Secure Device Management Protocol

Itay Grudev

A dissertation submitted in partial fulfilment
of the requirements for the degree of
Bachelor of Science
of the
University of Aberdeen.

Department of Computing Science

May 23, 2019

Declaration

No portion of the work contained in this document has been submitted in support of an application for a degree or qualification of this or any other university or other institution of learning. All verbatim extracts have been distinguished by quotation marks, and all sources of information have been specifically acknowledged.

Signed:

Date: May 23, 2019

Abstract

Over the last decade we have integrated cloud services very close in our lives. Many these services are offered by large companies like Google, Apple and Microsoft. These come with severe privacy and security implications and considerable amount of people and businesses opt to deploy cloud services of their own. We currently have decent open source replacements for most cloud services like storage, calendar, social networking, instant messaging, etc. As a result over the last several years we have seen an ever growing number of open source replacements of cloud services. There is one service that is yet to receive an implementation focused on privacy and security. The aim of this project is to design the protocol for communication of one such service.

The Secure Device Management Protocol is the first such protocol, designed from the ground up to comply with the ethical design philosophy. The goal of the protocol is to provide a messaging and management service for mobile devices that allows operations like remotely ringing, tracking, locking or erasing the data on your device. There are multiple issues when designing such service and this thesis explains the problems, possible solutions and finally describes a communication protocol that achieves it in a very high security and elegant method.

Contents

Chapter 1

Introduction

We have grown to rely on Cloud Services in our everyday life. Whether that is checking your email, social networking on messaging friends and colleagues our digital life reflects and supplements our own. We store all of our contact information (including information about other people) and our day-to-day activities on the Cloud. We use Cloud Storage to save and backup our most important files and we use social networks for our most intimate communication. Businesses rely on the Cloud for audio/video conferencing and messaging services and even storing their closely guarded proprietary information. However relying on software as a service (SaaS) products provided by big companies like Google, Microsoft, Facebook, Apple, etc. introduces significant security and privacy implications. Naturally many businesses and privacy-aware individuals opt to deploy such services on their own, away from the control of a third party. Deploying open source, free software[1] cloud services on your own is not only cost efficient, but also offers additional security as you own and control the data stored. These free software projects are an important alternative to the services provided by big companies and are being endorsed ever so widely.

Fortunately with the increase of popularity of such services, we have also seen an increase on the number of such Open Source projects. Both like minded individuals and businesses have endorsed the idea and started working on leveling out the field. In the last several years we have seen an ever growing array of Libre software cloud services. We have cloud storage software like *ownCloud*[2], *Nextcloud*[3] and *Pydio*[4], Chat and Messaging relays like the distributed *Matrix.org*[5] federation, *XMPP*[6] and even Distributed Social Networks like *Diaspora*[7] and *Mastadon*[8].

There is one service that isn't used very often and does not get the recognition it deserves. But we often resort to it when we can't find or lose our devices. This is the Device Management Service that allows you to locate our devices when they are lost. These are products like Google's Android Device Manager (later renamed Find My Device) and Apple's Find My iPhone. These

[1] Free software means software that respects users' freedom and community. Roughly, it means that the users have the freedom to run, copy, distribute, study, change and improve the software. Thus, "free software" is a matter of liberty, not price. "Open source" is something different: it has a very different philosophy based on different values. Its practical definition is different too, but nearly all open source programs are in fact free. – The Free Software Foundation

[2] `https://owncloud.org`

[3] `https://nextcloud.com`

[4] `https://pydio.com`

[5] `https://matrix.org`

[6] `https://xmpp.org`

[7] `https://diasporafoundation.org`

[8] `https://mastodon.social`

services provide 4 basic functions: ringing, tracking, locking and erasing your device.

These services are usually integrated in the device's operating system and continuously report its location to a server. Additionally upon request from the server the device will lock or erase erase all its contents. There are severe privacy and security implications the company that manages your device can not only track your movement, but has the power to erase all of your data from the device. The latter could be an efficient denial of service attack, especially when the device contains important information.

This aim of this project design an open source, free software alternative that allows end-users to securely control and monitor their devices without any third-party being able to track their movements or be able to perform denial of service attacks on their own devices. The first priority in the development of this protocol is the privacy and security of its users.

While device management software already exists, this project is innovative in the way it solves the problem. It introduces significant security improvements over existing solutions and last but not least it is open source.

1.1 Problem Description

To allow users to remotely and securely manage their devices this project needs to establish a protocol and a reference implementation of a service, that allows secure, encrypted bi-drectional messaging between an end user and a device. The devices in question are usually mobile phones, tablets and laptops.

These devices often have multiple means to connect to the Internet, like cellular network, WiFi or ethernet. Not only is intermittent connectivity an issue, but the connection method will tend to change. Additionally most connections including WiFi and cellular are extremely likely to use a NAT [1] server to translate IPv4 addresses or some sort of firewall for security reasons, so accessing your device from the Internet directly and reliably is becomes simply unfeasible.

The way this issue is solved is for connections instead of going from the end user to the device, to go through a relay server. The relay has a public address so connections even from behind NAT or a firewall to the relay server should under normal circumstances work unencumbered. The device then maintains a constant connection with the relay server waiting for commands and/or continuously reporting information. When the user wants to retrieve information or send a command to the device, they log into the relay server send the command which is then relayed to the device.

The main problem becomes obvious. The relay server is the component that actually has full control over your information and access to all of the information that gets passed along. When this relay server is ran by the end user this isn't a problem, as they already own it. But having to set up your own server is not a trivial issue and definitely not accessible to the average member of the public.

To solve that problem one needs to use end-to-end encryption. The communication through the relay server is passed along in its encrypted form from the end user to the device in such a way that the relay server cannot decrypt the information nor can it impersonate the user and issue commands on their behalf. It also shouldn't be capable of replay attacks. That is when the relay server records some of the cryptotext send through it that is correctly encrypted and signed and

later sends it verbatim to the user's device. If the communication protocol correctly accounts for such a malicious (or erroneous) behaviour the message should simply be dropped. At the end of the day the relay server should have little or no more insight about the nature or the contents of the messages than any router, switch or networking equipment along the chain that allows your message to travel through the Internet.

The user should also be able to obtain a confirmation receipt from the device proving that the relay server has not withheld relaying the message. Alternatively the relay server should only assist in connection establishment similar to how Web Real-Time Communication (WebRTC) or Session Initiation Protocol (SIP) works using the Interactive Connectivity Establishment protocol (ICE [2]) through a third-party (TURN [3]) server.

Cellular data is usually expensive and continuous reporting may accumulate charges. Additionally every data transmission will likely be draining the battery of mobile devices. A good such protocol would have to be highly bandwidth efficient. This will not only save cellular data costs, but it will also conserve battery and extend the battery life of the device, increasing recovery chances when it is lost.

Finally a requirement for security software is verifiability such that anyone should be able to inspect the code implementing the protocol and analyse it as opposed to security through obscurity.

To summarise, the protocol should:

1. Handle intermittent connections that will tend to change medium, public address and will often be behind a NAT and/or a firewall.
2. Provide end-to-end encryption.
3. Protect against replay attacks.
4. Provide message receipts.
5. Be bandwidth efficient.
6. Have an open protocol with free software implementations.

These are challenging and non-trivial features to achieve. This research is going to analyse the problems of each feature and propose an architecture and a communication protocol of a system the will achieve them.

Chapter 2

Background and Related Work

This chapter introduces the ethical design philosophy and the concept of end-to-end encryption. These are the core principles of this project and are critical to its development. The chapter concludes with an analysis of existing solution and especially where they fail with regards to ethical design, privacy and security.

2.1 Ethical Design

One of the primary aims of this project and one main conceptual difference compared to other similar projects is that it adopts the ethical design philosophy. Ethical design is at the core of the project and is the first consideration in every decision.

The goal of Ethical Design is to create technology that respects you as a human being. The design philosophy originates from the Ethical Design Manifesto published by Ind.ie[1]. Taken from the manifesto, ethical design refers to software that respects your Human Rights, Human Effort and Human Experience. These can be visualised as a hierarchy, three layers, three principles as they are dependent on each other.

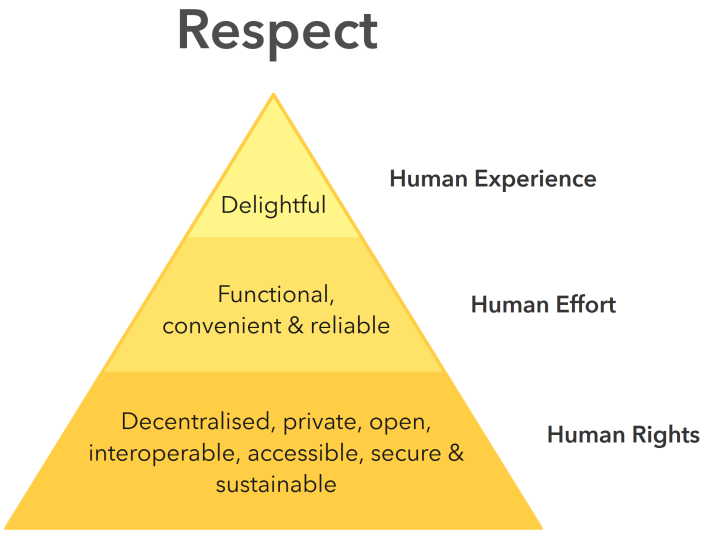

Figure 2.1: Ethical Design. License: CC BY 4 © Ind.ie

Human Rights refers to software that is decentralised, private and free. This is software that is inter-operable, widely accessible and most of all secure. It respects your civil liberties, privacy

[1] https://2017.ind.ie/ethical-design/

and security, it reduces inequality and benefits democracy. This means that often such software is peer-to-peer and end-to-end encrypted, distributed as free software and available for anyone to copy, modify, distribute or just deploy on their own.

Human Effort refers to technology that is functional, convenient and reliable, that respects different people and differently-abled. It respect you and the limited time you have and tries not to waste it. It is software that is thoughtful, accommodating and genuinely improving human life.

Human Experience refers to technology that is intuitive, fast and invisible. It sits quietly in the background yet it empowers you in your every-day life, and brings you joy.

From a practical perspective this is software that doesn't exploit your data or has some other unethical business model. It is software that is designed to truly empower people while respecting their rights. These principles do not only apply to individuals, but to businesses as well. The same security benefits a person would get from using a technology built by these principles would benefit a company as well. As such ethical design is a desirable trait and many businesses are proud to adhere to it.

2.2 End-to-end encryption

End-to-end encryption (E2EE) is key a feature of ethical design. It is a property of communication systems that makes eavesdropping practically impossible. The basic concept is that data travels encrypted and can only be found encrypted at the channel end points. It is

Lets consider an example where two clients are messaging each other over the internet through a server. Every message sent is encrypted is encrypted prior to transmission using a shared secret only the two clients are aware of. The message then travels in encrypted form through the internet. This includes all networking infrastructure along its path, whether its property of the internet service provider the clients are using or their corresponding governments. The message arrives at the server they are using and and remains in its encrypted form. The server too cannot encrypt it and just passes along a string of seemingly random bytes. The message then continues its journey travels along the internet until it reaches the other client. Only then is the message decrypted using the shared secret. Note that nobody along the message path is able to decrypt the massage except for the two clients sitting at the end points of the communication channel - thus end-to-end encryption.

End-to-end encryption is a very desirable feature that guarantees very high levels of privacy and security, however there are significant challenges a protocol needs to solve in order to achieve it.

With regards to ethical design, end-to-end encryption is the only way it could be guaranteed that information stored or transmitted through a system will remain private. Thus it is a key feature and one of the requirements of a highly secure system compliant to the ethical design philosophy. There are many ways E2EE can be achieved. It is not a specific technology, more of a methodology and an approach to how data is handled within a system. One of the main disadvantages is that E2EE systems treat data as a black box. They have zero knowledge of the contents or structure of the data and are only aware of its existence. Thus for example to provide lookup or search services in an E2EE service that has no knowledge of the data contained is a significant challenge.

2.3 Existing Solutions

One of the primary aims of this project is to adhere to the ethical design principles and is one of the primary characteristics that differentiates it from its competitors. Multiple software solutions exist to this date. Many cloud service providers and security companies including antivirus providers have developed similar services of their own. It is an important aspect of their software protection packages by protecting devices against physical attacks such when they are stolen. These solutions though do not have the privacy and security benefits of the protocol described in this project. Some of the most common shortcomings are proprietary software with closed source or using a relay server without proper end-to-end encryption. While most of these products are proprietary software and how they work internally is only a speculation, certain product features are often a dead giveaway of how they are internally structured. A true end-to-end encrypted service would have a separate client software, independent of the server, where the device data is decrypted. The rest of this section discusses how some of the most popular projects approached the problem and where they fall short in terms of security, privacy, usability and ethical design.

2.3.1 Google - Find My Device (a.k.a Android Device Manager)

The Find my Device[2] service from Google is one of the most popular examples. It is incorporated in billions of Android devices all over the world. Google's business model is collecting and using as much information for its users as possible in order to provide a targeted advertisement service. The company business model is a violation of the ethical design principles and works by disrespecting its users right to privacy. The products they offer are exclusively made to collect data about their users and the Find My Device product is no exception. It is proprietary software and does not use end-to-end encryption.

2.3.2 Apple - Find My iPhone

Apple's iCloud is one of the very few services that actually use the Secure Remote Password (SRP) [4] protocol. SRP is a type of password authenticated key agreement (PAKE) protocol, that allows a user to securely authenticate with a server without sending their password. This feature is key for a good implementation of Find My iPhone[3] and shows that Apple's engineers did indeed put a lot of thought in the privacy and security features of their product. Unfortunately as their service is proprietary software exactly how it works and if the software engineers did indeed implement it securely cannot be verified. As we've known Apple has been subject to much controversy when it comes to breached iCloud accounts. Another significant problem of their service is that it is only available on the iOS and MacOS operating systems, which means that it is almost exclusively available for Apple devices.

2.3.3 Open Device Manager for Android (ODM)

A very curious example is the Open Device Manager (ODM)[4]. It is a free software, open source device management application for Android devices with a corresponding web server written in PHP[5]. The project provides a set of desirable features including tracking SIM card swaps and

[2] https://www.google.com/android/find
[3] https://support.apple.com/en-gb/explore/find-my-iphone-ipad-mac-watch
[4] https://github.com/Fmstrat/odm
[5] https://php.net/

taking photos with the front and rear cameras. The project though does not use end-to-end encryption. The project uses a client server architecture and all information is received and collected by the server. This goes against the target the Secure Device Management Protocol (SDMP) sets to achieve where no intermediary device between the user and their mobile device has access to the data. And last but not least, as of now unfortunately the project is abandoned and non-functional.

2.3.4 Other services

Other not as popular services are Prey[6], AndroidLost[7], Wheres My Droid[8], Norton Anti-Theft[9]. They are all proprietary software and do not implement end-to-end encryption. Without end-to-end encryption the service provider can intercept messages, violating the ethical design principles. This means the service provide can potentially track its users and it can even impersonate them and issue commands to their devices on their behalf. This means ringing the device, locking or erasing it without the user's permission.

[6] https://preyproject.com/

[7] https://www.androidlost.com/

[8] https://wheresmydroid.com/

[9] https://support.norton.com/sp/en/nz/home/current/solutions/v59377890_NAT_Retail_1_en_us

Chapter 3

Requirements and Architecture

This chapter introduces the requirements of the protocol and the reasons behind them. It begins with a narrative in the form of user stories to help understand the motivations behind specific requirements. Further an overall architecture of the system is derived based on these requirements.

3.1 User Stories

User stories are a very helpful way to illustrate the possible use cases of the protocol. Along with several positive user stories, included are several negative, showing how such a service can be abused, however the SDMP protocol addresses these security issues of the device management problem.

3.1.1 A phone lost in a messy room

James is unable to find his phone. He remember it is somewhere in his room. He wants to ring it so he can locate it, but he doesn't have another phone and there is no one nearby who he can ask. Using his device manager client software on his laptop he can send a ring command to his mobile device. His phone then rings for 5 minutes at maximum volume allowing him to quickly recover it. When he finds his device under several books on his desk, he presses a button on the device to stop the ringing, indicating he located the device.

3.1.2 A phone forgotten at a cafe

Maya just got back from a cafe with her friends. She can't find her phone and she has no recollection where she left it. She rings it using the landline at her house, but she can't hear it. She then recalls that she installed a device management app on it. She logs in securely through a client software on her computer and issues a track command. Upon receiving the command the device uses its location service to acquire its location using Assisted GPS (A-GPS) and WiFi (or other available location services) and reports it back through a secure end-to-end encrypted channel. Maya's computer now show a map with an indicator showing exactly where the device is and a timestamp of when the location was acquired. Knowing where her phone is, Maya issues a lock command so no one at the cafe will be able to access her phone until she recovers it. The device then locks itself and requires a password to unlock.

3.1.3 A stolen corporate laptop

Hugo's laptop was stolen. He works at a financial institution and his corporate laptop contains sensitive information. His laptop is equipped with a GSM/GPS unit for remote management. Hugo immediately contacts his IT department who remotely wake up the computer using the

mobile modem and issue a track command via the device management software. The device then reports back its position, speed and direction of travel. The IT department calls the police to pass along the information and then issues a lock command that makes sure all data drives are encrypted and locked, so no data can be recovered by the perpetrators. In rare occasion when the data is too sensitive the IT department will choose to issue an erase command deleting all information from the data drive.

3.1.4 An evil third-party

The Microswift company decided to provide a service to its users that allows them to locate their devices if they have been lost or stolen. The service uses a relay server between devices and clients where all of the tracking data from the devices passes through. Another company Roogle realised that if they can tap into the relay server they can use the tracking information of the Microswift users for targeted advertising. They hired a team of Russian security experts to hack Microswift and provide a backdoor into the relay server. However since all of the information passing through the relay server is end-to-end encrypted this, the data the relay server has access to is meaningless and cracking it is unpractical.

3.1.5 Sensitive data blackmail

Mr. Blackwell uses the service from the Microswift company that allows him to erase his devices if they have been stolen so the perpetrators cannot access the information on the device, due to the sensitive nature of the information. A rogue server administrator working at the Microswift company decides to blackmail Mr. Blackwell telling him they will erase all of their data if a ransom isn't paid promptly. The server administrator plans to use the relay server the company uses to issue erase commands to user devices. Unfortunately for him, the protocol the company uses employs end-to-end encryption that prevents the relay server from impersonating users without knowledge of their password rendering his whole operation meaningless.

3.2 Goals

The motivation for this project is an upcoming GNU/Linux phone, with completely open source software. This is the Librem 5 device from Purism[1]. Purism is a social purpose company manufacturing computer hardware that adheres to the ethical design philosophy. Their latest project is the Librem 5 smartphone which would be the first successful smartphone project that was designed from the ground up as an open source software phone. For the project to be successful, the device needs to offer a rich set of features and services like the ones we are already used to, but the services need to adhere to the same principles. Namely, they need to be free software. A truly free device of course is able to run non-free software as well, as would be the case with the Librem 5, but to have a phone free of proprietary software the services we use the most also need to be free software.

With that in mind this research concluded that there is one service that has not yet been implemented according to the ethical design principles with up to date privacy and security features. The goals of the project are to implement protocol that describes such a service that has built in end-to-end encryption in is not susceptible to any known applicable attacks. The protocol needs to be extensible so that in the future it could be improved and be able to be backwards compatible.

[1] `https://puri.sm/products/librem-5`

Additionally I wanted to design the protocol to support purposes other than ringing, tracking, locking and erasing a device, so that it can be used as a event mechanism for clients to send messages to a device, no matter the message.

3.3 Requirements

The goals of the protocol along with the problems outlined in Chapter 1 loosely define a set of requirements on the protocol design. To formalise the problem the protocol needs to solve, a more solid set of functional and non-functional requirements is necessary. The requirements listed here attempt to describe the problem as loosely as possible without forcing a given architecture or technologies. The idea is that the protocol needs to achieve the goals in the purest most elegant way possible. Each requirement is followed by a paragraph that elaborates on why is it there and the problem that it is there to solve.

3.4 Functional Requirements

FR-1: End-to-end encryption – The protocol must provide a way for communication with remote devices over the internet using a high security end-to-end encryption and signature.

One of the key requirements to ethical design is end-to-end encryption that guarantees the privacy and security of the user. As such it is the most important functional requirement. A signature is also required as each message should not only be authenticated, but it should also be tamper-proof. A modified message will no longer have a valid signature. This measure also serves as an error detection system and ignore messages containing an error.

FR-2: NATs and firewalls – The protocol must work when devices are behind a NAT and/or a firewall limiting incoming connections.

Most WiFi networks, cellular data networks and the majority of IPv4 networks use a private network and a Network Address Translation (NAT) [1] server to convert a private network address into a public network address. Additionally because of IT security concerns, most businesses set up a firewall disabling incoming connections to their internal network. This means that for all practical purposes in such networks only outgoing connections work.

FR-3: Intermittent Connections – The protocol must work and recover gracefully in the case of on intermittent connection or when the device suddenly switches its public address.

A mobile device may switch its connectivity method at any time due to coverage issues and/or availability of a certain medium. Along with the new connection method, its public address on the internet will change. Additionally even if the device was able to listen on ports on its public address to that point, that is no longer guaranteed, as the new connection may be behind an even stricter NAT and/or a firewall (see FR-2).

FR-4: Replay attacks – The protocol should withstand replay attacks and ignore packets that have already been received and just being replayed.

Since likely the protocol is going to be message based, each message will be cryptographically signed and encrypted. Each individual message is thus self sufficient and if recorded and replayed later verbatim along with its signature, a device may interpret it as a valid

message and execute it. The relay server could then easily perform a replay attack by just recording messages relayed through it. The protocol needs to be designed in such a way as to prevent replay attacks and allow identification of such malicious messages.

FR-5: Late message delivery – The protocol should be able to store and deliver messages at a later time when the device is currently offline or outside network coverage.

Often devices may be turned off, or simply outside of network coverage. Without network connectivity commands shouldn't be lost. The protocol needs to save messages in permanent storage and deliver them at a later point in time, when connection with the device has been re-established.

FR-6: Receipt confirmations – The protocol should implement receipt confirmations when a message has been received and/or executed.

Message receipt confirmation is important for several reasons. Firstly it can help detect connectivity issues along the system. Secondly once a message receipt is confirmed the client user interface can indicate that the given message or command is successfully executed and the user could rest assured that his device executed his command. Last but not least, confirmation receipts can help detect and mitigate a possible attack where the relay server intentionally does not forward messages.

FR-7: Multiple relay servers – The protocol should allow for multiple relay servers to be employed simultaneously.

Multiple relay servers would increase the reliability of message delivery in the case when a relay server is experiencing downtime or it has been compromised.

FR-8: Storage Limits Expiration – The protocol should communicate relay server data storage limits per user. Additionally the relay server may discard old messages, but should clearly communicate message discard timeout, which should be no less than 30 days. If the server has discarded messages it should notify the user.

As specified in FR-5, the relay server needs to store undelivered messages as often mobile devices may be turned off or outside network coverage. Of course a system without storage limits can be easily abused. On the other hand a system that does not communicate these limits well and suddenly starts discarding messages is unreliable. A careful balance between the two is important.

FR-9: Multiple device support – The protocol should support multiple devices per user.

A user should be able to have multiple devices per account.

FR-10: Backwards and forwards compatibility – The protocol should be designed in such a way as to support backwards and forward compatibility.

The initial version of the protocol won't likely be flawless and amendments and improvements will be required in the future. For this reason the protocol should include some feature for version checking that allows backwards and forwards compatibility.

FR-11: Arbitrary messages – The protocol should allow sending of arbitrary messages used by different applications.

The protocol should add support for sending arbitrary messages intended for different applications and services. This means that it could be used simultaneously for tracking your phone as well as sending SMS text messages via your phone from your computer. The protocol needs to implement some application identification feature so it can later send the message to the correct app.

FR-12: Password storage – The user's password should be stored securely so it cannot be extracted even in the case when an attacker has physical access to the user's device.

This requirement is important as the protocol is designed to be used when for example the user's device had been stolen. In that situation an attacker will have physical access to the device and would be able to recover the user's password if unprotected.

3.5 Non-functional Requirements

NFR-1: Free Software – The protocol, its reference implementation and all associated documentation should be distributed under a free software license.

Ethical design playing a key part in the development of this protocol, this requirement is extremely important. Ethical software must be released as free software so that users can study it, change it, distribute it and run it freely. This requirement implies the usage of a strong free software license.

NFR-2: Exceeded Limits – The protocol should allow for the relay server to specify how to limit message storage. That is on a temporal basis, storage limitation or maximum message rate. If devices exceed these limits the relay server needs to produce a meaningful error and gracefully fail.

This non-functional requirement elaborates additional requirements to FR-8.

NFR-3: Message length optimisation – The protocol should use a minimal message headers and efficient format as to minimise transmission time and processor usage.

Mobile devices will often rely on cellular data or WiFi. There are several important points here. First, cellular data is not cheap and its usage should be minimised. Secondly wireless transmissions reduce battery life. A protocol that has minimal message length and requires fewer CPU cycles to encode and decode would consume less battery. In the case where the protocol is used to locate a stolen device, this will increase the time the device will be able to transmit its location thus increasing the chances for its successful recovery. One other aspect is denial of service attacks (DoS). Another factor is that the smaller bandwidth and processor time required to process a single message, the more messages a relay server can process thus increasing its efficiency. This also decreases denial of service vectors.

NFR-4: Single password – The protocol should use only a single password for all cryptographic operations.

The aim of this protocol is to provide an elegant solution that doesn't require users to remember multiple passwords. This is a severe constraint as this password needs to be used for both authentication with the relay server and message encryption and signing. A very careful cryptographic algorithm is necessary to solve this issue with a single password.

NFR-5: Widely used technologies – The protocol should employ only widely used, verified and well established technologies.

Verified technologies refers to software that has been verified to work correctly. For example a verified cryptographic library or algorithm that has been well studied and does not have any vulnerabilities. The use of widely used and well established will assist in adopting the protocol and would allow developers to more easily implement it in their software.

3.6 Architecture

This section introduces the overall architectural decisions of the protocol, such as how data is handled and what components the system should have. This doesn't include specific decisions regarding technology choices and encoding method as those are covered in Section 4 - Protocol Design.

3.6.1 Message based or stream based communication

An important decision for the protocol is how data should be treated - i.e. whether it should be in the form of streams or messages. Both approaches have different advantages. A stream based approach is more suited for applications with large quantities of real-time data. It can handle large files and high data bandwidth. A message based protocol on the other hand is more suited for short irregular messages. Given the nature of the protocol, short irregular messages are the more likely scenario. It is also the most efficient approach, both in terms of power usage and computation.

3.6.2 Relay Server

Requirement FR-2 sets a very strict limit on the architecture of the protocol. Depending on the type of NAT and/or firewall a technique known as hole punching could be employed to bypass them. This is for example how Web Real-Time Communication (WebRTC) or Session Initiation Protocol (SIP) works using the Interactive Connectivity Establishment protocol (ICE [2]) work. Using an external STUN [5] server it is possible to detect the type of NAT or firewall employed and to check if NAT traversal is possible. If it is possible a relay server is used to so the client and the device could exchange connection information such as ports and public addresses. Then the client and the device can connect directly without the need of a relay server anymore. More often than not this proves to be impossible. Either because the network uses a very strict firewall or a very restrictive NAT, where traversing and hole punching is impossible. In these situations a relay server is mandatory. An extension of the STUN protocol is the TURN [3] protocol. It provides both STUN services, but also provides packet relay for clients that are unable to connect to each other directly.

Employing the TURN protocol is efficient and if configured correctly it is a very powerful instrument for peer-to-peer connections. But employing TURN adds a significant complexity and makes implementation really hard. Instead a simple architecture with a simple message relay server would be much more easier to implement. For this reason one of the main architecture

decisions is to use a relay server. This means that the device has to connect to the server and maintain its connection constantly. The system then has three important components - a device, a relay server and a software client (Fig. 3.1). The latter is the user interface software program a user would use to interact with their device remotely.

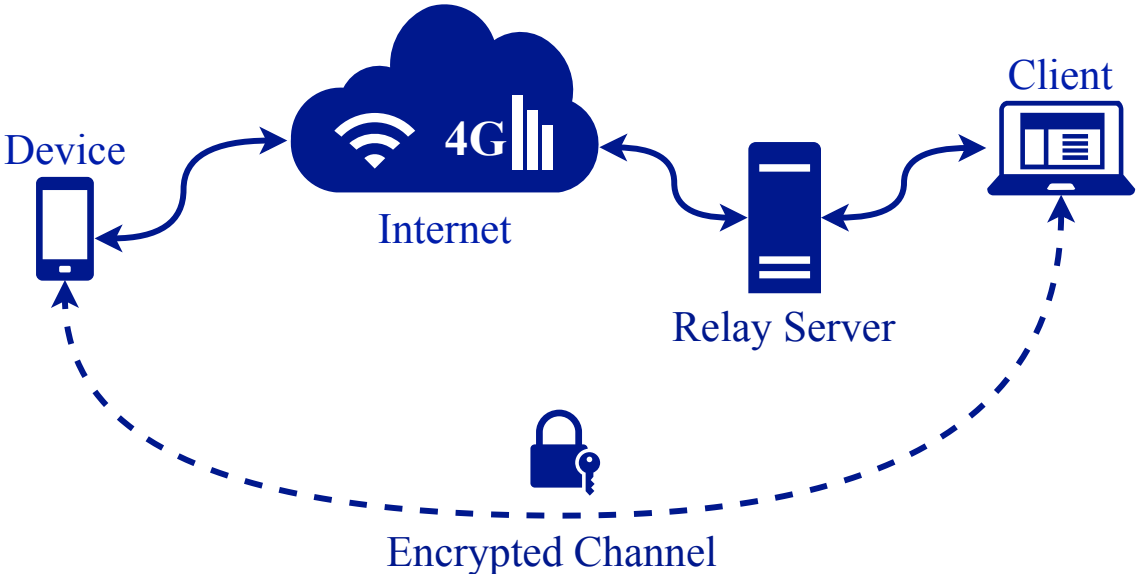

Figure 3.1: Architecture Diagram

3.6.3 End-to-end encryption

Traditional systems establish a secure connection to the relay server using Transport Layer Security (TLS) [6]. The data sent by the device is then decrypted and stored in plaintext on the server. Ethical design dictates that data should be end-to-end encrypted. This means that in the relay server, messages should be encrypted. They are stored in their encrypted form and unreadable to the relay server. A separate software running on a user's computer than pulls messages from the server and decrypts them. Note that the messages only exist in their decrypted from on the device and the client, both of which are controlled by the user.

The protocol uses a password to encrypt and sign messages. Without knowledge of that password the relay server is unable to decrypt the message content. Furthermore as messages are both encrypted and signed messages are also tamper evident. An attempt to modify the message would render the signature invalid. While tamper evident is sufficient for security, to increase reliability the protocol also needs to be tamper proof. The protocol solves this by allowing a device to use multiple independent relay servers, which if ran by independent entities would make separate paths a message could travel. An attacker will need to control all relay server to stop messages from reaching their destination.

3.6.4 Password

Traditionally end-to-end encrypted services such as online messaging require two sets of keys. The first set are the keys/passwords you use to authenticate with the service provider. The second set of keys are individual keys used to encrypt/decrypt messages with a given person. These keys are usually stored on the users computer and are not kept on the service provider's server. This is a problem as every time you need to use the service from a different device, these keys need to be

securely transferred from an existing device that has them to the new device. It can be easily seen how this is unpractical in this scenario, when the assumption is that the user no longer has control of their device and needs to locate it.

One solution is to simply use two passwords. One for authenticating with the relay server and one for encrypting/signing messages. The first key would be known by the server while the second only by the device and the client. But this creates an extreme inconvenience for the user. It is likely that a significant portion of users will opt to use the same password, rendering the security of their device compromised. Instead a way to solve the problem with a single password is necessary, while still maintaining the same security characteristics.

A solution that uses only one password would mean that the authentication with the relay server needs to happen without the server knowing the user's password. This is a very interesting cryptographic problem.

While such a requirement at first seems strange and intuitively one might think that it would be impossible. After all how can one authenticate with a server without the server knowing their password? This problem has two components. First you need to authenticate without sending your password over the network and second the server has to authenticate the user without knowing their password. But such problems have been solved. An example are solutions to the Socialist Millionaire Problem. There is also a class of protocols that allow you to authenticate without the server ever knowing your password. I will discuss two possible solutions.

A very simple solution would be to hash the password and use the hash to authenticate with the server. We can use a hash function specifically made for hashing passwords. An example of such function is bcrypt [7]. Bcrypt uses a salt to protect against attacks using rainbow tables [8]. To protect against brute force attacks it also uses an adaptive function that can increase the complexity of the hashing step over time to match the increase in computing power. A solution like this seems sufficient, but it has several issues. While this scheme is sufficient to shield the user's password, if an attacker obtains the password hash they would still be granted access to the relay server.

A better solution would be if the protocol employs the Secure Remote Password [4] protocol. It is a type of Augmented Encrypted Key Exchange (EKE) [9, 10] protocol or a password authenticated key exchange protocol. It has two steps. The first step is registration with a server. During this step a password verifier is computed using a Diffie-Helman [11] like process based on a password hash. This password verifier is then sent to the server along with the user's username. Note that this is a one-way operation and the verifier contains no information about the password itself. The second step is authentication. During the authentication step another process similar to a Diffie-Helman key exchange is used so both sides can obtain a shared secret. The end result is authentication without any knowledge of the password with the added benefit of forward secrecy as a new shared secret is computed upon connection. This protocol is immune to both passive and active man in the middle attack (MITM) unlike the traditional Diffie-Helman key exchange where an active MITM can spoof the exchange and intercept the connection. A version of the Transport Layer Security (TLS) [12] exist that uses SRP for authentication. The TLS-SRP protocol [13] is the perfect solution that allows transport layer security with SRP authentication.

By authenticating with the relay server using TLS-SRP the user's password will thus never

reach the server and it could be safely used for encrypting and signing messages. The protocol assumes that both the client software and the device software have knowledge of the password. TLS-SRP is an industry standard technology defined in (RFC 5054) [13] with implementations in several major cryptography libraries including OpenSSL and GnuTLS.

3.6.5 Clients and Devices

There is a major difference in the requirements and desirable features of the individual communication channels between devices and the relay server and clients and the relay server. For example, communication between a device and the relay server has to be very bandwidth efficient and processing messages should incur minimal processor time. This is because data transferred from a mobile device may be expensive and wireless transmission and data processing consume power and reduce the device battery life. These limitations do not necessarily apply to the client software. The client on the other hand needs to be available for multiple platforms and needs an API accessible easily from multiple programming languages and technologies.

As the requirements of the two communication channels are different, the protocol treats them separately and defines two distinct methods for communication. These are described in more detail along with the specific differences in Chapter 4.

Chapter 4

Protocol Design

This chapter introduces specific protocol decisions such as formats, encoding methods and technologies along with justification for each choice. These extend further on the architecture introduced in Chapter 3

The SDMP protocol aims to provide general purpose message passing interface that could be used for arbitrary messages, command and remote procedure calls (RPC). While the original purpose of the protocol is ringing, tracking, locking or erasing the data of a device, the protocol could be used for any kind of message service that requires high security end-to-end encrypted non-ephemeral message transfer.

4.1 TCP or UDP

Having decided on a message based protocol in Chapter 3, the next important decision is whether connections should be in the form of TCP [14] sockets (essentially data streams) or UDP [15] datagrams (messages). Upon first review it seems that a message based protocol would be best suited by UDP datagrams, however TCP has several important advantages. The most important characteristic of TCP is reliability. TCP ensures packet delivery, consistency, packet order and performs error checks on the packet data. UDP however does not performs any such checks and does not guarantee packet delivery at all. Since reliability is of high importance for the protocol, TCP is the most suitable choice even though it is a stream based protocol.

4.2 Ports

The protocol will use ports 2565, 2566 and 2567. The avoid collisions, the ports were selected using the true random number generator service provided by Random.org[1] and verified that they don't collide with existing well-known ports. The ports a server uses can be overridden using a DNS SRV [16] record. Client and device software should always check for the existence of such record prior to connecting to a relay server.

The three ports 2565, 2566 and 2567 are used for TLS-SRP device API, TLS-SRP HTTP client API and HTTPS registration API respectively.

4.3 Message attributes

Messages in the protocol have several key attributes. These are its protocol and its contents. The message protocol is a unique identifier of the application or service this message belongs to. This

[1] `https://www.random.org/`

attribute is required by FR-11 to support arbitrary messages using different format for different purposes.

Messages also have a version attribute. This attribute is important for future and backwards compatibility. If the relay server contains messages from different versions of the protocol, the version attribute will help differentiate them and handle them appropriately.

Messages also include a timestamp. The timestamp is there for several reasons. Firstly it can be used to filter messages by date and pull messages from the relay server that have not yet been delivered, and secondly it can be used to mitigate replay attacks depending on the message content and its purpose.

4.4 Message priority

The protocol identifies three different types of messages in terms of importance:

Low Priority Automated regular status report messages such as periodic tracking.

Medium Priority Non-critical requests initiated by a user and their responses. Example: A ring command.

High Priority Critical requests initiated by a user and their responses. Example: track, lock, or erase commands.

Because a relay server may impose soft limits like maximum storage and maximum transmission rate on the user messages, the relay server needs to identify which messages should be prioritised and which messages could be discarded.

The server is allowed to discard any message older than 30 days as specified by FR-8. However if the user has exceeded their limitations in a period of less than 30 days the server will be forced to discard some messages. The protocol specifies a strict order when discarding messages according to priority and timestamp. Messages are discarded by timestamp, oldest first, Lowest priority messages are discarded first and then if this is insufficient medium priority messages can be discarded too. If this too is insufficient high priority messages can be discarded as well, though the protocol recommends the usage of soft limits and a slightly higher limits for high priority messages.

4.5 Password hash function

The SRP protocol requires a password hash function to operate. Additionally to secure passwords against physical intrusion into devices, devices side software implementation are advised to store passwords in their hashed form. Since the same password hash step is required for both TLS-SRP authentication and message signing and encryption it makes sense to optimise computation and simplify the protocol by using the same hash function.

This hash function needs to be strong and should have a sliding computational cost to mitigate vulnerabilities against brute force attacks. RFC 8018 [17] recommends the usage of the Password Based Key Derivation Function 2 (PBKDF2) [17] for password hashing. The standard also recommends using a salt with length of at least 64 bits. The US Institute of Standards and Technology recommends 128 bits. This protocol will use salts of length of 32 bytes (256 bits).

While most services use an iteration count of less than 10,000 for the protocol to remain secure even when a user has chosen a weak password 100,000 iterations should be more appropriate. Any higher number would significantly degrade the performance of the protocol on mobile processors.

The output length of the PBKDF2 should be set to 256bit so it can be reused as input for message encryption and signing.

Note that clients need to store the password hashed according to the specifications above. This is done so that the user's plaintext password could not be extracted from a device even when the device has been physically compromised as specified in FR-12. When performing a TLS-SRP handshake the stored hash is used as an input in the hash function, essentially doubling the amount of PBKDF2 iterations to 200,000.

4.6 Message encryption and signing

Message encryption should be done using the user's hashed password as an input. The encryption cipher recommended is the 256 bit version of the industry standard Rijndael also known as Advanced Encryption Standard (AES-256) [18]. Note that the output of PBKDF2 should be set to 256 bits so it could be used as input to to AES-256.

Message signing should be performed using the industry standard method for message authentication - the hash-based message authentication code (HMAC) [19]. HMAC requires a hash function as input. The hash function selected for this protocol is the US NIST recommended 256 bit SHA-2 (SHA-256) [20]. SHA-256 is a widely supported hash function that has been incorporated into hardware on most modern CPUs including mobile processors. Using a hash function with hardware support would significantly benefit the performance of message signing. Additionally as it uses a reduced number of processor instructions to compute it uses smaller amounts of energy and contributes to battery savings.

4.7 Client side

In the protocol there are two direct communication channels (see Fig. 3.1) – one from the client to the relay server and one from the device to the relay server. The protocol thus needs to describe both channels. These channels have different requirements associated with them and require significantly different approaches.

The first communication channel is from the client to the relay server. This is where the user registers and creates an account on the server. This is also where they will login and issue commands to their device. The client software is the piece in the system the user will use when they lose possession of their device. It may be installed on another device of theirs or a friend's device or a public computer (like in a library). Thus this software needs to be available for multiple platforms including desktop, mobile and event web platforms. Developers can use the protocol for other arbitrary services where a communication with a remote device based on a single password is necessary. For these reasons, it is important that the client to relay server API uses easy to learn, popular and well supported technologies. This API does not need to be bandwidth efficient because restrictions like constant cellular data usage and battery life don't apply on the client side.

There are multiple technology choices available here, but the key argument is cross platform and cross-language interoperation. One of the platforms that dictates the most restrictions is the

web. Web-based applications are HTTP based and use data encoding protocols like XML and JSON. HTTP, XML and JSON are technologies that are so widespread, that there are very good, fast and efficient implementations in almost any language.

The choice between XML and JSON is not as important, but an analysis of the two can help identify the more efficient encoding method. A research paper by Zunke and D'Souza [21] shows that JSON outperforms XML in all performance, memory and size (even with compression enabled) at the expense of flexibility when it comes to complex attributes, etc. But since the messages communicated along this API are going to be quite simple in nature, the flexibility XML offers is not a concern.

Thus I chose to use an HTTP based RESTful API using JSON encoding with the idea that it will make development of clients for all platforms including web based clients significantly easier. JSON has very CPU efficient implementations and could be compressed using technologies like.

The client side of the protocol has two separate APIs. The first is the registration API. The second is the control API. There is a subtle difference between how each of them operates which will be analysed separately.

4.7.1 Registration API

During the registration step, the user cannot use a TLS-SRP session as there is no established password verifier with the server yet. Instead the registration API uses traditional TLS via HTTPS. To register the client needs to send a JSON POST request to the root path of the relay server on its HTTPS port as defined in Section 4.2.

Registration is made according to the SRP protocol. The client needs to pick a small random salt. The recommended salt size as defined in Section 4.5 is 32 bytes. The client then needs to compute its SRP password verifier and send all three along with a username of choice. The server then needs to verify that the username is unique and register the new account on the server.

A relay server may optionally collect additional information for account password recovery purposes such as email. It is important to note that since the protocol is end-to-end encrypted, a recovered password will only allow re-establishing connection to the relay server. The new password will not be accepted by the device, as encryption and message authentication will fail. The relay server may also employ anti-spam protection challenge such as an image verification.

A brief reference of the registration API can be found in Appendix B.

4.7.2 Client side API

The client side API uses a TLS-SRP socket. Upon connection the client needs to authenticate to the API based on the SRP protocol. Instead of the user's password the client needs to send the user's password after running it two consecutive times through the hash function specified in Section 4.5 bringing the amount of total PBKDF2 iterations to 200,000.

The client side API is an HTTP API encoded using JSON similar to the registration API. A brief reference of the API can be found in Appendix C.

4.8 Device side

The communication between the device and the relay server is a far more interesting scenario. Since mobile devices will often have limited memory, CPU, battery life and high data transfer costs it is very important to optimise this side of the protocol as much as possible. Messaging over

this channel will be over the raw TLS-SRP socket with a very efficient encoding method. Kazuaki Maeda [22] analyses the output size and performance cost of twelve different methods for data encoding. Out of those the method that has one of the best performances, language support and efficiency is Google's Protocol Buffers. Protocol Buffers is a binary encoding method with an emphasis on simplicity and performance. The method uses a schema to define message formats and has the added benefit of being backwards and forwards compatible. The latter is an important feature and is one of the protocol requirements - FR-10.

The Protocol Buffer schema files of the Device API can be found in Appendix A. To understand them an analysis of the significance of each attribute is required. A `Device` represents a single device software and is identified via an id, a user friendly name and an International Mobile Equipment Identity number (IMEI) [23] if available. The id may be based on the device hardware or a randomly generated UUID. It needs to uniquely identify a device within the user's account. All device attributes but the id are sent encrypted so that the relay server cannot infer any information about the device. When a device connects for the first time to a relay server it will send a Device record with an id, a user friendly name and if the device is a mobile phone or has a modem - its IMEI number. The relay server will than return a `CommandResponse` message with either OK status or an error. If this is a new connection, but not a new device registration the device needs to only send its id as to associate the active session with the corresponding device.

The device can also request a list of pending undelivered messages using the `MessageRequest` command. The relay server will return `|`CommandResponse`|` with status error or a `MessageSet` containing messages marching the specified criterion.

4.8.1 Modes of operation

The device software can be toggled between continuous or on-demand mode of operation. In continuous mode location data data should be periodically transmitted while in on-demand mode transmission should occur only when tracking is requested by the client by issuing a track command.

4.8.2 Defending against replay attacks

To defend against replay attacks, the device needs to establish a way to keep track of messages it has already received and identify them.

There are multiple solutions to this problem. The classical cryptography textbook solution requires a challenge-response authentication of the client. In the context of this protocol this isn't possible. Often messages will be received late and the device and client might not even be online at the same time to perform a challenge response authentication. Due to the nature of the protocol, a defence of against relay attack is quire hard to achieve.

One way to solve this problem works in the following way: The client generates a random message id, signs it and sends it to the device. The device than keeps track of all message ids that were already used. Obviously this method would require to store 32 bit ids for every packet received. Over time this data will accumulate to the point where it is no longer possible to store it. An improvement of this algorithm uses the message timestamp to prune old messages. Let's assume that we are not interested in messages that arrived more than a week ago. Then the algorithm works in the following way:

1. Delete all messages ids from the index that have a timestamp smaller than 1 week ago.
2. Check if the new message id is in the index
3. If it is - discard the message
4. If it isn't - it's a valid message

But even with this optimisation, the algorithm would need enormous space requirements. If we use 32 bit ids a 512MB bitmap indicating whether an id has been used in the last week would be required in addition to storing each of the messages timestamps. This algorithm is thus not suitable for mobile platforms due to its high storage requirements.

Another way to defend against relay attacks is to keep track of a message counter. A valid packet will have a message counter greater than the current message counter. Usually this strategy will work but when there are two or more clients simultaneously attempting to message the device this strategy will fail (see Fig. 4.1). In this scenario it is possible for a race condition to occur and two packets with the same or smaller counter to arrive at the relay server at the same time.

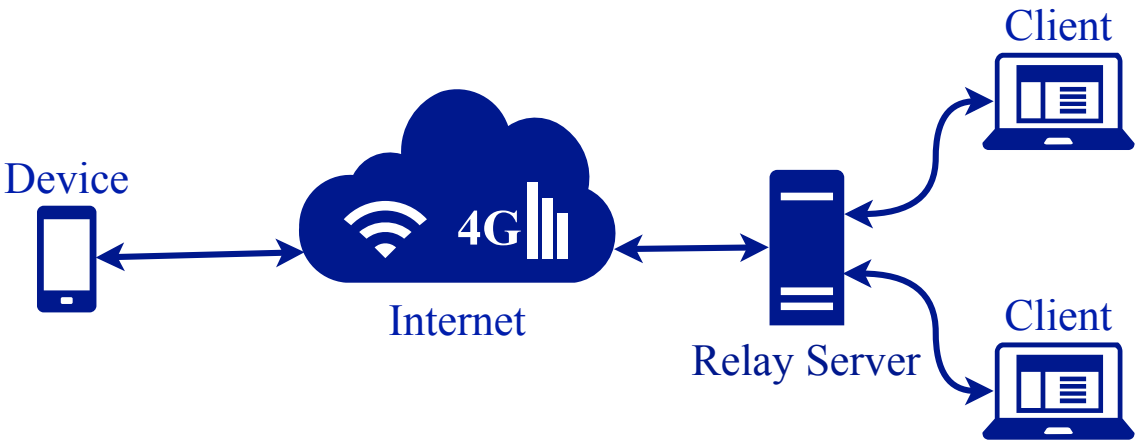

Figure 4.1: Multiple clients connected to the same device.

A modification to the relay server that assigns a unique id (with respect to a single user account) to each client would allow to bypass this issue. Every single client would then have a unique identifier and a separate counter. The device software will need to keep track of multiple counters, but the storage requirements of this scheme are negligible. This is how the algorithm would work:

1. When receiving a message pull from the message the client id and the message counter.
2. Lookup a small local database the current counter for that client.
3. If the current counter is equal or larger than the message counter - discard the message and message the client to notify it that there has been a collision. This message should have low priority.
4. If the client counter is smaller than the message counter - it's a valid message.
5. Store the new counter for the corresponding client id.

With this modification, multiple clients can send simultaneous messages to the same device without the possibility of counter collision and without the possibility of replay attacks.

4.8.3 Encrypting and authenticating messages

When sending a message the device must follow these steps:

1. Constructs a `Message` object and populate it.
2. Serialises the object and converts it to string.
3. Encrypt that string using the user's hashed password using AES-256.
4. Compute the HMAC-SHA-256 of the encrypted message string.
5. Construct a `SignedEncryptedMessage` object and populate it with the encrypted message, the computed hmac and the public message attributes (client id, timestamp and priority).
6. Increment the message counter
7. Serialise `SignedEncryptedMessage` object and send the message to the relay server.

This process renders messages unreadable to a third party and makes them tamper evident. When verifying a new message the reverse procedure is followed:

1. Deserialise the received message.
2. Construct `SignedEncryptedMessage` from the deserialised message.
3. Validate the encrypted message HMAC against the stored user password hash.
4. Decrypt the message.
5. Run replay attack detection algorithm.
6. Deserialise it into a `Message` object.
7. Save the message counter to the corresponding client id
8. Process the message.

4.8.4 Device re-connection and multiple relay servers

A device can be connected to multiple relay servers simultaneously. If any of those connections are lost however, the device has to reconnect. The TLS-SRP protocol demands a new session initialisation every time a new connection is established. However the complexity of the TLS-SRP session initialisation is minimal and would not consume significant resources.

By using more than a single relay server, the system is more fault tolerant. In case of a relay server downtime or a completely compromised relay server, as long as there is at least one relay server that functions correctly the protocol will continue to function well.

When there is more than one relay server available, if a relay server suddenly starts transmission large quantities of invalid messages, whether replayed or just with an invalid HMAC the device needs to disconnect from the server and wait a certain amount of time before reconnecting. This feature mitigates denial of service attacks initiated from a rogue relay server.

4.9 Relay Server

The relay server is one of the most important components in the system. In addition to hosting the registration, client side and device-side APIs, the server also stores the most recent messages. The storage method employed is implementation specific and the protocol itself does not enforce a given implementation.

One important task for the relay server is to assign a unique id (with respect to the user's account) to any connecting client. This is an important task as this client id is a key feature in the algorithm that protects the protocol against replay attacks. The protocol does not force a given algorithm for generating these id, however an internal connection counter could be employed for the task. It is important that these ids are unique for each client logged in with the corresponding user account.

4.10 Securing an account after it has been compromised

The protocol forces devices to store the user's password hashed with the user's salt. In the even the device has been compromised even temporarily an attacked could have extracted the password hash from the data storage unit of the device. In this situation the protocol is compromised as the user will be able to authenticate with the relay server and log into the user's account. Additionally they will be able to decrypt all messages as they will be in possession of the user's password hash which is the key used for encryption and signing messages.

At this point the user has to reset their password to make their device secure again. But since the attacker never had access to the original user plaintext password, it is sufficient to just regenerate the salt and verifier while keeping the same password. The security of the system will remain the same, but the user won't have to change their password. This could be implemented conveniently with a single click of a button on either the client or the device.

4.11 Design summary

Overall the protocol can be described as an end-to-end encrypted publishâĂŞsubscribe service, that stores messages for later retrieval. The choice of technology is heavily dictated by implementation requirements. The protocol introduces security features to protect itself against replay attacks and employs measures to protect the user's plaintext password even in the event the user's device is completely compromised.

Chapter 5

Implementation

This chapter discusses an attempt to create a reference implementation of the SDMP protocol. The most important and most complex component of the protocol is the relay server. This chapter focuses on the development, the challenges involved and the technologies selected to implement it.

5.1 License

As the primary principle of this project is ethical design, the reference implementation of the protocol needs to be free software. The choice of license for free software project is important is it may dictate how the project will evolve or fail. Using a copyleft [24] license would mean that every modification of the software must also be published under the same terms. This would help the development of the protocol as individual and businesses who use it and adapt it will contribute back their modifications allowing the implementation to grow.

One of the strongest such licenses is the GNU General Public License (GPL) v3 [25]. Richard Stallman, the creator of The Free Software Foundation recommends releasing software under the latest or future versions of the GNU GPL[1] to avoid incompatibility with programs released under a different GPL version.

Another important aspect in terms of licensing is that all open source project used in the development of the reference implementation must be distributed under a compatible license.

Taking all of these into arguments into consideration, the SDMP project will be released under the terms of the GNU GPL v3 or later.

5.2 SRP Protocol

The SRP protocol is unfortunately not supported by many cryptographic libraries. Additionally TLS-SRP is only supported by OpenSSL and GnuTLs.A detailed table with all of the major SRP libraries can be found in Table 5.2.

As shown in Table 5.2, the only libraries with TLS-SRP support are OpenSSL and GnuTLS. They are both distributed

5.3 C and C++

Both OpenSSL and GnuTLS are C libraries which forced the implementation of the relay server to be written in C and C++. While this meant that it will take significantly longer to develop a

[1] `https://www.gnu.org/licenses/identify-licenses-clearly.html`

Library	Language	TLS-SRP	Website
OpenSSL	C	**Yes**	`https://www.openssl.org/`
GnuTLS	C	**Yes**	`https://www.gnutls.org/`
Botan	C++	No	`https://github.com/randombit/botan`
go-srp	Go	No	`https://github.com/opencoff/go-srp`
GNU crypto	Java	No	`https://www.gnu.org/software/gnu-crypto/`
pysrp	Python	No	`https://github.com/cocagne/pysrp`
clipperz	Javascript	No	`https://github.com/clipperz/javascript-crypto-library`
srp-rb	Ruby	No	`https://github.com/lamikae/srp-rb`

Table 5.1: SRP Library support comparison

prototype, compared to other more modern languages, C and C++ have the advantage of being extremely fast and memory efficient.

A major disadvantage of writing a security software in C and C++ is security. A developer needs to be constantly on alert double checking code for buffer overflows or denial of service possibilities. Functions like `strcpy` are strictly forbidden. User input needs to be carefully treated and validated so that no security vulnerability is introduced. A great source for methods and advice on security is the book by D. LeBlanc and M. Howard - Writing Secure Code [26]. Last but not least one needs to be extremely careful with pointers, remembering to free their memory as introducing memory leaks in C and C++ is extremely easy.

5.4 HTTP, HTTPS and REST library

The relay server implementation of the registration and client side APIs require an HTTP REST framework for C++ that could be integrated to work with TLS-SRP. One excellent such project is Pistache[2]. The Pistache performance is several times greater than alternatives like the HTTP utilities in the Boost[3] and the Qt[4] frameworks. Upon initial testing, Pistache serving HTTP and HTTPS requests from memory was able to process more that 18,000 request per second on commodity hardware. Even when latency is artificially introduced to simulate network delay and at extremely high concurrency Pistache still manages to perform at similar level (drops to 17,000 request per second). These results were amazing and extremely promising so the project was adopted to provide the HTTP infrastructure required.

5.5 TLS-SRP integration

Initial attempts to implement the TLS-SRP support was done using the OpenSSL SRP API. The OpenSSL library however has several important issues. Firstly, the SRP API is very poorly documented with no code examples. Secondly the OpenSSL implementation only loads SRP credentials (username, salt, verifier, etc) from a file. Using the OpenSSL API to provide the SRP parameters dynamically from a relational database for example proved to be very hard. Finally, OpenSSL developers plan to drop the SRP API in the future. These issues forced this project to re-implement the TLS-SRP using GnuTLS.

While GnuTLS isn't documented that much better compared to OpenSSL and it too does not

[2] `http://pistache.io/`
[3] `https://www.boost.org/`
[4] `https://www.qt.io/`

have good up to date examples, implementation with GnuTLS initially seemed significantly easier compared to OpenSSL. Additionally the GnuTLS API seemed significantly richer with regards to SRP support. Though the implementation did not go without issues.

5.6 Data Storage, Databases and Object Relational Mapper

Given the relatively simple structure of messages, users and devices this project would be best suited by a simple relational database. A non-relational database would only introduce additional complexity. Alternative approaches to data storage such as files are not applicable because the protocol specification require message filtering by timestamp and priority. A relational database would provide both the performance, flexibility and filtering capabilities required for the implementation of the relay server.

Because server administrators should be able to choose the database engine they want to run on, the reference implementation should use an object relational mapper with drivers to support most relational database management systems, such as PostgreSQL[5], MySQL[6], MariaDB[7], Amazon RDS[8], etc.

Most object Relational Mappers (ORM) for low level languages like C++ are usually proprietary software and are not available for free, nor are they available under a free software license. Nevertheless a very good such solution is ODB[9] by Code Synthesis. ODB is distributed under the terms of the GNU GPL v2. This created a problem as this license is incompatible with the license selected for this project (GNU GPL v3 or later). However CodeSynthesis agreed to provide a waiver for the SDMP project to use ODB while still being able to distribute the protocol and its reference implementations under the terms of the GNU GPL v3 or later.

5.7 Implementation method

The implementation of the relay server started from the GnuTLS SRP client and server examples. This ensures that the base implementation is correct in terms of security and checks required to make sure the TLS-SRP socket is established correctly according to security standards. The examples were then extended and adapted to include several servers for the completely different registration API and client side API.

The first major problem was rewriting the Pistache TCP/SSL transport classes to add support for TLS-SRP. Originally Pistache was designed to use exclusively OpenSSL. Porting it to use another cryptography library was a significant and time consuming challenge. At every step the GnuTLS documentation had to be carefully reviewed to make sure there weren't any missing important steps and all return values and error codes were properly handled.

Significant problems were encountered due to the lack of good documentation of how GnuTLS allocates and frees memory. These issues were with the

During the beginning of April, a bug in the GCC packages on some Linux distributions made compiling more complicated projects practically impossible. The parts of the software the GCC

[5] `https://www.postgresql.org/`
[6] `https://www.mysql.com/`
[7] `https://mariadb.org/`
[8] `https://aws.amazon.com/rds/`
[9] `https://www.codesynthesis.com/products/odb/`

bug affected were the build systems of the Pistache and the ODB library. There were multiple compatibility problems during both compilation and link time. Pre-compiled versions of both Pistache and ODB failed to work with newly compiled code.

The bug took two weeks to resolve, before Debian Buster[10] released GCC v8 prematurely in attempt to mitigate the problem. This indeed solved the issue and development of the SDMP reference implementation was resumed.

5.9 Implementation status

Currently the core components of the system are almost completed. In particular the Pistache library was almost completely ported to work with GnuTLS instead of OpenSSL. This is important because as was mention earlier, the OpenSSL SRP support and API is inferior to the one provided by GnuTLS.

Apart from the core functionality of the relay server, an extremely basic implementation of both the device software and the client software are ready.

The SDMP protocol was designed to be as simple as possible, but to achieve its strict security and privacy features it has a lot of components that require significantly more work to implement. While still mostly incomplete, the reference implementation show great results in terms of performance, memory usage and capacity to handle extremely large volume of devices on a single machine.

[10] `https://wiki.debian.org/DebianBuster`

Chapter 6

Testing and Evaluation

This chapter will focus on analysis and evaluation of the protocol rather then its reference implementation. Specifically it will evaluate how well the protocol manages to solve the requirements specified in Chapter 3 and how it achieves the project goals. Further while the reference implementation is still not complete it will discuss testing methods to guarantee high software quality free of security issues.

6.1 Requirement Compliance

This section will go over each requirement and will discuss how the SDMP protocol addressed and solved it.

FR-1: End-to-end encryption SOLVED **–** The protocol must provide a way for communication with remote devices over the internet using a high security end-to-end encryption and signature.

The protocol successfully implements end-to-end encryption. Messages are encrypted using the user's password hash using AES-256 and a high security password hashing function.

FR-2: NATs and firewalls SOLVED **–** The protocol must work when devices are behind a NAT and/or a firewall limiting incoming connections.

Due to its architecture using a relay server the protocol bypasses the problem that incoming connection behind a NAT or a firewall are impossible. Instead devices create an outgoing connection to the relay server and maintain it indefinitely.

FR-3: Intermittent Connections PARTIAL **–** The protocol must work and recover gracefully in the case of on intermittent connection or when the device suddenly switches its public address.

The protocol mandates that devices should reconnect upon connection loss and requires TLS-SRP session renegotiation. The protocol does not specify a more efficient approach to re-establish a connection.

FR-4: Replay attacks SOLVED **–** The protocol should withstand replay attacks and ignore packets that have already been received and just being replayed.

The protocol successfully protects against replay attacks by using a message counter based algorithm to identify replayed messages.

FR-5: Late message delivery SOLVED – The protocol should be able to store and deliver messages at a later time when the device is currently offline or outside network coverage.

The protocol includes specification to store messages on the relay server whenever a device or a client is temporally unavailable.

FR-6: Receipt confirmations PARTIAL – The protocol should implement receipt confirmations when a message has been received and/or executed.

While the protocol does not specify an explicit receipt confirmation scheme, receipt confirmations could easily be implemented by returning a message to the client with appropriate content The protocol is designed as a general purpose messaging protocol and does allow bi-drectional communication.

FR-7: Multiple relay servers SOLVED – The protocol should allow for multiple relay servers to be employed simultaneously.

The protocol does define the requirement for support for multiple relay servers and the corresponding architecture.

FR-8: Storage Limits Expiration SOLVED – The protocol should communicate relay server data storage limits per user. Additionally the relay server may discard old messages, but should clearly communicate message discard timeout, which should be no less than 30 days. If the server has discarded messages it should notify the user.

The protocol does specify a method for communicating storage and rate limits and strict rules for discard messages based on priority and age (timestamp).

FR-9: Multiple device support SOLVED – The protocol should support multiple devices per user.

The protocol does include support for multiple devices and the corresponding APIs.

FR-10: Backwards and forwards compatibility SOLVED – The protocol should be designed in such a way as to support backwards and forward compatibility.

The SDMP protocol includes a version number in all messages and APIs so that version information can be communicated easily. It also uses data encoding methods that can easily be changed without breaking backwards compatibility.

FR-11: Arbitrary messages SOLVED – The protocol should allow sending of arbitrary messages used by different applications.

The protocol does specify a method for sending arbitrary messages. It uses a `protocol` field that identifies the "protocol"and format messages use the application they belong to so the device software can dispatch them to the correct service.

FR-12: Password storage SOLVED – The user's password should be stored securely so it cannot be extracted even in the case when an attacker has physical access to the user's device.

The protocols stores the user's password hashed so even in the event of a physical intrusion the original password could not be restored.

NFR-1: Free Software SOLVED – The protocol, its reference implementation and all associated documentation should be distributed under a free software license.

The protocol, all of its documentation and reference implementation are distributed under the terms of the GNU GPL v3 or later license. The SDMP is open source and free software.

NFR-2: Exceeded Limits SOLVED – The protocol should allow for the relay server to specify how to limit message storage. That is on a temporal basis, storage limitation or maximum message rate. If devices exceed these limits the relay server needs to produce a meaningful error and gracefully fail.

The relay server implementation is free to choose how this is handled as long as it communicates the limits through the API.

NFR-3: Message length optimisation SOLVED – The protocol should use a minimal message headers and efficient format as to minimise transmission time and processor usage.

Devices communicate through an efficient binary encoding (Protocol Buffers) which is compact and high performance.

NFR-4: Single password SOLVED – The protocol should use only a single password for all cryptographic operations.

The protocol relies only on a single password thanks to the elegant authentication solution using the Stanford Secure Remote Password (SRP) protocol.

NFR-5: Widely used technologies SOLVED – The protocol should employ only widely used, verified and well established technologies.

The protocol uses only widespread technologies like TLS-SRP, AES-256, SHA-256, PBKDF2, Protocol Buffers, HTTP, HTTPS and JSON.

6.2 Testing and evaluation of the reference implementation

While the reference implementation is far away from being completed and would require a large amount of time and effort before it completely implements the SDMP protocol as specified in this research, we can discuss possible methods to test and validate individual components of the software.

6.2.1 Correct TLS-SRP implementation

To test whether the relay server implements the TLS-SRP protocol correctly one can use an external TLS-SRP client like the one built in the OpenSSL

To evaluate whether the reference implementation is compliant with the protocol definition an extensive black box test suit could be employed. For a given set of messages and requests each component is expected to react and output specific data. For example, sending an invalid registration request should output an error with an HTTP status code of 400 (Bad Request). Furthermore more sending a tracking message from a client should result in a corresponding response from the device containing its GPS coordinates. Of course in a test environment services like an external location service should be stubbed to return specific test control data so we can analyse the rest of

the system. Additionally the tests should not only cover valid behaviour but invalid as well and observe if the system returns corresponding error messages and fails gracefully.

6.2.3 Code security and accidental vulnerabilities

The reference implementation is written in C an C++. Because of how low level these languages are they known for how easy it is to introduce a security vulnerability like remote code execution or a denial of service attack due to negligence. In C and C++ it is trivial to make mistake that causes buffer overflows or memory leaks.

To protect against such errors, the software needs to implement an extensive white box test suit that test the behavior of individual mission critical parts of the system. Especially parts that interact with user data. These tests need to test how functions behave when provided with edge cases such as incorrectly or maliciously formatted data. A classical example is sending a payload much larger than the specified length. When a developer uses a function like `strcpy` [26] that does not terminate at a given length but when it finds a terminating zero character in the input stream they can introduce a buffer overflow attack that could potentially lead to denial of service, remote code execution or information leakage. In fact dangerous function like `strcpy` and `strcat` should be blacklisted and testing should fail if similar functions are used in the code.

Critical sections of the code need to be analysed and tested extremely carefully. Additionally white box testing for allocation and dealloction of critical resources (this includes cases like signal handling, exception handling, etc). Resources need to be deallocated carefully to prevent memory leaks and white box testing can help eliminate such issues.

Chapter 7

Conclusions, Discussion and Future Work

7.1 Threat model and attack vectors

A threat model analysis is an important part of a security protocol. A threat model is a structured way to describe an analyse the security of a system. This section attempts to describes the attacks the protocol protects against and the possible attack vectors the protocol is vulnerable to. Using this architecture there are several ways to attack the protocol, but as we will see the security provided is well above any similar technologies.

7.1.1 Strengths

Communication between from devices or clients to the relay server is encrypted using TLS-SRP. The SRP algorithm is an augmented EKE protocol and is invulnerable to both passive and active man in the middle (MITM) attacks.

Furthermore messages passed between a device and a client are encrypted and signed using the user's password and are thus end-to-end encrypted and tamper evident. This way the protocol also prevent the relay server from impersonating the user and issuing rogue commands.

While in transit messages actually use two layers of encryption, once encrypted with the user's password and once by the TLS-SRP connection. An attacker without access to the relay server would have an even harder time reading or tampering with messages.

The protocol also employs measures to protect against replay attacks by tracking message ids and timestamps.

7.1.2 Password attacks

The first and most trivial attack would be on the password itself. Physical access to either the device or the hardware running the client may be used to obtain the password and compromise the protocol. Unfortunately this is an attack a software protocol simply cannot protect against. It is generally accepted that if an attacker has physical access to a machine the machine is compromised. However to protect the actual user's password it is never stored in its plaintext from. Instead only a hash/digest is stored, so even though an attacker with physical access to the devise will be able to compromise the protocol, they will not be able to recover the user's password.

If an attacker who has access to the relay server can record a message they can attempt a dictionary attack to try and recover the user's password. A dictionary ultimately attack exploits weaknesses in the password itself. While the user's password choice is outside of the protocol control, it can still mitigate dictionary and brute-force attacks by employing a strong and slow to compute hash function.

7.1.3 Denial of Service

The relay server can conditionally choose to reject and/or delay the transmission of messages. This is partially mitigated by the support for multiple relay server. If one of the relay servers is compromised the other will likely not be. However an attacker with full control of all relay servers employed by the user will be able to execute a denial of service attack.

Another way for a relay server to create a very subtle denial of service attack is to assign multiple clients the same client id. This will cause packets to be systematically rejected due to counter collisions.

Additionally an attacker with close proximity to the device can also jam its wireless transmission. This is also an attack the protocol cannot protect against as it also is a physical level attack.

7.1.4 Overall analysis

Overall the security of the protocol well is above what is currently used. It hold well to the ethical design standards and provides high levels of encryption and guarantees privacy. It defends itself well against man in the middle and replay attacks. It also uses sign messages as to provide tamper evidence and it mitigates denial of service attacks. Devices and clients are vulnerable to physical level attacks but even then the protocol takes measures to protect the user's password.

7.2 Improvements

Currently the protocol uses a polling method to pull messages on the client side. The protocol could be refined to provide an alternative solution using WebSockets [27] in addition to the polling method. This would allow clients to receive messages in real time.

In terms of cryptography an alternative cryptographic protocol to the SRP that has been recently published is the OPAQUE [28] protocol. It is also a PAKE protocol that would allow authentication with zero password knowledge. The protocol though is very new and not mature enough as of now, but has the promise to work as a suitable replacement to SRP.

It has been shown by Shor [29] that cryptography algorithms based on the integer factorization problem, the discrete logarithm problem or the elliptic-curve discrete logarithm problem can be easily solved by a quantum computer using Shor's algorithm. The SRP protocol itself is no exception and it like most popular cryptography algorithms employed today is vulnerable to attacks using a quantum computer. Fortunately as of now, quantum computing seem to have reached an engineering barrier and the industry consensus is that it will take at least another 15-20 years before the cryptography employed today is actually vulnerable. Nevertheless, there have been proposed alternatives like the RLWE-SRP [30] protocol which are post-quantum secure. Additionaly the RLWE-SRP has been shown to be significantly faster than SRP. Unfortunately RLWE-SRP is not yet part of any official standards and has not yet been verified properly. Thus an implementation based on it is not without risks, but is definitely worth investigating in the future to help protect the security of this protocol in the post-quantum world.

7.3 Future Work

I have secured the `https://sdmp.dev` domain name where all of the information about the protocol will be hosted, as well as the code for its reference implementation and links to client software.

As the protocol was created with the idea to extend the features of the upcoming open source, free software GNU/Linux mobile phone - the Librem 5, from Purism. I plan to implement a fully protocol compliant reference implementation relay server, a graphical user interface client programs and a device background service for GNU/Linux distributions, all of which will be distributed via the website.

The key device implementation of the system will be extracted into a library developers can use in their own applications to provide and extend services for use an all operating systems. The protocol has great potential and developers are encouraged to implement it for Android and iOS.

7.4 Conclusion

The Secure Device Management Protocol (SDMP) is open source, free software protocol that uses easy to implement popular technologies. It is designed to be computationally inexpensive and bandwidth efficient in order to extend the battery life of mobile devices. It is flexible, extensible and future-proof. The security of the protocol is well above what the industry currently uses and its adoption would most certainly benefit many individual and businesses. The SDMP protocol will make a great addition to people who value their privacy and security. It is the first ethically designed protocol for device management and will successfully contribute a missing functionality to the ever growing array of free software cloud services.

References

[1] Jonathan Rosenberg. Ip network address translator (nat) terminology and considerations. RFC 2663, IETF, 1999.

[2] Jonathan Rosenberg. Interactive connectivity establishment (ice): A protocol for network address translator (nat) traversal for offer/answer protocols. RFC 5245, IETF, 2010.

[3] Philip Matthews, Rohan Mahy, and Jonathan Rosenberg. Traversal using relays around nat (turn): Relay extensions to session traversal utilities for nat (stun). RFC 5766, IETF, 2010.

[4] Thomas D Wu et al. The secure remote password protocol. In *NDSS*, volume 98, pages 97–111. Citeseer, 1998.

[5] P. Matthews D. Wing J. Rosenberg, R. Mahy. Session traversal utilities for nat (stun). RFC 5389, IETF, 2008.

[6] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, IETF, 2018.

[7] Niels Provos and David Mazieres. A future-adaptable password scheme. In *USENIX Annual Technical Conference, FREENIX Track*, pages 81–91, 1999.

[8] Philippe Oechslin. Making a faster cryptanalytic time-memory trade-off. In *Annual International Cryptology Conference*, pages 617–630. Springer, 2003.

[9] Steven M Bellovin and Michael Merritt. Encrypted key exchange: Password-based protocols secure against dictionary attacks. In *Proceedings 1992 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 72–84. IEEE, 1992.

[10] Steven M Bellovin and Michael Merritt. Augmented encrypted key exchange: a password-based protocol secure against dictionary attacks and password file compromise. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 244–250. ACM, 1993.

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

[12] T. Dierks E. Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, IETF, 2008.

[13] N. Mavrogiannopoulos T. Perrin D. Taylor, T. Wu. Using the secure remote password (srp) protocol for tls authentication. RFC 5054, IETF, 2007.

[14] DARPA Internet program. Transmission control protocol. RFC 793, IETF, 1981.

[15] J. Postel. User datagram protocol. RFC 768, IETF, 1980.

[16] P. Vixie L. Esibov A. Gulbrandsen, Troll Technologies. A dns rr for specifying the location of services (dns srv). RFC 2782, IETF, 2000.

[17] B. Kaliski K. Moriarty, B. Kaliski. Pkcs 5: Password-based cryptography specification version 2.1. RFC 8018, IETF, 2017.

[18] Joan Daemen and Vincent Rijmen. Aes proposal: Rijndael. 1999.

[19] R. Canetti H. Krawczyk, M. Bellare. Hmac: Keyed-hashing for message authentication. RFC 1997, IETF, 1981.

[20] Wouter Penard and Tim van Werkhoven. On the secure hash algorithm family. *Cryptography in Context*, pages 1–18, 2008.

[21] Yanpei Chen, Archana Ganapathi, and Randy H Katz. To compress or not to compress-compute vs. io tradeoffs for mapreduce energy efficiency. In *Proceedings of the first ACM SIGCOMM workshop on Green networking*, pages 23–28. ACM, 2010.

[22] Kazuaki Maeda. Performance evaluation of object serialization libraries in xml, json and binary formats. In *2012 Second International Conference on Digital Information and Communication Technology and it's Applications (DICTAP)*, pages 177–182. IEEE, 2012.

[23] 3GPP. Numbering, addressing and identification. Technical Specification (TS) 36.331, 3rd Generation Partnership Project (3GPP), 01 1999. Version 14.2.2.

[24] The Free Software Foundation. What is copyleft?, 2018. `https://www.gnu.org/copyleft`. Accessed: 2019-05-23.

[25] Free Software Foundation. Gnu general public license. `http://www.gnu.org/licenses/gpl.html`.

[26] David LeBlanc and Michael Howard. *Writing secure code*. Pearson Education, 2002.

[27] A. Melnikov I. Fette. The websocket protocol. RFC 6455, IETF, 2011.

[28] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. Opaque: an asymmetric pake protocol secure against pre-computation attacks. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 456–486. Springer, 2018.

[29] Peter W Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM review*, 41(2):303–332, 1999.

[30] Xinwei Gao, Jintai Ding, Jiqiang Liu, and Lin Li. Post-quantum secure remote password protocol from rlwe problem. Cryptology ePrint Archive, Report 2017/1196, 2017. `https://eprint.iacr.org/2017/1196`.

Appendix A

Device API

.

A.1 Protocol Buffers Schemas

```
message Device {
    string id = 1;
    string encrypted_name = 2;
    string encrypted_imei = 3;
}
```

```
message Message {
    int32 version = 1;
    int32 counter = 2;
    string protocol = 3;
    string content = 4;
    int32 timestamp = 5;
    int32 client_id = 6;
}
```

```
message SignedEncryptedMessage {
    enum Priority {
        LOW = 0;
        MEDIUM = 1;
        HIGH = 2;
    }

    int32 version = 1;
    string cryptotext = 2;
    string hmac = 3;
    int32 timestamp = 4;
    Priority = 5;
}
```

```
message MessageSet {
    repeated SignedEncryptedMessage = 1;
    int32 page = 2;
    int32 total_pages = 3;
    int32 total_results = 4;
    enum Status {
        OK = 0;
```

```
        ERROR = 1;
    }
    Status status = 5;
    string error = 6;
}
```

```
message MessageRequest {
    int32 timestamp = 1;
    int32 page = 2;
    int32 results_per_page = 3;
}
```

```
message CommandResponse {
    enum Status {
        OK = 0;
        ERROR = 1;
    }
    Status status = 1;
    string error = 2;
}
```

Appendix B

Registration API

B.1 Registration request

```
POST / HTTP/1.1
```

```
{
    "challenge_id": 6456253,
    "challenge_response": "bf73de",
    "username": "itay-grudev",
    "verifier": "SGNqMzgyNDN1amZwbWRza0loYTQoKiQl",
    "salt": "NDN1amZwbWRza0loYTQoKiQldWR5dGFz"
}
```

On success response

```
HTTP/1.1 200 OK
```

```
{
    "status": "ok"
}
```

Error response

```
HTTP/1.1 400 Bad Request
```

```
{
    "status": "error",
    "error": "A very friendly message explaining what went wrong."
}
```

B.2 Anti-spam (CAPTCHA) request challenge

```
GET /challenge HTTP/1.1
```

```
{
    "id": 6456253,
    "challenge": "https://sdmp.dev:2566/6456253.jpg"
}
```

Appendix C

Client side API

C.1 Version checking

```
GET /status HTTP/1.1
```

```
{
    "version": "0.1",
    "salt": 32,
    "pbkdf2_iterations": 100000,
    "hmac_hash": "SHA-256",
    "account_storage_limit": "50MB",
    "account_max_message_rate": "0.3m/s",
}
```

Note: the status API is also available in the registration API.

C.2 Resetting the user's SRP password verifier and salt

```
POST /reset HTTP/1.1
```

```
{
    "current_verifier": "qMzgyNDN1SGNqMzgza0loYzaSGNqMTQo",
    "new_verifier": "SGNqMzgyNDN1amZwbWRza0loYTQoKiQl",
    "new_salt": "DNDN1amZwSGNqMzgyNKiQRza0loYTGFz"
}
```

On Success

```
HTTP/1.1 200 OK
```

```
{
    "status": "ok",
}
```

On Error

```
HTTP/1.1 500 Internal Server Error
```

```
{
    "status": "error",
    "error": "Invalid verifier."
}
```

C.3 Listing devices

```
GET /devices HTTP/1.1
```

Parameters:

page Optional argument. Show the specified page of results. Defaults to: 1.


```
{
    "length": 2,
    "total_pages": 1,
    "total_results": 2,
    "devices": [
        {
            "id": "60c15aa8-6ecd-48c3-8ff5-48973ebc7abd",
            "encrypted_name": "NjBjMTVhYTgtNmVjZC00OGMzLThmZjUtNDg5NzNlYmM3YWJk
    ",
            "encrypted_imei": "mVjZC00OGMTVg5NzNlYmM3YmV4LWZjUtNc4LWNmDg5g5NzNk
    "
        },
        {
            "id": "5ad5fab1-839a-4f78-b532-c1e9bd8f3608",
            "encrypted_name": "NjBjMTVhYTgtNmVjZC00OGMzLThmZjUtNDg5NzNlYmM3YWJk
    ",
            "encrypted_imei": "LThmZjUtNDg5NzNlYmM3YWJkNjjc4LWNmVjZC00OGMz"
        }
    ]
}
```


C.4 Deleting a device

```
DELETE /device/{device_id} HTTP/1.1
```

Parameters:

device_id Device UUID

On Error

```
HTTP/1.1 200 OK
```


```
{
    "status": "ok",
    "device": {
        "id": "5ad5fab1-839a-4f78-b532-c1e9bd8f3608",
        "encrypted_name": "NjBjMTVhYTgtNmVjZC00OGMzLThmZjUtNDg5NzNlYmM3YWJk",
        "encrypted_imei": "LThmZjUtNDg5NzNlYmM3YWJkNjjc4LWNmVjZC00OGMz"
    }
}
```


On Error

```
HTTP/1.1 500 Internal Server Error
```


```
{
    "status": "error",
    "error": "Unable to delete device."
}
```

C.5 Retrieving messages from a device

```
GET /device/{device_id}/{timestamp} HTTP/1.1
```

Parameters:

device_id Device UUID

timestamp Pull messages sent after the specified timestamp. Defaults to 300 seconds ago.

page Optional argument. Show the specified page of results. Defaults to: 1.

per_page Optional argument. Show the specified number of results per page. Defaults to: 50

Note: If the server does not accept the specified per_page parameter the default value will be used.

```
{
    "length": 1,
    "total_pages": 1,
    "total_results": 1,
    "messages": [
        {
            "version": 1,
            "cryptotext": "
    M2JjNGEzODctOGU3YS00N2Y0LTgwYmYtYzg1Y2NkNGIxODA4OzE7ZmluZDsxNTU4NjEyNzk5O1lTM
    ",
            "hmac": "M2JjNGEzODctOGU3YS00N2Y0LTgwYmYtYzg1Y2NkNGIxODA4OzE",
            "priority": "medium",
            "timestamp": 1558612799
        }
    ]
```

On Success

```
HTTP/1.1 200 OK
```

```
{
    "status": "ok",
}
```

On Error

```
HTTP/1.1 500 Internal Server Error
```

```
{
    "status": "error",
    "error": "Unable to relay message. Please try again later."
}
```