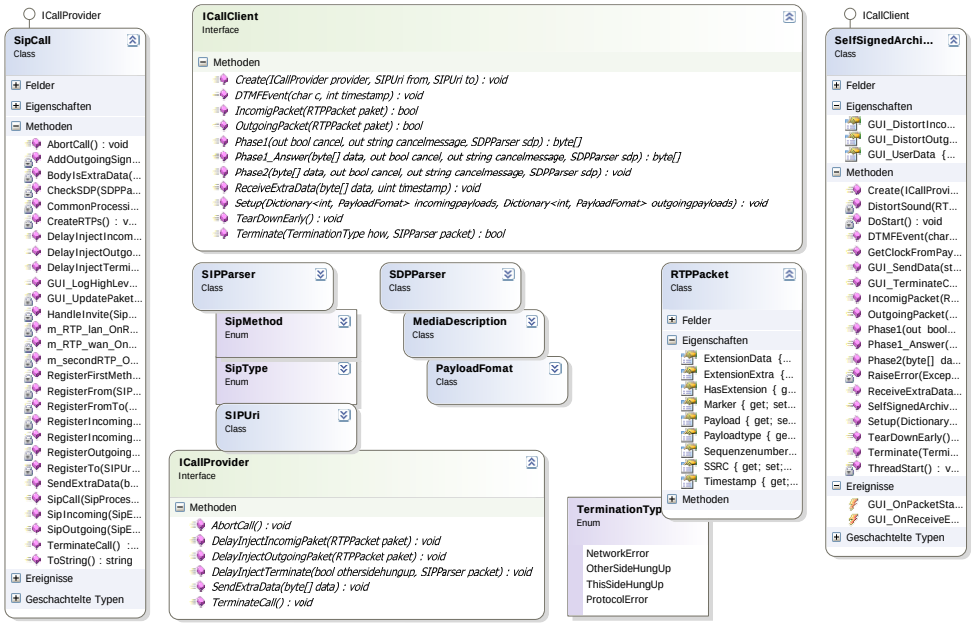


Fig. 7.1: This part of the class-diagram shows the interface between the proxy-implementation and the highlevel client that uses a proxy, e.g. for archiving a call. `SIPCall` is part of the proxy, implementing the `ICallProvider` interface. `SelfSignedArchiveCallEventsListener` provides a self signed archive for calls and implements `ICallClient`. The center of the diagram shows the helping classes like `RTPPacket`, which is passed between both implementations or `SDPParser`, which stores the important payload mapping. The enumeration `TerminationType` describes the reason for call-termination from the Terminate-method

proxy and a `SipCall.CreateCallChannelDelegate` delegate has to be specified. The delegate is a factory that creates classes that implement ICallClient. This pattern provides a flexible way of using the basic SIP-proxy with different implementations of `ICallClient` in parallel.

`SipProxy` One object from this class exists for each defined proxy. This class is derived from `UDPListener` and handles incoming SIP-requests from the clients. It does not communicate with the external network or the original outgoing SIP-proxies, but instead contains a dictionary of client-IPs which map to `SipProcessing`-objects. If a new internal client connects to this proxy-object, it will create a new entry in the dictionary.

SipClientConnection One object of this class exists for each used combination of internal SIP-client-IP and port and defined `SipProxy`. Thus perfect separation of the connected SIP-clients is guaranteed.
This class is derived from `UDPListener` and listens on an external reachable socket. With this socket it receives SIP-packets form the external network (e.g. the original outgoing proxy) to process them. Internal SIP-packets from the SIP-client are forwarded from `SipProxy` to this class.
The main task of this class is to trigger the events `SipOutgoing` and `SipIncoming` for every incoming and outgoing SIP-packet. Subscribers to these events can modify or replace the packet, but can also decide whether the packet should be dropped, forwarded or answered without involvement of the SIP-client respectively the external SIP-partner. These options are part of the enumeration `SipEventDestination`.
As seen in section 3.1.3, SIP has a tendency to not send SIP-responses to the originator of the packet, but to use VIA-headers. Furthermore, for follow-up SIP-requests (like BYE or ACK), SIP always tries to skip the intermediate proxy servers and establish direct connections. This of course is problematic for the implementation. Therefore, this class inserts it own IP-address and port (which is detected with STUN) into the contact-header, the VIA-header and the Record-Route-header. For responses, these headers are again removed.

SipProcessing This class derives from `SipClientConnection` and provides another service above: It contains a dictionary of calls which maps Call-IDs to objects of the class `SipCall`, which provides all methods of the `ICallProvider` interface.

SipCall This class handles calls, forceful termination and RTP redirection. It also provides a second communication channel using various methods as described in chapter 5. This is simplified slightly as this class is instantiated exactly once for every call by `SipProcessing`. Therefore it can have simple member-variables for From- and To- URIs and at most five sockets: For every direction one for communication with the internet/partner and one for the SIP-client and an additional one for the separate communication channel, if a second RTP-stream is used. Otherwise it can also modify RTP-packets to use a special payload or the extension field.
SIP-URLs are cached on first encounter of INVITE and checked

that they remain constant. SDP is also modified whenever it is sent (e.g. as part of response type 180 for ringing) and checked that it doesn't change. Depending on communication method, an additional payload type or a second RTP-channel is added. Before an incoming SDP-body is forwarded to the SIP-client, the additional RTP-stream is removed as well as the `k`=-line used for signaling.
A simple state machine consisting of boolean member variables tracks the state of the call, e.g. if it has been established, if forwarding of packets needs to be forcefully stopped, etc.

As seen in Figure 7.1, `SipCall` is an implementation of `ICallProvider` and therefore it offers the following services and interface to implementations of `ICallClient` like the self signed archive:

`DelayInjectTerminate(bool othersidehungup)` This methods is to be used if the `ICallClient` object returned `false` for `Terminate` to delay the actual termination of the call. In this case the call can continue between the SIP-client and the proxy while the other party already hung up. This way a final statement could be created using speech synthesis or a final classification for an long-term archive could be queried from the user.

`AbortCall()` This can be called if an incoming call should be rejected, i.e. this method aborts calls which are still in the creation phase. For SIP this means to send a SIP-CANCEL-request. This can be used to e.g. reject all incoming calls that offer signature service where the signature can not be verified to be valid.

`TerminateCall()` This terminates a call that has already been established. This is used to e.g. terminate a call with insufficient QoS.

`DelayInjectOutgoingPaket(RTPPacket paket)` and
`DelayInjectIncomigPaket(RTPPacket paket)` These methods are to be used together with `IncomigPacket` and `OutgoingPacket` that are called on an `ICallClient` implementation: If `ICallClient` returns false to these methods, the RTP-packets are not forwarded. Using these two methods here they can later be reinjected. These methods can also be used to inject speech synthesised communication into the ongoing call on any side. This can be used to implemented a dialogue with the user where he confirms signing or enters his PIN.

`SendExtraData(byte[] data)` Sends a datagram over the additional communication channel, e.g. signature data. Usually this is an unreliable

means of transport, i.e. the caller of this methods must implement retransmission.

On the other side, `ICallClient` implementations have to implement the following interface, which notifies them of any highlevel events happening in the call so that they can handle them:

`Create(ICallProvider provider,SIPUri from,SIPUri to` After `SipCall` used a delegate factory to created a new instance of its `ICallClient`, this is the first method that is called. Through it `ICallClient` knows where it can send commands to and can setup From- and To- URLs.

`byte[] Phase1(out bool cancel,out string cancelmessage,SDPParser sdp)` If this side (the SIP-client behind this SIP-proxy) is making a call, this method is called to determine whether initial signaling should be send. This would be returned by this method as a byte array. The call can also be rejected (e.g. if no certificate can be found to be used with this user) and the used codecs can be restricted by modifying SDP.

`Phase2(byte[] data,out bool cancel,out string cancelmessage,SDPParser sdp)` If this side made a call, this method is called after the partner answered. His signaling is provided in `data`. If it contains e.g. an invalid signature, the call can still be canceled at this point by providing `cancelmessage`.

`byte[] Phase1_Answer(byte[] data,out bool cancel, out string cancelmessage, SDPParser sdp)` If the call is incoming, then the highlevel implementation has only one and not two opportunities to react: The incoming call may carry signaling data which is then provided in `data`. It can then reject the call, continue it or even provide its own signaling answer by returning a byte array.

`TearDownEarly()` If the call was not taken or if the caller hung up before the call started, this method is called (to e.g. clean data structures or delete files)

`Setup(...incomingpayloads, ...outgoingpayloads)` This method is called when the call has successfully started and RTP-redirection is established. The highlevel class is informed about the final codec mapping, which can e.g. be stored as metadata along with the archived call.

`DTMFEvent(char c)` This is called whenever a DTMF-event is encountered. This currently does not use digital signal processing to detect special frequencies, but relies on the DTMF-packets being transported using SIP-INFO or a special RTP payload (see RFC 2833 [16]). This can be used to enter PINs.

`ReceiveExtraData(byte[] data)` Whenever the partner sends additional data (e.g. signatures) over the secondary communication channel, this method is called.

`bool Terminate(TerminationType how)` If the call is terminated, this methods is called. It is possible to delay termination by returning `false`. The call can then be continued with the SIP-client to e.g. query the user who this call should be classified for the long-term archive.

`bool OutgoingPacket(RTPPacket paket)` and

`bool IncomigPacket(RTPPacket paket)` These methods are called for every RTP-packet. The packets can then be processed by the `ICallClient`, e.g. collected into intervals. Returning false omits sending of the packets. Instead `DelayInjectIncomigPaket` can be used to inject other or modified packets (like speech dialgoues with the user or announcements about signature status).

7.2.3 Self signed archive classes

`ChunkedFile` This class extends normal `FileStream` based IO-streams to support a chunked fileformat, so that bytearrays can be stored as entities and read back in the same form. A normal stream would not care whether its Write-method was called once with all the data or multiple times with pieces. This class is necessary because the whole archive file format is based on the concepts of intervals. This class also stores a flag for each chunk whether this is the last chunk. Together with the regular EOF-property of a stream this enables detection whether the call was properly ended or interrupted.

`Signing` This class is a wrapper around PKCS#7 signed envelope signatures and the process of signing data. It can keep track of additional certificates that only need to be stored with the first interval. Furthermore it supports the scenario of smartcard readers where the user has to enter his PIN-number for every operation (see section 6.1). For this case the class has the option to create an extra RSA keypair for each file. Then only the first interval is signed using PKCS#7 and contains also the public key. All further intervals are signed using this individual private key. After the file is closed, the private key is wiped out.

`IntervalStatisticData` This class is used to pass statistical data about a complete interval to the statistic module, which can subscribe to an event to get these from the `SelfSignedArchiveCallEventsListener`. This data comprises date/time, packet count, packet loss and interval number

`MetaDataBlock` This class is used to encode and store all metadata for the first interval of the call as seen in Figure 4.5. This comprises URLs for caller and callee, the complete payload mapping,

date and time of the call and a nonce-value. The latter was used in [43] to mitigate a specific attack where the attacker submits a call twice to the archive to weaken its evidentiary value.

`MoreChecks` The QoS checks mentioned in section 4.9.1 are implemented in this class, e.g. check for strict monotonic increase of sequence numbers or the time difference between system time and time calculated from the timestamp.

`Sequencer` This class reconstructs absolute sequence numbers, timestamps and implements the replay window. This needs to be done because RTP stores sequence numbers as 16 Bit values starting with a random number (see section 3.3.2) which can easily overflow multiple times during one call. Also further checks need a real time in milliseconds instead of the sampling-rate specific RTP-timestamp, which can also wrap around and usually starts with a random value. In order to disambiguate these values, packets must be passed in the same order to this class as they are received over the network, i.e. this class is sensitive to the order in which the packets are passed. With the first processed packet the class is initialised. This first packet forms the basis of all relative values like absolute packet number and timespan into the call. RTP timestamp and RTP sequence number are stored as offset values in order to remove the randomness of the initial values. Wraparounds are addressed with a rollover counter. Every new packet is tried first for the current rollover counter, then for the rollover counter+1 and for the rollover counter-1. The sequence number of these three that passes the replay window is accepted. In case of rollover counter+1, the rollover counter itself is incremented by one. For timestamps no replay detection is performed, but the most likely value is used, too: Three different timespans are calculated with three different values for the rollover counter. The one which has the lowest difference to the last packet is taken. For gaps larger than 10h this is rejected, because silence suppression for such a long time disallows disambiguation.

The replay window itself is realized with the bits of a long integer: Each bit stands for one packet sequence number, 1 means that this sequence number was already seen. Bit 0 corespondents to absolute packet sequence number `m_absseq_windowsstart`. If this variable is incremented, the windows is bitshifted to the left accordingly thus creating a “sliding” window. Per definition, all packets with sequence numbers lower than `m_`–

`absseq_windowsstart` are too late respectively have too high jitter to be considered. Packets inside the range of the sliding window are examined in detail whether their bit is set and they are duplicates. And packets in the other side of the range of the sliding window will shift the window in their direction until their corresponding bit is the highest one. This provides an (memory-) efficient way to detect duplicate packets which would otherwise allow an attacker to insert audio to be signed that nobody would normally hear.

`RTPPaketPlusSeq` This class is used to pass along RTP-packets together with additional data like receive time, reconstructed absolute sequence number and absolute timestamp.

`SelfSignedArchiveCallEventsListener` This class is the main class for the self signed archive scenario. It implements `ICallClient` and therefore is instantiated for every call. From the proxy infrastructure it receives preprocessed highlevel notifications about the call. In its `Setup` method which is called after the successful start of a call, it opens a new file, creates the data structure for metadata and encodes and signs it into the first chunk. To do so it uses `Signing`, `ChunkedFile` and `MetaDataBlock`.

After that it creates two instances of `MoreChecks`, `Sequencer` and `PacketCollector` for each channel, which are called to process every incoming or outgoing RTP-packet. After everything is ready and the first interval is signed and written to disc, a new thread is started. This thread will wait for the configured interval length and then do the following for each channel: The collected packets are retrieved from `PacketCollector` by calling its `Swap` method. They are already sorted. They are then checked for monotony, packet loss and time drift. If they are okay, the interval data is formed and encoded, covering also the hash of the preceding interval. This encoded interval data is then signed using the `Signing` instance and written to disc.

7.2.4 Main program

`Certificates` This class implements an user control to choose a certificate from the Windows Certificate Store, for which the private key is present. It enumerates the stored certificates and shows them in a list box.

ProxyForm This class contains the complete user interface seen in Figure 6.1(a). It creates a `SipProxies` object and adds the four Sip-Proxies as in Table 6.1. It subscribes to the events that are provided for GUI-updates. Therefore there is no direct access of the GUI to the internals of the proxy, all GUI-updates happen through notifications.

7.3 Statistics module

Blocks This class implements an user control that visualises up to two `BitArray`-classes as block-diagram. It also reacts to window-resize messages accordingly and displays the packet number as tooltip if the mouse is hovered above one block. To determine how many blocks per row are displayed, the aspect ratio of the client area of this control is measured and rounded. The last row is not completely filled

PaketStatistics This class implements the complete statistics module for packets. It exposes methods for reset and adding of packet-statistics by passing instances of `RTPPaketPlusSeq`, which contain all needed data like reconstructed time.
It uses ZedGraph [42] and `Blocks` to draw diagrams. These are created by determining the duration of all stored packets and dividing it by the amount of pieces, which the user can configure. The packets are then binned into these segments and packet loss and number of packets are counted. Optionally this data is processed with the Fourier-transfomation. Routines to export and import statistical data as CSV are also implemented here.

IntervalStatistics This class is simpler than `PaketStatistics` because it does not need to bin packets into linear segments. Instead it simply uses the data that is supplied with the `IntervalStatisticData` class for each interval and plots a bar for every interval. Routines for CSV export and import of statistical data for intervals are also implemented here.

7.4 Call Verifier

In principle, the call verifier can verify very long phone calls without increased memory usage. The complete design of the tool processes the files interval by interval and reassembles and mixes audio using streams. Also

playback and export are using streams and do not store the whole conversation in memory at once. This is not trivial to implement as new data for both channels arrives in bursts: Whenever an interval is decoded, a large amount of packets are released to be appended to either channels. Figure 7.2 shows

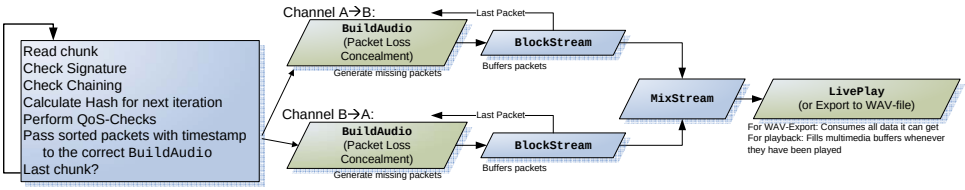


Fig. 7.2: Architecture and flow chart of voice processing

how the main decoder loop yields packets that are then processed for packet loss concealment and reconstruction of the channel stream. Depending on the chosen MixMode, they are then mixed and passed to `LivePlay` or copied to a WAV-file using the `DeferedAudioOut`-class.

`WaveLib` This namespace contains classes and methods to play PCM audio. The .NET framework only has methods to play sounds that were loaded completely into memory. Here interop-techniques are used to access the native Windows multimedia APIs. A second thread is used to wait for Windows to notify the application that a buffer was played and that a new buffer must be provided.

`DeferedAudioOut` This class is able to encode 16 Bit PCM-audio data to WAV-files. In contrast to the `AudioChunk` class, which processes small pieces of audio like synthesised speech, this class can be used to stream audio data to a WAV-file. When constructed, it writes a dummy WAV-header to disk which contains the wrong length-fields. During writing, it counts the amount of samples. When the file is finally closed, the existing header is patched to produce a correct WAV-file.

`LivePlay` This class fulfills the events that `WaveLib` raises when new audio data is requested. It is an interface between generic streams and the audio interface.

`BlockStream` This class is a simple stream that merges blocks into an continuous byte stream. It contains a linked list/queue of byte arrays and will satisfy Read-requests of arbitrary length by taking the number of blocks needed from the head of the queue and keeping track of which portion of the current block has already been read. An method `AddBlock` allows to add byte-arrays.

The most recently added block can be accessed with the public property `LastBlock` to support the repetition packet loss concealment technique.

`MixStream` This class is able to mix two streams with PCM-encoded 16 Bit mono audio data into one output stream. The constructor supports two modes: Stereomix and normal mixing. Normal mixing simply adds two samples, one from each stream and divides them by two. This is the classical method to mix two audio channels into one. The other mode produces a stereo PCM-signal and uses stream A for the left channel and stream B for the right channel. To do so, samples from A and B have to be used in turn.
If the `Read` method is called while either of the input streams for channel A or B are not able to produce data, then `MixStream` will buffer whatever it got from A. It will also return to its caller the amount of data where it was able to get data from both streams.

`BuildAudio` This class converts RTP-packets to PCM-data. It knows about the payload mapping and calls the appropriate codec. It demands that packets are handed to it in the correct order. If it detects gaps according to the timestamp-values, it will perform one of four methods of packet loss concealment as stated in section 6.6 and produce filling audio data for the gaps. The output of this class is a stream. It uses the `G711`-class to decode the payload and the `BlockStream` class to create the actual stream.

`ICheckItemList` This interface separates GUI and implementation. The process of decoding and checking signed calls produces not only packets and a dictionary of metadata, but can also constantly inform its caller about progress and results of its steps. This happens through this class which abstracts the concept of a checklist of steps as seen in the main windows of the Verifier, e.g. in Figure 6.1(b).
List items/steps are identified by an internal keyword and have a clear text for the user. One item can always be selected to be the active one. Every item contains one of five states: Error, Warning or Success (displayed with corresponding icons), None (if not processed yet) or Insignificant (if that item is not anymore worthy to be shown to the user, e.g. opening the file. If an error occurs in the beginning, such an item is interesting, but later it can be removed from the list).
Every step can also be associated with success and failure counters which are then displayed in parentheses.

CheckItem This class represents a single item in a check item list.

CheckListe This class is the only actual implementation of `ICheckItemList` and displays its contents in a window with nice icons. The current workitem is visualized with a green playback icon.

Checker This class provides the lowest layer for processing and checking of archived calls. It processes the first chunk with metadata and decodes the interval data for each subsequent chunk. It checks the timestamp, all signatures and certificate chains and the hash-chain, but leaves actual processing of the interval contents to classes which inherit from it. It keeps track of the `ChunkedFile` instance for the open file and the last hash and checks this with each call to `ReadNextBlock`. This method returns an `BinaryReader` for every interval which can be used to decode the actual interval data.

This class also maintains a `StringDictionary` which contains all the results and data shown in Figure 6.6(b). Most values like timestamp-time, start time, correctly closing of file, number of chunks, etc. are measured by this class.

Reader This class inherits `Checker` and decodes the interval data from its binary format. This contains the metadata for the interval, the absolute sequence number list and the actual packets. It also measures signature- and RTP-overhead. Interval data is passed to the virtual `ProcessInterval` method.

Reader2 This class inherits `Reader` and implements the virtual `ProcessInterval` method. It reconstructs absolute sequence numbers and timestamps from the RTP-packet using the Sequencer-class, thus emulating the behavior of the proxy. These values are then tested using `MoreChecks` so that a non conforming proxy can be detected. In detail it checks the strict monotony of the absolute sequence numbers, the strict monotony of timestamps, time drift of the reconstructed timestamps, packet loss and correctness of the stored interval time.

This class also passes packets along to `BuildAudio` and `MixStream` to provide a continuous audio stream of the whole conversation, if requested

CheckDialog This class is the main window. It changes the program configuration and calls `Reader2` to actually process the file. It also selects if the generated audio should not be processed at all, written to the soundcard or to a WAV-file.

8

Outlook

During this work, a vast field of future research- and implementation ideas related to the presented concepts arose which could not be implemented or discussed in the available time and space:

- In scenario 2 the signer expresses his explicit will to have the call recorded by providing signatures. But in for the implemented archive in scenario 2 there are legal requirements that all involved parties have to agree to the recording which would otherwise be void. A real-world implementation needs to consider conditions for, and signaling and negotiation of recording of a conversation. The draft standard [44] describes a method by which "One party may assert either their desire to record or their restriction of the other party's recording". Using these assertions in the archiving architecture in the sense that the proxy evaluates and respects them would be a nice way to disarm privacy reservations to indiscriminate recording of calls.
- The signaling of the archiving status and reasons for termination of the archiving or the call should be improved. Using speech synthesis would be a device independent way which works with every kind of phone. This could also be used to announce recording of the call to both parties.
  To do this, the synthesized speech must be encoded using the correct codec and mixed in or replace the RTP-stream, ideally with correct sequence-numbers and timestamps.
- Based on scenario 2, an extended protocol could be created that supports signatures by both parties or even for conference calls. In this scenario, every partner needs to poss a certificate and archive the call. As it is favorable that both parties have a bit-identical signed recording of the call, the signatures creation of A and B must be synchronized.
  This slight variation provides mutual non-repudiation.
- Instead of having a monolithic program (the implemented proxy) which records, secures and archives calls, a more modular system can be created

resulting in a even higher security against compromised components as shown in figure 8.1.
Here the functionality is split in the two components VSec and Arc, ac-

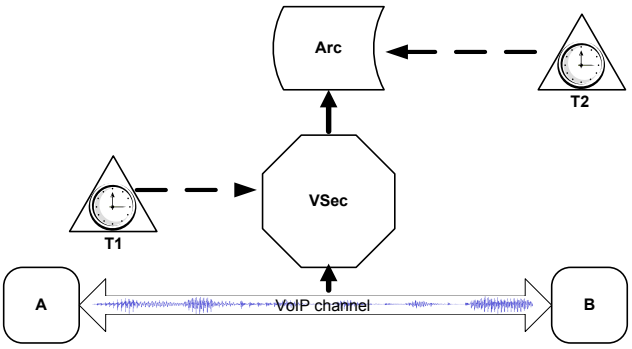


Fig. 8.1: System-model for a distributed secure archive solution

companied by the two trusted time-sources or timestamp authorities T1 and T2. VSec listens to the communication, secures it with its private key and requests the initial timestamp. Arc is the long-term archive which e.g. applies regular timestamps to archived calls like in [45, 46].
The secured call is streamed by VSec to Arc, so that VSec needs a constant amount of memory not related to the length of the call. While Arc receives the call, it continuously checks the signatures, quality of service and timestamps of the call. Together with checking the trusted timestamp of T1, an attacker who compromised the component VSec would have only a very short amount of time of about 1 second to forge the archived call.
This was presented together with an exhaustive security analysis in [43].

- Direct implementation of the signing solution inside an existing SIP-client implemenation would lead to slightly increased security, because replay-windows and jitter-buffer need not be modeled in the proxy anymore, but the real list of packets that made it into the playout-buffer would be the basis for what is signed.
  The disadvantage of this would of course be that this would not be flexible regarding used SIP-clients anymore.
- The signing-solution should be implemented on mobile phones. The concepts lends itself very well to this because of the tunable, bounded requirements on CPU-power and memory for buffers. But the implementation could probably suffer from languages like Java and C# which utilise garbage collectors. If this is really the case on todays devices remains to

be seen. In any case a mobile device is unlikely to provide enough long-term storage space for B, so the archived call needs to be transmitted to a storage-service, ideally under control of B. A third variation mixing both concepts to avoid transmission of the archived call from B to an archive can be envisioned.

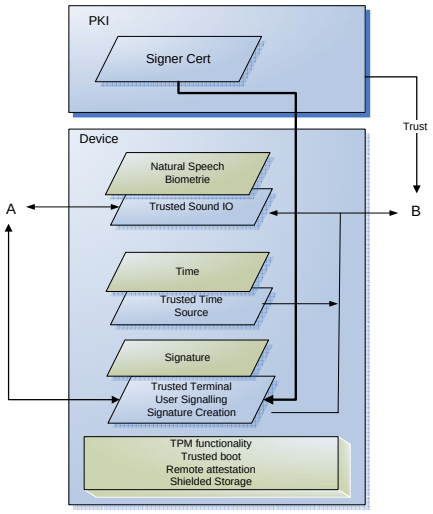


Fig. 8.2: Block diagram of a highly secure signature terminal based on trusted computing, see [1]

- The presented technology could be used to fulfill the high security requirements for signature terminals. Though difficult in general, the this task is simplified since only one channel (audio) carries the signed information. Methods of trusted computing could be employed to turn a VoIP capable handset into a trustworthy signature platform. Figure 8.2 shows a schematic presentation of the envisaged Trusted Voice Signature Terminal.
- The self-signed archive scenario could be the basis for a successful voice-archiving product in form of an appliance. It currently lacks ways of finding archived calls like a database. Also load-tests have to be carried out and error handling must be improved. The use of the Linux IPTables-firewall would allow for building a transparent proxy for calls. This would result in a plug'n'play device that can record calls without changing the configuration of every single SIP-client.
- Additional codecs, especially iLBC and Speex must be integrated into the proxy and the verifier for further experiments regarding packet loss.
- Video conferencing over SIP could be tested with the presented concepts. A litte extension to store, sign and archive multiple RTP-streams instead of only one could be included.

- The used card reader contains a keypad to enter the PIN-number. Methods should be found to supply the PIN through the proxy software, so that the DTMF dialpad of a remote phone could be used instead of the dialpad of the cardreader.

A

Glossary

ASN.1 Abstract Syntax Notation One, a joint ISO and ITU-T standard and flexible notation that describes data structures for representing, encoding, transmitting, and decoding data.

a-law A standard companding algorithm, used in European digital communications systems to optimize, i.e., modify, the dynamic range of an analog signal for digitizing.

Codec Abbreviation for Coder-Decoder or compressor/decompressor, software or a device that encodes or decodes a signal.

DNS Domain name system, a system that stores and associates information with domain names, most importantly the IP-address of the host. It can also store mail exchange servers for each domain and SIP-proxies for handling SIP-calls to this domain.

DOS-attack Denial-of-service attack, an attack that makes a computer resource unavailable to its intended users.

DTMF Dual-tone multifrequency, is used for telephone signaling over the line in the voice frequency band to the call switching center. In VoIP-scenarios, an emulation of the telephone keypad with the keys `0-9#*` can be provided without frequency modulation.

G.711 An ITU-T standard for audio companding.

GUI

HMAC Keyed-hash message authentication code, used to verify both the data integrity and the authenticity of a message with a shared secret key.

HTTP Hypertext Transfer Protocol, a protocol used to transfer or convey information on the World Wide Web.

IAX Inter-Asterisk eXchange protocol, used by Asterisk, an open source PBX server from Digium.

IETF Internet Engineering Task Force, develops and promotes Internet standards.

iLBC Internet Low Bit Rate Codec, a royalty free narrowband speech codec, developed by Global IP Sound.

IPSec IP security, is a standard for securing Internet Protocol (IP) communications by encrypting and/or authenticating all IP packets.

ISP Internet service provider, a business or organization that offers users access to the Internet.

ITU International Telecommunication Union, an international organization established to standardize and regulate international radio and telecommunications.

IV Initialization vector, a block of bits to allow a cipher to produce a unique stream independent from other streams produced by the same encryption key.

MIME Multipurpose Internet Mail Extensions, an internet standard for the format of mail.

MOS Mean Opinion Score, provides a numerical indication of the perceived quality of received media after compression and/or transmission. The MOS is expressed as a single number in the range 1 to 5, where 1 is lowest perceived quality, and 5 is the highest perceived quality.

μ-law A standard analog signal compression algorithm used in digital communications systems of the North American and Japanese digital hierarchies to optimize, i.e., modify, the dynamic range of an analog signal for digitizing.

NAT Network address translation, the process of network address translation.

PCM Pulse-code modulation, a digital representation of an analog signal where the magnitude of the signal is sampled regularly at uniform intervals, then quantized to a series of symbols in a binary code.

PKCS#7 Public Key Cryptography Standards #7, used to sign and/or encrypt messages under a PKI.

PKI Public key infrastructure, an arrangement that provides for trusted third party vetting of, and vouching for, user identities.

QOS Quality of service, the probability of the telecommunication network meeting a given traffic contract. Here mainly concerned with jitter, delay and especially packet loss.

RTP Real-time Transport Protocol, defines a standardized packet format for delivering audio and video over the Internet.

SBC Session Border Controller, a device used in some VoIP networks to exert control over the signaling and media streams involved in setting up, conducting, and tearing down calls.

SDP Session Description Protocol, used to describe RTP-sessions including codecs, parameters and port-numbers.

SIP Session Initiation Protocoll, an IETF standard providing signaling for VoIP technology.

SMIME Secure MIME, a standard for public key encryption and signing of e-mail encapsulated in MIME.

SOAP Simple Object Access Protocol, a protocol for exchanging XML-based messages over a computer network, normally using HTTP. SOAP is the foundation layer for Web services.

Speex A free software speech codec that claims to be unencumbered by patent restrictions, licensed under the BSD License.

SPIM Messaging SPAM, a type of SPAM where the target is instant messaging services.

SPIT Spam over Internet Telephony, an as-yet-nonexistent problem.

SRTP Secure RTP, a variant of RTP that allows replay detection, symmetric encryption and HMAC-based integrity protection and authentification.

STUN Simple Traversal of UDP over NATs, a network protocol allowing clients behind NAT to find out its public address and other information about the NAT.

TLS Transport Layer Security, a successor to Secure Socket Layer (SSL), providing endpoint authentication and communications privacy over the Internet using cryptography.

URL Uniform Resource Locator, is a string which refers to a resource on the Internet by its location.

VoIP Voice over IP, is the routing of voice conversations over the Internet or through any other IP-based network.

VPN Virtual private network, *often* used to provide confidentiality, integrity and authenticy for network traffic. E.g. based on IPSec.

WAV Waveform audio format, a Microsoft and IBM audio file format standard for storing audio on PCs.

B

List of Figures

C

List of Tables

D

References

[1] Christian Hett, Nicolai Kuntze, and Andreas U. Schmidt. Security and Non-Repudiation for Voice-Over-IP Conversations. Poster presentation at the ISSA 2006 From Insight to Foresight Conference, Sandton, South Africa, 5th-7th July 2006.

[2] J. Kavanagh. Voice over IP special report: From dial to click. http://www.computerweekly.com/Articles/2006/02/14/214129/VoiceoverIPspecialreportFromdialtoclick.htm. [Online; visited 1.3.2006].

[3] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman. The Secure Real-time Transport Protocol (SRTP). RFC 3711 (Proposed Standard), March 2004.

[4] Roberto Barbieri, Danilo Bruschi, and Emilia Rosti. Voice over IPsec: Analysis and Solutions. In *ACSAC*, pages 261–270. IEEE Computer Society, 2002.

[5] Norman Poh and Samy Bengio. Noise-Robust Multi-Stream Fusion for Text-Independent Speaker Authentication. The Speaker and Language Recognition Workshop (Odyssey), 2004.

[6] L. Rodriguez-Linares and C. Garcia-Mateo. Application of fusion techniques to speaker authentication over ip networks. *IEE Proceedings-Vision Image and Signal Processing*, 150(6):377–382, December 2003.

[7] H. Hollien. *Forensic Voice Identification*. Academic Press, London, 2001.

[8] C. Goodwin. *Conversational Organization: Interaction Between Speakers and Hearers*. Academic Press, New York, 1981.

[9] Athina Markopoulou, Fouad A. Tobagi, and Mansour J. Karam. Assessment of VoIP quality over Internet Backbones. In *Proceedings of the 21st Annual Joint Conference of the IEEE Computer and Communications Society (INFOCOM-02)*, volume 1 of *Proceedings IEEE INFOCOM 2002*, pages 150–159, Piscataway, NJ, USA, June 23–27 2002.

IEEE Computer Society.

[10] Christian Hoene, Holger Karl, and Adam Wolisz. A Perceptual Quality Model for Adaptive VoIP Applications. In *Proceedings of SPECTS'04*, San Jose, CA, May 24 2004.

[11] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler. SIP: Session Initiation Protocol. RFC 3261 (Proposed Standard), June 2002. Updated by RFCs 3265, 3853, 4320.

[12] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: A Transport Protocol for Real-Time Applications. RFC 3550 (Standard), July 2003.

[13] Perrig, Canetti, Tygar, and Song. Efficient Authentication and Signing of Multicast Streams over Lossy Channels. In *RSP: 21th IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[14] Wojciech Mazurczyk and Zbigniew Kotulski. Alternative security architecture for IP Telephony based on digital watermarking, June 18 2005.

[15] Wojciech Mazurczyk and Zbigniew Kotulski. New security and control protocol for VoIP based on steganography and digital watermarking, February 10 2006.

[16] H. Schulzrinne and S. Petrack. RTP Payload for DTMF Digits, Telephony Tones and Telephony Signals. RFC 2833 (Proposed Standard), May 2000.

[17] M. Handley and V. Jacobson. SDP: Session Description Protocol. RFC 2327 (Proposed Standard), April 1998. Updated by RFC 3266.

[18] A. B. Roach. Session Initiation Protocol (SIP)-Specific Event Notification. RFC 3265 (Proposed Standard), June 2002.

[19] B. Campbell, J. Rosenberg, H. Schulzrinne, C. Huitema, and D. Gurle. Session Initiation Protocol (SIP) Extension for Instant Messaging. RFC 3428 (Proposed Standard), December 2002.

[20] S. Donovan. The SIP INFO Method. RFC 2976 (Proposed Standard), October 2000.

[21] Audio-Video Transport Working Group and H. Schulzrinne. RTP Profile for Audio and Video Conferences with Minimal Control. RFC 1890

(Proposed Standard), January 1996. Obsoleted by RFC 3551.

[22] ITU-T. Recommendation G.114 - One-way transmission time. *Geneva, Switzerland*, February 1996.

[23] Dan Wing Flemming Andreasen, Mark Baugher. Session Description Protocol Security Descriptions for Media Streams. http://www.ietf.org/internet-drafts/draft-ietf-mmusic-sdescriptions-12.txt, September 2005.

[24] J. Arkko, E. Carrara, F. Lindholm, M. Naslund, and K. Norrman. MIKEY: Multimedia Internet KEYing. RFC 3830 (Proposed Standard), August 2004.

[25] Wikipedia. μ-law-Verfahren – Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=%CE%9C-law-Verfahren&oldid=17867559, 2006. [Online; accessed 11-July-2006].

[26] Wikipedia. A-law-Verfahren – Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=A-law-Verfahren&oldid=14786406, 2006. [Online; accessed 11-July-2006].

[27] Hans Weibel. Kommunikationstechnik 2. http://www-t.zhwin.ch/ki/kt2/06-VoiceoverIP/KT2-06-S-V02-VoIP.pdf.

[28] S. Andersen, A. Duric, H. Astrom, R. Hagen, W. Kleijn, and J. Linden. Internet Low Bit Rate Codec (iLBC). RFC 3951 (Experimental), December 2004.

[29] Global IP Sound. iLBC Freeware. http://www.ilbcfreeware.org/.

[30] J. Rosenberg, J. Weinberger, C. Huitema, and R. Mahy. STUN - Simple Traversal of User Datagram Protocol (UDP) Through Network Address Translators (NATs). RFC 3489 (Proposed Standard), March 2003.

[31] Wikipedia. UDP hole punching – Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=UDP_hole_punching&oldid=63610246, 2006. [Online; accessed 15-July-2006].

[32] Newport Networks Limited. QoS and Bandwidth Management. http://www.newport-networks.com/product/qos.html. [Online; accessed 15-July-2006].

[33] Peter Landrock and Torben Pedersen. WYSIWYS? - What you see is what you sign? *Information Security Technical Report*, 3 (1998) 55–61.

[34] Dimitrios Lekkas and Dimitris Gritzalis. Cumulative notarization for long-term preservation of digital signatures. *Computers & Security*, 23(5):413–424, 2004.

[35] J. P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, USAF Electronic Systems Division, Hanscom Air Force Base, Bedford, Massachusetts, October 1972.

[36] B. Kaliski. PKCS #7: Cryptographic Message Syntax Version 1.5. RFC 2315 (Informational), March 1998.

[37] R. Housley, W. Polk, W. Ford, and D. Solo. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 3280 (Proposed Standard), April 2002. Updated by RFC 4325.

[38] Microsoft .NET Framework. http://msdn.microsoft.com/netframework/. [Online; visited 1.7.2006].

[39] The Mono Project. http://www.mono-project.com/. [Online; visited 1.7.2006].

[40] TUD chip card "TUDCard" project page. http://www.tu-darmstadt.de/hrz/chipkarte/. [Online; visited 1.7.2006].

[41] KOBIL cardreader KAAN Advanced. http://www.kobil.de/d/products/smartcard/kaan-advanced.php. [Online; visited 1.7.2006].

[42] ZedGraph, a graph library for .NET. http://www.zedgraph.org. [Online; visited 20.7.2006].

[43] Christian Hett, Nicolai Kuntze, and Andreas U. Schmidt. A secure archive for Voice-over-IP conversations. In Dorgham Sisalem et al., editor, *Proceedings of the 3rd Annual VoIP Security Workshop (VSW06)*. ACM, June 2006.

[44] R. Shacham, H. Schulzrinne, W. Kellerer, and S. Thakolsri. Use of the SIP Preconditions Framework for Media Privacy. RFC 3261 (Proposed Standard), June 2006. Expires: December 27, 2006.

[45] U. Viebeg, T. Kunz, and S. Okunick. Warum elektronische Signaturen altern und brechen - und wie man ihren Wert erhält. *IT-Sicherheit & Datenschutz*, June 2006.

[46] Archisoft project. Fraunhofer-Institute for Secure Information Technology SIT. http://www.sit.fraunhofer.de/cms/de/referenzprojekte_menu/ArchiSoft_1.php.

[47] Wikipedia. A-law-Verfahren – Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=A-law-Verfahren&oldid=14786406, 2006. [Online; accessed 11-July-2006].

[48] Wikipedia. μ-law-Verfahren – Wikipedia, Die freie Enzyklopädie. http://de.wikipedia.org/w/index.php?title=%CE%9C-law-Verfahren&oldid=17867559, 2006. [Online; accessed 11-July-2006].

Erklärung zur Diplomarbeit
gemäß § 19 Abs. 6 DPO/AT

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den _____

_____

Unterschrift