Technische Universität München

Department of Informatics

Master's Thesis in Informatics

HRTP: A Broadcast-Based System for Unobservable Internet Telephony

Daniel Hugenroth

Technische Universität München

Department of Informatics

Master's Thesis in Informatics

HRTP: A Broadcast-Based System for Unobservable Internet Telephony

HRTP: Ein System für unbeobachtbare Internet-Telefonie mittels Broadcast

Author	Daniel Hugenroth
Supervisor	Prof. Dr.-Ing. Georg Carle
Advisor	Lukas Schwaighofer
Date	15th August 2015

Informatik VIII
Chair for Network Architectures and Services

I confirm that this master's thesis is my own work and I have documented all sources and material used.

Garching b. München, 14th August 2015

Signature

Abstract

Internet telephony and Voice-over-IP (VoIP) in general are popular and ubiquitous means of communication. In the last years research has led to a high standard of securing VoIP conversations in terms of confidentiality, authenticity and integrity. However, privacy is generally left unaddressed despite the fact that the calls' meta-data themselves offer valuable information to an adversary. This is increasingly important as Internet applications are vulnerable to comprehensive surveillance and active attacks.

State-of-the-art anonymity systems, such as Tor, neglect the attack vector of a global active attacker (GAA) and are vulnerable to traffic confirmation attacks. Furthermore, their high transport delay renders them unsuitable for low-latency applications like VoIP.

This thesis presents the design and implementation of a prototype named Hidden Real-Time Protocol (HRTP) for unobservable Internet telephony. It uses continuous and encrypted pair-wise traffic of fixed rate and packet size between all peers. We show that it both provides complete unobservability against GAA and offers an end-to-end latency that is suitable for VoIP. A real-world setup and its evaluation demonstrate that it is practically feasible to deploy such a system.

Obvious areas of applications for HRTP are in the environment of embassies, non-governmental organizations (NGOs) and other settings of highly sensitive communication.

Finally, this thesis develops a variant of HRTP that scales more efficiently. To this end, a combination of small broadcast groups and onion-routing relaying in between is used. The approach still maintains strong unobservability against GAA, but reduces the per peer traffic to $O(\frac{n}{log^2 n})$ as opposed to a previous growth of $O(n)$.

Contents

Outline of the Thesis

Chapter 1: Introduction & Motivation

The first chapter introduces the central terms of privacy, anonymity and unobservability. It then provides motivation for unobservable Internet telephony by putting them into context of global Internet surveillance.

Chapter 2: Fundamentals

This chapter introduces the reader to the fundamentals of IT security and related work in the field of anonymous communication systems. Moreover, an exemplary voice-over-IP pipeline is explained and existing solutions in the field of anonymous Internet telephony are discussed.

Chapter 3: Approach & Design

The approach chapter provides an abstract view on the proposed idea. It defines a notation and describes the protocol on a theoretical level. The chapter concludes by showing that the designed solution is indeed unobservable under given assumptions.

Chapter 4: Implementation

This chapter describes how the solution has been put into practice on a detailed technical level. Firstly, it discusses the HRTP protocol covering packet formats and state transitions. Secondly, the implementations of the gateway and the Android application are presented.

Chapter 5: Evaluation

The evaluation demonstrates that HRTP holds its promises in a real-world setup. Measurements of latency quantities including the end-to-end audio delay show that it is suitable for voice conversation.

Chapter 6: Quo vadis: Transformation into a Scaling System

In the sixth chapter, it is proposed how the current implementation can be extended in order to provide better scaling properties.

Chapter 7: Further Work & Conclusion:

This chapter points out areas for further work based on this thesis and related work. A short conclusion summarizes the central findings of this work.

Appendix:

The appendix contains information regarding notational conventions, the evaluation setup, a log file of a gateway test and screenshots of the Android application. Furthermore, it gives a comprehensive list of used tools and literature. Finally, it provides the DVD containing the implementation including instructions for usage.

CHAPTER 1

Introduction & Motivation

"Big Brother is watching you."

— 1984 by George Orwell

Privacy and anonymity are two fundamental concepts. With modern, electronic communication, both their importance and feasibility have reached new levels. We will define the basic concepts of privacy, anonymity, unobservability and will motivate them with regard to the Internet and global surveillance.

Privacy, Metadata, Anonymity and Unobservability

Privacy is a fundamental human right as stated in the Universal Declaration of Human Rights adopted by the United Nations (UN) General Assembly: "No one shall be subjected to arbitrary interference with his privacy, family, home or correspondence, nor to attacks upon his honour and reputation. Everyone has the right to the protection of the law against such interference or attacks." [1, article 12]. In the context of communication, privacy states how much a participant is exposed while communicating with his peer or a third party.

When communicating, not only the content (the message) is exchanged, but also additional information is generated. Examples for such information are the time the message was exchanged, the sender, the recipient , the used medium and the size of the message. We call this information *metadata*. And while confidentiality of the content can be easily achieved using encryption, hiding metadata is much more of a challenge.

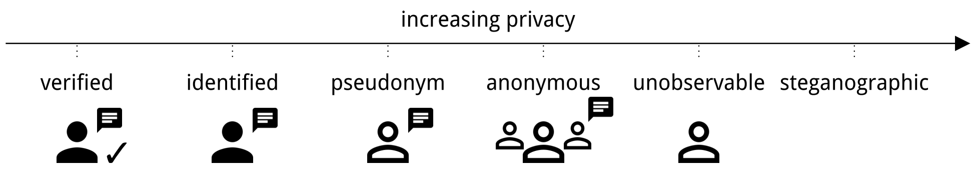


Figure 1.1: Our scale of communication privacy guarantees ordered from least private (publicly verifiable identity) to most private (steganographic communication) from an observer's point of view. Own graphic.

In most conversations we provide parts of our *identity.* This allows the other party to match the communication to any other information they have about us. It might also go a step further and demand to *verify* our identity by asking for something only the two participants know (e.g. a password) or alternative means of certification (e.g. a passport).

It is not always necessary to reveal our real identity and often it is not advisable. For this purposes we "identify" ourselves using a *pseudonym* (e.g. a nick name) that is not directly linked to our real identity. The other side can still link all information belonging to a pseudonym, but not match multiple pseudonyms used by a single entity.

Anonymity has been defined as "*state of being not identifiable within a set of subjects, the anonymity set*" [2, p.2]. When using strong anonymous communication, no other party can determine who has sent a message (sender anonymity) or received a message (recipient anonymity). We find anonymity e.g. in the secret ballot in elections.

In an *unobservable* communication system, only the participation of a member can be seen, but it is not known whether it is currently communicating or not. In a *completely unobservable* system, it cannot be determined if any communication is happening at all. The system that we have built provides this strong property.

For hiding the mere participation in a system, *stenographic* techniques can be used. Here the protected information is hidden within the entropy of another legitimate communication channel. One example is hiding a short text message in the noise of a picture [3]. The high overhead makes it difficult to apply this approach to voice-over-IP communication.

The Ubiquitous Internet and its Implications

The ubiquity of the Internet neither changes nor redefines the aforementioned concepts of privacy and anonymity. But it interacts with them in a novel and unprecedented manner. First, we have to make substantial privacy decisions more often. The individual using a service faces the challenging task to understand and evaluate the repercussions on private data. Privacy is often willingly traded for free access and comfort, but it remains unclear how many of these free decisions are actually well-founded decisions.

An extreme, fatalistic position is the total denial of effective privacy at all. A famous example is the quote of Scott McNealy, the former chief executive officer of Sun Microsystems, who said in 1999: ”You have zero privacy anyway. [...] Get over it.“ [4]

Secondly, the Internet both reduces everyday anonymity and provides means for anonymity that have not existed before. This might appear contradictory in the first place. On the one hand, the extensive nature of the Internet with its core connection hubs allows parties to observe a huge amount of everyday actions at a single point. On the other side, new technical solutions (such as the later introduced MIX networks) offer new levels of anonymity that have not been achievable ever before.

Motivation for Unobservable Internet Telephony

The recent years have unveiled that large-scale surveillance of Internet traffic is no fictive worst-case scenario but status-quo. In particular, the existence and feasibility of global adversaries has to be considered. This challenges the designs of popular anonymous communication systems such as Tor [5]. And while the content is usually inaccessible due to encryption, the metadata itself often provides sufficient insight. The implications of metadata being exploitable are illustrated by the following quote of Gen. Michael Hayden (former head of the National Security Agency) from 2014: “We kill people based on metadata [. . .]”. [6]

In spite of emerging asynchronous communication channels such as instant messaging, Internet telephony remains an important and fast-growing medium. The international traffic of Skype, a widely-known Internet telephony software, “[. . .] grew 35 billion minutes in 2014, to 248 billion minutes [. . .] and Skype’s 2014 traffic growth was nearly 30 percent greater than the volume growth of every carrier in the world, combined.” [7, p.8]

However, voice-over-IP brings new challenges to existing anonymity systems by requiring very low end-to-end latency. We will see that traffic patterns of Internet telephony make attacks against anonymity easier. As current solutions fail to provide reasonable anonymity facing a global adversary, we design, implement and evaluate a new solution for unobservable Internet telephony.

Our solution fits to scenarios with a few dozens of stationary communication partners which can legitimately deploy strongly protected communication channels. Prime use cases can be found in the environment of diplomatic embassies and non-governmental organizations (NGOs).

CHAPTER 2

Fundamentals

"The only truly secure system is one that is powered off, cast in a block of concrete and sealed in a lead-lined room with armed guards."
— Gene Spafford

For meaningful security analysis of IT systems, attacker models are necessary. They are introduced here along with basic IT security terminology. An overview on anonymous communication systems shows that related work lacks strong security guarantees against global active attackers or are unsuitable for VoIP. The technical discussion of VoIP and its components unveils very specific and interesting challenges for this field of Internet Protocol (IP) application. Reviewing available and deployed solutions shows that they provide authenticity and integrity guarantees in an exemplary manner, but lack protection against observation of meta-data.

This chapter serves the purpose of providing the fundamentals to the aforementioned topics. However, even the experienced reader will find interesting details that play into the design and implementation of the developed solution.

IT security is fueled by creative attacks that challenge the status-quo and show weaknesses. *Attack boxes* are added wherever an attack illustrates the limitations of the discussed concept and idea.

2.1 Terminology in IT Security

Security Guarantees

The classic IT security model is build around the key guarantees of confidentiality, integrity and availability. These concepts are referenced as the CIA[1] triade. Modern literature often extends this by further aspects such as privacy [8]:

Confidentiality describes the protection information against disclosure to a third party. If Alice shares a message with Bob in a confidential manner, no one else can access the private information of that message. However, the event of message passing itself might be observable.

Integrity ensures that an exchanged message cannot be altered by a third party without being noticed by the recipient.

Availability refers to protection against distortion by an attacker. As an unavailable system is of no use, the design goal is to provide a scalable way of increasing the effort required by an attacker to do so.

Accountability allows to prove that certain actions and information have been issued by a certain participant. Using accountable methods he cannot deny his actions later.

Authenticity ensures the identity of a participant or object. For message exchange, both parties usually want to make sure that the message came from the expected sender.

Privacy or Anonymity captures the ability of an individual to be indistinguishable from a set of peers. The degree of anonymity is often coupled to both one self's and others' behavior.

For this paper, the guarantees of confidentiality, integrity, authenticity and anonymity are the most interesting and important ones.

Attacker Model

For a profound and meaningful analysis of security solutions, one must know about the anticipated attacker. As absolute security is considered unachievable, this reductions allows to argue about relative security for different scenarios. This is done by specifying an attacker model that defines the adversary's motivation and abilities. The following paragraphs introduce the most relevant characteristics for the topic of anonymous communication as in [9].

[1] An acronym of the covered guarantees and not the US intelligence service

Position: An *internal* attacker participates in the anonymous communication system (e.g. as a relay node). An *external* attacker only controls the connections between the nodes.

Scope: Here, we differentiate between *global* and *local* attackers. While a global attacker controls all connections of the system, a local attacker is limited to a (geographical) subset. For example, he can only control one autonomous system (AS).

Flexibility: A *static* attacker has predefined observation points (e.g. controlled nodes) and cannot change them. An *adaptive* attacker is able to change its observation points and "follow" the traffic. Global attackers are always considered adaptive.

Participation: While a *passive* attacker is limited to observe the traffic, an *active* attacker can also manipulate traffic. This includes dropping packets, replaying them at a later point of time or modifying their content.

An attacker might not need (nor intend) to fully break an anonymity scheme. Depending on the scenario, he might succeed neither knowing the information exchanged nor discovering new sender-recipient relations.

The *traffic analysis attack* denotes the idea of extracting intelligence from traffic patterns. This covers e.g. packet sizes, inter-packet delays or times where no conversation takes place. Such data can easily be recorded and analyzed offline. For example, from an increasing communication intensity that finally drops, the adversary might deduce that a plan has been coordinated and finalized. Regarding Internet telephony, there is an interesting approach where one can reconstruct speech just from the different packet sizes of a variable bitrate codec (see box 2.3.1).

Another example is *traffic confirmation*. Here the attacker starts with a strong a-priori suspicion that A and B are communicating and seeks to confirm it. As most solutions are designed to only prevent the discovery of sender-recipient pairs, the confirmation of a suspicion can often be performed involving significantly lower efforts. A typical (active) approach is to distort the traffic on the sender's side and observing the same distortion on the recipient's end.

2.2 Anonymous Communication Systems

This section introduces anonymous communication systems ranging from proxy servers to more elaborate peer-to-peer (P2P) systems. However, we will discover that they lack resistance against traffic confirmation attacks by a global passive adversary (GPA) and/or are not suitable for voice-over-IP applications. Table 2.1 gives an overview of the covered systems and our solution named HRTP.

	VoIP	IP	LPA	GPA	GAA+	Efficiency
Proxy	●	●	●	○	○	●
MIX	○	●	●	◑	◑	●
Tor	○	●	●	○	○	◑
Drac	●	●	●	◑	◑	◑
ISDN	●	○	●	●	◑	○
HRTP	●	●	●	●	●	○

Table 2.1: Comparison of well-known anonymous communication systems. *VoIP:* The system is usable for Internet telephony; *IP*: built using an IP network; *LPA:* The system protects against a local passive adversary; *GPA:* The system protects against a global passive adversary; *GAA+:* The system protect against a global active adversary including protection against traffic confirmation; *Efficiency:* Efficiency as ratio between payload traffic and total traffic;
●=fulfilled, ◑=fulfilled with restrictions, ○=not fulfilled

2.2.1 Proxy Servers

Using *proxy servers* is a simple, straight-forward solution for escaping a local adversary. The traffic of the client is tunneled to a remote proxy server which is out of reach of the local attacker. Then, the proxy uses its own identity to forward the traffic and tunnels replies back to the original client. This approach requires the client to fully trust the proxy as it can reconstruct the client's communication pattern. Therefore, proxies are rather a pseudonymous communication system than an anonymous one.

However, proxy servers come with many advantages. Usually they provide a high-bandwidth and a low latency, which makes them suitable for every internet applications. Many proxy servers have application-layer aware functionality and can e.g. cache websites and perform privacy-preserving protocol modifications.

Today, often a *virtual private network* (VPN) is used to achieve a similar effect. The main advantage lies within the fact that a VPN routes the complete traffic of a machine. Therefore, it is less likely to leak information via third-party plugins e.g. Adobe Flash.

Common use cases for proxies and VPNs are securing company traffic and circumventing geo-blocked websites. The former allows the IT to apply security regulations to all company traffic, since even remote traffic is tunneled through the main gateway beforehand. The later makes use of the fact that the proxy is physically located within one of the white-listed countries. Therefore, the client can access e.g. geo-blocked services from abroad.

2.2.2 MIX-Networks and Onion-Routing

Both MIX networks and onion-routing use (multiple) intermediate nodes to hide the actual path of a message. In general, the batch processing of MIX networks, lead to higher end-to-end latency. Both provide anonymity even with some intermediate nodes being compromised by an attacker.

MIXes

The idea of MIX networks has been originally developed by David Chaum [10]. In a MIX networks, intermediate nodes called mixes are designated for forwarding packets. They collect a certain amount of packets, process them and send them all in one batch. The links to and from the mixes are encrypted s.t. packets entering and exiting the mix appear different to an observer. When sending the batch of packets, there is no particular order of the packets. The changing encryption and the re-ordering makes it hard for an attacker to "follow" a certain packet. Implementations of the original concept are mixmaster [11] and mix-minion [12].

MIX networks make use of different mixing strategies in order to form the individual batch sets. This paragraph briefly covers the most interesting ones[2]. The original strategy is *threshold mixing* where the mix collects exactly n messages before sending a new batch. This, of course, imposes a high delay on most messages. The *timing mix* sends a regular batch every t seconds. However, this can decrease anonymity when there have been only few messages within one such interval. In *pool mixes* a certain amount of messages is hold back every batch. This increases the resistance against $n-1$. The *exponential mix* falls in the same category where messages are independently delayed by a random delay d (exponential distribution).

Attack: $n-1$ attack In a $n-1$ attack, the active attacker can control the incoming messages of a mix. He will replace all other messages except for the one he wants to track, with his own ones. After the mix, he then can easily filter out his own messages and knows the destination of the message of interest. When the messages are dropped instead of replaced, its also referenced as a *trickle attack*.

Onion Routing (Tor)

In onion routing the outgoing packets are encrypted with multiple layers. One for every intermediate node which are named *relays*. Upon receipt, a relay can remove the outermost layer and forward the packet to the next node. We call such a sequence of relay nodes a *circuit*. For an (active) attacker it is hard to determine the complete circuit as he only gains information about the preceding and succeeding nodes.

[2]More mixing strategies and detailed discussion can be found in the introductory survey by Serjantov et al. [13]

The onion router (Tor, [5]) is the predominant implementation of onion routing. It focuses on a practical and easy-deployable overlay network that can be used by any application that supports SOCKS proxies. In order to reduce the latency, the relay nodes of Tor do not perform any of the aforementioned mixing.

Nevertheless, given the limited amount of bandwidth per relay node and the additional hops, the Tor network often imposes a multi-second latency. Additionally, the Tor network is completely built upon TCP/TLS connections which are also used to tunnel UDP packets. The insightful paper by Raerdon et al. [14] performed a detailed dissection of Tor's performance issues and evaluated an alternative implementation using TCP-over-DTLS which could drastically improve the latency. Despite the efforts, the circuit latency still remained >500ms.

The Tor network is not designed to protect against a global passive adversary, but a partial active adversary that controls a fraction ($\ll$ 50%) of the network [5, p.5]. The main focus lies within preventing traffic analysis attacks rather than traffic confirmation.

Attack: Remote covert channel analysis Murdoch et al. [15] showed that it is possible to perform certain traffic analysis given only very limited resources on the attacker's side using a covert channel analysis. Here the attacker does not need any physical access to the relay node or its wires.

In its most common usage for anonymously surfing the web, Tor only provides sender anonymity. However, with its capability called *hidden services*, one can provide services over the Tor network which guarantee both sender and responder anonymity. For this, the services is announced via an *introductory point* and then both sides build a circuit to a chosen *rendezvous point*.

2.2.3 Alternative Approaches

Besides the well-known MIX and onion routing approaches, the literature provides countless alternative approaches for anonymous communication such as ISDN-Mixes, DC-Nets and Drac. However, most of them have remained scientific designs without actual implementations. The following approaches are chosen due to their direct and indirect relation to the solution of this thesis.

ISDN-Mixes

The ISDN-Mixes, as proposed by Pfitzman et al. [16], focus on untraceable *Integrated Services Digital Network* (ISDN) voice calls. It is one of the first technical descriptions of a system that uses constant heartbeat messages (here called "time-slice channels") in order to protect against traffic observations. However, it has been designed before the Internet reached its today's capacity, thus its design requires a special ISDN infrastructure.

In the ISDN-Mix network, local ISDN exchange groups form independent sets that continuously broadcast to their local exchange point. The local exchange point is behind a cascade of mixes. When a participant is currently inactive, it establishes a channel with itself (setting up a connection through the mixes to the local exchange which will mirror the traffic back the same way).

In an actual call, the participant will use the time-slice channels to inform the other party about its incoming call. Both then connect to the local exchange through the mixes and the local exchange forwards the packets to each other. For calls over the long distance network, the local exchange forwards the packets to the other side's local exchange.

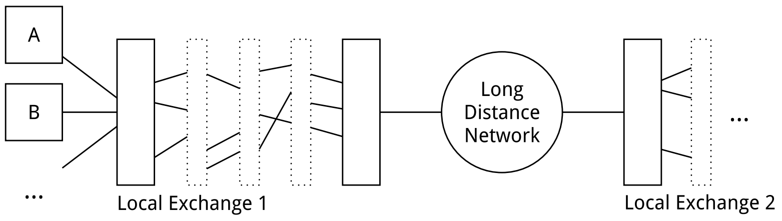


Figure 2.1: Overview of the ISDN-Mix design: The left side shows one local area consisting of the subscribers (A, B, ...) and the local exchange 1. The local exchange is separated in the connection accepting part, the intermediate mixes and the part that finally processes the messages. All local exchanges are connected via the long distance network. Own graphic.

DC-Net

The Dining Cryptographers (DC) problem [17] has been published by David Chaum. It is not designed for voice communication, but shows how mathematical properties can be used for building a completely unobservable communication system: Three cryptographers want to find out if one of them payed the dinner without disclosing who actually payed. In Chaum's solution every cryptographer throws a coin (head=1, tail=0) such that it is only seen by him and its right neighbors. In a next step, every cryptographer announces the result when he exclusively-ors the coins he sees. Except the one, who actually payed: That one negates his result. When all the single results are exclusively-or-ed again, it results in 1, if one of the cryptographers has payed. This is illustrated in figure 2.2.

This concrete example can easily be extended. In order to communicate more than one event ($\hat{=}$ one bit), the protocol is repeated multiple times. For larger dining tables, the cryptographers must form a fully connected graph. However, with increasing number of cryptographers the likelihood of collisions increases. The Herbivore [18] prototype is built upon the concept of DC networks. It uses small sub-graphs in order to scale to a larger number of participants.

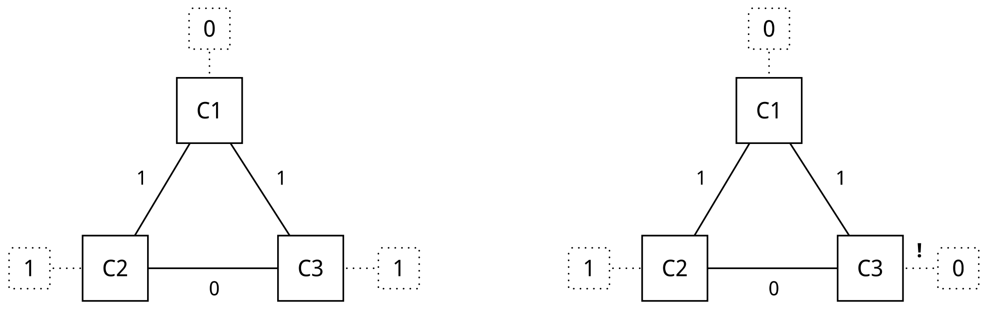


Figure 2.2: The Dining Cryptographers problem with the cryptographers C1, C2 and C3 and their announced results in the dotted boxes. The coin tosses on the edges. Left side: $0 \oplus 1 \oplus 1 = 0$, therefore no one has payed. Right side: C3 negates its announcement, $0 \oplus 1 \oplus 0 = 1$. Therefore, one has payed. Own graphic.

Drac

The Drac concept [19] describes a broadcast system based on an underlying social network. It explicitly focuses on protection of voice-over-IP against a global adversary. The paper's result are based on a software simulation.

During operation, constant heartbeat connections are established to friends of the underlying social network. The heartbeat connections are fully padded with cover traffic, s.t. an observer cannot deduce valuable information. The friendship graph is considered to be publicly available anyway. For establishing a call to a non-friend node, the friends are used as relay nodes.

However, this leaves the caller with the challenge to fully trust his friends (and the set of their friends). This implies that an attacker, which by assumption knows the social graph, has a nicely pre-defined set of nodes that he needs to compromise.

2.2.4 Attacks on Anonymous Communication Systems

This subsection briefly discusses a set of attacks that representatively show limitations of the aforementioned anonymous communication systems. Much more is summarized in [20]. However, most of them do not apply to our solution. Attacks that are concrete to a specific implementation detail (e.g. codec leakage) are presented in "attack boxes" in the right context.

Attack: Statistical Disclosure The idea of a statistical attack is to observe a set of participants and collect statistical measurements about their traffic. Examples for such quantities are consumed bandwidth, inter-packet delay and other patterns. Those are then pair-wise correlated in order to find communication pairs. See: [21, 22]

Attack: Intersection Attacks In an intersection attack, the statistical disclosure approach is repeated several times. The sets of potential communication pairs of each round are intersected until the result is of size one. This allows to use less significant statistical measures. One approach would be to observe the times a peer is online. See: [21].

Attack: Tagging/Watermarking When the normal statistical variations are not significant, one can try to watermark the packets. Wang et al. [23] showed this for VoIP low-latency anonymous networks. On the sender's side, they watermarked the inter-packet delay and were able to correlate this with the resulting packet flow at a receiving node. This can also be an example of an *active traffic confirmation attack*.

Attack: Protocol Leakage An often overlooked problem is the leakage of information via the tunneled protocol that can de-anonymize the user. An impressive example is the Panopticlick[3] demonstration by the Electronic Freedom Foundation (EFF). Solely based on the characteristics of the browser (user agent, headers, installed fonts, ...) many user can be uniquely identified. The VBR codec leakage as discussed in 2.3.1 is another example of this category.

2.3 Voice-over-IP

Voice-over-IP (VoIP) summarizes all techniques that allow speech communication over IP based networks. Before, speech communication (i.e. telephone calls) was mainly routed using circuit switching. The main advantage of circuit switching is that it easily provides bandwidth and latency guarantees. However, building an application specific network is inefficient due to the extra infrastructure and underutilized links. Packet switching networks (such as the global IP network) are more efficient as they allow to multiplex many applications with varying bandwidth on a single link. The modern protocols and significant capacities of today's Internet leave most guarantees a secondary concern.

Today, there are two prevailing applications of VoIP. Firstly, VoIP is replacing the circuit switching backbones of the classical telephone network[4]. This has been transparent to most users, as the existing telephone numbers have been maintained. Secondly, there are direct internet telephony services such as Microsoft's Skype and Facebook's Messenger. Those come with their own identities and provide software that can run on various platforms.

As soon as entering the microphone, VoIP data is a highly perishable good. This section outlines how the typical VoIP pipeline is structured and highlight how implementations deal with the end-to-end latency requirements. An overview on available solutions concludes this section.

[3]https://panopticlick.eff.org/

[4]The number of VoIP-based connections competing with classical phone connections has increased from 7.8m of 38.3m (2010) to 17.1m of 36.9m (2014) in Germany [24, p.75].

2.3.1 The VoIP pipeline

The term *VoIP pipeline* denotes all steps taking place from the voice recording on the sender's side to the the playback on the recipient's side. Figure 2.3 shows the characteristic steps and the intermediate artifacts. They are explained in the following sub-sections.

One must consider that every transformation requires some sort of buffering of the previous artifacts. Therefore, every step adds up to the total end-to-end delay of the voice transported. Good implementations must take in-depth details and interplay of the single steps into consideration in order to provide a reasonable communication experience.

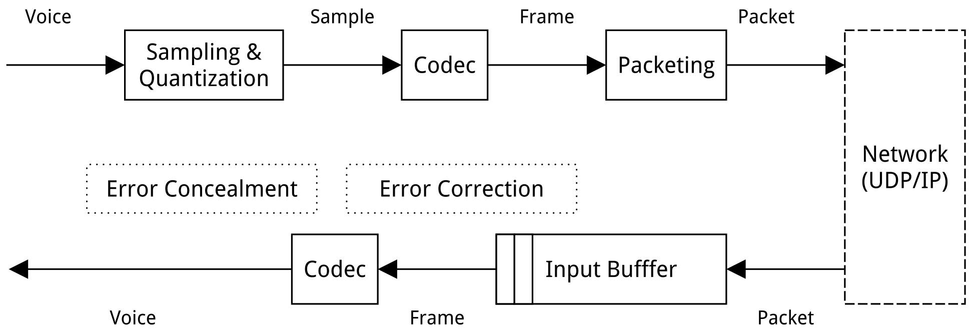


Figure 2.3: High-level overview of a typical VoIP pipeline. The upper part represents the sender's part while the recipient's portion is shown below. Own graphic.

Sampling and Quantization

The first step of the VoIP pipeline is the transformation of the continuous sound signal into a sampled, digital one. This process involves sampling and quantization and is illustrated in figure 2.4. Sampling denotes the reduction of the originally time-continuous signal to a time-discrete signal. Commonly, sampling frequencies for voice are between 8 (narrow-band) to 48 (full-band) kHz. Quantization then performs the mapping of the analogous value to a (finite) set of digital values. This mapping can be linear or non-linear[5] for providing in important regions.

Codec

A codec compresses digital audio into a proprietary codec data stream which has a significantly smaller bandwidth. While there are loss-less codecs, the majority of use-cases use lossy codecs as they provide a higher compression rate. Within the lossy

[5]Examples are the A-law or μ-law algorithm for the Pulse Code Modulation (PCM) method.

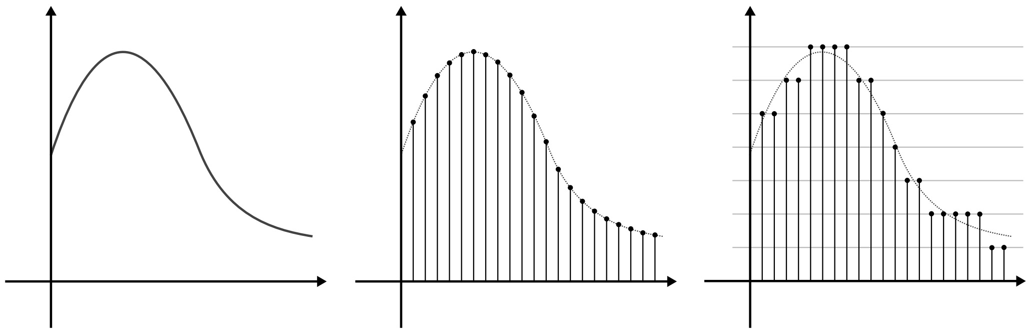


Figure 2.4: Transforming an analogous signal into a digital one (from left to right): The original signal, the signal after being sampled and the signal after sampling and 3-bit quantization. Own graphic.

codec family one differentiates between constant bit rate (CBR) and variable bit rate (VBR) encoding.

The application programming interface (API) of codecs usually expects calls with a predefined amount (e.g. $\{10, 20, 40\}$ms) of audio. Those are then encoded and returned as single *frames*. Each frame typically has a small header segment and can be decoded independently. The larger the frame size, the more efficient the compression becomes, as there is more neighbor data for the compression algorithm and the header portion shrinks. In VBR mode the sizes of the frames vary depending on the complexity of the encoded audio.

Attack: VBR Codec Leakage Encoding voice using a variable bit rate codec has many advantages in a normal VoIP application. Firstly, the bandwidth can be quickly adapted to the current congestion. Secondly, silence can be compressed very efficiently and reduces the overall traffic. However, it has been shown that the resulting variations of the size of encoded packets are sufficient to recover single tones and complete words. [25]

Transport

The created frames then need to be put into packets and handed over to the transport layer. The minimal meta information for the outer packet packet contains a sequence number for detecting loss and a timestamp for synchronization purposes. Typical synchronization challenges are multi-stream coordination (lip sync), error concealment and correction of clock skew.

For real time applications, such as VoIP, the *user datagram protocol* (UDP) is the method of choice. It's state- and connection-less nature imposes a minimal overhead and provides fast and direct delivery. Importantly, it does not integrate any mandatory buffering, s.t. the overall end-to-end delay is no further increased.

When sending packets over the Internet, a good VoIP application has to mitigate a number of different anomalies. The most obvious one is packet loss. In real-time applications, there is usually no time to request a retransmission. If a UDP packet is fragmented along the way and a single fragment is missing, the whole packet is dropped. This increases the overall loss ratio by the factor of fragmentation. Therefore, it is crucial to keep the packets as small as possible. Delayed packets that arrive after their intended playback, are useless and can be considered as lost packets. Interestingly, it's often not the transport that causes the most-significant delay, but the input buffer that has to deal with the effects of transport.

For these reasons, the *transmission control protocol* (TCP) is considered a bad choice for real time applications. It's retransmission ability is typically useless, as the packets would arrive too late. Also, it provides little means of timing control, thus increasing the overall delay. [26, p. 639]

Playback

An often overlooked aspect of VoIP is the playback coordination on the recipient's side. And it is probably the one that is the most challenging to get satisfyingly right.

In the beginning, all received packets are parsed and queued into the input buffer. The buffer serves the purpose of re-ordering late arriving packets. Together with a buffer of out-dated packets it is also used for error correction purposes (see next section).

Next, the packet gets parsed, decoded and queued into the sample buffer. The sample buffer is usually in a special memory region that is available to the sound device driver. This buffer serves the purpose of mitigating delays and short interruptions (e.g. IP congestion, context switches) in any of the previous steps. Often the playback rate is adapted when the buffer fills too slowly or too quickly.

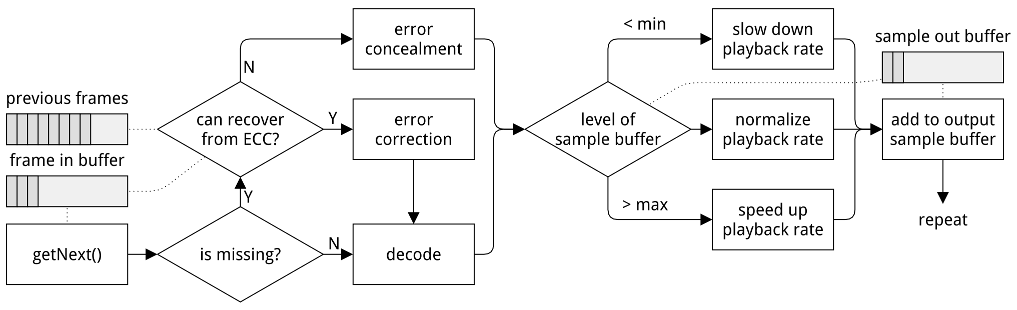


Figure 2.5: Algorithmic overview of a typical playback process. The actual implementation contains more technical details such as coordination of different threads and possible delaying of decisions. Own graphic.

Error Concealment and Correction

Error concealment and error correction are two different approaches for reducing the audible effect of packet loss. However, they can be used in combination where error concealment takes place when error correction failed.

When frames get lost, the audio stream on the recipient's side has a gap. A simple *error concealment* strategy is filling the gap by silence, using the next packet's timestamp for estimating the required duration. Interestingly, substituting random noise instead of pure silence results in better intelligibility[6]. The quality can be further improved by repeating the previous samples or trying to reconstruct the missing samples based on a speech model. [27, p.233]

The two dominant strategies for *error correction* in VoIP are *forward error correction* (FEC) and *audio redundancy coding*. In former, groups of packets are formed and used to compute an additional FEC packet. The computation is usually performed using a simple parity scheme or Reed-Solomon codes [28]. In event of a loss, the FEC packet can be used in conjunction with the other packets of the group for reconstructing the lost packet. One major drawback is that the input buffer must wait for the FEC packet for correction, adding additional delay to the pipeline.

When using *audio redundancy coding* an additional audio stream of very low quality is established. The frames of this stream are then added to the packets of the original stream, but always delayed by an offset of one. In case of a loss, the missing audio sequence can be reconstructed from the redundancy frame of the preceding packet. A comprehensive discussion on error correction for media streams can be found in [29].

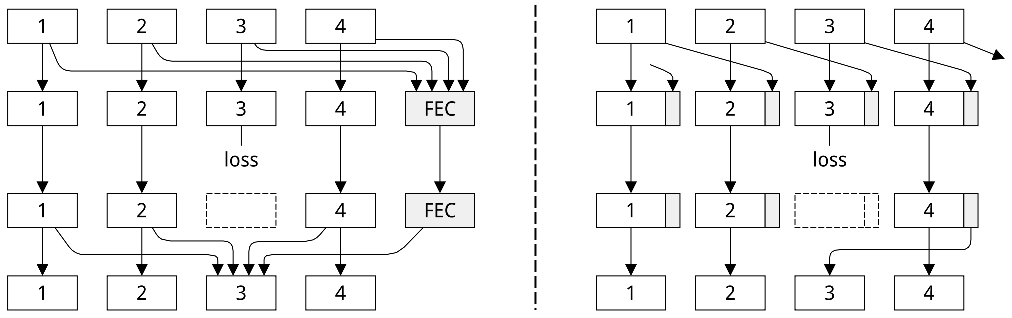


Figure 2.6: When using FEC (left side), the missing third packet is reconstructed from the extra FEC packets and the other successfully received packets. On the right side, it is recovered using the redundant audio frame in the fourth packet. Own graphic based on [29].

[6]This has been shown by various listening tests. It is suspected that the brain's ability to reconstruct speech works better when there is no complete silence. [27, p.233]

2.3.2 State of the Art

Modern VoIP solutions address all those challenges and often provide quality superior to conventional phone lines. One important building block for many applications is the *Real-time Transport Protocol* (RTP). It provides an extensive framework around uni- and multicast transportation of audio and video over IP.

RTP as defined in RFC 3550 [30] comes together with the *Real-time Transport Control Protocol* (RTCP) and both build upon UDP. The RTP packets are transporting data from the sender to the receiver, while the RTCP is used to transport metadata and coordinate sessions. The standard covers multi-sender scenarios, middle boxes called mixes, rate controlling and quality control. By this, it allows interoperability of many applications.

The outline of a RTP packet is shown in figure 2.7. The sequence number is used for estimating packet loss and the timestamp allows to provide synchronization of multiple streams as well as dealing with clock skew. Contributing source (CSRC) identifiers are used when the synchronization source (SSRC) mixes multiple streams. As many fields of the header remain constant or change by a constant difference between two subsequent packets, middle boxes can perform transparent *RTP Header Compression*. This reduces the combined header of IP/UDP/RTP to a total of 2–4 bytes.

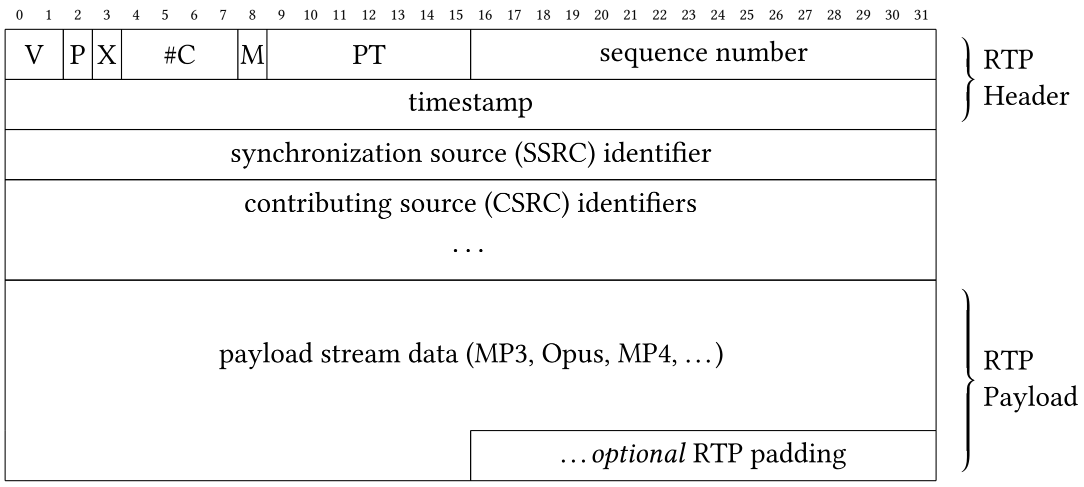


Figure 2.7: Outline of a RTP packet: V = version, P = padding bit, X = extension header bit, #C = number of CSRC identifiers, M = payload specific marker bit (e.g. for synchronization frames in video codecs), PT = payload type. Own graphic.

The choice of the payload is not specified by the original standard and left to be agreed on application level. Typical codecs for VoIP are the *ARM-NB* and *ARM-WB* codecs, the *G.7xx* codecs and the open source codecs *Speex* and *Opus*. Opus is used for the HRTP implementation as described in 4.3.3.

Also, the *Session Initiation Protocol* (SIP) should not be left unmentioned. SIP is used for establishing VoIP calls. It can be used in a stand-alone client or in a proxy manner between conventional phones and IP services. SIP takes care of registering clients, ringing the opposite and agreeing on codecs. The media transport is usually using RTP.

2.4 State of the Art in Secure VoIP

Standardized protocols such as SRTP and ZRTP form valuable building blocks for modern and secure VoIP applications. While providing high-levels of confidentiality, integrity and authentication, they all lack strong privacy guarantees against a global passive attacker.

2.4.1 Secure Real-Time Protocol

The Secure Real-Time Transport Protocol (SRTP) as defined in RFC 3711 [31] adds confidentiality, integrity and authenticity to the popular RTP protocol. It does so by encrypting the payload and adding an authentication tag (optional, over the whole packet) to the RTP and RTCP packets. SRTP supports AES-CM and AES-f8 for encryption and HMAC-SHA1 for integrity.

As the header remains unencrypted, an attacker can deduce information from the transmitted timestamps, the sequence number and the packet size itself. However, this allows middle boxes to apply RTP header compression techniques (e.g. RFC 2508) to encrypted packets.

The SRTP key hierarchy has the $master_{key}$ and the $master_{salt}$ as its given root keys. The actual session keys called $srtp\_session_{enc}$, $srtp\_session_{auth}$ and $srtp\_session_{salt}$ are derived using AES-CM as the derivation function and XOR-ing the label (`0x00`, `0x01`, `0x02`, respectively) with the $master_{salt}$:

$$srtp\_session_{\{enc,auth,salt\}} \quad := \quad AES\_CM(master_{key}, label \oplus master_{salt})$$

It is important to note, that SRTP itself does not define any key establishing and agreement steps. It is assumed that key establishing is handled off-band. One example is ZRTP, which is described in the next section.

2.4.2 ZRTP

The *Z Real-Time Transport Protocol* (ZRTP[7]) is a key agreement protocol for establishing SRTP sessions. It runs on the same port as RTP and uses a very similar packet format. This way, it can be integrated transparently as a middleware to existing RTP applications. ZRTP comes with nice cryptographic properties such as *Perfect Forward Secrecy*[8] and protection against *Man-in-the-Middle* (MitM) attacks. The caller is called *initiator* and the callee is referred to as *responder*.

In this sub-section gives a very brief overview on a representative ZRTP session. It skips the details of using cached secrets or multi-streams and focus on giving a high-level understanding of which cryptographic operations are performed in a simple session. Further it omits all technical details, such as message formats, algorithm negotiations and context parameters in hash functions.

ZRTP is multiplexed on the same port as RTP. In order to be distinguished reliably, it adapts to the RTP format, but uses an invalid RTP version of `0x00`. A client supporting ZRTP can then check the field that normally carries the timestamp. In ZRTP, this field has the "magic cookie" value `0x5a525450` (hex-ascii for zrtp). The first message received will be a `Hello` message that is replied to by an `HelloAck`.

In the next step, the responder computes its Diffie-Hellman (DH) key-pair DH_pub_R, DH_secret_R and commits using a `Commit` message. The initiator then generates its DH key-pair. Now the `DHPart1` message containing DH_pub_I is send and the responder responds with his `DHPart2` message containing DH_pub_R. Both parties then can compute the DH value. This is used in conjunction with a hash h_m over all previously exchanged messages to compute a shared secret s_0. The fields $s_{\{1,2,3\}}$ refer to information from the cached secrets[9] and are *null* during the first call.

$$\begin{aligned} DH &= {DH\_pub_I}^{DH\_secret_R} \bmod q \\ &= {DH\_pub_R}^{DH\_secret_I} \bmod q \end{aligned}$$

$$s_0 = \mathcal{H}(DH, h_m, s_1, s_2, s_3)$$

[7]The Z is meant as a reference to Phil Zimmermann, the main inventor, but *Zimmermann* is not part of the acronym itself.

[8]In a cryptographic protocol with Perfect Forward Secrecy, an attacker, who manages to compromise the long-term secret key and/or a single session key, cannot decrypt messages exchanged earlier. This is usually achieved using Diffie-Hellman key exchange for establishing temporary session keys.

[9]The idea of *cached secrets* is similar to key continuity of the SSH protocol. After the first call, both sides store a retained secret that is derived from the shared secret s_0. By this, the parties can be sure to talk the already authenticated partner next time by comparing their secrets. Perfect forward secrecy is ensured by updating the cached secrets at the beginning of every session.

The shared secret s_0 is used for deriving the master keys and salts for SRTP for each direction. Also, the ZRTP session key $zrtp_{sess}$ is derived that is used in the context of supporting multiple streams in a single RTP session:

$$\begin{aligned}
srtp\_master_{key,I} &= \mathcal{K}(s_0, \texttt{"Initiator SRTP master key"}) \\
srtp\_master_{salt,I} &= \mathcal{K}(s_0, \texttt{"Initiator SRTP master salt"}) \\
srtp\_master_{key,R} &= \mathcal{K}(s_0, \texttt{"Responder SRTP master key"}) \\
srtp\_master_{salt,R} &= \mathcal{K}(s_0, \texttt{"Responder SRTP master salt"}) \\
zrtp_{sess} &= \mathcal{K}'(s_0, \texttt{"ZRTP Session Key"})
\end{aligned}$$

ZRTP uses *Short Authentication Strings* (SAS) in order to prevent MitM attacks. The SAS_{hash} finally depends on the DH value. An attacker in the middle, will fail to generate two DH key exchanges with both parties resulting in the SAS message due to the early `Commit` message [32, p. 20f]. Even for SAS as short as 16 bit. Using the rightmost n bytes as the SAS_{value}, it can be rendered as Base32 (`B32` with $n = 20$, e.g. “bjt1”) or using the PGP word list (`B256` with $n = 16$, e.g. “glitter unicorn”). The SAS strings are compared over the then established voice channel using human judgment.

$$\begin{aligned}
SAS_{hash} &= \mathcal{K}(ZRTP_{sess}, \texttt{"SAS"}) \\
SAS_{value} &= rightmostbits(n, SAS_{hash})
\end{aligned}$$

2.4.3 Overview on Available Solutions

Finally, we will give a brief overview on chosen existing solutions that promise secure VoIP communication, namely OnionPhone, Skype and SilentCircle.

VoIP over Tor One tempting approach is to tunnel VoIP applications over an existing general purpose anonymity communication system such as Tor. The OnionPhone[10] is an open source implementation that provides sender and recipient anonymity using Tor’s hidden services. It can be used as a plugin for the Torchat application and comes with additional features such as a *vocoder* that can distort the speaker’s voice. The OnionPhone comes with an additional layer of encryption, mutual authentication, perfect forward security and key management.

Although Tor is a “low-latency” network, the end-to-end latency is often still within the range of several seconds and makes voice conversation difficult. Also, we have seen that Tor does not protect against strong attackers such as an global passive adversary and is vulnerable to traffic confirmation attacks.

[10] http://torfone.org/onionphone/index.html

Skype The Skype[11] network is probably the world's most popular direct Internet telephony system. It comes with clients for various platforms and despite its central registration server, it is mainly a P2P system. Its developers claim that it uses strong encryption for skype-to-skype calls, but the proprietary protocol makes independent research difficult. It provides little to no privacy guarantees due to its direct communication links and its compliance with lawful interception.

Silent Circle The SilentCircle[12] company offers commercially available solutions including secure telephony, messaging and mobile phones. It's technology is based upon ZRTP (see 2.4.2), thus providing its MitM defenses and strong cryptographic operations. However, it does not address any anonymity guarantees, but focuses on avoiding compromised devices by providing a hardened smartphone.

[11]http://www.skype.com/en/
[12]https://home.silentcircle.com/

CHAPTER 3

Approach & Design

"Simplicity is prerequisite for reliability."

— Edsger W. Dijkstra

The proposed design is based on a broadcast scheme. All participating peers establish continuous pair-wise streams by sending encrypted packets of same size at a constant rate. The continuous broadcast without exploitable[1] metadata makes all communication *completely unobservable* for a global attacker even with a strong a-priori suspicion. By splitting the communication into a control stream and a data stream, both low latency and strong protection are achieved.

Gateways separate the observable Internet from the unobservable internal network where the handsets of the local users are registered. They also manage the handsets' keys and are the origin of the broadcast traffic. By this separation, the status of the handsets is not leaked, as the traffic is never disrupted.

3.1 Setting and Scope

The proposed solution anticipates a global active adversary (GAA). Even in such a scenario, it provides its strong unobservability guarantees against communication-confirmation attacks. However, it is neither censorship-resistant nor stenographic. The approach as described here only scales to a small number of participants and does not provide bootstrapping for new users.

[1]An information is called *exploitable* if it unveils new information about a secret property, the change of a secret property or the exchange of such.

3.1.1 Adversary and Assumptions

The description and analysis anticipate a global active adversary (GAA). The adversary controls all of the public network from the interface of one gateway to the interface of any other gateway. Let Y be such an adversary. Y has capabilities to record all traffic on every link, analyze it instantaneously and perform modifications at any point. Further, Y can participate in the protocol with multiple identities as Y has access to all specifications and source-code. This, of course, makes Y also an global passive adversary (GPA) that can act adaptively.

The attacker has a strong a-priori suspicion of two known participants who are communicating or not. The traffic confirmation attack is considered successful if the attacker can verify his hypothesis with $p \gg 50\%$.

The guarantees of unobservability, confidentiality and integrity are built upon two relatively weak assumptions: Firstly, it is assumed that all communication behind the gateway (that is, between gateway and handsets) happens in an unobservable private network. Secondly, the assumption is made that carefully applied encryption and signatures do not leak any information about the used keys or the content processed.

3.1.2 Non-Covered Aspects

The current solution provides no decentralized bootstrapping for joining clients. A system with multiple users is expected to be set up manually by exchanging public keys and IP addresses. This information is entered locally in a configuration file and remains static during operation. This is intended as configuration changes might disclose exploitable information to the attacker.

Another aspect that is not covered by the approach presented in this chapter is scalability. By using broadcast traffic, the required resources grow supra-linear with the number of participating users. While this is acceptable for the aforementioned use case of embassies, this is definitely limiting the number of feasible scenarios. Chapter 6 discusses how the system can be extended in order to provide a better scalability.

A global active attacker Y' can not only observe packets but also manipulate traffic. Y' might simply drop all packets of the protocol, thus attacking the system's availability. The system is not censorship-resistant, but it never leaks any information about communication when disturbed in any manner.

While providing unobservable VoIP, it is no stenographic system. An attacker (even with very limited resources) can determine that the system is used and who is participating. However, it remains completely unobservable (following the definitions in the first chapter) as the attacker cannot determine that there is meaningful communication happening over the system.

3.2 Approach

In the approach, continuous traffic is sent between each pair of participants. The packets are encrypted such that only the recipient can decrypt them. As the packets have a fixed size and are sent at a fixed rate, an observer cannot deduce information from the traffic patterns. Each participant is registered at a trusted gateway which ensures that the continuous traffic never stops and that replaces the payload with dummy data when there is no active call.

The user interacts with its terminal using an application on a smart phone or handset device. It is assume that the communication between the handset and the gateway takes place in a private, unobservable network. By this, there is no need for continuous traffic between those two, reducing the required resources on the handset.

3.2.1 Overview, Terminology and Notation

The following paragraphs introduce some notation that will allow to describe the approach more precisely. This will be helpful in the sections on provable anonymity (section 3.3) and when discussing the scalability of the solution (chapter 6). More general notational conventions are given in the appendix A.1.

System S, Gateways G_S and Users U_S

We define a system S consisting of a set of k gateways $G_S = \{G_1, \ldots, G_k\}$ and a set of n users $U_S = \{U_1, \ldots, U_n\}$. There might be multiple users per gateway, but only one gateway per user U_i which we denote by writing $G(U_i)$.

The users connected to a gateway G_j are called *local members* and are referred to as $L(G_j)$. The set of all users known to G_j are referred to as *peers* or $P(G_j)$. This includes all local members. Therefore, $L(G_j) \subseteq P(G_j), \forall j$. Consequently, in a fully-connected system S (that we will most-often assume) it holds that $\bigcup\limits_{g \in G_S} L(g) = \bigcup\limits_{g \in G_S} P(g) = U$ and $\bigcap\limits_{g \in G_S} L(g) = \emptyset$.

All gateways of the system S are connected via a public network and know each others' IP addresses IP_{G_i}. We also define a set of handsets $H_S = \{H_1, \ldots, H_n\}$ and for simplicity, we assume that user U_i uses exactly one handset, namely H_i. A handset H_i is connected to the gateway $G(H_i) = G(U_i)$ via an internal, unobservable network.

Each user U_i has a public-private key pair $KP_i = (pub_i, priv_i)$ consisting of a public key and a private key. Furthermore, each user has an identifier ID_i.

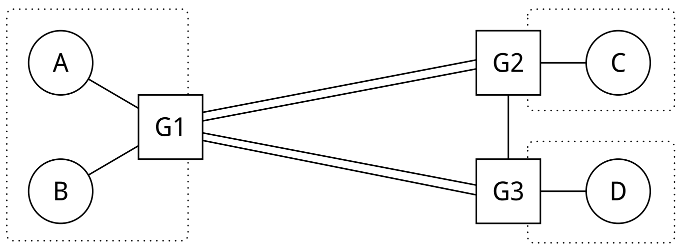


Figure 3.1: Overview of a system S with $G_S = \{G_1, G_2, G_3\}$ and $U_S = \{A, B, C, D\}$. An edge between the a pair of gateways denotes a data- and control stream for each direction.

Control- and Data-Streams D_i, C_i

In the protocol, for every U_i we have two distinct streams (data and control) between $g = G(U_i)$ and all $p \in P(g)$. For this, every peer $p = U_i$ has two distinct ports at gateway: $port_{c,i}$, $port_{d,i}$ for its control- and data stream, respectively.

When saying "U_A sends data stream packets to U_B", we mean that the gateway $G_A = G(U_A)$ sends data to $G_B = G(U_B)$ using the address tuple (IP_{G_b}, $port_{d,B}$). The data stream from U_A to U_B is written as $D_{A,B}$. The i-th packet is referred to as $D_{A,B}[i]$. Similarly, the control stream denoted by $C_{A,B}$.

Due to the broadcast approach, we conclude that $D_{A,x}$ is identical for all $x \in P(G(U_A))$. It is sufficient to write D_A, C_A for referencing the outgoing streams of A and $D_A[i]$, $C_A[i]$ for the i-th packet of these. We denote the point of time the packet was sent as the packet's timestamp $D_A[i]_{time}$.

3.2.2 The Streams

The broadcasted communication originating from user U_i is split into the control stream C_i and the data stream D_i. Both have in common that packets are sent at a fixed rate and have fixed size. These two characteristics never change through the course of operation. More details of the actual implementation can be found be found in the next chapter.

Control-Stream C_i: Control stream packets are sent at a very low rate and have a relatively large payload size. The content is protected using public key cryptography. They are used for signaling session initiation and for initial key establishment.

Due to the use of public key cryptography for both encryption of the payload and authenticity using signatures, their processing requires relatively large computational resources. The used methods must not leak any information about the key used or the content protected.

A packet $C_A[i]$ sent from U_A contains the sender's identifier, a sequence number, the encrypted payload and a signature of the complete content. When no session is established, the payload is encrypted with an unused public key.

$$C_A[i] = \{ID_A,\ seqnr = i,\ \mathcal{E}(pub_{peer}, payload_{C,i}),\ \mathcal{S}(priv_A, C_A[i])\}^{2}$$

Data-Stream D_i: Data stream packets are sent at a high rate and have a size large enough for containing single codec frames. They are protected using symmetric cryptography and contain multiplexed audio data and messages for the final key agreement.

A packet M_D sent from U_A contains the sender's identifier, a sequence number, the encrypted payload and a message authentication code (MAC, $\mathcal{H}$) of the complete content. The used methods must not leak any information about the key used or the content protected. When there is no established session key, a random key is chosen.

$$D_A[i] = \{ID_A,\ seqnr = i,\ \mathcal{E}(session_{enc}, payload_{D,i}),\ \mathcal{H}(session_{auth}, D_A[i])\}$$

Dummy Packets When a stream is missing payload for its next packet, the payload is filled up with a random string. The resulting packets are called *dummy packets*. It is important to see that they are still encrypted the same way and have a valid signature or MAC, respectively. An attacker cannot distinguish them from packets containing "real" payload.

3.2.3 The Gateway

The main purpose of the gateway is the handling of incoming packets and the continuous casting of outgoing packets to all connected peers. While doing this, the gateway is in one of two main states: Either there is a session with an explicit remote peer, or there is no current session.

Another important task is the decoupling of the handset from the public observable network. By doing so, no information about the handset can be observed, e.g. its online times, used subnets or general latency. Moreover, this allows for the handset to remain in standby while all costly and continuous cryptographic operations are handled by the more powerful gateway.

For simplicity, this section focuses on a single user A and its gateway G_1. A multi-user gateway can be imagined as multiple single-user gateways running on a single machine. The actual implementation, of course, uses much finer states which are described in detail in the implementation chapter. Figure 3.2 shows the process of a straight-forward call establishment.

[2]Here, $C_A[i]$, as second argument to $\mathcal{S}$, refers to the complete message *without* the signature

State: no peer When no peer is connected, outgoing control packets $C_A[i]$ are encrypted with a random public key, such that no peer can decrypt its content. Similarly, outgoing data packets $D_A[i]$ are encrypted using a random key.

All incoming data packets are dropped, as there is no current session. However, all incoming control packets $C_X[i], X \neq A$ are carefully processed after their signature has been checked using the sender's public key pub_X. When the packet can be decrypted, it was encrypted with A's public key pub_A. This way the gateway knows that it is called and will notify the handset.

Upon acceptance, G_1 changes its encryption key for outgoing packets to X's public key. It also generates an initial session key key_{AX} for the data stream and sends it to X using the next packet in the control stream. Both gateways move into the *established peer* state.

State: *established* peer When the gateway has an active peer X, all incoming packets not originating from that peer are dropped. The current peer's packets' authenticity is verified using their signature and MAC. Voice data is forwarded to the handset. On the other hand, voice data from the handset is accepted and forwarded to X within the data stream.

When there has been no authenticated packet from X for a certain amount of time, the session will time out. The same happens when the user decides to terminate the call. A final *CLOSING* control is sent to the other side. Then the session key as well as the outgoing public key is replaced by a random key. The gateway is now in the *no peer* state again.

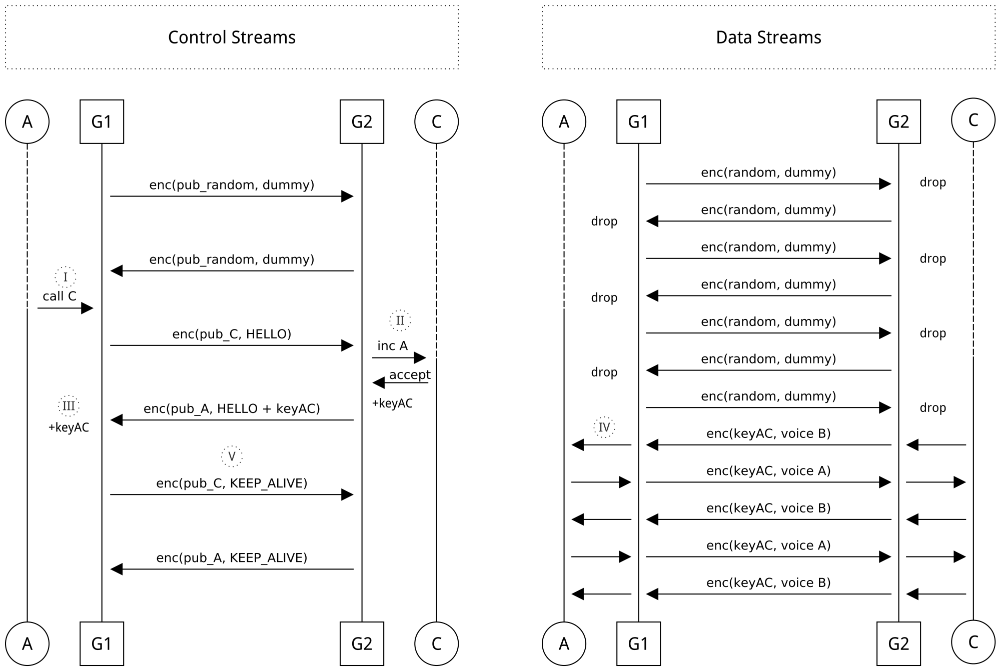


Figure 3.2: A simplified sequence diagram showing the interaction of *A* calling *C*. The transition from dashed to solid lines show the change of the state (*no peer* to *established peer*). **I:** *A* initiates the call by informing his gateway *G1*. When the next control packet is due, *G1* will send a *HELLO* encrypted with *C*'s public key to *G2*. **II:** *G2* will signal the incoming call to *C* which accepts it. *G2* computes an initial session key *keyAC*. **III:** *G2* sets its outgoing key to the public key of *A* and transmits *keyAC* within the next control packet. **IV:** Both gateways can now encrypt and decrypt the data stream and forward the payload to the handsets. **V:** As initial signaling is done, the data stream payload is no longer of interest. By still using the opposite's public key, the session does not time out. Own graphic.

3.3 HRTP is Unobservable

In this section it is shown that the proposed communication system is unobservable as per definition below. Without loss of generality, the communication stream $C = C_A, D = D_A$ from U_A to U_B is discussed.

3.3.1 Definitions and Axioms

Our more formal definition of unobservability is built on the understanding of exploitable information:

Definition 1 An information is *exploitable* if it unveils new information about a secret property, the change of a secret property or the exchange of such. An example for such an property is the internal call state of the gateway.

Definition 2 A stream is *unexploitable* iff only A and B can extract exploitable information from its packets. A stream is exploitable iff any of its packets is exploitable for $Y \neq A, B$.

Definition 3 A communication system S between A and B is unobservable if all of its communication (i.e. its streams) are unexploitable.

The only assumptions made are that the cryptographic routines do not leak information about the used key or plaintext:

Axiom 1 The encryption routines $\mathcal{E}_{symm}$ and $\mathcal{E}_{asymm}$ do not leak any information about the used key or encrypted content except the number of bytes encrypted. Using (implicit) nonces within the encrypted payload, the ciphertext is different every time. Without the key, the ciphertext cannot be distinguished from a random string. Note that this implies that same plain text and key combinations result in different cipher text

Axiom 2 The signature routine $\mathcal{S}$ and MAC routine $\mathcal{H}$ do not leak any information about the used (private) key. Without the key, the MAC cannot be distinguished from a random string.

3.3.2 Showing Unobservability

We first show that individual fields of the packets are unexploitable. Those lemmata are then combined for showing the unobservability of the communication system between A and B against any $Y \neq A, B$.

Lemma 1.a The timestamp of the packets $C[i]$ and $D[j]$ is not exploitable, $\forall i, j \geq 0$.

Proof: Full induction for $C[i]$ ($D[j]$ analogously):

$i = 0$: The initial timestamp is per definition $C[0]_{time} = 0$.
$i = 1$: The timestamp of the second packet does not reveal any information, as its difference to the first packet's timestamp appears random to an observer.
$i > 1$: The timestamp for any further packet can be predicted. For this, Y computes $\Delta_C = C[i-1]_{time} - C[i-2]_{time}$. Then the timestamps of the next packets are $C[i]_{time} = C[i-1]_{time} + \Delta_C$. Information that can be predicted cannot disclose any secret information. □

Lemma 1.b The size of the packets $C[i]$ and $D[j]$ is not exploitable, $\forall i, j \geq 0$.

Proof: As per specification, the size of $C[i]$ and $D[i]$ remains constant. Therefore, they are predictable and do not disclose any secret information. □

Lemma 2.a The source identifier field the stream packets $C[i]$ and $D[j]$ is not exploitable, $\forall i, j \geq 0$.

Proof: The source identifier field is always ID_A. Therefore, it is predictable and do not disclose any secret information. □

Lemma 2.b The sequence numbers field of the stream packets $C[i]$ and $D[j]$ is not exploitable, $\forall i, j \geq 0$.

Proof: Full induction for $C[i]$ ($D[j]$ analogously):

$i = 0$: For the first packet the source identifier is randomly chosen. Therefore is discloses no secret information.
$i > 0$: The source field $C[i].seqnr$ can be predicted by Y computing $C[i].seqnr = C[i-1].seqnr + 1$. Analogous for $D[i].seqnr$. □

Lemma 3 The signature of the stream packet $C[i]$ is not exploitable, $\forall i \geq 0$.

Proof: We analyze both the steps of signing and verifying:

The process of signing does not provide exploitable information about its input. First, the private key $priv_A$ is not leaked as per axiom 2. Second, the string that is signed is available in the same packet and therefore is not secret.

The verification result of the signature (which can be done by Y) does not unveil any secret information: If it succeeds, it shows that the message has been created by A which is expected. If it fails, either A or an attacker has changed it. This does not unveil any secret between A and B. □

Lemma 4 The MAC of the data packet $D[j]$ is not exploitable, $\forall j \geq 0$.

Proof: Following axiom 2 the MAC does not leak any information about secret properties. □

Lemma 5 The encrypted portion $\mathcal{E}(priv_B, payload_{C,i})$ is no exploitable information for $Y \neq B$, $\forall i \geq 0$.

Proof: Follows from axiom 1, as $priv_B$ is only known to B. □

Lemma 6 The encrypted portion $\mathcal{E}(session_{enc}, payload_{D,j})$ is no exploitable information for $Y \neq A, B$, $\forall j \geq 0$.

Proof: Only A and potentially B are in possession of $session_{enc}$. Following axiom 1, no information about the key and the content is leaked to anyone else. □

Theorem: Unobservability The proposed communication system between A and B consisting of stream C, D is unobservable for $Y \neq A, B \in U$, given Y knows U, G, all public keys pub_i and $C[i], D[j]$ with $i, j \geq 0$.

Proof: Unobservability From lemmata 1.a, 1.b, 2, 3 and 5 it follows that the packet $C[i]$ is unexploitable, as every of its fields is unexploitable. From lemmata 1.a, 1.b, 2, 4 and 6 it follows that the packet $D[j]$ is unexploitable, as every of its fields is unexploitable. As $C[i]$ and $D[j]$ are unexploitable $\forall i, j \geq 0$, the streams C and D are unexploitable (definition 2). Therefore, the proposed communication system between A and B is unobservable (definition 3). ■

CHAPTER 4

Implementation

"Let us change our traditional attitude to the construction of programs: Instead of imagining that our main task is to instruct a computer what to do, let us concentrate rather on explaining to human beings what we want a computer to do."

— Donald E. Knuth

The main outcome of this thesis is a prototype providing unobservable Internet telephony. For this the HRTP protocol has been been designed and specified to match the properties declared in the previous chapter. The gateway can be deployed as a stand-alone Java application. The handset is implemented as an application for Android devices.

The main focus is avoiding leakage of any side-channel information from the gateway. Moreover, the challenges of VoIP, as discussed in 2.3, are explicitly addressed by the implementation. All components are designed to run on low-resource devices and provide very low end-to-end latency.

The prototype is built with the intention to serve as a foundation for further extensions and modifications (as described in chapter 6: "Quo vadis: Transformation into a Scaling System") in mind. Therefore, modularity and testability are considered throughout the architectural design.

	HRTCP Control Stream	**HRTP Data Stream**
	Session initiation and coordination	Transport of ZRTP and audio
OSI L3/L4	IP/UDP	IP/UDP
Rate	1/5 s	1/30 ms
Total Size	522 Byte	80 Byte
Payload Size	190 Byte (36.4%)	44 Byte (55%)
Bandwidth	0.102 KiB/s	2.604 KiB/s
Encryption	RSA-2048-OAEP	AES-256
Authenticity	SHA-1 with RSA-2048	HMAC SHA-512 (truncated)

Table 4.1: Overview on the control stream and data stream from a technical perspective. All data is calculated from specification, not measured.

4.1 The HRTP Protocol

The protocol is based on pair-wise data and control streams. The implementations are referenced as the *Hidden-Real-Time-Protocol* (HRTP) stream and the *Hidden-Real-Time-Control-Protocol* (HRTCP) stream, respectively. The name HRTP also refers to the complete protocol which includes both streams.

Despite the similarity of the acronym, HRTP is not based on RTP nor SRTP. The main reasons are that (S)RTP has a relatively large overhead, does not protect the timestamp and provides no conform way to guarantee fixed-sized messages. However, the design adapts a lot of the best practices from RTP. Examples are the UDP transport protocol as well as the formats of sequence numbers and timestamps. This allowed us to easily integrate the existing ZRTP standard for key agreement and exchange.

4.1.1 Streams and Packets

Both the HRTP and the HRTCP streams are generated and sent by the gateway for every single local user. The packets have a specified fixed size and rate - see table 4.1 for an overview. UDP has been chosen as the transport protocol. It provides small overhead and it is, in contrast to TCP, suitable for very low latency communication. HRTP has been designed to be tolerant with regard to packet loss and re-ordering.

HRTP

HRTP stream packets have a fixed size of 80 byte and the effective payload size of 44 byte results in a payload ratio of 55%. HRTP packets are emitted every 30 ms (see table 4.1).

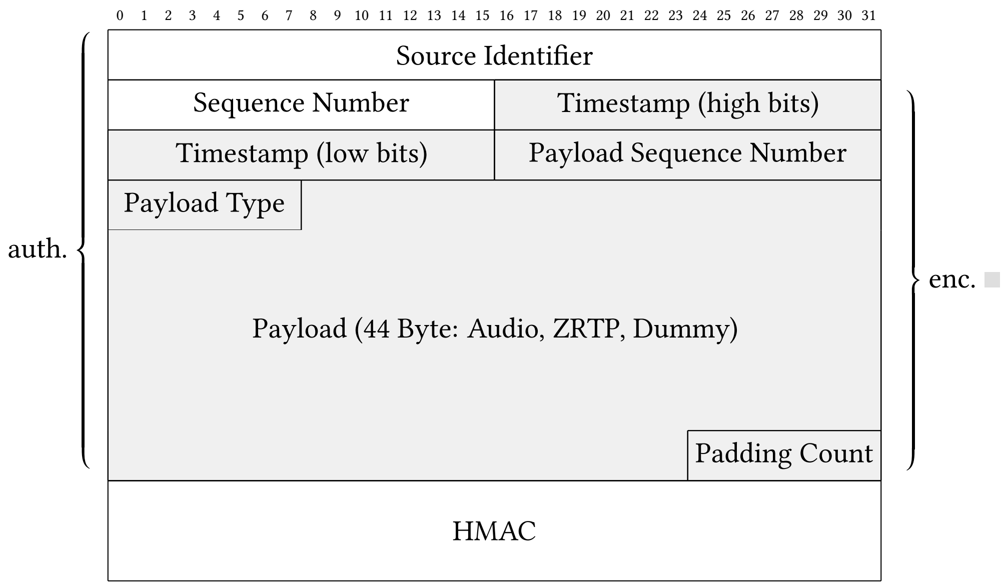


Figure 4.1: Structure of a HRTP packet. Fields highlighted in gray are encrypted using $session_{enc}$. Own graphic.

The packet's private information are protected using AES with 256 bit key size and keyed-hash message authentication code (HMAC) using SHA-512.

HRTP packets start with a *public header* consisting of the source identifier and a sequence number. The source identifier allows the identification of the sender behind middle-boxes where the source IP address has been replaced. The sequence number allows packet re-ordering. In combination with the HMAC it also provides protection against replay attempts.

In HRTP multiple input streams can be multiplexed into the data stream. Examples are audio codec frames, ZRTP messages and dummy traffic. As the bandwidth is fix, payloads from low-priority sources are dropped at will by the gateway. See section 4.2.2 for details.

The *encrypted part* starts with the *private header*. It contains a mandatory 32 bit timestamp and a 16 bit sequence number. The sequence number is used to detect loss of packets in a particular payload stream (e.g. audio codec). It is also used by the playback pipeline in conjunction with the timestamp for error concealment. The 8 bit payload type is used for demultiplexing different payload streams. The valid payload types are `0x00` (`DUMMY`), `0x20` (`AUDIO_OPUS_LOW`), `0x40` (`ZRTP`) and `0x42` (`FRAGMENT`).

The *private payload* is padded with `0x00` to a length of 44 byte. The number of added bytes is stored in the 8 bit padding count field and will be removed at the receiver's gateway. The length of the payload field has been chosen to fit 99% of the frames created by the used Opus codec configuration. For this, a pre-recorded speech sequence has been processed on the handset and the resulting frame sizes have been measured. The results showed that p_{99} = 44 bytes (see figure 4.10 in section 4.3.3).

The portion consisting of the private header and the private payload is encrypted using AES in counter mode with a key size of 256 bit. The initialization vector depends on the packets sequence number for preventing reusage of the cipher stream. The whole packet is authenticated and integrity protected by an SHA-512 HMAC over the complete packet. In order to reduce the size, the HMAC result is truncated to 8 bytes[1].

Important payloads (e.g. ZRTP messages) that are to large for a single payload, are transparently fragmented by the gateway. The detailed implementation is discussed in section 4.2.2.

HRTCP

HRTCP stream packets have a fixed size of 552 byte and the effective payload size of 190 byte results in a payload ratio of 36.4%. HRTCP packets are emitted every 5 seconds (see table 4.1). The packet's private information are protected using RSA with 2048 bit key size and Optimal Asymmetric Encryption Padding (OAEP). Integrity and authenticity is ensured by a SHA-1 RSA-2048 signature.

Similar to the HRTP packet, the HRTCP *public header* starts with a 32 bit source identifier and a 16 bit sequence number. Additionally, the HRTCP contains three 8-bit fields specifying the encryption algorithms used in both the HRTP and HRTCP stream. This allows to change the used algorithms at one gateway without having to update the configurations of all other connected gateways.

The *encrypted part* consists of the 8 bit payload type field and the actual payload. Possible payload types are `0x00` (`DUMMY`), `0x10` (`KEYS`) and `0x20` (`CLOSE`). The unused payload length is padded with zero bytes similar to the HRTP packet.

The encryption is performed using RSA-2048 with Optimal Asymmetric Encryption Padding (OAEP) as specified in [33]. Prior padding schemes had weaknesses when encrypting similar messages using the same key. Using OAEP solves this problem [33, p.19]. Since it uses random padding, the same plaintext results in different cipher texts. However, due to the padding, the usable input size is reduced to $blockLen - 2 \cdot hashLen - 8 = 2048 - 2 \cdot 256 - 8 = 1520b = 191B$. Because of an implementation bug in the OAEP padding of the used cryptographic library (*BouncyCastle*), only 190 byte can be effectively used in the prototype.

The complete message is signed with A's private key $priv_A$ using RSA-2048 with SHA-1. In combination with the sequence number this also provides replay protection.

[1]The standard SHA-512 HMAC would result in a $512/8 = 64$ byte tag – almost doubling the packet size. The implementation truncates the result to 8 byte. Considering the birthday paradox, an attacker still has to try $\approx 5.1 \cdot 10^9$ messages for creating a collision with probability 50%. Given that the attacker can only change a very small amount of data valid for a short amount of time, we think this choice is reasonable. Note: unsafe integrity does not affect the unobservability properties!

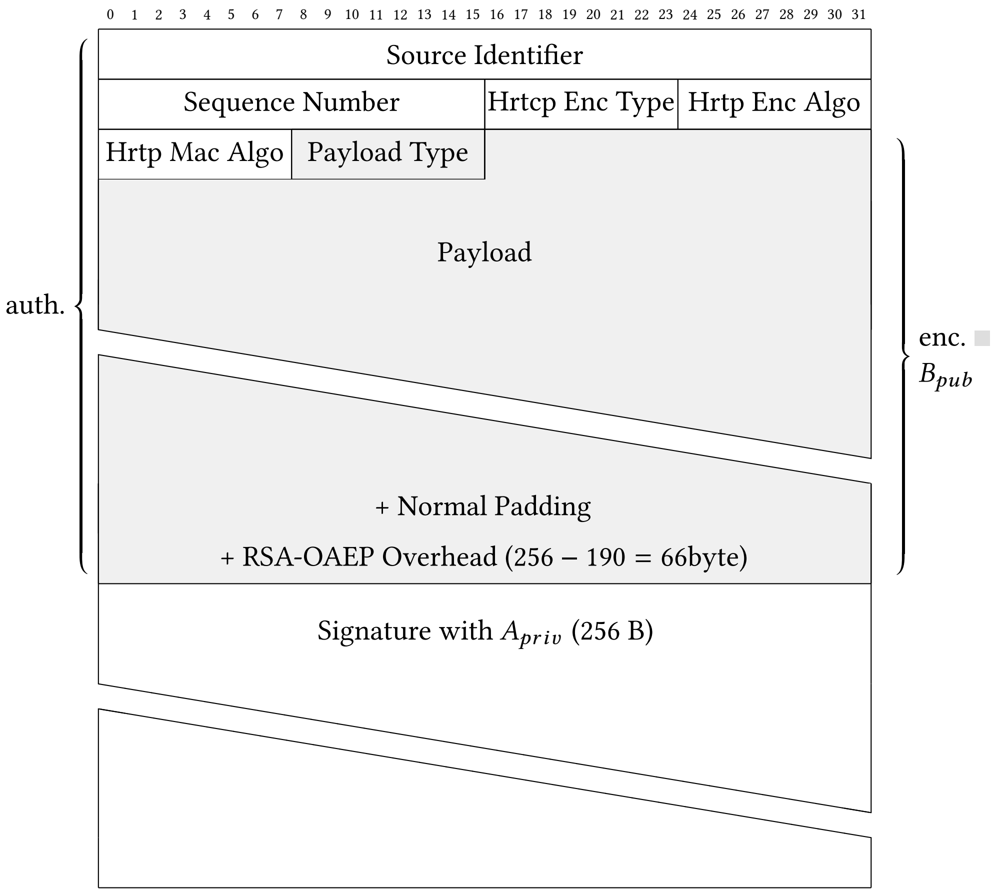


Figure 4.2: Structure of a HRTCP packet from A to B. Fields highlighted in gray are encrypted using pub_B. Own graphic.

4.1.2 Cryptographic Overview

This sub-section explains the keys and their creation that are involved for communication between gateways. The protection of the communication between the gateway and handset is discussed in sub-section 4.3.4. Figure 4.3 provides an overview of all involved keys and their relation.

Upon registration, each user A creates and saves a RSA-2048 key pair consisting of his private key $priv_A$ and the corresponding public key pub_A. Additionally, he creates a dummy key pair and saves only the public key as $dummy_A$. A shares its public key with all peers that he wants to communicate with. The identifier ID_A, is calculated as the 4 most-significant bytes of the SHA-256 hash of his public key:

$$ID_A = \mathcal{H}_{SHA-256}(pub_A)[0:4]$$

Upon establishment of a session, both users create two secrets $session_{master}$ and $session_{salt}$. The secrets are exchanged through the HRTCP channel. The newly received pair, will be stored as $session_{master,in}$, $session_{salt,in}$ and is used to decrypt incoming HRTP packets. The locally created secrets are becoming $session_{master,out}$, $session_{salt,out}$ and are protecting all outgoing HRTP packets.

From those master secrets, the individual keys for encryption, authentication and salting are derived similar to the scheme used in SRTP (see 2.4.1):

$$\begin{aligned}
session_{enc,\{in,out\}} &:= AES_{CM}(session_{master,\{in,out\}}, 0x00 \oplus session_{salt,\{in,out\}}) \\
session_{auth,\{in,out\}} &:= AES_{CM}(session_{master,\{in,out\}}, 0x01 \oplus session_{salt,\{in,out\}}) \\
session_{salt,\{in,out\}} &:= AES_{CM}(session_{master,\{in,out\}}, 0x02 \oplus session_{salt,\{in,out\}})
\end{aligned}$$

Based on the, now secured and accessible, HRTP stream, ZRTP is run. ZRTP will perform a Diffie-Hellman key exchange and therefore provides perfect forward secrecy for all further data in that session. The resulting master keys and master salts are replacing the initial session secrets and key derivation is taking place once more. Comparison of the SAS also protects against the (very unlikely) existence of a man-in-the-middle.

When no session is active, outgoing HRTCP packets are encrypted using the dummy key $dummy_A$. By this, it is ensured that no information, whether encryption has been applied or not, leaks. Equivalently, the session secrets are replaced by random values and all previous secrets are forgotten. All cryptographic operations meet the requirements of the axiom in sub-section 3.3: They do not leak any information about the key or the plaintext used.

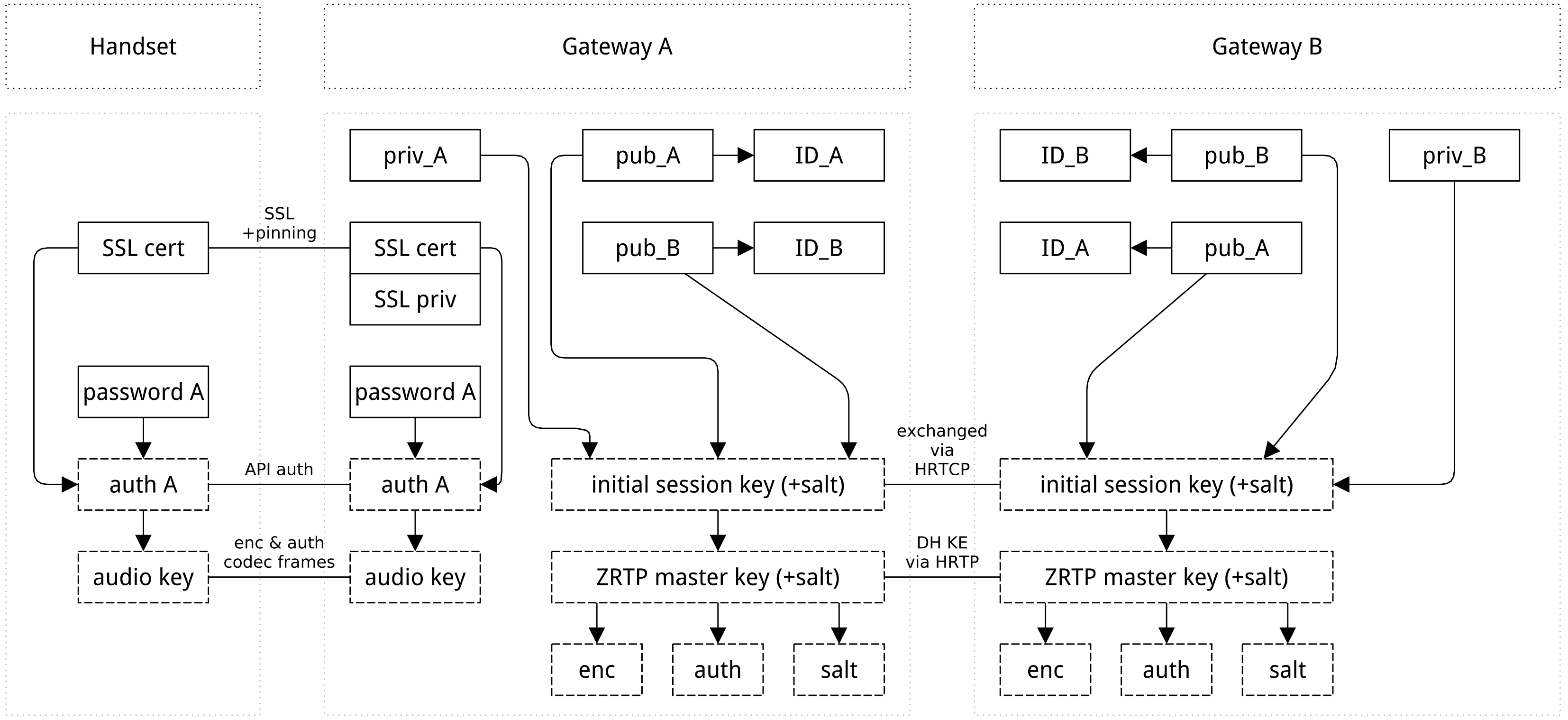


Figure 4.3: Overview of involved keys and secrets. The solid boxes denote permanent keys while the dashed lines show that the key is temporary. Arrows denote depending on the context, that the key has been derived from the ingoing secret or the key has been created on a channel that is secured by the ingoing keys. Own graphic.

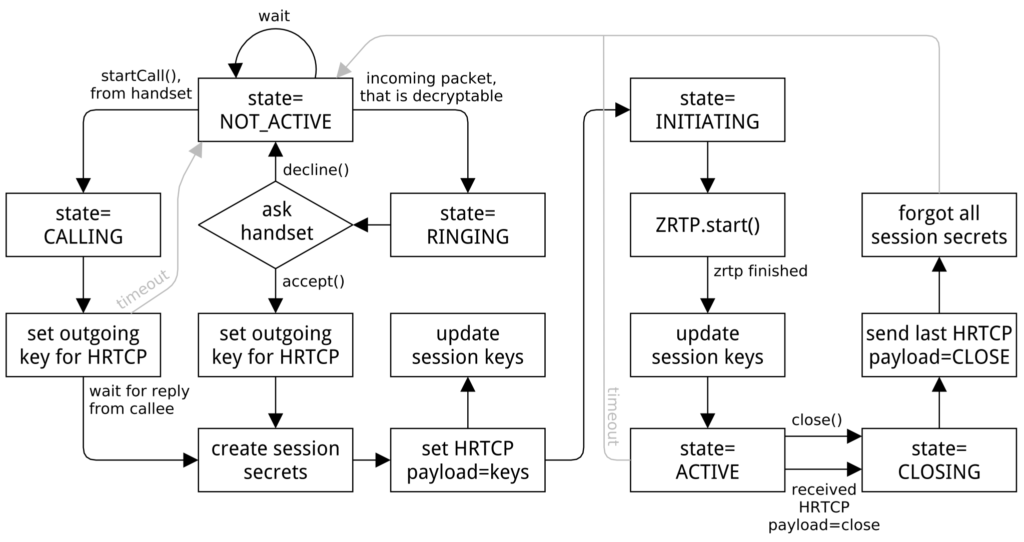


Figure 4.4: Overview of call states and their transitions. The procedure is identical for caller and callee after the "create session secrets" step. Own graphic.

4.1.3 Protocol Flow

This section explains all state transitions in an exemplary protocol run. A graphic illustration can be found in figure 4.4. Every user has a running session on the gateway that represents the communication and cryptographic status. The session is always in one of six states: `NOT_ACTIVE`, `CALLING`, `RINGING`, `INITIATING`, `ACTIVE` or `CLOSING`. The default state is `NOT_ACTIVE`, where no remote peer is set. Here the dummy key $dummy_A$ is set for outgoing HRTCP packets, as well as random session secrets for HRTP.

Calling *B* When the user decides to call another peer *B*, the gateway will set the key for outgoing HRTCP packets to *B*'s public key pub_B. At the same time, the gateway will create $session_{master,out}$ and $session_{salt,out}$. The payloads of all outgoing HRTCP packets are now set to `0x10` (`KEYS`) and contain the aforementioned session secrets.

Receiving a call from *A* The gateway of *B* tries to decrypt every incoming packet with $priv_B$. As soon as one of the decryption attempts is successful and unveils a `KEYS` payload, the gateway is aware that another peer is calling. Let's assume *A* is calling *B*. The gateway then connects to *B*'s handset and ask whether the user wants to accept the call from *A*. The state during this time is `RINGING`.

Only upon acceptance by the user, the gateway updates its key for outgoing HRTCP packets to pub_A. By this *A* learns about the acceptance by *B*, as *A* can now decrypt *B*'s HRTP packets. Analogously to *A*, *B* computes its secrets and sends them as payload of the HRTCP packets. *A* will also parse the received payload and use the keys as local $session_{master,in}$ and $session_{salt,in}$.

Both A and B can now decrypt incoming HRTCP and HRTP packets from the other side. Both advance to the state `INITIATING`.

Initiating and ZRTP During the `INITIATING` state, both partner perform the ZRTP protocol over the HRTP stream. When the ZRTP protocol is finished, the session secrets are replaced and $session_{\{enc,\ auth,\ salt\}}$ are derived again. The session's state changes to `ACTIVE`. Both users can now compare the short authentication string (SAS) on their handset. Now, voice packets are multiplexed into the HRTP stream as well.

Closing the call When a party decides to leave the call, its session moves to the state `CLOSING`. Here, the gateway sends a final HRTCP packet with the payload type `0x20` (`CLOSE`). This signals the other side to end the session as well. Alternatively, a sessions times out when it couldn't decrypt any HRTP or HRTCP packet for the last 15 seconds. Before moving to `NOT_ACTIVE`, A's outgoing public key is replaced by $dummy_A$ ($dummy_B$, respectively) and all session secrets are replaced by random values.

4.2 The Java Gateway

The gateway is implemented as a stand-alone Java application that can run on low-resource devices. It's modular design using components with clearly separated responsibilities allows extension and comprehensive testing. The gateway can run multiple sessions with local members at once, involving multiple datagram services but only on API service.

For guaranteeing unobservability in practice, it is of uttermost importance that the pattern of outgoing packets is completely uncorrelated to the internal state. For this, the protocol state and payload calculation is isolated by using a priority buffer and a dedicated broadcasting scheduler and performer.

4.2.1 Architecture

The architecture of the gateway implementation has been developed with a stand-alone deployment and a modular design in mind. This sub-section discusses the implementation's modularity and describes the single components involved.

Stand-Alone and Modular Design

The gateway can be compiled into a single *Java Archive* (.jar) file that runs on every device providing the *Java Runtime Environment* (JRE). This allows to use a wide range of systems for running the gateway. Due to its low resource consumption, also small devices such as classic Internet home routers can be used as a target platform. For the

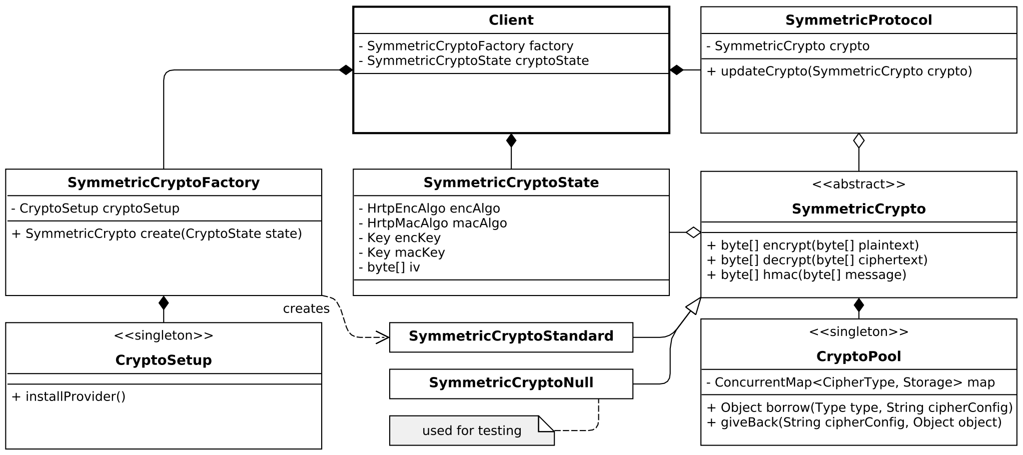

Client
- SymmetricCryptoFactory factory
- SymmetricCryptoState cryptoState
SymmetricProtocol
- SymmetricCrypto crypto
+ updateCrypto(SymmetricCrypto crypto)
SymmetricCryptoFactory
- CryptoSetup cryptoSetup
+ SymmetricCrypto create(CryptoState state)
SymmetricCryptoState
- HrtpEncAlgo encAlgo
- HrtpMacAlgo macAlgo
- Key encKey
- Key macKey
- byte[] iv
<<abstract>>
SymmetricCrypto
+ byte[] encrypt(byte[] plaintext)
+ byte[] decrypt(byte[] ciphertext)
+ byte[] hmac(byte[] message)
creates
<<singleton>>
CryptoSetup
+ installProvider()
SymmetricCryptoStandard
SymmetricCryptoNull
used for testing
<<singleton>>
CryptoPool
- ConcurrentMap<CipherType, Storage> map
+ Object borrow(Type type, String cipherConfig)
+ giveBack(String cipherConfig, Object object)

Figure 4.5: UML diagram showing the pattern used for the symmetric classes of the cryptographic framework component. Own graphic.

use of strong cryptographic ciphers (e.g. AES 256) the *Java Cryptography Extension* (JCE) has to be installed.

The single components are composed using *dependency injection* provided by the *dagger* library. Dependency injection frameworks take care of creating and initializing objects and free the programmer from individually creating the dependencies of a class. This greatly simplifies the exchange of individual implementations. Furthermore, it allows to mock classes for comprehensive and precise unit testing.

Many classes are based on the `Service` interface of the Guava library. The composition of the main components is in fact a hierarchy of `Service` implementations. This simplifies the management of the components' life-cycles and makes them more easily exchangeable. The Guava library provides abstract implementations for services that require to run a thread or scheduled execution. Most prominently, we use the `AbstractScheduledService` for periodic broadcasting of packets.

Components

The gateway implementation can be divided into the following components: The overall orchestration, the bridge towards the handset, the session worker component, the HRTP protocol implementation, the cryptographic framework, the model and configuration as well as the statistics component:

Overall Orchestration: The main entrance point is the `MainApplication` class. First the command line arguments and configuration file are parsed. After this, `MainControl` is injected and used for starting the main services of all sub-components.

Model and Configuration: The model is stored in the singleton `GlobalState` object which consists of all `LocalMember`s, `Peer`s and `Session`s. It also ensures the one-to-one relation between session objects and local members. For storing and retrieving the configuration, the `GlobalState` is converted into a `GlobalConfiguration` object which is then (de-)serialized using the *Jackson* library.

Protocol: The protocol component itself is state-less. The `Protocol` class specifies state transitions and how in- and outgoing packets are handled. For this it has an one-to-one relation with a `Session` object. The protocol components also defines the format of `HrtpPacket`, `HrtcpPacket`, `HrtpPayload` and `HrtcpPayload`. All cryptographic state is referenced in `ProtocolCrypto` which also controls the `ZrtpManager`.

Bridge: The bridge component specifies the communication with the Android handset. The API communication is based on *io.netty* and supports SSL certificates specified in the configuration file. The HTTP handling is setup in `BridgeManager` then parsed and passed to the `ApiHandler`. The bridge component also includes the `AuthManager` for authenticating the handset. This all is described in detail within the Android implementation section 4.3.4.

Crypto Framework: The crypto framework has been designed for allowing testing of primitives and on-the-fly updating of keying material and algorithms. Furthermore, it separates the keying management from the application of cryptographic operations. The keying information, as well as the algorithm choice, are stored in `SymmetricCryptoState`[2] objects.

Those state objects are used as input for the `SymmetricCryptoFactory` for creating `SymmetricCrypto` objects. The `SymmetricCrypto` objects provide the basic `encrypt(...)`, `decrypt(...)` and `hmac(...)` operations transparently. Figure 4.5 shows the classes involved in an exemplary usage scenario. The *Bouncy-Castle* library is used for implementations of ciphers.

SessionWorker: The `SessionWorker` is initiated once per session and managed by the `SessionManager`. It controls all processes for handling in- and outgoing packets as well as the protocol. For ingoing packets the `DatagramListener` is used, and the `DatagramBroadcaster` is responsible for distributing outgoing packets. Figure 4.7 shows the components and their detailed discussion takes place in sub-section 4.2.2.

Statistics: The individual statistic components are injected by classes that provide monitoring data. Examples are bandwidth monitoring for HRTP traffic or measuring the distribution of time required for decrypting incoming HRTCP packets. For this purposes, constant memory implementations (e.g. `MovingAverageListPooling`) are provided and centrally managed.

[2]The same pattern is implemented for asymmetric encryption using `AsymmetricCryptoState` objects

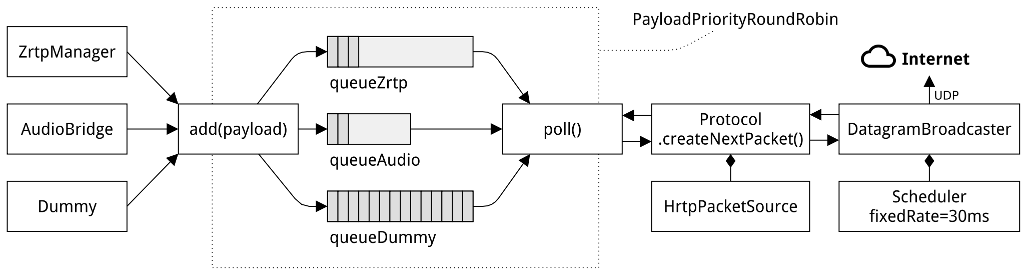


Figure 4.6: Overview of the `PayloadPriorityRoundRobin` usage in the protocol component. Its `FixedSizeQueues` decouple the payload creation from the broadcasting. Own graphic.

4.2.2 Isolated Payload Broadcasting

For the unobservability promises of the HRTP protocol, it is important that the gateway does not leak any side channel information. For instance, the timing of the packets must always be precisely 30 ms. Other operations such as API communication or updating the keys must not interfere with this.

Therefore, the implementation completely decouples the processes of creating packet payloads and broadcasting the actual packets.

Payload Priority Round Robin Datastructure

The `PayloadPriorityRoundRobin` is the central datastructure for isolating the payload creating components from the broadcasting service. It allows to multiplex different input stream into a stream with a limited payload rate and is illustrated in figure 4.6.

Internally it contains different `FixedSizeQueues` for each payload type. `FixedSizeQueues` provide the standard queue interface (`add()`, `poll()`), but also have a maximum size. When adding an object to a full `FixedSizeQueue`, the oldest element is removed before adding the new one.

Our `PayloadPriorityRound` has such a `FixedSizeQueue` for ZRTP, audio and dummy payloads. It provides an `add(HrtpPayload payload)` adding the payload to the queue matching its type. Its `poll()` method first tries to retrieve an element from the ZRTP queue, then from the audio queue and finally from the dummy queue. By this, the first-in element of the queue, which has the highest priority and is not empty, is returned.

The implementation uses a ZRTP queue size of 30, an audio queue size of 5 and a dummy queue size of 10. The high ZRTP queue size provides enough spaces for all fragments of large ZRTP messages. The audio queue is intentionally small to rather drop audio packets than storing them to long ($5 \cdot 30\, ms = 150\, ms$)). The dummy queue size needs to be large enough to never run empty, as this would result in a disruption of the outgoing packets.

Isolated Payload Broadcasting Pipeline

The core classes for the isolated payload broadcast are summarized in the UML diagram in figure 4.7. The core component is the `SessionWorker` which is responsible for both the datagram services and the `Protocol`. It also hold the definite reference to the `Session` model. The `SessionWorker` is initiated and started by the `SessionManager` when the gateway is booted. It will first create the `SocketManager` that provides a transparent facade to the `DatagramSocket`. This allows later extension with support for multicast protocols.

When payload creating services have created a new `HrtpPayload`, they call the `offer(HrtpPayload payload)` method on the `Protcol` object. Where applicable, the `offer(...)` method will split the payload into fragments as described below. After this, the payload is added to the aforementioned `PayloadPriorityRoundRobin` data structure.

Upon creation, the `SessionManager` has created a `DatagramBroadcaster` provided with reference to the `Protocol` object. The `DatagramBroadcaster` implements an `AbstractScheduledService` with a fixed rate of 30 ms. In every iteration it calls the `createHrtpPacket()` on the `Protocol`. In that method, the highest-priority payload is retrieved using `PayloadPriotrityRoundRobin`'s `poll()` method. A `HrtpSource` (not in the UML diagram) is used for decorating the payload with correct encryption and HMAC, as well as increasing sequence numbers. The resulting `byte[]` packet is passed by the `DatagramBroadcaster` to the `SocketManager` along with a list of all peers.

The most important observation is that all steps executed from the beginning of the `DatagramBroadcaster` iteration, are not depending on the payload in any manner. In fact, every iteration the very same steps are executed down to the lowest layer of the AES and HMAC algorithm. This of course does not account for dynamic runtime optimizations by the *Java Virtual Machine* (JVM).

The `SessionWorker` also starts the `DatagramListener` which uses thread pools for handling incoming HRTP and HRTCP packets. The individual thread pools will concurrently call the `handleHrtp()` and `handleHrtcp()` packets on the `Protocol`.

Fragmentation of Large Payloads

With an average size of 147 B, ZRTP packets are much larger than the available payload of HRTP packets (see figure 4.8). Fragmentation is used to mitigate this. The resulting HRTP packets marked with the `FRAGMENT` (`0x42`) type. They start their payload with a 8-bit counter of the current fragment of the payload, a 8-bit integer denoting the total number and the original 8-bit payload type.

The fragmentation is implemented in `Fragmenter` (`ArrayList<HrtpPayload> split(HrtpPayload payload)`) and re-assembling of fragments takes place in `FragmentAssembler`(`HrtpPayload merge(HrtpPayload fragment)`). Both methods are designed such that they run in asymptotic $O(1)$ using concurrent hash maps.

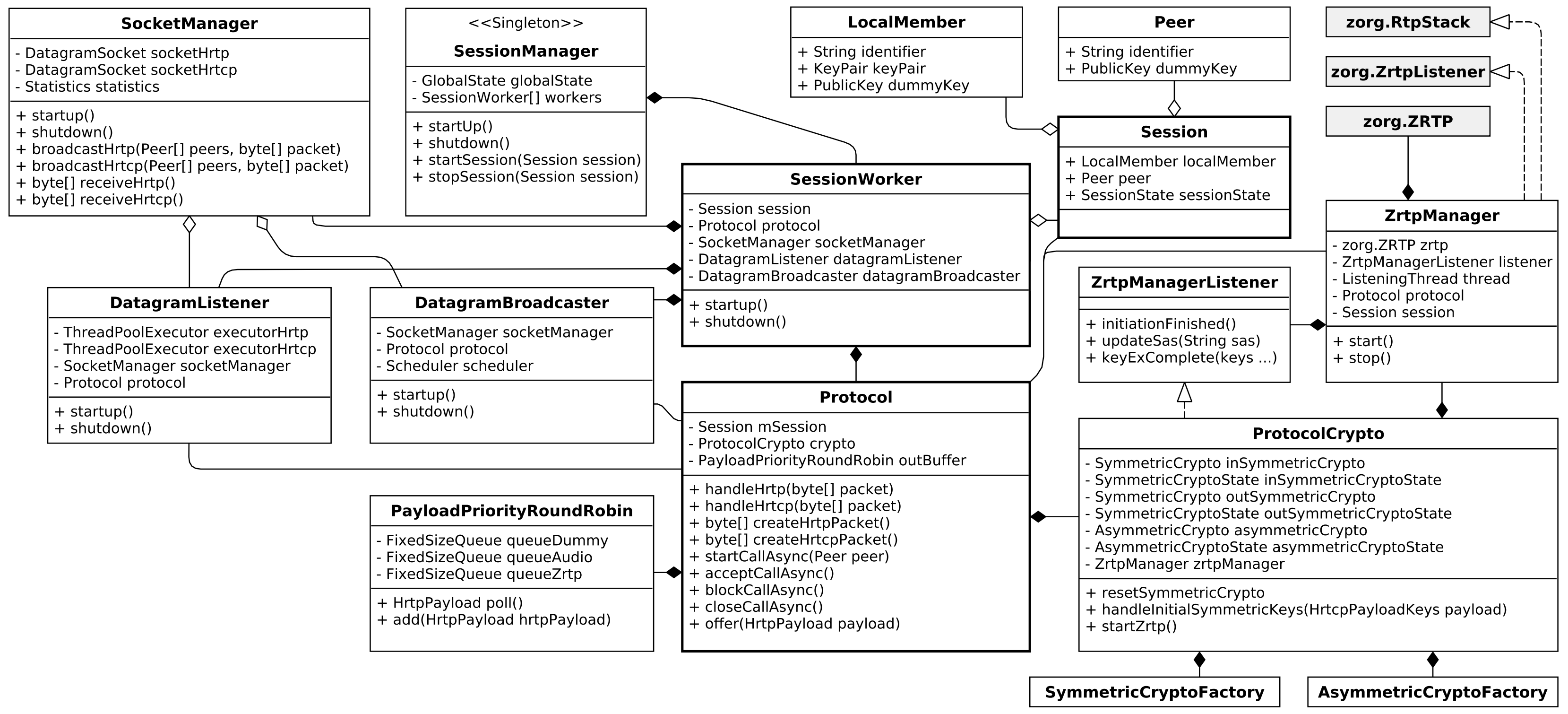

SocketManager
- DatagramSocket socketHrtp
- DatagramSocket socketHrtcp
- Statistics statistics
+ startup()
+ shutdown()
+ broadcastHrtp(Peer[] peers, byte[] packet)
+ broadcastHrtcp(Peer[] peers, byte[] packet)
+ byte[] receiveHrtp()
+ byte[] receiveHrtcp()
<<Singleton>>
SessionManager
- GlobalState globalState
- SessionWorker[] workers
+ startUp()
+ shutdown()
+ startSession(Session session)
+ stopSession(Session session)
LocalMember
+ String identifier
+ KeyPair keyPair
+ PublicKey dummyKey
Peer
+ String identifier
+ PublicKey dummyKey
zorg.RtpStack
zorg.ZrtpListener
zorg.ZRTP
Session
+ LocalMember localMember
+ Peer peer
+ SessionState sessionState
SessionWorker
- Session session
- Protocol protocol
- SocketManager socketManager
- DatagramListener datagramListener
- DatagramBroadcaster datagramBroadcaster
+ startup()
+ shutdown()
ZrtpManager
- zorg.ZRTP zrtp
- ZrtpManagerListener listener
- ListeningThread thread
- Protocol protocol
- Session session
+ start()
+ stop()
DatagramListener
- ThreadPoolExecutor executorHrtp
- ThreadPoolExecutor executorHrtcp
- SocketManager socketManager
- Protocol protocol
+ startup()
+ shutdown()
DatagramBroadcaster
- SocketManager socketManager
- Protocol protocol
- Scheduler scheduler
+ startup()
+ shutdown()
ZrtpManagerListener
+ initiationFinished()
+ updateSas(String sas)
+ keyExComplete(keys ...)
Protocol
- Session mSession
- ProtocolCrypto crypto
- PayloadPriorityRoundRobin outBuffer
+ handleHrtp(byte[] packet)
+ handleHrtcp(byte[] packet)
+ byte[] createHrtpPacket()
+ byte[] createHrtcpPacket()
+ startCallAsync(Peer peer)
+ acceptCallAsync()
+ blockCallAsync()
+ closeCallAsync()
+ offer(HrtpPayload payload)
ProtocolCrypto
- SymmetricCrypto inSymmetricCrypto
- SymmetricCryptoState inSymmetricCryptoState
- SymmetricCrypto outSymmetricCrypto
- SymmetricCryptoState outSymmetricCryptoState
- AsymmetricCrypto asymmetricCrypto
- AsymmetricCryptoState asymmetricCryptoState
- ZrtpManager zrtpManager
+ resetSymmetricCrypto
+ handleInitialSymmetricKeys(HrtcpPayloadKeys payload)
+ startZrtp()
PayloadPriorityRoundRobin
- FixedSizeQueue queueDummy
- FixedSizeQueue queueAudio
- FixedSizeQueue queueZrtp
+ HrtpPayload poll()
+ add(HrtpPayload hrtpPayload)
SymmetricCryptoFactory
AsymmetricCryptoFactory

Figure 4.7: UML diagram showing an simplified overview of the environment of the `SessionWorker` and `Protocol` class. All classes having a `startup()` method are in fact services implementing the `guava.Service` interface. Many fields, methods and relations are left out for easier understanding. Own graphic.

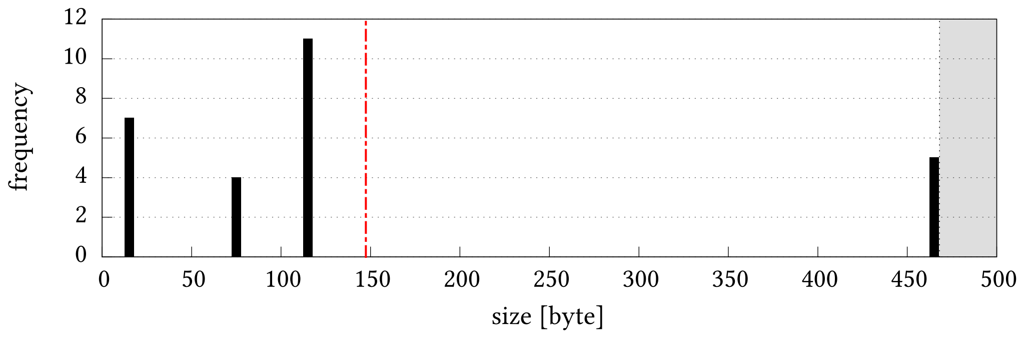


Figure 4.8: Histogram showing frequency of message lengths for both directions of a ZRTP run: $avg = 147.4$, $p_{90} = p_{99} = 468$. Own graphic.

4.2.3 Performance Considerations

The implementation makes use of several measures to reduce computational efforts. The most important ones are efficient packet filtering, thread pooling for packet verification and pooling of cryptographic objects.

Efficient Filtering and Verification of Incoming Packets

Based on the source identifier, HRTP packets that do not match the current peer can be dropped without performing any cryptographic operation. Even if the attacker is able to guess and spoof the current peer identifier, the HMAC verification is of very low cost. All filtering for both HRTP and HRTCP take place before any part besides the source identifier and HMAC (or signature, respectively) is parsed.

Verification of HRTCP packets is more costly due to the asymmetric algorithms that are involved. Here, packets from peers can be randomly sampled when they exceed a certain threshold. This is acceptable, since (1) packet loss is expected by the protocol and (2) once a `KEYS` payload got successfully handled, all communication takes place in the cheap to verify HRTP stream.

Thread Pooling for Handling Incoming Packets

Creation of threads is costly on the *Java Virtual Machine* (JVM). Therefore, the handling of incoming HRTCP and HRTP packets is done using `ThreadPoolExecutor` objects in the `BroadcastListener`. This provides a scalable and reliable setup for multi-core machines. Moreover, it also minimizes thread creation by to caching finished ones. The `ThreadPoolExecutor` uses a task queue for limiting the load. Packets exceeding the queue are dropped. Different thread pools are used for HRTCP and HRTP as the packet types have very different characteristics regarding their handle time and computational efforts

Pooling of Cryptographic Cipher Objects

Early runs showed that frequent recreation of ciphers is a costly operation in Bouncy-Castle. It required costly computations and created a lot of churn with regard to the garbage collection. Therefore, the implementation makes use of pooling for all cipher objects. This is thread-safely implemented in `CryptoPool`.

`CryptoPool` maintains an internal `ConcurrentMap` with `Storage` objects for each different cipher type . Clients borrow a cipher using the `borrow(Type type, String cipherConfig)`. If there is no object available in the storage object, the storage object is filled up to a minimum level. Clients can return the cipher object later using `giveBack(String cipherConfig, Object object)`. It is then made available on the next call of `borrow(...)`.

4.3 The Android Handset

For the implementation of the handset, the Android platform has been chosen. It is the most popular smartphone platform today. Moreover, applications are developed in Java, which allows to reuse code from the gateway. We implemented a custom VoIP pipeline on Android. It uses the Opus codec as native library written in C for high performance. The pipeline is build modular and can generate statistics.

4.3.1 Introduction to Android

The Android platform is an operation system for mobile devices like smartphones and tablets. Being an open and free platform it quickly reached the leading position in the smartphone market. In 2014, over 80% of all sold devices ran Android [34]. This development is mainly driven by the huge engagement of Google and a wide cooperation network of with other partners.

The mobile applications – called apps – are developed in Java and can be installed on devices of different vendors. Performance sensitive parts can be realized in C, shipped pre-compiled and accessed via the Java Native Interface (JNI). The main compatibility factor is the supported API level of the device. The HRTP handset app supports a minimal API level of 15[3] which is supported by more than 85% of current Android devices [35].

Android protects the user's security and privacy by employing a permission model. Every app has to specify a given set of permission it requires. The user then has to accept those during the installation. Our Android app requires the following permissions: The `android.permission.RECORD_AUDIO` for accessing the microphone and `android.permission.INTERNET` for establishing connections to the gateway.

[3]code name: ice cream sandwich, version: 4.0.3+

Applications on Android consist of activities and services. Activities are launched upon user action and are responsible for graphical user interaction whereas services can run independently of the activities and perform background tasks. Besides this, Android uses the pattern of content providers. Content provider have a standardized query interface and offer a specific type of content, e.g. all contacts in a SQLite database.

4.3.2 Architecture

Similar to the gateway, the Android application is built using dependency injection provided by the *dagger* library. Also, `guava.Service` classes are employed for uniform control of sub-components. Both measures drastically improve the modularity and exchangability of the components.

The application has an always running background service `BackgroundAndroidService` that listens for state changes of the session at the gateway. When something changes, events are asynchronously passed via a global `EventBus`. In case of an incoming call (state=`RINGING`), a notification is shown. Stored local contacts are made available using a content provider `PeerProvider`.

The `MainActivity` contains a `PeerListFragment` showing all stored HRTP contacts, a `StatisticsFragment` showing statistics about the gateway and a `TestCodecFragment` for locally testing the Opus codec using different settings. Furthermore, the `LoginActivity` is shown when the app is started for the first time or the user decides to logout.

Figure 4.9 shows an overview of the app's components. All details regarding the VoIP pipeline component including the `CallActivity` and `VoipService` are explained in the section below.

4.3.3 VoIP Implementation

The VoIP implementation is built around the `CallActivity` and the `VoipService`. The `CallActivity` is opened whenever a call is accepted or initiated. On the one side, it uses the `BridgeApi` to keep up-to-date with regard to the session state on the gateway. On the other side, it employs the `VoipService` for managing the VoIP pipeline. Both share a instance of `VoipStatistics` for debugging purposes.

The `VoipServices` manages two independent pipeline services: One for outgoing voice and one for incoming voice. The one for outgoing voice is managed by the `RecordEncodeSendService`. It has an internal thread that reads samples from the microphone and passes them to the `Encoder`. When encoding has finished, the frame is passed to the `AudioPacketSender`. The `AudioPacketSender` then will built a valid `AudioPacket` using `AudioCrypto` and send it as an UDP packet to the gateway.

The incoming voice packets are handled by the pipeline managed by the `ReceiveDecodePlayService`. When an `AudioPacket` is received by the

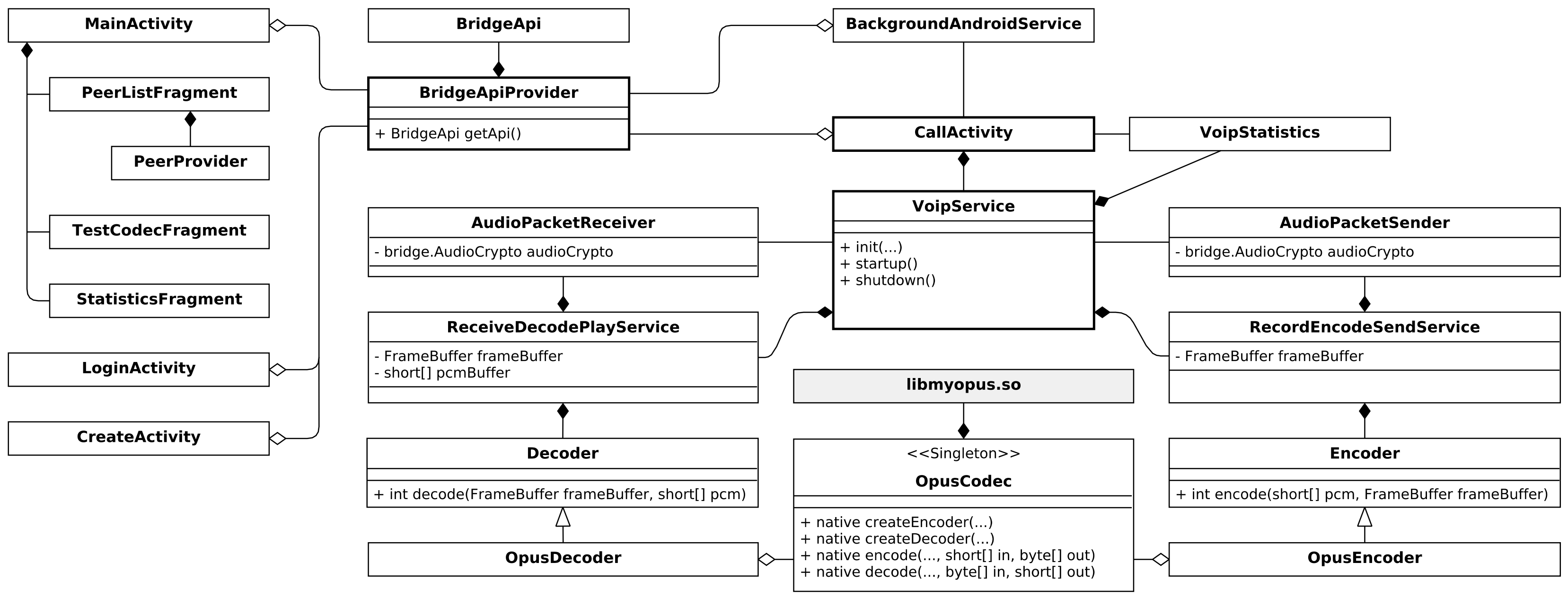


Figure 4.9: Simplified UML diagram showing the central classes and components of the Android application with a focus on the VoIP pipeline. Own graphic.

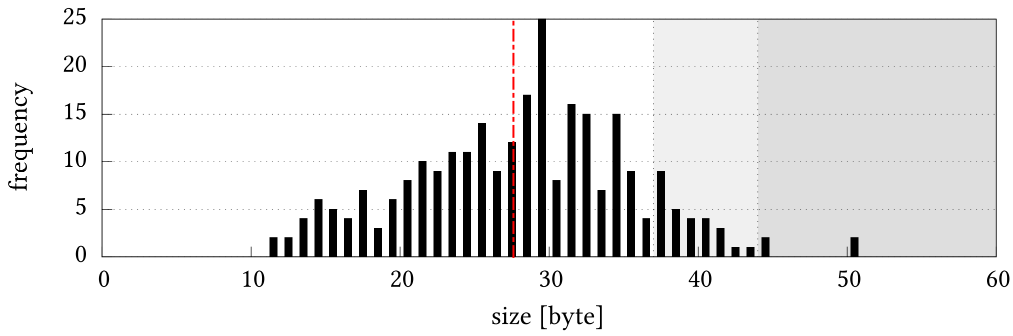


Figure 4.10: Results of running the Opus codec with 8 kHz sampling rate, 40 ms frame size and 6 kbit/s bitrate on a conversation example: $avg = 27.6$, $p_{90} = 37$, $p_{99} = 44$. Own graphic.

`AudioPacketReceiver`, it is passed to the internal, concurrent `FrameBuffer`. The buffer level is used for adaptive playback as described in section 2.3.1. Finally the frames are handled by the `Decoder` and passed to the `AudioTrack` instance for playback.

The Opus Codec

For voice encoding, the free and open-source Opus codec [36] has been chosen. While the encoder is still in active development, listening tests have shown that it meets or outperforms the performance of commercial codecs [37]. The Opus codec comes with error concealment, forward error correction and flexible bitrate and frame sizes. The implementation employs Opus with a default configuration of 8 kHz sampling rate, 40 ms frame size and 6 kbit/s bitrate. Tests have shown no need for explicit forward error correction, as the drop rate was very low.

The Opus integration can be found within the `de.tum.hrtp.android.codec` package. For this, the original C implementation is compiled for the fixed-point ARM/ARMv7 architecture and integrated via JNI. This makes the encoding fast on mobile devices (<2 ms runtime per 40 ms frame). We plan to release the Opus binding as an independent open source project.

The use of a strategy pattern with the interfaces `Decoder` and `Encoder` allows to easily exchange the current implementation found in `OpusDecoder` and `OpusEncoder`. The binding to the native `libmyopus.so` library is performed in the global `OpusCodec` instance. It provides methods for creating new natives instances and performing calls to the native encode and decode methods. The pointers to the native instances of the Opus library are casted to 64 bit long integers, since Java has no native pointer type. Those references are maintained by the `OpusDecoder` and `OpusEncoder` instances.

The codec with is evaluated a 10 second pre-recorded conversation using the aforementioned configuration. The main goal was to find out about the actual frame size distribution. The results of the test can be seen in figure 4.10. From calculations (see below) one expects an average frame size avg_{exp} of 30 B. In the experiment, Opus stayed below that limit with $avg_{real} = 27.6$ B. However, due to the variable bitrate, more than 10% of the frames were larger than 37 B.

$$\frac{6000\,\mathrm{b/s}}{8\,\mathrm{b/B}} \cdot 40\,ms = 30\,B$$

4.3.4 Implementation Details

This section explains how the communication between the handset and the gateway is performed. Furthermore, it briefly discusses the built user interface.

Secure API between Handset and Gateway

Communication between the handset and its gateway is performed via a simple *HyperText Transfer Protocol Secure* (HTTPS) API. The current implementation employs a poll-based architecture where the gateway replies with JSON data structures. The definition of the exchanged objects is specified by plain old Java objects (POJOs) in `.hrtp.gateway.json.*`. That package is linked to the Android app as well. The libraries `Jackson` and `GSON` are used for mapping between JSON strings and the POJOs.

The HTTPS service is implemented on the gateway's side using the `io.netty` library. The request handling on the Android side is implemented based on the `square.retrofit` library. The API communication is secured using SSL with certificate pinning[4]. Upon login, the user identifies itself using its identifier and password. It then gets a temporary authentication token that is added to all following request as a HTTP header parameter.

Secure Audio between Handset and Gateway

For exchange of the audio frames, a temporary UDP connection is created during the call. The format of the audio packets and their protection is implemented in `.hrtp.bridge.AudioPacket` and `.hrtp.bridge.AudioCrypto`. Again, this package is linked into both projects. The packets are encrypted, integrity protected as well as authenticated. For this, the `JsonSession` objects specifies a `srtpKey` chosen by the gateway for each new call. For the encryption AES-256 in counter mode is used and the authentication tag is computed using HMAC with SHA256. The implementation allows the handset to be behind NATs or switch between subnets during a call.

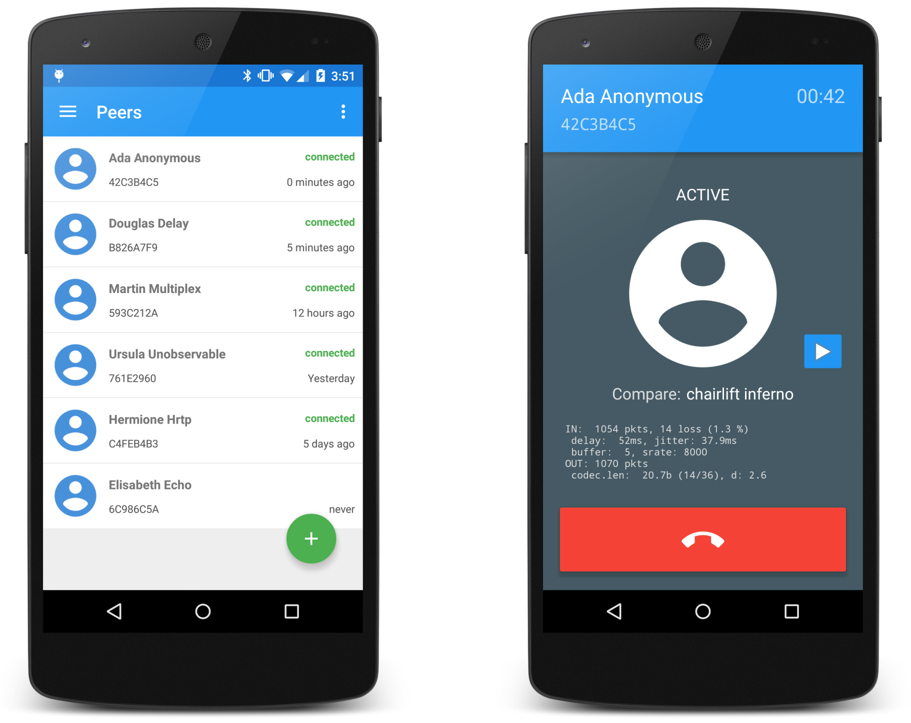
[4]When using certificate pinning, the client (here the Android device) can validate the certificate independently of the normal certification hierarchy. Often the certificates fingerprint or a hash of the public key is used. By this also self-signed certificates can be used responsibly.

User Interface Considerations

The user interface follows the Material Design pattern by Google. This allows the app to fit well into the existing Android environment. Also, it's usage patterns follow the best-practices established by popular VoIP applications. This allows to use the app for user studies in the field of secure VoIP.

When the app is first started, the user is presented the login screen (appendix figure A.1, left). He can also create a new identifier for its gateway (appendix figure A.1, middle). After login, the list of his contacts is shown (figure 4.11, left). By clicking on one, the call is initiated. As soon as the state changes to `ACTIVE`, the short authentication string (SAS) is also shown besides more debug information (figure 4.11, right).

With the idea of being a prototype for future development, the app allows helpful access to internal information. For instance, the user can access statistics about his current gateway. Those include the currently consumed bandwidth for HRTP and HRTCP and the handle time for incoming HRTCP packets (appendix figure A.2, left). Furthermore, the `CallActivity` displays information about the VoIP pipeline such as length of codec frames and the level of the input buffer (figure A.2, right). Finally, the `PreferencesActivity` allows to change important settings without re-compiling. Examples are usage of SSL, the gateway, disabling adaptive playback and configuring the background service.



Ada Anonymous	connected
42C3B4C5	0 minutes ago
Douglas Delay	connected
B826A7F9	5 minutes ago
Martin Multiplex	connected
593C212A	12 hours ago
Ursula Unobservable	connected
761E2960	Yesterday
Hermione Hrtp	connected
C4FEB4B3	5 days ago
Elisabeth Echo	
6C986C5A	never

Ada Anonymous 00:42
42C3B4C5

Compare: chairlift inferno

```
IN:  1054 pkts, 14 loss (1.3 %)
 delay:  52ms, jitter: 37.9ms
 buffer:  5, srate: 8000
OUT: 1070 pkts
 codec.len:  20.7b (14/36), d: 2.6
```

Figure 4.11: Screenshots of the Android handset application. Left: The `PeerListFragment` showing all local contacts. Right: An active call within the `CallActivity`. More screenshots are shown in appendix A.3.2. Own graphic created using *Android Open Source Project*'s (AOSP) Device Art Generator.

CHAPTER 5

Evaluation

“Program testing can be used to show the presence of bugs, but never to show their absence!”

— Edsger W. Dijkstra

We created an automatic test system that can deploy HRTP gateways on four servers in different locations. Those are then instrumented to perform calls resulting in large number of verifiable benchmark measurements. The results show that 90% of all calls are established in less than 11 seconds. Furthermore, the average HRTP round-trip-time of 50 ms (including network delay) confirms that the prototype is a very low-latency communication system. We also measured the packet jitter in different contexts and see that is does not unveil information about the internal session state.

A setup using a microphone and a sound generator is built for measuring the perceived voice delay. By analyzing the time of single sounds traveling through the system, an end-to-end delay of less than 400 ms is determined. This shows that the prototype is feasible for voice communication.

5.1 Method

The gateway was installed on a set of servers at multiple locations in order to incorporate real-world characteristics into the evaluation. In its biggest deployment it serves $n = 25$ users. The quantitative evaluation focuses on the metrics of latency, inter-packet characteristics and call establishment times.

		small: $n = 8$		large: $n = 25$	
Name	Location	LocalMembers	Peers	LocalMembers	Peers
`LDN`	London, GB	2	8	5	25
`FRA`	Frankfurt, DE	2	8	5	25
`TUM`	Garching, DE	2	8	9	25
`LRZ`	Garching, DE	2	8	6	25

Table 5.1: Overview of the test setup with respect to the two modes *small* and *large*. Each gateway also has own local members as its peers. More information on the machines can be found in A.2.1.

5.1.1 Test Setup

For testing the implementation, a setup consisting of four servers was deployed. The setup can be either run in mode *small* (8 users) or *large* (25 users). Deployment is handled by a local script that sets up the remote machines using SSH. Table 5.1 summarizes the setup.

All servers provide at least 2 GHz CPU and 512 MiB RAM. They run a GNU/Linux and the most recent version of Oracle's Java 7. For incorporating implications of a real-world scenario, the machines are located in Munich, Frankfurt and London. More detailed information is offered in the appendix A.2.1.

For automating testing a `mock_caller` and a `mock_callee` instrumentation script was written. The former can initiate a call to another user and the latter can accept calls from other users. Both interact with the API over HTTPS/JSON as a real handset would do. When the `mock_caller` has established a call, it will start a Java component that sends audio packets and measures their round-trip-time (RTT). When accepting an incoming call, the `mock_callee` will send all audio packets back to the gateway immediately. For real end-to-end testing a setup with real smartphones is used and described later.

5.1.2 Test Metrics

For the evaluation, we have to clarify the measured quantities. In figure 5.1 one sees that the total path of the audio from the sender's handset 1 to the receiver's handset 2 can be divided into several segments. The identical notation is used for the name of a segment and its latency:

$T_{HS1,encode}$ The time between the events of a voice sample entering the microphone and the sample being transmitted in an audio packet by the datagram socket of the sender's handset.

T_{Inner1} The network latency between handset 1 and its local gateway 1.

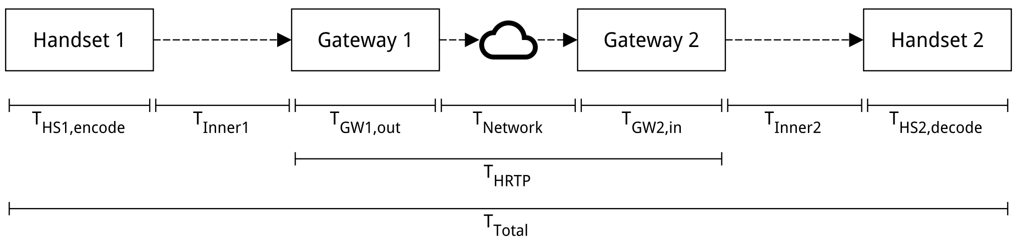


Figure 5.1: The total path from the sender's handset to the receiver's handset can be broken down into functional segments. Own graphic.

T_{GW1} The processing time within gateway 1, measured from the arrival of the audio packet to sending of the wrapping HRTP packet. This is strongly coupled to the HRTP rate of 30 ms, as the packet is first queued at the `RoundRobinPriorityPayload` datastructure where it gets picked up by the `DatagramBroadcaster`.

$T_{Network}$ The network latency between gateway 1 and gateway 2. We have measured an average $T_{Network}$ of 10 ms (= 20 ms RTT) for the setup when taking all combinations of gateways into account.

T_{GW2} The processing time within gateway 2, from arrival of the HRTP packet to sending the audio packet to handset 2. This time is negligible since the packet is directly forwarded at the receiver's gateway without buffering.

T_{Inner2} The network latency between gateway 2 and its handset 2.

$T_{HS2,decode}$ The time from arrival of the audio packet at the receiver's handset to the playback of the voice sample.

Composed Latency Metrics The aforementioned single segments are hard to measure individually. Furthermore, their individual meaningfulness is quite low, as many of them are dependent on each other. Therefore, we define the following latency metrics that are more suitable for the evaluation (see figure 5.1):

$T_{END2END}$ defines the total end-to-end (E2E) delay as the sum of all aforementioned segments. This is the most important measure as it describes the perceived delay in a voice communication.

$T_{HRTP} = 2 \cdot (T_{GW1} + T_{Network} + T_{GW2})$ is the minimum RTT delay, that the approach would add to a local-running VoIP solution.

$T_{VoIP} = T_{HS1,encode} + T_{HS2,decode}$ denotes the delay that is created by the VoIP pipeline.

Inter-Packet Metrics When analyzing the output of the gateway, we are interested in verifying that the packets are emitted at a constant rate. The expected inter-packet delay for HRTP is $\Delta = 30\,\text{ms}$. The jitter describes the deviation from the expected inter-packet delay. More formally, the jitter j_i for the i-th packet is defined as $j_i = (t_i - t_{i-1}) - \Delta$, where t_i denotes the arrival of the i-th packet.

Call Establishment Metrics For the end user it is important how fast a call can be established. The process of call establishment is divided into two main steps. First, $T_{Calling}$ denotes the waiting time for the user in the calling state given that the other party immediately responds to the incoming call. Second, $T_{Initiating}$ describes how long the ZRTP negotiation runs. The sum $T_{Establish} = T_{Calling} + T_{Initiating}$ denotes the time the user has to wait from the click of the call button until the actual voice communication happens.

5.2 Results

Manual testing confirms that HRTP provides a stable call establishment and good speech quality. The automated setup reveals that 90% of the calls have a call establishing time $T_{Establish} \leq 10118\,\text{ms}$ in the small setup. The is larger than the usual delay of other products, but can be improved by a higher HRTCP rate.

For both setups, the protocol round-trip-time T_{HRTP} meets the theoretical expectation of 50 ms (see section 5.2.3). The total perceivable voice delay T_{E2E} is less than 400 ms which is very suitable for VoIP communication. We illustrate the stream multiplexing in order to discuss its correct behavior. Finally, we verify that the jitter of outgoing packets shows no simple correlation with the internal call state. Important notation such as percentiles are explained in the appendix A.1.

5.2.1 General Results

Before discussing time related metrics in the following sub-sections, we have a look at general characteristics. First, we examine the packet sizes. The effective network consumption for each packet is the sum of its payload, the HRT(C)P overhead, the UDP header and the IP header. The UDP header has a fixed size of 8 byte. The IP header has a minimum size of 20 bytes and 40 bytes for IPv4 and IPv6, respectively. Figure 5.2 illustrates their relative sizes.

The rate of HRTP (30 ms) is much higher than the one for HRTCP (5 s). Therefore, the HRTP traffic (24 kib/s including IPv4 + UDP) clearly dominates the HRTCP traffic (0.86 kib/s). In figure 5.3 the total outgoing traffic of a single gateway with respect to a given number of peers is shown. One can see, that an average internet connection of ≈10.000 kib/s can serve several local members with up to 40 remote peers.

During manual testing further observations were made that are briefly explain here without detailed analysis: The average end-to-end packet loss is around 0.0-2.5%. The largest portion of loss is due to the drop of codec payloads larger than 44 byte. Furthermore, we could not observe any packet re-ordering during the test.

Professional analysis of the Opus codec quality in literature suggests a good performance even at the low bitrate of 6 kbit/s (discussed in 4.3.3). During practical testing, we confirmed that the speech quality is comparable to other internet telephony services[1] and that the end-to-end delay is sufficiently low for a smooth conversation.

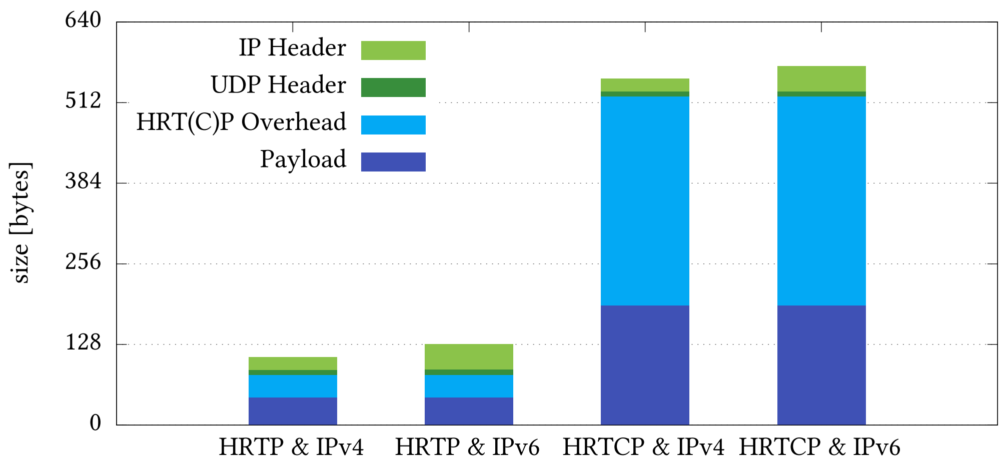


Figure 5.2: The diagram shows the effective size of the HRTP and HRTCP packets in IPv4 and IPv6 networks. Own graphic.

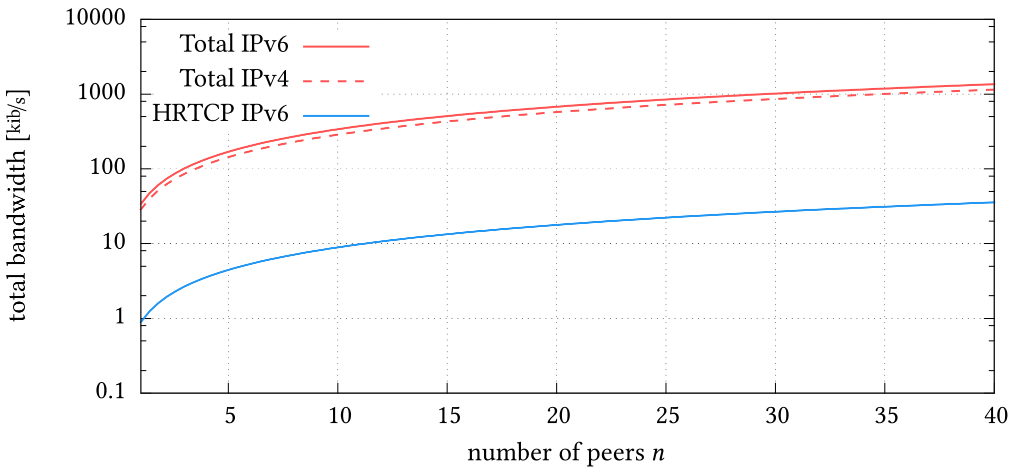


Figure 5.3: The total outgoing IPv6 traffic for a local member having a given number of peers. For comparison, the HRTCP traffic is shown separately. The total traffic for IPv4 is shown as a dashed line. Own graphic.

[1]For comparision Skype and Facebook's Messenger were used

5.2.2 Call Establishment Metrics $T_{Establish}$, $T_{Calling}$, $T_{Initiating}$

For $T_{Calling}$ we will first formulate the theoretical expectations using a simplified model where A calls B under ideal circumstances. Then we perform quantitative experiments using the small and large setup. Finally, we account for $T_{Initiating}$.

The calling procedure can be modeled using two random distributions. When we have set A's status to `CALLING`, we wait for the next HRTCP transmission cycle. The waiting time is modeled by the uniformly distributed random variable $X_1 = \mathcal{U}(0, 5000)$. We consider the time between receiving the packet from A at B and B's next HRTCP transmission. Again, this results in a uniformly distributed random variable $X_2 = \mathcal{U}(0, 5000)$. The total time is the sum of these two random variables. Applying convolution, we get a triangular distribution $f_{triang}(x)$ with the following probability density function (PDF):

$$
\begin{aligned}
f_{triang}(x) = f_{X_1+X_2}(x) &= (f_{X_1} * f_{X_2})(x) = \begin{cases} \frac{2x}{10000 \cdot 5000} & 0 \le x \le 5000 \\ \frac{2 \cdot (5000 - x)}{10000 \cdot 5000} & 5000 \le x \le 10000 \\ 0 & \text{otherwise} \end{cases} \\
\text{where } f_{X_1} = f_{X_2} &= \begin{cases} \frac{1}{5000} & x \in [0, 5000] \\ 0 & \text{otherwise} \end{cases}
\end{aligned}
$$

For the practical evaluation of $T_{Calling}$ we automatically performed $n > 200$ calls on both the small and the large setup. We used the `mock_caller` script for measuring how long the client was within the `CALLING` state. The results of outgoing calls from the server `LDN` were excluded as the server showed inconsistent behavior when sending large UDP packets due to reasons that we could not explain. The results are summarized in table 5.2 and illustrated in figure 5.4.

One can see that the small setup reproduces the theoretical expectations from above. The average length for establishing a call is just 1% larger than the average of f_{triang}. However, in the first large setup we see a shift of the average value by ≈ 5,000 ms. This means that on average, during one interval of calling, one HRTCP interval is missed due to long processing queues or the packet is dropped, respectively. The offset of almost exactly 10,000 ms for p_{90} indicates, that in 90% of all call establishing attempts no more than two HRTCP intervals are missed. In general, we consider an average $T_{Calling} \le 10,000$ suitable for most use cases.

We performed the same procedure for $T_{Initiating}$ which denotes the time in state `INITIATING` (see 5.2). The results are illustrated in figure 5.5. Here we see a very small deviation between individual calls and between the small and the large setup. The average value only increases from 2,408.6 ms (small) to 2,506.0 ms (large). This shows that the performance considerations of the implementation (such as early packet dropping) are indeed efficient. Due to the small variance, the results for $T_{Establish}$ are quite unspectacular. For the small setup, it takes on average 7,463.4 ms, while the average in the large setup is 12,537.6 ms (see 5.2).

	Setup	n	avg [ms]	p_{90} [ms]	p_{99} [ms]
$T_{Calling}$	Theory	∞	5,000.0	7,771.4	9,298.0
	Small	287	5,054.8	7,768.2	9,516.1
	Large	208	10,031.6	17,874.8	50,467.6
$T_{Initiating}$	Small	287	2,408.6	2,688.5	3,021.6
	Large	208	2,506.0	2,841.3	3,111.7
$T_{Establish}$	Small	287	7,463.4	10,118.7	11,913.2
	Large	208	12,537.6	20,328.7	53,301.4

Table 5.2: Table showing the results for $T_{Calling}$, $T_{Initiating}$ and $T_{Establish}$ from both the theoretical calculation and the practical evaluation.

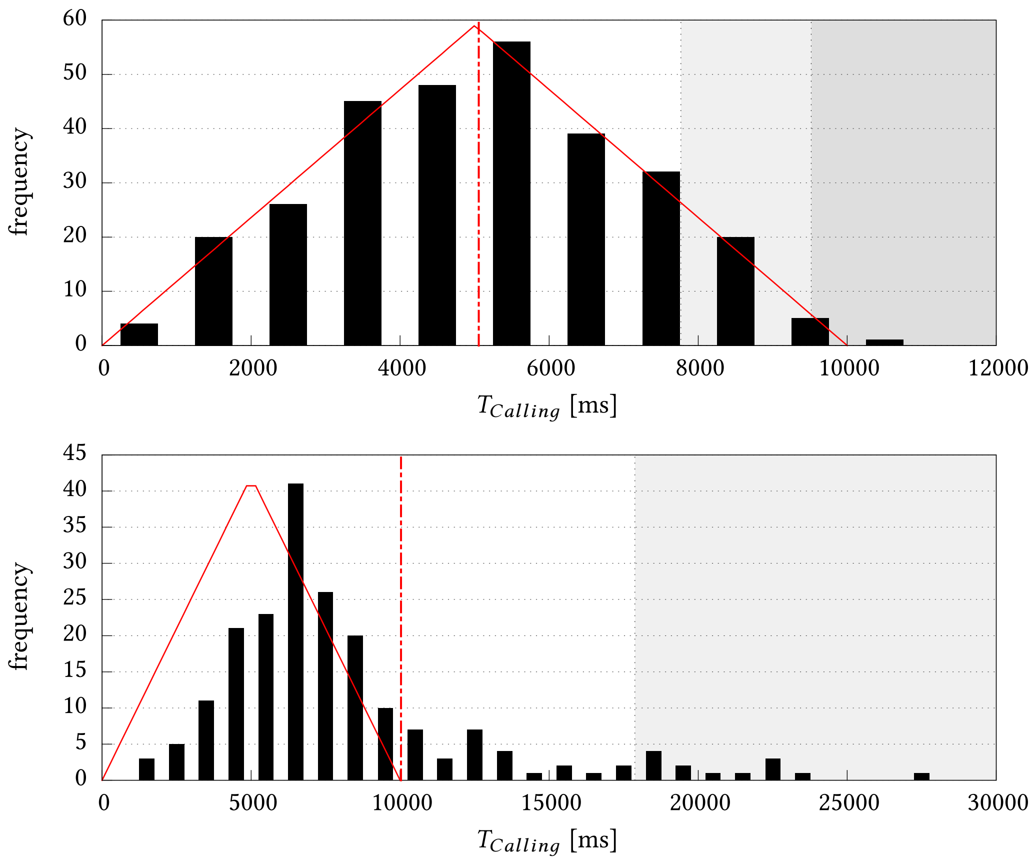
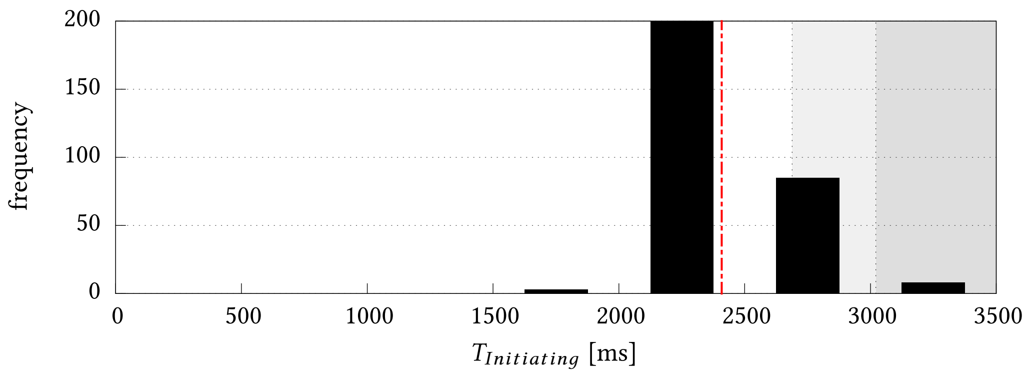


Figure 5.4: Histograms showing the distribution of $T_{Calling}$ for the small setup (top) and large setup (bottom). The red triangle shows the expectation from the calculations of T_{triang}. Compare table 5.2 for details. Own graphics.



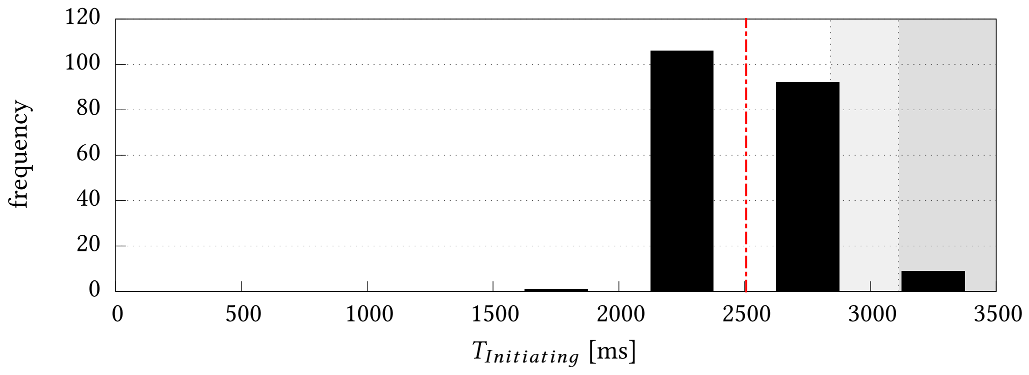


Figure 5.5: Histograms showing the distribution of $T_{Initiating}$ for the small setup (top) and large setup (bottom). Own graphics.

5.2.3 Latency of the HRTP Network T_{HRTP}

For the claim of HRTP being a very low-latency anonymous communication system, the delay T_{HRTP} is an important benchmark. It measures the efficiency of the payload multiplexing, the round-robin datastructure and the HRTP packet format. It is the absolute lower bound for T_{E2E} of every kind of every application running over HRTP.

We automatically performed $n > 200$ calls using the `mock_caller` script on both the small and the large setup. In order to take different link characteristics into account, the calls are evenly distributed among all combination of gateways for the caller and callee. In every call we sent approximately 90 audio packets in a round-trip fashion.

Again, one can model this using two uniform distributions $\mathcal{U}(0, 30)$. However, here the network delay of $\approx$ 10 ms per direction (or 20 ms RTT) can no longer be neglected. Since the resulting convolution is excessive for this purpose, we provide a qualitative analysis here. Intuitively, one can still see that the sum of both distributions distributions $\mathcal{U}(0, 30)$ results in a triangular distribution with a mean value of 30 ms. When adding an average network RTT offset of 20 ms, we obtain the theoretical expectation of T_{HRTP} =50 ms.

The results of both the small and the large setup show an average RTT that is very close to that expectation. This confirms once more, that the performance considerations of the implementation scale well. The relatively high p_{99} value of the large setup indicates outliers. Those are probably caused by the high bandwidth or buffer queues on lower layers.

	n = #calls	m = #packets	avg [ms]	p_{90} [ms]	p_{99} [ms]
Theory	∞	∞	50.00	—	—
Small	396	34,788	48.83	69.4	88.9
Large	280	23,679	50.01	73.8	123.2

Table 5.3: Results for T_{HRTP} for both the small and large setup.

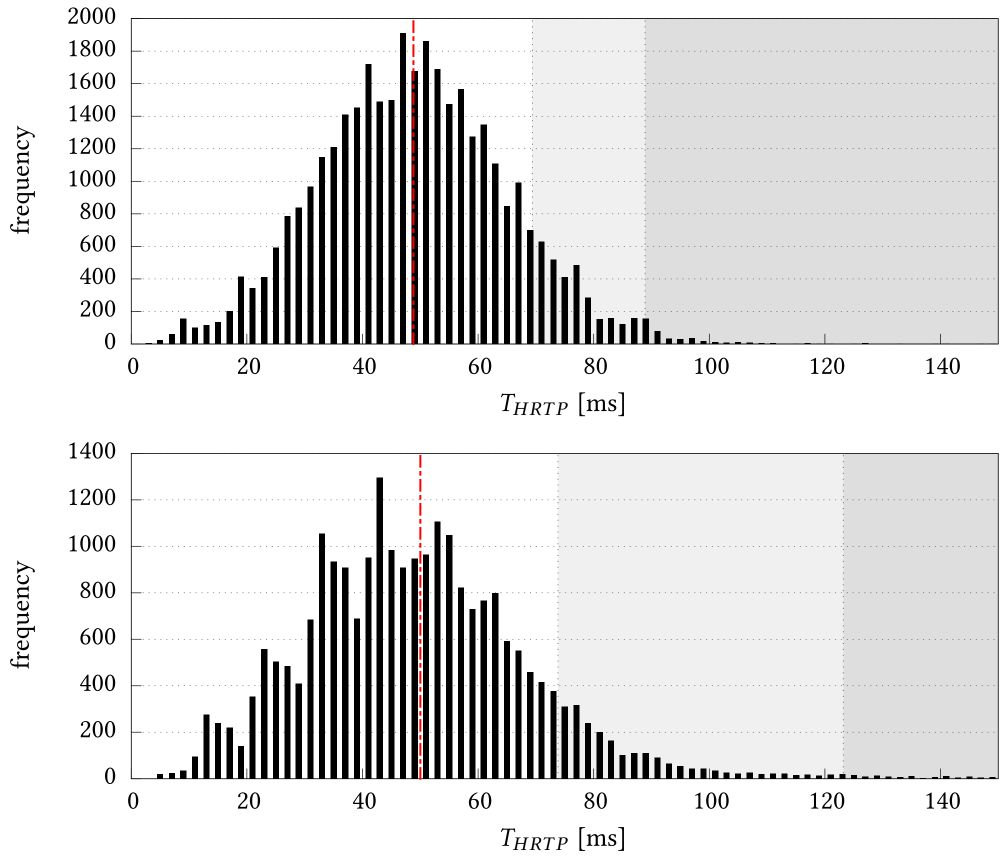


Figure 5.6: Histograms showing the distribution of T_{HRTP} for the small setup (top) and large setup (bottom). Own graphics.

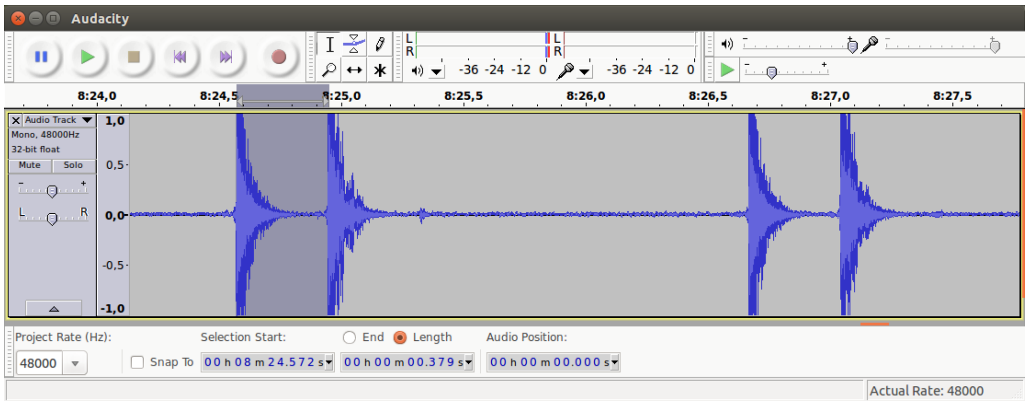


Figure 5.7: Screenshot showing four seconds of the recorded wave file containing two iterations opened in the audio-editing program *Audacity*. Each iteration consists of the initial "click" and the "echo". The first one has a measured delay of 379 ms. Own graphic.

5.2.4 Total Perceived Voice Delay T_{E2E} (and T_{VoIP})

The total perceived voice delay was measured in a local setup with two smartphones, a local gateway and audio equipment: Firstly, a microphone is positioned in a manner that it both records the second smartphone's loudspeaker and all ambient sounds. Secondly, a loudspeaker is installed which plays a "click" sound every $\approx$ 2 seconds. Finally, a voice call between both smartphones is established and the first smartphone's loudspeaker is muted.

Every iteration the microphone will first record the click sound coming directly from the loudspeaker. In the meanwhile, the click sound is also processed by the VoIP pipeline on the first smartphone, makes it way through the HRTP protocol including all gateway handling and then is played back by the second smartphone. The microphone then records that "echo" from the second smartphone's loudspeaker as well.

The difference between the click-event and the echo-event is the total perceived end-to-end delay T_{E2E} (minus $T_{Network}$[2]). Figure 5.7 shows an excerpt of the recorded wave file. In order to measure T_{VoIP}, the Android application has been changed to forward the audio packets to itself and not to the gateway. Of course, recording and playback now happens on a single phone.

For both versions 10 minutes were recorded and analyzed. Table 5.4 summarizes the results. They are visualized in figure 5.8. For analysis, the file is processed by a *Python* script that measures the distances between all pairs of click-events and their echo-events.

The most important observation is that the HRTP pipeline does not significantly increase

[2]The $T_{Network}$ is not included, as the HRTP packets are directly handled by the loopback network interface. Experiments showed that $T_{Network}$ is $\approx 20ms$. As it also has a every low influence on the HRTP payload jitter as shown in the next section, one can get a good approximation of the *real* T_{E2E} value by simply adding 20ms.

the average total perceived delay: $avg(T_{VoIP}) \approx^{\Delta<2\%} avg(T_{E2E})$. This is mostly due to the fact, that the input buffer's length of ≈120 ms dominates T_{HRTP}.

However, one can see that the standard deviation for T_{E2R} is ≈ 20 times higher than for the single setup without HRTP. This is caused by the jitter of the audio transport stream within HRTP. As the jitter also causes packets to arrive late, it can drain the buffer. The drainage of the buffer then results in the next packets played faster than intended (even faster than in the T_{VoIP} setup).

	n	avg [ms]	p_{90} [ms]	p_{99} [ms]	std [ms]
T_{VoIP}	286	373.15	375.0	380.0	45.0
T_{E2E}	287	378.92	408.0	446.0	897.0
$T_{E2E} + T_{Network}$	—	398.92	—	—	—

Table 5.4: Results from the measurement of the total perceived voice delay. For the last row, an estimate of 20 ms RTT for $T_{Network}$ was added.

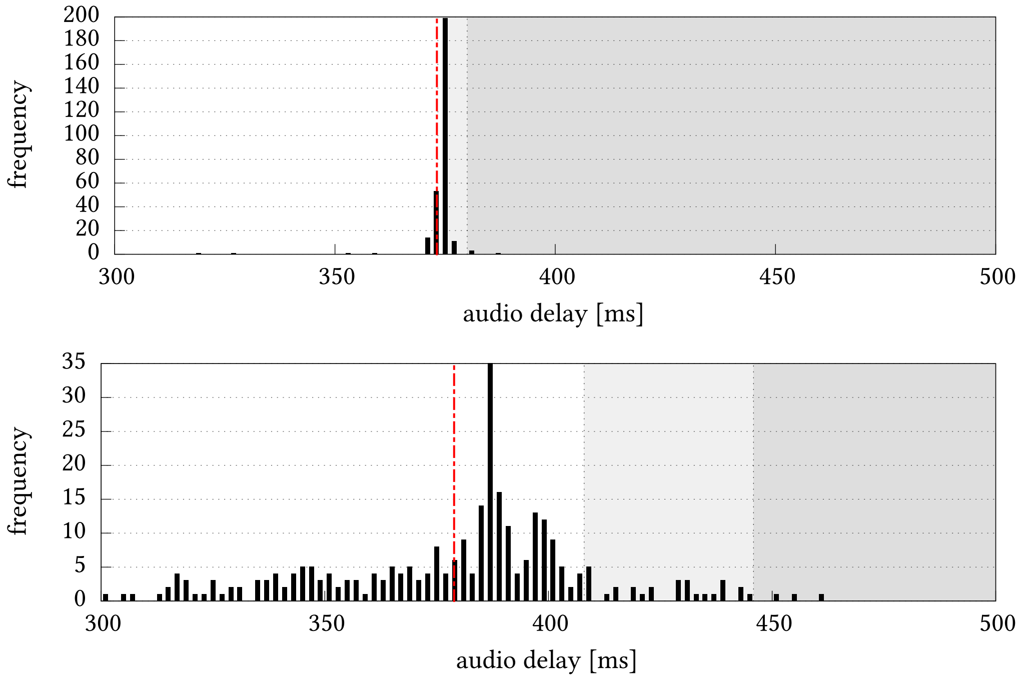


Figure 5.8: The graphic shows the total perceived voice delay. The top graph was created using a single smartphone without a gateway involved (T_{VoIP}). The bottom one shows T_{E2E} using a two smartphone setup with a local gateway. Own graphic.

5.2.5 Verifying Stream Multiplexing

For verification of the multiplexing procedure in HRTP, we analyze the queue behavior. To this end a local setup was modified to log the currently outgoing HRTP payload type and the length of the queues withing the `PayloadPriorityRoundRobin` data structure. The results for an outgoing call are shown in figure 5.10.

The diagram verifies several interesting properties. First, the dummy packets are created as desired and never reach a critically low buffer level. In fact, they never drop below a level of 9. Second, the priority of HRTP and audio payloads is obeyed. Whenever, there is an HRTP or audio packet, the dummy buffer grows to its maximum level of 10.

Furthermore, one can observe that the output throughput is in fact constant. When adding up the lines of the accumulated packet output, the resulting line will have linear slope of 1. Finally, it is important to note that the audio buffer (blue) almost never queues up to a level higher than 2. This shows that the gateway can handle the incoming audio payloads in a timely manner.

5.2.6 Outgoing Packet Jitter

For verifying that the HRTP output is decoupled from the internal state, we captured and analyzed outgoing HRTP packets. They were analyzed with respect to their jitter. Jitter denotes the deviation in inter-packet delay.

The capturing was performed on `TUM` (compare the setup section 5.1.1). We observed two scenarios: (A) No call was established and (B) a call to `LRZ` was established. For each of the scenarios, we looked at two metrics: (1) The jitter of packets sent by `TUM` and (2) the jitter of packets sent by `LRZ`. For capturing the command-line tool *tshark* (part of the *wireshark* package) were used. Packets going from `TUM` to `LRZ` have been captured using:

```
tshark -i any  -c 10000
  -f "dst host 141.40.254.58 && udp && src port 22001 && dst port 22002"
```

Vice versa, the opposite direction was captured (on `TUM` as well) using:

```
tshark -i any -c 10000
  -f "src host 141.40.254.58 && udp && src port 22002 && dst port 22001"
```

Each series captured 10,000 packets. The results are shown in table 5.5 and figure 5.9. The jitter of the locally sent packets is less than 0.02 ms for p_{90}. Compared to the sending rate of 30 ms, this is less than 0.07%. The jitter of the incoming packets is much wider due to the network transport in between.

We also note that there is no significant difference between the jitter during a call and while there is no active call.. However, this is only one possible metric and does not prove that the decoupling in the gateway is solid. More approaches for verification are discussed in the chapter 7 on further work.

			avg [ms]	p_{90} [μs]	p_{99} [μs]
A	1	TUM_{out}	0.0072	20.00	78.06
B	1	$TUM_{out,call}$	−0.0025	21.00	56.03
A	2	TUM_{in}	−0.0290	115.00	250.00
B	2	$TUM_{in,call}$	−0.0150	126.00	243.00

Table 5.5: Experiment results for all combinations of scenarios and metrics. The regular inter-packet delay is 30 ms. Note that different units are used: $1s = 10^3\,ms = 10^6\mu s$

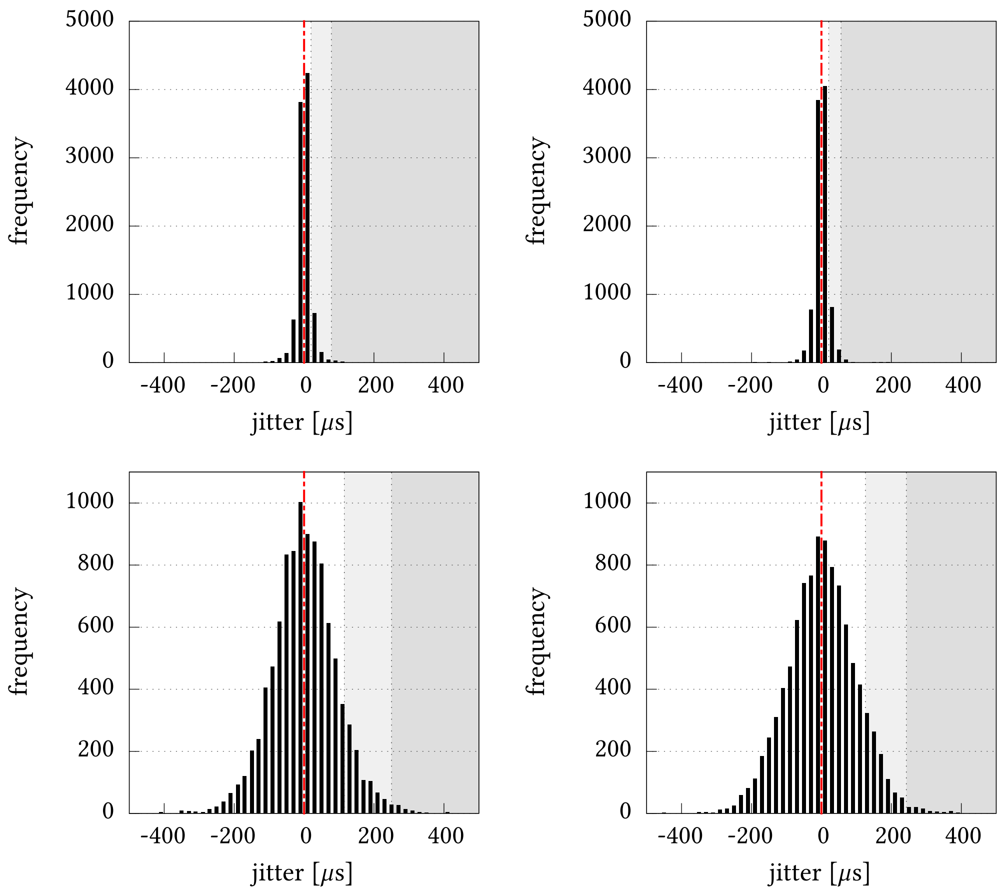


Figure 5.9: Histograms showing the jitter for all combinations of scenarios and metrics. Own graphic.
Left=A: no call; Right=B: call;
Top=1: outgoing packets; Bottom=2: incoming packets.

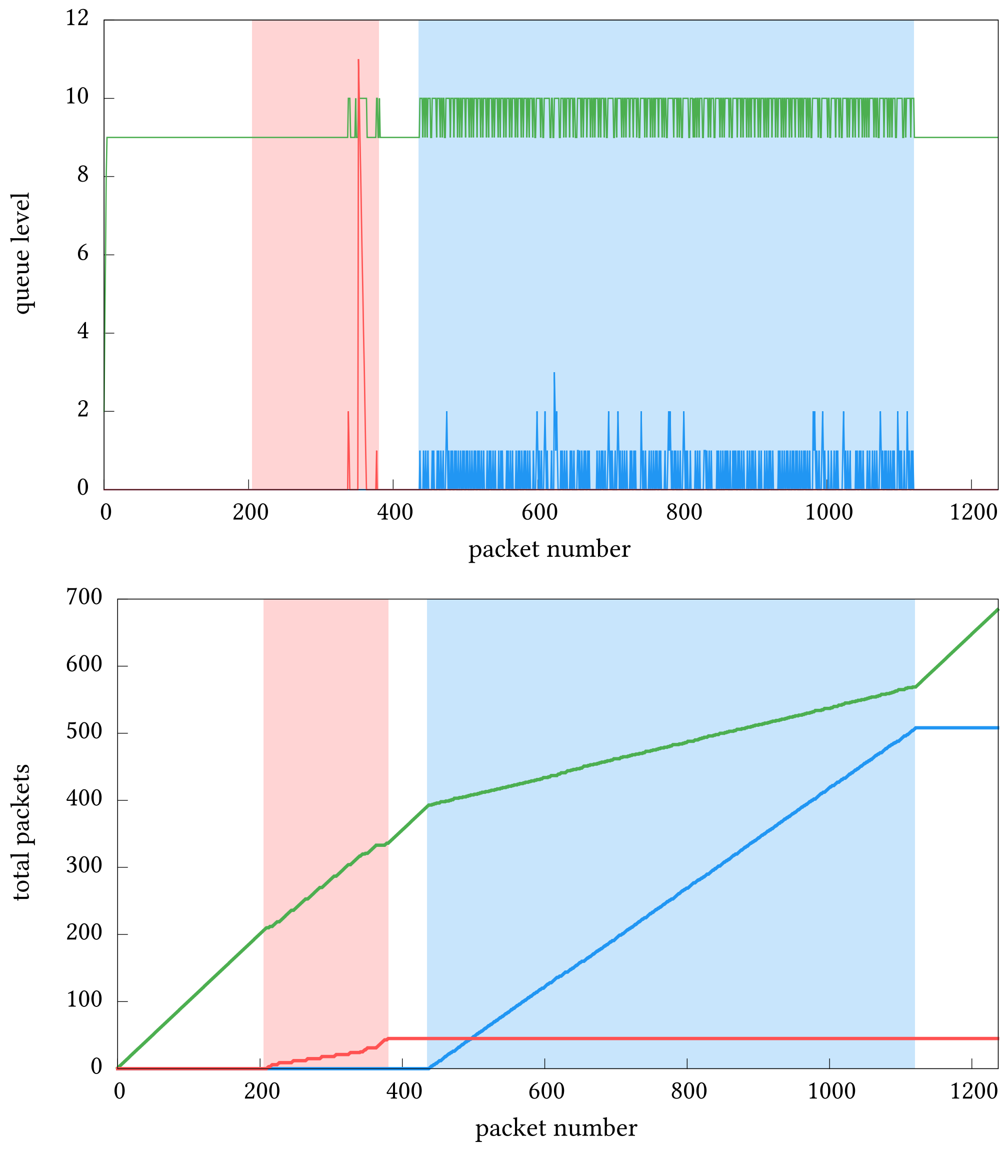


Figure 5.10: The upper plot shows the queue size after the last packet has been removed. The bottom plot shows the cumulative number of packets for the given payload type. The ZRTP negotiation (status=`INITIATING`) and the `ACTIVE` call phase are denoted by red and blue shadings, respectively. Line colors: green=dummy, blue=audio, red=zrtp. Own graphic.

CHAPTER 6

Quo Vadis: Transformation into a Scaling System

"Divide et impera" — LATIN PROVERB

This chapter presents a modification of the proposed approach that allows for HRTP to scale more efficiently to a large number of users. The basic idea is to create single islands of $m(n) = \log_2 n$ users. This concept borrows from ISDN mixes (described in 2.2.3) which employs small broadcast groups. . Besides the intra-island connections, every peer is also connected to remote peers of other islands. Those connections are used to relay traffic from one island to another one. Each user can control on which islands his traffic is present without an attacker noticing. Consequently, the approach becomes much more efficient as all users are no longer pair-wise connected.

The first part describes the feasibility of this approach using concepts from the field of peer-to-peer systems such as onion-encrypted relaying and proof-of-work verification. Furthermore, it argues how the system still provides strong unobservability characteristics. The second part describes the choice of $m(n)$ and discusses a theoretical analysis of the total bandwidth required.

6.1 Idea and Approach

For this chapter's approach, the n users are split into groups[1] of $m(n) \approx \log_2 n$ members. We call those groups broadcast islands as they work similar to the "original"[2] approach

[1]The choice of this concrete function is motivated in section 6.2 below.
[2]We refer to the original approach presented in chapter 3, 4 and 5 as "original" or "old".

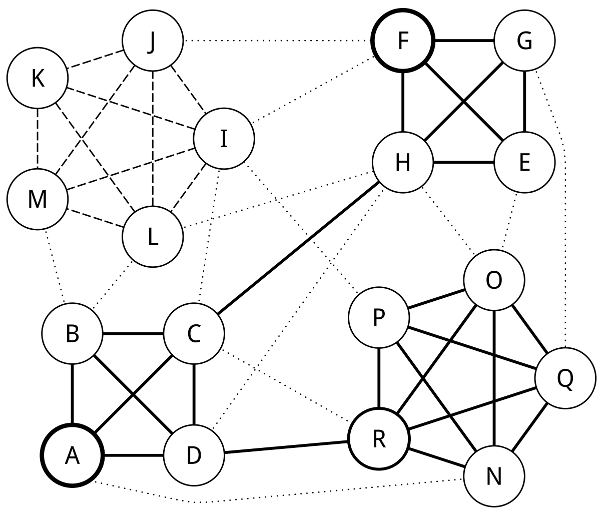


Figure 6.1: Sample graph with $n = 18$ forming 4 islands. A is calling F. All solid lines mark links that contain A's outgoing HRTP and HRTCP broadcast traffic. A has established the relay links C–H and D–R. Own Graphic.

but with a much smaller set of users. Within the islands, every peer establishes bidirectional broadcasting with each other local peer. We refer to this as *intra-island communication*. Figure 6.1 illustrates this using an example.

Connections Between Islands Every peer has an average of p connections to peers not belonging to his local island. We call this *inter-island communication* and the properties of p are described in 6.2.2. When a user tries to establish a connection to another island, he tries all peers of that island until he has found one having free capacity.

Inter-Island Relaying Consider we are trying to talk to a peer on another island. We will first ask a random peer on our local island to forward all of our traffic to another island. The local peer will establish that connection to a random peer on the remote island. We then agree with that remote peer that he will inject all of our HRTP and HRTCP traffic into his local intra-island broadcasting. Withing this relay chain, traffic is protected using onion-routing encryption. The negotiation and initial key exchange with the individual peers in handled using the HRTCP stream.

Parallel and Ephemeral Relaying Each user is encouraged to establish a reasonable amount of parallel relay routes. This increases the anonymity set of the island. Furthermore, each user changes them in a regular manner (e.g. once per minute) and not only for call establishment. This prevents an attacker from observing exploitable information from inter-island communication.

Preservation of Unobservability Due to the number of parallel relay routes, an attacker has to control all $m - 1$ local peers to find out about the set of islands the victim is broadcasting to. After this, he still does not know whether actual communication is taking place. However, over a long period of time, he can start an intersection attack. If that is successful, this can provide the observation that, with a certain probability, the victim is talking with a remote peer on a specific island. The attacker then needs to infiltrate all $m - 1$ peers of that remote island.

Proof-Of-Work for Limiting Relaying Proof-of-work (PoW) is a concept from the field of peer-to-peer systems and is used to limit the users consumption of shared resources (here: relay connections). A PoW is a cryptographic puzzles that verifies that a user possesses actual computational resources. This makes execution of sibling attacks[3] harder. Such a sibling attack could exhaust the relay capabilities of all local peers except those of the attacker. The victim would then have to pick one that is controlled by the attacker.

The cryptographic challenges are created by all relay nodes and are refreshed periodically. They are announced using the HRTCP stream. Creating a hash with leading zero-bits is a common example for a PoW: Here, the verifier announces a string C and a number b. For solving the puzzle the supplicants have to find a string x s.t. $\mathcal{H}(x \| C)$ has b leading zero-bits. This is computationally intensive for the client, as he has to try $\Theta(2^b)$ random x on average. On the other hand, the verifier can cheaply verify the correctness of the solution by computing the hash.

Increase of Latency The relay tunnels will add additional latency to the voice transport. In fact, the intermediate nodes will add $2 \cdot T_{Gateway}$ for processing and $2 \cdot T_{Network}$ for the extra links. A sophisticated implementation would take the geographic neighborhood into account when assigning the island neighborhoods. This can reduce the effect of the $T_{Network}$. The $T_{Gateway}$ latency can be reduced by increasing the sending rate for HRTP. This appears feasible, as we show that the bandwidth of an individual peer is much lower in the approach proposed in this chapter.

Churn The term *churn* describes the entering and exiting of users in a peer-to-peer system. For the approach we expect a very low churn rate from legitimate users, as the unobservability guarantees are built upon the continuous broadcasting. A small churn rate can be enforced by requiring a fresh solution for a strong PoW challenge before being able to join an island.

[3]In a sibling attack, the attacker incorporates many users at the same time [38]. This increases the ratio of attacker controlled nodes to legitimate nodes. As an attacker does not have unlimited computational resources, he it is hard for him to perform PoWs for all of its fake identities.

6.2 Parameter Design and Analysis

We first make a choice for the size and number of the islands as functions of the total number of users n. We take care that the islands can easily split up when they become too large. Later, we show that the total bandwidth – and especially the bandwidth for individual peers – scales nicely with increasing n.

6.2.1 The Number of Islands $i(n)$ and Their Size $m(n)$

For the design we decide that the average size of a broadcast island is $m(n) = \log_2 n$, where n is the total of users. This choice appears to be arbitrary at this point. We justify the choice with the results shown below: It is the best performing choice of all simple candidate functions that we have considered. However, a practical application will require further fine-tuning. One can see that the number of islands $i(n)$ depends on both n and the choice of $m(n)$:

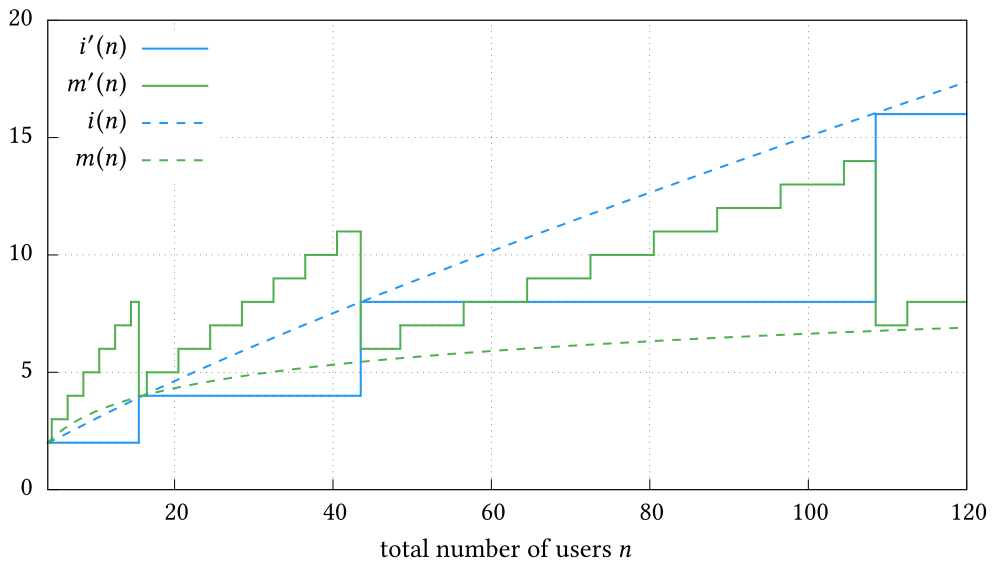
$$\begin{aligned} m(n) &= \log_2 n \\ n = i(n) \cdot m(n) \;\Leftrightarrow\; i(n) &= \frac{n}{m(n)} = \frac{n}{\log_2 n} \end{aligned}$$

Obviously, the number of islands must be integral. We discretize $i(n)$ and call the result $i'(n)$. For $i'(n)$ we would also like to have the following property: For every change of its value, it doubles the number of islands. This simplifies the redistribution of too large islands. With the aforementioned property, every island is split into exactly two new ones, whenever the island count is increased.

This is done by mapping the function values to their next smaller power of two using $d(x)$. We formulate $i'(n)$ using this new function. We also define $m'(n)$ as the size of the largest island given that new users are distributed equally amongst all islands:

$$\begin{aligned} d(x) &= 2^{\lfloor \log_2(x) \rfloor}, x > 0 \\ i'(n) &= d(i(n)) = 2^{\left\lfloor \log_2\left(\frac{n}{\log_2 n}\right)\right\rfloor} \\ m'(n) &= \left\lceil \frac{n}{i'(n)} \right\rceil \end{aligned}$$

The following plot[4] shows the behavior of $m'(n)$ and $i'(n)$. For comparison, also the original $m(n)$ and $i(n)$ are shown as dotted lines:



6.2.2 Bandwidth Scaling Properties

For analyzing the bandwidth, we use the original functions $m(n)$ and $i(n)$ instead of their discrete siblings. We choose the number of HRTP and HRTCP connections as the main metric instead of concrete kbit/s. However, both metrics are proportional. We first analyze the intra-island bandwidth $bw_m(n)$ and then discuss the inter-island bandwidth $bw_i(n)$. Both add up to the total bandwidth $bw_{total}(n)$:

$$bw_{total}(n) = bw_m(n) + bw_i(n)$$

Before we analyze the new approach, we recall that for the "old" approach, each user is connected to all other peers except himself: $n - 1$. This results in a quadratic growth of the total bandwidth $bw_{old} = O(n^2)$ in big O notation[5]. The bandwidth for each user grows linearly with its number of peers:

$$\begin{aligned} bw_{old}(n) &= n \cdot (n-1) = O(n^2) \\ bw_{peer,old}(n) &= n - 1 = O(n) \end{aligned}$$

[4]For all plots: We ignore the corner cases of $n < 4$, as the logarithm in the denominator leads to meaningless numbers. One can assume that we only have one island with $m(n) = n$, for $n < 4$.

[5]This thesis uses the common convention of writing $f = O(g)$ instead of $f \in O(g)$.

The intra-island bandwidth $bw_m(n)$ is similar to this, as it reassembles the old approach in a small setting. Each user is connected to each of the other peers on his island. The extra traffic that is forwarded from external users is accounted for with a factor of 2:

$$bw_m(n) = 2 \cdot i(n) \cdot m(n) \cdot (m(n) - 1) \leq 2 \cdot i(n) \cdot m(n)^2 = 2 \cdot n \ \log_2 n$$

Before analyzing the inter-island traffic, we need to define the parallel factor p. The parallel factor denotes the number of outgoing inter-island connections for each user. We think it is a reasonable choice to pick p such that each island has on average two connections to each other island:

$$p(n) = 2\frac{i(n) - 1}{m(n)} \leq 2 \ \frac{i(n)}{m(n)} = 2 \ \frac{\frac{n}{\log_2 n}}{\log_2 n} = 2 \ \frac{n}{\log_2^2 n}; \quad p(100) \approx 4.53$$

As every node has $p(n)$ outgoing inter-island connections on average, the total inter-island bandwidth $bw_i(n)$ is composed as follows:

$$bw_i(n) \quad = \quad n \cdot (n) \leq 2 \ \frac{n^2}{log_2^2 n}$$

The total bandwidth $bw_{total}(n)$ grows sub-quadraticly in the number of users:

$$bw_{total}(n) = bw_m(n) + bw_i(n) \leq 2n \ \log_2 n + 2 \ \frac{n^2}{\log_2^2 n} = O(\frac{n^2}{\log^2 n})$$

For correctness of the last step: First the term $\frac{n}{\log^2 n}$ is factored out. One can show that $\forall n > 1 : \frac{n}{\log^2 n} > 0$. The fact that linear functions grow asymptotically faster[6] than poly-logarithmic functions ($\log^c n$) then leads to the final estimation:

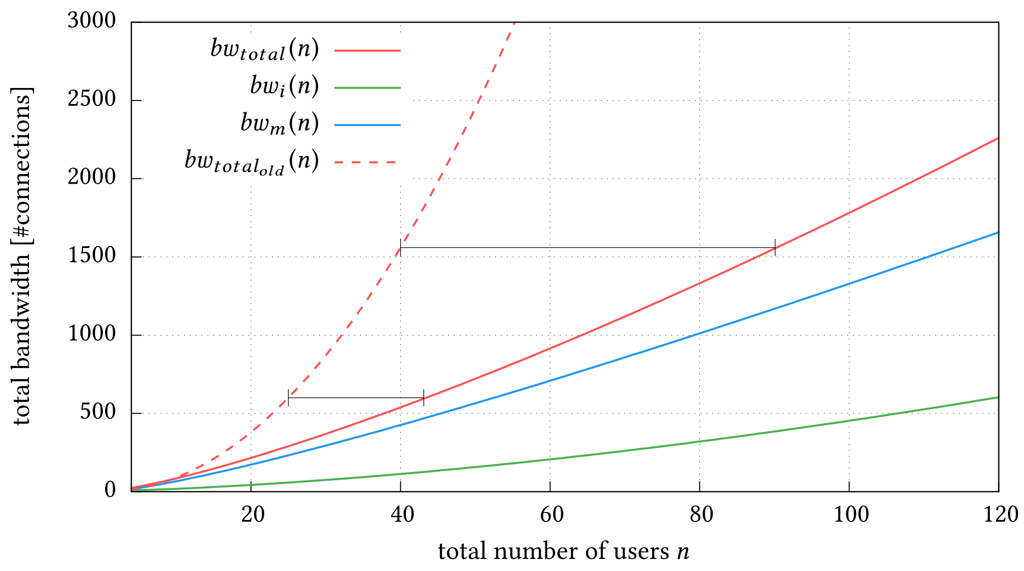
$$\forall c > 1 : \ \log^c n = O(n) \quad \Rightarrow \quad n \log n = \frac{n}{\log^2 n} \ \cdot \ \log^3 n = O(\frac{n}{\log^2 n} \ \cdot \ n) = O(\frac{n^2}{\log^2 n})$$

Note that the base b of the logarithm is not significant for the big O notation, since it basically reduces to a constant factor:

$$\forall b > 0, \ n > 0 : \ \log_b x = \frac{\log n}{\log b} = \frac{1}{\log b} \cdot \log n = \Theta(\log n).$$

[6]A proof for $\forall a > 0, \ c > 0 : \ \log^c n = o(n^a)$ can be found in "Introduction to Algorithms" by Cormen et al. [39, p.56f].

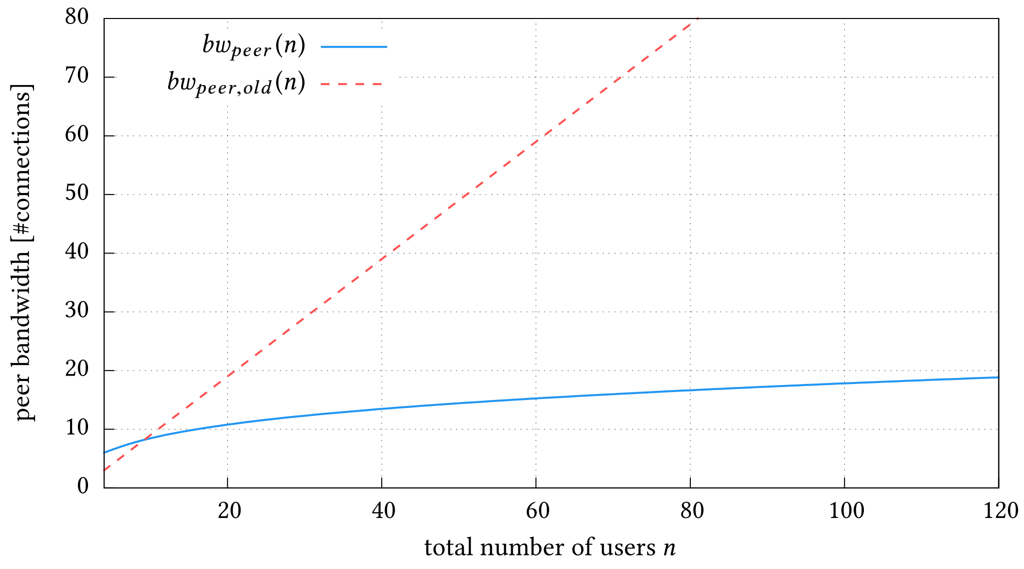
The individual portions of inter- and intra-island bandwidth as well as the total bandwidth are displayed in the plot below. For comparison, the bandwidth of the old approach is shown as a dashed red line:



The lower arrow shows that instead of 25 users the new approach can serve 42 with the same amount of total traffic (upper arrow: 40 → 90, respectively).

The difference to the average per peer bandwidth bw_{peer} (blue) defined as $bw_{peer}(n) = bw_{total}(n)/n$ is even more significant:

$$bw_{peer}(n) = \frac{bw_{total}(n)}{n} \leq 2\log_2 n + 2\,\frac{n}{\log_2^2 n} = O(\frac{n}{\log^2 n})$$



CHAPTER 7

Further Work & Conclusion

"Science is what we understand well enough to explain to a computer. Art is everything else we do."
— Donald E. Knuth

This thesis points out further work in the fields of verification of unobservability, extensions based on the prototype and implications of design simplicity for end-user acceptance. We conclude that the HRTP approach stands out from the state of the art due to providing both strong unobservability and very low-latency communication. A variable combination of broadcast traffic and onion-routing appears to be a promising direction for further research.

7.1 Further Work

Verification of unobservability is still an open topic and we think that one can build a widely applicable approach using formal logic or machine learning concepts. We will also discuss how the current HRTP prototype can be extended. The most interesting dimension is probably to create a solution with an adjustable combination of broadcast traffic and onion-routing elements. Furthermore, we believe it is worth to evaluate how simplicity of protocols not only serves implementation quality but also increases end-user acceptance.

7.1.1 Verification of Unobservability

Verification of unobservability is still a vague undertaking. We are missing a widely accepted and generally applicable method for analyzing unobservability properties of systems – or we have failed to find such. We think that such a method can be developed on the foundation of the terminology of Pfitzmann et al. [2] that we used throughout this thesis. We identified two promising approaches:

Firstly, using formal logic for verification of unobservability would be a natural and broadly accepted approach. Formal logic has already been used to verify anonymity systems. The modular framework by Hughes et al. [40] as well as the work by Van Eijck et al. [41] could form a foundation for this. We think that they need to be extended by observation properties with regard to implementation specifics such as packet emission.

Secondly, a more practically oriented, quantitative approach would be the application of machine learning (ML). Here the existence of a working classifier would show that the analyzed protocol has observable characteristics. Labeled test sets can be generated by a setup similar to the evaluation. ML is already used for intrusion detection of networks and the review by Tsai et al. [42] provides an introduction to this field.

7.1.2 Quo Vadis HRTP

Chapter 6 has presented an extension to HRTP that scales efficiently. However, the implementation and evaluation of such a broadcast-island system is out of the scope of this theses. Nevertheless, it would be tempting to build such a system.

An important observation is that the proposed extension combines broad-cast techniques with onion-routing. We think that it can lead to a solution where the individual user can choose his own mixture of both. He could control the ratio of cover traffic, broadcasting and onion-routing for his traffic. This would allow to adaptively choose the right balance of delay, bandwidth and privacy for each application.

Furthermore, the primary protocol and its implementation require further fine-tuning. We think that the consumed bandwidth and latency can be improved by optimizing parameters such as the HRTP rate. Also other payload types such as text messages could be introduced. Using GNU Privacy Guard (GPG) instead of the proprietary key management would greatly simplify deployment.

Currently, the need of unobservable private networks between the handset and the gateway limits the mobility of the end-user. In order to overcome this, we see two central starting points: Firstly, one finds a way to create cover traffic from a handset or smartphone without draining the mobile device's battery too fast. Such a low-energy cover traffic would also be interesting for other applications, e.g. sensor networks.

Secondly, one could design a way for the user to move between trusted gateways. When visiting another private network, the user brings his private key and identity on a secure element (SE)[1] that he plugs into a gateway. As all traffic is sent to every gateway anyway, this migration can be performed in a manner such that it does not lead to any observable changes.

7.1.3 Understandable Anonymity Systems

We think that the actual security of communication systems also depends on the end-user's understanding of its concept, its capabilities and its short comings. In this thesis, we have proposed an approach that is much simpler to explain than other services such as Tor.

Our first hypothesis is that understandable concepts leads to a higher acceptance: A user understanding why the call establishment takes longer than a regular land-line call, is more likely to accept this short coming. Moreover, we suspect that the achieved security is increased: By knowing the necessity of the gateway running without interruption, the end-user won't act with negligence by turning it off after each call.

7.2 Conclusion

In this thesis we have discussed the design, implementation and evaluation of a prototype for unobservable Internet telephony using a broadcast-based approach named HRTP. We have shown that it in fact provides unobservability guarantees against strong attackers. A practical deployment and quantitative evaluation has revealed that such a broadcast-based approach is indeed feasible. The prototype can serve as a foundation for further practically oriented research in the field of secure VoIP. The source code is made publicly available with this thesis and *you* are encouraged to hack.

Compared to the state of the art, HRTP stands out with regard to two aspect: It provides strong unobservability guarantees against global active attackers, while other popular implementations limit their attack vector to partial active attackers. Secondly, it provides very low-latency communication suitable for VoIP. However, this comes at the cost of a total traffic that is quadratically growing with the number of users.

Finally, we have described how the approach can be adopted to provide a scaling unobservable communication system. The brief analysis of the broadcast island concept shows that it drastically reduces the required per peer bandwidth to $O(\frac{n}{log^2 n})$. A variable combination of broadcasting and onion-routing forms an interesting starting point for further research.

[1]A secure element is a tamper-resistant chip that allows to securely store information such as private keys. It can also perform cryptographic operations on-chip without the sensitive information leaving the trusted environment. Examples for secure elements include credit cards, subscriber identity modules (SIMs) or trusted platform modules (TPMs).

APPENDIX A

Appendix

A.1 Notation and Conventions

We use the following notation for writing packets, strings and cryptographic operations. For conventions regarding the HRTP system, see section 3.2.1.

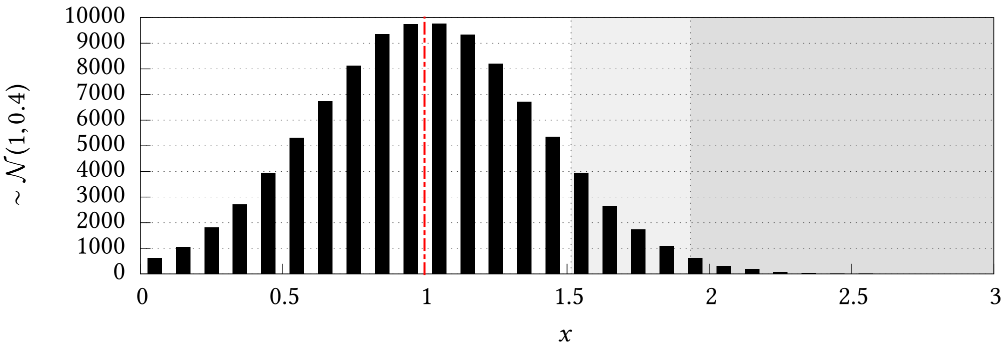
Bits, Bytes, Packets and Strings: Bits are abbreviate as *b* and bytes as *B*. This thesis sets that 1 B has 8 b. For larger quantities the IEC prefixes Ki (= 2^{10}), Mi (= 2^{20}), Gi (= 2^{30}) and Ti (= 2^{40}) are used. The concatenation of strings is written using the vertical bar operator: $s_{12} = s_1 \| s_2$. Sub-strings are denoted using the *Python* index operator: `"HELLO"[1 : 3] = "EL"`

When defining messages and packets in protocols, we often do not care about the exact representation, but want to show the individual information stored. This is expressed using the following notation: $msg_{hello} = \{\texttt{"HELLO"}, A, B\}$.

Cryptographic Operations $\mathcal{E}, \mathcal{D}, \mathcal{H}, \mathcal{K}, \mathcal{V}, \mathcal{S}$: The paper will use cryptographic operations such as (a)symmetric encryption, decryption, hash functions, signatures and key derivations. All operations will take the key as the first argument and more input as further arguments. Initialization vectors (IV) are usually omitted. Where it is helpful, the operations use the name of the implementation of the cryptographic operation such as *aes* or sha_{256}.

(A)symmetric en-/decryption of a plain-text p, with a key k is written as $enc(k,p)$ or $\mathcal{E}(k,p)$ or $\mathcal{D}(k,c)$ or $aes(k,c)$ or $rsa(pub,p)$. For a cryptographic hash function or a hash-based message authentication code (MAC) we write $hash(k,\dots)$ or $\mathcal{H}(k,\dots)$ or $sha_{256}(k,\dots)$. Key derivation functions are denoted using $kdv(k,\dots)$ or $\mathcal{K}(k,\dots)$. Signatures are written as $sign(pub,text)$ or $verify(priv,signature)$ or $\mathcal{S}(pub,text)$ or $\mathcal{V}(priv,signature)$.

Results and Plots: For most results the 90th and 99th percentile are given and referenced as p_{90} and p_{99}, respectively. Histograms (example below) show both percentiles using a very light gray and a medium light gray. The notation $avg(\dots)$ is used for the average value. It is displayed by a dotted red line. std is the standard deviation.



A.2 Evaluation Details

A.2.1 Configuration of Servers

HrtpLondon (LDN)

- Virtual Server hosted by *DigitalOcean* in London, UK
- 1x Intel Xeon CPU 2.4 GHz; 512 MiB
- Ubuntu 14.04 LTS; Oracle Java 7

HrtpFrankfurt (FRA)

- Virtual Server hosted by *DigitalOcean* in Frankfurt, GER
- 1x Intel Xeon CPU 2.4 GHz; 512 MiB; Oracle Java 7
- Ubuntu 14.04 LTS; Oracle Java 7

HrtpTum (TUM)

- Virtual Server hosted by *Chair for Network Architectures and Services* in Garching[1], GER
- 2x Intel CPU 3.2 GHz; 512 MiB; Oracle Java 7
- Debian 3.16.7; Oracle Java 7

HrtpLrz (LRZ)

- Virtual Server hosted by the *Leibniz Rechenzentrum* (LRZ) in Garching, GER
- 1x CPU 2.5 GHz; 2010 MiB; Oracle Java 7
- Debian 3.16.7; Oracle Java 7

[1] *Garching bei München* is a (small) city north of Munich

A.3 Logs & Screenshots

A.3.1 Log of End-To-End Gateway Test

The following output shows a end-to-end unit test case. It is chosen here as it produces a very exemplary output of the gateway application. The case simulates the whole process from creating key-pairs, over establishing a call, performing ZRTP and sending messages. For this it uses just the normal HTTP APIs. The output has been reformatted for better readability:

```
------------------------------------------------------------
Step: setup

20:28:00.812 [main] TRACE - Created RSA keys of length 2048 in 2193 ms
20:28:02.791 [main] TRACE - Created RSA keys of length 2048 in 1959 ms
20:28:04.588 [main] TRACE - Created RSA keys of length 2048 in 1797 ms
20:28:06.113 [main] TRACE - Created RSA keys of length 2048 in 1524 ms
20:28:08.482 [main] TRACE - Created RSA keys of length 2048 in 2367 ms

------------------------------------------------------------
Step: startup

20:28:08.546 [MainControl STARTING] TRACE - Loaded configuration with 2 (2 new) local
     members and 3 (3 new) peers
20:28:08.557 [BridgeManager STARTING] TRACE - I've registered 11 ApiLookups
20:28:08.557 [BridgeManager STARTING] WARN  - No SSL certificate registered nor loaded!
     This is insecure!
20:28:08.657 [BridgeManager STARTING] DEBUG - Started bridge manager on port
     /0:0:0:0:0:0:0:0:8080 {}
20:28:09.273 [MainControl STARTING] INFO  - Up and running :) after 778 ms
20:28:09.289 [SocketManager STARTING] DEBUG - ports opened: hrtp=20001 hrtcp=30001
20:28:09.322 [SocketManager STARTING] DEBUG - ports opened: hrtp=20002 hrtcp=30002
20:28:09.361 [SessionWorker:A01B4A24 STARTING] TRACE - started
20:28:09.361 [SessionWorker:D22EA391 STARTING] TRACE - started

------------------------------------------------------------
Step: 1a: authorize A

------------------------------------------------------------
Step: 1b: authorize B

------------------------------------------------------------
Step: 2a: A calls B

20:28:09.284 [main] TRACE - Start calling A01B4A24 @ /127.0.0.1:20002/30002
20:28:09.462 [SessionWorker:D22EA391 RUNNING] TRACE - state: NOT_ACTIVE -> CALLING, peer
     =A01B4A24
20:28:12.970 [SessionWorker:A01B4A24 RUNNING] TRACE - state: NOT_ACTIVE -> RINGING, peer
     =D22EA391

------------------------------------------------------------
Step: 2b: check RINGING status on B's side
```

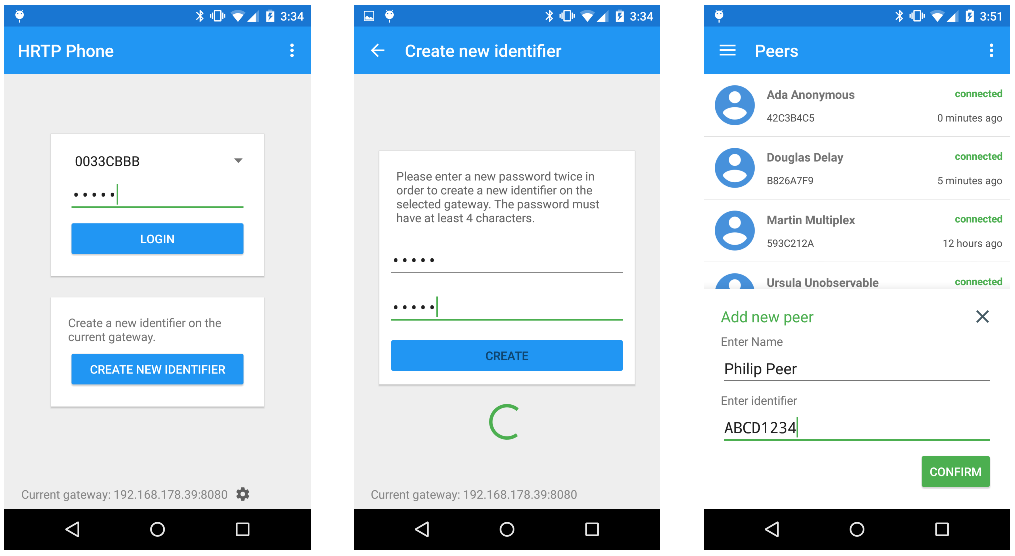
```
------------------------------------------------------------
Step: 3a: B accepts call

20:28:17.293 [main] TRACE - Accepting call
20:28:17.332 [main] DEBUG - [A01B4A24] [ZRTP] created new ZID= a6f2535d70a944d12386347d
20:28:17.334 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Thread Starting
20:28:17.340 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] DH algorithm set: <NA> -> DH3k
20:28:17.381 [SessionWorker:A01B4A24 RUNNING] TRACE - state: RINGING -> INITIATING, peer
     =D22EA391
20:28:17.384 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Sending HELLO...
20:28:17.536 [Timer-0] DEBUG - [A01B4A24] ZRTP: Sending HELLO...
20:28:17.837 [Timer-0] DEBUG - [A01B4A24] ZRTP: Sending HELLO...
20:28:18.261 [DatagramListener:D22EA391-hrtcp0] DEBUG - [D22EA391] [ZRTP] created new
     ZID= eec224f7c516dd05e0431e79
20:28:18.262 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Thread Starting
20:28:18.263 [ZRTP-ZRTP-1] DEBUG - [D22EA391] DH algorithm set: <NA> -> DH3k
20:28:18.279 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Sending HELLO...
20:28:18.283 [SessionWorker:D22EA391 RUNNING] TRACE - state: CALLING -> INITIATING, peer
     =A01B4A24
20:28:18.361 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Received Hello, word length 28
20:28:18.362 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: HELLO - FarEndClientID
     PWaveIPrivateGSM
20:28:18.364 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] DH algorithm set: DH3k -> DH3k
20:28:18.365 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTPCache: selectEntry(
     eec224f7c516dd05e0431e79)
20:28:18.366 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] DH algorithm set: DH3k -> DH3k
20:28:18.371 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] Writing public key for DH3K mode:
     144715746101581623712250303795625375149 [...]
20:28:18.379 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] Writing key bytes: 3
     fc4d80e01b3729d42a96520867544f614ce1d8d45637d6621fc46 [...]
20:28:18.383 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Sending HELLOACK...
20:28:18.390 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received HelloACK
20:28:18.438 [Timer-0] DEBUG - [A01B4A24] ZRTP: Sending HELLO...
20:28:18.510 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received Hello, word length 28
20:28:18.510 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: HELLO - FarEndClientID
     PWaveIPrivateGSM
20:28:18.511 [ZRTP-ZRTP-1] DEBUG - [D22EA391] DH algorithm set: DH3k -> DH3k
20:28:18.511 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTPCache: selectEntry(
     a6f2535d70a944d12386347d)
20:28:18.511 [ZRTP-ZRTP-1] DEBUG - [D22EA391] DH algorithm set: DH3k -> DH3k
20:28:18.513 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Writing public key for DH3K mode:
     5344216802091970408264038254962466603442 [...]
20:28:18.518 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Writing key bytes:
     eb7e1e32abab93668cbb3994ce104414c7af774db3a7592fe77b513 [...]
20:28:18.521 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Sending HELLOACK...
20:28:18.539 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Received HelloACK
20:28:18.539 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Sending COMMIT...
20:28:18.540 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] DH algorithm set: DH3k -> DH3k
20:28:18.629 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: COMMIT MESSAGES RECEIVED
20:28:18.630 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: hash type - S256
20:28:18.630 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: cipher type - AES1
20:28:18.630 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: auth type - HS80
20:28:18.630 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: key type - DH3k
20:28:18.630 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: sas type - B256
```

```
20:28:18.631 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received Commit, acting as responder
     , iState: ZRTP_STATE_GOT_HELLO_ACK
20:28:18.631 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Sending DHPart1...
20:28:18.631 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTPCache: selectEntry(
     a6f2535d70a944d12386347d)
20:28:18.632 [ZRTP-ZRTP-1] DEBUG - [D22EA391] DH algorithm set: DH3k -> DH3k
20:28:18.633 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Writing public key for DH3K mode:
     534421680209197046603442374991151030385 [...]
20:28:18.637 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Writing key bytes:
     eb7e1e32abab93668cbb3994ce104414c7af774db3a7592fe43237 [...]
20:28:18.990 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Processing DHPart1.....
20:28:18.990 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Received DHPart1 or DHPart2
     usingDH3k
20:28:18.990 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] Getting DH result for mode DH3k
20:28:18.990 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] Reading key bytes:  00
     eb7e1e32abab93668cbb3994ce104414c7af774db3a7592fe44 [...]
20:28:18.993 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] Read public key for DH3K mode:
     534421680209197040826403825496246660344237 [...]
20:28:19.015 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] DH shared secret:
     e256cc1d4d935d7deaa020a8209f4e2d9183346efa1b7d633c1ed3 [...]
20:28:19.016 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Sending DHPart2...
20:28:19.017 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: calculateS1: no retained secrets.
20:28:19.017 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: iKDFContext:
20:28:19.017 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: S0:
20:28:19.018 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: New retained secret:
20:28:19.020 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: calculateSharedKeys(), SAS: tempest
     coherence
20:28:19.020 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: iTxMasterKey:
20:28:19.020 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: iTxMasterSalt:
20:28:19.020 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: iRxMasterKey:
20:28:19.020 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: iRxMasterSalt:
20:28:19.349 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received DHPart2
20:28:19.350 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received DHPart2 - not initiator
20:28:19.350 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Processing DHPart2.....
20:28:19.350 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received DHPart1 or DHPart2
     usingDH3k
20:28:19.350 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Getting DH result for mode DH3k
20:28:19.350 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Reading key bytes:  003
     fc4d80e01b3729d42a96520867544f614ce1d8d45637d6621f [...]
20:28:19.352 [ZRTP-ZRTP-1] DEBUG - [D22EA391] Read public key for DH3K mode:
     144715746101581623712250303795625375149595 [...]
20:28:19.365 [ZRTP-ZRTP-1] DEBUG - [D22EA391] DH shared secret:
     e256cc1d4d935d7deaa020a8209f4e2d9183346efa1b7d633c1ed3 [...]
20:28:19.365 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: calculateS1: no retained secrets.
20:28:19.366 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: iKDFContext:
20:28:19.366 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: S0:
20:28:19.367 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: New retained secret:
20:28:19.367 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: calculateSharedKeys(), SAS: tempest
     coherence
20:28:19.367 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: iTxMasterKey:
20:28:19.367 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: iTxMasterSalt:
20:28:19.367 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: iRxMasterKey:
20:28:19.368 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: iRxMasterSalt:
20:28:19.368 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Sending Confirm1...
```

```
20:28:19.368 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Confirm plainBytes:
20:28:19.370 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Total length in bytes: 76
20:28:19.409 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Received Confirm1
20:28:19.410 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Confirm plainBytes:
20:28:19.410 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Confirm optlen=20;signLen=0;d=false;
     a=false;v=false;e=false;cacheExpiry=4294967295
20:28:19.410 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTPCache: resetTrust(
     eec224f7c516dd05e0431e79
20:28:19.411 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Sending Confirm2...
20:28:19.411 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Confirm plainBytes:
20:28:19.412 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Total length in bytes: 76
20:28:19.412 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] got tx master key (sha256 hashed): 3
     b9648105c193dfe4dcff872243ecce54b987851d5b5d101a20e82f2e068b67c
20:28:19.413 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] got rx master key (sha256 hashed):
     b341e162c46ca891f10dad2a0de9f3c197a32bdd197c0e6050973ce4ba9d6234
20:28:19.470 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Received Confirm2
20:28:19.470 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Confirm plainBytes:
20:28:19.471 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Confirm optlen=20;signLen=0;d=false;
     a=false;v=false;e=false;cacheExpiry=4294967295
20:28:19.471 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTPCache: resetTrust(
     a6f2535d70a944d12386347d
20:28:19.472 [ZRTP-ZRTP-1] DEBUG - [D22EA391] got tx master key (sha256 hashed):
     b341e162c46ca891f10dad2a0de9f3c197a32bdd197c0e6050973ce4ba9d6234
20:28:19.472 [ZRTP-ZRTP-1] DEBUG - [D22EA391] got rx master key (sha256 hashed): 3
     b9648105c193dfe4dcff872243ecce54b987851d5b5d101a20e82f2e068b67c
20:28:19.473 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: sessionCompleted(true)
20:28:19.474 [ZRTP-ZRTP-1] DEBUG - [D22EA391] ZRTP: Thread Ending
20:28:19.500 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Received Conf2ACK
20:28:19.500 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: sessionCompleted(true)
20:28:19.501 [ZRTP-ZRTP-0] DEBUG - [A01B4A24] ZRTP: Thread Ending
20:28:20.504 [Timer-2] TRACE - ZRTP negotiation finished
20:28:20.524 [Timer-3] TRACE - ZRTP negotiation finished
20:28:20.588 [SessionWorker:D22EA391 RUNNING] TRACE - state: INITIATING -> ACTIVE, peer=
     A01B4A24
20:28:20.588 [SessionWorker:A01B4A24 RUNNING] TRACE - state: INITIATING -> ACTIVE, peer=
     D22EA391
20:28:20.589 [SrtpBridge:D22EA391] DEBUG - srtpBridge: localPort=53975 remoteAddress
     =127.0.0.1 remotePort=55004
Updating fda90812076f91ed97ec50e0824ec8a1 -> 5fb30933af49ecdb1a3b70351871ba8f
20:28:20.638 [SrtpBridge:A01B4A24] DEBUG - srtpBridge: localPort=54576 remoteAddress
     =127.0.0.1 remotePort=56483
Updating dea4b09c444b7871b65829f55b708d21 -> c02b4f1cf781077d4d3598388f07e6b0

------------------------------------------------------------
Step: 3b: check ACTIVE status on A's side

------------------------------------------------------------
Step: 3c: compare negotiated SAS strings

A's SAS is: tempest coherence
B's SAS is: tempest coherence

------------------------------------------------------------
Step: AUDIO: A sends a packet to B
```

```
Bridge A: 55004 -> 53975 using key = 2iO1P+wGUhDdSe37JjKkjPDqYTuy7ghmrg27tKRLPYc=
Bridge B: 56483 -> 54576 using key = dtJ15LPdVV9YsaeWLLPr/D8VBer6uGcWdX9K/+d09/o=

------------------------------------------------------------
Step: 4a: A closes call

20:28:27.483 [main] TRACE - Stop calling
20:28:27.502 [SessionWorker:D22EA391 RUNNING] TRACE - state: ACTIVE -> CLOSING, peer=
     A01B4A24
20:28:28.003 [SessionWorker:A01B4A24 RUNNING] TRACE - state: ACTIVE -> CLOSING, peer=
     D22EA391
20:28:28.004 [SessionWorker:D22EA391 RUNNING] TRACE - state: CLOSING -> NOT_ACTIVE, peer
     =null
20:28:28.305 [SessionWorker:A01B4A24 RUNNING] TRACE - state: CLOSING -> NOT_ACTIVE, peer
     =null
20:28:29.470 [Thread-4] TRACE - ListeningThread stopped!
20:28:29.500 [Thread-2] TRACE - ListeningThread stopped!

------------------------------------------------------------
Step: 4b: Check B's state

------------------------------------------------------------
Step: 5: A shuts down

20:28:35.494 [MainControl STOPPING] TRACE - Shutting down...
20:28:35.509 [DatagramBroadcast:A01B4A24 STOPPING] TRACE -     time stats: avg: 0.0 dev:
      0.047 min/max: 0/64
20:28:35.509 [DatagramBroadcast:D22EA391 STOPPING] TRACE -     time stats: avg: 0.0 dev:
      0.047 min/max: 0/64
20:28:35.511 [DatagramBroadcast:A01B4A24 STOPPING] TRACE - interval stats: avg: 30.0 dev
     : 0.000 min/max: 1/64
20:28:35.512 [DatagramBroadcast:D22EA391 STOPPING] TRACE - interval stats: avg: 30.0 dev
     : 0.000 min/max: 1/64
20:28:35.515 [DatagramListener:D22EA391] DEBUG - Finished: mExecutorHrtcp.
     getLargestPoolSize()=1, maxQueueSize=2
20:28:35.515 [DatagramListener:A01B4A24] DEBUG - Finished: mExecutorHrtcp.
     getLargestPoolSize()=1, maxQueueSize=2
20:28:35.515 [DatagramListener:D22EA391] DEBUG - Finished: mExecutorHrtp.
     getLargestPoolSize()=1, maxQueueSize=1
20:28:35.516 [DatagramListener:A01B4A24] DEBUG - Finished: mExecutorHrtp.
     getLargestPoolSize()=1, maxQueueSize=1
20:28:35.522 [SessionWorker:D22EA391 STOPPING] TRACE - stopped
20:28:35.522 [SessionWorker:A01B4A24 STOPPING] TRACE - stopped
20:28:35.609 [MainControl STOPPING] TRACE - ...everything is shut down
```

A.3.2 Screenshots of the Android App


HRTP Phone

0033CBBB

Create a new identifier on the current gateway.

Current gateway: 192.168.178.39:8080

Create new identifier

Please enter a new password twice in order to create a new identifier on the selected gateway. The password must have at least 4 characters.

Current gateway: 192.168.178.39:8080

Peers

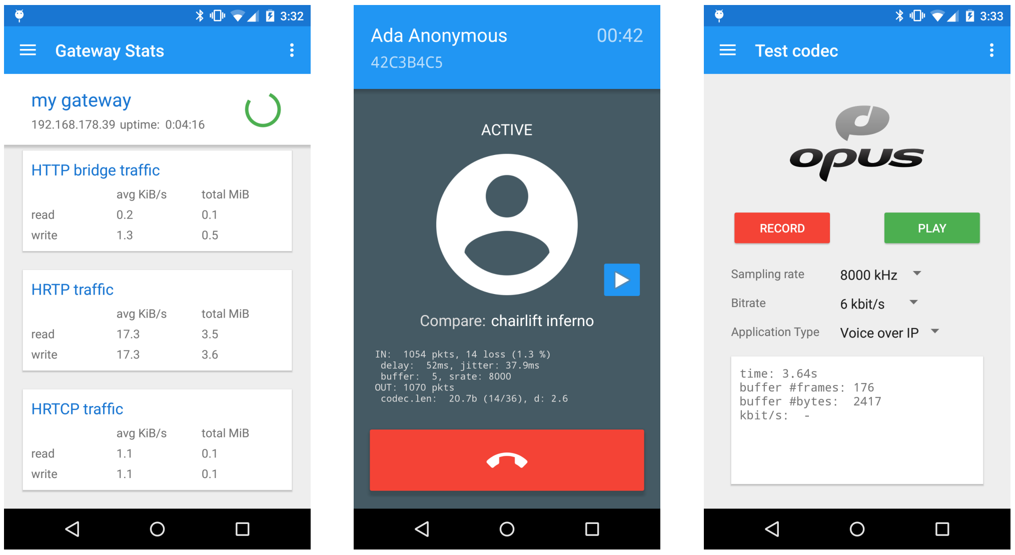
Ada Anonymous	42C3B4C5	connected, 0 minutes ago
Douglas Delay	B826A7F9	connected, 5 minutes ago
Martin Multiplex	593C212A	connected, 12 hours ago
Ursula Unobservable		connected

Add new peer

Enter Name: Philip Peer

Enter identifier: ABCD1234

Figure A.1: Left: Login screen after the first start of the app. Middle: Creating a new local member on the gateway from within the app. Right: Adding a new locally stored contact. Own graphics.


Gateway Stats

my gateway

192.168.178.39 uptime: 0:04:16

HTTP bridge traffic

	avg KiB/s	total MiB
read	0.2	0.1
write	1.3	0.5

HRTP traffic

	avg KiB/s	total MiB
read	17.3	3.5
write	17.3	3.6

HRTCP traffic

	avg KiB/s	total MiB
read	1.1	0.1
write	1.1	0.1

Ada Anonymous 00:42

42C3B4C5

ACTIVE

Compare: chairlift inferno

```
IN:  1054 pkts, 14 loss (1.3 %)
 delay:  52ms, jitter: 37.9ms
 buffer:  5, srate: 8000
OUT: 1070 pkts
 codec.len:  20.7b (14/36), d: 2.6
```

Test codec

Sampling rate: 8000 kHz

Bitrate: 6 kbit/s

Application Type: Voice over IP

```
time: 3.64s
buffer #frames: 176
buffer #bytes:  2417
kbit/s:  -
```

Figure A.2: Left: Statistics about the gateway showing the HRTP and HRTCP bandwidth. Middle: An active call with another peers showing the SAS and debug information. Right: Fragment for testing the Opus codec integration and performance. Own graphics.

A.4 Used Tools

A.4.1 Development Tools

JetBrain IntelliJ 19 Version 14.0.3 — IDE for Java

Oracle Java Development Kit Version 1.7.0_80 — Compiler and tools for Java

AndroidStudio Version 1.1.0 — IDE for Android

Android SDK Version 24.2 — Source Development Kit for Android

Android NDK Version r10d — Native Development Kit for Android

A.4.2 Used Libraries

Apache Commons CLI Version 1.2 — Library for parsing command line arguments

BouncyCastle Version 1.52 — Cryptographic library used for implementations of RSA, AES, HMAC and similar

Club Mate Version 2015 — Refreshing beverage high in caffeine

Dagger Version 1.2.2 — Dependency injection framework used for both the gateway and the Android application

Google Guava Version 18.0 — Library providing data structures and concurrency implementations

Jackson Version 2.5.1 — JSON library used for parsing and generating configuration and API body

Logback Version 1.1.2 — Logging library

io.netty Version 4.0.26 — Network and HTTP library used for providing the API server

okio Version 1.3.0 — Library for secure and convenient byte buffer handling

zorg Version 25 Jan 2015 — Open-source implementation of ZRTP

A.4.3 Documentation

yED Version 3.14.2 — Graph editor used for flow charts and UML diagrams

LyX Version 2.0.8.1 — Visual LaTeX editor

PDF LaTeX Version 3.1415926-2.5-1.40.14 — Type setting program

JabRef Version 2.10b2 — Bibtex reference manager

gnuplot Version 4.6 — Plotting program

Bibliography

[1] United Nations General Assembly, "Universal declaration of human rights," *UN General Assembly*, 1948.

[2] A. Pfitzmann and M. Köhntopp, "Anonymity, unobservability, and pseudonymity - a proposal for terminology," in *Designing privacy enhancing technologies.* Springer, 2001, pp. 1–9.

[3] N. F. Johnson and S. Jajodia, "Exploring steganography: Seeing the unseen," *Computer*, vol. 31, no. 2, pp. 26–34, 1998.

[4] P. Sprenger, "Sun on privacy:"get over it"," *Wired News*, vol. 26, pp. 01–99, 1999.

[5] R. Dingledine, N. Mathewson, and P. Syverson, "Tor: The second-generation onion router," DTIC Document, Tech. Rep., 2004.

[6] L. Ferran, "Ex-nsa chief: "we kill people based on metadata"," *ABC News*, May 2014. [Online]. Available: http://abcnews.go.com/blogs/headlines/2014/05/ex-nsa-chief-we-kill-people-based-on-metadata/

[7] I. TeleGeography, *Telegeography Report Executive Summary.* TeleGeography, Inc, 2014.

[8] C. Eckert, *IT-Sicherheit: Konzepte-Verfahren-Protokolle.* Oldenbourg Verlag, 2013.

[9] C. Diaz, S. Seys, J. Claessens, and B. Preneel, "Towards measuring anonymity," in *Privacy Enhancing Technologies.* Springer, 2003, pp. 54–68.

[10] D. Chaum, "Untraceable electronic mail, return addresses, and digital pseudonyms," *Communications of the ACM*, vol. 24, no. 2, pp. 84–90, 1981.

[11] U. Möller, L. Cottrell, P. Palfrader, and L. Sassaman, "Mixmaster protocol-version 2," *Draft, July*, 2003.

[12] G. Danezis, R. Dingledine, and N. Mathewson, "Mixminion: Design of a type iii anonymous remailer protocol," in *Security and Privacy, 2003. Proceedings. 2003 Symposium on.* IEEE, 2003, pp. 2–15.

[13] A. Serjantov, R. Dingledine, and P. Syverson, "From a trickle to a flood: Active attacks on several mix types," in *Information Hiding.* Springer, 2003, pp. 36–52.

[14] J. Reardon, "Improving tor using a tcp-over-dtls tunnel," Master's thesis, University of Waterloo, 2008.

[15] S. J. Murdoch and G. Danezis, "Low-cost traffic analysis of tor," in *Security and Privacy, 2005 IEEE Symposium on.* IEEE, 2005, pp. 183–195.

[16] A. Pfitzmann, B. Pfitzmann, and M. Waidner, "Isdn-mixes: Untraceable communication with very small bandwidth overhead," in *Kommunikation in verteilten Systemen.* Springer, 1991, pp. 451–463.

[17] D. Chaum, "The dining cryptographers problem: Unconditional sender and recipient untraceability," *Journal of cryptology*, vol. 1, no. 1, pp. 65–75, 1988.

[18] S. Goel, M. Robson, M. Polte, and E. Sirer, "Herbivore: A scalable and efficient protocol for anonymous communication," Cornell University, Tech. Rep., 2003.

[19] G. Danezis, C. Diaz, C. Troncoso, and B. Laurie, "Drac: An architecture for anonymous low-volume communications," in *Privacy Enhancing Technologies.* Springer, 2010, pp. 202–219.

[20] J.-F. Raymond, "Traffic analysis: Protocols, attacks, design issues, and open problems," in *Designing Privacy Enhancing Technologies.* Springer, 2001, pp. 10–29.

[21] G. Danezis and A. Serjantov, "Statistical disclosure or intersection attacks on anonymity systems," in *Information Hiding*, ser. Lecture Notes in Computer Science, J. Fridrich, Ed. Springer Berlin Heidelberg, 2005, vol. 3200, pp. 293–308. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30114-1_21

[22] N. Mathewson and R. Dingledine, "Practical traffic analysis: Extending and resisting statistical disclosure," in *Privacy Enhancing Technologies.* Springer, 2005, pp. 17–34.

[23] X. Wang, S. Chen, and S. Jajodia, "Tracking anonymous peer-to-peer voip calls on the internet," in *Proceedings of the 12th ACM conference on Computer and communications security.* ACM, 2005, pp. 81–91.

[24] Bundesnetzagentur für Elektrizität, Gas, Telekommunikation, Post und Eisenbahnen, "Jahresbericht 2014," April 2015.

[25] A. White, A. Matthews, K. Snow, and F. Monrose, "Phonotactic reconstruction of encrypted voip conversations: Hookt on fon-iks," in *Security and Privacy (SP), 2011 IEEE Symposium on*, May 2011, pp. 3–18.

[26] J. Kurose and K. Ross, *Computer Networks: A Top-Down Approach*, 6th ed., M. Horton, Ed. Wiley-Interscience, 2013.

[27] C. Perkins, *RTP: Audio and Video for the Internet.* Addison-Wesley Professional, 2003.

[28] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[29] C. Perkins, O. Hodson, and V. Hardman, "A survey of packet loss recovery techniques for streaming audio," *Network, IEEE*, vol. 12, no. 5, pp. 40–48, 1998.

[30] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, *RFC 3550: RTP: A Transport Protocol for Real-Time Applications*, Std., 2003.

[31] M. Baugher, D. McGrew, M. Naslund, E. Carrara, and K. Norrman, *RFC 3711: The Secure Real-time Transport Protocol (SRTP)*, Std., 2004. [Online]. Available: https://tools.ietf.org/html/rfc3711

[32] P. Zimmermann, Z. Project, E. A. Johnston, Avaya, J. Callas, and I. Apple, *RFC 6189: ZRTP: Media Path Key Agreement for Unicast Secure RTP*, Std., April 2001. [Online]. Available: https://tools.ietf.org/html/rfc6189

[33] R.-O. E. Scheme, "Algorithm specification and supporting documentation," *RSA Laboratories, RSA Security Inc*, vol. 20, 2000.

[34] Garnter, "Gartner says smartphone sales surpassed one billion units in 2014," online, March 2015. [Online]. Available: http://www.gartner.com/newsroom/id/2996817

[35] Google Inc., "Dashboards - Platform Versions," June 2015. [Online]. Available: https://developer.android.com/about/dashboards/index.html?utm_source=suzunone

[36] J.-M. Valin, K. Vos, and T. Terriberry, "Definition of the opus audio codec," Tech. Rep., 2012. [Online]. Available: http://www.rfc-editor.org/info/rfc6716

[37] C. Hoene, J.-M. Valin, K. Vos, and J. Skoglund, "Summary of opus listening test results," 2013. [Online]. Available: http://tools.ietf.org/html/ietf-codec-results-03

[38] J. R. Douceur, "The sybil attack," in *Peer-to-peer Systems.* Springer, 2002, pp. 251–260.

[39] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to algorithms.* MIT press, 2009.

[40] D. Hughes and V. Shmatikov, "Information hiding, anonymity and privacy: a modular approach," *Journal of Computer security*, vol. 12, no. 1, pp. 3–36, 2004.

[41] J. Van Eijck and S. Orzan, "Epistemic verification of anonymity," *Electronic Notes in Theoretical Computer Science*, vol. 168, pp. 159–174, 2007.

[42] C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin, “Intrusion detection by machine learning: A review,” *Expert Systems with Applications*, vol. 36, no. 10, pp. 11 994–12 000, 2009.

APPENDIX B

DVD

B.1 Content of the DVD

evaluation/ All raw data from the evaluation

implementation/ Implementation projects of this thesis

- **android/** Implementation of the handset as an AndroidStudio project
- **gateway/** Implementation of the gateway as an IntelliJ project
- **misc/** Various Python scripts used for the evaluation

readme.txt Information file

thesis.pdf Digital version of this documentation

[Place for the DVD]

B.2 Quick-Start Guide

This section explains how the individual components can be built and run in a local setup. The procedure has been tested on an Ubuntu 14.04 LTS machine and Google Nexus 5 smartphones (Android version 5.1.1). The projects come with all their libraries included as JAR archives. While describing the manual building via command line here, one is encouraged to use the respective IDEs.

B.2.1 Java Gateway

Prerequisites & Building For compiling the gateway, the `ANT >= 1.9.3` build tool as well as Oracle's latest JDK for Java 7 is required. Also verify that the Java Cryptography Extension (JCE) is properly configured. For instrumenting the test cases, the IntelliJ IDE is recommended. The gateway can be built using the following command:

```
cd implementation/gateway
ant -f gateway.xml all
```

Configuration Subsequently, the following commands will first create two create two local members and the local configuration file:

```
cd implementation/gateway/out/artifacts/gateway_jar/
java -jar gateway.jar -create -config hrtp.conf
java -jar gateway.jar -create -config hrtp.conf
```

Now, the local configuration file `hrtp.conf` needs manual editing. First the `password` fields of the local members need to be set. Next, the `peerConfigurations` are inserted. For this the following skeleton is adjusted for each desired peer of the gateway and inserted separated by commas within the empty array structure:

```
{
    "encodedPublicKey": " << insert the peer’s public key here >> ",
    "hrtcpPort": 33001,
    "hrtpPort": 22001,
    "identifier": " << insert peer identifier here >> ",
    "inetAddress": "127.0.0.1"
},
```

Starting Finally, the gateway can be started using the following command. It will start the session for the individual local members and offer its HTTP API on port TCP:8080.

```
cd implementation/gateway/out/artifacts/gateway_jar/
java -jar gateway.jar -config hrtp.conf
```

B.2.2 Android Handset App

Prerequisites & Building For building the Android application, the API levels 14 and 22 need to be installed using the Android SDK Manager. The best way to start is downloading the AndroidStudio[1] package that includes the IDE. Without the IDE, the app can be compiled and installed on a smartphone using the following commands. For this the device has to have the ADB debug bride enabled and be connected via USB:

```
cd implementation/android
./gradlew installDebug
```

Configuration The Android app starts automatically as soon as it installed. The settings are accessible by clicking the "wheel" on the login screen or the three dots in the upper-right corner. Here, the IP of the gateway needs to be set. In a normal setup, SSL should be deactivated, as the process of creating pinning information is not covered here.

B.2.3 Android Opus Codec (optional)

The source folder for the Android app comes with a pre-compiled version of the Opus codec, namely `app/src/main/jniLibs/armeabi/libmyopus.so`, for the ARM-EABI architecture. It supports the instruction sets `ARMV5TE` and later as well as `Thumb-1`.

Prerequisites & Building If one really needs to build the Opus codec manually (e.g. for x86), the following commands will do so after the NDK is properly installed. A change of the target architecture, also requires one to change the build targets and types within the `build.gradle` configuration for the Android app.

```
cd implementation/android/app/src/main
/absolut_path_to_your_sdk_folder/android-ndk-r10d/ndk-build -B
```

[1] `https://developer.android.com/sdk/index.html`

Left Blank