MURMUR

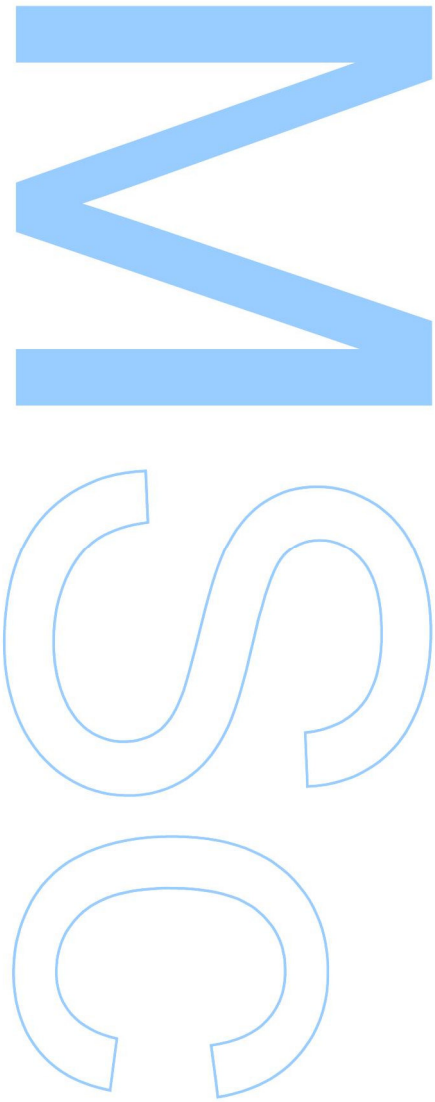
by ADYTA

Secure VoIP Communications

João Miguel Maia Soares de Resende
Mestrado Integrado em Engenharia de Redes e Sistemas Informáticos
Departamento de Ciência de Computadores
2016

Orientador
Prof. Dr. Manuel Eduardo Carvalho Duarte Correia, Professor Auxiliar, FCUP

Coorientador
Mestre Luís Daniel Freitas Azevedo Maia, CEO, Adyta



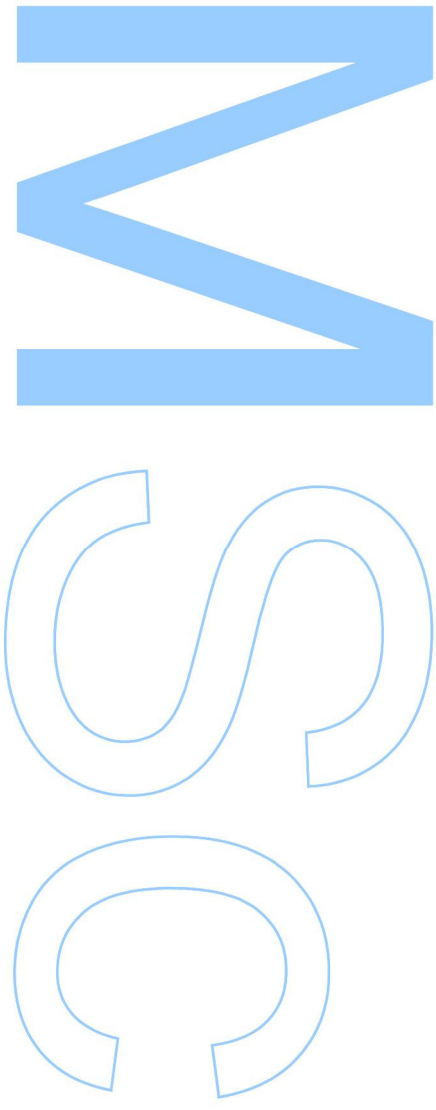
U.PORTO

FACULDADE DE CIÊNCIAS
UNIVERSIDADE DO PORTO

Todas as correções determinadas pelo júri, e só essas, foram efetuadas.

O Presidente do Júri,

Porto, _____/_____/_____



João Miguel Maia Soares de Resende

Secure VoIP communications

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2016

João Miguel Maia Soares de Resende

Secure VoIP communications

Dissertação submetida à Faculdade de Ciências da
Universidade do Porto como parte dos requisitos para a obtenção do grau de
Mestre em Engenharia de Redes em Sistemas Informáticos

Orientador: Prof. Dr. Manuel Eduardo Correia
Co-orientador: Mestre Luís Maia

Departamento de Ciência de Computadores
Faculdade de Ciências da Universidade do Porto
Junho de 2016

To my parents and girlfriend for all the incoditional love and support during this period...

Acknowledgments

I have to thank my advisor professor Manuel Eduardo Correia for all his support and guidance that made this work possible. I also want to specially thank Luís Maia for all his support and useful tips during the development of this work and the security knowledge transmitted.

I also want to thank Professor Luís Antunes for the interest that has always shown for this thesis and for the invitations to make part of c3p.

Also, a word of appreciation is due to Rolando Martins, for the useful tips to improve the application.

I want to thank my friends for their patience, support, happiness and smiles during this period.

Does not exists words to express the gratitude and love to my parents, without them this will not even be possible to imagine...

And last but not the least, for Patrícia for their unconditional love, for all the support not only the thesis but during this path to the future.

Abstract

Voice over Internet Protocol (VoIP) has been developed to enable massive real time voice communications over an Internet Protocol (IP) network. Moreover, Internet availability on mobile devices and much lower costs on cellular data usage, made VoIP very competitive with other digital cellular voice services, specially for international calls.

With the current widespread use of VoIP based communications, securely encrypting the voice channel is now even more important due to the unregulated and independent transnational points where a single communication may pass through. This has all become much more pertinent due to the well know massive governmental spying activities disclosed by the Wikileaks and "John Snowded" incidents. The main challenge here consists on how to securely exchange encrypting session keys, directly on the voice channel, in a peer-to-peer manner and in such a way that it cannot be intercepted by the service providers.

The Zimmermann Real-time Transport Protocol (ZRTP) protocol is a well known solution to this problem. It does all its session keys management directly on the voice channel, securely hidden and independently from the service providers. These session keys are then used to directly encrypt the voice channel, making ZRTP clients highly resistant to attempts from the service providers and governmental spying agencies to decrypt VoIP calls.

In this thesis, we analyse two different implementations of a Session Initiation Protocol (SIP) library with ZRTP support. We employ these libraries to implement two different versions of a secure mobile Android app for VoIP with ZRTP support. These apps maintain all its local data, including cached session keys, fully encrypted on the mobile device.

During the development process we have also found an intrinsic vulnerability on the ZRTP protocol that we illustrate by the means of a fully functional Man in the midle

(MITM) attack to the ZRTP protocol. This attack works with the most popular VoIP clients with ZRTP support, such as *Linphone*,*CSipSimple* and *Silent Phone*. The vulnerability is based on the lack of information given by the device ZRTP identifier (ZID). Our app has this problem mitigated by taking advantage of SIP signalling.

We also describe some scenarios where this MITM attack succeeds and we then provide multiple solutions to mitigate it. We also suggest a possible small change to the ZRTP specifications and suggest another solution that does not need to change its Request for comments (RFC).

Resumo

As comunicações VoIP têm vindo a ser desenvolvidas para permitir comunicações em tempo real através de uma rede IP. Além disso, a disponibilidade de Internet em dispositivos móveis e o baixo custo da utilização, tornam o VoIP competitivo com os restantes serviços digitais de comunicação, especialmente para chamadas internacionais.

Com a atual difusão do uso dos sistemas de comunicação baseados em VoIP, cifrar o canal de áudio é agora ainda mais importante devido à falta de regulamentação e pontos transnacionais independentes, onde uma única comunicação pode passar. Isto tornou-se tudo mais importante devido aos ataques bem conhecidos de espionagem realizados pelo governo reveladas pelo incidente do Wikileaks e "John Snowded". O desafio principal consiste em como trocar, de forma segura, uma chave de sessão diretamente no canal de áudio, de uma forma ponto a ponto e de uma maneira que não permite a intercepção pelo provedor de serviço.

Uma solução para este problema é o protocolo ZRTP. Ele faz toda a troca de chaves diretamente no canal de áudio, de forma segura e independente do provedor de serviço. Estas chaves de sessão são utilizadas para cifrar os pacotes de áudio, tornando os clientes que suportam ZRTP, altamente resistentes a tentativas por parte dos provedores ou instituições governamentais de espionagem para decifrar chamadas VoIP.

Nesta dissertação, nós analisamos dois tipos diferentes de implementações de bibliotecas SIP com suporte para ZRTP. Nós utilizamos estas bibliotecas para implementar duas versões diferentes de uma aplicação móvel para comunicações VoIP com ZRTP. Estas aplicações mantêm todos os dados locais cifrados no dispositivos.

Durante a fase de desenvolvimento, nós encontramos uma vulnerabilidade intrínseca no protocolo ZRTP, que demostramos com base num ataque de MITM ao ZRTP. Este ataque funciona com os mais populares clientes de VoIP que suportam ZRTP, tais

como o *Linphone*, *CSipSimple* e *Silent Phone*. A vulnerabilidade é baseada na falta de informação dada pelo ZID de um dispositivo ZRTP. A nossa aplicação mitiga este problema recorrendo ao protocolo de sinalização SIP.

Também descrevemos alguns cenários onde o ataque pode decorrer e propomos múltiplas soluções para o mitigar, como uma pequena alteração a especificação do ZRTP, bem como, outras soluções que não precisam da alteração no RFC.

List of Acronyms

AES Advanced Encryption Standard

AIDL Android Interface Definition Language

API Application Programming Interface

ARP Address Resolution Protocol

CID Client Identifier string

CPU Central Processing Unit

DB database

DH Diffie-Hellman

DTLS Datagram Transport Layer Security

DUM Dialog User Manager

GB Gigabyte

GPL General Public License

GSM Global System for Mobile Communications

HD High definition

ICE Interactive Connectivity Establishment

IETF Internet Engineering Task Force

IMSI International Mobile Subscriber Identity

IP Internet Protocol

JDK Java Development Kit

JNI Java Native Interface

MAC media access control

MGCP Media Gateway Control Protocol

MIME Multipurpose Internet Mail Extensions

MITM Man in the midle

NATO North Atlantic Treaty Organisation

NAT Network Address Translator

NFC Near Field Communication

PBKDF2 Password-Based Key Derivation Function 2

PID Process ID

PKI Public key infrastructure

PSTN Public Switched Telephone Network

QoS Quality of Service

QR Quick Response

RAM Random Access Memory

Recon Conversation Manager

RFC Request for comments

RTCP Real-Time Transport Control Protocol

RTCP Real-Time Transport Control Protocol

RTP Real-time Transport Protocol

S/MIME Secure/Multipurpose Internet Mail Extensions

SAS short authentication string

SDES Session Description Protocol (SDP) Security Descriptions for Media Streams

SDK Software Development Kit

SDP Session Description Protocol

SEE SQLite Encryption Extension

SHA256 Secure Hash Algorithm-256

SIM Subscriber Identity Module

SIP Session Initiation Protocol

SRTP Secure Real-time Transport Protocol

SSL Secure Sockets Layer

STUN Session Traversal Utilities for NAT

SWIG Simplified Wrapper and Interface Generator

TCP Transmission Control Protocol

TLS Transport Layer Security

TURN Traversal Using Relays around NAT

UDP User Datagram Protocol

UI User Interface

URI Uniform Resource Identifier

VoIP Voice over Internet Protocol

VPN Virtual private network

XML eXtensible Markup Language

ZID ZRTP identifier

ZRTP Zimmermann Real-time Transport Protocol

Contents

Acknowledgments **i**

Abstract **ii**

Resumo **iv**

List of Tables **xiii**

List of Figures **xvi**

1 Introduction **1**

1.1 Motivation 3

1.2 Proposed Solution 4

1.2.1 Objectives 5

1.2.2 Features 5

1.3 Contributions 6

1.4 Outline 6

2 Background Concepts **8**

2.1 Public and Symmetric Keys 8

2.1.1 Diffie-Hellman 9

2.1.2 Security properties 10

2.2 Media Security 10
2.2.1 RTP 10
2.2.2 SRTP 11
2.3 Key Exchange Mechanisms 11
2.3.1 SDES 11
2.3.2 S/MIME 12
2.3.3 ZRTP 14
2.4 Normal VoIP Call 17
2.5 NATO alphabet 18

3 State of the art **19**

3.1 VoIP 19
3.1.1 SIP 19
3.1.1.1 SDP 20
3.1.1.2 Presence 20
3.1.2 H.323 21
3.1.3 MGCP 21
3.2 Android 21
3.2.1 Problems from the operative system 22
3.3 Similar applications 23
3.3.1 *Linphone* 23
3.3.2 *PJSIP* 25
3.3.3 *reSIProcate* 25
3.3.4 *Silent Phone* 26
3.3.5 *PrivateWave* 26
3.3.6 *Whatsapp* 26

3.3.7 Comparison 27
3.3.8 Security of *Linphone* and *CSipSimple* 27
3.4 Differences between application 28

4 Implementation **30**
4.1 Implementation scheme 30
4.1.1 Linphone databases 31
4.1.2 Interface 32
4.1.3 Provision with QR Code 39
4.2 Second version - PJSIP 41
4.2.1 Why change to a new implementation 41
4.2.2 Add the ZRTP 45
4.2.3 Integrate PJSIP in Android 46
4.2.4 Authentication system 48
4.2.5 PJSIP database 51
4.3 Final application 53
4.3.1 Doze mode 56
4.3.2 Preliminary analysis performance 58

5 Security Issues **60**
5.1 MITM attacks on ZRTP 60
5.2 The attack discovered on ZRTP 63
5.2.1 Get the signalling path 63
5.2.2 Media session 66
5.2.3 SAS defeating 67
5.2.4 A Practical Implementation of the Attack 68

5.2.5 The problem . . . 72
5.2.6 Solution to the MITM attack . . . 72
5.2.7 What can we conclude . . . 74

6 Conclusion and Future Work **76**

6.1 Research Summary . . . 76
6.2 Main contributions . . . 77
6.3 Future Work . . . 77
6.4 Final conclusions . . . 78

Bibliography **79**

A Development notes **82**

A.1 Programming languages used . . . 82
A.2 Software used . . . 82

B Manual **85**

B.1 Patch to ZRTP . . . 85
B.2 Compile *PJSIP* . . . 89
B.3 Use Android Interface Definition Language (AIDL) . . . 92
B.4 *PJSIP* stack supported features . . . 94
B.4.1 SIP Capabilities . . . 94
B.4.2 NAT Traversal . . . 96
B.4.3 Media/audio capabilities . . . 97

List of Tables

2.1 NATO alphabet 18

3.1 Comparison of VoIP systems 27

4.1 System supported RFC 54

4.2 Permissions of the final application 55

List of Figures

2.1 SDES eavesdropping problem 12
2.2 Using S/MIME to encrypt SIP messages [1] 13
2.3 ZRTP protocol 16
2.4 SIP signalling between and the media channel Alice and Bob 17

3.1 Doze mode Android 22
3.2 Linphone Architecture 24

4.1 Murmur contacts menu 33
4.2 Linphone contact menu 33
4.3 Contact Menu Linphone 34
4.4 Contact menu Murmur 34
4.5 Options Menu Linphone 35
4.6 Messages menu Murmur 35
4.7 Dialer menu Linphone 35
4.8 Linphone secured call 36
4.9 Linphone, SAS needs verification 36
4.10 SAS for the call, and the accept, deny SAS march 37
4.11 Murmur first call 38
4.12 Murmur after accept the call 38

4.13 Murmur while don't perform SAS exchange, or without encryption . . . 38
4.14 Linphone configure SIP account 40
4.15 Murmur configure SIP account 40
4.16 Quick Response (QR) code example to pass to the application 41
4.17 Caller, in the first call 42
4.18 Callee, in the first call 42
4.19 Caller, in the second call with Linphone 43
4.20 Callee, in the second call with Linphone 43
4.21 Caller, in the second call with PJSIP 44
4.22 Callee, in the second call with PJSIP 44
4.23 First structure, with *PJSIP* library 47
4.24 Final version of the system with *PJSIP* library 48
4.25 First use of the system - model A 49
4.26 All the login attempts after the first one - model A 50
4.27 First use of the system - model B 51
4.28 All the login attempts after the first one - model B 51
4.29 Consumption of the Android interface when not in a call 58
4.30 Consumption of the Android interface when in a call 59
4.31 Android service consumptions 59

5.1 Attack architecture [2] 61
5.2 Attack on the SIP signalling [2] 62
5.3 Address Resolution Protocol (ARP) poison connection 64
5.4 SIP *INVITE* transformation 65
5.5 Packet flow 66
5.6 Architecture of Mallory device in the MiM mode 67

5.7 After exchange the SAS, the verified flag 68

Chapter 1

Introduction

Communication is a fundamental characteristic of *being human* and one of the main pillars of modern life. The value that is put on its privacy is a sure sign of an healthy and mature society. To ensure it, has always been a challenge that has been evolving over time. Historically there have been many efforts to secure communications, by for example avoiding cleartext communications by encrypting everything, independently of the communication media being used.

VoIP is a technology developed to transport voice over the IP network. With the higher availability of Internet in mobile devices and with low cost of mobile data usage, VoIP can even now support High definition (HD) codecs in mobile contexts. While in the IP realm it may also explore other channels such as video or data. Higher flexibility and low price are the principal ingredients that attract many clients to IP bases systems. This way, it is expected that VoIP will ultimately replace the Public Switched Telephone Network (PSTN) [3].

There are many attack scenarios for Global System for Mobile Communications (GSM) communications, such as the *Gemalto* Subscriber Identity Module (SIM) hack[1]. There are even devices (International Mobile Subscriber Identity (IMSI)-catchers) that allow an attacker to eavesdrop and wiretap communications. Essentially a IMSI-catcher is a device that simulates a mobile tower, and is mainly used for example by security forces to perform a MITM. This device acts as a relay between the hacked mobile phone and a real tower, thus intercepting all traffic between the attacked devices. IMSI-catchers for 3G networks are readily available for sale, as illustrated for example by

[1] https://theintercept.com/2015/02/19/great-sim-heist/, 2015. [Online; Accessed 2016-6-17]

the 3G-Cat[2], that basically works by being able to downgrade mobile devices cellular communications back to GSM.

Moreover VoIP products have been repeatedly found to be riddled with vulnerabilities in different key areas of the protocol, that can be readily exploited. In some cases, the server can even see the keys to decrypt the packets, as shown for examples with *Cisco TelePresence* [3] systems.

A major problem with encrypted VoIP communication is also the quality of service of time dependant services, mainly audio, between the caller and callee. Moreover the overhead introduced by the crypto sometimes results in delays that can produce audio/video jitter. Users can thus have a bad usage experiences caused by the encryption overhead. Consequently the encryption process has to be fast to reduce the *play out delay* between the two endpoints [4, 5] that take part in a call.

We want to focus in a solution that provides to the users, not only a better audio, but also the guarantee of a protocol that keeps the session keys out of reach, even from the service providers infrastructure servers. This way, we want to produce one application that can for example be resistant to service providers wanting to "wiretap" or otherwise interfere with the communications between two trusting peers.

ZRTP is a security protocol that allows for VoIP client peers to perform a secure session key exchange on top of a media session that has been established between them, using regular VoIP signalling facilities. This way the secret session key that is established between the peers is completely independent from the VoIP signalling being used to establish the call and thus cannot be intercepted by the VoIP service providers.

There are however are some practical challenges to the security of practical implementations of point-to-point communications and the real trust one can put on those applications, specially if the deployment scenario involves for example questions of *national sovereignty* where we can easily think of a realistic scenario where a restrict group of members of government want to be sure that no one else (specially foreign governments or political adversaries) is capable of listening to their phone conversations.

[2] http://www.interceptors.com/intercept-solutions/detects-parameters-3G-networks.html, [Online; Accessed 2016-6-17]

[3] `http://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Video/telepresence.html`, 2009. [Online; Accessed 2016-6-17]

There are already some real implementations and deployments of ZRTP, *Silent Phone*, *Linphone* being the most popular. However these applications have not been independently certified and are being developed by companies from foreign countries with unknown agendas. We thus cannot fully trust them to be completely free from foreign tampering and free from "trojan" code, as illustrated by the well know massive governmental spying activities disclosed by the Wikileaks and "John Snowded" incidents.

We want to be able to completely audit the code and libraries being used in the development of a mobile VoIP client supporting ZRTP, in order to be able to provide a service users can fully trust and cannot be easily be subjected to wiretaps.

1.1 Motivation

With the adoption of VoIP based systems, attackers have an incentive to shift their attentions to service providers in order to intercept calls. With regular VoIP applications, intruders may be able to listen conversations between the caller and callee, log all the received calls by the callee or all the calls made by caller. This allows the intruder to store the data collected and search through the data for important information. A server with a packet sniffer on the network where the call takes place or where caller or callee are present, can collect the data and aggregate the VoIP calls, giving the opportunity to the hacker to get the confidential information or manipulate packets. [6]

We want to create an auditable VoIP application that supports ZRTP to promote cyber-sovereignty and security. With the application and the use of strong and well established cryptographic protocols we can ensure to the national institutional the security to exchange important information through a call.

Towards this goal we have audited two different libraries, the *PJSIP* and *Linphone* supporting ZRTP for VoIP application and we have programmed an Android based app where all the useful security features are present by default i.e. Transport Layer Security (TLS) support and where we adapted some of the security mechanism given by the ZRTP to alert the user and guarantee the security of the system, one example can be the display of a message to alert the user to exchange the short authentication string (SAS). We have also found and patched a vulnerability in ZRTP, a protocol described in the RFC [7].

During the development process we remove the components unnecessary to perform

a VoIP call (i.e. the possibility of using *bluetooth* to perform a call or the possibility to send messages) but maintaining the interoperability with other applications that support ZRTP. This way we were able to substantially reduce the surface of attack by reducing the number compiled features of the library and by reducing the set of system permissions to minimum required to run the Android application, capable of perform and receive calls.

The auditing of the libraries source code and the discovery of the vulnerability, allowed us to develop a best-of-breed Android App for mobile applications we can trust. We can now have secure and trusted point-to-point VoIP communications without being subjected to the ZRTP vulnerability we have identified.

1.2 Proposed Solution

The work detailed in this thesis aims to promote the privacy of *Android* users when using VoIP.

Towards this goal, we can take advantage of the existence of supporting libraries with VoIP implementations (i.e. *Linphone* or *PJSIP*). We audited the security features and we reduced the attack surface by dropping unnecessary features, to be possible the certification of the code by a third party. In relation to cryptographic protocols, we want to provide institutions the tools to better protect their communications, for example, if the entity's normally use in email certificates and are adapted to this methodology, they should be able to use certificates in VoIP communications to authenticate the users. We intend to use this with ZRTP for more decentralised hierarchies such as government, public institutions where if they listen each other in a VoIP call, they can identify each other.

To the proposed solution we want to give to the application the following principals:

- **Interoperability:** Application will be capable of communicate with other ZRTP applications;
- **Point-to-point security:** Data exchanges in the media, maintains the confidentiality and integrity;
- **Local security:** Protecting the data stored in the devices from malicious applications;

- **Limited functionality:** Offering limited features to the users, to reduce attack surface.

1.2.1 Objectives

The main goals of this thesis is to develop an *Android* application (*Murmur*) able to make VoIP communication with a reduced risk of hackers being able to wiretap into the content of the call. We want to maintain the application interoperable with different implementations of ZRTP to allow communication with other VoIP devices, making the exchange keys flow by the media session. We have the following tasks:

- **Technology Research:** Identifying the current state of the art technologies employed by VoIP stacks, understand the way their different components interact each other and identify the best cryptographic protocols to fit the costumer requirements.
- **Architecture Design:** Design an *Android* application capable of make efficiently VoIP communication and make the data exchange securely (*signaling*).
- **Implementation:** Develop and code the necessary components in order to deploy the proposed architecture in a real world environment and reducing the code to the minimum necessary.
- **Security tests:** Create and perform tests, to test the resistance to attacks on the protocol and in the implementations to the application.
- **Testing:** Deploy our *Android* application to a real use case scenario (like a university), and get feedback from users.
- **Authentication:** Provide the source code to a third entity to be able to validate the code, to make our system trustful and secure.

1.2.2 Features

The main features of our proposed system are as follows:

- **Easy Migration:** The final version supports ZRTP. This support is made by a library that has ZRTP support and that is capable of be integrated in new architectures, such as *iOS* devices.

- **High privacy level:** Personal information is only revealed to a user who is registered. A registered user has a password to authenticate in a SIP server.
- **Secure Call:** Users can make secure calls, independent from the signalling layer. This means that the SIP provider don't have any information about the keys used in a secure call.
- **High Availability:** Users can have a long period of time connected with their SIP server, without the need of introducing the password, after the application starts.
- **Interoperability:** - Guarantee that the application will be capable of communicate with the remaining existent applications that support ZRTP;
- **Limited functionality:** - Limited features and permissions to reduce the attack surface.

1.3 Contributions

We have created two different Android application, one based on *Linphone* with video support and one starting from a much smaller code base supported by a simpled SIP stack, *PJSIP*[4].

We have also discovered what we think is a non-trivial vulnerability in the ZRTP protocol. We have proposed and implemented a solution to this vulnerability, in the second more simpler VoIP android app we have developed.

1.4 Outline

This thesis is divided into 6 Chapters. The current chapter presents an introduction to all the work performed in this dissertation.

The next chapters of the thesis are organised as follows:

Chapter 2 presents a brief overview of the security components used on VoIP systems, dividing the components in two different fields: the secure media channel, where the user already have to contain a secret with the other end, and the key exchange protocol,

[4]http://www.pjsip.org/ [Online; Accessed 2016-6-17]

to exchange the symmetric key to accomplish a secure media session. Applications such *Linphone* and *Silent phone* and the signalling protocol are detailed in the State of the art, Chapter 3. In Chapter 4 we present the implementation details and approaches used, specifying some *Android* properties that helped in deployment. We also present the decision and evolution that we find and perform along the path to the application. With the develop of the application we find a cross library vulnerability detailed in Chapter 5, in this chapter we present the previous MITM attack and we explain how to exploit this security breach, and how can it bee mitigated with a possible change in the RFC from ZRTP. Lastly, the Chapter 6 presents some final remarks and lays the ground for future work to be performed to accomplish the main ideas proposed in 1.2.

Chapter 2

Background Concepts

In this chapter, we present an overview of security mechanisms, to the different steps from the VoIP communications, starting with the introduction of concepts such as public and symmetric keys, through the media session encryption and secure ways to exchange session keys to communicate without eavesdropping.

2.1 Public and Symmetric Keys

Establish a secure connection between two users, can be done with public keys or symmetric keys.

The idea behind the symmetric key is that, both participants, have a shared secret k between them and when, for example, *Alice* wants to send a message m to *Bob*.

She ciphers m with the k and send $E(m, k)$ to *Bob*. *Bob* takes the $E(m, k)$ and decipher m with k because $D(E(m, k), k) = m$. In the case that *Alice* want to send a message to *Carol*, she should use a new secret key to cipher the message. If this don't happen, *Bob* can decipher the message sent to *Carol*.

Public key cryptography, instead of having a key for each pair of participants (k to Alice and Bob), exists a pair of keys, public and private key, for each participant. The public key only cypher the message and the private key decipher the message. If *Bob* wants to speak with *Alice*, *Bob* should request *Alice* or a Public key infrastructure (PKI) [1] for the correspondent public key and after this, can cypher a message with

[1] PKI is a system responsible for maintaining and response to request of public keys

the *Alice* public key and send the message to *Alice*. If Bob cipher the message with Alice public key, only *Alice*, with their private key, can decipher the message.

2.1.1 Diffie-Hellman

Diffie-Hellman (DH) as described in [8, 9], is a protocol to generate a shared secret between two users in such a way that the secret can't be seen by one observer of the communication channel.

We can imagine two integers, P and E, where P is a prime number and E the primitive root of P, both P and E are public values, and that *Alice* and *Bob* want to establish a shared secret between them, K, for that:

1. Alice chooses a value X, where X is smaller than P, and calculate:

$$Xa = E^X \pmod{P}$$

2. Alice send Xa to Bob;

3. Bob choose a number Y and verify the same property as Alice to X and calculates:

$$Yb = E^Y \pmod{P}$$

4. Bob send Yb to Alice;

5. Alice computes:

$$K = Yb^X \pmod{P}$$

6. Bob computes:

$$K = Xa^Y \pmod{P}$$

7. Both ends have the same K, the shared secret.

As the DH key exchange does not offer a way to protect against MITM attacks, if an attacker is able to intercept a key agreement, can perform a MITM, however, in such cases, the attacker acts as an invisible partner where he have two different secrets: one in the channel between *Alice* and the attacker, and other in the channel with *Bob*.

2.1.2 Security properties

Authentication, confidentiality and integrity are the main principals, of a secure system.

- **Authentication** - A password and user can be used to authenticate a client. This way, if a client presents the correct password and user to a service, the service will authenticate the client as being that specific user;
- **Integrity** provides security that the data stored or send was not altered. Integrity ensures that the data is not modified after being cyphered by a person. If the data or the hash has been modified, the data becomes invalid;
- **Confidentiality** is the guarantee of the data privacy. The entities that are not allowed, cannot listen the call/data, except for the specific owners or receivers.

2.2 Media Security

Media Security, is one of the most important topics in the VoIP communication. In this channel, passes all the data regarding the audio of the call. In this case, exists two well established protocols, the Real-time Transport Protocol (RTP) when we want to establish clear text stream between two clients or Secure Real-time Transport Protocol (SRTP) when we want to use encryption and have a session key shared between the two endpoints of a call.

2.2.1 RTP

RTP is an application-level transport protocol for data, which provides end-to-end packet delivery with real-time properties, such as audio and video. RTP sessions consist of two streams: one for audio or video and a stream of control packets called Real-Time Transport Control Protocol (RTCP). RTP can operate either in a connection-oriented Transmission Control Protocol (TCP) or connection-less User Datagram Protocol (UDP) in the lower-layer. To send real-time data to multiple users it is possible to use RTP if the underlying network protocol provides multicast distribution. RTP alone does not provide any mechanism to provide Quality of Service (QoS), RTP relies on lower-layer protocols to accomplish this. If RTP uses

UDP, the sequence number allows at the application layer that the stream receiver reconstruct the packet sequence and decode one packet even if the previous still missing. The sequence number also allow that if the packet arrives out of order, we can organize the packet in the queue [10].

2.2.2 SRTP

The SRTP is a complement of the RTP [10, 11]. SRTP adds confidentiality, integrity and authentication to the RTP media channel and RTCP traffic. SRTP uses Advanced Encryption Standard (AES)[2] to cypher the media packets. AES is a symmetric cypher and requires the existance of a key between the participants, to guarantee the confidentiality and the integrity. The key used in AES is maintained secret. SRTP depends from an external key management to exchange a secret key and parameters to be able to use the secret key.

2.3 Key Exchange Mechanisms

SRTP does not offer a mechanism to the key exchange. To do the key exchange it is possible to use different options depending on the objective of the user. Some of this key exchange methods trust in the SIP header and their safety to perform the key exchange. In contrast, exists the ZRTP that uses the media channel to agree in a secret. The advantages are the key exchange can be performed in a peer-to-peer manner and without support of the SIP server.

2.3.1 SDES

SDES is an extension to the SDP format to allow SDP to configure SRTP sessions key. SDES specifies a new SDP attribute called *crypto*, which is used to signal and negotiate the key to be used in the SRTP sessions [13].

This can be seen as a secure way to exchange the key if two assumptions are made:

[2]The AES is a symmetric encryption protocol, this protocol was proposed to replace the SDP Security Descriptions for Media Streams (SDES)[12]

1. The SIP transport uses TLS[3], otherwise if we use TCP, the session key will be visible to the attacker, when the packets are in transit through the Internet[4];

2. The user has to trust in the SIP server, because with SDES, the SIP server has knowledge of the session key (figure 2.1). When *Alice* wants to speak with *Bob*, the *Alice* device generates a random session key to encrypt the channel. To get the key into the *Bob* device, she needs to send through the SIP signalling that can be encrypted with a TLS connection in each hop. During the hops of the connection, the SIP servers involved have access to the full *INVITE* message. This way, the SIP administrator just needs to collect the SRTP packets, and since the key will be visible in the *INVITE* message, the administrator can decipher the SRTP packets. This way, one attacker can see everything that *Alice* and *Bob* say to each other. [14]

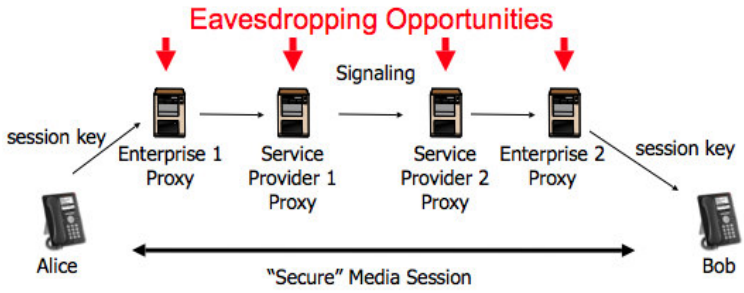


Figure 2.1: SDES eavesdropping problem[5]

2.3.2 S/MIME

Secure/Multipurpose Internet Mail Extensions (S/MIME) provides a way to send and receive secure Multipurpose Internet Mail Extensions (MIME) data. MIME is an Internet standard that extends the format of e-mail [15]. S/MIME provides authentication, message integrity and non-repudiation of the source if the system uses digital signatures. The system has to use encryption of the message if wants to provide confidentiality [16]. SIP message can include MIME bodies with S/MIME features.

[3]TLS is a protocol that wants to provide privacy and data integrity between two communicating computer applications

[4]https://www.acritelli.com/hacking-voip-decrypting-sdes-protected-srtp-phone-calls/ , [Online; Accessed 2015-12-11]

[5]http://zfoneproject.com/images/SDES_eavesdropping.png, [Online; Accessed 2015-12-11]

This solution forces the integration with PKI, otherwise the exchanged public keys would be self-signed, which makes possible the existence of a MITM attack [17].

To use S/MIME, the caller of a SIP message can encrypt parts of the SIP message, with the callee public key and include the encrypted information as an applications/pkcs-mime MIME as enveloped-data. As the SIP proxies need to access some of the header field, such as FROM and TO, we can't send only the SIP message encrypted. To solve this problem and encrypt the SIP message but still being able to allow SIP proxy to route the message from the caller to the callee, the encrypted SIP messages have two parts, as shown in figure 2.2. The none encrypted SIP message outside part of the Figure 2.2, where the headers fields needed for correctly processing and forwarding the message, are in plain text. The inner part of the SIP message, where is included the entire packet guaranty the integrity of that components of the message, and the additional headers not needed to route the message between the two endpoints and the remaining message as the SDP field [1].

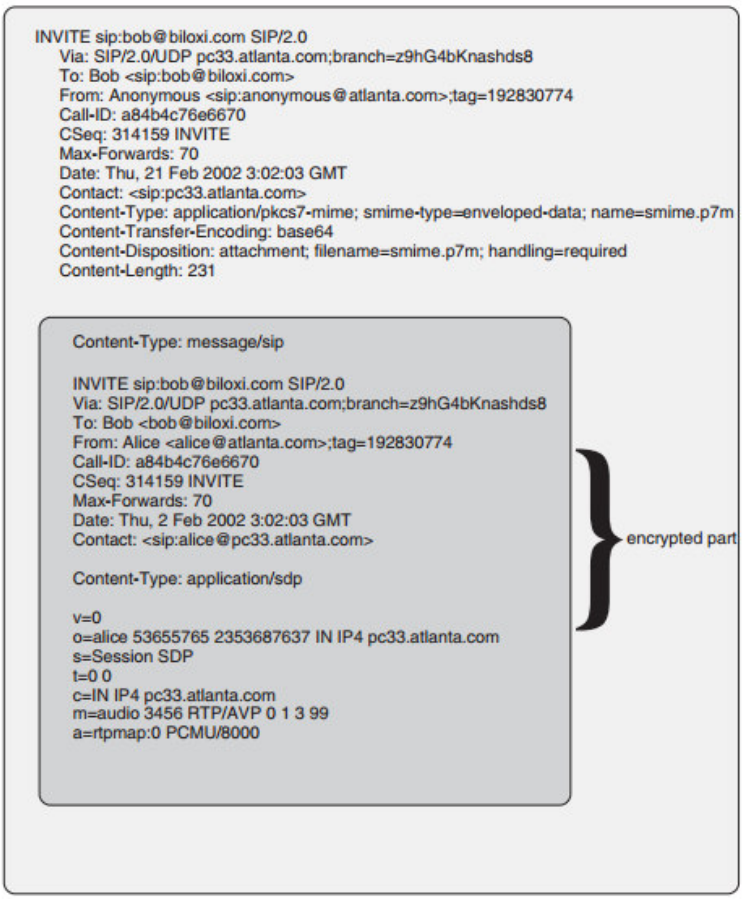


Figure 2.2: Using S/MIME to encrypt SIP messages [1]

As shown in the figure 2.2, the sender encrypts the SIP headers and SDP body. The caller can maintain, in this stage, his identity confidential, providing in the *FROM* header of the none encrypted part of the SIP message, a fake name. The callee is able to decrypt the second part of the message and read the real identity of the caller with his private key.

This way, S/MIME can provide end-to-end confidentiality and integrity of the SIP body and identity authentication to the sender of the SIP message. The SDES with S/MIME can be used to exchange key's between the caller and the callee without being the key transparent to SIP proxy in the model with TLS encryption. [1]

With S/MIME, the use of SDES can be a secure way to exchange the key to the SRTP session, because the server will not be able to get the SDP and the key from the *INVITE* message, as shown in Figure 2.1.

2.3.3 ZRTP

ZRTP is a key exchange protocol to agree on a session key and parameters to establish unicast SRTP sessions for VoIP. [7]

The principal idea of ZRTP is based on the approach to directly negotiate the shared key from endpoint to endpoint within the media channel, allowing ZRTP to work independently from SIP signalling [2].

ZRTP uses a Diffie-Hellman key exchange during the call setup, this generates a shared secret key, which is used on the SRTP session. [7]

All the devices that are capable of using ZRTP always have to compute a ZID, a random 96-bit ZRTP-ID, that is generated during the first start of the application. This ZID is the only identifier used in ZRTP to identify a different client and device. When a client needs to look up for a retained secret in the local cache, it uses the ZID. [7]

This solution follows the concept of peer-to-peer connection neither the SIP server is needed to this exchange, but it can use the SIP server to exchange some information to get a faster response for example the use of the ZID to search if contains a secret with that ZID. [7]

ZRTP uses a public key algorithm, but it does not need PKI or require certificates in end devices, it implement a different approach to detect MITM attacks. The MITM

attack can be discovered through the display of an SAS to both users. The SAS is generated based on the master key exchanged on the DH. To authenticate the SAS between both users, they should verbally compare the SAS with the media session created between them. If both SAS match the probability of exists a MITM attack is very low [7]. The SAS are destroyed in the end of the call, but if the users don't verbally compare the SAS, after the first call, the media stills with authentication against a MITM attack based on a form of key continuity, if both ends have in a previously call accepted the SAS, so after the first call the users don't need to continuously check the SAS.

To establish a multiparty secure conference, a separated ZRTP session key must be generated between each participant [18].

The protocol has three different modes to exchange the key: Diffie-Helman mode, Preshared mode and multistream mode, that is similar to the Preshared mode.

To explain how the protocol works, we can see one example of a connection between Alice and Bob, based on Figure 2.3. In this example, Alice starts the call in the Diffie-Helman mode.

1. Alice sends the *Hello Message*, where exists her ZID and other parameters such as the optional field where Alice can tell if she can make Signature of the SAS;
2. Bob to acknowledge the reception, send one *HelloACK* and after sends one *Hello* message with their information;
3. Alice will have to send the correspondent *HelloAck* after receive the Hello message. Alice, in this moment, can initiate the key agreement process, both sides have received successfully the Hello message from the other peer and the confirmation (*HelloACK*). To initiate the key agreement Alice sends the message *Commit* with the public value X without send it, this mitigate the possibility to send false ZRTP packets, section 9 of the ZRTP [7];
4. The *DHPart1* message begins the DH exchange. It is send by the responder if the *Commit* message is valid and if received by the Initiator. This message contains the out-dated hashed key material to implement the key continuity feature;
5. The *DHPart2* ends the DH exchange. Now both ends have a shared secret for the SRTP session, for that, they just need to calculate a hash of the DH-key as some other parameters. After this point the media session is secure with SRTP;

6. To finish the key agreement exists the message *Confirm1* that is sent in response to a valid *DHPart2* message after the session key is calculated. A *Confirm2* to acknowledge the reception of *Confirm1*. The *Conf2ACK* is sent to finalize the process after the *Confirm2* arrive to the destiny.

The big difference between the *Diffie-Helman* mode and the other modes, is that the device uses the local cached secrets to derive the next session secret. This way *DHPart1* and *DHPart2* messages are not exchanged, also some of the field of the *Commit* message are exchanged to identify the cached shared secret. If some error occur during this step, should be performed the DH again [1].

Users can use **Preshared mode** when both parties can skip the *Diffie-Helman* calculation. To make the **Preshared mode**, they had to establish a ZRTP media session before and exchange and verify the SAS. *Preshared mode* uses the previous shared secret to establish the next call and contains forward secrecy. Forward secrecy is a property that guarantee that if one attacker gains access to the keys present in one device, the attacker is not able to decrypt previous communications because the shared secret are replaced in each new call from the same user. So, if the attacker gains access to the cached secret and can get access to all future calls, while the Preshared mode remain to be used, the attacker can decrypt all the calls from that moment on.

The key continuity features, in another feature from ZRTP, caches some hashed key information to be used in the future call and exchange with each session. If Mallory (*the attacker*) is capable of steal shared secrets caches from one user, Mallory has only one opportunity to perform a MITM attack in the next session. If Trudy misses the interception the shared secret is updated and the opportunity to intercept all the calls from their on is lost.

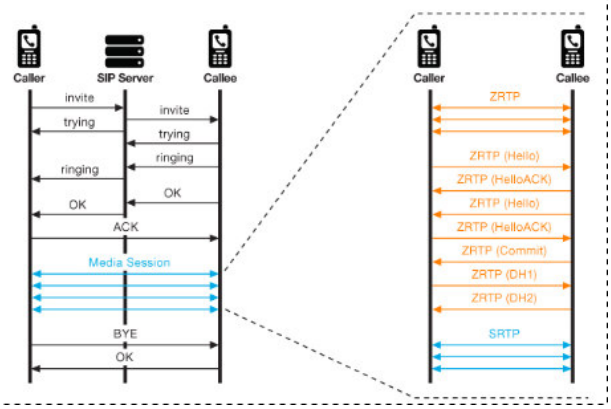


Figure 2.3: ZRTP protocol[6]

Bresciani [19] performed testes regarding the security of the ZRTP with the DH mode, and concluded that the protocol is secure if the two endpoints share the secrets needed to generate the key, and if this fails the users will always be capable of detect the MITM based on the SAS.

2.4 Normal VoIP Call

Normal call flow from VoIP communication are performed using SIP, the call architecture follows the pattern of figure 2.4, where both Alice and Bob have to get and maintains a connection with their SIP provider this communication with the SIP server, allows to get and share information with the other SIP clients. We can divide one SIP call in different stages [20]. First we need to send the *INVITE* to negotiate parameter of the call as the security mechanism. After, and in case the system uses encryption in the media session (SRTP) instead of RTP, exists the opportunity to agree in a session key, for both terminals be able to communicate. The third stage, and final, is the secure channel setup where we have the shared secret and both caller and callee can use the SRTP to send video or audio through the Internet.

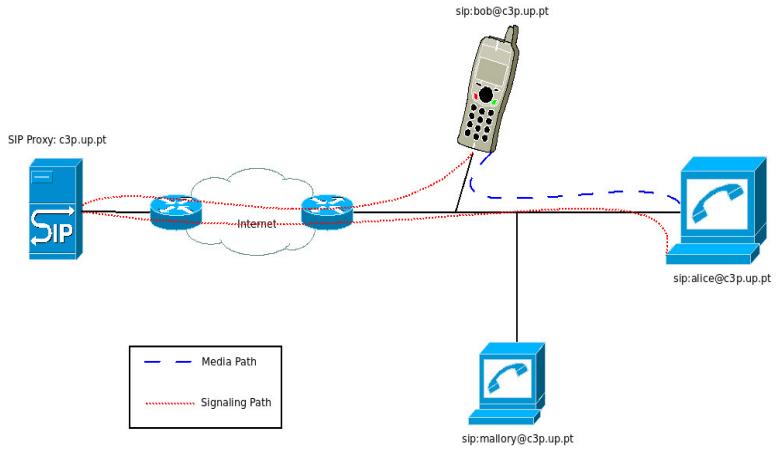


Figure 2.4: SIP signalling between and the media channel Alice and Bob

[6]http://www.itwalker3.com/wp-content/uploads/2013/02/zrtp-call-flow.png

2.5 NATO alphabet

North Atlantic Treaty Organisation (NATO) alphabet, is one widely used radio telephonic spelling alphabet. The NATO alphabet contain 26 code word, one for each of the 26 letters of the English alphabet. This way when we want to send a message *Hello* by an audio channel, and based on the table 2.1, the message to be transmitted is *Hotel Echo Lima Lima Oscar*.

Letter	*Code word*	*Letter*	*Code word*
A	Alfa	N	November
B	Bravo	O	Oscar
C	Charlie	P	Papa
D	Delta	Q	Quebec
E	Echo	R	Romeo
F	Foxtrot	S	Sierra
G	Golf	T	Tango
H	Hotel	U	Uniform
I	India	V	Victor
J	Juliett	W	Whiskey
K	Kilo	X	Xray
L	Lima	Y	Yankee
M	Mike	Z	Zulu

Table 2.1: NATO alphabet

Chapter 3

State of the art

In the following section, we present an overview of VoIP with the most known signalling protocols. We also present the SDP and the concept of presence. We present an overview about the *Android* choice.

As our goal is to create a secure VoIP application, we also present an overview about related works and an overview of the features and benefits. We also make some security considerations of the *Linphone* and *CSipSample*.

3.1 VoIP

VoIP is the technology to transmit voice packets over IP networks. This technology connects people via packet switched network instead of the traditional PSTN. As the Internet nowadays have a high bandwidth, can be performed over the Internet video calls in HD in contrast with calls in PSTN system. A connection between caller and callee is usually established using the SIP in VoIP connection [21]. The following Subsection describe some components of VoIP, as the signalling protocol SIP and opponents.

3.1.1 SIP

The SIP can be seen as a way of exchange data allowing that two or more end users can establish, modify and terminate connection between them. In one existing session SIP allow user invites other users, not present in the original call, and add or remove media

stream like video. SIP supports name mapping and redirection services, users can maintain a single identifier regardless of their network changes. These enable personal mobility that is defined as the ability of end users make and receive calls on any terminal or location, and the ability of the network to identify the same user despite the location changes. SIP has five features to establish and terminate multimedia communications:

- User location that finds the end system to be used in the communication;
- User availability i.e., provides the information if the user is willing to start a media communication;
- User capabilities that provides the determination of the media parameters, available at the user terminal such as codecs supported, and Internet speed;
- session setup that mediates the agreement in the media parameters between the users interested in a communication;
- session management, including transfer and termination of sessions, modifying session parameters and invoking services.

SIP can not be seen as a complete multimedia architecture. SIP is rather a component that can be used with other Internet Engineering Task Force (IETF) protocols to build a complete secure multimedia architecture [22]. The SIP message use SDP when caller and callee need to agree and inform some parameters, such as media encoding and transport addresses, SDP is the standard protocol to represent this data exchanges [23].

3.1.1.1 SDP

SDP, is used as one extension of the SIP that allows to both ends of a call convey media details between them. When SDP, is used to represent this information, both users can agree the media session, in the SIP signalling. This have to be performed in the SIP, because the SDP does not contain any transport protocol.

3.1.1.2 Presence

Rosenberg presented in [24], introduced in the SIP signalling protocol the options of *subscriptions* and *notifications* of presence. Presence can be seen as the willingness

of a user to enter in a voice call with another user. The state of the user can have multiple forms, traditional is just *Online* and *Offline*, but nowadays can also be *away* or *Busy* for example, and that state can include a personal message as "lunch time", to alert users that should call after the period of lunch.

3.1.2 H.323

The H.323 is a *call control* protocol, as SIP, proposed as one ITU-T standard [25] for multimedia communications over local area networks that do not guarantee packet delivery. H.323, allows real-time audio or video communication. It allows call control, multimedia/bandwidth management and interfaces between different Internet connections.

3.1.3 MGCP

Media Gateway Control Protocol (MGCP) is a protocol proposed in [26], described as a call control architecture where the call control, is outside the gateways, controlled by the Call Agents. The protocol assumes that the Call agents will synchronise with each other to send request and responses through the gateway. We can see in a MGCP environment the network gateway as a slave where the master are the call agents.

3.2 Android

We choose start by one Android application. The Android platform allow us, with a low budget, start develop and test Android application. If we start by *IOS* application we will have costs related with the license needed to work in that platform. Also if we want to develop in *IOS* and share the applications in the store, we will need to pass the United State compliance. In the Android case, we just need of one *Android* device. *Android* provides a platform called *Android Studio* that runs the applications directly to *Android* device. We do not need the compliance of the United States to publish the application, being the development of the application entirely free.

3.2.1 Problems from the operative system

In the *Android* environment, over time some changes have been made in *Marshmallow*, the differences to the previous versions of Android has increased.

In this new version, the *OpenSSL* library has changed to *BorringSSL*[1] hence the applications that use platform libraries, such as libcrypto.so, in the C or C++ will likely break. One of the examples, is the PJSIP that we used, therefore, we need to load *OpenSSL* in our application.

Another change was the introduction of runtime permissions, where some of the permissions need to be requested to the user, this way users can grant or revoke permission for the installed apps in real time, and the application should be ready to request in real time the permissions.

The biggest change for application that need a frequent connections with the Internet, i.e to receive calls, is the *Doze*[2]. Doze reduces the battery consumption reducing the background Central Processing Unit (CPU) usage and network activity to the applications when the devices are unused for long periods of time. When a user leaves a device unplugged and unused for a period of time, with the screen off, the device enters in Doze mode. Being the access to the resources made periodically, giving a brief time to the application complete their background jobs. During this maintenance windows, the system runs all pending jobs and alarms and let application access to the network.

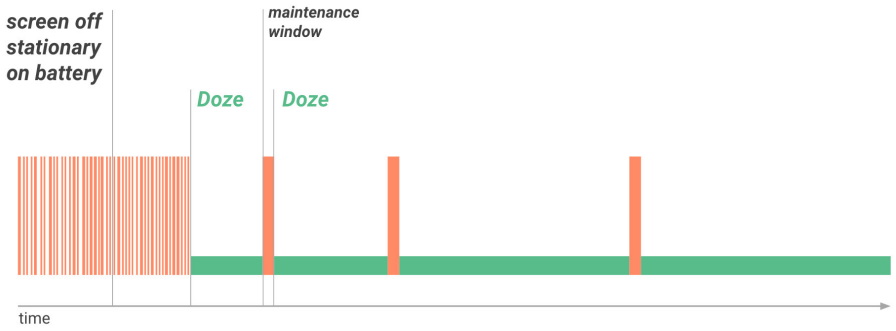


Figure 3.1: Doze mode Android[3]

[1]https://boringssl.googlesource.com/boringssl/
[2]https://developer.android.com/training/monitoring-device-state/doze-standby.html
[3]https://developer.android.com/images/training/doze.png [Online; Accessed 2015-12-17]

In Figure 3.1, is represented the *Doze* mode in green with the correspondent maintenance window at orange.

3.3 Similar applications

In this section, we present the most known VoIP applications and their SIP library.

We now proceed to describe some systems that we test along this work, to try to find the best way to implement an android application based on the stack that best fulfil our interests. We describe also some systems that we don't test but they interesting features.

3.3.1 *Linphone*

The *Linphone* is an open source VoIP application that allow to make calls with audio and/or video and text instant messaging between different clients. The architecture (figure 3.2) shows the different layers of the application and their connection.

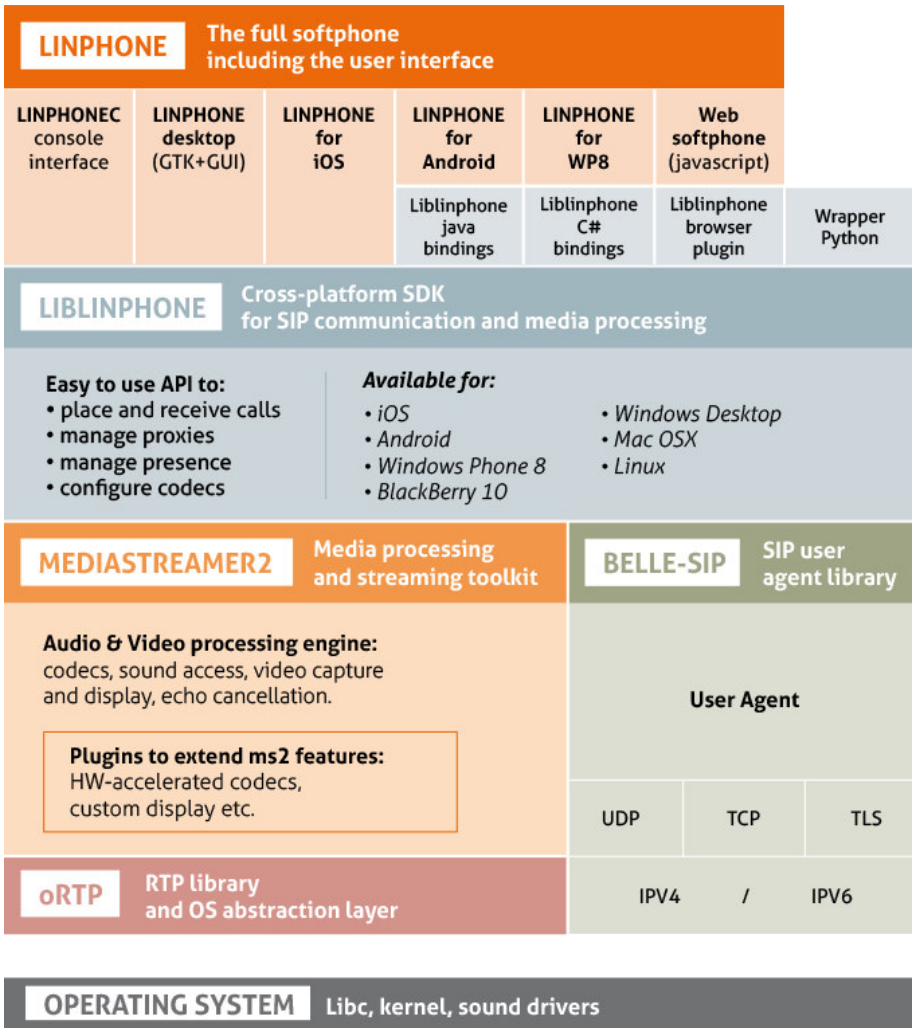


Figure 3.2: Linphone Architecture[4]

Linphone project has some features such as, multiple calls management; calls transfer, pause and resume; audio conferencing; quality of service; secure communications with ZRTP, SRTP and Datagram Transport Layer Security (DTLS). This application also has advanced features such as, HD video support; integration with push notification; ICE support; low bandwidth mode for audio calls over 2G networks; connection with TLS to the server.

We are interested in android environments, so, we try to find the features for android such as, multiple SIP accounts, ARM v4 to v7, Android 2.2 to 6.0 support. This way we can verify the possibility to run the application in multiple devices, with different processor and Android library.

[4]http://www.linphone.org/uploads/images/schema-linphone.png [Online; Accessed 2015-12-17]

The *Linphone* project have two licenses under General Public License (GPL) version 2 license or under a proprietary and commercial license if the developer needs a closed source applications.

3.3.2 *PJSIP*

PJSIP is an open source multimedia communication library written in *C* language, and is the SIP stack behind the *CSIPsimple*[5] project. The *PJSIP* stack is licensed under GPL version 2 or later and a proprietary license. *PJSIP* supports the principal Network Address Translator (NAT) protocols, Traversal Using Relays around NAT (TURN) , Session Traversal Utilities for NAT (STUN) and Interactive Connectivity Establishment (ICE). The developers from *PJSIP* and *PJMEDIA*, was designed to have very small footprint to run in devices from low capacities such as android devices. The ZRTP protocol can be implemented in *PJSIP* with the extension *ZRTP4PJ*. Despite the other examples of SIP stacks and VoIP applications, there is the need to use Simplified Wrapper and Interface Generator (SWIG) in Android environment. SWIG is responsible for connecting programs written in *C* and *C++* with the *java* classes in android case but can be implemented also with scripting languages such as python or non-scripting languages like java or C#. Use the SWIG add one more layer of complexity, normally a connection between the C and Java, in Android environment, is performed just with the use of Java Native Interface (JNI).

3.3.3 *reSIProcate*

The *reSIProcate* is an open source project that contains three different basic modules: Dialog User Manager (DUM), stack and *recon*.

The stack is responsible for the low level functionality such as: message parsing, message synthesis and transaction handling. DUM is above the stack and provides user agent functionality such as handling the registration process. Conversation Manager (Recon) is a user agent with media support above DUM it uses sipXtapi project to convert to RTP packet, and is responsible for processing incoming RTP packets to and from the reflow layer.

The reflow is the way to send and receive packets through the Internet. The principal

[5]https://code.google.com/p/csipsimple/ , [Online; Accessed 2015-06-19]

difference between the *reSIProcate* and the other projects is the support to S/MIME. [6] When we try to use the *reSIProcate* we stop in the stage of compiling *sipXtapi*, finding that the project in the future will try to leave the *sipXtapi* otherwise it is impossible to work with *reSIProcate* to implement an android app without leaving the *sipXtapi*.

3.3.4 *Silent Phone*

Silent Phone is an application for *Android* and *IOS* developed by *Silent Circle*. *Silent Circle* is a company founded by *Phil Zimmermann* the creator from ZRTP protocol, so Silent Phone uses ZRTP to exchange the key between the caller and the callee [7]. Despite the previous examples, this application is not available under any open source license. But the code is available for peer review in github [8]. Based on the code provided for peer review, we analyse some of the features of the Silent Phone such as the presence, STUN, TURN, ICE and allows to send the SIP signalling over TLS.

3.3.5 *PrivateWave*

PrivateWave[9], is a multiplatform application and is based in the *PJSIP* stack. The application use the ZRTP as the key agreement protocol, and are the sponsor of the *ZORG*[10] project, the ZRTP implementation that they use in *PJSIP*. The only information, available from this application is that also support SIP and TLS.[11].

3.3.6 *Whatsapp*

Whatsapp[12], is the most recognised and deployed application for secure communication over the *Internet*. The only difference between this application and the previous one, is that this system is a closed system. Closed system means that this application is only capable of communicate with another application of the same type, instead of for

[6] https://www.resiprocate.org/Get_Started [Online; Accessed 2016-6-17]

[7] https://support.silentcircle.com/customer/en/portal/articles/2118686-what-is-silent-phone- [Online; Accessed 2016-6-17]

[8] https://github.com/SilentCircle/silent-phone-android [Online; Accessed 2016-6-17]

[9] https://www.privatewave.com/ [Online; Accessed 2016-6-17]

[10] http://support.privatewave.com/docs/display/KB/Support [Online; Accessed 2016-6-17]

[11] https://play.google.com/store/apps/details?id=com.privategsm.beta [Online; Accessed 2016-6-17]

[12] https://www.whatsapp.com/ [Online; Accessed 2016-6-17]

example *Linphone* and *CSipSample* that can communicate with each other without major efforts. Their clients can have different providers that can also communicate. With *Whatsapp* the same does not happen, we can just communicate with client that use the same software and they have to mandatorily use their signalling and media services. Reasons for this, can be the creation of a robust system, capable of control all the user activity and also the implementations of cryptographic protocol with all the needs to block eavesdroppers.

3.3.7 Comparison

The table 3.1 demonstrates the most relevant features of a VoIP application and for each application/stack which features has. In this comparison, we do not compare *PrivateWave* and *Whatsapp*, as the code is not available for peer review.

Table 3.1: Comparison of VoIP systems

	Linphone	*CSipSimple*	*reSIProcate*	*Silent Phone*
Open Source	✓	✓	✓	X
TLS	✓	✓	✓	✓
ZRTP	✓	✓	X	✓
SRTP	✓	✓	✓	✓
DTLS	✓	X	✓	X
S/MIME	X	X	✓	X
STUN	✓	✓	✓	✓
TURN	✓	✓	✓	X
ICE	✓	✓	X	✓
SWIG	X	✓	X	-
Presence	X	✓	X	✓
Structured in layers	✓	✓	✓	-
audio	✓	✓	✓	✓
video	✓	✓	✓	✓
instant messaging	✓	✓	✓	✓

3.3.8 Security of *Linphone* and *CSipSimple*

During the start of the project, we tested the most known application open source in this case *Linphone* and *CSipSimple*, and their support to ZRTP.

When we launch both applications, we noticed that *Linphone* sets immediately the type of connection with the SIP service as TLS. But regarding the media channel established between both ends, the same does not happen being the encription disabled, in this scenario the media encryption can use SRTP, ZRTP or None. In this case, we can assume that is a simplification of the User Interface (UI), where they state ZRTP as a media encryption protocol, for example, SRTP, and not a key exchange mechanism. But the media encryption by default is None, meaning that the audio can be listen by someone on the network.

In the *CSipSimple*, all the encryption available are disable, the default connection with the server is TCP, media encryption is deactivated and the ZRTP mode is as a consequence disabled also.

This way, the use of this application by a normal user, without the knowledge of the risk that VoIP communication contains, and without the security mechanism available to the VoIP communications, can become a case study for the company (i.e. analyse the data exchanged during a call for advertisement). Companies that want to take advantages of such clients are capable of recording all the communications, or almost all the calls, for example if they use ICE, it is normal that the packets pass by a company server and this way they can collect and search in all the voice calls that the users perform.

3.4 Differences between application

With all these different apps, we can think that we accomplish the secure communication system, but that is not the true. From the multiple application that we presented the most secure application appear to be the *Silent phone* and *Whatsapp*. As presented in subsection 3.3.8, the other applications, can also be secure, but they need some interaction and knowledge from the user to accomplish that execution modes.

But we want to create a robust client that can communicate with other application such as *Silent phone*. Also add some features regarding the data stored in the Android devices. None of this clients is robust regarding the Android devices. When we test these systems, the data is in clear text, and the application contain a fully access to credentials and data stored in the device by analysing the permission of the Android application. We know, that this way, one intruder can collect data regarding the owner

of the device.

Chapter 4

Implementation

In this chapter, we describe the main choices we have made for developing our Android apps. We start by enumerating the changes we had to make in *Linphone* to produce our first application. For the second app we managed to reduced the code-base complexity by substantially reducing the app feature-set and basing it on the popular and well supported open-source library *PJSIP*. This allowed us to produce a simpler and more secure app, capable of working for much longer periods of time by employing battery saving strategies on Android. All of this was in accordance with the goals enumerated in 1.2. The applications that we have developed are called *Murmur*.

4.1 Implementation scheme

The principal objective is to create an Android application that is capable of making and receiving calls securely. To accomplish this, we have developed two different Android apps.

We started by developing an application based on the source-code of the well known *Linphone* application. In this application, we focus in the reduce of the code complexity. The application contains some features we need and we also managed to reduce the complexity of the original UI. We have analysed the major libraries employed by *Linphone* and concluded that its implementation of ZRTP was hard to modify and made it very difficult to implement the security features we needed.

We also concluded that the library violates the "Limited functionality" principal, that we described in the subsection 1.2 and that we want to maintain in our application,

such as, the limited functionality. Offering limited features to reduce the attack surface.

So, during the development of the application, we made the decision to change to a simpler more flexible supporting library set. This decision allowed us to change to a new ZRTP library. We have tried two different libraries: the *reSIProcate* and *PJSIP*. We choose the *PJSIP* library because it contains the same ZRTP library of the *Phil Zimmermann* (the ZRTP inventor) application and contains all the specifications of its RFC.

To create our application, we need to implement all methods to communicate with *PJSIP* and we build from the *PJSIP* stack up until the UI, to get a full capable application, in this version we reuse the interfaces from the first version.

To perform this SIP compliant application we use the Open Source SIP Server to test the application, in this case the kamailio[1].

4.1.1 Linphone databases

When we start to reduce the complexity the code, we find that Linphone stores every data in a *sqlite* database (DB), but the access are made from multiple places. The things related with the message can be accessed from the *C* code and in *Java* code but from different tables, so, the Linphone maintains two data bases with the same content. The call logs are stored in C and when we need to access some information to display to the user we have to pass through the JNI layer to get that information, that is constantly used by the user interface. The Linphone also contains information about the presence[2] of friends, but at least for the Android platform does not appear to exists an implementation that uses that functionality. The *Liblinphone* already contains DB to implement this feature. All the information related with the contact list, such as sip address or phone contact, is stored using the Contacts Provider. The Contacts Provider is the device central repository of data about people, containing all the contact information stored in the list of contacts from the device. This central information is accessed from the *Java* part of the code, and, in this repository, the user adds new contacts and, from that moment on, the contact becomes visible in all application that use this provider.

[1] https://www.kamailio.org/w/ [Online; Accessed 2016-6-17]

[2] http://www.linphone.org/docs/liblinphone/group__buddy__list.html [Online; Accessed 2016-6-17]

The use of the Contacts provider allows an opportunity of making a call to the PSTN by the network, when the Linphone reads one contact from the PSTN network and displays on the user interface, if the user wants to call to that number, the application will try to fit the number followed of host of the user. For example the user wants to call to the number 912345678 and is registered in the application with Uniform Resource Identifier (URI) *sip:someuser@c3p.pt*, the application will try to call to *sip:912345678@c3p.pt*, this can be a bug from the application or a feature, where the user can get free or cheaper calls, for that the SIP provider just need to allow this calls. However, as our focus is the secure calls and we don't want to use the PSTN environment to perform calls, in this case this is a call from a SIP provider to the PSTN. We do not want this features, at most the communication will be cyphered up to the SIP server, but it will be impossible to carry a SAS verification.

The *Linphone* has the messages in the *Java* side, this way, we remove the DB of the messages, because as we want to accomplish a secure application, we want start by getting all the databases in the *Java* side allowing to do not need to access to pas the JNI layer to get information, just relevant for the UI, this way, we can manipulate all the databases in the same place with this approach we can simplify the process of cypher and decipher data we want to perform this process in a next phase. We also remove all the code associated to this (read and write messages). The database with the history of calls, is located in the *libLinphone*, that is the library of *Linphone*, so in a first stage, we move the DB to the Java side, and pass to access to all the information in the UI.

Another thing that we removed completely, was the use of the Contact provider, passing to store all the information in a data base of the applications, and this way remove the possibility of other application read our private data.

With this process we want to guarantee the principal of the Limited functionality and in the future guarantee the local security of the data.

4.1.2 Interface

In the first version, we choose to use the entire application. The *Linphone* already includes the entire flow of a call and a UI well defined. But the UI, in some menus, contains more information than we need. Based on this, we remove some of the features that the UI allows and we add another features that we think that can be relevant to the user experience, and to the simplicity of process.

Based on this, and as we can see in figures 4.1 and 4.2, we decide to remove the entire bottom layer of Menu that allows to pass between all the menus. We also reduce the number of menus, passing the number of menus that the user can use to only 3 instead of the previous 5. This way, this allow user to slide between menu. Also, we remove the possibility of having the SIP contacts, and PSTN contacts as shown in both Figures. We add in this menu, Figure 4.1, the option of directly call when the user presses the icon of the phone instead of having to pass to a new menu, as happen in Linphone, Figure 4.3, to give the user the opportunity to perform the call immediately, instead of having to pass throw menus.

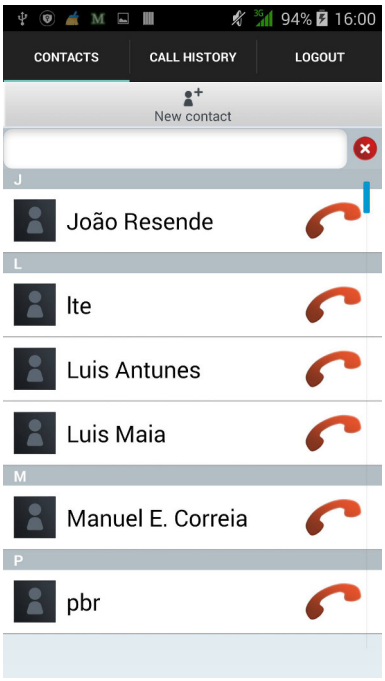


Figure 4.1: Murmur contacts menu

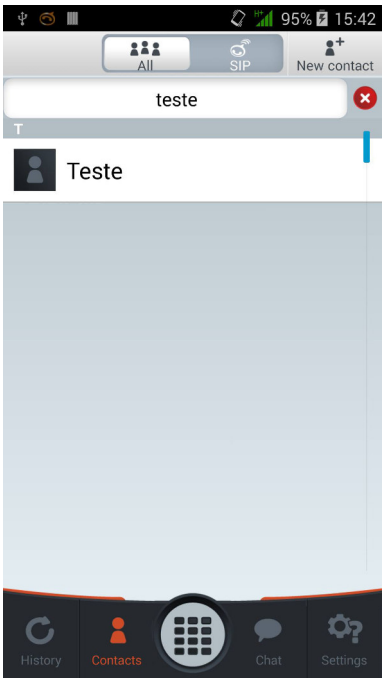


Figure 4.2: Linphone contact menu

In this application, we want to create the option to a group of contacts, where a user can have a group of friends, this friends can be defined by the SIP provider. We can see this, for example, in a company that uses our application and has a private SIP provider, can immediately and continuously update and set all the contacts from the employees, without any effort. This way, when a new employee is hired and have the application, is able to call to the accounting department to ask when we will be paid out.

This way, a user will have two different behaviour in the Menu, for example, in the Figure 4.4, the user João Resende is from the group of the current user. The user can't

use the button edit or delete, because they are disable. If the contact had been added by the user, the user will see the buttons darker, as in Figure 4.3, in this case from Linphone, but with the same type of buttons on top. The contact from the PSTN as completely removed to do not support the possibility to call to a number from that network.

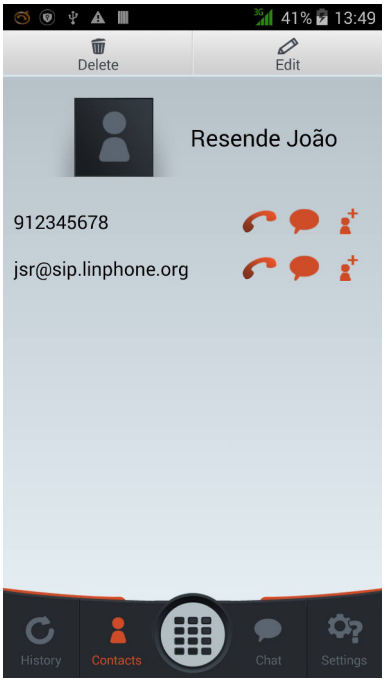


Figure 4.3: Contact Menu Linphone

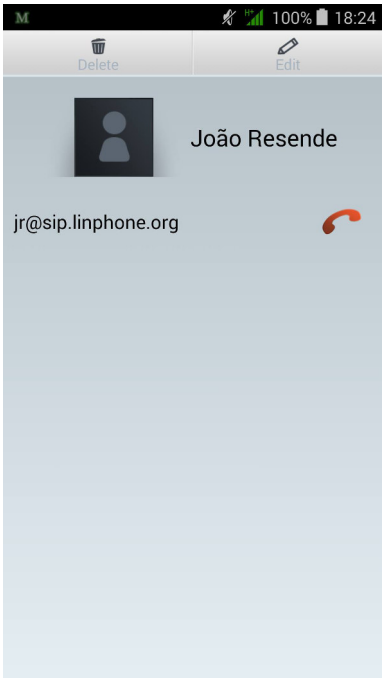


Figure 4.4: Contact menu Murmur

Previously, we mention the removal of some menus. We completely remove the dialler, Figure 4.7, from the application, because as this will be a closed service is not normal a person call just one time for one person and that person is not in their list.

Another menu that we remove was the option of receiving messages or send, that is represented in Figure 4.6, as we don't pretend to give the user the possibility of sending messages.

The definition menu was erased, Figure 4.5, because we implemented a solution with QR codes, to simplify the user experience, presented in subsection 4.1.3, and this way, the menu is not needed, because a user should have the default definition, to communicate with their SIP contacts.

With this replacement, we also remove other Activity/menu, but as they make part of the normal flow of menu that we remove, they become unused menus.

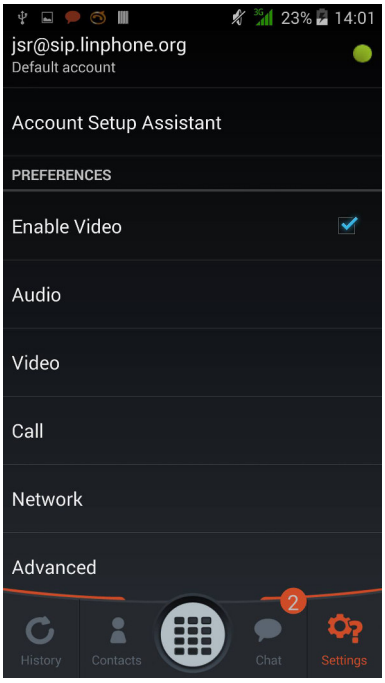


Figure 4.5: Options Menu Linphone

Figure 4.6: Messages menu Murmur

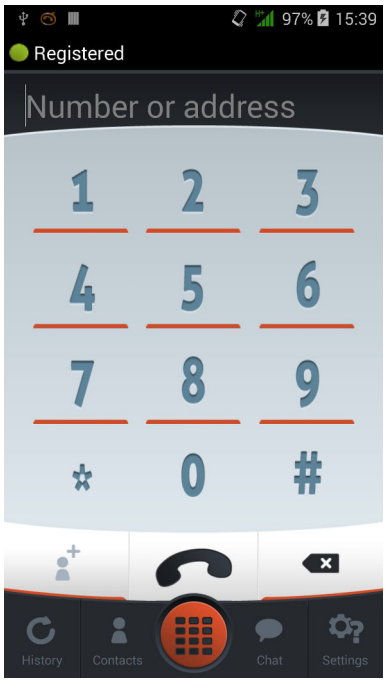


Figure 4.7: Dialer menu Linphone

In the chapter 2, we mention the SAS of the ZRTP, as the way that the ZRTP found to eliminate the MITM attack, possible in the DH algorithm used in ZRTP.

With this approach, and assuming that the user when uses *Linphone*, knows how to add the key exchange to be performed with ZRTP, we have an icon that alerts to this SAS exchange. The icon is in top right corner (figure 4.9). If the user press the button gets the SAS. Also, will appear the Accept Deny option as shown in Figure 4.10. After this, the user can compare the SAS, and see if it matches or not.

In figure 4.8, is represented the case after the user has performed the SAS comparison,in this case, the icon shows a poor feedback. If the user performs a relevant number of calls per day, the possibility of do not check the icon to see if the call is established with ZRTP will decrease.

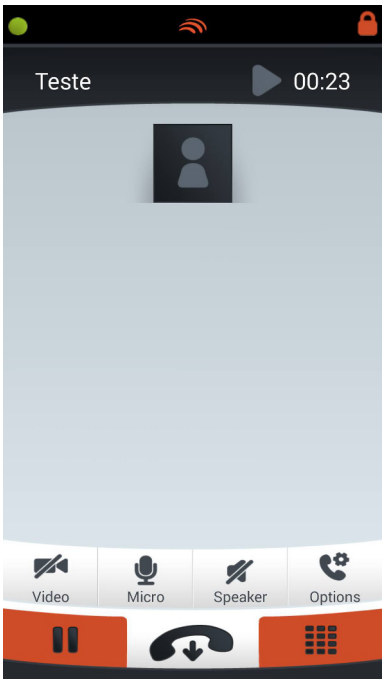


Figure 4.8: Linphone secured call

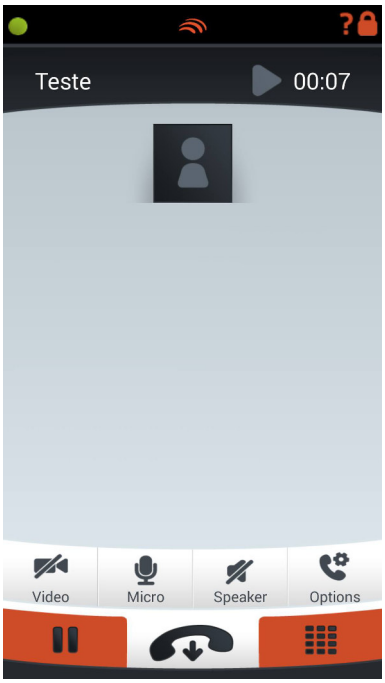


Figure 4.9: Linphone, SAS needs verification

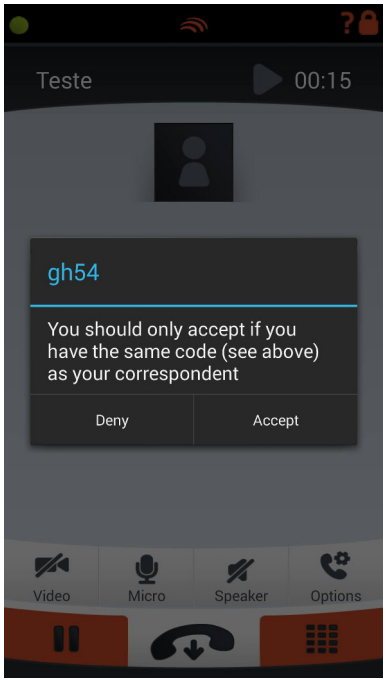


Figure 4.10: SAS for the call, and the accept, deny SAS march

As we think that the SAS should be always compared, and at any point, the SAS should be visible, we decide to implement one interface where this happens.

When the user receives or executes the first call, users will see an interface as the Figure 4.11, where the user, if wants to hung up or stop the audio transmissions, will always accept or reject the SAS. This way, the user always exchanges the SAS. Another features that we add is the play of a message to alert the user, to perform this comparison. In this case, the user listen "Warning verify codes". When the user is already in a secure channel, after in the previous call have exchanged the SAS, will get an interface like the figure 4.12, where they have the SAS comparison and a lock to confer the insurance idea, and will be played a message telling *"secure"*.

When the other peer, for some reason, does not support the ZRTP, we allow the media channel to work, but in this case, will be displayed a locker without the SAS on top (Figure 4.13). We do not block RTP packets, because the RTP packets are the packets used to exchange the messages from ZRTP, because in the begin, they can not use SRTP, as they don't have a secret key between them. As this is a transition the ZRTP can be exchanged at any moment, in this case, we don't display any message, because the ZRTP can be performed in a near future, because after the audio starts we do not control the ZRTP exchange.

Figure 4.11: Murmur first call

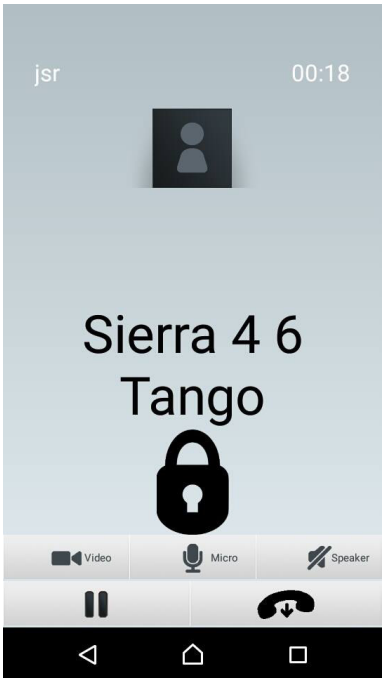


Figure 4.12: Murmur after accept the call

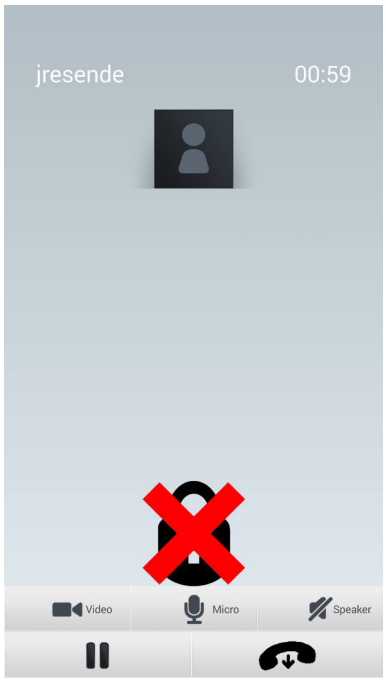


Figure 4.13: Murmur while don't perform SAS exchange, or without encryption

Another feature, is the use of the NATO alphabet to represent the SAS, as shown in Figure 4.12 and 4.11.

4.1.3 Provision with QR Code

When a user needs to authenticate in any VoIP service, he need in the first place, to introduce the URI and password, that will authenticate. Users usually, have to type directly into the application interface, as shown in Figure 4.14. The authentication is one of the first things that users must do, but exists other things to be configured by the users, as the use of STUN, TURN, ICE or TLS support. This features, normally, add a lot of complexity to the UI, with buttons, check boxes, and a different configurations for each customer or service.

To simplify the application configuration, we want to give to the user a faster process of authentication and definition of options. For that, we want to use a QR code. The QR code will contain all the information, such as the user or password, needed for the configuration. This way, the customer device will be ready in seconds, saving customer time and giving a better user experience, with a valid installation in the device, with only a use of a QR code.

To use the QR, to the configuration, we will need to use a device with a Camera, nowadays all the smartphones of the market contain one or two cameras. With the camera, the user can point the QR code to the app, and the application will detect the QR code as shown in Figure 4.15.

With the need of using the Camera, we need to use a QR code reader, for that we could use a specific android application, such as *Barcode Scanner*[3], or one library such as *ZBAR*[4]. We choose to integrate with an open-source library. With this library, we gain an important feature, the independence of other applications. If we choose to integrate with another application, the customer, when using the application will need to have in the device another application, that can collect data from the device, and as one of our objectives is to validate the application by a third party, force the user to use a none authenticated applications is not the best solution compared to the use of the open source library.

ZBAR library gives the information contained in a QR code. To use this library we need to pass one object of type Image[5], this way the *ZBAR*, checks if exists a QR code, and performs the decode of the QR code, passing this information to the UI.

[3]https://play.google.com/store/apps/details?id=com.google.zxing.client.android [Online; Accessed 2016-6-17]

[4]http://zbar.sourceforge.net/ , [Online; Accessed 2016-6-17]

[5]https://developer.android.com/reference/android/media/Image.html, [Online; Accessed 2016-6-17]

One of the problems, that we can think regarding the use of this method, is the need of sending the credentials password included, to the administrator of the SIP server to be performed one password exchanged. For that, we want to create a portal where a user can introduce a new password and download the new QR code, to destroy the previous one, without the need of sending this information to the administrator.

The QR code is generated with one order defined, as shown one example in Figure 4.16. All the fields are separated by a ";", the fields start with the username followed by the password, host of SIP server and the type of connection TCP or TLS. After that, if we need a STUN server, we should write STUN followed by the IP, and the same with the other option without any defined order in this part.

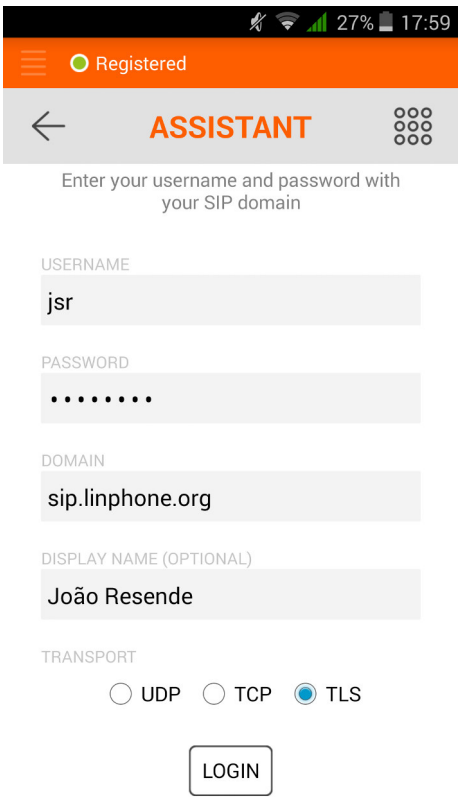


Figure 4.14: Linphone configure SIP account

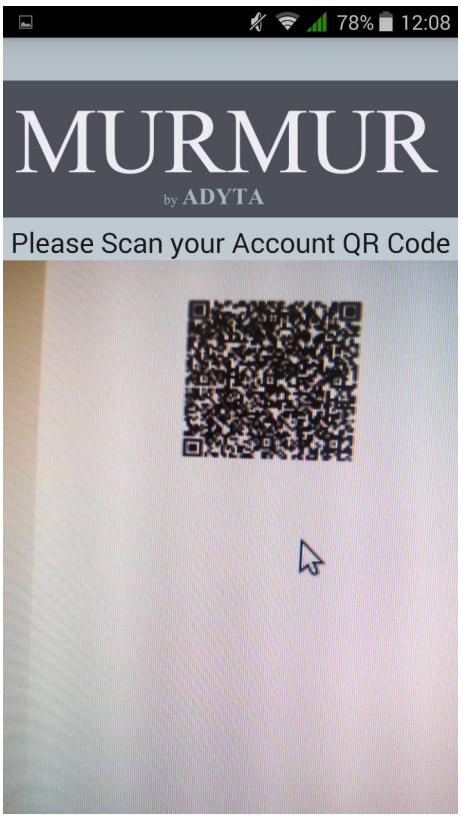


Figure 4.15: Murmur configure SIP account

Figure 4.16: QR code example to pass to the application

4.2 Second version - PJSIP

4.2.1 Why change to a new implementation

In the development of the first version, we always try to keep the principal defined in the beginning of the project. With that, continue with the use of *Linphone*, will be against the the third principal "Limited functionality", where we will not be capable of reducing the capabilities to limited amount necessary to perform a SIP call. This happens based on the years of cross-platform development, and the demand by the users and developers to support new features, ending with one application, with a lot of dependencies, making the code hard to compile, understand and making the validating process more difficult.

Also during the process with the *Linphone* library, we notice that something are not well implemented in the *bzrtp*[6]: the ZRTP implementation of the *Linphone* library.

To show that the things that are not well implemented, we use our application to show it. As we have a popup to alert the user to validate the key, our application contains a more visible problem to demonstrate the problem. In the figure 4.17, we have a *BQ™* smartphone running out application, and in Figure 4.18 a *Sony™* smarthphone running the same application with the *Linphone* library.

When both of them are in a call for the first time, they have to validate the SAS,

[6]https://github.com/BelledonneCommunications/bzrtp , [Online; Accessed 2016-6-17]

but let's imagine that the user in the BQ^{TM} device rejects the SAS and the $Sony^{TM}$ user accepts it. In a future call (Figures 4.19 and 4.20), both users will have different notifications. The BQ^{TM} user as rejected the SAS in a previous call, the application will still request the exchange of the SAS, but the $Sony^{TM}$ user, will get a different notifications, for him the call will be encrypted and the key continuity feature will be working.

Both users should get the same behaviour, from the application, this behaviour should be independent from the decision taken regarding the SAS exchange. Generating a new SAS should be the correct decision, because with this the application do not run the DH key exchange algorithm again.

Figure 4.17: Caller, in the first call

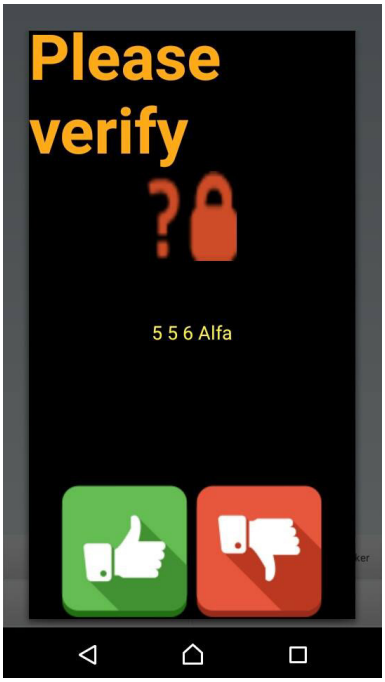


Figure 4.18: Callee, in the first call

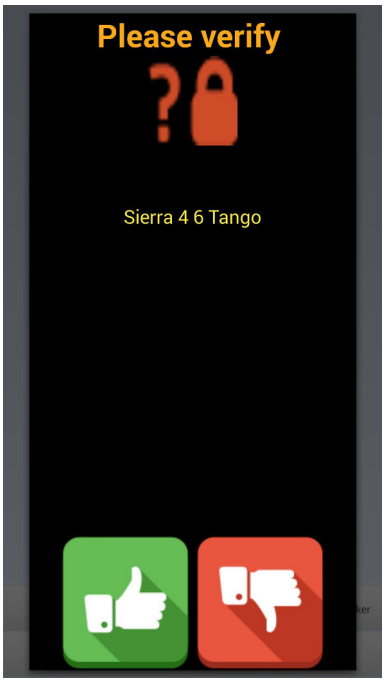


Figure 4.19: Caller, in the second call with Linphone

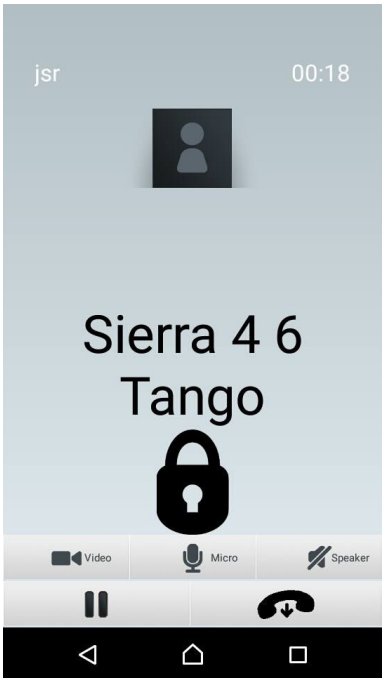


Figure 4.20: Callee, in the second call with Linphone

The same problem does not appear in the *PJSIP* library, as we can see in Figures 4.21 and 4.22. In this case is the application that we develop with the *PJSIP* library and with the same process described in Figure 4.14 and 4.18.

Figure 4.21: Caller, in the second call with PJSIP

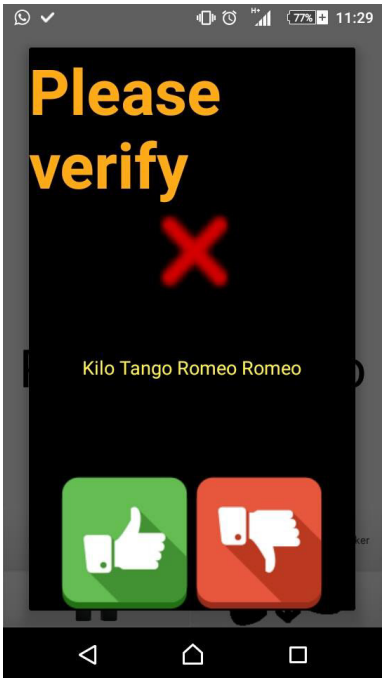


Figure 4.22: Callee, in the second call with PJSIP

This different implementation is caused by a different ZRTP system. The *ZRTP4PJ* that is the ZRTP implementations of the *PJSIP*, implements the SAS Verified Flag (chapter 7.1 of the ZRTP draft [7]). In this chapter, they present this flag, that indicates when the comparison has been made with success in a previous call. The SAS Verified flag is exchanged in the *Confirm1* and *Confirm2* messages of the ZRTP section 2.3.3. This way, if an user, in a future session wants, to remove the verified flag from a call, it becomes possible, for that the user just have to change in the UI the flag to rejected. In the RFC from ZRTP [7], the authors claim that is normal one endpoint that never sets the verified flag, because this flag requires adding complexity to the application. However as we want one application that contains all the best practises/implementation regarding the security of the system, this is one important feature, because gives a better user experience and when the application request a new SAS will alert both users.

During tests, we also find the none existence of some important parts of the ZRTP, as the chapter 7.3 of the ZRTP draft [7], where the relaying of the SAS through a server is not possible in this implementation, the missing of the zrtp-hash attribute in SDP, to allow to identify the user, the alert message to pass to send the packets in clear, without the implementation of the *Go Clear* and *Clear ACK*, and the SAS signing

implemented in the *ZRTP4PJ*, but not in the *Linphone*.

Based on this, we change from library from *LibLinphone* to *PJSIP*, because some of the none implemented features for us are important as the Verified flag. In the future we may want to give the user the opportunity to sign, the SAS with certificates, so this exchange may allow a faster implementation of new features.

We could always try to implement the ZRTP but, during the development, as shown previously, we find some problems regarding the compilation of the *Liblinphone*, with a lot of dependencies, and with broken codecs, or version. This way we decide that spend time trying to implement the *ZRTPCPP* for the *Linphone* that will cause a slow down in the process in the future, and will get problems when we, in the future need to upgrade some version or validate the code. The most reasonable idea to perform this implementations, based on the principal of the application, is to exchange from SIP library.

During the change from library, we considered change to a different protocol we think in some solution such as S/MIME, present in *reSIProcate* library, and afterwards implement the ZRTP using the *ZRTPCPP* adapted to the *resiprocate* stack. We decide to use the ZRTP, but, we want in the future give, support to S/MIME. This is one of our goals as future work.

With this platform we were able to send messages using the *Recon*[7] library from *resiprocate*. In this library to use the media session is used the *sipXtapi*. When we try to compile and use, we find that the project is deprecated, as shown with the extinction of the website. So we let this idea, the implement of all the audio transfer and different audio codecs will be a bad decision, regarding time and effort, with the other library full capable with audio included.

4.2.2 Add the ZRTP

In the *Linphone*, the ZRTP is already implemented. The same does not happen with the *PJSIP* stack. In this library, we need to add the ZRTP implementations, for that, we need to choose between two ZRTP implementations, the ZORG[8] and *ZRTP4PJ*[9]. We choose the *ZRTP4PJ*, because the implementation is used in *CSipSimple* and in

[7] https://www.resiprocate.org/Recon_Overview , [Online; Accessed 2016-6-17]

[8] http://support.privatewave.com/docs/display/ZORG/ZORG+HOME , [Online; Accessed 2016-6-17]

[9] https://github.com/wernerd/ZRTP4PJ , [Online; Accessed 2016-6-17]

Silent-Phone.

With *ZRTP4PJ*, after compile, we need to add the patches to the code. For that, we need to change the *on_ create_ media_ transport* function from the *endpoint.cpp* as shown in Appendix B.1

4.2.3 Integrate PJSIP in Android

In opposite of the first version of our application, that we just simplify the code of the *Linphone* application, and it add some of features, this version has developed from the stack to simplify all the communication between the UI and the *PJSIP* library.

To run the *PJSIP*, we decide connect with a Service. This way, we are able to connect with the SIP server in background. Another feature that we need is the *Wakelock* this guarantee the keep of the CPU on and, this way, is capable of receiving calls, and refresh the connection with the SIP provider.

Based in Figure 4.23, we can see that to connect the Service with UI, we need of two different connections. To send message from the UI to the Service, we use AIDL. AIDL is a way to define the programming interface that both the client and service agree, in order to communicate with each other[10].

To communicate from the Service to the Activity, we start by implement with Local-BroadcastManager. This broadcast allows to communicate to any of the Activity that the user is currently using. The only disadvantage of the *LocalBroadcastManager* is that the Service and the UI, have to stay with the same Process ID (PID), but this way the other processes can not listen the messages sent. When the Service wants to notify the call state changed, the service can send through a Broadcast message. This way, the Activity can listen that message and give feedback in the UI. With this implementation, we conclude that is not our best solution to fulfil our goals, because in a some cases, as when the user receives a call, when the device is ringing, if the user that receives the call answers and the other one at the same time, quit of trying to call, in the period between the transaction of menu in the UI, to a call menu, the application will get a few seconds here is not listening the Broadcast messages. With this we will lost information and spend more memory, because we will need to have different *LocalBroadcastManager* for each different activity, and for different type of message, from the call state to the login refresh message.

[10]https://developer.android.com/guide/components/aidl.html, [Online; Accessed 2016-6-17]

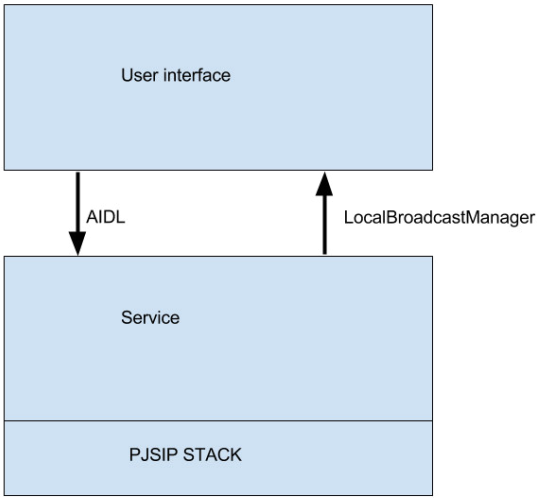


Figure 4.23: First structure, with *PJSIP* library

To solve this issue, and knowing that the Service and UI share the same process, the *Service* thread can communicate with the UI. To communicate the Service just needs to find the object of the UI, and then can send the information. This way if we always enter from the principal activity, we will be capable of determinate the current interface.

During the development of the application, we find some problems regarding the *PJSIP* stack. This stack, when finds a condition that fails, in some cases uses the assert command. The assert command, aborts the program if the assertion is false. In a first analyses, we decide to remove the assert and let the code flow as normal, but during this process we conclude, that this is not a good idea, because in some cases, the function that return the error, should return pointer or structures and, this way, will fail in something else, or have undefined behaviour.

To mitigate the kill with the assert, we need to separate both threads, UI and Service, getting two different process for each one. This way, when the service crashes, can send a message to the UI, and the UI will be capable of detect and restart the service. With this the application will not loose any information such as the credentials used to authenticate in the device as described in Subsection 4.2.4. To be able to split, in two different process, we have two different option, using messages or AIDL. In

this case, we choose the AIDL, because we can create one Application Programming Interface (API), to connect the service to the activity, and the activity to the service. This way, both process can communicate with a well established API, as shown in Figure 4.24, where the Service just have to invoke a callback to perform an action in the UI.

With this separation in two process, we will gain in the lifetime of the application and performance. The performance will increase because the UI will run separated from the Service. Both process will have a different heap, so in a low resources device, the application will get a better performance.

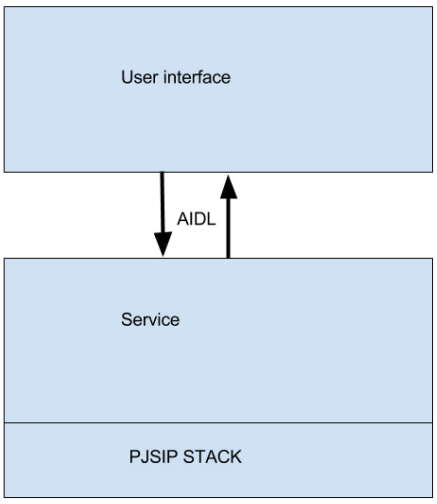


Figure 4.24: Final version of the system with *PJSIP* library

4.2.4 Authentication system

In this subsection, we describe the security features implemented in the Android device to ensure the confidentiality, integrity and authenticity of the data stored, i.e contacts, last calls and credentials to the SIP server, presented in the section 2.1.2.

For that, we developed two different authentication systems.

The first one is a system, as the Figures 4.25 and 4.26 represent. The Figure 4.25 represents the first login attempt where the user insert a password in the UI and this

password is validated. To be a valid password just need to contain more than four characters, after this, we can pass to the Password-Based Key Derivation Function 2 (PBKDF2)[11], in this step we want to get a secret and to accomplish that we need to pass the number of iteration in our system, 4096, and a salt with 32 character, generated by the *SecureRandom*[12]. When we get the secret, we need to see if this is the first login or not. If is the first login (Figure 4.25) we use a random string, acquired as the salt, and we stored in *SharedPreferences*, after this we will need to cypher this random string with the output of the PBKDF2, and stored in the same place. If this is not the first login, Figure 4.26, we just need to get the random string stored in the *SharedPreferences* and the cypher version of the string, and decipher the string with the secret from the PBKDF2. If the random string matches with the deciphered string, the user is authenticated. With this architecture to authenticate a user, we can suffer a brute force attack, because we have the start point and the end point, so we can try the most common password, and not all the users will insert into the password, a complicated password or too long so will not be very complicated the brute force attack. With this architecture, we also have to store the credentials in the *SharedPreferences*.

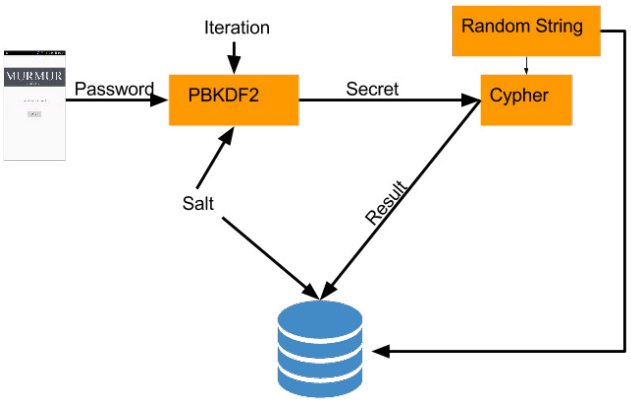


Figure 4.25: First use of the system - model A

[11]PBKFD2 is used to derive encryption keys from a initial password[27]

[12]https://developer.android.com/reference/java/security/SecureRandom.html , [Online; Accessed 2016-6-17]

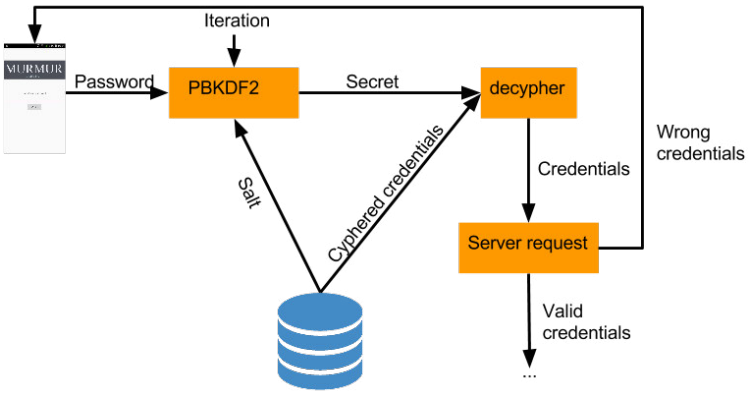


Figure 4.26: All the login attempts after the first one - model A

To solve this possible attack, we create a different approach where the server, performs a key feature of the authentication system. In this system, the user will need to add a password to use in the PBKDF2, with the same iteration and the same type of generation of the salt. But in this version (Figures 4.27 and 4.28), the process is changed after the PBKDF2. After we get the secret from the PBKDF2 in the first login (Figure 4.27), we will need the credentials to the server, and is that information, with the following structure user;password;domain, that we will cypher and store in the *SharedPreferences*. In the second login, Figure 4.28, the program needs to decipher the SharedPreferences with the secret generated by the PBKDF2, with that output (the credentials from server) he try to connect with the SIP server, if the SIP server response with one accepted of the credentials, the user is authenticated, otherwise will need to try again. This mitigates the possibility of the attack, because the attacker do not known the credentials of the user, just knows the end from of the cypher data, and will have to try to guess the form of the authentications credentials, and always that the attacker try to connect with a remote server we can delay the response time, this way we can delay a brute force attack.

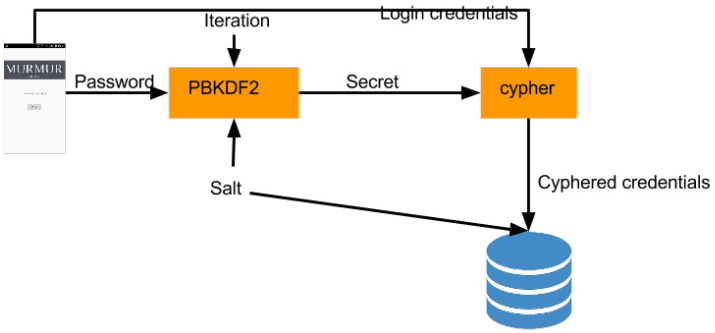


Figure 4.27: First use of the system - model B

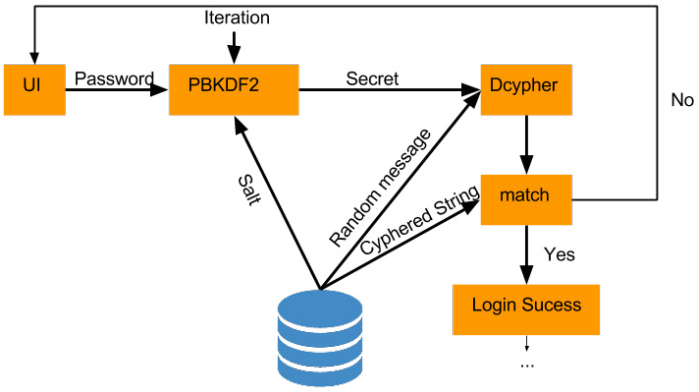


Figure 4.28: All the login attempts after the first one - model B

4.2.5 PJSIP database

When we change the library to the *PJSIP*, we decide to utilise the DB structure but, as we implemented the security mechanism presented in 4.2.4, we have a secret shared between the user and the Android application, so we can use that secret to cypher the DB. The first option that we thought, was to use the SQLite Encryption Extension (SEE)[13]. This is one official release to cypher, both data and metadata. So, to one attacker the file of the DB will be like garbage without the key, but this software has one annual cost, limiting the opportunity of using this feature. The same happened with *sqlcipher*[14]. To don't use this library, and as we are only interested in

[13]http://www.hwaci.com/sw/sqlite/see.html , [Online; Accessed 2016-6-17]
[14]https://www.zetetic.net/sqlcipher/ , [Online; Accessed 2016-6-17]

cypher the data, we implemented a solution in the java side of the Android, with all the DB created in the Linphone version of the application.

To add integrity, confidentiality and authenticity, we use AES to cypher the data and Secure Hash Algorithm-256 (SHA256) to store a hash of the values. For each table, we convert all the fields to string. We can store the cyphered message, and in all the tables we add one extra field that is one hash of all the values before being cyphered. This way, when we want to insert, for example, a new contact in the list, we need to collect all the line info in the end calculate a hash of that string. After that we cypher all the elements and store in the DB.

With this DB, when one attacker try to change some element of the DB, the attacker has to know the password to cypher and calculate the hash again, because otherwise the attack will be detected. If the attack is detected in the DB, the application immediately clean all the DB content in the device and present a message the error, to the user contact the responsible for the application, and be alerted of the problem in is device.

The Confidentiality is guaranteed by the password inserted by the user, the Integrity is maintain by the hash of all the other fields and, the Non-repudiation is guaranteed by the hash and the cyphered data.

This secure implementation limits the complexity of the query to the DB, for example, to make a query to the database by a certain URI to identify if it is present in the DB, in the Linphone version, we just need to make the query:

SELECT d.* FROM (select * from " + TABLE_NAME_LOCAIS + " UNION ALL SELECT * FROM " + TABLE_NAME_REMOTOS + ") as d WHERE contact=?

But with the data cyphered, we need to read the entire DB, decipher and check the respective hashes, to find the URI and the correspondent name. So, after the query there is the need of process the data.

SELECT * FROM (select * from " + TABLE_NAME_LOCAIS + " UNION ALL SELECT * FROM " + TABLE_NAME_REMOTOS + ")"

We need a Cursor[15] to use in the UI a *ListView*, for that, we need to decipher the entire data and verify the integrity, gather all of data in a *LinkedList*. As we need a Cursor, we create a *MatrixCursor*, and with a Collector we can order the *LinkedList*,

[15]https://developer.android.com/reference/android/database/Cursor.html , [Online; Accessed 2016-6-17]

after we get the *LinkedList* object with the right order. We need to insert them in the *MatrixCursor* and we can pass the Cursor to the UI. But in the case of a non cyphered DB, we just need to query and return immediately the result return by the DB.

4.3 Final application

With all this steps, we ended with one application, that can receive and make voice calls to VoIP clients, with all the security mechanism needed, starting from the TLS support, to the SRTP encryption for the media session and with the ZRTP key agreement. We don't give support to video call in the final application, because one of our objectives is to get a stable application, and as we start from the library, the implementation time was very limited. During the development, we implemented the presence, described in 3.1.1.2, and this way give the opportunity to the users, that are from the same SIP server, share and get their contacts state. To communicate with the SIP stack from *PJSIP* we use a service and the service communication with the UI in both directions with AIDL. The password requested in the start of the application, guarantee the confidentiality of the data stored in the DB, and in the credentials to get access to the SIP server, we also guarantee the integrity of the data, using for that one hash in all the lines of the DB.

The user when uses the application is always authenticated, after he has introduced the password. With the development we have the opportunity to test the application in different version of the Android, starting in the 4.2.2 up until the new version 6.0 *Marshmallow* where they introduce the Doze mode and the runtime permissions, this permissions allows users to manage permission from the application in real time while they are interacting with the application, this way we need to perform some changes to the UI in order to guarantee that he have the correct permission when the user receives one call. We also test the application in different architectures i.e. *arm* and *arm64*, being the application capable of run in different architectures. The final application supports the protocols presented in Table 4.1. This way the application is capable of communicate or interact with application, that contains support to the protocols mentioned.

Table 4.1: System supported RFC

Protocol Supported	*RFC number*
TLS	5246
ZRTP	6189
SRTP	3711
RTP	3550
STUN	5389
TURN	5766
ICE	5245
Presence	3856
SIP	3261

In the application, we always maintain the principals. To accomplish that, one of our goals is that the application should supports the minimum amount of permissions in the Android environment, to work properly. With that, the table 4.2, represents the difference between the permission, of the firs application, *Linphone*, and the last application, that we develop, entitled *Murmur*. With the table we can see that we pass from 22 permissions to just 12, where there is no need to access to information such as local accounts or read logs from the device.

Table 4.2: Permissions of the final application

Permissions	*Linphone*	*Murmur*
ACCESS_NETWORK_STATE	✓	✓
ACCESS_WIFI_STATE	✓	✓
AUTHENTICATE_ACCOUNTS	✓	-
BLUETOOTH	✓	-
BROADCAST_STICKY	✓	-
CALL_PHONE	✓	-
CAMERA	✓	✓
CHANGE_WIFI_MULTICAST_STATE	✓	-
CHANGE_WIFI_STATE	-	✓
REQUEST_IGNORE_BATTERY_OPTIMIZATIONS	-	✓
GET_ACCOUNTS	✓	-
INTERNET	✓	✓
MODIFY_AUDIO_SETTINGS	✓	✓
PROCESS_OUTGOING_CALLS	✓	-
READ_CONTACTS	✓	-
READ_LOGS	✓	-
READ_PHONE_STATE	✓	✓
READ_SYNC_SETTINGS	✓	-
RECEIVE_BOOT_COMPLETED	✓	✓
RECORD_AUDIO	✓	✓
VIBRATE	✓	✓
WAKE_LOCK	✓	✓
WRITE_CONTACTS	✓	-
WRITE_EXTERNAL_STORAGE	✓	-
WRITE_SYNC_SETTINGS	✓	-

The information regarding the possibility that each permission give to the application perform can be find in the developers page of android[16]. This application contains the minimum necessary permission to run:

- **ACCESS_NETWORK_STATE:** Allows applications to access information about networks, through the *ConnectivityManager*;

[16]https://developer.android.com/reference/android/Manifest.permission.html , [Online; Accessed 2016-6-17]

- **ACCESS_WIFI_STATE:** Allows applications to access information about Wi-Fi networks, through the WifiManager allowing to add callback when the device connects to a new Wi-fi point;
- **CAMERA:** Allows to access to the camera from the device;
- **CHANGE_WIFI_STATE:** Allows applications to change Wi-Fi connectivity state;
- **INTERNET:** Application can open network sockets;
- **MODIFY_AUDIO_SETTINGS:** Modify global audio settings;
- **READ_PHONE_STATE:** Detect if device state;
- **RECEIVE_BOOT_COMPLETED:** Allow the application to boot after the booting finished;
- **RECORD_AUDIO:** Application can with this permission record audio;
- **VIBRATE:** Allow the application to vibrate;
- **WAKE_LOCK:** Allows using *PowerManager* and *WakeLocks* to keep processor and screen awake.
- **REQUEST_IGNORE_BATTERY_OPTIMIZATIONS** this permission allows to use *ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS*. Only for Android 6.0

If one of this permission is not present the application will not be capable of running.

4.3.1 Doze mode

To make the application compatible with the last version of Android, we need to give support to the Doze mode. For that, we need to know how the maintenance windows work. The *AlarmManager*[17] introduced two functions *setAndAllowWhileIdle()* and *setExactAndAllowWhileIdle()*, none of these functions allow to fire alarms more that once every 9 minutes, per application. So as our applications requires a real-time connections with the server to send and receive information about the friends and

[17] https://developer.android.com/reference/android/app/AlarmManager.html, [Online; Accessed 2016-6-17]

calls that arrive, Android suggest that we should use the *Google Cloud Messaging*[18], but this is not an option, because we can not allow that data from the signalling pass by a public cloud messaging. This way, we need to find a way to pass Doze mode.

For application that can not use Doze, the systems allows a whitelist of apps. But for that the user must accept that request from the applications, as a permission from the system. To be performed that request, the application should start one activity with *ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS* or *ACTION_IGNORE_BATTERY_OPTIMIZATION_SETTINGS*. This allows to request users to add the application to the whitelist. However not all devices implement this requests, so when the user try to start one Activity with that, will be throw one exception *ActivityNotFoundException*. The request appears in the Nexus devices but not in the *Samsung S5*. This way, to include all the devices in the market, we create the Code 4.1, where if the device does not contain the requests, will be display to the user a message to request to the user to add the application to the whitelist.

Code 4.1: Request to whitelist

```
try {
    if (Build.VERSION.SDK_INT > Build.VERSION_CODES.LOLLIPOP_MR1) {
        String pkg = getPackageName();
        PowerManager pm = getSystemService(PowerManager.class);

        if (!pm.isIgnoringBatteryOptimizations(pkg)) {
            Intent i =
                    new Intent(Settings.ACTION_REQUEST_IGNORE_BATTERY_OPTIMIZATIONS)
                            .setData(Uri.parse("package:" + pkg));
            startActivity(i);
        }
    }
}catch (ActivityNotFoundException e){
    ProgressDialog progressDialog = new ProgressDialog(this);
    progressDialog.setMessage("Please go to the Settings ->
    Battery -> ... -> Battery optimization add MurmurVoice
    to the list of non optimized");
    progressDialog.setTitle("Battery optimization");
    progressDialog.setCancelable(false);
    progressDialog.setButton(DialogInterface.BUTTON_NEGATIVE, "Cancel",
    new DialogInterface.OnClickListener() {
        @Override
        public void onClick(DialogInterface dialog, int which) {

        }
    });
    progressDialog.show();
}
```

[18]https://developers.google.com/cloud-messaging/, [Online; Accessed 2016-6-17]

4.3.2 Preliminary analysis performance

In this section we want to present the preliminary result, from the Murmur application. This test aim to provide a valuation to the performs expected from the application. We perform this test using *Android Studio*. *Android Studio* allow collect the Memory[19], CPU[20] usage and the Network Monitor[21] from the applications.

This test has performed using a *Samsung S5*, with the following characteristics:

- **Operative system:** Android 6.0 (Marshmallow);
- **CPU** Quad-core 2.5 GHz Krait 400;
- **Random Access Memory (RAM)** 2 Gigabyte (GB);

The application separates the Service and the UI in two different process, so in our analyses we will separate in Interface consumption and Service consumption, in this last we include the consumption from the library *PJSIP*.

The Figure 4.29, represents the consumption when the device is with the screen on and the user interacting with them, and we can see the memory and CPU usage necessary to maintain the application up and running.

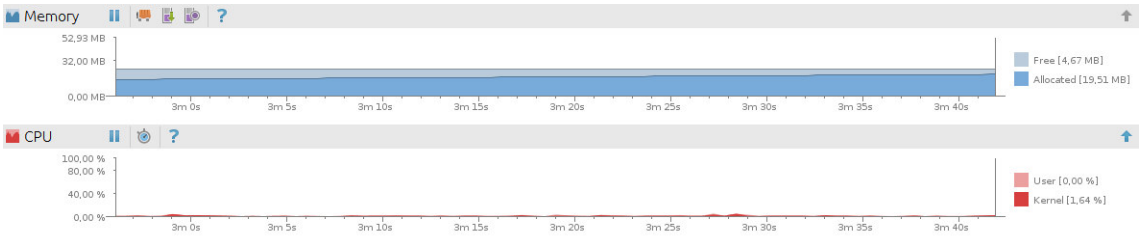


Figure 4.29: Consumption of the Android interface when not in a call

Figure 4.30, represents the consumption when the user receives and is in the Incall menu, the memory and CPU usage during this period.

[19]https://developer.android.com/studio/profile/am-memory.html [Online; Accessed 2016-6-17]
[20]https://developer.android.com/studio/profile/am-cpu.html [Online; Accessed 2016-6-17]
[21]https://developer.android.com/studio/profile/am-network.html [Online; Accessed 2016-6-17]

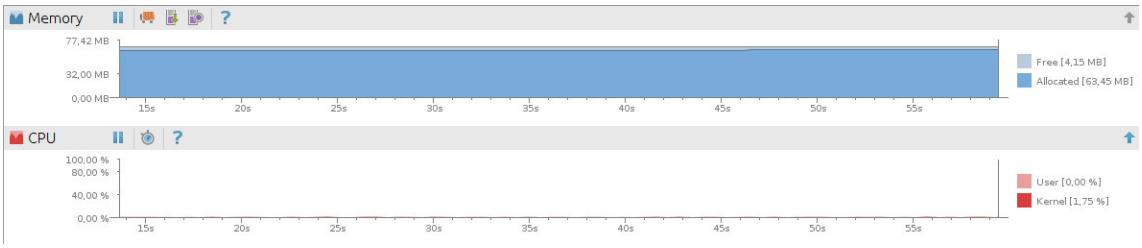


Figure 4.30: Consumption of the Android interface when in a call

The Figure 4.31, is where is represent the consumption of the Service, and are performed all the ciphered and deciphered of the audio packets. In this case the memory consumption is smaller than the previous, with the UI, this is related with the fact that the *PJSIP* library was developed thinking in spend the smallest amount of resources from the devices and using C, so this way the UI uses the images and all the resources that spend more memory compared with the PJSIP library.

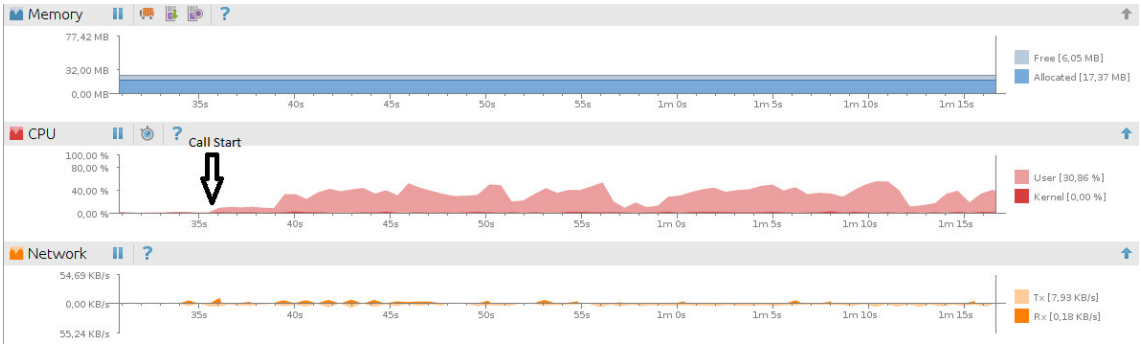


Figure 4.31: Android service consumptions

Chapter 5

Security Issues

In this chapter, we will describe the technical MITM attack that ZRTP can suffer, and present a new ZRTP attack based on the lack of information provided by the identification of a peer on ZRTP. We also present the solution to fix this issues, that is present in the most used library, open-source, that support ZRTP.

5.1 MITM attacks on ZRTP

Instead of cracking the DH key exchange as in [28], we will focus this section in attack entirely the ZRTP.

As present in [2], already exists a MITM attack on ZRTP, where is proposed multiple attack scenarios in case the authentication is not properly managed by the SIP server.

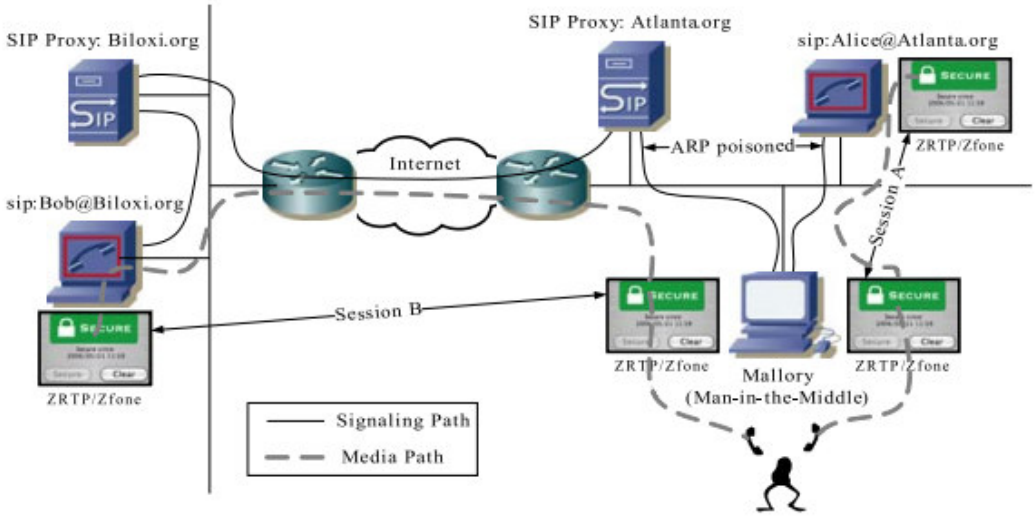


Figure 5.1: Attack architecture [2]

Mallory, the attacker as shown in Figure 5.1, will try to intercept the call. To be able to intercept the call she has to force the MITM, where *Alice* and *Bob* connect directly to *Mallory* without their knowledge, for that she perform changes in the *INVITE* and *RINGING* packets from the SIP protocol as shown in 5.2.

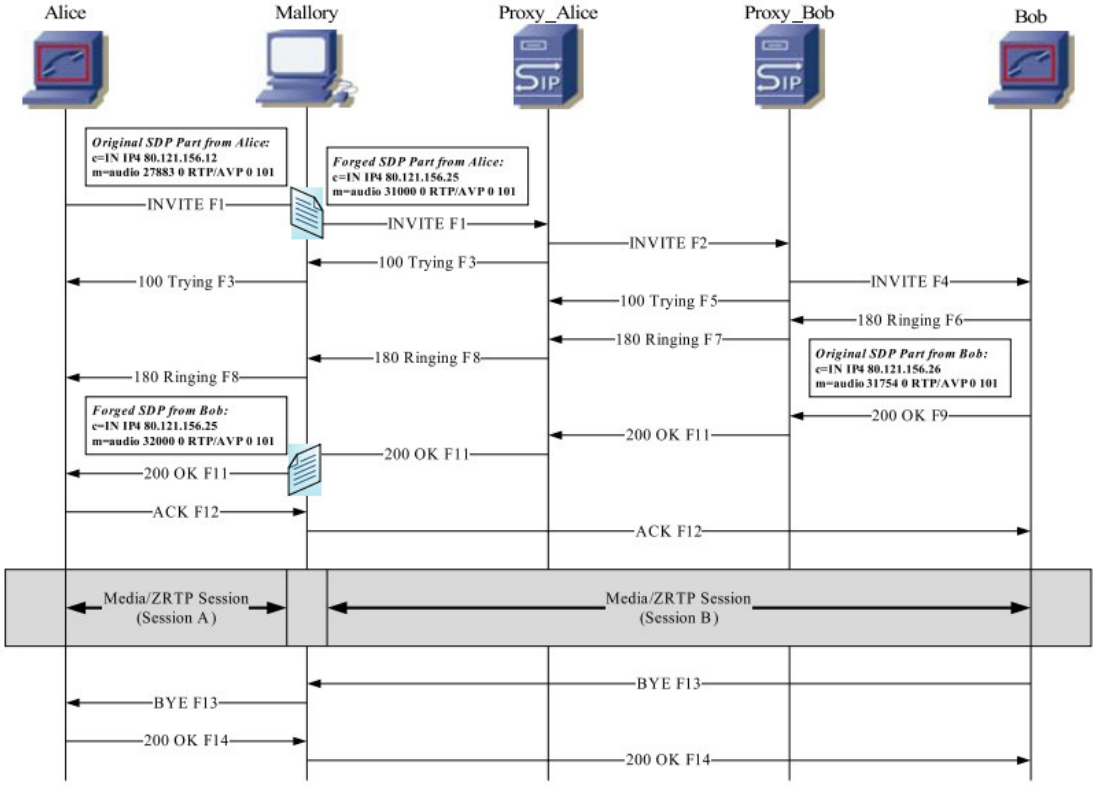


Figure 5.2: Attack on the SIP signalling [2]

The attacker, *Mallory* has two different connections with *Bob* and *Alice*. *Mallory* just needs to relay the packets from one connection to the other. These two session that *Mallory* has, get different session keys and different SAS. If *Alice* or *Bob* decide to exchange the SAS, Mallory will be detected. For this situation, the paper [2] suggest ways to *Mallory* avoid this detection:

1. Direct relay - *Mallory* just needs to send the packets from one connection to the other;

2. Direct relay + Imitation SAS - *Mallory* changes in real time the voice packet that contains the SAS and performs one exchange for the correspondent value of the SAS from *Mallory* channel with the other end;

3. Direct relay + Slide-stepping SAS - Instead of waiting for the request of the SAS, Mallory requests to the other end to tell the SAS being the message simple to exchange, that having to get all the possible SAS to perform the attack. In this case, the channel has to be separated. This way, Alice and Bob will not listen each other saying the SAS;

4. Masquerade - In this case, *Mallory* will have to impersonate the entire call, for both calls, of Alice and Bob. But this way, *Mallory* can omit all the information that she wants and exchange the SAS in real time.

To be able to perform this attacks, *Mallory* always needs to perform a new DH key exchange and, for that, they use different ZID to make the end users to exchange new keys. In this paper, the authors suggest a different ZID in a new call is normal, it can be originated by a new installation on the device or by one exchange of device, that always will end in a new ZID.

However they agree that these attacks are difficult to carry out and exists a high risk of detection, and suggest that users should bring attention and awareness when using ZRTP, to exchange currently and carefully the SAS, and look at some implementation details such as the verified flag.

5.2 The attack discovered on ZRTP

During this section, we demonstrate how can we do a MITM attack passing the security given by the *Preshared mode* and *key continuity* features, based on the lack of information provided by the ZID.

To perform the attack, we will need to gain access to the SIP signalling to be able to exchange information on the call agreement and, after that, be able to route the audio packets SRTP to a different place, to perform the MITM attack and store all the conversations.

5.2.1 Get the signalling path

Based on the Figure 2.4 with the network environment, *Mallory*, the attacker, needs to get access to the signalling path to start the attack. The signalling path is the connection that both *Alice* and *Bob* maintain with their SIP server.

When Mallory gains the access to the path, she gets the ability to see the packets from *Alice* and *Bob* in the connection that both have with the server separately. This way, *Mallory*, when *Alice* sends one *INVITE* to speak with *Bob*, she can perform some changes in the header of the *INVITE* message or other messages to make the media pass through *Mallory* host's, instead of pass directly between *Alice* and *Bob*, as shown

in the Figure 2.4.

To achieve this, and based on the network environment (Figure 2.4), as *Mallory* is in the same network as *Alice* and *Bob* she can ARP poisoning[1] the connection. After she successfully performs this attack, the network gets the structure of the figure 5.3, where all network packets, instead of going directly to the router, start to pass through *Mallory* computer and, after that, the packet follows his normal path. The same happens with the packets to *Alice* and *Bob*.

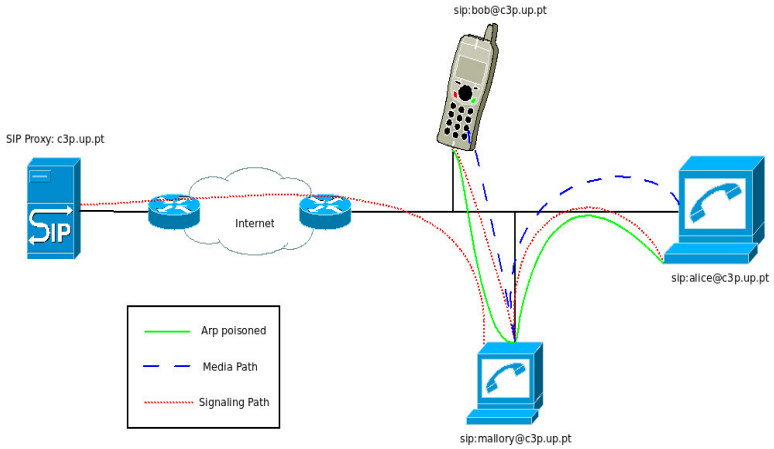


Figure 5.3: ARP poison connection

If the connection is not integrity protected, i.e., if the connection does not has TLS, for example, *Mallory* can change the *INVITE* request, as shown on Figure 5.4. This *INVITE* is sent from *Alice* with destiny to her proxy to start a conversation with *Bob* but, *Mallory* to perform the MITM changes the end user. In this case, *Bob* is changed to *Mallory* in the *INVITE* and, this way, she will get the call in their device. This change can be seen in the context of all the packets exchanged in Figure 5.5 (see *INVITE C1*).

[1]This is a technique by which an attacker sends ARP messages to the local network, this way associate the attacker's media access control (MAC) address with the IP address of another host, such as the default gateway, causing that any packet where the destiny IP address is the default gateway will be sent to the attacker instead. Carry this way a MITM attack.[29]

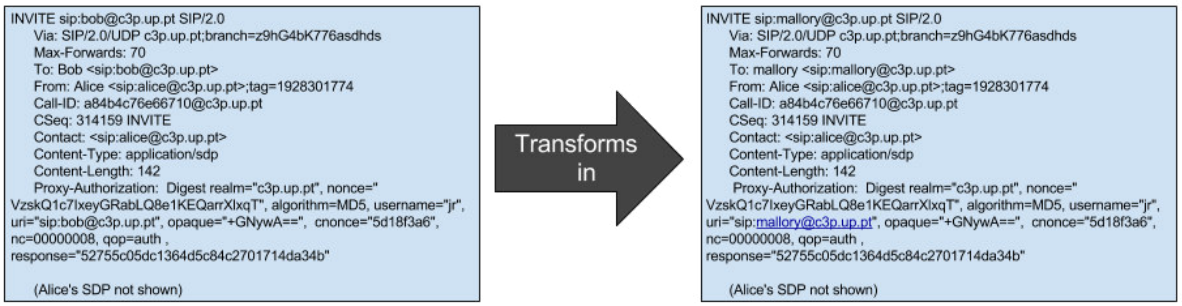


Figure 5.4: SIP *INVITE* transformation

After she gets the call, she just needs to make one more change in the signalling path. *Mallory* needs to fake the *INVITE* that she sends to *Bob* to pass by *Alice*. This can be performed based on the topology of the network in two different options, when *Mallory* sends the *INVITE* to the server change at that stage to *Bob* or when the *INVITE* comes from the server to *Bob*, any of the options can be performed without problems. The question is: *what do the attacker prefer?* Make a call as *Alice*, where the IP match with the IP from Mallory. The other option is to make the call as Mallory but, in this case, the device of Bob will appear the originator of the call as "Alice", but the MITM will be transparent in the logs of the server.

In the Figure 5.5 is represented the second option, *INVITE C6* and *C7*, where the *INVITE* is changed in the field *FROM*, from *Mallory* to *Alice*. This way, Bob will think that the call is made by a device with Alice account.

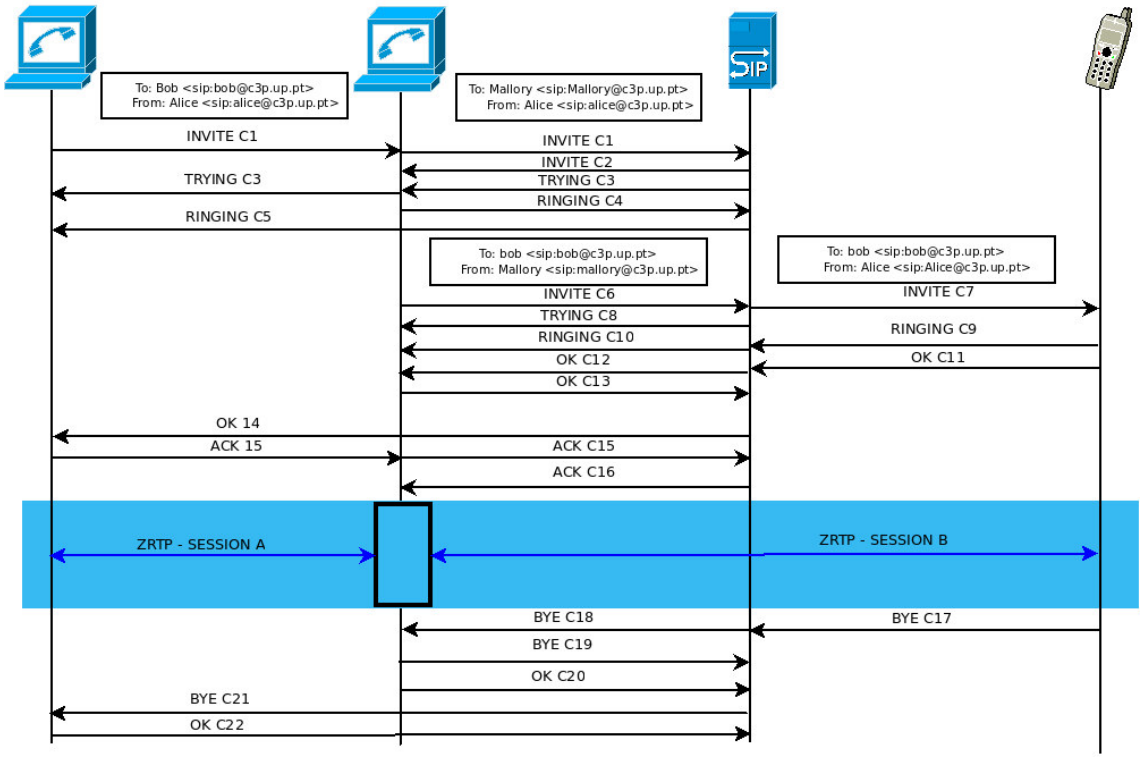


Figure 5.5: Packet flow

This way, *Mallory* will get two calls with both the participants.

5.2.2 Media session

With the two different calls, we will need two different media session. For this, we need to route the audio packets to pass through this two different calls, creating the possibility of a MITM attack. In order to *Mallory* be able to perform the MITM attack, she needs to have a program that acts as the figure 5.6, when the packets arrive to *Mallory* device, she deciphers with the correspondent key the SRTP packets. After this, they become normal packets where we can store the audio in a database. Then, she is able to cypher again with the other key to send through the other channel.

This way, *Bob* and *Alice* can communicate with each other and *Mallory* can listen and store the call without *Bob* and *Alice* suspect that the connection is being heard. *Mallory*, if she wants, can introduce at any moment audio packets or remove them.

The only difference is that the ZRTP session will not be the same between Alice and Bob. As result, the SAS will not match. If they exchange the SAS, they can discover

the ongoing attack, but this exchange is just made in the first call. Our focus is the discover of an attack after a caller and a callee established one ZRTP call.

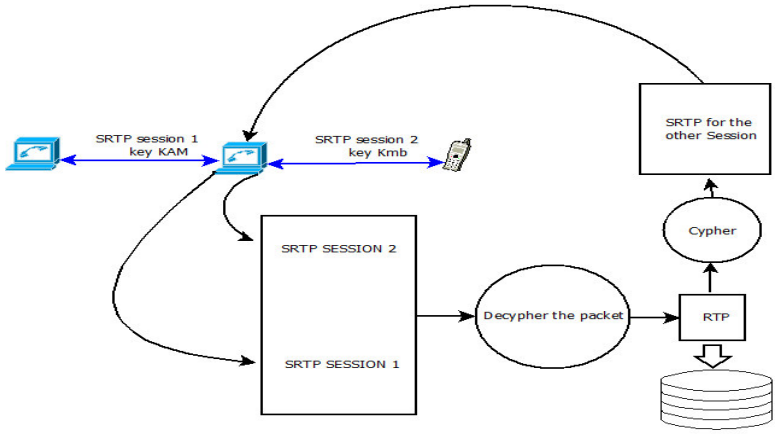


Figure 5.6: Architecture of Mallory device in the MiM mode

5.2.3 SAS defeating

As presented previously subsection 5.2.3, the MITM attack to the ZRTP is done using some ways to exchange the packets of the audio or control the voice of the other end. All of these problems are present only in the first call, because with the preshared mode the users, after having performed a confirmation that the SAS match, they will not check again in the future call if the SAS match, because the user supposes that the protocol and the application are well implemented. For this reason, they will trust that the call is secure.

We can see, for example, in Figure 5.7, that the *Linphone* application just shows one verified flag after the key has been exchanged. If we want to get information about the SAS, we have to press the button to the application show that information. The information information about the SAS should be always visible to alert users to exchange the keys with the other user.

So, to perform this attack, we will need to pass the verified flag. This way, mislead the users to don't exchange the SAS with the other end, passing the attack undetected between the caller and callee.

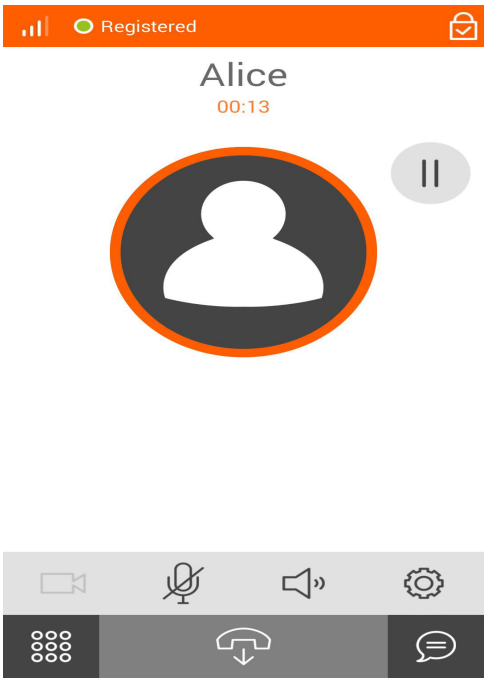


Figure 5.7: After exchange the SAS, the verified flag

To be able to pass undetected, *Mallory* needs to previously establish a call with *Bob* and another with *Alice*. This way, *Mallory* gets a shared secret with both of them.

As the only identifier of a connection in a ZRTP session is the ZID, as claimed in the RFC [7] section 4.9, *Bob* will have one entry in a cache that is from *Mallory* and another from *Alice*.

When *Mallory* performs the MITM and sends her ZID to the other end, *Bob* will see that is a valid ZID, despite *Mallory* pretended to be *Alice* in the *INVITE*. This happens because does not exists a correlation between a ZID and a user in the ZRTP protocol. This way, the program will not detect the MITM attack and will not request to exchange the SAS. In a previous call, Mallory validates the key with *Bob*, the same problem will happen in *Alice* device, and that's the reason for the MITM attack be possible.

5.2.4 A Practical Implementation of the Attack

To demonstrate how the attack is possible and how it works in a real environment, we use the SIP server without TLS support.

For the end users, we use Android clients. In this case, we use the *Linphone*[2] and *csipsimple*[3] application, because both support ZRTP and have different implementations of the ZRTP.

The *Linphone* is based on the *bzrtp*[4] and the *csipsimple* is based on the *ZRTPCPP*[5], that is a similar implementation to the Silent Circle[6] ZRTP implementations.

To perform the MITM attack we have to create one application that supports ZRTP. We choose an existing application, because with this implementation, we reduce the amount of time needed to perform the attack. In order to accomplish this, we create an application based on the *Linphone*. We choose this client because it has the conference mode where all the work of forwarding all the media packets, between caller and callee, is partial done. In this application, also allows to demonstrate the attack in a simple way and allows to store in the SD card all the call intercepted. This application writes the keys to the Android device memory, so, we can make the first part of the attack that consists in the made of a first call for both ends and exchange the key with them. After that, we use the same keys without the need of having two different implementation of the ZRTP, or different structures to store the keys.

To gain access to the signalling path, we use *ettercap* [7]. *Ettercap* is a software that allows to make MITM attacks to connections. *Ettercap* gives the opportunity to ARP poisoning the connections between the router of the network and the devices in that network. After we perform that, we can search in all packets by a filter. That filter can perform changes on the packets flow, such as, add/remove content, remove completely the packet or exchange the packets by a new one. All of this can be made on the fly by the *ettercap*.

With this tool, we create a network environment, as in the figure 5.3 with the connections between the clients and server start pass by *Mallory*. To accomplish this, we just need of one network interface, and have access to the network environment, as a normal user, such as *Alice*.

The filter that we implemented in *ettercap*, uses the following commands:

1. *search* - allows to search in the packet by a string, if the string is present the

[2] `http://www.linphone.org/`
[3] `https://play.google.com/store/apps/details?id=com.csipsimple`
[4] `https://github.com/BelledonneCommunications/bzrtp/[Online;Accessed2016-6-17]`
[5] `https://github.com/wernerd/ZRTPCPP` [Online; Accessed 2016-6-17]
[6] `www.silentcircle.com` [Online; Accessed 2016-6-17]
[7] https://ettercap.github.io/ettercap/ [Online; Accessed 2016-6-17]

command return true otherwise false;

2. *replace* - this command replaces all the occurrences of one element by another one;

3. *log* - stores in a file the content of the packet;

4. *drop* - this function marks the packet *to be dropped*. The packet will not be forwarded;

5. *exec* - this function executes a shell script;

6. *execinject* - this command allows to execute a program in the shell, and exchange the content of the file, by the output of the program.

The filter that we implement is the represented in the Filter 5.1, where we need to filter all the SIP packets from *Alice* and *Bob*.

As *Alice* is the initiator, we search by the *INVITE C1* from the figure 5.5 message and, see if it contains some *Proxy-Authorization* field. In such case, we need to replace the name from *Bob* to *Mallory* to the call be capable of arrive to the *Mallory* device, for that, we use the replace command. After this, we need to recalculate the *Proxy-Authorization*. With that, we store the data in a temporary file with the log command and drop the current packet. After this, we are capable of create a new packet based in the output of the python program. Then, we can erase the packet that we store in the hard disk previously. If the packet don't contain any field *Proxy-Authorization*, we just need to replace the name from *Bob* to *Mallory*.

In the case of *Bob* that will receive a message from the server with one *INVITE*, in this case the *INVITE C7* from the Figure 5.5, we will just need to replace the field *TO* from *Mallory* to *Alice*.

Filter 5.1: from *ettercap*

```
if (ip.proto == TCP && ip.src=='Alice IP') {
    if(tcp.src != 5060 || tcp.dst != 5060) {
        if (search(DATA.data, "INVITE")) {
                if (search(DATA.data, "Proxy-Authorization:")){

                        replace("bob@c3p.up.pt","mallory@c3p.up.pt");
                        log(DATA.data, "/tmp/payload1");
```

```
                        drop();
                        execinject("python /tmp/credentials.py
                            /tmp/payload1");

                        exec("/bin/rm /tmp/payload1");
                }
                else
                        replace("bob@c3p.up.pt","mallory@c3p.up.pt");
        }
    }
}
if (ip.proto == TCP && ip.dst=='Bob IP') {
    if(tcp.src != 5060 || tcp.dst != 5060) {
        if (search(DATA.data, "INVITE")) {
                replace("mallory@c3p.up.pt","alice@c3p.up.pt");
                msg("substitui jresende por jsr");
        }
    }
}
```

To create the *INVITE C1* message, we need to update the *Proxy-Authorization* from the TCP connection with the server. To pass this authentication, we need some parameters of the current packet and we also need the password of the user that sends the *INVITE*. This way, we can exchange and recalculate the *INVITE* message. When we call the python program, we will need of the entire packet to perform the following steps:

1. Calculate MD5 hash with the

$$h1 = username + \text{“}:\text{“} + realm + \text{“}:\text{“} + password$$

2. Another hash MD5 with the:

$$h2 = request\_method + \text{“}:\text{“} + request\_uri$$

3. Now as in this server the qop is always auth the hash is replaced by the MD5 hash of:

$$response = ha1 + \text{“}:\text{“} + nonce + \text{“}:\text{“} + nonceCount + \text{“}:\text{“} + cNonce + \text{“}:\text{“} + qop + \text{“}:\text{“} + ha2$$

In the end, we will recalculate the response field with the *python* program (Figure 5.4) and the changes performed by the *ettercap*.

5.2.5 The problem

As mentioned previously, the problem results from the lack of information given by the ZID, where it should exists information about who is at the other end.

To detect the problem, we discover that a user could change, as many times as he wants, from account in a *csipsimple* device. If it already has established a session key, the application would not request again to exchange a new SAS, this problem is present in the principal application that supports ZRTP and the code is public as the *Linphone* and *SilentPhone*.

To detect the origin of the problem, in this section, we will give the example of the *ZRTPCPP* implementation developed by *Werner Dittmann* of the ZRTP. When we look at the implementation, we find that the local cache is stored with the structure present, in the Structure 1, in a database or a file. We can see that the implementations use the ZID as the only identifier of a call, as suggested on the *RFC* of ZRTP. Also, doesn't exists a PKI where we can match the username that sends the *INVITE* with the ZID. This is the opportunity to Mallory perform the MITM attack, because it is only based on a valid previous cache entry. The attacker can always fake the *INVITE* and pass by another person.

Structure 5.2: indexed by ZID

```
typedef struct zidrecord2 {
    char version;    ///< version number of file format, this is #2
    char flags;      ///< bit field holding various flags, see below
    char filler1;    ///< round up to next 32 bit
    char filler2;    ///< round up to next 32 bit
    unsigned char identifier[ID_LEN];///< the peer's ZID or own ZID
    unsigned char rs1Interval[TIME_LEN];///< expiration time of RS1
    unsigned char rs1Data[RS_LENGTH];///< the peer's RS2 data
    unsigned char rs2Interval[TIME_LEN];///< expiration time of RS2
    unsigned char rs2Data[RS_LENGTH];///< the peer's RS2 data
    unsigned char mitmKey[RS_LENGTH];///< MiTM key if available
} zidrecord2_t;
```

5.2.6 Solution to the MITM attack

To be able to stop this MiM attack, we can find multiple solutions.

The first is to use a PKI as suggested in the RFC [7], where the authors of the

RFC purpose a solution. The solution is based on, when two users can't compare verbally the SAS, or when the users want to augment the ZRTP security. To fulfil this needs, the ZRTP allows one *OPTIONAL* signature field, where inside of the *Confirm1*, *Confirm2* or *SASrelay* message, it is possible to send the SAS signed. This solution doesn't works in multi-stream mode [7].

Another option is based in the implementation of the S/MIME, to allows to give integrity protection to the SIP and SDP payloads and ,this way, can validate and verify the integrity of the *INVITE* message. *Mallory*, this way, cannot change the *INVITE* message. But with this solution the system has the complexity of a PKI.

But the simplest way for us to implement this, is to add two comparisons methods. Instead of only the ZID to accomplish this, we have two different approaches. The first, where we can use the name that the client receives in the *FROM* field from the *INVITE*. Therefore, in the case where the user starts a call, is used the *TO* field. When the MITM attack is performed, the application will try to find in the cache entry the data to be able to discover the key. If the application find a match in a ZID but doesn't find the same name, is one ongoing attack. The person behind that attack is the owner of the ZID record, that can be discovered based the correspondent name of the ZID. This solution solves the problem in a SIP ZRTP environment. If we want to make the ZRTP alone capable of detect such attack, we have to add some information to the RFC of the ZRTP.

We propose the add of one more field to the *Hello Message*, such as the Client Identifier string (CID) that identifies the vendor and release of the ZRTP software. This way, when a client receives the *Hello message*, have the ZID of the client and a name. The name can be a SIP URI, such as *sip:alice@c3p.up.pt*. This way, when they want to establish a media session, he can compare if the ZID is known and if the name match with the stored in a cache. This allows that when the SAS is exchanged if both ends have the name, can listen and see if the voice matches, with the voice of the person. This adds a solution to the MITM attack demonstrated in the previous section, because from that moment, when *Mallory* try to intermediate the call, it is not possible, because in the *Hello message* she have to say that is Mallory and not *Alice*, otherwise the client can see that the originator of the call is *Mallory*, with *Alice* voice. With this solution the protocol alone is able to detect such MITM attack.

To solve this problem in *PJSIP*, we add a new field to the Listing 5.3, that represent the URI from the user that is trying to call. This way the MITM attack that we propose is not possible because if we see the URI or name of the person and that

name do not match with the ZID, a MITM is occurring and the responsible for such an attack is the owner of the ZID. To perform this exchange, we will have to adapt the entire code to when is searching if the ZID identifier match, with the one present in the cache, we will need to search by the URI field and check if both match. We can see one example of a change in the 5.3.

Structure 5.3: ZIDCacheFile.cpp

```
while (numRead == 1 &&
    memcmp(zidRecord->getIdentifier(), zid, IDENTIFIER_LEN)
       != 0 && memcmp(zidRecord->uri, uri, IDENTIFIER_LEN)
       != 0);
```

5.2.7 What can we conclude

Despite the claimed in [2, 7], where the authors suggest that the shared secret becomes a MITM attack much more difficult, we prove that is not true. We prove that the attacker do not need to try to get access to the victim device and his cached secrets.

To perform the attack, she just needs to perform a previous call to both the victims. After this, and based on the network environment, the attacker can always intercept any call, and does not need to intercept all the calls because he does not use the shared key of the users, he use is shared secret with both of the users.

With this attack, we show that we should always exchange the SAS with the other end. That application should always alert the users to do the authentication, otherwise the customer could always be vulnerable.

Our solution to patch the ZRTP RFC eliminates completely the problem of the MITM that we present. This is not the best solution to anonymize the RTP packets, because we will start to see who are the intervenients of a media channel. Despite being the best idea, in terms of solving the RFC problem, we adopt as the best solution the use of the identifier of the INVITE and, this way, don't send any information about the origin or destiny of the call in the RTP packet.

To conclude, in our opinion, the ZRTP will continue to be the best idea to implement a secure VoIP call, but for that, we should use information received by the signalling path and information from the media channel. This way, if the SAS match, we will not get any MITM attack and application should always be aware and motivate users

to exchange the SAS. In a case that the ZID change, the user should always be aware of that, ask to the other person by the SAS, and if she really change of device.

It is argued in the ZRTP draft that the cached shared secret becomes the MITM more difficult and, with this experience, we claim the opposite.

This way we need to change the following sentence from the RFC:

The continuity of the cached shared secrets makes it possible for us to detect the MITM when he inserts himself into the ongoing relationship, as well as when he leaves.

We also prove, that one attacker does not need to intercept all the calls to be able to perform the MITM attack, because he always can use their own credentials:

Also, if the attacker tries to stay with a long lineage of calls, but fails to execute a DH MITM attack for even one missed call, he is permanently excluded. He can no longer resynchronise with the chain of cached shared secrets.

Chapter 6

Conclusion and Future Work

The VoIP client apps that we have analysed put great emphasis on the quality of the voice service provided. However this is somewhat complementary to the security properties we want to guarantee for the voice calls, namely strong resistance to eavesdropping by third parties. Normally these apps do not provide to the users the necessary security mechanisms to achieve these goals. Overall we think we have achieved the main security objectives(1.2.1) we have proposed to accomplish for our Android app (*"Murmur"*).

6.1 Research Summary

We have conquered a series of challenges and obstacles in the development and deployment of a secure Android system for VoIP communications. In the course of development we came across different libraries and applications that would allow us to fulfil our goals and in process ended up choosing what we think where the best options available to accomplish our initial goals.

With the use of the *Linphone* library, we create an application capable of perform a ZRTP call, but with this application, we would not be able to validate the code by a third party. So, we change to a new library. This new library (*PJSIP*) does not contain the ZRTP implementations already performed, to implement the ZRTP we use *ZRTP4PJ*. To solve problems find with the over engineering present in *Linphone*, we decide to implement this application from the beginning using only the library.

During the project we have also found what we believe to be a vulnerability in the

ZRTP specification, that we managed to exploit by developing and testing a fully functional attack scenario and then provide a solution that we have implemented in our application, thus contributing to the overall security of the protocol.

6.2 Main contributions

The main contribution of this thesis has the development of a secure VoIP application and a vulnerability discovered in the ZRTP protocol. Based on this attack it is possible for any user to perform a MITM attack even within a private Virtual private network (VPN).

This vulnerability demonstrates that the communications that use ZRTP are not fully protected to attackers, and this vulnerability is present in the majority of the applications that use ZRTP.

6.3 Future Work

During the development, interoperability became one of the main topics when searching for information related with VoIP communications. VoIP providers core product and generally main focus of attention is the quality of voice in the media sessions, and security, specially resistance to MITM attacks and wiretapping are generally considered secondary features of their services. Namely normal application encryption algorithm or the authentication mechanism, are not the best of breed by default.

We also want to allow users to authenticate into the system using Near Field Communication (NFC), to mitigate the possibility of someone discovering the authentication key.

We also plan to add HD video support to the Android application within the ciphered channel and do the integration of our system in a *IOS* system, to allow users of both system can communicate with

We have also found out that current ZRTP implementations store credentials in clear text in the Android environment. We consider this to be a serious vulnerability. To mitigate it we want to add security properties as applied in the data base of the application, that uses a password introduced by the user. We implement it we will use

Protobuffers[1] to pass the information to the UI interface and this way can properly cypher the data. Using *Protobuffers* we can pass the multiple languages used in this application, that support *Protobuffers*

For future work we also intend to support S/MIME, to be able to use certificates, in cases where there already exists a PKI in place, we can take advantage of certificates to add more security features to the service as a whole, allowing users to sign the SAS, or to sign the body of the SIP messages.

However, our project still needs some improvements at the application level, regarding the security of the keys in the device as well as video support. We also want to reduce the power consumption always important when we think in devices with limited battery.

6.4 Final conclusions

We still do not have a fully trustable and well-accepted solution for peer-to-peer secure VoIP communication in the world. Our purpose is that this Android application, could be the solution to the Portuguese institution, as one trustable and fully auditable application, developed in Portugal for Portuguese Institutions. Our application implements the ZRTP protocol, and allows users to make secure peer-to-peer audio calls, between two persons. Our solutions also cypher all the content present in Android application generated by the user except the ZRTP data, for that the user have a shared secret with the mobile device. We also give the opportunity to the users be able to see and share, presence information to see the contacts from the list that are available or not.

Without the patch to the vulnerability that we expose, the current application that use ZRTP, are exposed to MITM attacks. With the patch to the ZRTP protocol that we presented, it is possible to completely mitigate the attack.

Therefore, our system presents a fast and secure method to perform audio calls to any given institutions taking advantage of the ZRTP protocol to exchange keys.

[1] https://developers.google.com/protocol-buffers/ [Online; Accessed 2016-6-17]

Bibliography

[1] Jiri Kuthan Ulrich Abend Henning Schulzrinne Dorgham Sisalem, John Floroiu. *SIP SECURITY*. WILEY, 2009.

[2] Martin Petraschek, Thomas Hoeher, Oliver Jung, Helmut Hlavacs, and Wilfried N Gansterer. Security and usability aspects of man-in-the-middle attacks on zrtp. *J. UCS*, 14(5):673–692, 2008.

[3] Claudio M de Farias, Leandro CG Lustosa, Paulo H de A Rodrigues, and Pedro R Moreira. Combinação de pilhas de protocolo para a construção de um softphone sip. In *Proceedings of the 14th Brazilian Symposium on Multimedia and the Web*, pages 178–185. ACM, 2008.

[4] Thomas J Walsh and D Richard Kuhn. Challenges in securing voice over IP. *IEEE Security & Privacy*, (3):44–49, 2005.

[5] Angelos D Keromytis. A survey of voice over IP security research. In *Information Systems Security*, pages 1–17. Springer, 2009.

[6] Kamiar Radnosrati. *An examination of keystroke dynamics for continuous user authentication*. PhD thesis, Trading-off compression, energy efficiency, and QoE in wireless network, 2013.

[7] Philip Zimmermann, Alan Johnston, and Jon Callas. Zrtp: Media path key agreement for unicast secure rtp. Technical report, RFC Editor, 2011.

[8] Whitfield Diffie and Martin E Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[9] Eric Rescorla. Diffie-hellman key agreement method. 1999.

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

[11] M. Baugher et all. The Secure Real-time Transport Protocol (SRTP). RFC 3711, RFC Editor, March 2004.

[12] U. Blumenthal et all. The Advanced Encryption Standard (AES) Cipher Algorithm in the SNMP User-based Security Model. RFC 3826, RFC Editor, June 2004.

[13] F. Andreasen et all. Session Description Protocol (SDP) Security Descriptions for Media Streams. RFC 4568, RFC Editor, July 2006.

[14] Prateek Gupta and Vitaly Shmatikov. Security analysis of voice-over-ip protocols. In *Computer Security Foundations Symposium, 2007. CSF'07. 20th IEEE*, pages 49–63. IEEE, 2007.

[15] Hamid R. Nemati Li Yang. *Applied Cryptography for Cyber Security and Defense: Information Encryption and Cyphering: Information Encryption and Cyphering.* Information Science Reference, 2010.

[16] B. Ramsdell et all. Secure/Multipurpose Internet Mail Extensions (S/MIME) Version 3.2 Message Specification. RFC 5751, RFC Editor, January 2010.

[17] Dimitris Geneiatakis, Georgios Kambourakis, Tasos Dagiuklas, Costas Lambrinoudakis, and Stefanos Gritzalis. Sip security mechanisms: A state-of-the-art review. In *Proceedings of the Fifth International Network Conference (INC 2005)*, pages 147–155, 2005.

[18] Helmut Hlavacs, Wilfried N Gansterer, Hannes Schabauer, Joachim Zottl, Martin Petraschek, Thomas Hoeher, and Oliver Jung. Enhancing zrtp by using computational puzzles. *J. UCS*, 14(5):693–716, 2008.

[19] Riccardo Bresciani. The zrtp protocol-analysis on the diffie-hellman mode. 2009.

[20] JoongMan Kim, SeokUng Yoon, Hyuncheol Jeong, and YooJae Won. Implementation and evaluation of sip-based secure voip communication system. In *Volume 2, Embedded and Ubiquitous Computing, 2008. EUC'08. IEEE/IFIP International Conference on*, pages 356–360. IEEE, 2008.

[21] S Manjur, Mosleh Abu-Alhaj, Omar Abouabdalla, Tat-Chee Wan, Rahmat Budiarto, and Ahmed M Manasrah. Conference gateway for heterogeneous clients: Real time switching clients and interasterisk exchange clients. *INTERNATIONAL JOURNAL OF INNOVATIVE COMPUTING INFORMATION AND CONTROL*, 7(1):395–406, 2011.

[22] J. Rosenberg et all. SIP: Session Initiation Protocol. RFC 3261, RFC Editor, June 2002.

[23] M. Handley et all. SDP: Session Description Protocol. RFC 4566, RFC Editor, July 2006.

[24] J. Rosenberg. A Presence Event Package for the Session Initiation Protocol (SIP). RFC 3856, RFC Editor, August 2004.

[25] Packet-based multimedia communications systems. ITU, TELECOMMUNICATION STANDARDIZATION SECTOR OF ITU, September 99.

[26] B. Foster et all. Media Gateway Control Protocol. RFC 3661, RFC Editor, December 2003.

[27] Burt Kaliski. Pkcs# 5: Password-based cryptography specification version 2.0. RFC, RFC Editor, 2000.

[28] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology-CRYPTO'96*, pages 104–113. Springer, 1996.

[29] Sean Whalen. An introduction to arp spoofing. *Node99 [Online Document], April*, 2001.

Appendix A

Development notes

A.1 Programming languages used

We use multiple programming languages to develop the Android application, and the attack.

- Java
- eXtensible Markup Language (XML)
- C
- C++
- Python

A.2 Software used

- ***Android-Studio***
  - Description: The Official IDE for Android.
  - Version: 1.5.1
  - Website: `https://developer.android.com/studio/index.html`
- **Java ™ SE Development Kit**

  - Description: Free and open source implementation of the Java Platform, Standard Edition (Java SE).
  - Version: 8, Update 91 (Java Development Kit (JDK) 8u91)
  - Website: `http://openjdk.java.net/`

- **Native development kit**
  - Description: The Android NDK is a toolset that lets you implement parts of your app using native-code languages such as C and C++.
  - Version: r10e
  - Website: `https://developer.android.com/ndk/index.html?hl=ml`

- **Android Software Development Kit (SDK)**
  - A software development kit that enables developers to create applications for the Android platform.
  - Website: `https://developer.android.com/ndk/index.html?hl=ml`

- **Simplified Wrapper and Interface Generator**
  - SWIG is a software development tool that connects programs written in C and C++ with a variety of high-level programming languages.
  - Version: 3.0
  - Website: `http://www.swig.org/`

- **liblinphone**
  - Liblinphone is a high level library integrating all the SIP video calls feature into one API.
  - Website: `http://www.linphone.org/technical-corner/liblinphone/overview`

- **PJSIP**
  - *PJSIP* is a free and open source multimedia communication library written in C language implementing standard based protocols such as SIP, SDP, RTP, STUN, TURN, and ICE. It combines signaling protocol (SIP) with rich multimedia framework and NAT traversal functionality into high level API that is portable and suitable for almost any type of systems ranging from desktops, embedded systems, to mobile handsets.

  - Version: 2.4.5
  - Website: `http://www.pjsip.org/`

- **ZRTP**
  - sources and build files to create the GNU ZRTP
  - Version: 3.0.0
  - Website: `https://github.com/wernerd/ZRTP4PJ`

- **Ettercap**
  - comprehensive suite for man in the middle attacks.
  - Version: 0.8.2-Ferri
  - Website: `https://ettercap.github.io/ettercap/`

- **ZBAR**
  - QR code reader
  - Version: 0.10
  - Website: `http://zbar.sourceforge.net/`

Appendix B

Manual

B.1 Patch to ZRTP

1. Access to the website `https://github.com/wernerd/ZRTP4PJ`. This is the repository that contains the sources that we need. We should clone this project to the pjproject third_party directory, with:

   ```
   git clone https://github.com/wernerd/ZRTP4PJ
   ```

2. After that we need the ZRTP sources for that:

   ```
   cd zsrtp
   sh getzrtp.sh
   cd ..
   ```

3. After this download the files are all in the computer, we are ready to compile for that we need at list one run, of the pjproject, for that check the Appendix . Note that for each type of processor we will need to build the ZRTP and the pjproject again.After that first we are able to compile ZRTP with the following commands:

   ```
   cd build/zsrtp
   make dep
   make
   ```

4. If make does not report errors, warnings are displayed, the build was successful and the library (libzsrtp-arm-unknown-linux-androideabi.a) is in the lib folder from the root of the ZRTP4PJ project. After that we need to copy this file to:

```
cp /development/pjproject/third_party/ZRTP4PJ/lib/*
/development/pjproject/third_party/lib/
```

5. We can now use the library in the pjproject. To patch the pjproject to support we need to go to the file endpoint.cpp located in

   ```
   cat /development/pjproject/pjsip/src/pjsua2/endpoint.cpp
   ```

6. In function on_create_media_transport change the function to call ZRTP function available by the transport_zrtp.h as we can see:

```
#include <transport_zrtp.h>

pjmedia_transport*
Endpoint::on_create_media_transport(
                            pjsua_call_id call_id,
                            unsigned media_idx,
                            pjmedia_transport *base_tp,
                            unsigned flags){
    pjmedia_transport *zrtp_tp = NULL;
    pj_status_t status;
    pjmedia_endpt* endpt = pjsua_get_pjmedia_endpt();


    PJ_LOG(3,(THIS_FILE, "ZRTP transport created"));

    status = pjmedia_transport_zrtp_create
            (endpt, NULL, base_tp,&zrtp_tp, flags);

    zrtp_cb_user_data* zrtp_cb_data =
    (zrtp_cb_user_data*) malloc(sizeof(zrtp_cb_user_data));
        zrtp_cb_data->zrtp_tp = zrtp_tp;
        zrtp_cb_data->call_id = call_id;
        zrtp_cb_data->cipher = pj_str("");
        zrtp_cb_data->sas = pj_str("");
        zrtp_cb_data->sas_verified = PJ_FALSE;

        zrtp_UserCallbacks* zrtp_cbs =
        (zrtp_UserCallbacks * )malloc(sizeof(zrtp_UserCallbacks));
        zrtp_cbs->zrtp_secureOn = &secureOn;
        zrtp_cbs->zrtp_secureOff = &secureOff;
        zrtp_cbs->zrtp_showSAS = &showSAS;
        zrtp_cbs->zrtp_confirmGoClear = &confirmGoClear;
        zrtp_cbs->zrtp_showMessage = &showMessage;

    zrtp_cbs->zrtp_zrtpNegotiationFailed = &zrtpNegotiationFailed;
    zrtp_cbs->zrtp_zrtpNotSuppOther = &zrtpNotSuppOther;




    zrtp_cbs->zrtp_zrtpAskEnrollment = &zrtpAskEnrollment;
    zrtp_cbs->zrtp_zrtpInformEnrollment = &zrtpInformEnrollment;
    zrtp_cbs->zrtp_signSAS = &signSAS;
    zrtp_cbs->zrtp_checkSASSignature = &checkSASSignature;
    zrtp_cbs->zrtp_sendToJava=&sendToJava;
    zrtp_cbs->zrtp_receiveFromJava=&receiveFromJava;
    zrtp_cbs->userData = zrtp_cb_data;

    pjmedia_transport_zrtp_setUserCallback(zrtp_tp, zrtp_cbs);
```

```
    /*
     * Initialize the transport. Just the filename of the ZID file that holds
     * our partners ZID, shared data etc. If the files does not exists it will
     * be created an initialized.
     */
    pjmedia_transport_zrtp_initialize(zrtp_tp,
        "/data/data/com.adytatech.murmurvoice/files/simple.zid", PJ_TRUE);
    PJ_LOG(3,(THIS_FILE, "ZRTP transport returned"));
    return zrtp_tp;
}
```

7. In this function we also define the zrtp_UserCallbacks, this callbacks, are responsible to send the information to the UI. We can resume the callback functionality as:

   - ***zrtp_secureOn*** - Inform user interface that is speaking in a secure channel.
   - ***zrtp_secureOff*** - Inform user interface when the security is compromised.
   - ***zrtp_showSAS*** - Gives the SAS to pass to the user interface.
   - ***zrtp_confirmGoClear*** - Inform the user that ZRTP received "go clear" message from its peer.
   - ***zrtp_showMessage*** - Show information to user for example if the other peer possibly down or if contains a retained secret in the local cache.
   - ***zrtp_zrtpNegotiationFailed*** - ZRTP exchange fails, this function will be used.
   - ***zrtp_zrtpNotSuppOther*** - If the other side does not support ZRTP, the user will be alerted by this callback.
   - ***zrtp_zrtpAskEnrollment*** - Inform about a PBX enrolment request.
   - ***zrtp_zrtpInformEnrollment*** - PBX enrolment result.
   - ***zrtp_signSAS*** - if we pretend to sign this SAS, when this method is called it should return a signed SAS.
   - ***zrtp_checkSASSignature*** - the library calls this callback to request a SAS signature check.
   - ***userData*** - This is not a function, is a valid field to store some content, to be used in the future by the user.

8. Another thing that the is need to implement is the way to reject and accept the SAS. To accomplish this we need of two function, ***zrtp_resetSASVerified***

and ***zrtp_SASVerified***, we can function responsible to pass this to and from the UI. The first should be used when the user reject the SAS, the second is when the user verified with the other end and both SAS are the same.

```
int Endpoint::zrtp_SASAccept(pjsua_call_id call_id)
{
        struct xContext ac = jzrtp_getContext(call_id);
        if(ac.cbUserData != NULL){
                ac.cbUserData->sas_verified = 1;
        }
        if(ac.zrtpContext != NULL){
                zrtp_SASVerified(ac.zrtpContext);
        }else{
                PJ_LOG(1, (THIS_FILE, "accept:
                No ZRTP context for call \%d", call_id));
        }

}

void Endpoint::zrtp_SASRevoked(pjsua_call_id call_id) {
        struct zrtp_SASRevoked ac = jzrtp_getContext(call_id);
        if(ac.cbUserData != NULL){
                ac.cbUserData->sas_verified = 0;
        }
        if(ac.zrtpContext != NULL){
                zrtp_resetSASVerified(ac.zrtpContext);
        }else{
                PJ_LOG(1, (THIS_FILE, "zrtp_SASRevoked: \\
                No ZRTP context for call \%d", call_id));
        }
}
```

9. After all this patches we are ready to get a *PJSIP* with ZRTP, with the functionality that we pretend.

B.2 Compile *PJSIP*

1. Access to the website `http://www.pjsip.org/`. This is the repository that contains the sources that we need. We should download the last library available after that we will need to open in the terminal in the root of the folder downloaded

2. To start the compilation process of the library we need to access to the file:

```
cd /path/to/your/pjsip/dir
nano pjlib/include/pj/config_site.h
```

3. Insert inside of the config_site.h the following code:

```
#define PJ_CONFIG_ANDROID 1
#include <pj/config_site_sample.h>
#define PJSIP_ENCODE_SHORT_HNAME 0
//SIP headers in their short forms to reduce size.
#define PJ_HAS_SSL_SOCK 1 //Enable TLS SIP transport
   support.
#define PJSIP_INCLUDE_ALLOW_HDR_IN_DLG 0 //remove HDR
   header


/*if we want debug of the stack */
#define PJ_DEBUG 0
#define PJ_DEBUG_MUTEX 0
#define PJ_POOL_DEBUG 0
#define PJ_TIMER_DEBUG 0

/* Disable some audio codecs */
#define PJMEDIA_HAS_L16_CODEC 0
#define PJMEDIA_HAS_G722_CODEC 0
```

4. in this stage we are ready to compile, and we need to add the NDK to the enviorment and other variable:

Structure B.1: exports

```
export ANDROID_NDK_ROOT=/path_to_android_ndk_dir
export TARGET_ABI=armeabi-v7a
export APP_PLATFORM=android-19
```

5. To compile with ZRTP the *PJSIP*, we need to add somethings as shown in the B.2, where the user should change "-lzsrtp-arm-unknown-linux-androideabi" based on the type of architecture. If we don't want ZRTP we could jump this step.

Structure B.2: To add ZRTP

```
export PJ_CFLAGS := $(APP_CFLAGS)
-lzsrtp-arm-unknown-linux-androideabi -lstdc++
-lssl -lcrypto -L/path/to/openssl-android/
-I/path/to/openssl-android/include/ -lssl
```

```
-lcrypto -lstdc++ -lssl -lcrypto  -lgnustl_static
-lstdc++  -lgcc
```

6. At this momment we are ready to run a script to check if everthing is correct, and set the configuration needed to compile to Android:

```
./configure-android --use-ndk-cflags
   --with-ssl=/path/to/openssl
```

7. To compile if the previous command does not get any error:

```
make clean
make dep
make
```

8. After the compilation, we need to create the interface to the java, to the C communicate with the java, communications made by SWIG. For that run:

```
cd pjsip-apps/src/swig
make clean
make
```

9. This way the library to run in Android is generated along with the swig interface to integrate with the java code. After that we need to copy the library and the java classes to the Android project:

```
cp -r pjsip-apps/src/swig/java/android/libs/*
/path/to/Android/project/libs

cp
   pjsip-apps/src/swig/java/android/src/org/pjsip/pjsua2/*
/path/to/Android/project/app/src/main/java/org/pjsip/pjsua2/
```

10. At this point the entire *PJSIP* library is compiled to *armeabi-v7a* if we want to change to another architecture we just need to change the *TARGET_ABI* defined in B.1

11. To be able to run in all the devices the *PJSIP*, we just need to load the *openssl* library and the *pjsua* to be able to invoke the swig function. We need the *openssl* just because the Android change in the recent version 6.0 the Secure Sockets Layer (SSL) library to the *boringSSL*, this way when we want to load the stack in the java side of *Android*, we need to run:

```
System.loadLibrary("crypto");
System.loadLibrary("ssl");
System.loadLibrary("pjsua2");
```

12. Another problem regarding the *PJSIP* library is that they can not listen in the 5060 in Android environment for that we have to change the server from the default port[1]. This just happen in case that we want to use TCP, with TLS we can use 5061.

B.3 Use AIDL

In this section we want to present the way to work with AIDL from the UI to the Service, and how to do the communication from the Service to the UI with AIDL.

When we want to use to start a Service with a AIDL interface, we need to start the Service, for that, and based on Structure B.3, we use the command *startService* that starts the service, after that we will need to bind the service, with the UI, for that we use the *bindService*, that will give one interface with the service.

Structure B.3: User interface activity code

```
sip_service_intent = new Intent(instance, SipService.class);
startService(sip_service_intent);
bindService(new Intent(instance, SipService.class), connection,
    Context.BIND_AUTO_CREATE);
```

Structure B.4: Connect the Service with the UI

```
private ISipService service;
private ServiceConnection connection = new ServiceConnection() {
    @Override
    public void onServiceConnected(ComponentName arg0, IBinder arg1) {
        service = ISipService.Stub.asInterface(arg1);
        try {
            instance.service.setCalback(instance.callback);
        } catch (RemoteException e) {
            e.printStackTrace();
        }
    }

    @Override
    public void onServiceDisconnected(ComponentName arg0) {
        service = null;
    }
};
```

[1]https://trac.pjsip.org/repos/ticket/1488 [Online; Accessed 2016-6-17]

When the UI get the connection with the service B.4, and wants to establish another AIDL connection from the service to the UI, we will need to set another AIDL interface, in this case B.6. This way the Service contains callback that can invoke and send information to the UI, as the example B.6 function that execute the code B.7, in the UI.

Structure B.5: AIDL from the UI to the Service

```
// ISipService.aidl
package com.adytatech.murmurvoice;

// Declare any non-default types here with import statements

import com.adytatech.murmurvoice.ICallback;
interface ISipService {
        /**
        * Force to stop the sip service (stack + everything that goes arround stack)
        */
        void forceStopService();
        /**
        * Restart the sip stack
        */
        void askThreadedRestart();

    /**
        * Change registration for a given account/profile id (id in database)
        * @param accountId the account for which we'd like to change the
            registration state
        * @param renew 0 if we don't want to unregister, 1 to renew registration
        */
        void setAccountRegistration(int accountId, int renew);

        /**
         * Place a call.
         *
         * @param callee The sip uri to call.
         * It can also be a simple number, in which case the app will autocomplete.
         * If you add the scheme, take care to fill completely else it could be
             considered as a call
         * to a sip IP/domain
         * @param accountId The id of the account to use for this call.
         */
        boolean makeCall(in String callee, int accountId);


        void setCallback(ICallback x);

}
```

Structure B.6: AIDL FILE

```
// ICallback.aidl
package com.adytatech.murmurvoice;

// Declare any non-default types here with import statements
```

```
interface ICallback {
    void onReceiveLoginInfo(boolean x);
}
```

Structure B.7: UI callback

```
private final ICallback.Stub callback=new ICallback.Stub() {
    public void onReceiveLoginInfo(final boolean state ) {
            runOnUiThread(new Runnable() {
                public void run() {
                    try {
                        instance.unbindService(instance.connection);
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                    if (state == false)
                        LoginActivity.getInstance().wrongPassword();
                    else
                        LoginActivity.getInstance().terminate();
                }
    });
}
```

B.4 *PJSIP* stack supported features

In this section we will show the different features as some RFC, that *PJSIP* supports. This section is based in the information provided in the *teluu* the developers of *PJSIP*[2].

B.4.1 SIP Capabilities

- Base specs:
  - Core methods: RFC 3261: INVITE, CANCEL, BYE, REGISTER, OPTIONS, INFO
  - Digest authentication (RFC 2617)
- Transports:
  - UDP, TCP, TLS (server or mutual)
  - DNS SRV resolution (RFC 3263)
  - IPv6 (UDP only)

[2]http://www.teluu.com/content/pjsip-features-and-datasheet [Online; Accessed 2016-6-17]

  - QoS (DSCP, WMM)

- Routing/NAT:
  - rport (RFC 3581)
  - Service-Route header (RFC 3608)
  - SIP outbound for TCP/TLS (RFC 5626)
  - Path header (with SIP outbound) (RFC 3327)

- Call:
  - Offer/answer (RFC 3264)
  - hold, unhold
  - redirection
  - transfer/REFER (attended and unattended):
    - Base (RFC 3515)
    - replaces (RFC 3891)
    - Referred-by (RFC 3892)
  - sipfrag support (RFC 3420)
  - norefersub (RFC 4488)
  - UPDATE (RFC 3311)
  - 100rel/PRACK (RFC 3262)
  - tel: URI (RFC 3966)
  - Session Timers (RFC 4028)

- SDP:
  - RFC 2337 (obsoleted by RFC 4566)
  - RTCP attribute (RFC 3605)
  - IPv6 support (RFC 3266)

- Multipart (RFC 2046, RFC 5621)

- Presence and IM:
  - Event framework (SUBSCRIBE, NOTIFY) (RFC 3265)

  - Event publication (PUBLISH) (RFC 3903)
  - MESSAGE (RFC 3428)
  - typing indication (RFC 3994)
  - pidf+xml (RFC 3856, RFC 3863)
  - xpidf+xml
  - RPID (subset) (RFC 4480)

- Other extensions:

  - INFO (RFC 2976)
  - AKA, AKA-v2 authentication (RFC 3310, RFC 4169)
  - ICE option tag (RFC 5768)

- Compliance:

  - Issues with Non-INVITE transaction (RFC 4320)
  - Issues with INVITE transaction (RFC 4321)
  - Multiple dialog usages (RFC 5057)

B.4.2 NAT Traversal

- STUN:

  - RFC 5389
  - Some RFC 3489 compatibility
  - DNS SRV resolution
  - short/long term authentication
  - fingerprinting

- TURN:

  - RFC 5766
  - DNS SRV resolution
  - UDP and TCP client connection

- ICE:
  - RFC 5245
  - host, srflx, and relayed candidates
  - aggressive and regular nomination
  - ICE option tag (RFC 5768)
- NAT type detection:
  - legacy RFC 3489
- Other:
  - QoS support on sockets (DSCP, WMM)

B.4.3 Media/audio capabilities

- Core:
  - any clockrates
  - N-channels support
  - zero thread
- Base:
  - DTMF (RFC 2833)
  - echo cancellation (Speex, CANEC, suppressor, or native)
  - client conferencing
  - inband DTMF/tone generation
  - WAV file playback and recording
  - WAV file playlist
  - memory based playback and capture
  - adaptive jitter buffer
  - packet lost concealment
  - clock drift recovery

- Codecs:
  - Bundled:
    - Speex 8KHz, 16Khz, 32KHz
    - iLBC, GSM,
    - L16, G.711A/U (PCMA/PCMU),
    - G.722,
    - G.722.1 16KHz/32KHz (Siren7/Siren14, licensed from Polycom)
  - with Intel IPP library:
    - AMR-NB, AMR-WB,
    - G.722, G.722.1,
    - G.723.1, G.726, G.728, G.729A,
  - Hardware codecs:
    - on Nokia with APS/VAS-Direct: AMR-NB, G.729, iLBC, PCMA, PCMU
    - on iPhone: iLBC
- Transports:
  - RTP and RTCP with media statistic (RFC 3550, RFC 3551)
  - RTCP XR (subset, RFC 3611)
  - UDP, STUN, ICE
  - IPv6 (UDP only)
  - SRTP (RFC 3711) and SRTP SDES (RFC 4568)
  - QoS (DSCP, WMM)
  - Symmetric RTP/RTCP (RFC 4961)
- Audio devices:
  - native WMME (Windows, Windows Mobile)
  - native ALSA (Linux)
  - native Symbian MMF (Symbian/Nokia S60)
  - native APS (Nokia S60) with hardware EC, and APS-Directto support hardware codecs

- native VAS (Nokia S60) with hardware EC, and VAS-Directto support hardware codecs
- native CoreAudio (Mac OS X, iPhone) with support for native/hardware EC
- PortAudio (WMME, DirectSound, OSS, ALSA, CoreAudio, etc.)