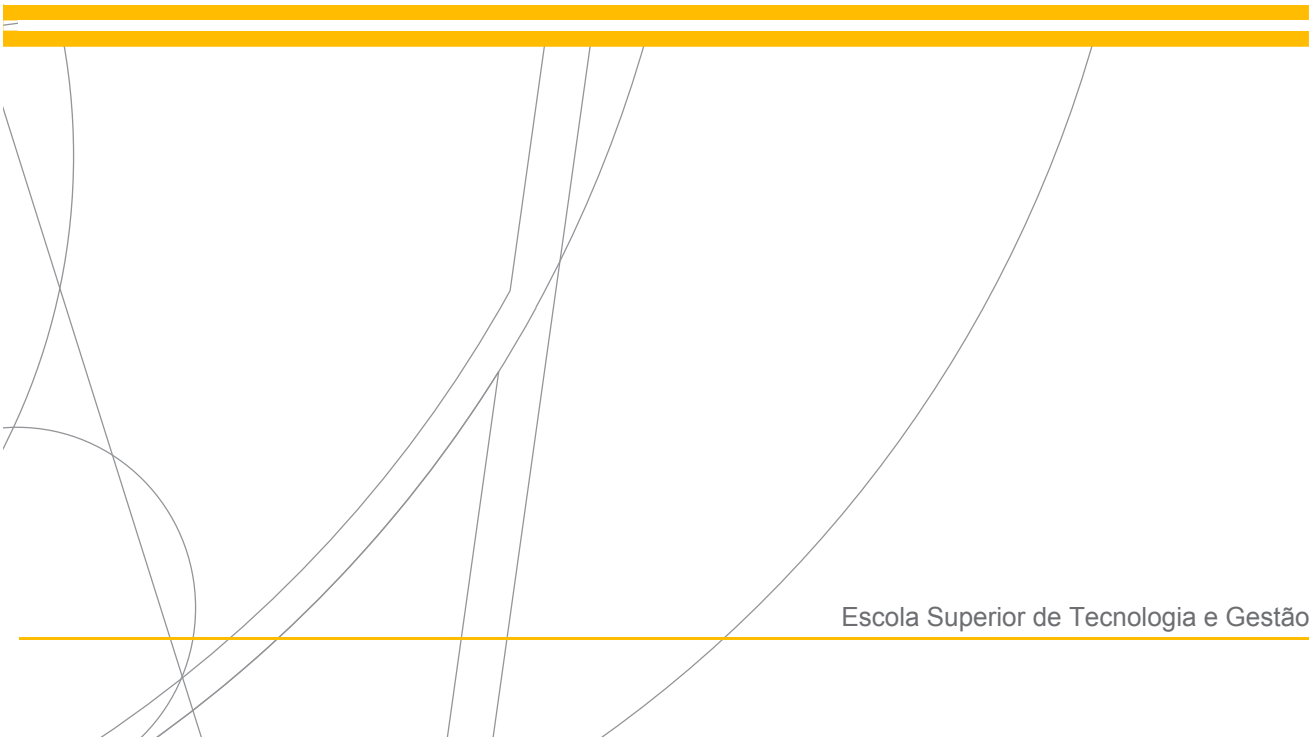
Instituto Politécnico
de Viana do Castelo

EXPLOITING ONLINE SERVICES TO ENABLE ANONYMOUS AND CONFIDENTIAL MESSAGING

Pedro Sousa



Escola Superior de Tecnologia e Gestão

Instituto Politécnico
de Viana do Castelo

Pedro Guilherme Alves de Sousa

EXPLOITING ONLINE SERVICES TO ENABLE ANONYMOUS AND CONFIDENTIAL MESSAGING

Nome do curso de Mestrado

Mestrado em Cibersegurança

Trabalho efetuado sob a supervisão de

Professor António Pinto e Professor Pedro Pinto

Novembro de 2022

Mestrado em Cibersegurança
Master in Cybersecurity

Exploiting Online Services to Enable Anonymous and Confidential Messaging

a master's thesis authored by

Pedro Guilherme Alves de Sousa

and supervised by

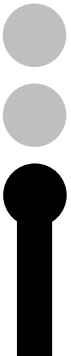
António Alberto dos Santos Pinto
Professor Coordenador, IPP

Pedro Filpe Cruz Pinto
Professor Adjunto, IPVC

This thesis was submitted in partial fulfilment of the requirements for the Master's degree in Cybersecurity at the Instituto Politécnico de Viana do Castelo

ipvc

18 of February, 2023



Abstract

Messaging services are usually provided within social network platforms and allow these platforms to collect additional information about users, such as what time, for how long, with whom, and where a user communicates. This information enables user identification and is available to the messaging service provider even when communication is encrypted end-to-end. Thus, a gap still exists for alternative messaging services that enable anonymous and confidential communications and that are independent of a specific online service. Online services can still be used to support this messaging service, but in a way that enables users to communicate anonymously and without the knowledge and scrutiny of the online services. In this paper, we propose messaging using steganography and online services to support anonymous and confidential communication. In the proposed messaging service, only the sender and the receiver are aware of the existence of the exchanged data, even if the online services used or other third parties have access to the exchanged secret data containers. This work reviews the viability of using existing online services to support the proposed messaging service. Moreover, a prototype of the proposed message service is implemented and tested using two online services acting as proxies in the exchange of encrypted information disguised within images and links to those images. The obtained results confirm the viability of such a messaging service.

Keywords: Covert. Anonymous. Communication.

Resumo

Serviços de envio de mensagens instantâneos são normalmente fornecidos por plataformas de rede social e permitem que estas plataformas recolham informações adicionais sobre os utilizadores, como a que horas, por quanto tempo, com quem e onde um utilizador comunica. Esta informação permite a identificação do utilizador e está disponível para o prestador de serviços mesmo quando a comunicação é encriptada de ponta a ponta. Assim, existe ainda uma lacuna para serviços de mensagens alternativos que permitem comunicações anónimas e confidenciais e que são independentes de um serviço online específico. Os serviços online ainda podem ser utilizados para apoiar este serviço de mensagens, mas de uma forma que permite aos utilizadores comunicarem de forma anónima e sem o conhecimento e escrutínio dos serviços online. Neste artigo, propomos mensagens usando esteganografia e serviços online para apoiar comunicações anónimas e confidenciais. No serviço de mensagens proposto, apenas o remetente e o destinatário estão cientes da existência dos dados trocados, mesmo que os serviços online utilizados ou outros terceiros tenham acesso aos contentores de dados secretos trocados. Este trabalho revê a viabilidade de utilizar os serviços online existentes para apoiar o serviço de mensagens proposto. Além disso, um protótipo do serviço de mensagens proposto é implementado e testado usando dois serviços online agindo como proxies na troca de informações encriptadas escondidas dentro de imagens e links para essas imagens. Os resultados obtidos confirmam a viabilidade de tal solução.

Palavras-chave: Encoberto. Anónimo. Comunicação.

Acknowledgements

This thesis is the result of a year of effort and hard work, which could not be achieved without the help and support of several people, to whom I want to show appreciation.

I would like to express my deepest gratitude to Professor António Pinto and Professor Pedro Pinto, my supervisors, for all the guidance, knowledge, patience and feedback they provided me during this year.

I am also thankful to my classmates and colleagues, with special mentions to Silvino, Pedro and Ricardo, who always showed availability and willingness to help with issues I found throughout the master's degree and, by extension, this thesis.

Lastly, I'd like to mention my family - parents, brothers and grandparents, who never once stopped supporting my choices in life and are always there to help when I make mistakes - as well as my friends, with special mentions to Eduardo, Bryan, André and Gustavo, who provided me with endless moral support and always kept my spirits up.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

CBC Cipher-Block-Chaining

CCA Chosen-Ciphertext Attack

CPA Chosen-Plaintext Attack

EXIF Exchangeable Image File Format

GDPR General Data Protection Regulation

HMAC Hash-Based Message Authentication Code

HTML HyperText Markup Language

IM Instant Messaging

ISP Internet Service Provider

IV Initialization Vector

JS JavaScript

KDF Key Derivation Function

LSB Least-Significant-Bit

MCyber Master in Cybersecurity

MLS Messaging Layer Security

OTP One Time Pad

PFS Perfect Forward Secrecy

PoC Proof of Concept

SMS Short Message Service

SRG Secure Random Generator

TLS Transport Layer Security

TOR The Onion Router

TTL Time To Live

URL Uniform Resource Locator

UUID Universally Unique Identifier

VPN Virtual Private Network

WN Wireless Network

Chapter 1

Introduction

The human being is extremely social - every day we face a requirement to interact with others, be it for personal, professional or even entertainment matters. With the advances of the digital age, a focus has been developed in regard to social interactions. One of the main results of these developments was Instant Messaging (IM) services, allowing users to keep in touch with family, friends and colleagues if, for example, they're far away or even when physical distancing is necessary (COVID-19 pandemic) [22].

That said, in most of these services, the messages exchanged between their users when these exchanges happen, with whom and even where the users were at the time of sending the message are collected and may be analyzed, scrutinized and profiled [12].

This thesis presents a viable design solution that offers one-to-one anonymous and covert messaging by using steganography and a combination of other online services as proxies for data transfer. Future work entails expanding the design to allow for group chats.

Short Message Service (SMS) and IM are two choices of digital communication, commonly used by billions of people all over the world [37, 34, 65] - these two are similar, however, SMS often requires a payment while IM, usually, allows users to send messages for free as long as they have an Internet connection. Another difference is that SMS text messages, normally, go through the cellular provider, which can be requested through a warrant, allowing for unauthorized access to personal information.

Certain IM services permit users to participate in conversations with others in real-time, offering security through end-to-end encryption. However, some IM services, includ-

ing the most popular ones, such as WhatsApp, WeChat, and Facebook Messenger, require users to connect to a central server, maintained by the company, for authentication, record keeping and similar. For instance, Facebook Messenger is an IM application that requires users to connect to servers run by the company Meta. These social media companies usually have the ability to access all data that goes through their servers which may cause a breach of trust between the users since companies could reveal or use personal information for their own gain. Even if it is the case of an end-to-end encrypted IM service, the encrypted data still gets routed between users through their servers. There are also IM applications that are serverless (Ricochet, RetroShare) however these are not as popular.

1.1 Problem Statement and Motivation

Most IM applications require the user to access a central server run by a company. Any data that goes through the server is accessible to the company. Cases of misuse of personal information by social network companies, such as the Facebook-Cambridge Analytica data scandal [12], have also brought to light some unlawful uses of the personal information gathered and processed by these platforms. As a way to try to maintain the users' trust, many of these applications adopt end-to-end encryption [17] to enable confidential exchanges. Even with end-to-end encryption, it may be possible to identify the communicating parties through web traffic analysis [25]. The knowledge that someone is talking to another person is already a breach of privacy and may lead to repercussions in certain situations, such as whistleblowers disclosing information to journalists. Serverless applications also have their issues: Ricochet Refresh [47], a peer-to-peer solution, for example, uses The Onion Router (TOR) to achieve anonymous, untraceable and decentralized communications at the risk of being censored in certain countries which do not allow TOR [67, 70]. RetroShare, a friend-to-friend structured solution, makes it basically impossible for the Internet Service Provider (ISP) or a third party to know what is happening, but the same is not true for users on the inside. Any users that are part of the circle of "trusted friends" will be able to see what is happening and even the other user's IPs.

Both server-centric and serverless solutions do not offer true anonymity and covert

communications. This thesis will show that this anonymity can be achieved with viability through a combination of features of both architectures and the usage of steganography.

1.2 Objectives

This project has the objective of presenting and implementing a covert and private messaging application with a focus on steganography as the main cyber protection factor. In order to achieve this, the design has certain requirements. Messages should not leave a noticeable track and the application should encourage anonymity. All messages must also be encrypted as a safety precaution. Since the main weakness of this solution is the physical system of the user being accessed by a third party, an authentication process must be done to confirm that only the owner can see messages. Finally, it is imperative that all messages are temporary and do not remain on the proxy servers for longer than necessary.

1.3 Contributions

The contribution of this project is a novel system that focuses on ensuring the user's anonymity. This contribution resulted in the following article publication:

1. Sousa, P.; Pinto, A.; Pinto, P. Exploiting Online Services to Enable Anonymous and Confidential Messaging. Journal of Cybersecurity and Privacy, 2022, 2, 700-713. `https://doi.org/10.3390/jcp2030035`

1.4 Organization

This paper is structured as follows. Chapter 2 presents an overview of cryptography, steganography and other concepts which are required in the context of the proposed solution. Chapter 3 presents related work. Chapter 4 provides an assessment of online services to be used by the proposed messaging service. Chapter 5 presents the requirements, design, and specification of the proposed solution. Chapter 6 presents the details of implementing the proposed messaging service. Chapter 7 provides results and discussion

on the security of the concept. Chapter 8 presents the final conclusions and results and possible points of interest towards future work.

Chapter 2

Background

In order to fully grasp the project developed in this thesis, a few concepts must be understood beforehand. Both cryptography and steganography are used to achieve the main objectives of confidentiality and anonymity.

2.1 Cryptography

Cryptography refers to communication techniques focusing on ensuring that information is confidential, or in other words, only the sender and the chosen receiver will be able to know the contents of the message.

Although nowadays cryptography is usually connected to digital communications, it does not only refer to that. Even before computers, cryptography was used in various ways. For instance, one of the more known pre-modern age methods of encryption is the Ceaser Cipher [39], where each letter of the message (plaintext) is shifted a certain amount of times in the alphabet - so, if someone wrote "HELLO" with a cypher of 1, the result would become the cyphertext "IFMMP", because the letter I is right after H, F is after E and so on. For an outsider the cyphertext obtained is incomprehensible, it does not mean anything and only the sender and the receiver would understand the meaning behind it.

An example of why cryptography was and still is important can be seen during World War 1 and 2 [21, 9], where it would be common to encrypt messages to ensure that enemy sentries would not be able to figure out what was being communicated, which could, for example, be meeting locations or times which would create a perfect setup for ambushes

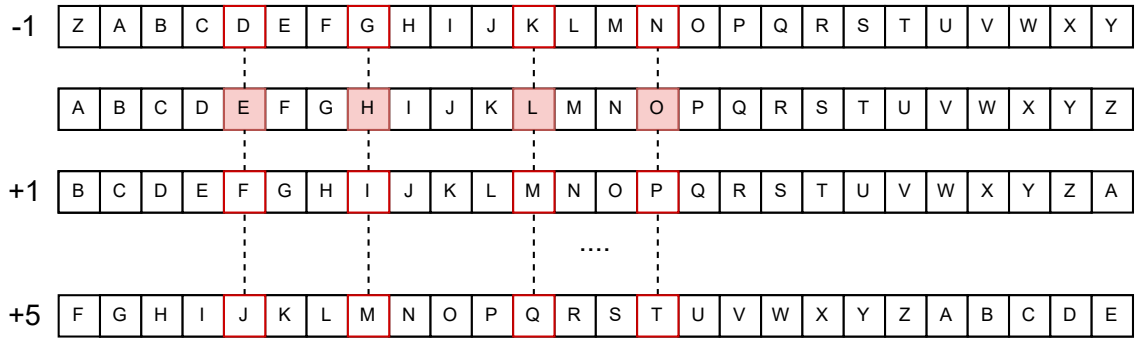


Figure 2.1: Caeser Cipher

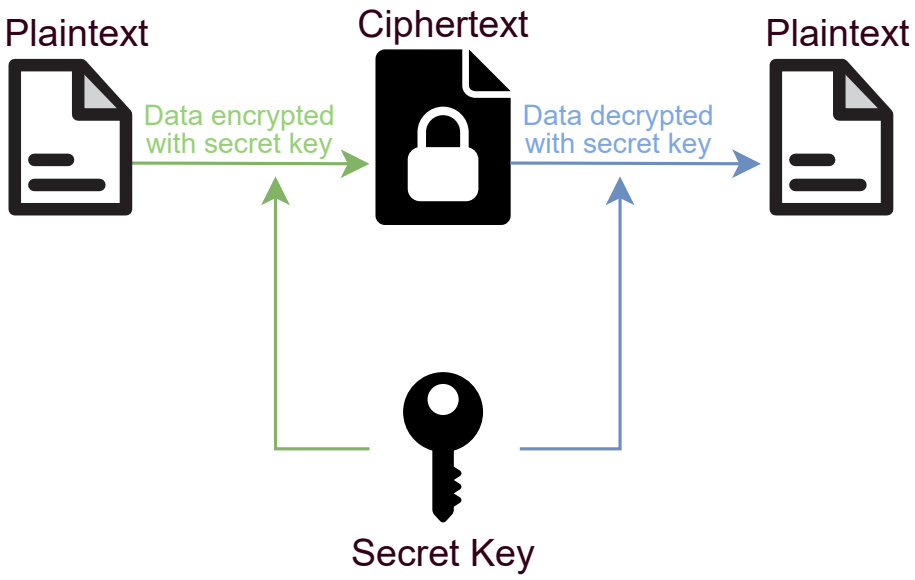


Figure 2.2: Data encryption with symmetric key cryptography.

if known by the opposition.

With the advancement of technology and computers, cryptography has also seen a big jump in complexity. Modern cryptography algorithms rely on mathematical and computational processing, making them unfeasible to break into, requiring far too many resources to do so, be it computational resources or even time.

All of the cyphers used up until and during the pre-computer times were symmetric encryption techniques. This entails the usage of the same secret key that both the sender and receiver will use. The sender will use it to encrypt the plaintext, while the receiver will use it to decrypt the ciphertext. Image 2.2 shows a basic presentation of how symmetric key encryption works.

Even today, there exist many different symmetric key algorithms that still see use and are considered secure. Examples are the AES (Advanced Encryption Standard) [16], the Twofish [51], or the IDEA (International Data Encryption Algorithm) [11]. These algorithms can either be block cyphers or stream cyphers: The former is focused on encrypting blocks of data of a certain fixed size, for example, 256 bits (hence the name block cyphers), and normally outputs encrypted blocks of the same size as the input blocks. Meanwhile, the latter is focused on encrypting data with unpredictable sizes that can also come as streams, such as videos.

In order to ensure an environment as safe as possible where only symmetric key algorithms are used, a combination of algorithms is required: an algorithm that generates a key from passwords/passphrases, such as Argon2 [6, 7], which will be used as the secret key to encrypt/decrypt the content, removing the weakness of using the password/passphrase itself as the key; an unpredictable Initialization Vector (IV) should also be generated, through a secure random number generator for each message, that is used in the next step; a symmetric key algorithm should be chosen to encrypt data with the secret key and the generated IV, ensuring that the security infrastructure is less likely to be broken through analysis and guaranteeing confidentiality; finally, a message authentication algorithm, such as Hash-Based Message Authentication Code (HMAC) [36, 5, 64, 4], should also be used, to ensure the authenticity and integrity of the message - a hash is created from the plaintext and secret key and is sent alongside the cyphertext and the IV. When decrypting the cyphertext, a hash is created again from the decrypted plaintext and secret key and then compared with the embedded hash - if they are different, then that means the message has been tampered with in some way [19].

Alternatively to symmetric key algorithms, asymmetric key algorithms (also known as public key algorithms) make use of multiple keys to encrypt and decrypt data. This choice can also be used as a way to digitally sign data. Some of the more known public key algorithms are RSA (Rivest-Shamir-Adleman), ECC (Elliptic-curve Cryptosystems) and ElGamal [50].

Image 2.3 presents the basic premise behind public key cryptography, where a user wants to send an encrypted message to another user. Each user has a pair of keys, one public and one private, generated with a basis on mathematical operations, such as

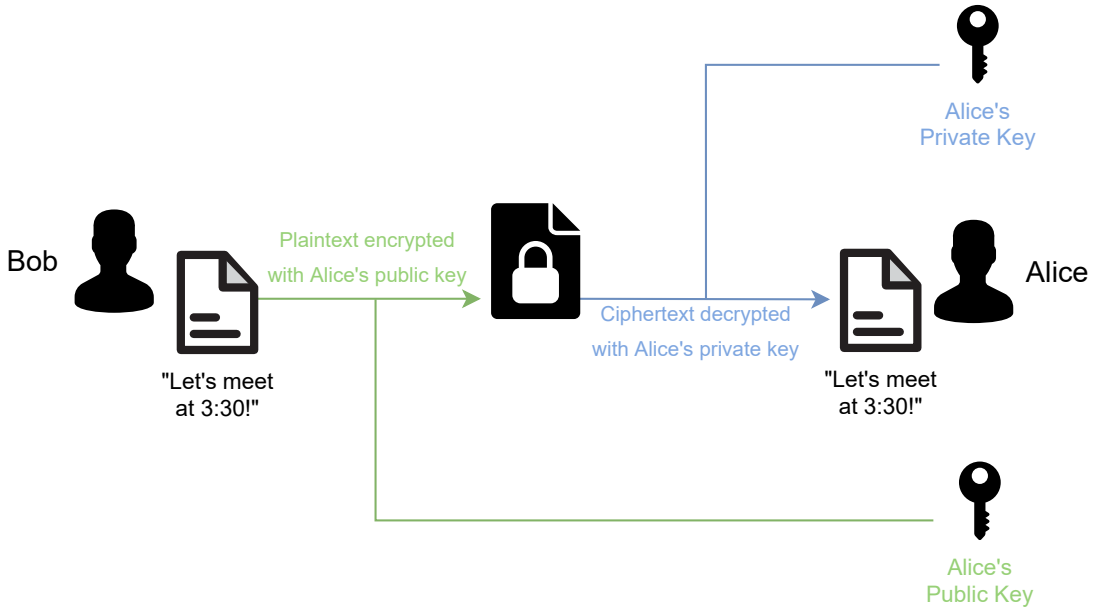


Figure 2.3: Data encryption with public key cryptography.

factorization in prime numbers, that are simple to perform with all necessary information, but hard to perform without all information. The public key, as the name states, is a key that can, and should, be shared in cyberspace, while the private key is a key that only the owner should know. Public keys can be used to encrypt data and only the private key can decrypt messages encrypted with the public key. So, the sender grabs the public key of the receiver, encrypts the message and sends it. The receiver gets the encrypted data and uses its private key to decrypt it.

This is enough to guarantee confidentiality. But how can the receiver be sure that the person that sent the message is who they are claiming to be? The keys have some extra functions besides encryption/decryption: the private key can be used to sign digital data while the public key can be used to verify a digital signature. As such, the sender signs the message with their own private key and then appends their signature to the message. This whole package is then encrypted with the receiver's public key. When the receiver decrypts it with their private key, they can grab the sender's public key and verify that, at the very least, the person that sent the message has access to the private key of the sender (which means it is very likely that it is the person).

When considering symmetric key cryptography and asymmetric key cryptography, each approach has its own advantages and disadvantages. Symmetric has the downside of using

a single key that is shared among all the users, which can eventually lead to it being leaked and creating a system that offers no protection whatsoever. Asymmetric, does not have this issue, since all users have different pairs of keys and you would require both the public and private keys to be able to encrypt and decrypt data.

On the plus side, the symmetric key approach is better on all fronts when considering sizes: key size is much lower than asymmetric (256 bit vs 2048 bit or higher), encryption is faster and it is also better than asymmetric at transmitting more data.

A common practice is to use both these approaches together since they complement each other well: one of symmetric key cryptography's biggest issues is the act of securely handing the secret key to another person. Since asymmetric key cryptography takes longer to be processed and is more focused on transmitting smaller-sized data, it can be used to give the encrypted symmetric secret key to the receiver in a safe way.

2.2 Steganography

While cryptography can be used to ensure the confidentiality of the message, with some web-traffic analysis it may be possible to figure out who are the users communicating with. In general, even if the encrypted messages do not give any information about themselves, it still arouses suspicion and could even be illegal. Steganography circumvents that issue by focusing on being covert. Steganography [40, 24, 74, 29] is the science that studies techniques of hiding data in plain sight - within an ordinary, non-secret file, image, video, audio, or message. A secret message hidden in a selfie which is then posted on a social media network is much less likely to arouse suspicion than encrypted data is. Incidentally, this opens up the possibility of anonymity by itself, because even if the secret message is found out, as long as it is not related to the user and does not have any metadata that leads back to the user, their software, or their hardware, the user can just claim that they found the image online and just wanted to share it.

Images, videos, and audio are good choices for steganography due to their size. The amount of data that can be hidden through steganography is dependent on the size of the cover file, the file which will "host" the secret data. The bigger the file, the more data that can be hidden and the process is also easier and more secure. A secret message can

be hidden in an audio file by converting an image or text into sound files and then hiding it into an unused audio channel in music, for example.

To use steganography with an image file as a cover, the Least-Significant-Bit (LSB) technique is, nowadays, the more known and used [33, 35]. To understand this technique, it is required to understand how digital images work. When we are presented with a digital image, what we see is a combination of pixels, each representing a different colour. Each pixel will have information on the colour it should represent using three different channels (red, green, and blue) and eight bits of information for each. Every channel can go from 0, which represents the absence of colour, to 255 which represents the maximum of that colour. For instance, if a pixel has the value 0 for all three channels, the result colour would be black. With the value 255 for all channels, the colour it would display would be white. For 255 on just the red channel and 0 on the green and blue channels, the colour displayed would be red.

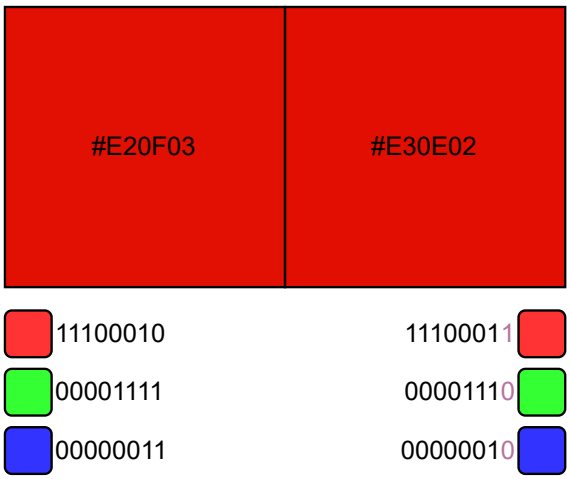


Figure 2.4: LSB and how it affects digital color.

So for each pixel, we have 8 bits representing each colour, however, if we were to change the last bit of each channel, switching any ”0”s to ”1”s and vice versa, even though the colour would change, the difference is not detected with the naked eye. Figure 2.4 depicts this concept.

With that in mind, the amount of bits needed to hide a single letter, when converted to binary, is 8 bits. If a person were to hide a letter through LSB, they would need 8 bits. So, in order to hide a single letter, three pixels would be altered slightly - the last

bit of each 8-bit value representing the colour channel for the first and second pixel and the last bit of the 8-bit value of the red and green colour channel are altered (if needed). Figure 2.5 depicts a situation where a user wants to hide the letter "H" (ASCII value 72) that corresponds to the binary value of 01001000.

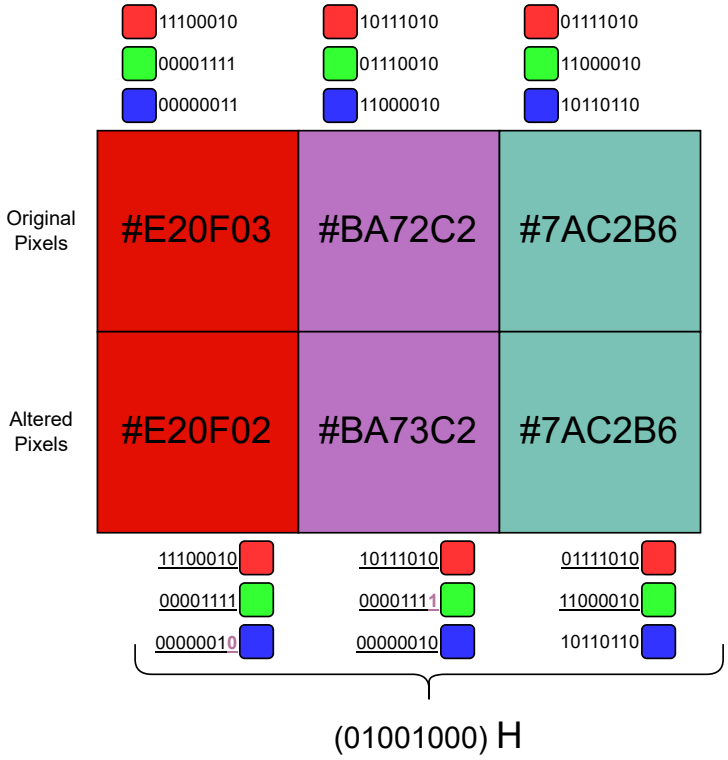


Figure 2.5: Hiding "H" in images.

There is another method of hiding data in images, based on LSB known as YUV [58] which derives its name from how it hides images - while LSB focuses on hiding data on images in RGB colour model, YUV hides data on images in YUV colour model.

LSB can also be used in videos [48]. The process is roughly the same considering that videos are just a combination of frames (still images) being shown one after the other.

Chapter 3

Related Work

This thesis studies and focuses on solutions that enable and provide anonymity and confidentiality. The presented proposal, which will be described in detail in Chapters 5 and 6, makes use of cryptography and steganography as a way to achieve a decentralized and "serverless" anonymous and confidential experience.

3.1 Current Applications

Multiple applications identified as related work were analysed in order to understand if these provide anonymity and confidentiality, and if so, how they do it. Ultimately, this analysis helped to understand if the proposed solution is a novel one. If the proposed solution provides no innovation, because another application already achieves the required objectives in a similar way, then there is no point in developing it. As such, a comparison of various messaging applications, marketed as being focused around security was studied to better gauge in what ways the developed solution would be advantageous over other applications.

Nowadays, when being suggested [52] a secure messaging application, the likely recommendation will fall under one of the following: Signal [55], Threema [63], Wire [68], or Session [53].

Developed by the Signal Foundation, Signal is a cross-platform instant messaging application that allows users to communicate, through text messages, audio messages, and video. It was released on the 29th of July 2014. Signal makes use of XEdDSA and

VXEdDSA [62], as a way to generate an output of differing hashes which can be used for different functions from a single input which will be used mostly for signature purposes, coupled with X3DH [61], which is the public key cryptography protocol adopted, in order to achieve a secure status. Their "Double Rachet" (new keys derived for every message from a shared secret key) [59] and "Sesame" (message encryption in asynchronous settings) [60] algorithms also ensure the maintenance of that security even in the face of a compromise. Its' code is also open source, which is a great boost for ensuring proper security. Even though Signal does not show any major security flaw it does not achieve total anonymity - each user account is bound to a mobile phone number, required on registration. This information is sent to Signal's server, which in turn can link a user's cyberspace presence to their real person. As a result, anonymity is not achieved.

Wire [69], another cross-platform instant messaging application, that allows users to communicate through text messages, audio messages, and video, developed by the software company Wire Swiss, was first released on the 3rd of December 2014. Adopting a security-by-design approach, Wire encrypts its' messages with the Proteus protocol, based on the Signal protocol developed by Open Whisper Systems and introduced in the TextSecure application (which would later become Signal), which makes use of 3-DH handshakes, Curve25519 ECC, AES-256 and HMAC-SHA256, and the aforementioned Double Rachet algorithm. Like Signal, its' code is available for any and all to review. That said, although it achieves confidentiality and authenticity, much like Signal, Wire faces an anonymity issue in the user registration process, where it requires either an e-mail address and/or mobile phone number to start using the service. It is impossible to use Wire without some link or connection to the user's real person.

Developed by Threema GmbH, Threema [14], first released in December 2012, is an end-to-end encrypted messaging service, where the messages may have text, audio, and video format. It uses public-key cryptography, firstly generating a key pair and afterwards sending the public key to the Threema servers. All messages sent have to go through the Threema servers but are claimed to be deleted as soon as the message reaches the recipient. Using the NaCl Network and Cryptography Library for both end-to-end encryption and transport-level security, Threema uses Curve25519 as key derivation, XSalsa20 as symmetric encryption and Poly1305-AES for authentication and integrity protection. The

client side is open source however the application's API and server code are not. Threema is different from Signal and Wire in the sense that it does in fact offer anonymous communication - it gives the user the choice to input a mobile phone number for recovery purposes, but it is purely optional, not required at all for using the service. That said, this application has a big drawback compared to the other two which is that it is a paid application.

Session [54], another messaging service providing text messages, audio messages, and video communications, makes use of blockchain technology to provide its own onion-routing service. When a user sends a message, it is carried through three different nodes on the network before passing through a listening node and, once again, hopping through three different nodes. Since each node only has information regarding its adjacent nodes (i.e., the first node to receive the message only knows the IP of the sender and the second node - not the recipient's IP), privacy is ensured. The application relies, for the most part, on local storage, with the exception being the temporary storage of messages in multiple nodes, designated a swarm, which get deleted after the messages Time To Live (TTL) is surpassed, which can happen if the recipient is offline. However, this network requires nodes. In order for a node to be authorized and accepted to the system, a stake must be made, requiring the owner of the node to lock a certain monetary value to the node. Afterwards, the person responsible receives a reward for their node's usage. On the one hand, this makes it quite costly for anyone attempting to attack the network; however, the reward system may also create a conflict of interest between maintaining privacy vs. abusing this system.

An overview of what each of these applications achieves in terms of security level is presented in Table 3.1.

3.2 Research Works

Research works in the area of secure messaging may count on specific protocols. Messaging Layer Security (MLS) protocol [3] is a protocol resulting from the combined effort of multiple researchers, from Cisco, Mozilla, Google, Facebook, Twitter, the University of Oxford, MIT and INRIA with the main objective of increasing efficiency and security of

Table 3.1: Messaging applications and proposed solution security comparison.

	Signal	**Threema**	**Wire**	**Session**	**Proposed Solution**
Provides anonymity?	**No***	**No***	**Yes**	**Yes**	**Yes**
Data gathered	Contact Info	Contact Info, Diagnostics	Contact Info, Diagnostics, Usage Data	None	None
Supports self-destructing messages?	Yes	No	Yes	Yes	Yes
Can be used without a mobile phone?	No	No	Yes	Yes	Yes
Is a free application?	Yes	No	Yes	Yes	Yes
Is Open source?	Yes	No	Yes	Yes	Yes
Is resistant to conflicts of interest?	Yes	No	Yes	No	Yes
Requires cryptocurrency?	No	No	No	Yes	No

* A phone number is required to register a new account and data are processed by parent or third-party companies.

end-to-end encrypted messaging in large groups. In a MLS secured group chat, when a new user wants to join a group, they first send a Key Package to the creator of the group, which then provides the public key and authentication information. Once that is done, the creator of the group also broadcasts two messages to the group, one to announce that a new user has joined and another to hash the old key to generate a new one. When removing a user, a similar process occurs - two messages are broadcasted to warn the users that remain in the group chat that a user has left and another to hash the old key to generate a new one. It uses a combination of ChaCha20Poly1305 or AES-GCM (depending on the system) and X25519/X448 curves or FIPS 140-2 compliant curves for Diffie-Hellman key negotiations.

MLS protocol is not based on steganography, however, steganography can be used as a way to ensure more secure communications is not a novel idea and has already had various research works [57, 10, 27, 38].

The research done by the authors of [57], argued that steganography could be used for message exchange between users in a way that only the sender and receiver were able to decrypt the message and were the only ones aware that messages were being exchanged. It used simple steganographic algorithms based on two different methods (LSB and YUV), plus an extra method that makes use of KLT along with LSB, running on three different systems: an ARM7-based microcontroller, a multi-core architecture digital signal processor and a personal computer. This work reached the conclusion that between the different systems, the personal computer had the worst results, requiring the most time to decode

the message while the multi-core architecture digital signal processor was the best choice. It was also found that the LSB method had quicker results and the KLT method had slower results. The authors also comment, as a final conclusion, that execution times are highly influenced by the size of the host image.

In [10], the authors start by presenting an assessment of various online services and how they process images, at the time. From the information gathered from that assessment, the authors propose and analyze two different steganography approaches to hide data: one through the name of the image based on naming conventions (which depends on the type of digital camera used); and another by using tags - by using images in a pre-emptively decided album and a combination of previously decided users it is possible to hide information. For instance, if an album has 2 pictures and 8 people, it would be possible to hide 2 bytes of information as a whole by seeing which users were tagged in each picture. If a user was tagged, then it would represent 1, otherwise, it would represent 0. These proposals allow for message exchanges without the servers being aware, however, their proposal did not encrypt hidden messages and could present size constraints difficult to overcome.

Authors of [27] studied the concept of using Facebook as a way to exchange hidden messages through images. One of their findings was that image compression in Facebook, due to storage space and bandwidth constraints, generally disrupted steganography. With that information in mind, the authors decided on what image format to use for standard test images. The JPEG format was chosen, due to the fact that Facebook converts uploaded images to JPEGs, and compression towards this format is different from other formats, such as PNG. From there the resolution of the images was decided: images of 2048xYYYY and 960xZZZ. According to the authors, these resolutions were preferred because, once again, these were the resolutions preferred by the social network. With the standard images prepared for testing, a variety of steganography programs and algorithms were chosen to hide messages. As a whole, the chosen programs and algorithms had a lot of problems hiding information on images already processed by Facebook with the best result being the "JP Hide & Seek" program that had a 50% success rate but even then, out of that 50%, plaintext was only recoverable out of 5% of the test images and it still had some characters altered.

In [38], the authors researched ways to bypass steganography disruption due to modification of the submitted images. The way this is achieved is by first passing the original image through JPEG compression and then hiding data through steganography and thus creating a stego-image (image with data embedded through steganography). This stego-image is studied and from that, a coefficient adjustment scheme is used to create an intermediate image. This image when going through JPEG compression will result in the stego-image and as such, compression should not impact the extraction of secret data.

From the analysis of these works, it can be concluded that the topic, despite not being new, is still under active research. Further, none of these solutions allows user anonymity, as all require user authentication before using the service. To the best of our knowledge, there is no solution that allows anonymous and confidential messaging between users without user authentication or dedicated servers. The design of a novel proposal should not compromise performance when using different image sizes, all messages processed and hidden in images should be encrypted beforehand, and the image compression imposed by using services should be circumvented to maintain the original images.

Chapter 4

Online Services Assessment

Although one of the research works that was studied went into detail in regards to how different online services processed images, it was done in late 2011, which means that these same online services will likely work differently nowadays. Since steganography, image sharing and messages being hidden in said messages are a cornerstone of the proposed solution, it is very important that there exists an updated understanding of how these online services process images and confirm whether there is viability in the concept in the present day. For that purpose, an assessment of current steganography applications and online content-sharing services was carried out.

In regards to steganography applications, the following four applications were chosen, due to their availability as free versions and also since they have open source code: OpenPuff [42], OpenStego [66], StegHide [26], and StegoShare [20]. All applications use, from what was found, the LSB steganography method to hide data in images, however, some of the applications offer more functions. For example, OpenPuff has support for multiple image formats, even lesser used ones, and also provides some extra layers of protection, including encryption of secret data, scrambling to make it harder for an attacker to know where data begins or ends, whitening to mix the scrambled data with noise, and finally, encoding the whitened data through a non-linear function. OpenStego can be used as a watermark to ensure that no user can claim credit for an image of a second user. StegHide even gives users the option to hide data in audio formats.

As mentioned before, current online services are known to alter images/photos, in order to minimize traffic, bandwidth, and storage space needed, after upload. This could

mean a change in compression, resolution, and metadata which, due to how LSB works, will likely impact steganography. The chosen online services are divided into two groups: most popular online services and online services focused on image hosting. In the first group, Facebook, Twitter, and LinkedIn were tested, meanwhile, for the second group, Imgur, Flickr, and ImgBox were picked.

This assessment involved two different tests: the first one designed to do a general check of how images are changed by these online services and the second one designed with three different goals in mind which are:

1. Determining whether compression algorithms impacted steganography;
2. See how the systems react to images with different characteristics (different resolutions, monochrome);
3. Understand how much data could be hidden

For the first test, a random image with a resolution of 3840×2160 pixels was processed by the steganography programs chosen to hide a simple message ("mciber2022") which generated a stego-image, the carrier image with the secret message hidden in it. Afterwards, this stego-image was uploaded to the selected online services. Once uploaded, the image was downloaded from the service's servers and their characteristics were compared to the original stego-image in order to determine the changes introduced. Finally, the downloaded stego-image was processed by the steganography programs to verify if any secret message can be retrieved.

The results of this first test, conducted in September 2021, are presented in Table 4.1. It was found that Facebook and Instagram both lowered the resolution of the images down from 3840×2160 to 2048×1152. All the popular online services presented a high compression rate. The online services focused on image hosting, Imgur and ImgBox, had more favourable results but were still not viable, as the image indicated that a low-compression algorithm was used, and as a result, secret data was not retrievable. Flickr showed no compression whatsoever.

Admittedly, the results of the first test were not too positive. However, this can be attributed to the chosen image which has a relatively high resolution, which usually means

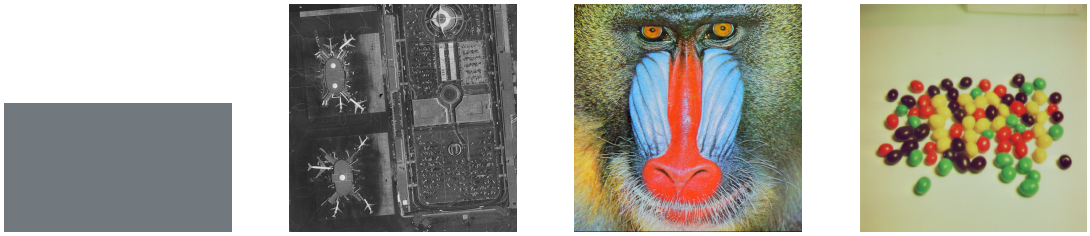
Table 4.1: Results of the first test.

		Input	Output Image					
		Stego Image	**Facebook**	**Linkedin**	**Twitter**	**Imgur**	**Flickr**	**ImgBox**
OpenPuff	Size	6.72 MB	312 KB	186 KB	546 KB	467 KB	6.72 MB	6.54 MB
	Res.	3840 × 2160	2048 × 1152	2048 × 1152	3840 × 2160	3840 × 2160	3840 × 2160	3840 × 2160
	Format	PNG	JPG	JPG	JPG	JPG	PNG	PNG
OpenStego	Size	7.8 MB	312 KB	186 KB	545 KB	467 KB	7.8 MB	4.49 MB
	Res.	3840 × 2160	2048 × 1152	2048 × 1152	3840 × 2160	3840 × 2160	3840 × 2160	3840 × 2160
	Format	PNG	JPG	JPG	JPG	JPG	PNG	PNG
StegHide	Size	791 KB	277 KB	187 KB	770 KB	749 KB	791 KB	877 KB
	Res.	3840 × 2160	1920 × 1080	2048 × 1152	3840 × 2160	3840 × 2160	3840 × 2160	3840 × 2160
	Format	JPG	JPG	JPG	JPG	JPG	JPG	JPG
StegoShare	Size	19.2 MB	339 KB	190 KB	647 KB	467 KB	19.2 MB	N/A
	Res.	3840 × 2160	2048 × 1152	2048 × 1152	3840 × 2160	3840 × 2160	3840 × 2160	N/A
	Format	PNG	JPG	JPG	JPG	JPG	PNG	N/A

that the file size is also higher. Higher sizes mean more storage space, higher traffic, and bandwidth so it is more likely to suffer higher compression. With that in mind, the second test was planned out.

The second test, conducted during the months of September and October of 2021, consisted of two specific text messages of different lengths encrypted with AES-256 and afterwards hidden by one of the steganography programs, also used in the previous test, in four base test images of different colours and sizes. The messages hidden are "The quick brown fox jumps over the lazy dog" and a random 256-character message. As for the chosen images, presented in Figure 4.1 to use as test images, one is a solid grey colour image with a resolution of 1920×1080 pixels and the remaining three are images from Volume 3 of the database of standard test images of the University of Southern California. In particular, images "4.1.08—Jellybeans" (with a resolution of 256×256 pixels), "4.2.03—Baboon" (with a resolution of 512×512 pixels), and "5.3.02—Airport" (with a resolution of 1024×1024 pixels) were selected due to their different resolutions and the need to have both coloured and grayscale images. Due to limitations with each steganography program, all images were converted to both PNG and JPEG. A total of 192 images were uploaded, processed, and analyzed. As a way to verify if the images were altered after being uploaded to online services, secure hashes (MD5 and SHA) were obtained and compared.

The results obtained from this test were much more favourable and presented the viability of the concept of using steganography in online services. Although Facebook, LinkedIn, and Twitter all had a success rate of 0% in retrieving the secret message, the other services ranged from success rates of 87.5% and 100%. Table 4.2 presents the success rate for each service and program.



(a) (b) (c) (d)

Figure 4.1: Adopted base images: (**a**) solid colour; (**b**) airport; (**c**) baboon; (**d**) jellybeans.

Table 4.2: Percentage of successfully retrieved messages.

	Facebook	**LinkedIn**	**Twitter**	**Imgur**	**Flickr**	**ImgBox**
OpenPuff	0%	0%	0%	100%	100%	87.5%
OpenStego	0%	0%	0%	100%	100%	100%
StegHide	0%	0%	0%	100%	100%	0%
StegoShare	0%	0%	0%	100%	100%	100%

In more detail, Facebook, Twitter, and LinkedIn compression algorithms do not appear to be the same, since the hash values of all images were different. Further, all images, regardless of resolution, file size or format, were converted into JPG format, and the resulting file size was smaller. Images having a resolution of 3840×2160 pixels were reduced to 2048×1152 pixels on Facebook and LinkedIn. The metadata of the image was also changed. These social network platforms use image compression, directly hampering steganography. Similar behaviour was assumed to exist in all social network platforms.

Imgur had a 100% success rate when using images with a resolution of 1920×1080 pixels or less. Images with higher resolutions or in PNG format were converted to JPEG and compressed in such a way that their size became too small to retain the secret message. However, if the original image was already in JPEG format, the image file changes did not disrupt the steganography. Further, the secure hash results of the retrieved images were different from those of the uploaded pictures. After additional analysis, we concluded that Imgur stripped metadata from all images, resulting in different hash values for the same images. These conclusions were drawn by extracting the core data of the images and then comparing the secure hash values of these.

Flickr had a 100% success rate due to the fact that Flickr allows the user to download the original images without any changes. Even image metadata and resolutions are preserved. The secure hash values of the downloaded images confirmed that no modifications

were made to the uploaded images.

ImgBox was able to retrieve information for all steganography programs except StegHide. In our testing, StegHide processed only JPEG images, which, in this case, underwent enough changes to disrupt steganography-based message retrieval.

Chapter 5

Proposed Solution

The main objective of the proposed solution is confidential, anonymous and covert end-to-end communications between users. For the solution to be considered confidential, all exchanged messages must be encrypted end-to-end, which means that messages are encrypted at the sender and only decrypted at the destination. In order for it to also be considered anonymous, only the participants should know the identity of other participants. Finally, covert means that all communications and exchange of data must be hidden. The most suitable way of achieving all of these requirements is by avoiding dedicated servers and making use of existing online services which do not lock their service behind user authentication.

The proposed solution makes use of steganography as a way to keep data hidden, which requires that images, with the message embedded into them, are exchanged between users. Because of this, an image-sharing online service is required. However this brings up an issue: how can the users exchange the uploaded image without sending it directly to the user, possibly compromising the anonymity of the solution? Another online service, in specific, a collaborative text editor, was chosen as a middle-way to exchange the image Uniform Resource Locator (URL), which the recipient can recover and use to download the image.

Therefore, with the tests performed in Chapter 4, the most fitting choice for image-sharing was considered to be Imgur, mainly because it offers an API with anonymous upload features. Moreover, for the text-sharing online service, responsible for exchanging the image URL, Dontpad was deemed a great choice, once again, due to having an easy-

to-use and fast API to write and retrieve data to a document of the user's choice.

Figure 5.1 presents a reference scenario. The Sender wants to send a message to the Receiver so they start by using the application which encrypts and hides data in an image. Afterwards, the image is uploaded to the image-sharing online service (Imgur), which generates and returns an URL. This URL is written to a document in the text-sharing online service (Dontpad). From there, the receiver will access the document in the text-sharing online service and retrieve the URL and download the image. Once downloaded, the image is processed and the hidden data is revealed. This hidden data is decrypted and the plaintext is recovered.

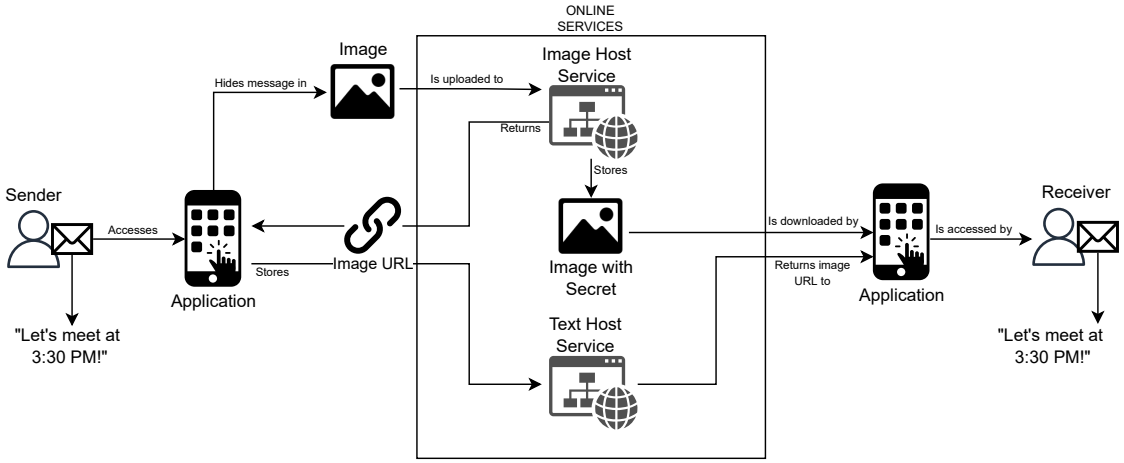


Figure 5.1: Reference scenario.

5.1 Identified requirements

The solution requires that certain requirements are followed. It needs to be private, ensuring that all data is encrypted and that only communicating users should have access to it. It should also be easily scalable - making it so a new user can start participating in the system without downtime. The following requirements are also what the application should achieve:

(R.1) **Covert communications between users** - Any and all data exchange should allow for the users to remain anonymous and not leave a noticeable track.

(R.2) **Encrypted message exchange** - Any and all data should be encrypted to ensure confidentiality.

(R.3) **User authentication** - The application should verify that the user accessing it and using the smartphone is its' owner, as such, anytime the application is rebooted, it must ask for a password.

(R.4) **Cryptographic key refresh** - Cryptographic keys should be refreshed at will by the users themselves if they wish to do so.

(R.5) **Temporary message exchange** - Messages should not remain in the servers (the server hosting the image carrier as well as the server acting as a proxy to exchange data).

(R.6) **Servers must not know users' identity** - The online servers must operate without requiring user authentication.

(R.7) **Settings - Message history - Turn on/off** - The user can choose to keep a local history of their conversations. By default, this option should be turned on.

(R.8) **Local DB - Has the message been read?** - The application should use local storage to discern if a message has been read and also if it has notified the text-sharing service of that fact.

5.2 Specification

Data is stored in four different structures: Local, Contact, Chatroom, and Message. Local is a structure focused on storing information about the user using the system - information such as the user's Universally Unique Identifier (UUID), generated automatically. The contact structure is used to store other users' information - name and UUID. Considering that the solution is serverless, this structure is mainly used for the sake of confirming the origin of messages. In order to send messages, a chatroom must first be created and accessed. The Chatroom structure holds information about the name given by the user to the chatroom, the UUID, and salt, both of which can be generated automatically or inserted manually. The message, the last data structure, is responsible for messages - storing the content, or the message itself, a timestamp of when the message was sent, the UUID of the user that sent the message, and finally the UUID of the chatroom

the message was sent to. All of these data structures also have a unique identifier, used only for database operations. The entity-relationship diagram and the database diagram are depicted in Figures 5.2 and 5.3.

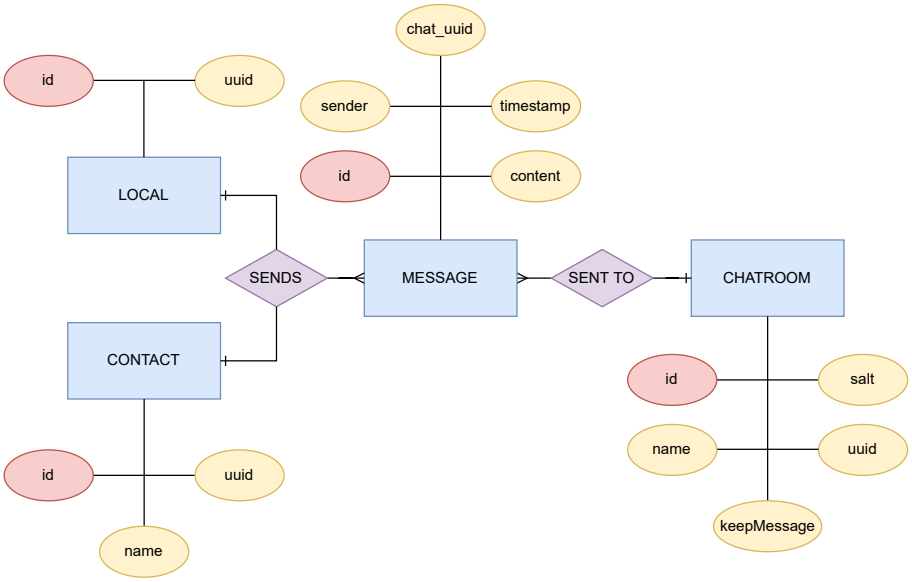


Figure 5.2: Entity Relationship Diagram.

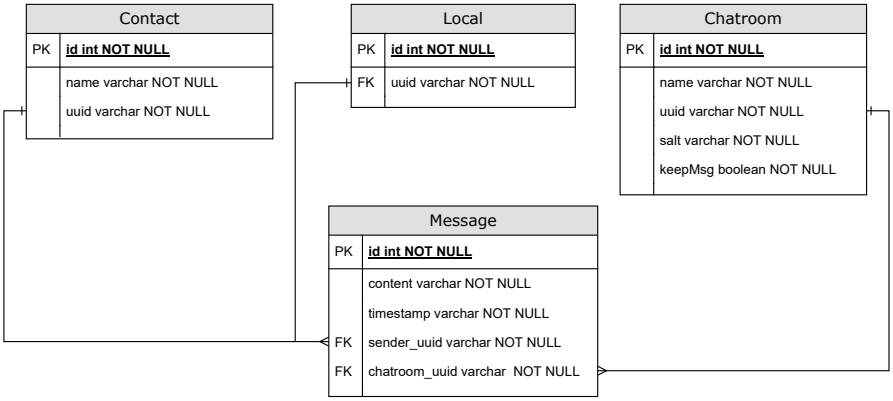


Figure 5.3: Database Diagram.

In regards to entities, there are only three participants - one of them human and the other two online services. The user can register a password to authenticate themselves. This action, in turn, also involves the creation of a database. Since the solution is serverless, there is no way to communicate with a central server to confirm the identity of the user, and as such, the password used to authenticate and log into the application is the password used to decrypt the database, which is encrypted. Once authenticated, the user can add new contacts, view information on contacts already added, or delete them. The

user can also add chatrooms (by generating new credentials or by inserting credentials of a chatroom created by another user). The user can also delete chatrooms and, with it, the messages linked to that chatroom. Once the user accesses a chatroom, they are able to send or receive messages. Both these processes involve communication with the other two entities, an online service to share images and an online service to share text. These are used to store some data temporarily until the recipient receives it and deletes it from the services themselves. The use cases of these entities are shown in Figure 5.4.

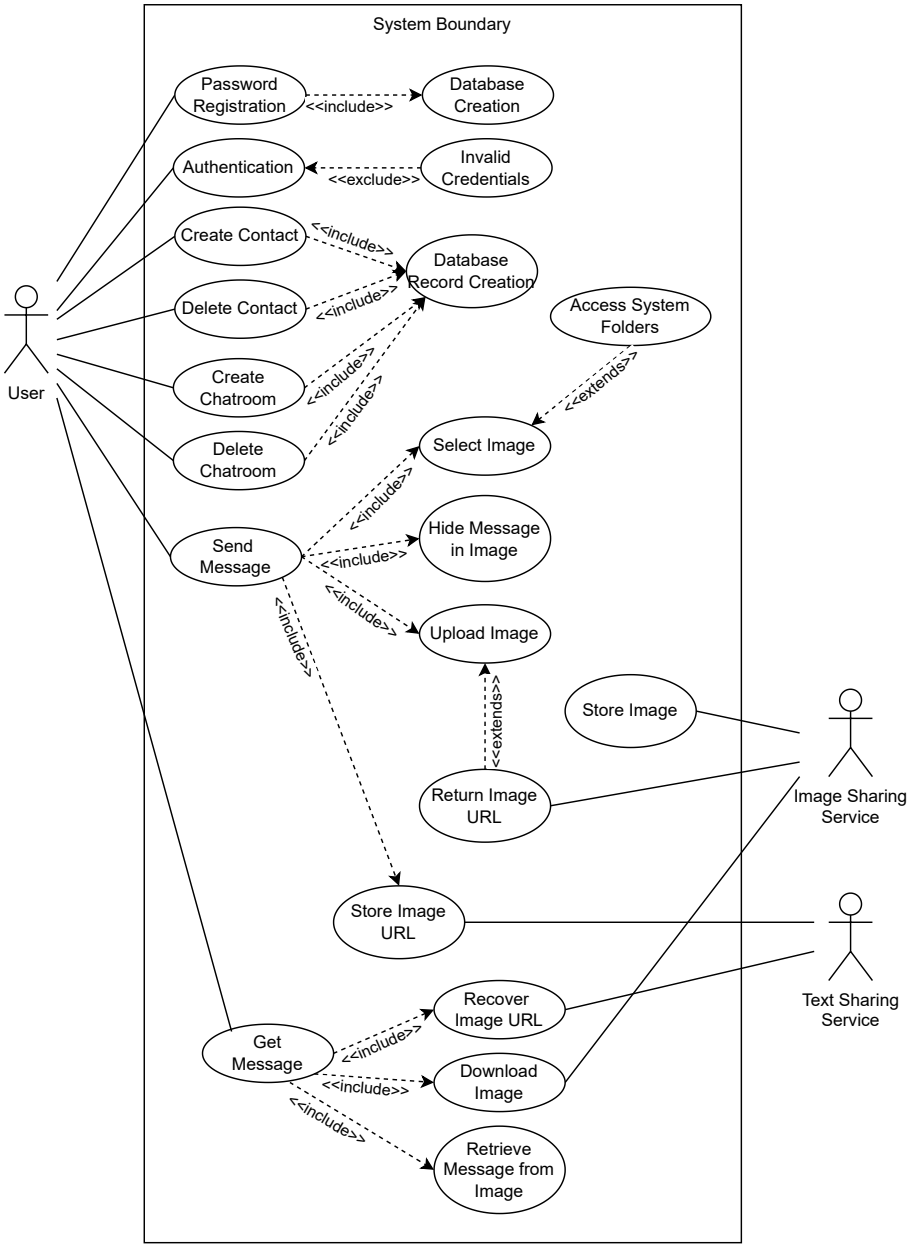
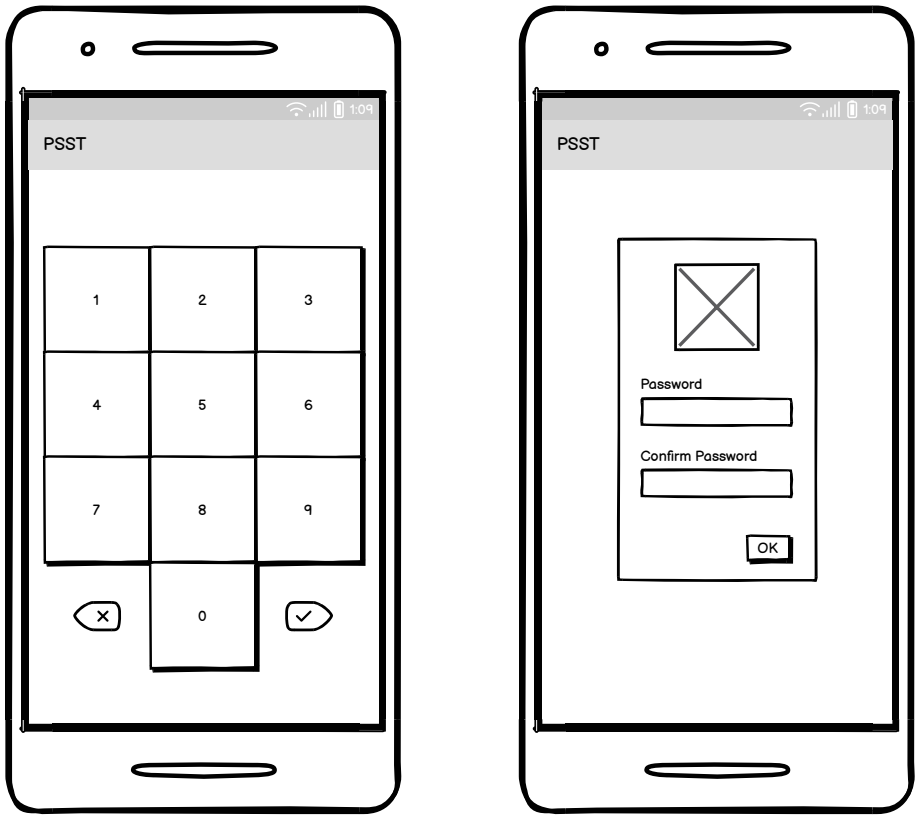


Figure 5.4: Use Case Diagram.

The application will have a user interface that will follow the requirements and use cases presented beforehand. Mockups were designed that attempt to make the user experience the best possible while ensuring that all the functional necessities are achieved.

At first, the initial idea to authenticate users was the usage of a PIN system, wherein the user would input a PIN code, likely 4 digits to then access the application. However, this felt very overwhelming alongside the request for a password to decrypt the database, right afterwards. Not only that, but each chatroom will also require the user to input a passphrase. Overall it felt unnecessary to have a PIN code, so instead, this screen adopted a more common method - inserting a password. The application verifies whether the database already exists and presents different widgets depending on the situation. For example, if the database does not exist yet, an extra text field is enabled that is used as a way to confirm the password inserted by the user. If the user inputs a password and it fails verification, a popup dialogue window appears to warn them. The mockup is shown in Figure 5.5



(a) (b)

Figure 5.5: Mockups: (**a**) PIN input (not used); (**b**) Password input

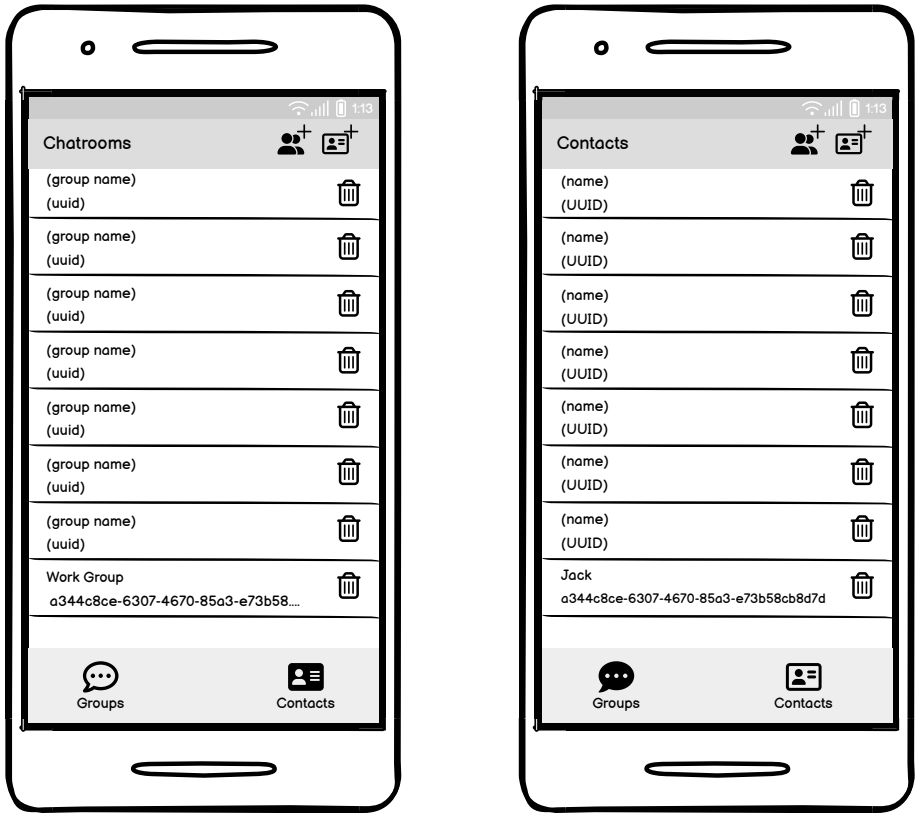
Once the user has inputted the valid password, he gains access to the home menu. Here he can change between a list of chatrooms and a list of contacts through the bottom navigation menu. On this screen, the user can add new chatrooms and contacts through the buttons in the top right. Each of the buttons will make a popup window appear, which will request input from the user. For the chatroom, the popup will, at first, request only the name of the chatroom, which is only used to differentiate chatrooms more easily. The remaining pieces of data are then generated by the application automatically. However, the user can activate a checkbox, which asks if the chatroom already exists, to manually input the UUID and salt of the chatroom. For creating contacts, all input must be inserted manually. Figures 5.6 present these popups.

(a) (b)

Figure 5.6: Mockups: (**a**) Chatroom creation; (**b**) Contact creation

Once users create chatrooms/contacts, they are shown in the list and can be accessed by clicking on it or deleted by clicking the trash-can icon shown to the right. The mockups of these operations are shown in Figure 5.7.

The users can click on chatrooms in the list to open a new screen dedicated to the chat,



(a) (b)

Figure 5.7: Mockups: (**a**) Chatroom list; (**b**) Contact list

to send or receive new messages and view older messages (if the user has not activated the setting to disable the storage of messages for that chatroom). In this screen, the user can choose the image that will be used to embed the message through a button that opens a window to choose the file, a text field to input the message, and finally another button to send the message. When clicking the send button, the application verifies the input of the image and message to ensure that it is valid. If the application sends or receives messages successfully, a new message is added to the scroll view. A mockup of this screen is shown in Figure 5.8.

5.3 Design

To begin an exchange of messages between users, it is required that they first agree on a channel identifier, which in this solution makes use of the UUID format, and a salt to go with it. A passphrase must also be agreed on to use along with the salt and a Key

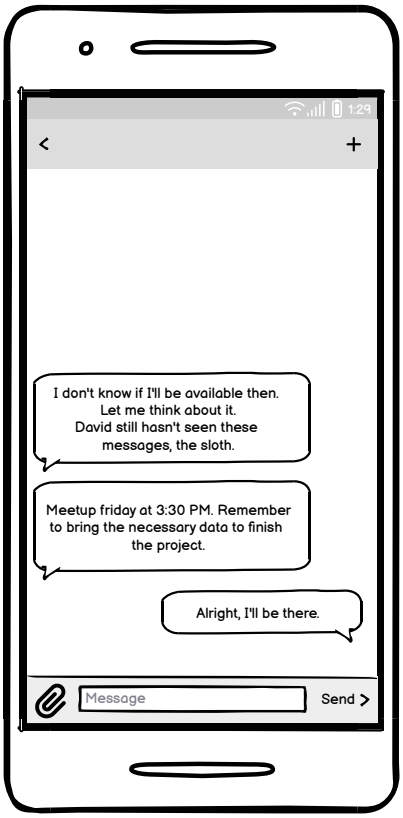


Figure 5.8: Mockup - Chatroom.

Derivation Function (KDF) to create a channel key (C_i) which is shared by both users. The way the application is programmed, whenever the user accesses a chatroom they are requested a passphrase while a salt is stored for it. This allows users to refresh the used key to encrypt/decrypt data, as long as everyone knows what passphrase to use. These three pieces of information are assumed to be exchanged in a secure manner, preferably offline, or in person, for the sake of anonymity. All these values are generated for each conversation using a secure random number generator. The URL of the room, in the text-sharing online service, dedicated to exchanging the URL of the image with hidden data is obtained by concatenating the UUID belonging to the chatroom with the text-sharing online service's URL. If we assume that Channel 1 has the UUID: UUID_1, the chatroom's shared online folder in DontPad will be available at: http://dontpad.com/UUID_1. This URL is then used to exchange an image's URL, with it being deleted as soon as possible when the recipient has read the secret message.

Afterwards, users can exchange messages. A sequence diagram is depicted in Figure 5.9. The sender starts by securely deriving a channel key (C_i) from the password and

the salt (Step 1). In Step 2, an initialization vector IV_i is generated with a secure random generator. Afterwards, in Step 3, the key C_i and the IV_i are used to encrypt the plaintext message using symmetric encryption (AES), generating the ciphertext E_i. In Step 4, the sender generates a HMAC H_i from the ciphertext E_i and IV_i. The ciphertext E_i, HMAC H_i, and a nonce are encrypted with the key C_i and then concatenated with IV_i in Step 5 – these shall be referred to as M_i. In Step 6, the secret data M_i is hidden through steganography, being embedded into an image file F_i, which has had its EXIF data scrubbed clean, resulting in the stego-image SF_i. The sender then, anonymously, uploads SF_i to the image-hosting service (Step 7), obtains its URL (Step 8), and writes the URL to a folder on the text-hosting service (step 9). Next, the receiver can read the shared folder on the text-hosting service, obtain the new URL (Step 10), and proceed to download SF_i (Step 11). Then, he/she extracts the embedded data M_i (Step 12) to retrieve IV_i and the encrypted E_i and H_i (Step 13). In Step 14, E_i and H_i are retrieved after decryption, and afterwards, a new HMAC H_{igen} is generated from the retrieved E_i and IV_i (Step 15). H_i and H_{igen} are compared to ensure the integrity of the message (Step 16). Finally, in Step 17, the plaintext message is retrieved by decrypting E_i with the key C_i and the initialization vector IV_i.

To communicate with the two services chosen, HTTP requests directed to each of the service's APIs were sent. For Imgur, the only requirement was to upload an image anonymously and get the URL of the image in return, when the process was completed. This was achieved by sending a POST request to the URL `https://api.imgur.com/3/image` with a payload of the "image" data (in base64), the "type" being base64, a "name" for the image and a "title" as well. A client ID is required to use the API but it does not need to be the user's real ID. The response data includes a variety of information, for example, the timestamp of when the image was uploaded, however, the application only grabs the image URL. For Dontpad, the application requires the ability to write and read data in a room of the choice of the application. In order to write data to a room a POST request is sent to the URL `https://api.dontpad.com/uuid` (with the *uuid* being the UUID of the chatroom) with a "text" payload of the message that needs to be written. To read data from a room a GET request is sent to the URL: `https://api.dontpad.com/uuid.body.json` with a "lastModified" payload with a value

of 0. This will return a JSON string that includes the body text with which the value of the text currently found in the room is associated.

Encrypting the image URL which is inserted into the Dontpad folder is a possibility to further increase confidentiality since the users already have a secret key that can be used, however, one may argue that this comes at the cost of more suspicion compared with the URL being in plaintext. Since this prototype focuses on anonymous and covert communications, even if it sacrifices confidentiality, the URL will not be encrypted.

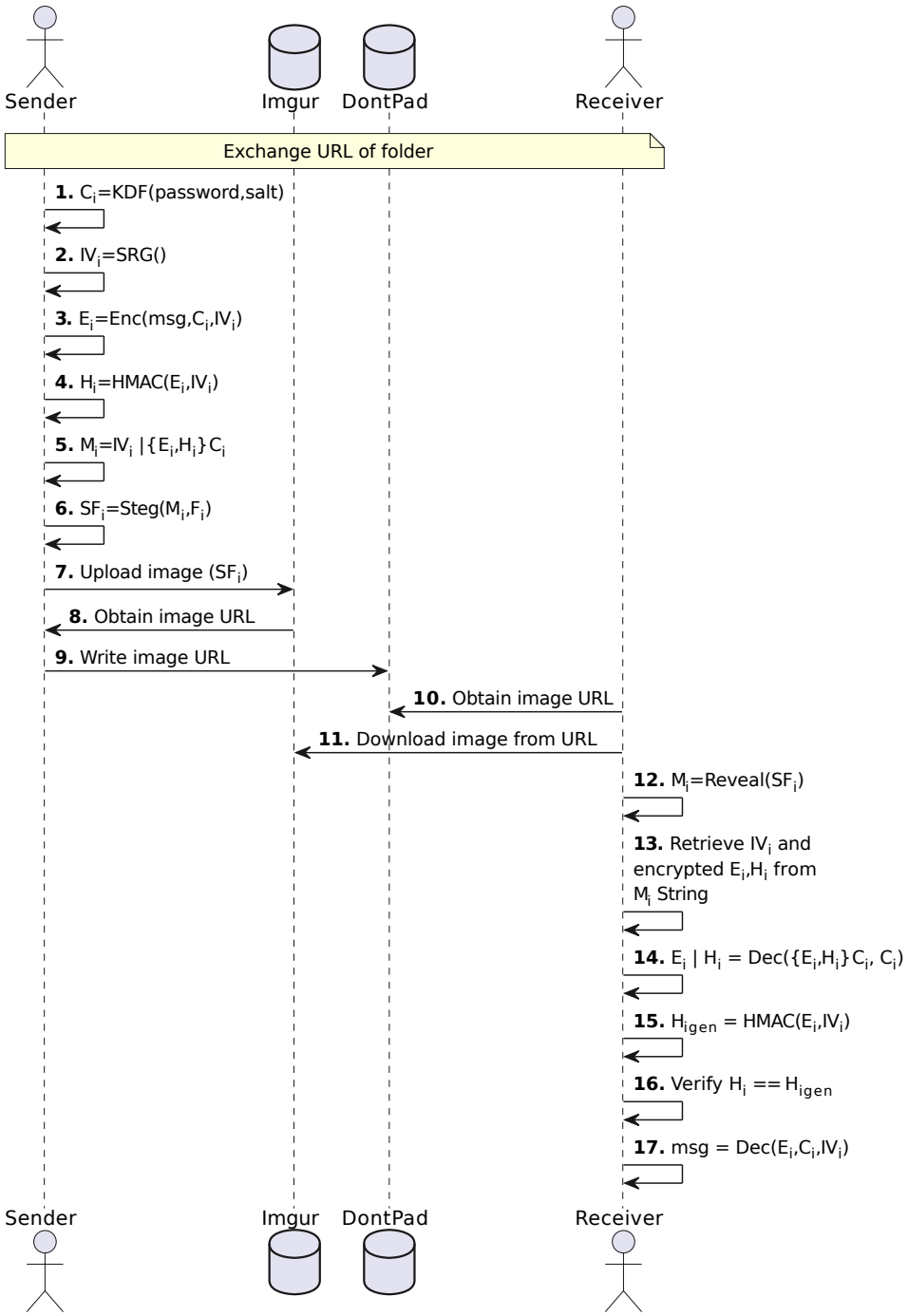


Figure 5.9: Sequence diagram of the proposed message service.

Chapter 6

Implementation

The application was implemented in Python and makes use of the cross-platform graphical framework Kivy to offer a touch-enabled user interface. KivyMD, a collection of material design-compliant widgets for Kivy, was also used. Figure 6.1 shows the structure of the code. A GitHub repository of this prototype can be found here [56].

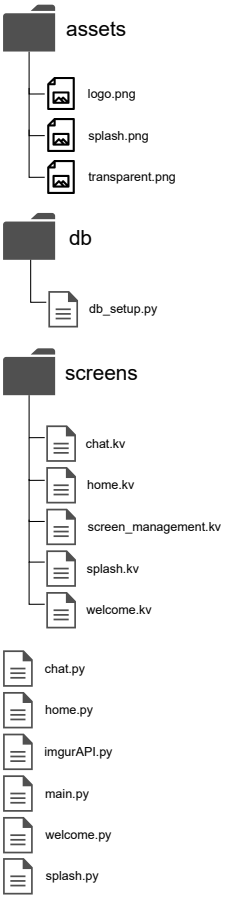


Figure 6.1: File structure.

The root folder has multiple python files and a few folders. The "assets" folder holds some visual assets used by the program, such as the application's logo or a still image used for a splash screen when opening the application. The "db" folder holds a python file responsible for the creation and first setup of the local database that will be used by the application. The "screens" holds ".kv" (kivy) files which are files that have information responsible for the formatting of the user interface. To each "screen" a python file is associated. For instance, for the "home.kv" screen file, there exists a python file named after it, "home.py" which holds the functions that the screen has. The "screen" folder has an extra ".kv" file, "screen_management.kv" which, as its' name says, is responsible for all operations that involve switching screens such as the act itself of going from one screen to another and what transition animation to show when that happens.

The next sections will review portions of code fundamental for the application's functionalities while mostly setting aside the code responsible for the user interface.

6.1 Main and Screen Manager

In the "main" python file, shown in Listing 6.1, the kivy files are loaded, the application class is initialized and certain values are setup - the default theme style (which defines whether the applications run in "Light" or "Dark" mode), the primary colour palette as well as the title of the application. The screen manager is also initialized in line 8 and the style of transition is chosen as well. After that, in lines 9-14, a list "screens" is populated associating to each focal python class the internal screen name, shown in the contents of the "screen_management.kv" file, presented in Listing 6.2. This ensures that when the name value is given to a certain variable, the application will move to the right class along with the screen. Afterwards, each item in the list is added as a widget to the screen manager object. The main application is run in lines 21-26 - the application will start a loop that will keep going until the user closes the application.

```
1 {
2     class MainApp(MDApp):
3         def build(self):
4             self.theme_cls.theme_style = "Dark"
5             self.theme_cls.primary_palette = "DeepPurple"
```

```
6               self.title = "PSST"
7
8               sm = ScreenManager(transition=SlideTransition())
9               screens = [
10                  SplashScreen(name="splash"),
11                  WelcomeScreen(name="welcome"),
12                  HomeScreen(name="home"),
13                  ChatScreen(name="chat")
14              ]
15              for screen in screens:
16                  sm.add_widget(screen)
17
18              return sm
19
20
21      if __name__ == "__main__":
22          loop = asyncio.get_event_loop()
23          loop.run_until_complete(
24              MainApp().async_run()
25          )
26          loop.close()
27  }
```

Listing 6.1: main.py contents

```
1   {
2       <ScreenManagement>:
3           id: "screens"
4           SplashScreen:
5               name: "splash"
6           WelcomeScreen:
7               name: "welcome"
8           HomeScreen:
9               name: "home"
10          ChatScreen:
11              name: "chat"
12  }
```

Listing 6.2: screen_management.kv contents

6.2 Splash Screen

This screen does not have a lot happening - it is the first screen loaded by the application and it is used only to show a still image for 5 seconds before transitioning to the welcome screen, nicknamed "welcome", where the user will be able to log in. The contents of "splash.py" are shown in Listing 6.3. It is important to note that the "on_enter" function (line 3) is called whenever the screen is accessed. The variable "self.manager.current", responsible for changing screens, is used in the function "switch_to_welcome". Whenever the application requires to change screens, the only thing required is to give the "name" value, shown in Listing 6.2, to this variable. In this case, since the application's screen needs to be redirected to the welcome screen, the value "welcome" is given to the variable.

```
1 {
2     class SplashScreen(Screen):
3         def on_enter(self, *args):
4             Clock.schedule_once(self.switch_to_welcome, 5)
5 
6         def switch_to_welcome(self):
7             self.manager.current = "welcome"
8 }
```

Listing 6.3: splash.py contents

6.3 Welcome Screen

In the "welcome" screen the user will be presented with a login form that requests a password. The application checks if a local database already exists and depending on if it exists or not it will either display one password input field or two. This piece of code is presented in Listing 6.4. If the application does not find a local database (line 3), it will set a local variable to false (line 4), which is used in another function, and the extra password input field, used to confirm the password input, is made visible to the user.

```
1 {
2     def check_file(self):
3         if not self.path.is_file():
4             self.f = False
```

```
5                 self.ids.wsPasswordConfirm.required = True
6                 self.ids.wsPasswordConfirm.opacity = 1
7                 self.ids.wsPasswordConfirm.disabled = False
8             else:
9                 self.f = True
10                pass
11    }
```

Listing 6.4: welcome.py check_file function

Once the user inputs their password in the password input fields, they have to click a button to proceed. This button will call the function presented in Listing 6.5. This function will start by checking the value of the local variable "f" to know whether a local database already exists or not. The local database is created with the package "pysqlitecipher", a lightweight SQLite wrapper with encryption built in. When creating a database it receives a plaintext passphrase as input and then hashes it with SHA-512 and creates a secret key through cryptography-fernet's algorithm. This secret key is then encrypted through One Time Pad (OTP) with an SHA-256 hash as a key and stored in the database itself, being decrypted whenever the user logins and starting an instance of fernet to encrypt all data. The SHA-512 password hash is also stored and can be accessed through the method "getVerifier" used in line 4. In line 5, the password which was input by the user is hashed and it is used in line 6 to verify if the credentials inserted match.

In the case that a database needs to be created another python file, "db_setup.py", dedicated to the creation of the database is called. In it, the tables and their columns are defined and then created after the creation of the database itself. Afterwards, a record is added to the "local" table with a generated UUID4 which will be the user's UUID. The method "open_message", used in lines 8, 10, 15, and 17, when called opens a dialogue popup box with the message received as an argument.

```
1  {
2      def press_ok(self):
3          if self.f:
4              x = SqliteCipher.getVerifier(dataBasePath="db/pydatabase.db
                   ", checkSameThread=False)
5              y = SqliteCipher.sha512Convertor(self.ids.wsPassword.text)
6              if x == y:
```

```
7                   self.parent.current = "home"
8                   self.open_message("Welcome back.")
9               else:
10                  self.open_message("Password incorrect.")
11          else:
12              if self.ids.wsPassword.text == self.ids.wsPasswordConfirm.
                    text:
13                  db_setup.db_creation(self.ids.wsPassword.text)
14                  self.parent.current = "home"
15                  self.open_message("Please do not forget your password.
                        You will need it whenever you open this app.")
16              else:
17                  self.open_message("Passwords do not match. Try again.")
18  }
```

Listing 6.5: welcome.py press_ok function

6.4 Home Screen

In the "home" screen, the user has access to a list of chatrooms and another for contacts. As soon as the user enters the screen the "update" method is called. This method's code is presented in Listing 6.6. It starts by clearing the scroll list of any widgets (effectively emptying out the whole chatroom and contact list) and afterwards getting all data from the "local" (line 6), "chatroom" (line 13), and "contact" (line 16) tables. The data recovered from the "local" table, the user's UUID, is given to the variable "user_uuid" in the "chat" screen in line 11. The data recovered from the "chatroom" and "contact" tables are run through in a loop in lines 18-22, where for each record found a new widget is added to the list.

```
1   {
2       def update(self):
3           self.ids.chatroomListWidget.clear_widgets()
4           self.ids.contactListWidget.clear_widgets()
5
6           localList = self.o.getDataFromTable(self.t_local,
                raiseConversionError=True, omitID=False)
```

```
7           localUUID = localList[1][0][1]
8
9           self.ids.currentUserUUID.text = localUUID
10          next_screen = self.parent.get_screen("chat")
11          next_screen.user_uuid = localUUID
12
13          chatroomList = self.o.getDataFromTable(self.t_chatroom,
                raiseConversionError=True, omitID=False)
14          chatroomListData = chatroomList[1]
15
16          contactList = self.o.getDataFromTable(self.t_contact,
                raiseConversionError=True, omitID=False)
17          contactListData = contactList[1]
18
19          for g in chatroomListData:
20              self.ui_add_chatroom(g[0], g[1], g[2], g[3], g[4])
21
22          for c in contactListData:
23              self.ui_add_contact(c[0], c[1], c[2])
24  }
```

Listing 6.6: home.py update function

It's important to note that the data received from all tables come in the following structure: {"C_1","C_2",...}{{"A_1","A_2",...},{"B_1","B_2",...}}. The response will always be two lists, the first one, containing C_1 and C_2, reference the table's column names. The second list has a list for each record in the table which contains the values for each column. For instance, for the first record in the table, the value for column C_1 is A_1, while for the second record, the value for the same column is B_1. The same logic applies for column C_2 and so on. That is why in lines 13 and 16 the data in index 1 is grabbed, which is the list that contains the values, and not the column names.

The method used in lines 20 and 23, are functions used only for the purpose of adding widgets to the user interface, which in the former, receives the "ID", "name", "uuid", "salt" and a boolean variable "keepMsg", responsible for whether messages are stored or not, of a group and, in the latter, receives the "ID", "name" and "uuid" of a contact. The code of these functions is presented in the Listing 6.7. This function, while adding each

widget, binds various methods to different actions. When the user clicks the widget on the chatroom list, this will call a method that will redirect the user to the chat screen associated with the item chosen. When the user clicks the trashcan, found on the right side of each list item, the functions shown in Listing 6.8 are called and after that, another method, found in lines 10 and 18, is called (remove_widget) which, in this situation, removes that specific item from the list. The icon widget is added to the list item widget and after that added to the main list widget.

```
1  {
2      def ui_add_chatroom(self, group_id, name_text, uuid_text, salt,
           keepMsg):
3          nItem = TwoLineRightIconListItem(text=name_text,
4                                           secondary_text=uuid_text,)
5          nItem.bind(on_release=lambda x: self.go_to_chat(name_text,
               uuid_text, salt, keepMsg))
6          nIcon = IconRightWidget(icon="trash-can",
7                                  on_release=lambda x: self.
                                       remove_chatroom(group_id, uuid_text,
                                        nItem),)
8          nItem.add_widget(nIcon)
9
10         self.ids.chatroomListWidget.add_widget(nItem)
11
12     def ui_add_contact(self, contact_id, name_text, uuid_text):
13         nItem = TwoLineRightIconListItem(text=name_text,
14                                          secondary_text=uuid_text,)
15         nIcon = IconRightWidget(icon="trash-can",
16                                 on_release=lambda x: self.
                                       remove_contact(contact_id, nItem),)
17         nItem.add_widget(nIcon)
18
19         self.ids.contactListWidget.add_widget(nItem)
20 }
```

Listing 6.7: home.py add UI functions

When the user clicks the trash-can icon to delete a chatroom the following functions are called. When deleting a contact, what happens is that the record is deleted from the

database. However, when deleting a chatroom, the message data linked to that chatroom is deleted alongside the chatroom record.

```
{
    def remove_chatroom(self, groupID, groupUUID, instance):
        messageList = self.o.getDataFromTable(self.t_message,
            raiseConversionError=True, omitID=False)
        messageListData = messageList[1]

        for m in messageListData:
            if m[4] == groupUUID:
                self.o.deleteDataInTable(self.t_message, m[0], commit=
                    True, raiseError=True, updateId=False)
        self.o.deleteDataInTable(self.t_chatroom, groupID, commit=True,
             raiseError=True, updateId=False)

        self.ids.chatroomListWidget.remove_widget(instance)

    def remove_contact(self, contactID, instance):
        self.o.deleteDataInTable(self.t_contact, contactID, commit=True
            , raiseError=True, updateId=False)
        self.ids.chatroomListWidget.remove_widget(instance)
}
```

Listing 6.8: home.py remove functions

6.5 Chat Screen

This screen is where all communications happen - all other screens work with local data and have no communication whatsoever with other systems. It is also the screen where encrypting/decrypting and hiding/revealing data happens. Like the previous screen, the "home" screen, whenever a user enters this screen an "update" method is called. The code is shown in Listing 6.9. This method starts by locking the widgets responsible for sending messages, to first ensure that the user has Internet access. Like the lists in the "home" screen, every time this screen is loaded, it clears the scroll view, dedicated to presenting the messages, of any messages. Afterwards, all messages are retrieved from the

database, and for each message that has a "chat_uuid" equal to the current chatroom, a method to add the message to the scroll view is run. This is merely the part of the script that presents past messages. The script then checks if the user has input a passphrase, and if it has not it will run a method that will open a popup window with an input field. After that, it begins running the method "start_get_messages_thread" every 60 seconds. Since there is no central server, there were a lot of difficulties implementing a broadcast system to notify users that there is a new URL available in the text-sharing service, and as a result, also a new image available in the image-sharing service. The alternative, which was chosen, requires running a script in an interval to check whether there are new messages and if there are, retrieve them. This method will start a new thread and run the method "get_messages". Starting a new thread to run this portion of the code is very important - without running it on a new thread, it will instead run on the main thread, which is the thread responsible for maintaining the user interface. This will, in turn, cause all of the graphic components to freeze for as long as the application takes to retrieve the messages, creating an undesirable user experience. By creating a new thread to run this code, the main thread does not need to "stop" working on the user interface.

```
1  {
2      def update(self):
3          self.ids.gallerybutton.disabled = True
4          self.ids.chatbox.disabled = True
5          self.ids.sendbutton.disabled = True
6
7          self.ids.chat_layout.clear_widgets()
8
9          messageList = self.o.getDataFromTable(self.t_message,
               raiseConversionError=True, omitID=False)
10         messageListData = messageList[1]
11
12         for m in messageListData:
13             if m[4] == self.chat_uuid:
14                 self.m_builder(m[1], m[2], m[3])
15
16         if self.pwd is None:
17             self.pwd_insert()
```

```
18
19          self.task = Clock.schedule_interval(self.
                start_get_messages_thread, 60)
20  }
```

Listing 6.9: chat.py update function

In Listing 6.10 the code of the method that is called in intervals of 60 seconds as well as the method that starts a new thread to run it is shown. For the former, the script starts by checking if a passphrase was inserted and if it has not, the script will not go any further. Another check is made to verify if the system has an Internet connection. After the checks are successful, all buttons will be disabled to ensure the user cannot disrupt the process in any way. Afterwards, a key is generated in the "get_key" method, which is available in Listing 6.13. This key is generated through Argon2, which receives a passphrase and a salt which is, securely generated when creating the chatroom, being a 32-byte hash-token. After generating the key, the method "download_function", available in Listing 6.11 is called to receive a list of plaintexts. A check is made to verify if the list returned a populated list. If it has not, it stops the method and enables the buttons again. If the list received is populated, the values of the sender's UUID, the message itself, and the timestamp of when it was sent are passed as arguments to the method "m_builder" which will add the message to the scroll view. It is worthy of note that the method "m_builder" is set to always run in the main thread, even if called from another thread, since it is responsible for adding messages to the scroll view, or in other words, updating the user interface, which can not be done from a thread other than the main one. Finally, the messages are inserted into the "message" table and the buttons are enabled. The latter function will only do two things - check if another thread has already been created (which means the application is already retrieving messages or sending messages at the moment) and if it has not, start a new thread to run the "get_messages" function.

```
1  {
2      def get_messages(self, arg):
3          if self.pwd is None:
4              return
5
6          net = self.safe_internet()
```

```
7         if not net:
8             return
9
10        self.ids.backbutton.disabled = True
11        self.ids.dotsbutton.disabled = True
12        self.ids.gallerybutton.disabled = True
13        self.ids.chatbox.disabled = True
14        self.ids.sendbutton.disabled = True
15
16        ci = self.get_key(self.pwd, self.salt)
17        plaintext = self.download_function(self.chat_uuid, ci)
18
19        if plaintext is None:
20            self.ids.backbutton.disabled = False
21            self.ids.dotsbutton.disabled = False
22            self.ids.gallerybutton.disabled = False
23            self.ids.chatbox.disabled = False
24            self.ids.sendbutton.disabled = False
25            return
26
27        for p in plaintext:
28            text = p.split("&#")
29
30            sender = text[0]
31            content = text[1]
32            timestamp = text[2]
33
34            self.m_builder(sender, content, timestamp)
35
36            i_message = [sender, content, timestamp, self.chat_uuid]
37            self.o.insertIntoTable(self.t_message, i_message, commit=
                  True)
38
39        self.ids.backbutton.disabled = False
40        self.ids.dotsbutton.disabled = False
41        self.ids.gallerybutton.disabled = False
42        self.ids.chatbox.disabled = False
43        self.ids.sendbutton.disabled = False
```

```
44          self.working = None
45
46      def start_get_messages_thread(self, arg):
47          if self.working is None:
48              self.working = True
49              Thread(target=self.get_messages).start()
50  }
```

Listing 6.10: chat.py get messages function

In the method "download_funtion", a GET request is sent to the text-sharing service Dontpad to retrieve the text available in the room at the time. The response is then split by newlines. After that, for each line, the content of each message is processed and the sender's UUID and image's URL are retrieved. The sender is compared to the current user and if it is the same UUID, the line is concatenated to a variable "rewrite" which will be used later to rewrite the text-sharing service. If the UUIDs do not match the image URL is accessed and the image is downloaded. After that, the hidden data in the image is revealed and then the encrypted data is decrypted, shown in Listing 6.12 and the plaintext is appended to a list, which will be returned.

```
1   {
2       def download_function(self, room, ci):
3           myurl = 'https://api.dontpad.com/'+ room +'.body.json'
4           posturl = 'https://api.dontpad.com/'+ room
5           rewrite = ""
6
7           plaintextList = []
8
9           payload = {'lastModified': '0'}
10          response = requests.get(myurl, data=payload)
11
12          responseJSON = json.loads(response.text)
13          text = responseJSON['body']
14
15          if text == "":
16              return
17
18          lines = text.split("\n")
```

```
19
20          for content in lines:
21              altered_content = content[:-1]
22              altered_content = altered_content.split("#&")
23              sender = altered_content[0]
24              imageUrl = altered_content[1]
25
26              if sender != self.user_uuid:
27                  try:
28                      image = requests.get(imageUrl).content
29                  except:
30                      self.popup_error("Could not retrieve image from
                            Imgur.")
31                      return
32
33                  x = imageUrl.split("/")
34                  filename = x[3]
35
36                  path = "files/"
37                  path += filename
38
39                  with open(path, "wb") as handler:
40                      handler.write(image)
41
42                  try:
43                      mi = lsb.reveal(path)  # secret message is revealed
44                  except:
45                      self.popup_error("Could not retrieve hidden data
                            from image. Steganography compromised.")
46                      return
47
48                  plaintext = self.decrypt(bytes.fromhex(mi), ci)
49                  plaintextList.append(plaintext.decode("utf-8"))
50              else:
51                  rewrite += content+"\n"
52
53          if rewrite == "":
54              payload = {'text': ''}
```

```
55              requests.post(posturl, data=payload)
56          else:
57              payload = {'text': rewrite}
58              requests.post(posturl, data=payload)
59
60          return plaintextList
61  }
```

Listing 6.11: chat.py download function

The "decrypt" function, responsible for decrypting data, is present in Listing 6.12. Data is encrypted with AES-256-CBC. The function receives the encrypted data and a key. It starts by decoding the encrypted data from base64. The IV is retrieved from the first 16 bytes. A cypher object is initialized with the key and IV in Cipher-Block-Chaining (CBC) mode. Afterwards, the HMAC and ciphertext are retrieved from the remaining data. The data is decrypted and a padded message is returned. This padded message is unpadded and then the HMAC is retrieved from the first 64 bytes of the message. The remaining bytes are the message itself, the ciphertext. This ciphertext is turned into binary from hex. With the HMAC available, a new HMAC is generated with the IV and ciphertext and then compared to the HMAC received. If they do not match then it is assumed that the file has been compromised in some way. If they match, a new cypher object is created and then the ciphertext is decrypted which, now, returns a padded plaintext. This padded plaintext is unpadded with the result being the final plaintext. The reasoning behind a new cypher object being initialized in line 23 is due to how Cipher-Block-Chaining works. If the cypher object was not reinitialized, the first 16 bytes of the plaintext end up remaining encrypted, while the rest of the message is retrieved. This is because, with Cipher-Block-Chaining, the IV is only used on the first 16 bytes, and the remaining blocks make use of data from the previous blocks as its' "IV".

Another available choice for encryption and authentication is "XChaCha20-Poly1305" [41]. "ChaCha20-Poly1305" is a combination of the encryption algorithm "ChaCha20" and the message authentication code generation algorithm "Poly1305". When encrypting a large volume of messages with the same secret key in this combination of algorithms, since the nonce is only 12 bytes, it becomes more likely that a nonce will be re-used accidentally,

which is a disadvantage. "XChaCha20-Poly1305" fixes this issue by upping the size of the nonce to 24 bytes, however, AES-256-CBC was chosen for the prototype due to its support and implementation details information.

```
{
    def decrypt(self, enc, key):
        enc = base64.b64decode(enc)
        iv = enc[:16]
        cipher = AES.new(key, AES.MODE_CBC, iv)
        etMsg = enc[16:]

        try:
            padMsg = cipher.decrypt(etMsg)
        except:
            self.popup_error("Error occured during decryption - key was
                not accepted.")
            return

        tMsg = unpad(padMsg)
        hi = tMsg[:64].decode("utf-8")
        eiHex = tMsg[64:len(tMsg)]
        ei = bytes.fromhex(eiHex.decode("ascii"))
        ahi = hmac.new(iv, ei, hashlib.sha256).hexdigest()

        if not self.check_signature(hi, ahi):
            self.popup_error("Signature mismatch - file has been
                compromised.")
        else:
            cipher = AES.new(key, AES.MODE_CBC, iv)
            paddedRaw = cipher.decrypt(ei)
            unpaddedRaw = unpad(paddedRaw)
            return unpaddedRaw
}
```

Listing 6.12: chat.py decrypt function

The secret key generated whenever the application sends or receives messages is generated through a KDF. The chosen KDF, as mentioned before, is Argon2. The following method "get_key" is the function responsible for secret key generation. Since the users get

asked whenever they enter the chatroom to input a passphrase, and the salt remains the same, users can refresh the secret key used in the chatroom by using a different passphrase. This, in turn, will generate a different secret key. This does create a problem in this implementation - if there are unread messages in the text-sharing and image-sharing services that have been encrypted with an old passphrase-generated key, decryption will fail with the new one.

```
1 {
2     def get_key(self, pwd, s):
3         k = hash_password_raw(
4             time_cost=16, memory_cost=2 ** 15, parallelism=2, hash_len
                =32, password=pwd.encode("utf-8"),
5             salt=s.encode("utf-8"), type=argon2.low_level.Type.ID
6         ) # key derivation function initialization & key created (
              nonce/salt used as well)
7         return k
8 }
```

Listing 6.13: chat.py get key function

In Listing 6.14, the application first verifies if an appropriate image file has been chosen and also that the text input field for the message is not empty. A check for if the user has Internet access is also made. If all of these situations are valid, all buttons are disabled and then the file path of the image file is processed so only the filename is recovered. The message is processed to include the user's UUID, the message, and the current timestamp. A new secret key is generated and then the upload function is called, available in Listing 6.15, which will attempt to upload the chosen image with the encrypted message hidden. If the result is favourable, the message is saved to the database and added to the user interface, otherwise, the buttons will be re-enabled and the text input will be cleared of text. Like "get_messages", the function "send_message" is called by a function (lines 55-58) that will start a new thread to run it.

```
1 {
2     def send_message(self):
3         if self.filepath is None:
4             self.popup_error("Please␣choose␣an␣image␣to␣hide␣data.")
5         else:
```

```
6               if self.ids.chatbox.text == "":
7                   self.popup_error("Please write a message.")
8               else:
9                   net = self.safe_internet()
10                  if not net:
11                      return
12
13                  self.ids.backbutton.disabled = True
14                  self.ids.dotsbutton.disabled = True
15                  self.ids.gallerybutton.disabled = True
16                  self.ids.chatbox.disabled = True
17                  self.ids.sendbutton.disabled = True
18
19                  path = self.filepath.replace("\\", "/")
20                  filename = path.split("/")
21                  f = filename[-1]
22                  f = token_hex(8) + f[-4:]
23
24                  msg = self.user_uuid
25                  msg += "&#"
26                  msg += self.ids.chatbox.text
27                  msg += "&#"
28                  t = time.time()
29                  msg += str(t)
30
31                  ci = self.get_key(self.pwd, self.chat_salt)
32                  result = self.upload_function(path, f, msg, self.
                        chat_uuid, ci)
33
34                  if result:
35                      if self.keepMsg == "False":
36                          pass
37                      else:
38                          i_message = [self.user_uuid, self.ids.chatbox.
                                text, t, self.chat_uuid]
39                          self.o.insertIntoTable(self.t_message,
                                i_message, commit=True)
40                      self.m_builder(self.user_uuid, self.ids.chatbox.
```

```
                        text, t)
41                      self.ids.backbutton.disabled = False
42                      self.ids.dotsbutton.disabled = False
43                      self.ids.gallerybutton.disabled = False
44                      self.ids.chatbox.disabled = False
45                      self.ids.sendbutton.disabled = False
46                      self.working = None
47                  else:
48                      self.ids.backbutton.disabled = False
49                      self.ids.dotsbutton.disabled = False
50                      self.ids.gallerybutton.disabled = False
51                      self.ids.chatbox.disabled = False
52                      self.ids.sendbutton.disabled = False
53                      self.working = None
54
55      def start_send_message_thread(self):
56          if self.working is None:
57              self.working = True
58              Thread(target=self.send_message).start()
59  }
```

Listing 6.14: chat.py send message function

In the method "upload_function", presented in Listing 6.15, the plaintext is encrypted and afterwards hidden in the image in hex format. The resulting image of the message being hidden is saved and then encoded in base64 and uploaded anonymously to the image-sharing service (Imgur) through their API. The final message, to put in the text-sharing service, is processed and a POST request is sent to their API with the final message as the payload.

```
1   {
2       def upload_function(self, imgOriginal, imgSecret, msg, room, ci):
3           myurl = 'https://api.dontpad.com/'+ room
4
5           mi = self.encrypt(msg, ci)
6
7           try:
8               sfi = lsb.hide(imgOriginal, mi.hex())  # message hidden in
```

```
            image
        except:
            self.popup_error("Image not appropriate for steganography.")
            return

        sfi.save(imgSecret)  # altered image file saved

        imageName = "testImage"
        imageTitle = "Test"

        try:
            with open(imgSecret, "rb") as secretImage:
                data = base64.b64encode(secretImage.read())
                result = imgurAPI.uploadImageAnonymous(data, imageName,
                    imageTitle)
        except:
            return

        finalMsg = self.user_uuid
        finalMsg += "#&"
        finalMsg += result

        payload = {'text': 'test2night'}
        requests.post(myurl, data=payload)

        return True
}
```

Listing 6.15: chat.py upload function

Encrypting the plaintext, shown in Listing 6.16, requires the generation of a new IV of 16 bytes (AES.block_size). A cypher object is initialized with the secret key and newly generated IV in CBC mode. Afterwards, the plaintext is padded and then encrypted. An HMAC is generated from the ciphertext and IV. The ciphertext is turned to hex from binary and has the newly-generated HMAC prepended to it. This result of HMAC+ciphertext is padded and encrypted after initializing a new cypher object. Finally,

the result is encoded in base64 and returned.

```
{
    def encrypt(self, raw, key):
        iv = Random.new().read(AES.block_size)  # iv generation
        cipher = AES.new(key, AES.MODE_CBC, iv)  # cipher object
        paddedRaw = pad(raw)  # plaintext padding
        ei = cipher.encrypt(paddedRaw)  # plaintext encryption
        hi = hmac.new(iv, ei, hashlib.sha256).hexdigest()  # hmac
            generation
        eiHex = ei.hex()  # cyphertext bynary to hex
        tMsg = hi + eiHex  # prepend hmac to cyphertext
        padMsg = pad(tMsg)  # pad hmac+cyphertext
        cipher = AES.new(key, AES.MODE_CBC, iv)  # cipher object
            refresh
        etMsg = cipher.encrypt(padMsg)  # encrypt hmac+cyphertext
        return base64.b64encode(iv + etMsg)
}
```

Listing 6.16: chat.py encrypt function

Chapter 7

Validation

In this chapter, it is shown that the developed prototype achieved all requirements. Some graphs detailing the runtimes and performance of the prototype, as well, as a brief discussion on the security of this concept, were also shown.

7.1 Functional Validation

All the requirements described in Chapter 5 were accomplished as follows:

(R.1) In terms of ensuring covert communications, this is achieved by foregoing using centralized servers, using steganography, and using multiple servers as proxies to transfer data, shown in Figure 5.9.

(R.2) Encrypted message exchange is achieved by all messages being encrypted with AES-256-CBC with an Argon2 generated secret key, as presented in Listing 6.16 and 6.13.

(R.3) As for user authentication, whenever a user opens the application, it will boot up to a login screen where they must input a password which will be verified against a hash in a local database, furthermore, when opening a chatroom, the user will once again be requested a passphrase making it unlikely for anyone other than the owner to use the application effectively, shown in Listing 6.5;

(R.4) Cryptographic keys can be refreshed at will by changing the passphrase input when opening the chatroom. This requires both users to agree on a passphrase and when

it will be implemented;

(R.5) The text data in DontPad is deleted upon being received at the destination and the image data in Imgur is unlisted being only available to those who have the URL;

(R.6) All communications are anonymous, without, the user requiring to login. This anonymity can be furthered even more by using Virtual Private Networks (VPNs) and TOR.

(R.7) The users can opt in/out of storing messages locally through a slider in the chatroom screen.

(R.8) As soon as a message is read by the recipient, the text present in the text-sharing online service is deleted. The application is able to discern whether the sender of the message was the current user or someone else through the UUID linked to the message. If it matches, the message remains as is, for the recipient to eventually read.

A script was developed that runs the same download/upload functions shown in Listing 6.11 and Listing 6.15, however without UI functionality. This script generated a secret key, encrypted a six-byte message, hid it into three different images of different sizes (256×256 - 83.4KB, 512×512 - 611KB, 1024×1024 - 597 KB), uploaded the resulting image into Imgur, wrote the image URL to a room in Dontpad, then downloaded that text from the room, downloaded the image, revealed the message and finally decrypted the message to get the plaintext six-bytes message. This was done fifty times for each image, totalling one-hundred-fifty images. Elapsed times for each execution were recorded and are presented in Table 7.1 and graphically in Figure 7.1.

It is worthy of note that the phase that uses up the most amount of time is the image upload followed by retrieving the image URL from Dontpad, for small images, and downloading the image from Imgur, for medium and large-sized images. The encryption and decryption phases were negligible, only ever reaching a max runtime of about 3 milliseconds, this is why these do not seem to appear in Figure 7.1.

The application was tested in a system running an AMD Ryzen 5 5600H with 16 GB of RAM and an NVIDIA RTX 3060. Overall the system has more than enough power.

	SMALL		MEDIUM		LARGE	
	ms	%	ms	%	ms	%
Encrypt	0.42 ± 0.67	$\sim 0\%$	0.12 ± 0.33	$\sim 0\%$	0.18 ± 0.38	$\sim 0\%$
Hide	31.43 ± 3.55	1.12%	34.27 ± 0.47	0.60%	460.68 ± 15.71	6.12%
Imgur Upload	1280.09 ± 149.91	45.45%	3786.86 ± 547.53	66.69%	4954.71 ± 827.26	66.67%
Dontpad Upload	487.05 ± 33.78	17.29%	497.02 ± 125.19	8.75%	492.64 ± 143.06	6.63%
Dontpad Download	490.81 ± 100.33	17.43%	519.69 ± 238.07	9.15%	516.59 ± 158.26	6.95%
Imgur Download	372.78 ± 91.39	13.24%	681.54 ± 207.38	12.01%	837.70 ± 342.33	11.27%
Reveal	7.43 ± 0.62	0.26%	12.28 ± 1.33	0.22%	21.89 ± 1.84	0.30%
Decrypt	0.12 ± 0.33	$\sim 0\%$	0.04 ± 0.19	$\sim 0\%$	0.16 ± 0.37	$\sim 0\%$
Key Generation	146.35 ± 0.97	5.20%	146.51 ± 1.89	2.58%	146.46 ± 0.88	1.97%

Table 7.1: Average runtimes by phase with standard deviation and %.

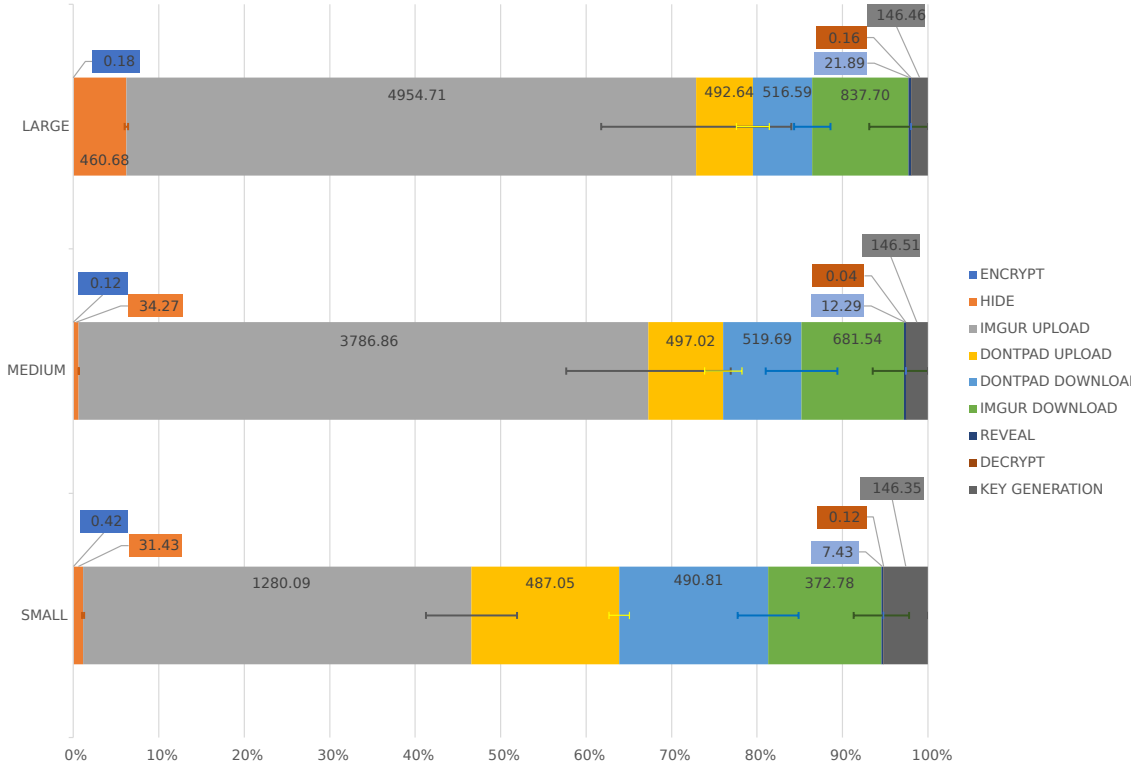


Figure 7.1: Runtimes

The application reached a peak of 7% CPU usage and a little over 1% system memory over 5 minutes of execution. The usages of this system are presented in Figure 7.2.

This graph can be divided into seven different stages:

A. Stage A includes the login phase represented in the first 8 seconds of runtime. The application sees a spike of CPU usage to a little under 1% that occurs when the application goes through the passphrase verification with the local database, decrypts it and then when it changes screen it sees a CPU spike of 2.8%. Over the remaining graph, most of the CPU spikes that go to around 2%-4% are caused by UI updating,

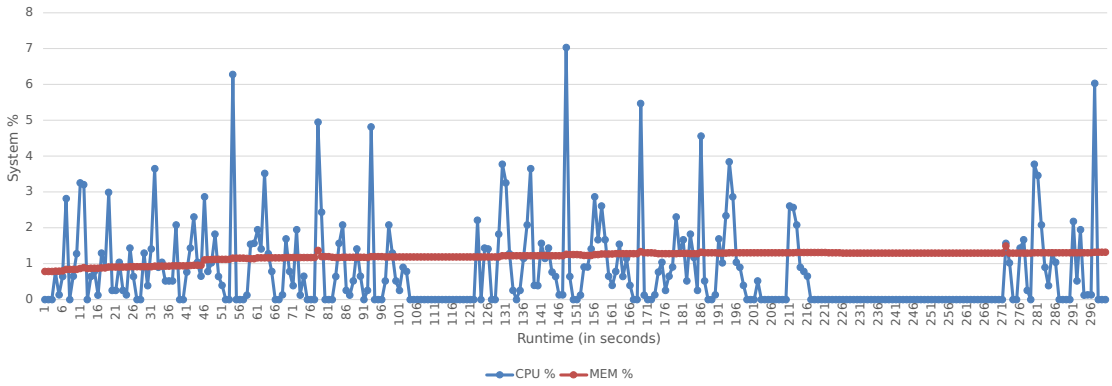


Figure 7.2: Usage - CPU/Memory Usage

which ranges from switching screens, adding widgets or removing them.

B. Stage B includes the process of creating, reading and deleting chatrooms or contacts. These processes usually did not cause much stress to the CPU, peaking at around 1.43%. This stage can be seen happening from seconds 9 to 32. The 3.65% CPU peak at approximately 31 seconds, which entails the screen switching, is when stage C starts.

C. Stage C is the act of accessing the chatroom and acting within it. It is divided into three steps since, in this scenario, the user accesses chatrooms three different times. The 1% spikes are caused by the user using the scroll view to scroll the messages shown or writing in the text input field. The 2%-3% spikes are caused by the user opening the chatroom's information widget, *filechooser* (to select an image). This stage is one of the more prominent ones. From seconds 33 to 128; 140 to 194; and 281 to 300 the user is found in a chatroom. As such the user is found in Stage C, however, inside this stage the user can also start other stages - Stages D, E, and G.

D. Stage D represents the act of the user sending the message, but it does not include the act of choosing an image and writing the message. It is only the act of the application sending the message. This stage was divided into 6 steps corresponding to the 6 messages sent during it. This stage is categorized by its' noticeable CPU usage spikes, which can range from 4% up to 7%. The spikes in seconds 54, 78, 148, 169, 186 and 297 are all spikes originating from the user sending a message. These spikes are usually followed by a smaller CPU usage spike of around 2% after 5 to 6

seconds have passed since the bigger usage spike. This smaller usage spike is caused by the application adding a widget to the scroll view (the message sent).

E. Stage E represents the act of not doing anything, leaving the application idle. This stage was divided into 2 steps corresponding to the user idling inside a chatroom and inside the "home" screen. This can be seen whenever the CPU usage drops to 0% and stays there for a prolonged amount of time. Seconds 104-122 show this stage while the user is idle inside a chatroom. Seconds 217-271 show this stage while the user is idle inside the "home" screen.

F. Stage F represents the act of leaving a chatroom and then re-entering another room. This can be recognized by two consecutive 3%-4% CPU usage spikes. In seconds 129 to 139, this stage can be seen, where the user leaves a current chatroom, thus causing a screen change (which causes the spike) and then a few seconds later re-entering a chatroom (once again causing a screen change).

G. Stage G represents the act of receiving a message from another user. This is a method called periodically, every 60 seconds, that checks whether a new message is available and if so, retrieves it and presents it to the user. An instance of that happening can be seen in seconds 93 to around 99.

Data collecting the average, peak, and standard deviation values for CPU and memory usage are presented in Table 7.2.

	Seconds	CPU Average	CPU Standard Deviation	CPU Peak	MEM Average	MEM Standard Deviation	MEM Peak
Stage A	1 to 8	0.54%	0.91%	2.82%	0.80%	0.02%	0.84%
Stage B	9 to 32	1.08%	1.09%	3.65%	0.89%	0.02%	0.94%
Stage C	33 to 129	0.79%	1.14%	6.28%	1.14%	0.08%	1.37%
	139 to 194	1.15%	1.37%	7.03%	1.27%	0.03%	1.33%
	281 to 300	1.01%	1.50%	6.03%	1.31%	0.01%	1.32%
Stage D	54 to 60	1.36%	2.12%	6.28%	1.15%	0.01%	1.16%
	78 to 84	1.37%	1.70%	4.95%	1.21%	0.07%	1.37%
	148 to 154	1.37%	2.34%	7.03%	1.25%	0.01%	1.26%
	169 to 175	1.08%	1.83%	5.47%	1.30%	0.02%	1.33%
	186 to 192	1.13%	1.51%	4.56%	1.31%	0.01%	1.32%
	297 to 300	1.51%	2.61%	6.03%	1.32%	0.00%	1.32%
Stage E	104 to 122	0.00%	0.00%	0.00%	1.19%	0.00%	1.19%
	217 to 271	0.00%	0.00%	0.00%	1.30%	0.00%	1.31%
Stage F	130 to 138	1.75%	1.42%	3.78%	1.22%	0.00%	1.22%
Stage G	93 to 99	1.25%	1.63%	4.82%	1.19%	0.01%	1.20%

Table 7.2: CPU/MEM system usage.

7.2 Security Discussion

The objective of the proposed messaging service is to enable covert communication guaranteeing the confidentiality, integrity, and authentication of the exchanged messages while using images as containers for the message exchange. Considering that the envisioned concept uses online services, the Dolev–Yao intruder model [15] was considered adequate for this security analysis. The Dolev–Yao intruder model states that the intruder has complete control over the network, being capable of reading, altering, or deleting data in transit.

In the designed Proof of Concept (PoC), the KDF, Secure Random Generator (SRG) [71, 49, 43], and HMAC are assumed to be secure, and the channel's password and salt are assumed to be previously exchanged in a secure manner, preferably in-person, and generated using a SRG. Moreover, it is also assumed that the UUID is securely generated and that the used images are either chosen randomly from online galleries or from the user's gallery accordingly to user preference.

Following these assumptions, a discussion can be made about user anonymization, message confidentiality, message integrity, and system availability.

User anonymity is assured with the usage of generally available online services that accept unauthenticated usage, combined with the inability to distinguish between users that take part in a channel. Only users having the password and salt of a channel can read messages exchanged in that channel, and it is also assumed that user equipment is secure in a way that the used services do not collude or pursue user reidentification based on their access patterns or IP address (such as a non-rooted phone). Aside from using multiple services, users can also rely on VPN [8, 73, 18, 1, 30] services or a custom-built service on the TOR [28] network to enhance user anonymity. Furthermore, the Exchangeable Image File Format (EXIF) data is removed from all images used so that no location data, for instance, is left.

Message confidentiality is assured with the use of secure functions for KDF, SRG, and encryption. The channel key (C_i) is derived using a KDF such as Argon2 [6, 7]. A unique, random, and securely generated IV (IV_i) is also used for each message.

Integrity is guaranteed by calculating a hash value (H_i) for each message using a

secure HMAC function that receives the encrypted message (E_i) and the current IV (IV_i) as parameters. The hash value will always be unique due to the freshness of the IV_i, even if the message (msg) is the same. The msg and H_i are only exchanged while encrypted, making them unable to be changed outside of the channel without detection.

The availability of the proposed PoC is considered to be partially assured since the user is reliant on multiple, externally controlled online services. The PoC is based on currently existing technology and services; however, an advanced user may create his/her own image-hosting service as an alternative in case the available hosting services change operation procedures or experience a shutdown. Further, while an attacker may not be able to easily delete the image from the chosen image-hosting service, he/she can, however, delete the URL from the text-hosting service if he/she knows the used filename (UUID_i). Because the used online text-sharing service does support Transport Layer Security (TLS), the use of a VPN connection is assumed. If a URL is deleted from the text-hosting service prior to being read by the recipient, the recipient will not even realize that a new message had been sent. We argue that the risk of such behaviour is outweighed by the eventual loss of privacy associated with numbering messages and keeping a record of which ones are read by whom. If the recipient does not read the message, it will remain in the Dontpad and Imgur servers, until it is read. The proposed PoC also allows protection against specific attacks such as replay attacks, Chosen-Ciphertext Attacks (CCAs) [46, 31, 45, 13, 32], and Chosen-Plaintext Attacks (CPAs) [46, 31, 72, 2, 44]. Although an attacker without access to the channels' *password* and *salt* is unable to insert new messages in a channel, the Dolev–Yao intruder model assumes that previous messages can be sent, performing a replay attack. The proposed PoC adopts the per-message use of a securely generated random value that can only be used once (*nonce*). Each user stores previously exchanged *nonce* values, rejecting messages for which this value is repeated. Moreover, the *nonce* is encrypted when exchanged, and, thus, it is assumed that it is impracticable for the attacker to alter the *nonce*, making replay attacks infeasible. In a CCA, the attacker must be able to request the decryption of ciphertext of their own choosing. In a CPA, the attacker must obtain the encryption of plaintexts of their own choosing. The only way to generate a valid ciphertext, or to decrypt it, is by having the correct *password* and *salt*, which are assumed to have been securely exchanged between users. Moreover,

the used KDF and SRG are assumed secure, plus the adoption of a fresh IV for each per message ensures that these attacks do not break the system.

Chapter 8

Conclusions

Currently, mainstream messaging services are provided within social network platforms, and these platforms are able to collect information from users, even when these platforms use end-to-end encryption. The work presented herein proposes a message service to support anonymous and confidential communication without requiring dedicated online servers but exploiting existing ones, including social networks. The current work reviewed the viability of using existing online services to support the proposed messaging service. Steganography was chosen as the technique to enable covert communication, and existing online services were assessed in terms of their support for the use of steganography. Despite most social networks disrupting steganography through image compression, a selected list of online services can be used. A prototype was developed that made use of this list of online services to allow for covert communications.

In future work, other online services could be analyzed to create a list of similarly operating services for photo and text sharing between users. Such a list would increase the levels of anonymity and resilience. This list could extend to other types of social communication, such as forums or other text applications (eg. Slack or Discord). Furthermore, a possible way of ensuring redundancy is by sending the image through multiple online services at the same time with the help of an erasure-coding algorithm. Another possible addition is the use of QR codes as carriers for the secret message, which is then embedded in the image, which makes it easier to overcome possible image compression by online services [23]. Extending chatrooms to groups, allowing for group chats instead of one-on-one communication, is also planned. Further, rolling filenames in the text-hosting

service through seeds may be explored, since this would make it more difficult for third parties to find the filenames being used in the text-sharing services. The prototype could also be developed for other operating systems, such as Android, iOS, Linux or MacOS, thanks to the chosen framework which is focused on cross-platform development.

References

[1] VA Babkin and EP Stroganova. "Evaluation and optimization of virtual private network operation quality". In: *2019 Systems of Signal Synchronization, Generating and Processing in Telecommunications (SYNCHROINFO)*. IEEE. 2019, pp. 1–4.

[2] Gregory V Bard. "The vulnerability of SSL to chosen plaintext attack". In: *Cryptology ePrint Archive* (2004).

[3] Richard Barnes et al. URL: `https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html`.

[4] Mihir Bellare. "New proofs for NMAC and HMAC: Security without collision resistance". In: *Journal of Cryptology* 28.4 (2015), pp. 844–878.

[5] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. "Message authentication using hash functions: The HMAC construction". In: *RSA Laboratories' CryptoBytes* 2.1 (1996), pp. 12–15.

[6] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications". In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. 2016, pp. 292–302. DOI: 10.1109/EuroSP.2016.31.

[7] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. *phc-winner-argon2*. Online, `https://github.com/P-H-C/phc-winner-argon2`.

[8] T Braun et al. "Virtual private network architecture". In: *Charging and Accounting Technology for the Internet (Aug. 1, 1999)(VPNA)* (1999).

[9] S. Budiansky. *Battle of Wits: The Complete Story of Codebreaking in World War II*. Free Press, 2000. ISBN: 9780684859323.

[10] Aniello Castiglione, Bonaventura D'Alessio, and Alfredo De Santis. "Steganography and Secure Communication on Online Social Networks and Online Photo Sharing". In: *2011 International Conference on Broadband and Wireless Computing, Communication and Applications*. 2011, pp. 363–368. DOI: 10.1109/BWCCA.2011.60.

[11] How-Shen Chang. "International data encryption algorithm". In: *jmu. edu, googleusercontent. com, Fall* (2004).

[12] Nicholas Confessore. *Cambridge Analytica and Facebook: The scandal and the fallout so far*. Online, https://www.nytimes.com/2018/04/04/us/politics/cambridge-analytica-scandal-fallout.html. Apr. 2018.

[13] Ronald Cramer and Victor Shoup. "Design and analysis of practical public-key encryption schemes secure against adaptive chosen ciphertext attack". In: *SIAM Journal on Computing* 33.1 (2003), pp. 167–226.

[14] *Cryptography Whitepaper*. Online, https://threema.ch/press-files/2_documentation/cryptography_whitepaper.pdf.

[15] D. Dolev and A. Yao. "On the security of public key protocols". In: *IEEE Transactions on Information Theory* 29.2 (1983), pp. 198–208. DOI: 10.1109/TIT.1983.1056650.

[16] Morris Dworkin et al. *Advanced Encryption Standard (AES)*. en. 2001-11-26 2001. DOI: https://doi.org/10.6028/NIST.FIPS.197.

[17] Kseniia Ermoshina, Francesca Musiani, and Harry Halpin. "End-to-End Encrypted Messaging Protocols: An Overview". In: *Third International Conference, INSCI 2016 - Internet Science*. Ed. by Franco Bagnoli et al. Vol. 9934. Lecture Notes in Computer Science (LNCS). Florence, Italy: Springer, Sept. 2016, pp. 244–254. DOI: 10.1007/978-3-319-45982-0_22. URL: https://hal.inria.fr/hal-01426845.

[18] Paul Joan Ezra et al. "Secured communication using virtual private network (VPN)". In: *Cyber Security and Digital Forensics* (2022), pp. 309–319.

[19] Niels Ferguson, Bruce Schneier, and Tadayoshi Kohno. *Cryptography engineering: design principles and practical applications*. John Wiley & Sons, 2011.

[20] Digital Equality Foundation. *StegoShare (version 1.01)*. Online, `http://stegoshare.sourceforge.net/`.

[21] W.F. Friedman. *Solving german codes in world war I*. A cryptographic series. Aegean Press, 1977.

[22] Wei-Jie Gong et al. "Family E-Chat Group Use Was Associated with Family Wellbeing and Personal Happiness in Hong Kong Adults amidst the COVID-19 Pandemic". In: *International Journal of Environmental Research and Public Health* 18.17 (2021). ISSN: 1660-4601. DOI: `10.3390/ijerph18179139`. URL: `https://www.mdpi.com/1660-4601/18/17/9139`.

[23] Vladimír Hajduk et al. "Image steganography with using QR code and cryptography". In: May 2016. DOI: `10.1109/RADIOELEK.2016.7477370`.

[24] Nagham Hamid et al. "Image steganography techniques: an overview". In: *International Journal of Computer Science and Security (IJCSS)* 6.3 (2012), pp. 168–187.

[25] Dominik Herrmann et al. "Analyzing characteristic host access patterns for re-identification of web user sessions". In: *Nordic Conference on Secure IT Systems*. Springer. 2010, pp. 136–154.

[26] Stefan Hetzl. *Steghide (version 0.5.1)*. Online, `http://steghide.sourceforge.net/`.

[27] Jason Hiney et al. "Using Facebook for Image Steganography". In: *2015 10th International Conference on Availability, Reliability and Security*. 2015, pp. 442–447. DOI: `10.1109/ARES.2015.20`.

[28] Hsiao-Ying Huang and Masooda Bashir. "The onion router: Understanding a privacy enhancing technology community". In: *Proceedings of the Association for Information Science and Technology* 53.1 (2016), pp. 1–10.

[29] Mehdi Hussain and Mureed Hussain. "A survey of image steganography techniques". In: (2013).

[30] Muhammad Iqbal and Imam Riadi. "Analysis of security virtual private network (VPN) using openVPN". In: *International Journal of Cyber-Security and Digital Forensics* 8.1 (2019), pp. 58–65.

[31] Bappaditya Jana et al. "An Overview on Security Issues in Modern Cryptographic Techniques". In: *Proceedings of 3rd International Conference on Internet of Things and Connected Technologies (ICIoTCT)*. 2018, pp. 26–27.

[32] Dingding Jia, Xianhui Lu, and Bao Li. "Constructions secure against receiver selective opening and chosen ciphertext attacks". In: *Cryptographers' Track at the RSA Conference*. Springer. 2017, pp. 417–431.

[33] Neil F Johnson and Stefan Katzenbeisser. "A survey of steganographic techniques". In: *Information hiding*. 2000, pp. 43–78.

[34] Simon Kemp. *The global state of digital in October 2022 - DataReportal – Global Digital insights*. Oct. 2022. URL: https://datareportal.com/reports/digital-2022-october-global-statshot.

[35] Jasleen Kour and Deepankar Verma. "Steganography techniques–A review paper". In: *International Journal of Emerging Research in Management & Technology* 3.5 (2014), pp. 132–135.

[36] Hugo Krawczyk, Mihir Bellare, and Ran Canetti. *HMAC: Keyed-hashing for message authentication*. Tech. rep. 1997.

[37] Published by L. Ceci and Sep 22. *Mobile messaging users worldwide 2025*. Sept. 2022. URL: https://www.statista.com/statistics/483255/number-of-mobile-messaging-users-worldwide/.

[38] Wei Lu et al. "Secure Robust JPEG Steganography Based on AutoEncoder With Adaptive BCH Encoding". In: *IEEE Transactions on Circuits and Systems for Video Technology* 31.7 (2021), pp. 2909–2922. DOI: 10.1109/TCSVT.2020.3027843.

[39] Dennis Luciano and Gordon Prichett. "Cryptology: From Caesar Ciphers to Public-key Cryptosystems". In: *The College Mathematics Journal* 18.1 (1987), pp. 2–17. DOI: 10.1080/07468342.1987.11973000. eprint: https://doi.org/10.1080/

07468342.1987.11973000. URL: https://doi.org/10.1080/07468342.1987.11973000.

[40] Tayana Morkel, Jan HP Eloff, and Martin S Olivier. "An overview of image steganography." In: *ISSA*. Vol. 1. 2. 2005, pp. 1–11.

[41] Y. Nir and A. Langley. *Chacha20 and Poly1305 for IETF protocols*. June 1970. URL: https://www.rfc-editor.org/rfc/rfc8439.

[42] Cosimo Oliboni. *OpenPuff (version 4.01)*. Online, https://embeddedsw.net/OpenPuff_Steganography_Home.html.

[43] Fatih Özkaynak. "Cryptographically secure random number generator with chaotic additional input". In: *Nonlinear Dynamics* 78.3 (2014), pp. 2015–2020.

[44] Yi Qin, Yuhong Wan, and Qiong Gong. "Learning-based chosen-plaintext attack on diffractive-imaging-based encryption scheme". In: *Optics and Lasers in Engineering* 127 (2020), p. 105979.

[45] Charles Rackoff and Daniel R Simon. "Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack". In: *Annual international cryptology conference*. Springer. 1991, pp. 433–444.

[46] B Srinivasa Rao and P Premchand. "A Review on Combined Attacks on Security Systems". In: *Int. J. Appl. Eng. Res. ISSN 0973* 4562 (2018), pp. 16252–16278.

[47] Ricochet Refresh. *Ricochet Refresh*. Online, https://www.ricochetrefresh.net/.

[48] Mennatallah M Sadek, Amal S Khalifa, and Mostafa GM Mostafa. "Video steganography: a comprehensive review". In: *Multimedia tools and applications* 74.17 (2015), pp. 7063–7094.

[49] K Sathya, J Premalatha, and Vani Rajasekar. "Investigation of strength and security of pseudo random number generators". In: *IOP Conference Series: materials Science and Engineering*. Vol. 1055. 1. IOP Publishing. 2021, p. 012076.

[50] B. Schneier. *Applied Cryptography: Protocols, Algorithms and Source Code in C*. Wiley, 2017. ISBN: 9781119439028. URL: https://books.google.pt/books?id=OkOnDwAAQBAJ.

[51] B. Schneier et al. *The Twofish Encryption Algorithm: A 128-Bit Block Cipher*. Wiley Computer Publishing. Wiley, 1999. ISBN: 9780471353812.

[52] *Secure messaging apps comparison*. Online, https://www.securemessagingapps.com/. Nov. 2021.

[53] *Session*. Online, https://getsession.org/.

[54] *Session: A Model for End-To-End Encrypted Conversations With Minimal Metadata Leakage*. Online, https://arxiv.org/pdf/2002.04609.pdf.

[55] *Signal Messenger*. Online, https://signal.org/pt_PT/.

[56] softwarepgs. *softwarepgs/psst-chat: Prototype - DOI*. Version v0.1.1. Nov. 2022. DOI: 10.5281/zenodo.7379263. URL: https://doi.org/10.5281/zenodo.7379263.

[57] Daniela Stanescu, Valentin Stangaciu, and Mircea Stratulat. "Steganography on new generation of mobile phones with image and video processing abilities". In: *2010 International Joint Conference on Computational Cybernetics and Technical Informatics*. 2010, pp. 343–347. DOI: 10.1109/ICCCYB.2010.5491253.

[58] Daniela Stanescu et al. "Steganography in YUV color space". In: *2007 International Workshop on Robotic and Sensors Environments*. IEEE. 2007, pp. 1–4.

[59] *The Double Ratchet Algorithm*. Online, https://signal.org/docs/specifications/doubleratchet/.

[60] *The Sesame Algorithm: Session Management for Asynchronous Message Encryption*. Online, https://signal.org/docs/specifications/sesame/.

[61] *The X3DH Key Agreement Protocol*. Online, https://signal.org/docs/specifications/x3dh/.

[62] *The XEdDSA and VXEdDSA Signature Schemes*. Online, https://signal.org/docs/specifications/xeddsa/.

[63] *Threema*. Online, https://threema.ch/.

[64] Sean Turner and Lily Chen. *Updated security considerations for the MD5 message-digest and the HMAC-MD5 algorithms*. Tech. rep. 2011.

[65] *USA Text Messaging Statistics 2022*. Oct. 2022. URL: https://www.smscomparison.com/mass-text-messaging/2022-statistics/.

[66] Samir Vaidya. *OpenStego (version 0.82)*. Online, https://www.openstego.com/.

[67] Philipp Winter and Stefan Lindskog. "How the Great Firewall of China is Blocking Tor". In: (Apr. 2012).

[68] *Wire*. Online, https://wire.com/.

[69] *Wire Security Whitepaper*. Online, https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf.

[70] Emma Woollacott. *Russia doubles down on censorship with expanded block on tor*. Online, https://www.forbes.com/sites/emmawoollacott/2021/12/09/russia-doubles-down-on-censorship-with-expanded-block-on-tor/?sh=1a81407e19bc. Apr. 2022.

[71] Fei Yu et al. "A survey on true random number generators based on chaos". In: *Discrete Dynamics in Nature and Society* 2019 (2019).

[72] Ning Yu and Kyle Darling. "A low-cost approach to crack python CAPTCHAs using AI-based chosen-plaintext attack". In: *Applied Sciences* 9.10 (2019), p. 2010.

[73] Zhensheng Zhang et al. "An overview of virtual private network (VPN): IP VPN and optical VPN". In: *Photonic network communications* 7.3 (2004), pp. 213–225.

[74] Elżbieta Zielińska, Wojciech Mazurczyk, and Krzysztof Szczypiorski. "Trends in steganography". In: *Communications of the ACM* 57.3 (2014), pp. 86–95.