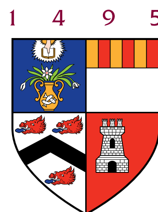
Platform-Agnostic End-to-End Encryption for Modern Instant Messaging Platforms

Mikko Ilmonen
mikko.ilmonen.16@aberdeen.ac.uk

BSc (Hons), Computer Science, University of Aberdeen, 2020

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



Department of Computing Science

2020

Declaration

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

Signed:

Date: 2020

Word Count: 17488

Abstract

This dissertation investigates whether it is possible to perform end-to-end encryption over an arbitrary Instant Messaging Platform (IM-P), placing no implicit trust in such platform itself.

In the current state of the world, people are fragmented across multiple different messaging platforms, alarmingly few of which are completely transparent about the data they collect and the security features they provide. Regardless of whether users trust their platform or not, they can be forced to use them for the simple reason of trying to reach someone they know.

The dissertation proposes this *implicit trust* should not be required in the first place, and users can use additional software to communicate securely with a set of recipients, *even if they do not trust the platform they communicate on*. While this has already been done in the past with PGP encrypted e-mails transmitted over an unsecure medium, it has never been widely successful either due to the difficulty of setup, decline of e-mail as a messaging platform, or more likely a combination of the two.

As the proposed software is open source, the trust is not only shifted from the Instant Messaging Platform to the software, but the user of the software themselves. There is no implicit trust placed onto any code; the proposed software can be fully inspected before any trust is placed on it.

This dissertation offers a fully transparent software solution for the user, binding seamlessly into their platform of choice and automatically encrypting/decrypting messages where appropriate. The Instant Messaging Platform will still be used to transmit ciphertext over to the other person, who then uses similar software to decrypt the messages.

Users will gain the benefit of end-to-end message encryption without additional overhead apart from installing the software, regardless of the trust placed for any Instant Messaging Platform.

Acknowledgements

I would like to thank the 205 Gang for always being there for me.
Having the room there has been of more importance than any degree could ever be.

Thanks to my supervisor Matthew for putting up with my crazy weekly babbling.
Thank you Wamberto, Bruce — you were there to support me, in matters either large or small.
I also want to give special thanks to my tea kettle for not giving up on me.

Yo Keeyan, thanks for the out-of-this-world discussions during these four years.
We still need to finish that Halo map.

Konrad, it's been fun working on random projects and stressing about studies together.
Now when that's over, we can finally relax, GVC, and stress about work instead.

Never change, Smartin. <3

Contents

Abbreviations **8**

1 Introduction **10**
1.1 Motivation . . . 10
1.2 Goals & Non-Goals . . . 11
1.2.1 Goals . . . 11
1.2.2 Non-Goals . . . 11

2 Background **12**
2.1 IM Platforms . . . 12
2.2 Why Encrypt? . . . 12
2.3 Public-Private-key Encryption . . . 13
2.3.1 PPK Scenarios . . . 13
2.4 PGP . . . 14
2.4.1 GPG . . . 15
2.5 Current Usage of PGP . . . 15
2.5.1 Why PGP? . . . 16
2.6 Previous Attempts . . . 16

3 Design & Architecture **18**
3.1 Design . . . 18
3.1.1 Core Requirements . . . 18
3.2 Architecture . . . 19
3.2.1 Morpheus . . . 20
3.2.2 Icelos - Morpheus Client Daemon . . . 20
3.2.3 Encryption on Chat . . . 22
3.2.4 Interfacing with IM-Ps . . . 22
3.2.5 Core Elements . . . 25
3.2.6 Browser Extension Design . . . 26
3.2.7 Recipient Validation . . . 28

4 Implementation **29**
4.1 Development Setup . . . 29
4.2 Morpheus . . . 29

4.2.1 Core Elements 30
4.2.2 Promises 30
4.2.3 Messaging 30
4.2.4 Context Separation 31
4.2.5 Modules 32
4.2.6 More Efficient Querying 33
4.2.7 Element Cloning 34
4.2.8 Message Chunking 35
4.2.9 Message Observer 36
4.3 Icelos 37
4.3.1 HTTP Messaging 37
4.3.2 Docker 38
4.3.3 Service 38
4.4 Interfacing 38
4.4.1 Inject Script 38
4.4.2 Reverse Engineering 39
4.4.3 Mobile 40
4.5 Usability 41
4.5.1 Colours & Clarity 41
4.5.2 Keys 42

5 Evaluation **44**
5.1 Overview 44
5.1.1 Portability 44
5.1.2 Performance 44
5.2 Security 49
5.2.1 Perfect Forward Secrecy 49
5.2.2 Abuse Cases 49
5.3 Testing 53
5.3.1 Component Testing 53
5.3.2 System Testing 53
5.3.3 Manual Testing 53
5.4 Known Issues 54
5.4.1 Store Availability 54
5.4.2 No separation of Own Keys from Recipient Keys 54
5.4.3 Difficulty of Creating Modules 54
5.5 Fulfilment of Requirements 54

6 Discussion **56**
6.1 Achievements 56
6.1.1 Platform-Agnosticism 56
6.1.2 Security 56
6.1.3 Respect for peoples' individual setups 57

6.1.4 Performance 57
6.2 Future Work 58
6.2.1 Usability & Commercialisation 58
6.2.2 Key Exchange 59
6.2.3 Message Protocols 59
6.2.4 Message Formats 59
6.2.5 Combating Metadata 60
6.2.6 Mobile 60
6.3 Learning Opportunities 61

7 Conclusion 62

A User Manual 63
A.1 Installing Morpheus 63
A.1.1 Building the Extension 63
A.1.2 Setting up Icelos 65
A.1.3 Setting up PGP 65
A.1.4 Exchanging Keys Securely 66
A.1.5 Frequently Asked Questions 66

B Maintenance Manual 71
B.1 Software Requirements 71
B.2 Installation 71
B.3 Implementing Modules 71
B.4 Project Tree 73
B.4.1 Root 73
B.4.2 Morpheus 73
B.4.3 Icelos 74

Abbreviations

API Application Programming Interface

ARIA Accessible Rich Internet Applications

CSS Cascading Style Sheets

CSV Comma Separated Value

DOM Data Object Model

DRA Double Ratchet Algorithm

ECC Elliptic Curve Cryptography

GDPR General Data Protection Regulation

GNU GNU's Not UNIX

GPG GNU Privacy Guard

HTML HyperText Markup Language

HTTP HyperText Transfer Protocol

ID Identifier

IM Instant Messaging

IMAP Internet Message Access Protocol

IM-P Instant Messaging Platform — refers to the collective front-end software that provides an Instant Messaging Service for its users.

IM-S Instant Messaging Service — refers to the systems in place that deliver Instant Messages between a set of users.

IRC Internet Relay Chat

JS JavaScript

JSON JavaScript Object Notation

MitM Man-in-the-Middle

MSG Message

NFC Near-Field Communications

OS Operating System

OTR Off-The-Record Messaging

PGP Pretty Good Privacy

POP Post Office Protocol

PPK Public-Private Key

RDF Resource Description Framework

SMS Short Message Service

SSH Secure Shell

SSL Secure Socket Layer

TLS Transport Layer Security

UI User Interface

UNIX The UNIX operating system family

URL Uniform Resource Locator

W3C World Wide Web Consortium

Chapter 1

Introduction

Electronic messaging has changed the way people communicate over long distances. Instant Messaging (IM) is a form of communication that has been widely adopted in the last decades, overthrowing most other means of messaging such as e-mail and SMS.

Perhaps unsurprisingly, we can find a few different providers of Instant Messaging Service (IM-S)s with varying levels of promised and realised security. Notable ones that are widely used include Facebook Messenger, WhatsApp, iMessage and Telegram[1], all of which promise to provide some form of end-to-end encryption for their messages [16, 18].

Since the majority of these IM-Ss are of proprietary nature, there is unfortunately little to no transparency in the security of the systems. Users are left to place their full trust on the company on their messaging app's[2] security. However, this status quo can be circumvented.

1.1 Motivation

There have been various attempts in creating open-source Instant Messaging Platform (IM-P)s to 'replace' the usage of the major proprietary providers. Projects such as Signal, Matrix, Wire[3] and the like take pride in being fully open-sourced and transparent, thus enhancing their security [12, 2, 21]. However, all of them have ultimately failed in having the masses adopt the services.

Even if such a platform was equal in every way to current platforms, users might not switch. One issue behind adopting an alternative could be users' 'fatigue' of yet another IM-P — switching always takes resources [17]. Another is that people will not want to move to a platform with no users. While a single user may feel the need to change, they will be discouraged to do so in their own social network as everyone else is using a different platform [3]. Yet another issue may be the fact that some of the open-source platforms require significant configuration or even running your own server (for maximum security), something which is viewed an infeasible task for an average user [3].

I will not go into much detail why open-source alternatives are not adopted, as there are probably many reasons behind this observed behaviour. However, this does raise the question that sparked motivation for this project — **in the current state of the world, can we explicitly choose not to trust the IM-Ps, yet use them securely?**

---

[1] Respectively: `https://messenger.com`, `https://whatsapp.com`, `https://support.apple.com/explore/messages`, and `https://telegram.org`

[2] Please note that hereinafter the word 'app' is used abbreviating the word 'application', not necessarily referring to a mobile app, even if it is the case most of the applications can run on mobile platforms.

[3] Respectively: `https://signal.org`, `https://matrix.org`, and `https://wire.com`

1.2 Goals & Non-Goals

Next we will briefly go over the goals and non-goals of the project.

1.2.1 Goals

Here we describe the main goals of the project, acting as core principles to shape the software requirements further on.

- Provide a way for a security-concerned user to send messages over an arbitrary IM-P securely.
- Do not enforce the installation of the application on the other users - only one user having the software should be enough to communicate securely.
- Refrain from enforcing maximum security on the user; the encryption/decryption should be something the user can toggle on and off as they like.

1.2.2 Non-Goals

Here we describe non-goals of the project, with a justification why an item is not considered a valuable goal to pursue.

- Provide a way to perform secure key exchange over an insecure IM-P.
  - This should be handled separately (e.g. generation, exchanging and signing of keys offline).
  - The software could expand further to include key exchange but for this project is out of scope[4].
- Combat the metadata exchanged on an IM-P.
  - Metadata is a large problem in cryptography, but is out of the scope for this project[5].
- Provide our own secure IM-P / IM-S.
  - While it is true it would be considered more 'secure' to just tell both the participants to install the latest version of the Signal messenger, it is both out of scope and very much inefficient to force users to do so every time they want to establish a secure communication.
- Create a new encryption mechanism only applicable to IM platforms.
  - Not only has this already been attempted, it is out of scope for this project.
- Interface with IM-P *X* using a specific Application Programming Interface (API) *Y* that they provide.
  - This project is meant to be extremely generally applicable to any IM-P in order to show that general encryption/decryption is possible.
  - Whilst providing an API is great, platform-specific configuration is out of scope for the time scale of this project.

[4]See Section 6.2 for more information on future work on these concerns.
[5]Ditto.

Chapter 2

Background

Despite there being protocols for encrypting online communication (such as Secure Socket Layer (SSL) or Secure Shell (SSH)) that automatically perform key exchange and certificate authentication, not many have been created for personal data transmitted over the Internet. Usually, these protocols have some form of of Public-Private Key (PPK) architecture that utilises asymmetric cryptography. These protocols mostly exist in the transport layer, and little has been done to perform encryption in the application layer.

2.1 IM Platforms

Instant Messaging Platform (IM-P)s are platforms that provide Instant Messaging Service (IM-S)s to users. They are usually connected to a large corporation, and tied to a user account of some kind. The term Instant Messaging (IM) is not clearly defined; nor is it clear how 'instant' it has to be to be considered IM. In this dissertation IM-P refers to any interface that gives the user access to any IM-S. Examples of this include, but are not limited to: WhatsApp, Facebook Messenger, Discord, Signal, Telegram[1].

2.2 Why Encrypt?

Data security, especially when it comes to user data, can be tricky to maintain in large scales. There is significant evidence suggesting that security of user data on a global company scale is an extremely difficult matter to handle. For example, Facebook has had almost yearly data leaks and breaches in the past three years [9, 23]. While this has not included private user conversation logs, it is alarming to say the least.

Apart from 'casual' IM-Ps that do not have a focused goal, there has been an influx of many that revolve around a single goal. Many dating apps provide a platform for users to 'match' and talk in a seemingly private chat setting. Naturally the data exchanged between parties on such apps can be considered quite sensitive, yet for the most major platforms such as Tinder and Grindr[2] there is no information on the security of the app itself — the former has a statement along the lines of 'you'll just have to trust us that we keep our systems secure'[3] while the latter has none.

So far, we have assumed the companies have good intentions regarding user data and security.

[1]Respectively: `https://whatsapp.com`, `https://messenger.com`, `https://discordapp.com`, `https://signal.org`, and `https://telegram.org`

[2]Respectively: `https://tinder.com` and `https://grindr.com`

[3]At the time of writing, Tinder's main website returns a 404 Not Found error when clicking on the 'Security' link in the site navigation menu, but it can be found through some additional searching.

Unfortunately there have been cases such as Cambridge Analytica [20], where personal data has been sold and processed with questionable or zero user consent.

Perhaps a rather ominous reason for end-to-end encryption is the concern of mass surveillance by governments or national entities. There is evidence of intelligence agencies actively pursuing the removal of end-to-end encryption [15, 19] or the demand of 'back doors' [11] under the reasoning of preventing terrorism or a variety of other criminal activities. Without going deep into politics, I acknowledge that this is a faceted issue — but an issue nevertheless.

All in all, there exists an implicit trust model of IM-Ps that users are being forced to use because of selection bias and lack of technical know-how. Surveillance is either taken for granted, ignored, or accepted as too difficult to circumvent. Many users do not understand the implications of their communications or personal data being collected, processed and stored for prolonged periods of time. Some people are either not aware, not interested or simply in ignorance of these security implications. It is an unfortunate status quo that a regular person might neglect for a variety of reasons, but a security-oriented person might be concerned of.

The problem lies partly in the fact that these IM-Ps never did offer an open protocol to extend the service on the users' side; the services were provided to users as-is. Marketing speech of '*very secure* end-to-end encryption' and 'secret *many-thousand-bit* keys' has lulled many into believing that security is not a concern of the individual; rather a service provided by a large conglomerate. While this no doubt has worked for the masses to adopt these platforms, there is a constant risk involved, and more importantly the notion of privacy gets twisted.

2.3 Public-Private-key Encryption

Public-Private Key (PPK) is a scheme where agents generate an asymmetric key pair. One of the keys is kept public and is used to encrypt messages. These messages can only be decrypted using the corresponding private key, which is kept hidden at a safe place. This ensures that messages meant for person X (i.e. encrypted using their public key) can only be decrypted using person X's private key — in other words, person X themselves.

In order to establish a secure connection between two agents where both can encrypt and decrypt messages, two key pairs are required, and so on. Here we will briefly discuss different PPK arrangements that are commonly seen, and discuss how they translate into IM.

2.3.1 PPK Scenarios

2.3.1.1 One-on-one Conversation

The simplest form of conversation is between two people. Here there are two sets of asymmetric keys, one for each person (See Figure 2.1). Both Bob and Alice share their **public keys** to everyone who might want to communicate with them.

Alice uses Bob's **public key** to encrypt a secret message to Bob. The message is then transmitted over any (secure or unsecure) channel to Bob. Bob can then decrypt the message using their **private key**.

2.3.1.2 Group Conversation

A slightly more complex scenario is where there are multiple recipients to a single message. This is the case of 'Group Chats' on IM-Ps. Here all recipients have shared their **public keys** with

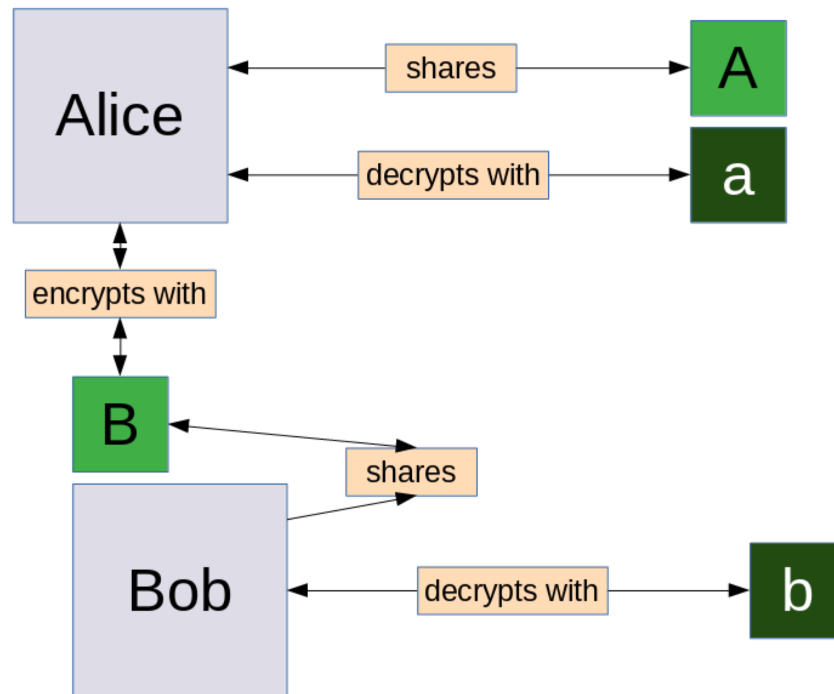


Figure 2.1: 1-1 conversation using PPK. An uppercase letter stands for a public key, a lowercase letter stands for a private key.

Alice, who then encrypts a message to all of those keys. The message will be encrypted in such a way that **any** of the corresponding **private keys** can decrypt the message.

It is worth noting here that a chat 'group' could contain more people than the encrypted message is encrypted for. This is no different from passing the message over brokers such as mail servers — an *encrypted message*'s contents are not directly compromised simply because it reached an audience it was not intended for. Moreover, this enables us to send a message to a majority of our intended recipients even if not all of the recipients are known in our keychain.

2.3.1.3 More complex setups

This brings us to the next point, which perhaps explains why key setups still have not been standardised. The possibilities of key setups are endless, and certainly not limited to only one-on-one or group conversations. For example, if Chris in Figure 2.2 has recently rotated their keys, and Alice has encrypted a message with their old public key, Chris might not be able to decrypt the message. They can, however, ask anyone in this web of trust, to re-send the message with Chris's *new public key* (that has been securely transmitted to everyone). There is no protocol to hold all of the possibilities in key setups; they are inherently dynamic. A design challenge for this software was to accommodate most of these situations by making the design as flexible as possible.

2.4 PGP

Pretty Good Privacy (PGP)[4] is a widely used standard that defines encryption and signing services. This standard can be implemented in various languages, and as such implementations in different languages exist.

[4] Hereinafter PGP (protocol) and OpenPGP (standard) are used interchangeably.

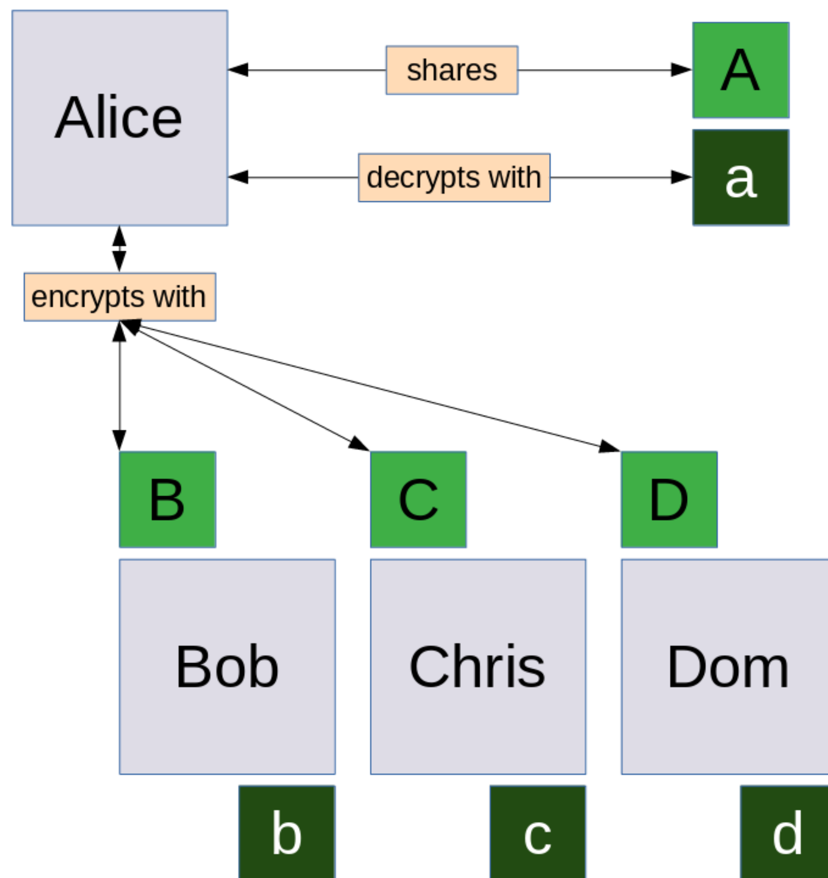


Figure 2.2: Group conversation using PPK. An uppercase letter stands for a public key, a lowercase letter stands for a private key.

2.4.1 GPG

Quite confusingly, GNU Privacy Guard (GPG) is the GNU[5] Free Software implementation of PGP. It supports many different encryption schemes utilising Elliptic Curve Cryptography (ECC), and the Curve25519 encryption[6] that remains one of the most popular schemes used in the current day [1].

There are other standards that provide encryption schemes and ECC. These include, but aren't limited to: CryptoCat, Peerio, Tor, Wire and GNUNet [6, 8, 7]. Many of these however are bundled with additional software not directly related to encryption and decryption, where GPG is the most widely used software suite for encryption, decryption and signing services. It also provides a command-line, graphical and socket/agent interface for both human-process and process-process communication, making it perfect for a portable software solution for encryption and decryption.

2.5 Current Usage of PGP

While PGP e-mail encryption schemes have been around for quite a while, these never got widely adopted in public. Making matters worse, some service providers have turned away from using

[5] https://www.gnu.org/home.en.html
[6] https://wiki.gnupg.org/ECC

open e-mail protocols such as Internet Message Access Protocol (IMAP) or Post Office Protocol (POP), leaving users tied to a browser interface with no means of encrypting or decrypting messages.

The situation is even more dire when it comes to IM; no truly open standard ever existed following the explosion of popular IM-Ps such as Facebook or WhatsApp. One might argue that Internet Relay Chat (IRC) is this needed open standard, but even this has fragmented support[7] and has various usability issues for the average (non-technical) end-user, including but not limited to having to keep IRC always running (or running your own server), configuring a client and choosing a network to join in, talking with some services by the use of text commands — all of which can be intuitive for a programmer but not anyone else.

There is a group effort to create a 'network of IM networks' using an IRC extension called BitlBee[8], to unify every IM-S under the IRC protocol. One could perhaps encrypt all of their communication using this method before it even leaves the client device via IRC, and send it to the respective recipients over BitlBee as encrypted messages. Even after this tremendous technical setup, there is a problem of making users adopt the same system on the other side — after all, if there is not a way *for someone else **not** using the system* to decrypt an encrypted message *when it appears on their IM-P*, then what's the point?

2.5.1 Why PGP?

PGP was chosen as the encryption/decryption mechanism as this is already widely used in e-mail messaging and has been proven quite effective. As the principle of IM is generally the same as e-mail, we can quickly build upon systems provided by PGP. This reinforces the security of the system as a new PPK encryption/decryption system is not developed; rather an existing, industry-proven one is used.

Interfacing with PGP is easier to run a security audit on, and we avoid all of the pitfalls of designing and implementing our own cryptographic algorithm. Furthermore, users that already have an existing PGP setup can enjoy the software out-of-the-box, as PGP is already configured and will handle encryption/decryption for them. This is useful for the demographic of people who are already using PGP for e-mail or file encryption, for example.

2.6 Previous Attempts

There have been some attempts in creating a protocol for secure IM over an arbitrary line of communication.

For example, Off-The-Record Messaging (OTR) is a protocol intended for encryption and authentication on IM conversations. However, OTR does not at the current moment support multiple user group chats or binary transport such as images or audio [22].

Some effort has also been made in using PGP on some web services. For example, there exist a few browser extensions that enable encryption of messages in web interfaces such as the Gmail

[7] `https://en.wikipedia.org/wiki/Comparison_of_instant_messaging_protocols` and `https://en.wikipedia.org/wiki/Internet_Relay_Chat#History`

[8] https://www.bitlbee.org/main.php/news.r.html

web interface[9][10][11]. Not all of these extensions are open-sourced, and they require access to Gmail accounts, etc. No generalised approach seems to have been created yet for a generic platform, be it instant or not-so-instant messaging.

In this dissertation, we will discuss an alternative, perhaps more straightforward approach to managing secure communication using PGP Public-Private Keys over current day Instant Messaging Platforms.

[9] https://www.mailvelope.com/
[10] https://www.streak.com/securegmail
[11] https://cryptup.org/

Chapter 3

Design & Architecture

Next, we will delve deeper into the design process, explaining the requirements and further solidifying the concept of *platform-agnosticism* in the case of this software.

3.1 Design

One of the main design concerns for the software was to keep the application complexity as minimal as possible, but interfacing with existing applications as much as possible. Additionally, a smaller and simpler project is easier to review and trust for the concerned user. It makes the software fully transparent and creates a zero-trust situation if employed correctly.

This clarifies the purpose of the application as a whole, makes sure the software remains in line with the goals and non-goals of the project.

3.1.1 Core Requirements

3.1.1.1 Functional Requirements

- Secure encryption of messages with PGP on an IM-P
  - Ability to turn off encryption of messages at will
- Secure decryption of messages with PGP on an IM-P
  - Ability to turn off decryption of messages at will
- Detection of recipient and their key using the name on an IM-P
  - Ability to change the recipient's name manually if it is defective
- Detection of errors when the recipient name is defective, either:
  - if the name is ambiguous
  - if the name cannot be found (i.e. when using a nickname)
- Interfacing with at least three different IM-Ps

3.1.1.2 Non-Functional Requirements

- Encryption/decryption should happen in a speed that is considered 'almost instantaneous' (< 500ms).
- Should use open-source software for security reasons

  - People can analyse and improve any security vulnerabilities
  - People can audit for themselves that the software is not malicious

- Should keep platform-specific code at a minimum in order to improve code re-usability
- Should allow users to interface (send encrypted messages) with their IM-P of choice

3.2 Architecture

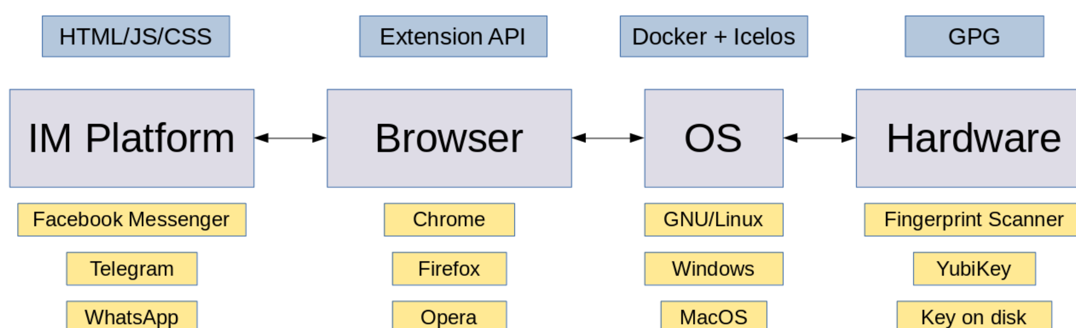


Figure 3.1: Differences in setups and how the software was designed to work on all of the combinations of these scenarios. The yellow labels indicate parts that can change depending on a single user's configuration, and the blue labels tell the common elements that are leveraged to produce modular and interchangeable code.

In order to achieve the most platform-agnostic approach, the software architecture needed to be designed to be as modular and extensible as possible. Targeting multiple platforms is not an easy task. Any combination of IM-P, browser, Operating System (OS), and hardware should be supported. Simply creating static code for all of the combinations is entirely infeasible, so an alternate approach was devised. This led to partitioning the design into four main categories, as seen in Figure 3.1. In this design, any of the categories can change based on user setup, and the software should still be usable as before. This leads to a modular design that is easy to change and update to adopt more categories as time passes.

The choice of programming languages was also heavily influenced by portability. JavaScript (JS) was chosen for the browser environment (IM-P and Browser in Figure 3.1) because of its high availability across modern browsers, and many websites written with JS. The extension was written in ECMAScript 2017 (ES8) which is supported by most browsers in 2020[1].

The back-end code was written in Go, which is an open-source language designed to be portable. Go was chosen because it is fast to write, strongly typed and memory-safe, and compiles to most used platforms. It also has parallelism features which make scaling extremely easy, and handles UTF-8 encoding natively. By default, Go also statically links all its binaries, this can of course be disabled. However, for maximum portability, statically linking the libraries to the binary is beneficial here, mainly because we are only looking at a single binary.

Dealing with security-critical applications, I did not want to deal with concerns of memory management, for example, when transporting plaintext for encryption to GPG. This is handled by Go's practical scoping and garbage handling.

[1] `https://kangax.github.io/compat-table/es2016plus/`

3.2.1 Morpheus

The software that provides encryption/decryption for modern IM-Ps is hereinafter referred to as *Morpheus*. It relates to the movie Matrix, where a character called Morpheus gives a choice to the main character — to either expand their reality to face the harsh truth, or to continue to live in the old world in blissful ignorance[2].

At its core, Morpheus has a simple design. The general idea is to act as a 'barrier' between the user and the IM-P, encrypting and decrypting messages seamlessly as the user sends them and receives them respectively (See Figure 3.2).

The user still chooses to use the IM-P, but only for message delivery. Any message contents are encrypted for transmission and can be decrypted by the intended recipient only. This means that data the IM-P interacts with is always encrypted.

As Morpheus is designed to 'extend' the functionality of regular IM-Ps, it is possible to only manipulate the input and output of the IM-P without touching any of the application code itself.

This approach has many positive indications in terms of data security and privacy. For one, the messages cannot be read without the private key that the intended recipient has in their possession. Any Man-in-the-Middle (MitM)-attack would intercept encrypted messages that are useless. Secondly, any data breach that manages to get access to the conversation history will again yield a set of useless encrypted messages.

The data retention time is brought down from s to $min(s,k)$, where s is the realised data retention (e.g. regulated by GDPR, prone to mistakes given a globally deployed application, ultimately infinite given a data breach), and k is the rotation frequency of one's encryption key pair[3].

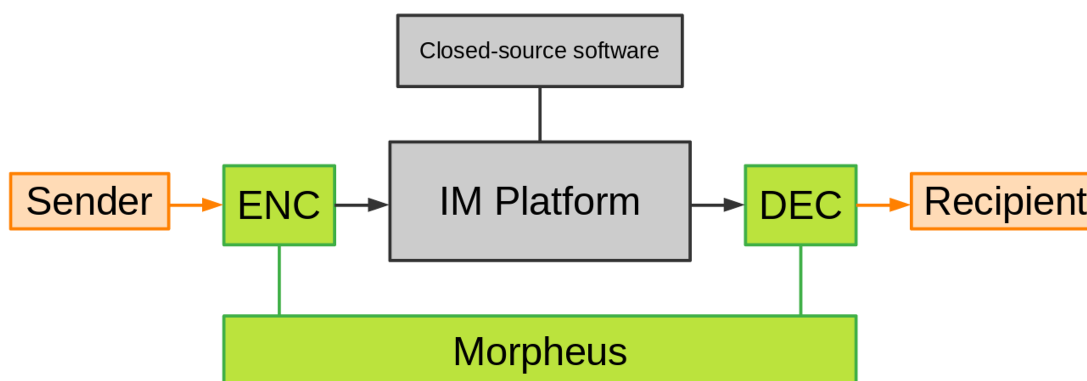


Figure 3.2: General architecture of Morpheus. The software acts as a second layer providing its own end-to-end encryption, regardless of whether the IM-P has one or not.

3.2.2 Icelos - Morpheus Client Daemon

Browser extensions are sandboxed and have no information over any applications or processes running on the machine. They also do not have access to regular OS resources such as a filesystem

[2]This falls in line with one of the core design goals — we should not enforce security to the user. They can choose to 'be secure' or not to do so. After all, the status quo is the latter.

[3]It is worth noting here that the rotation frequency of PGP encryption keys is traditionally quite infrequent. See Key Exchange in Section 6.2 for alternatives.

or execution capabilities. This would make it extremely difficult to communicate with GPG, even if it were running on the client machine. One of the very few pathways of inter-process communication is using listening HyperText Transfer Protocol (HTTP) ports on the local host to process requests sent using JavaScript's own HTTP libraries. This can then further be encrypted using TLS.

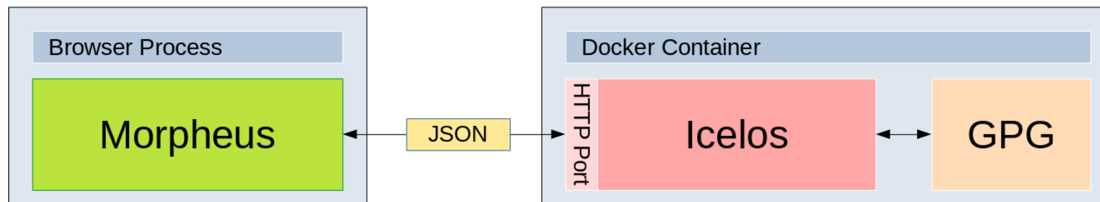


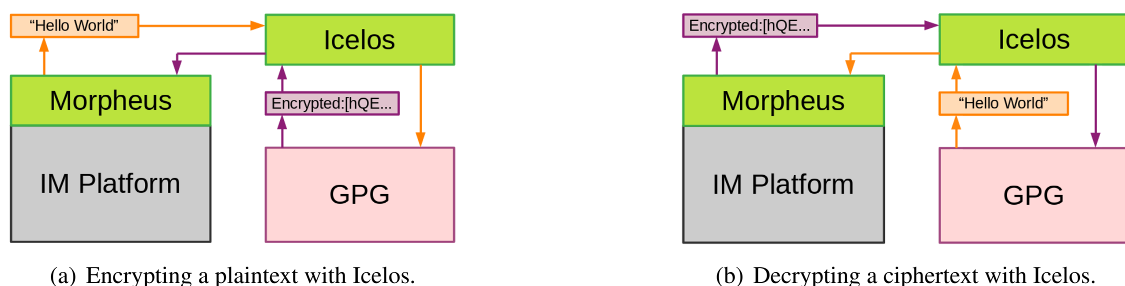
Figure 3.3: Architecture of Icelos. The inter-process communication between Morpheus and Icelos can be handled using JSON on an HTTP port, and optionally encrypted with Transport Layer Security (TLS).

Unfortunately, GPG does not have an HTTP endpoint to communicate with, so a message broker is needed. This broker is called *Icelos*; it interacts with Morpheus using the JavaScript Object Notation (JSON) format that is transported over HTTP requests, and subsequently communicates directly with the GPG UNIX process (See Figure 3.3).

There exist PGP implementations for entirely JavaScript, meaning it could have been possible not to use an extra piece of software such as Icelos for encryption and decryption. However, due to the way browser extensions are isolated from the rest of the machine, this would have led to an awkward situation of asking the user to 'upload' their private key into the extension. While this could technically be handled securely, in an internet or web context, it feels very wrong to ask people to move their private keys anywhere, let alone *upload them* somewhere.

Additionally, using an implementation running entirely in JS, we would lose all hardware support for authentication and encryption methods described in the last column of Figure 3.1. For example, the support across browsers of communicating with hardware keys (such as YubiKeys) is still sketchy in 2020.

Using a separate message broker allows us to utilise the users' own GPG setups fully. Encryption and decryption work seamlessly using Icelos as a broker between the browser and GPG, as seen in Figures 3.4(a) and 3.4(b). The feature set is restricted only by GPG, not by any implementation of OpenPGP in JS, as we would have to simulate GPG to reach full feature parity. For



(a) Encrypting a plaintext with Icelos.

(b) Decrypting a ciphertext with Icelos.

Figure 3.4: Icelos Architecture in relation to Morpheus Browser Extension.

example, if a user has set up a key rotation system, this works out of the box with Icelos as it lets GPG to choose the keys to decrypt with, and encrypts with only active (non-revoked) keys. Any user setup that allows for GPG encryption/decryption already on their client machine will work with Icelos.

Implementations such as OpenPGP.js[4] could be used as a backup for encryption; users would only need to upload their recipient's public key in order to encrypt messages to others without a broker such as Icelos. However, this is such an extreme situation that it was decided not to be implemented.

Icelos can also run on Docker for people that do not have an active GPG setup. This allows people to easily mount a private key directory for key generation and the retaining of recipient keys, and run Icelos and GPG in the same container on Docker. The same setup can also be used to generate the browser extension on the local machine in order to make compilation easier.

Docker enables Morpheus to be used on any major OS that can run Docker, including but not limited to Windows, Mac and Linux. The only requirements are a modern browser that can support extensions, and an open HTTP port to establish communication between the extension and the docker container.

3.2.3 Encryption on Chat

As the encryption targets a generic chat platform, we need to be careful with the encryption format. As encryption happens on the binary level, its output is often garbled binary as well. Sending binary characters on an IM-P is not reliable, as it may contain non-printable characters and may be formatted in strange ways, such as some sequences of characters converted to emoji.

Additionally, IM-Ps can place arbitrary restrictions on the length of the transmitted messages. This means that when receiving messages, the message may consist of multiple parts. The encryption and decryption algorithms should take these details into consideration.

The PGP armoured Base64 format was chosen because of its simplicity. Messages are encrypted into character sequences containing characters `A-Z`, `a-z`, `0-9` and a few special characters such as `+` and `/`.

3.2.4 Interfacing with IM-Ps

Interestingly, many companies with Instant Messaging Services do provide some form of a web IM-P to communicate with users, accessible using a modern browser. These web services are easier to interface with compared to a fully closed-source binary application. This is because they are all structured using HTML, with a JavaScript runtime that the browser executes.

Any modern application uses a tree-like structure for creating and manipulating user interfaces. This is true for web applications as well — the element structure is stored within the Data Object Model (DOM) and accessible within JavaScript. For example, the simple interface structure shown in Figure 3.5 could be described by the following code block in HTML:

[4] `https://github.com/openpgpjs/openpgpjs`

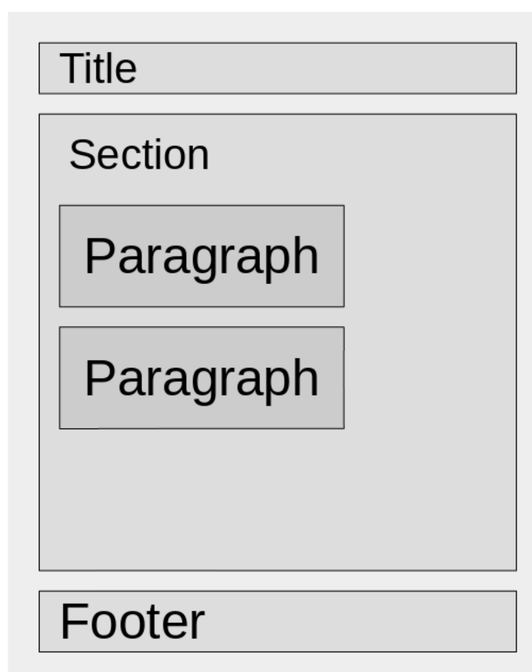
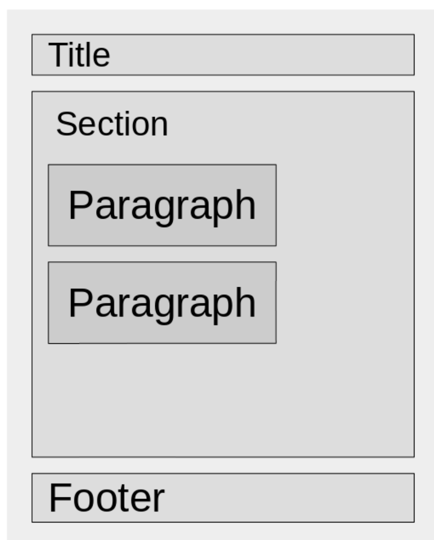


Figure 3.5: Simple DOM Structure.

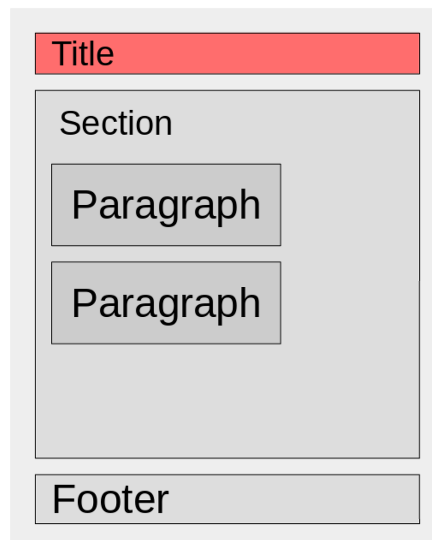
```
<html>
        <body>
                <header>Title </header>
                <section>
                        Section
                        <p>Paragraph </p>
                        <p>Paragraph </p>
                </section>
                <footer>Footer </footer>
        </body>
</html>
```

The structure of an HTML document is usually manipulated in the form of ‘queries’ — these are simple hierarchical statements that tend to return a single object or multiple of them. This allows for building modular applications and only focusing on manipulating content that is needed to be touched. For example, if we wanted to manipulate the header, our query might be: `header`, `body > header` or even `html > body > header`. It all depends on the amount of accuracy the programmer wants (if a document would somehow have two bodies, `body > header` would match both). These queries would give us a reference to the header object, like in Figure 3.6(b).

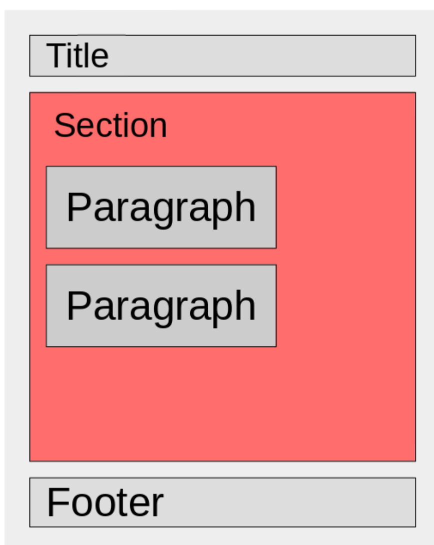
An arbitrarily complex querying rule will still return the same elements even if some of the complete DOM layout changes. The querying rules allow for many more complex selections and can be extremely handy when selecting, for example, all the messages in an application (see Figures 3.6(c) and 3.6(d)).



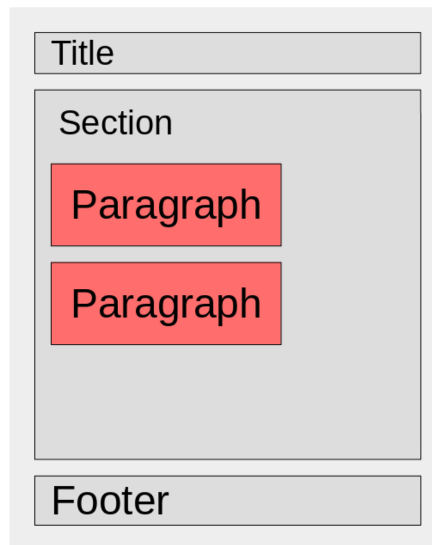
(a) A simple DOM structure.



(b) The element in red is returned when querying for `title`, `body > title` or `html > body > title`.

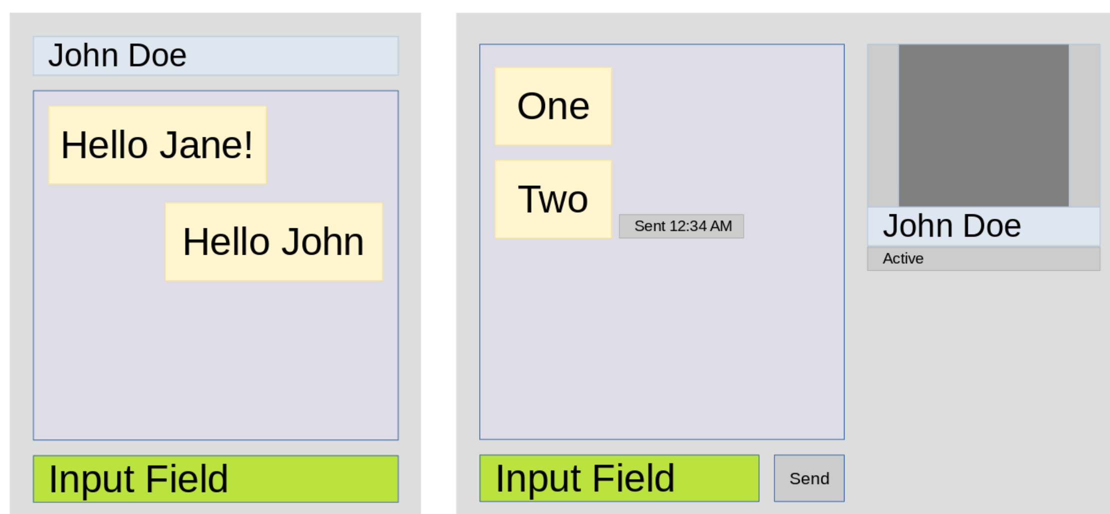


(c) The element in red is returned when querying for `section` or `title + section` or `body > section`.



(d) A query such as `p` or `section > p` will return all of the paragraphs, or the first one if only looking for one element. Note that `section > p` would only return paragraphs inside the section element.

Figure 3.6: DOM element structure and querying.



(a) Simple messaging layout, containing all the 'Core Elements'.

(b) Complex messaging layout. Note how, despite the increased complexity, the Core Elements still exist. This is true regardless of IM-P (for most of IM-S providers).

Figure 3.7: Element structure between IM-Ps. The simplified versions of Telegram Messenger and Facebook Messenger are presented on the left and right, respectively.

3.2.5 Core Elements

Another interesting observation to make is that all IM-Ps follow a finite set of common rules when it comes to setting the layout. By studying and comparing the layout of two arbitrary instant messaging apps (Figure 3.7) we can see that some 'Core Elements' remain the same regardless of the overall layout.

Below is a short list of the most essential Core Elements in Morpheus' design. For implementation details see Implementation in Chapter 4.

3.2.5.1 Bind Input

The Bind Input is an element used to write a message to the recipient, for example, the 'Input Field' element in Figure 3.7. Morpheus needs to 'bind' itself into this input for encryption, hence the name. The input is cloned, and all of the original functionality relating to the IM-P is stripped away; the user will always type into the Morpheus's input field.

Before sending the message, Morpheus encrypts the plaintext and pass it to the Encrypted Data element (usually the same input, see Section 3.2.5.2). This ensures that the web application will not know the plaintext, and only deals with the ciphertext. The unencrypted input is then programmatically sent input events to simulate the user typing the ciphertext and sending it to the recipient. Because the secure input does not have any events attached to it, this also successfully reduces the amount of metadata sent by the user in the form of any typing patterns or simply information *when* the user is typing.

3.2.5.2 Encrypted Data

Sometimes the encrypted message is not stored in the input that wrote it. This element stores the encrypted message before firing the Bind Input's send action. Especially in the case of websites that use custom JavaScript frameworks, the element containing the data can be different from the

element that the user types into.

3.2.5.3 Message Element

The Message Element is a query which returns all of the messages in a web page, for example, the beige messages ‘One’ and ‘Two’ in Figure 3.7(b). The structure allows this message element to be as complicated as needed — Morpheus only needs to decrypt any encrypted text inside this element.

3.2.5.4 Message Feed

The Message Feed is an element which contains Message Elements, for example, the blue containers above the Input Fields in Figure 3.7. This Message Feed is *listened on* for any changes. Listening allows for the automatic detection of new encrypted messages that can then be decrypted accordingly.

3.2.5.5 Message Text

Usually the Message Text and the Message Element are the same. Sometimes there are cases that the Message Element is so complicated that we need an additional query to find the text within. This also affects which element gets its contents replaced (when displaying decrypted plaintext in place of the encrypted message) if the element is more complicated.

3.2.5.6 Recipient Hint

The Recipient Hint element will contain text that determines (a portion of) the PGP recipient string for Morpheus. Usually this is the element that contains the recipient name, for example, the light blue element that contains the name ‘John Doe’ in Figure 3.7. This recipient string can be modified in case the user has overridden the keys for this name. This hint will be used to guess the recipient keys — if keys are not found, the user needs to input them manually.

3.2.5.7 Resetter

Some websites contain JavaScript to update the DOM without reloading the underlying page. It is practically impossible to detect these changes — the Resetter element will manually reset Morpheus when they are interacted with. For example, a Resetter element in a chat application would be clicking on a different ‘chat’. This is expected to change the contents of the Message Feed, Recipient Hint, and other Core Elements. It is important to note that the DOM elements can remain the same — only their contents change. When this happens, Morpheus needs to refresh its data, and this is done via the Resetter elements.

3.2.6 Browser Extension Design

Naturally, the web extension works the same way across all modern browsers that support web extensions. This is ensured by the WebExtensions API[5], which is maintained by the World Wide Web Consortium (W3C)[6]. The draft is not yet completed, which means the Web Extensions API has a few interesting quirks to consider, especially when developing for multiple different browsers. You can read more about these in Section 4.2.3.

The browser extension consists of various parts working together. Browser extensions usually have multiple different scripts with different scopes and privacy. The extension by default does not have access to any of the webpages that the user visits.

[5]`https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions`
[6]`https://www.w3.org`

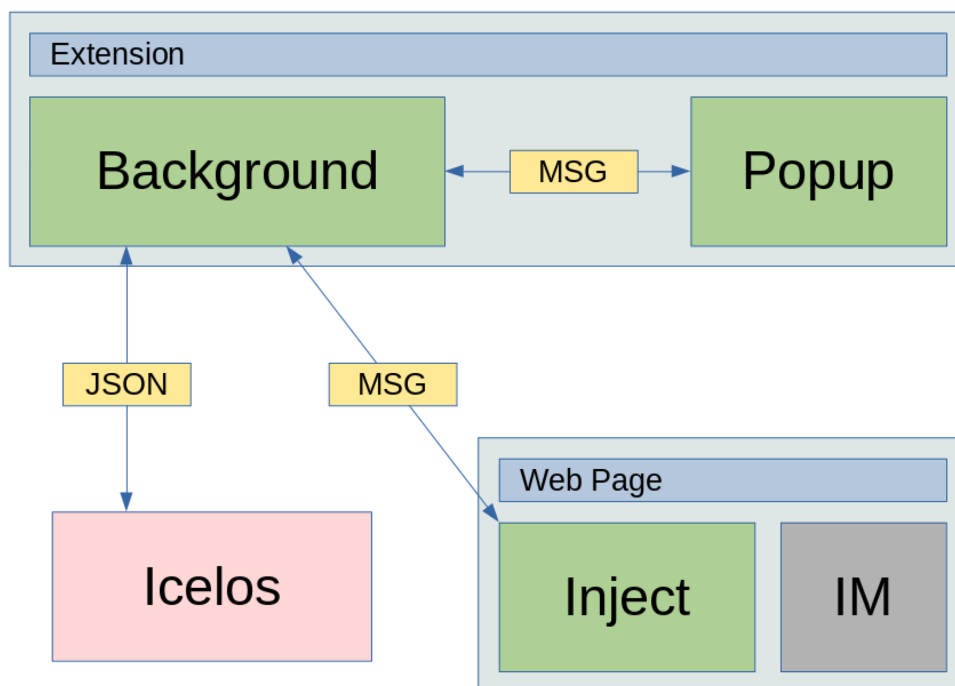


Figure 3.8: Architecture of the Morpheus browser extension.

However, it can explicitly request this and inject another script onto the web page. This script is called an *injection script* or a *content script*, depending on the way it gets injected on the web page. In Morpheus' case, it will hereinafter be referred to as the **Inject Script**.

There are three main scripts that the Morpheus Browser Extension contains: the Background Script, Popup Script and Inject Script (See Figure 3.8). These moving parts communicate together with messages that behave very similarly to JavaScript Object Notation (JSON)[7]. Below we will talk briefly about each.

3.2.6.1 Background Script

As the name suggests, the Background Script runs in the background when the user is browsing the Internet. The Background Script acts as the primary source of truth for the rest of the extension. It is wholly isolated, running in its own container, and can only serve as a message broker between other scripts.

The Background Script is in charge of saving session data and brokering requests of encryption and decryption to Icelos[8] and back. It keeps information about different tabs and the previous recipients for each request, so that encryption can happen as soon as the page is navigated to.

3.2.6.2 Popup Script

The Popup Script handles all the logic that relates to the visible icon of the extension. Due to the way browser extensions are designed, clicking of the icon will grant Morpheus special access to the current webpage that the user is looking at. Any time the icon is clicked, Morpheus will attempt to inject the Inject Script onto the website. Without this access, the content script could

[7]JSON is a data-interchange format designed to be both human- and machine-readable. `https://www.json.org/`
[8]Explained in Section 3.2.2.

not be injected onto the web page, and the extension would be completely isolated from the rest of the webpage that the user is browsing.

This can be overcome by using a special permission for the browser extension where Morpheus is active at all times. Naturally, this can be a concern to some users, so this can be disabled when creating the extension.

The Popup Script also handles all the promise chains for updating user keys, and changing settings of the extension. Any setting changes will be propagated back to the Background script and will take effect immediately.

3.2.6.3 Inject Script

The Inject Script is a script that is added on top of the rest of the DOM in any life cycle of the website. It is the only script that can interact with the DOM of the website it is on, and still has limited access to the rest of the scripts on this website. The Inject Script takes care of displaying Morpheus-specific elements, sending messages for encryption/decryption and dealing with Core Elements.

It has no concept of PGP, and will only be told whether the recipient is 'Verified' (success) or 'Not Verified' (failure) when dealing with recipient verification. This ensures no information about the user's PGP set up is leaking to the web app.

3.2.7 Recipient Validation

One of the perhaps intuitive but technically non-trivial tasks is to choose the right keys for each conversation. A recipient of a message in PGP terms is the person this message is encrypted for. The message does not contain any information on who to *deliver it for*; rather the delivery is done by some other system entirely. This separation causes a problem; how do we choose which keys to encrypt for, given we are talking with a specific person on the online platform?

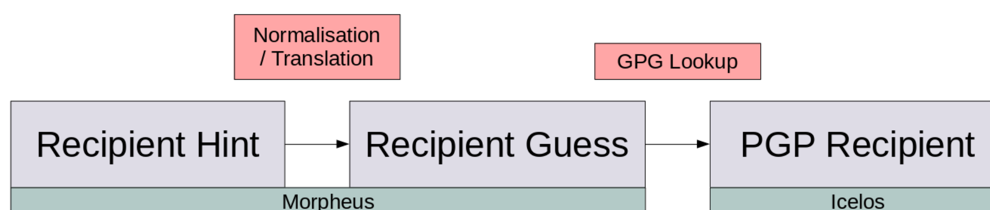


Figure 3.9: Morpheus recipient validation.

Fortunately, PGP keys are always embedded with user information such as first name, last name, and occasionally other details such as e-mail address or a comment. These keys can be queried from GPG using any substring of the user information.

Recipient validation is done via simple steps. First, the recipient hint is normalised using simple transformation rules into a string that can be used to query GPG[9]. Next, this recipient guess is used to query GPG for the recipient. The keys for this recipient are retrieved and saved in the extension for encryption. If the query does not yield recipients, or the query yields an ambiguous result, Morpheus will display an error next to the recipient string and an invalid recipient.

[9]This also includes any user-defined transformations, for example, if the user is talking with 'Cathie' (nickname) but the key is named 'Catherine' (full name).

Chapter 4

Implementation

Knowing the design is one thing; implementing it is another. Unfortunately often with software projects, only after you implement software you realise *how you should have implemented it*.

In this chapter, we delve deeper into how the implementation of the system designed in Chapter 3 happened, and reflect on how it went. Overall the design of the system did not differ too much from the realised implementation, but there were some faced difficulties and changes of approach.

4.1 Development Setup

Large portions of the project were developed using modern development tools like version control and a personal IDE setup. The version control was handled using Git, with a remote repository on GitLab. Every commit was tested before pushing into the repository to ensure that a working version remained on the master branch. I also kept separate backups on another Git repository just in case.

Linux was used as the main OS for development, as this is what I'm most comfortable developing on. I did initially want to test on more platforms such as MacOS and Windows, but unfortunately given the circumstances had to drop those plans. In theory, the software *should* work on all platforms. However, this has not been tested yet (See section 6.1.3).

The IDE setup was a combination of UNIX tools as well as the code editor Vim, which was used to write all of the code. Vim supports features such as spellchecking, syntax highlighting, autocompletion, macros, unlimited buffers and a set of handy commands to speed up development dramatically.

In the Project Planning stage, there was a clear cut between design and implementation, but most of the implementation phase was done using an **agile approach**. Meetings with the supervisor were held weekly, and during this time, the previous week's goals were ticked off and new week's goals gathered.

This agile approach ensured a quick feedback loop and development cycle. Any concerns and issues could be addressed quickly before working too much on the wrong premise. Overall, even though implementation differs slightly from the design, much of the design elements persisted throughout, giving confidence that the design was sound from the start.

4.2 Morpheus

This section will first discuss the implementation details of the Morpheus Browser Extension, then Icelos, and then discuss how they work together. The Extension is written in JavaScript and has

layout and styling defined using HTML5 and CSS.

4.2.1 Core Elements

Like described earlier, the concept of Core Elements was quite fuzzy in the design phase. Defining Core Elements was a challenging task from the software design standpoint, as so many different layouts of IM-Ps are to be targeted.

Fortunately, during implementation, the concept solidified. Often the addition of one IM-P created the realisation that either the current set of Core Elements was not entirely sufficient, or was a simplification that needed to be further generalised. Eventually, the set was diverse enough to be able to add more modules without further modifications to the generic code.

An example of this is the 'Input Field' element that was assumed to both listen to user input events and contain the input data, but it turns out some web interfaces have a separate element to hold the data of the input field[1]. Changing this retroactively was not impossible, but did bring the design of a 'Bind Input' back to the drawing board a few times throughout the implementation process.

4.2.2 Promises

The asynchronous nature of these message chains prompted the implementation to be done with Promises[2]: these are proxies for the storage of asynchronous values within JavaScript.

A promise has three states: it can either be pending, fulfilled or rejected. The promise is returned synchronously as an object, but will contain code to execute in case it gets fulfilled or rejected. This allows for waiting for a specific event 'asynchronously', even if the code is not necessarily such.

Any pending promises are sent when, for example, web requests are waiting for a response. This way, we can decrypt multiple messages at the same time. The extension can process and queue all of those messages in at the same time using promises, thus ensuring that all of them will either be fulfilled (success) or rejected (error) in the future.

Similarly, error scenarios are handled using promises. It is useful to think that any promise that emerges can also fail in an unexpected way; thus, the failure state needs to be clearly defined. This can consist of displaying an error message (for example when decrypting encrypted messages), retrying the request, 'bubbling' the promise upwards, or a similar action.

4.2.3 Messaging

The messaging system of Morpheus is divided across the different scripts, and is handled using Promises.

The Web Extension API is not quite finished, and thus browsers use slightly different non-conforming approaches implementation-wise. This makes for interesting quirks when developing an extension for multiple browsers at once. For example, in the Chrome browser, messages are still handled using callbacks[3], and thus requires a polyfill[4]. Similarly, some individual changes are yet

[1] For more information on the implementation in regard to the web platform, see Section 4.4.

[2] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise

[3] https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

[4] https://github.com/mozilla/webextension-polyfill

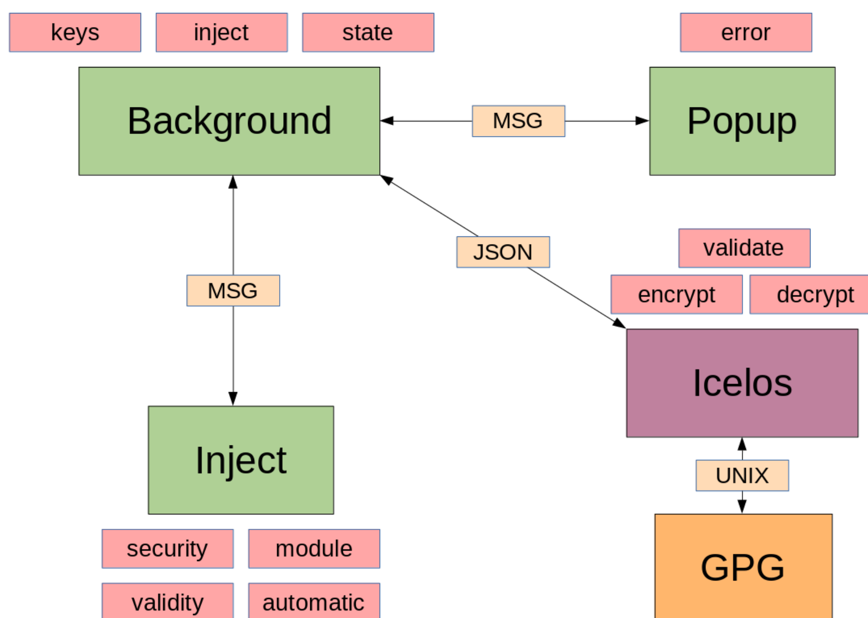


Figure 4.1: The full messaging system of Morpheus and its aforementioned components. The light red rectangles depict messages received by that module.

to be made (such as the usage of the keyword `browser` instead of `chrome` for web applications), and are again fixed with a polyfill.

Inside the extension system, a sender of a message is not necessary to be disclosed; for example, the Popup script displays an error every time the Error message is received, regardless of the origin. Similarly, any message telling the Inject Script to enable automatic message decryption will be processed regardless of the origin. Any messages passed within the extension are always **internal messages**, and will automatically be considered trusted as only a component of the extension has access to these messages.

4.2.4 Context Separation

The Icelos process never sends messages back to the rest of the extension — instead, it acts as a server, directly serving any message requests coming from the Background Script. The behaviour when handling errors is mimicked, so while in reality Icelos's messages are processed partly by the Background Script, it is well within reason to discuss Icelos as one part of the extension receiving messages in this sense.

In the same way, messages passed back and forth between the Inject Script and the rest of the extensions are actually brokered by the Background Script: the Inject Script does not have access to the **internal messages**; instead, it needs to use **tab messages** to communicate with the rest of the application. This enables for complete separation of concern, where the Inject Script does not have any more information than it needs. It does not know anything of the keys Morpheus is encrypting for, nor details of any errors that happen — only whether an error has happened or not.

That is, in Figure 4.1, the only senders privileged to send on this messaging network are the Background and Popup scripts. The Inject Script messages are sent on behalf of Inject by Background (only a subset of messages), and messages via Icelos are passed via Background as

well. This minimises coupling between the scripts.

Additionally, as all Morpheus Browser Extension parts share the same code for passing messages around, the format is also shared. Messages are passed mostly in the extension message format, with the exception of the Background Script communicating with Icelos via HTTP/TLS using JSON. Icelos then uses a separate UNIX socket to communicate with GPG to encrypt/decrypt arbitrary messages.

4.2.5 Modules

Morpheus has been implemented in the most generic way possible, where the Injection script only deals with DOM 'Core Elements' described in Chapter 3. Separate modules are needed to detect these Core Elements from a webpage. In the implementation, the injected Morpheus script will detect the URL of the webpage, and if it is detected to belong to a module, Morpheus loads this module.

The module is picked in the **Site Detector** of Morpheus using Regular Expression rules. For example, when the user navigates to `https://web.telegram.org/`, the Injection script will load the 'Telegram Web' module, which contains all the information required to inject Morpheus on this IM-P.

This information is mostly a list of queries to reach the Core Elements, and Morpheus knows how to handle the rest (decrypting an encrypted message, encrypting an input, showing a lock icon in the DOM...).

```
if (SiteDetector.Current() == 'telegram-web')
  Morpheus.SetModule({
    // Make the inputs look closer to Telegram Web's style.
    customCSS: `<snip...>`,
    bindInputReplace: false,
    bindInputs: [
      '.composer_rich_textarea',
    ],
    messageFeeds: [
      '.im_history_selected_wrap',
    ],
    messageElement: '.im_history_message_wrap .im_message_body',
    messageText: '.im_message_text',
    recipientHints: [
      '.tg_head_peer_title',
    ],
    resetters: 'ul.nav > li.im_dialog_wrap',
  });
```

Figure 4.2: A code snippet of a typical Morpheus module. This is the only part of the code in Morpheus that is specific to the Telegram Web IM-P.

The usage of modules and a generic approach to the rest of Morpheus dramatically reduces the amount of code required to implement Morpheus on a single IM-P (See Figure 4.2). Both in terms of implementation, and subsequent re-implementation (if, for example, the platform has a dramatic change of layout), having **fewer than 20 lines of code to change per IM-P** is an

extremely versatile approach.

As the modules are JavaScript-based, and are instantiated within the DOM environment, the queries can also be generated when the page loads. This allows Morpheus to combat *hostile IM-Ps* that may change the query names to prevent modifications by other scripts — however, there has not been a need to demonstrate this with the current implementation of Morpheus.

4.2.6 More Efficient Querying

The content and layout of an IM-P on the web could change in a moment's notice. Because Morpheus is a system that attempts to interface with a platform without a clear and defined API, it is vital to design the DOM queries in such a way that they are the most resilient to change. While this does not mean Morpheus will always continue to work after a substantial layout change in an IM service, it does:

- Minimise the probability of such layout changes affecting Morpheus' functionality in a detrimental manner
- Make it easier to discover and re-implement the module after this change[5]

For example, a query that bases its success in the knowledge of a Message Feed core element in regard to its position relative to the document body, the smallest change in the form of the entire webpage will mean the module has to be changed. If instead a smaller proportion of the body was used to identify the queried element, any changes outside of this proportion would not affect the functionality of Morpheus. Even better, if there was another way of identifying the target DOM element without basing on information *where it is*, only a critical change to the element itself would prompt for changing the module.

Accessible Rich Internet Applications (ARIA) attributes are one example of an efficient querying mechanism. ARIA is a set of unique HTML attributes defined to help accessibility software (such as screen readers) to focus on the important content on a given website[6]. This is defined in an Resource Description Framework (RDF)[7] and is designed to make complex websites (somewhat) computer-readable. This provides an exciting alternative approach to targeting Core Elements, as the accessibility concerns of a developer of software for a large company can allow software such as Morpheus to make use of these features.

An example of this can be found in Facebook Messenger. The message feed has a few aria attributes, such as: `role="presentation"` (meaning it is the main content to focus on a page) and `aria-label="Messages"` (label read by the screen reader would say 'Messages'). Looking at the structure of the document, it is safe to define the Message Feed core element to be the one labelled by 'Messages' in ARIA, as this DOM element will contain the messages somewhere in the tree structure. If this were not the case, then many screen readers would be broken as well. The accessibility features of an IM-P can thus be used as an API to access the required Core Elements.

Of course, not all platforms support modern accessibility features. This is why Morpheus has a hierarchy of querying:

[5] While not desirable, as long as a substantial layout change affecting the IM-P will take proportionally longer to implement compared to the time taken changing the query in Morpheus.

[6] `https://www.w3.org/TR/wai-aria-1.1/`

[7] `https://www.w3.org/WAI/PF/aria/rdf_model.png`

- ARIA attributes > other attributes > visual attributes

That is, Morpheus will first try to use the ARIA model to identify Core Elements. If that does not work, it will use other attributes (such as placeholder texts, element IDs) to find the elements. If even this does not work, it will finally use visual attributes (such as CSS classes) to find the elements. As visual classes are the most volatile to change on a moment's notice, they are obviously the least desirable.

If an element is not found for some reason, only that part of Morpheus will not be functional, and every other part will try to function as normal. For example, the absence of a Recipient Hint makes Morpheus unable to encrypt messages, in which it will display an error message instead of sending an unencrypted message. Morpheus will still be able to decrypt messages, as this functionality is not affected by the absence of a Recipient Hint.

The error message will prompt the user to input the recipient key manually in the Morpheus Extension. Morpheus will be very explicit about an input being secure or not being secure[8] and users can still override the functionality if needed. This makes the implementation very resistant to change, and the extension more usable in different circumstances.

4.2.7 Element Cloning

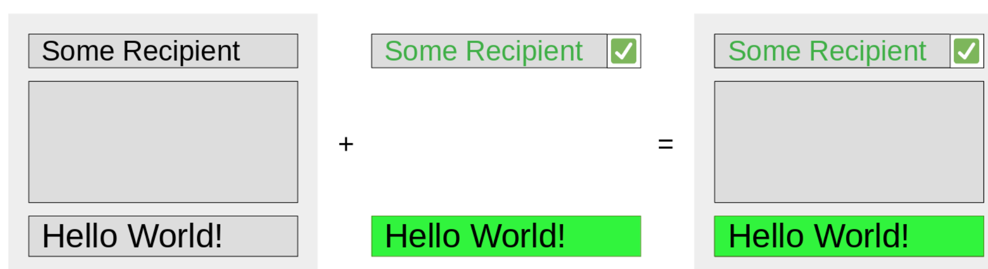


Figure 4.3: Element cloning in Morpheus, showing an example where a Recipient Hint has successfully been validated. Note that the old elements are not necessarily destroyed — they are simply hidden 'behind' the new cloned elements.

Morpheus makes modifications to DOM elements. For example, Recipient Hint core elements will be coloured and tagged based on whether the recipient is valid or not. In order to not interfere with the functionality of the IM-P, elements to be modified are cloned before modification. This allows for easy reverting to the IM-P if Morpheus needs to be disabled (see Figure 4.3) and does not modify the functionality.

The element that has been cloned can be dealt with in a number of ways. Usually, the easiest is to simply hide the old elements and only show the new elements, making the old elements impossible to be interacted with by the user. This prevents accidental typing into an unsecure input, for example.

Sometimes the IM-P requires the old elements to be in a specific place both in terms of their DOM position and visibility. Morpheus also allows Core Elements not to be replaced, and the new elements will appear next to the cloned element. They can then be hidden visually by CSS rules

[8]See section 4.5

(a) A whole encrypted message. Note the tokens wrapping the encrypted data.

(b) A chunked encrypted message. The tokens are still intact, and can be retrieved even if a single message does not contain an ending token.

Figure 4.4: A potential chunking scenario of an encrypted message, potentially enforced by a character limit or a similar delivery rule.

in the module, while still keeping the IM-P functionality intact. This is required for example if the Encrypted Input element has specific JavaScript events that the web application is listening to.

In order to prevent the user from accidentally typing into the old element, all user events are projected into the secure input instead. For example, in Facebook Messenger, clicking on the Message Feed will focus the (unencrypted) input element. Morpheus is listening for the Focus event of the unencrypted input, and will forcibly focus the secure input instead. This will mean the user will always type into the secure input, and the encrypted message will be *programmatically injected* into the unencrypted input.

4.2.8 Message Chunking

A message can be limited in character length due to transport restrictions. For example the SMS protocol historically only supported messages of 160 characters, and the length of a Twitter message used to be only 140 characters. In modern IM-Ps, these restrictions are much higher and not publicly advertised, as the platform simply chunks a large message into multiple messages if the length is too long.

This can, however, have detrimental effects on sending encrypted text. An encrypted text data is often much longer than its plaintext counterpart, and sending a lot of data can reach the character limit in a message.

Without precautions, this could lead to an encrypted message being not decryptable. For example, a chunked encrypted message in Figure 4.4 could span any number of messages. Morpheus utilises a custom token system to enclose the PGP data in, with clearly defined start and end tokens.

There are two tokens: `Encrypted:[` and `]` that the ciphertext data is enclosed in.

If message chunking has not been observed, then Morpheus will discover that both the start token and end token are in the beginning and end of the same message, respectively (Figure 4.4(a)). Morpheus can simply decrypt the contents of this message enclosed in these tokens.

If message chunking has been observed, then there is no end token to be discovered in the message (Figure 4.4(b)). This will cause Morpheus to keep reading messages until an end token is discovered. In the figure, this would mean reading the two consecutive messages, and finally discovering the end token in the third message.

In an unlikely event that the rest of the message had been chopped off, Morpheus will stop reading subsequent messages if:

- Another start token is discovered
- Message data does not look like encrypted data (armoured base64)
- Too many messages have been read

The broken ciphertext message will then not be marked as decryptable. This of course bases assumption on the ordering of the messages as they are received.

If message chunking and ordering is a concern when sending messages, it could be possible to identify the maximum message length and chunk the messages beforehand with an ordering 'token' attached. However, this was never a concern with the IM-Ps that Morpheus was interfacing with, so the feature was decided not to be implemented.

4.2.9 Message Observer

The Message Observer will attach to a Message Feed Core Element and listen to DOM changes. Any time the DOM changes, the Message Observer will see if there are any new Message Text Core Elements. These new elements will automatically be processed by Morpheus, and scanned for encryption tokens.

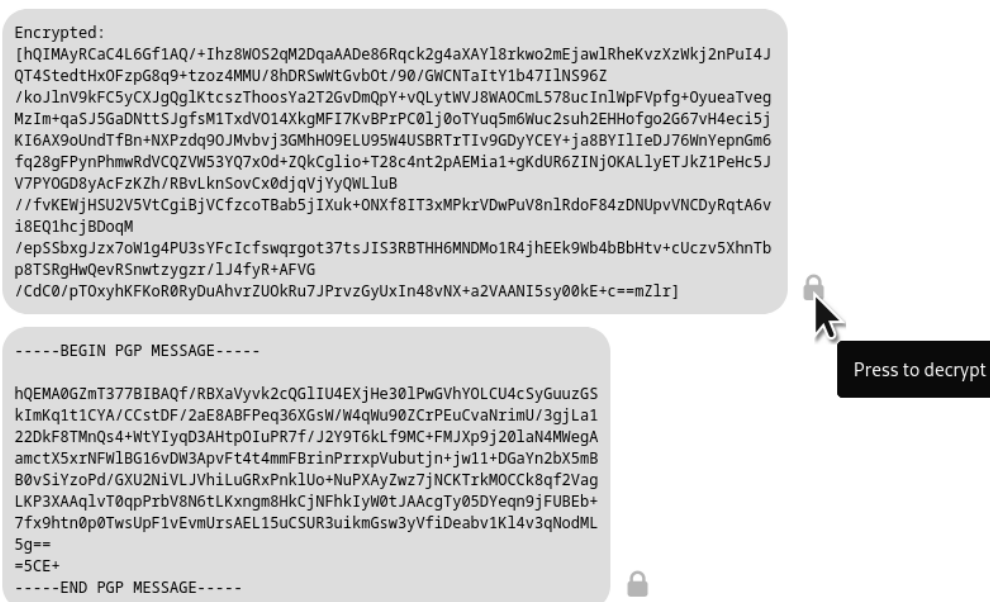


Figure 4.5: Attached User Controls that are displayed as lock icons next to encrypted messages. Clicking on a lock icon will decrypt the corresponding message. The Message Observer works both with Morpheus messages (above) and traditional PGP messages (below).

If encryption tokens are found, Morpheus will mark the message as 'decryptable', and attach user controls next to this message (Lock icon in Figure 4.5). All the other messages are left untouched for performance reasons. If automatic decryption is enabled, then after marking these messages they will be decrypted in sequence.

4.3 Icelos

Here we discuss the implementation details of Icelos. Icelos runs inside a Docker container and is built using Go. It listens on a network port and brokers requests from the web extension into GPG.

4.3.1 HTTP Messaging

As web extensions are very limited in how they can access the host machine, HTTP messaging was chosen to be the main form of communication. This also works well cross-platform, as a port can be opened on all Linux, Windows and Mac machines.

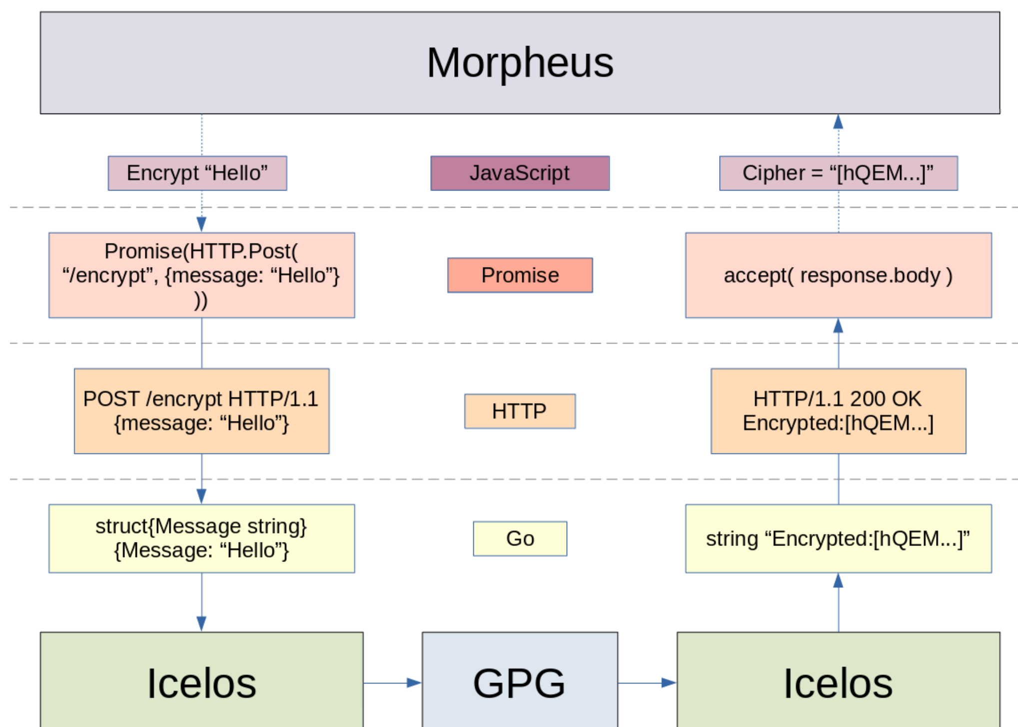


Figure 4.6: Encryption process via GPG using HTTP POST requests. The colour changes represent change in layers of abstraction. While Promises are a part of JavaScript, their asynchronous nature makes it easier for them to be seen in their own layer. The decryption and key fetching requests work in a similar fashion.

The HTTP messages are all POST requests with a JSON body. Depending on the context, the response can also be serialised JSON or a single string (for example when encrypting content). Transport Layer Security (TLS) can be used in encrypting the communication for additional security on the host machine. The diagram in Figure 4.6 shows the process in detail.

The JSON body will be serialised in the Morpheus Browser Extension, and passed over HTTP POST into Icelos. The asynchronous nature of the POST request will be represented as a Promise in JavaScript, which will be fulfilled when the POST request responds.

Icelos will then listen to the POST requests, deserialise the request body into its own `struct` format, and pass the request to GPG if needed. The backend will then create a response based on how the GPG exchange went, and respond over HTTP back to Morpheus with the response data.

Morpheus will then look at the HTTP status code and either accept (2xx code) or reject (4xx – 5xx codes) the promise based on the response status. The response data is often parsed further

in the Background Script, and then passed on via the internal messaging (see Figure 4.1) to the modules that require this information. This is important in order to keep data from leaking into the scripts on the webpage (for example, when telling the Inject Script whether key verification failed or succeeded, without disclosing any information about the GPG keys or recipients).

4.3.2 Docker

The HTTP messaging can be done through Docker as well. The only difference is that ports need to be opened and mapped via Docker. This can be left on by default or configured easily by users when installing the Docker container.

Icelos on Docker also allows users to mount their GPG directory into the Docker container rather than interfacing with their own GPG executable. This can be more secure if users want to control what kind of executables run as the specific user; the GPG Icelos runs can simply encrypt and decrypt inside a separate container, only interfacing with the user's keyring via a volume when needed.

4.3.3 Service

Icelos is also tied to a Systemd Service file which can be enabled to make Icelos run automatically as a daemon[9]. This makes the long-term usage of Morpheus less painful as Icelos does not require to be started every time Morpheus is wanted to be used.

The HTTP port can be kept open, as in order to decrypt or sign messages the private key needs to be accessed, and GPG will ask for a password. Thus, even if unattended, Icelos cannot decrypt messages without user consent. Obviously, the configuration settings of the **GPG Agent** determine the security of the user's GPG keys — if the password is set as not required, then any process could decrypt messages via Icelos. See Section 5.2.2 for more information about possible attack vectors.

4.4 Interfacing

This section describes the challenges in interfacing with arbitrary IM-Ps and how the browser extension overcomes them.

4.4.1 Inject Script

The Inject Script is the only part of Morpheus that interfaces with the web page. It runs under the same privileges as the rest of the Web App scripts do, and thus is not allowed to access any data it does not need. For example, data that would potentially tell about the GPG setup, such as key data, is never passed to the Inject Script.

The Inject Script can be appended into the webpage in two ways, as seen in Figure 4.7. The first way is perhaps the more intuitive, where Morpheus is 'always active' like some other common web extensions such as AdBlock. This injects the script onto every page that the user visits, *regardless of whether Morpheus would actually be useful on these pages*. It also requires the `content_scripts` permission on all the URLs that the user visits, which shows to the user as a request that the extension **wants to 'Read and access all data on all of the websites' that the user visits**. Understandably not all users are happy to consent to this, which is why there is another way to get the same permission.

[9] A daemon is a program that runs unattended in the background, often starting at boot time.

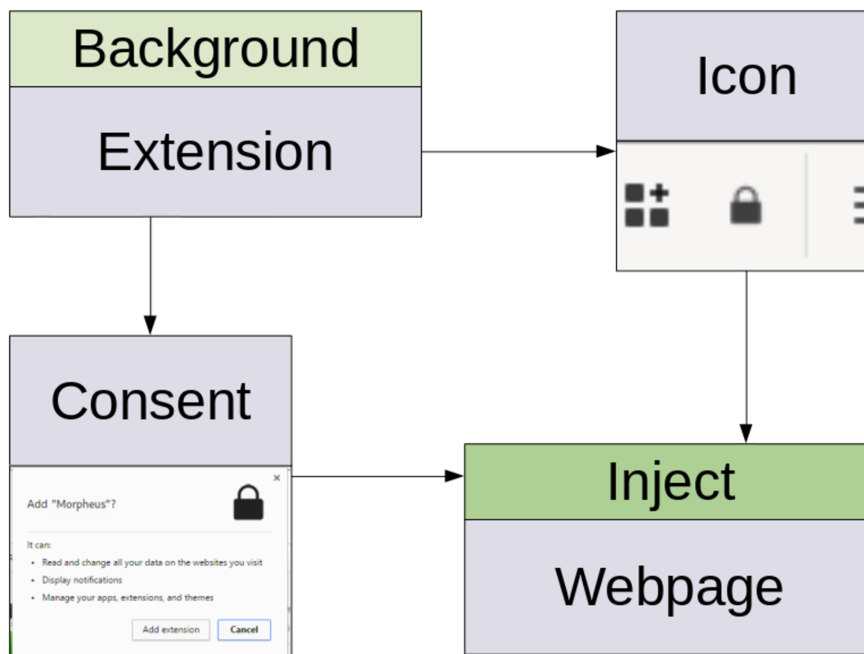


Figure 4.7: Two different pathways that the Morpheus Browser Extension can access and inject scripts on the webpage. Do note that the Background Script will keep running regardless of the permissions, but without them it will not be able to access any web page data.

Another way that the Inject Script can be appended to the web page is the `activeTab` permission. This waits for the user to click the Morpheus **extension icon** to grant the permission to inject the script onto the webpage that the user is currently visiting. The permission only stays for the duration of the browsing, and will need to be granted again, but the user does not need to accept the permission for full read access when installing the extension.

After injection, the script will evaluate which website it was injected on, and send a status report back to the Background Script. The Background Script will then evaluate using the **Site Detector** whether any modules need to be loaded.

Once a module has been identified, the Inject Script will then apply all of the queries inside this module and retrieve and do modifications to the corresponding Core Elements. This includes any cloning of elements, attachment of Message Observers and so on.

4.4.2 Reverse Engineering

As the IM-Ps often do not have clearly defined APIs to interface with, developing a Morpheus module can be close to reverse engineering how a web application works. For example, it is important to know which elements are receiving input events in order to simulate keystroke events programmatically to send a message. Some IM-Ps may send the message on the *input* event; others may listen for a *keydown* event when the Enter key is pressed.

It is thus important to devise an 'algorithm' in the web application to sending a message programmatically. This essentially consists of steps required in order to send a message successfully, regardless of the message contents. For example:

- Clear Bind Input value

- Set Bind Input value as 'Test Message'
- Fire *change* event on Bind Input
- Fire *keydown* event on Bind Input

If this algorithm consistently sends 'Test Message' to the recipient, it can be used in Morpheus to send encrypted messages. The module can then be constructed using a select few of possible event combinations keeping in mind that the message sending remains consistent.

4.4.2.1 JavaScript Frameworks

There are a few common JavaScript frameworks such as Angular and React[10], that drastically modify how the DOM behaves. Modification of native input elements becomes increasingly difficult as they are often riddled with events, and those events are required to fire in specific ordering for the framework to work properly.

A great example of this is the Facebook Messenger application, which at the time of writing uses a custom compositor for text messages. This compositor is an HTML `div` element with custom event bindings and content editability. The algorithm for sending a message gets trickier as the compositor expects all events it receives to modify the DOM, and vice versa. That is, keypresses are required to mark the `div` as *nonempty*, and only a *nonempty* `div` can then be used to send a message. Programmatically changing the value of the `div` does not mark it as *nonempty*, as no keys have been pressed. Fortunately, the value can be forcibly updated by firing a *change* event into the `div` after changing the content, and then the message sent by using a *keydown* event.

These algorithms take some trial and error to devise, but fortunately are finite to discover. The approach used with Facebook Messenger works with any input using a similar React *Rich Text Input framework*. Tackling common frameworks will most probably cover most of the required IM-Ps, but was not immediately done in this project.

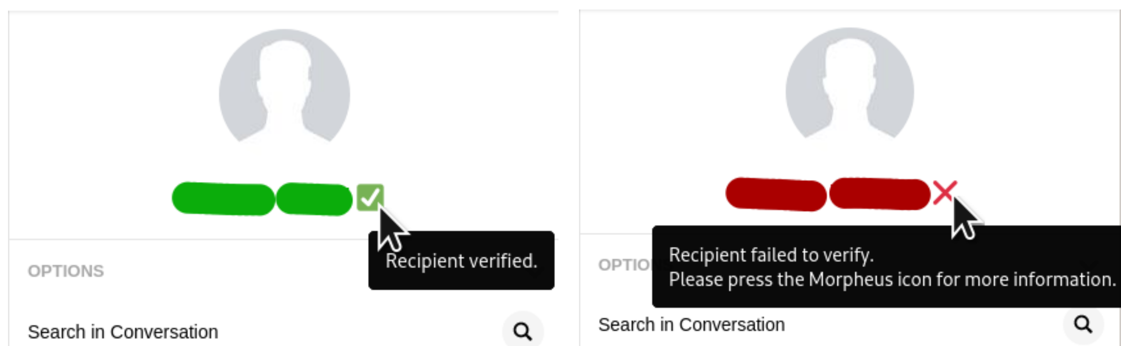
4.4.3 Mobile

There was plenty of discussion in developing Morpheus for mobile, but during the development of the extension, it was decided to be dropped.

Mobile development was deemed a difficult task due to multiple factors:

- No clear unified layout such as HTML to interface with applications (Android uses Java, iOS uses Swift)
- Mobile applications are much more restricted
- Mobile development requires an entirely new codebase with little reuse
  - Even when using a framework that works with JavaScript such as React Native, much of the extension code would remain unusable on a mobile platform
- There is very limited availability of PGP encryption/decryption mechanisms on Mobile
  - Hardware support is limited as well, although some hardware key usage can happen via Near-Field Communications (NFC) or Bluetooth

[10] Respectively: `https://angular.io` and `https://reactjs.org`



(a) The Recipient Hint displays that Morpheus has successfully determined the recipient from the name. This is shown by a modified Recipient Text Core Element in Facebook Messenger.

(b) The Recipient Hint displays a problem in verifying this user by their name. Care has been taken not to disclose any sensitive information to the page; instead, the user is redirected to the extension separate from the page for more information.

Figure 4.8: Integration with native UI elements in the web app by cloning of the Core Elements in the IM-P Facebook Messenger.

While mobile is an interesting frontier, it would probably require a double amount of time to develop. This is further discussed in Section 6.2.

4.5 Usability

4.5.1 Colours & Clarity

The focus of developing the extension was to clarify to the user whenever their messages would be secure, and when they would not. Instead of having a status quo of 'you have installed Morpheus, and now all of your messages will be secure', the extension makes it clear when a secure communication can be established, and when it cannot. Moreover, when secure communication cannot be established, the extension will explain the problem to the user.

4.5.1.1 Recipient Verification

The Recipient Hint elements are all looped through and sent over to the Background Script. The Background Script will try to match the recipient strings with a GPG name, and will report success/failure back on each of the Recipient Hints.

The Recipient Hint Core Elements will be cloned and coloured according to the success status (Figure 4.8). This will ensure that the user understands that the communication with the intended recipient is secure.

If the display name does not produce a match in GPG, or there is some other problem with Icelos, a failure message is displayed next to the recipient. This does not disclose any sensitive key information to the webpage (Figure 4.8(b)), but instead prompts the user to check the extension for more details on key management[11].

4.5.1.2 Unsecure Mode

The user is allowed to choose between 'secure' and 'unsecure' modes (see Figures 4.9 & 4.10). After all, the user may have a multitude of reasons to disable encryption temporarily. Perhaps the user does not deem their message worthy of securing. Perhaps the recipient is unable to decrypt

[11] See Section 4.5.2 for more information on key management.

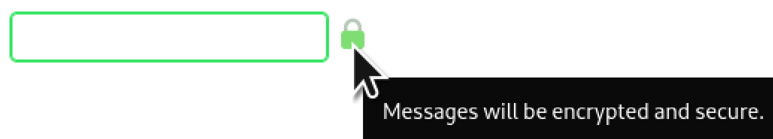


Figure 4.9: The cloned secure input that is showing a clear green lock theme and a message stating that messages will be secure.

Figure 4.10: The same secure input after pressing the lock icon. The theme is now very different with a red tint and an unlocked lock icon. The message states that messages will NOT be secure, and the user should press the icon to enable encryption.

the message and is asking the user to send it in plain text.

Regardless of the reason, Morpheus is designed to be simple to turn off momentarily, instead of forcing absolute security to its users. Care was taken to make sure that the 'secure' mode (Figure 4.9) remains very different visually from the 'unsecure' mode (Figure 4.10) in term of colour, appearance and contrast.

4.5.2 Keys

Clicking on the extension yields a more advanced menu for key inspection and management (see Figure 4.11(a)). It also shows the status of Morpheus at one single glance. Any keys that could not be matched for a reason or another are displayed in respective green and red colourings based on their status (Figure 4.11).

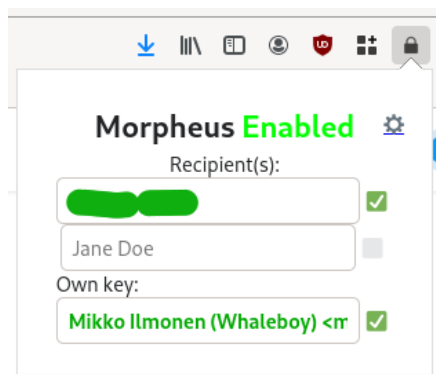
The user can change the settings of Morpheus by clicking the cogwheel icon on the top right. This shows a straightforward list of settings that can be viewed and adjusted quickly (Figure 4.12).

The user can also input a custom GPG key for any recipient if, for example, the messaging service is using a nickname instead of the recipient's GPG name (or the other way around). Hovering over any of the keys will give more information about the key that was matched.

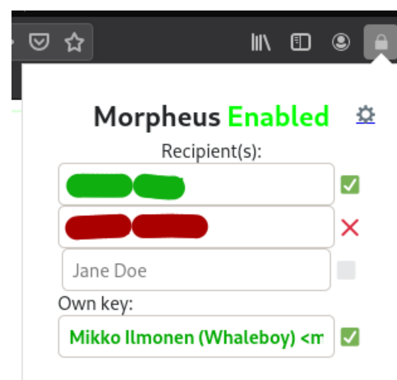
For any key that has been successfully found, the key holder name, e-mail, comment and its fingerprint will be displayed when the user hovers over the information (Figure 4.13). This allows the user to verify that the intended recipient is correct, and the key is the right one.

Similarly, for any failed verifications, hovering over the icon will reveal details as to why the verification failed (Figure 4.14). A key verification can fail due to many reasons, so it's important to display this to the user.

Most importantly, this detail of the keys is not passed into the Inject Script, but will instead only be displayed in the Popup. This is due to additional security of the Popup having its own environment. The Inject Script does not have the need to know about the user's GPG setup.



(a) Status message when clicking the extension icon. All of the keys that Morpheus has detected on the page are displayed, where users can now change them if needed.



(b) Status message in a group chat. Morpheus has detected some keys, but some other keys have a problem. Users are pointed to which recipients could not be found, and can change them if needed.

Figure 4.11: Morpheus status message when clicking the extension icon.

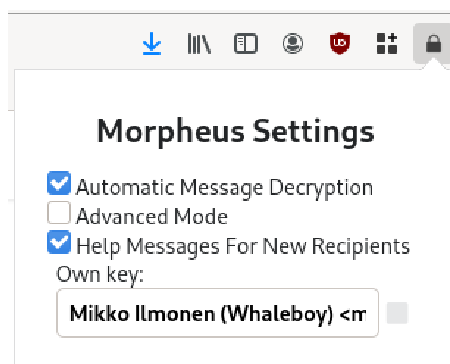


Figure 4.12: Settings page of Morpheus.

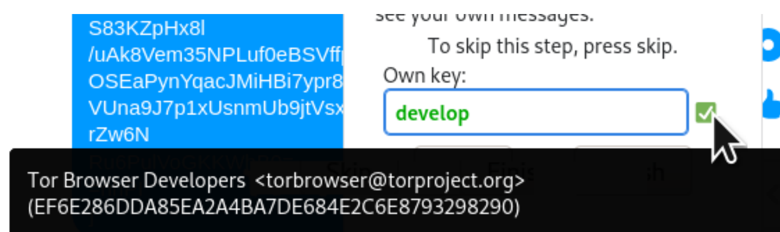


Figure 4.13: Detailed information about the selected key. The fingerprint is shown inside the extension to make sure the chosen key is correct.

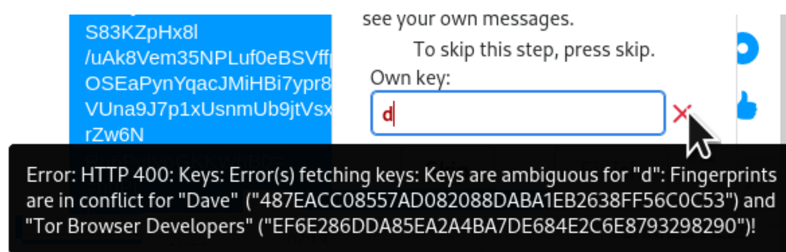


Figure 4.14: A conflict when multiple keys could be matched.

Chapter 5

Evaluation

Here we describe in detail how Morpheus was evaluated and tested, and how well these tests fit the requirements described in Chapter 3.

5.1 Overview

In this section, we discuss the maintainability and performance aspects of the software.

5.1.1 Portability

Morpheus works well as an addition to the workflow of sending instant messages, regardless of the platform it is being used on. It can stay in the background without much intervention, and even when people do not have a key setup, the unsecure mode allows people to talk while explicitly not encrypting their messages.

Some users may want to only establish secure communication for a few times, and this is possible due to the way Morpheus is designed. As it is not required to have Morpheus on the other side, people are not forced on the platform but start using it of their own will. The hope is that this will encourage more people to establish better personal security due to the availability of the extension and the relatively easy setup.

Eventually, as more people adopt the usage of PGP (or any other encryption mechanism), tools such as Morpheus should become more mainstream, and could even become supported by IM-Ps. This shift of trust away from the platform to the end-user is the primary goal of Morpheus, where this project provides one (not necessarily the best) tool to do so.

Like all proofs-of-concept, Morpheus offers an insight into one solution, yet has a lot it could do differently. It is important to remember that *exclusively secure messaging apps* already exist far and wide. As an engineer, it would be easy to simply state that ‘if people wish to be more secure, they should switch platform’. Morpheus provides a tidy solution that does not interfere with peoples’ workflows and yet enables them to be more secure if they so wish.

5.1.2 Performance

For Morpheus to work seamlessly with IM-Ps and not interrupt users’ workflows, it needs to perform well when encrypting and decrypting messages. There is obviously some performance overhead compared to not encrypting/decrypting at all, which is investigated in detail. As a web extension, it is also important to note the possible slowing down of page loads if a lot of processing occurs.

This section describes the performance implications of the aforementioned aspects. For discussion on the results, please see Section 6.1.4

5.1.2.1 Setup

All of the experiments were conducted on a machine with an `Intel Core i5-6300U CPU @ 2.40GHz`, running `Linux 5.5.10-arch1-1 x86_64`. All experiments on browsers were tested on both `Mozilla Firefox 74.0` and `Google Chrome 80.0.3987.149`. The browser cache was disabled for repeated tests.

The experiment setup uses a test HTML/JavaScript/CSS IM application that is running on a single Go server on localhost, port 8000. This eliminates any network variability which we are not interested in. All the code can be found on `https://gitlab.com/ilmikko/morpheus/morpheus-test`.

All of the CSV data was processed in the R language and the function `t.test` was used to retrieve the p-values for each experiment. The graphs were generated from the CSV data in *Libreoffice 6.4.3.2 40*.

5.1.2.2 Page Load Times

The hypothesis in page loading is that **Morpheus will significantly increase page loading times**, but **not for more than 500ms**, considered 'almost instantaneous' in the Non-Functional Requirements.

For page loading, we were interested in three main events representing the stages of page loading:

- DOMContentLoaded
  - Fired when the DOM is ready and all the HTML has finished parsing, but the rest of the page is still loading.
- Load
  - Fired when the page content has loaded, but not all scripts have finished loading.
- Finish
  - Fired when all scripts have finished.

The experiment consists of loading the localhost chat application. The application contains three encrypted messages that are marked for decryption, a Bind Input, and a Recipient Hint of a known recipient. It was observed how page load times change when Morpheus is disabled, and when it is enabled.

This was done by using the Network tab of the Web Developer Tools available on both browsers. The page was refreshed, and the value of the above metrics was noted down. This test was repeated 50 times.

The test on the Firefox browser yields on average 294.62ms page load times when the extension is disabled (see Figure 5.1). When enabled, the page load times were on average 383.2ms. The p-value of the t-test between the data sets is $p = 2.467 * 10^{-6}, p < 0.05$, making the change in page load times between having the extension disabled or enabled statistically significant.

Similarly, the test on the Chrome browser yields on average 235.52ms page load times when the extension is disabled (see Figure 5.2). When enabled, the page load times were on average

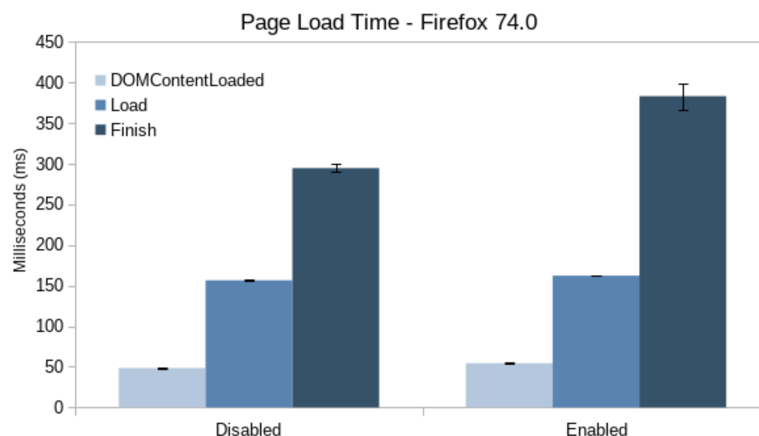


Figure 5.1: Average page load time in milliseconds on the Firefox browser. The experiment was repeated over 50 page loads each with Morpheus disabled and enabled. The error bars represent standard error.

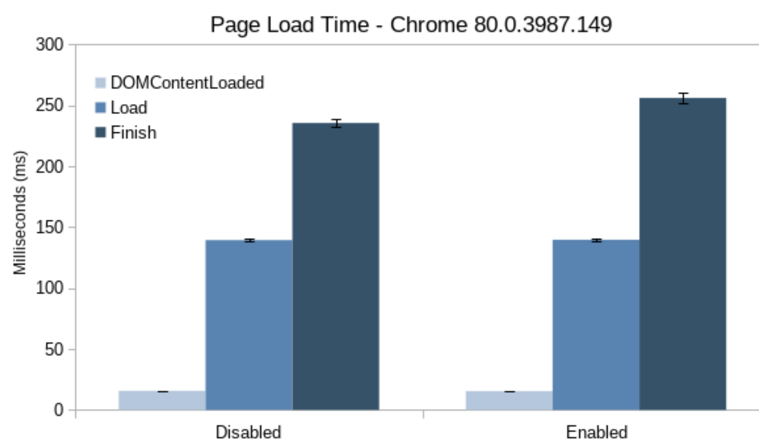


Figure 5.2: Average page load time in milliseconds on the Chrome browser. The experiment was repeated over 50 page loads each with Morpheus disabled and enabled. The error bars represent standard error.

256ms. The p-value of the t-test between the data sets is $p = 2.2 * 10^{-16}, p < 0.05$, making the change in page load times between having the extension disabled or enabled statistically significant.

5.1.2.3 Encryption

An experiment was held to measure the impact of message encryption on the latency of sending a message. The hypothesis was that **due to using Elliptic Curve Cryptography (ECC), the message delivery time would be significantly longer**, but **not more than 500ms** so that it still stays within the performance barrier set by the Non-Functional Requirements.

The test was carried out by modifying the Morpheus source code to output a START time stamp closest to the nearest millisecond when Morpheus receives a user input (user hits 'Send'), and again an END time stamp when the ciphertext is passed to the underlying IM-P (and 'sent' over the network). A bash script was written to type in a message 'Message Test' and then hit Enter, repeated for 500 times, sleeping a few seconds in between typing to ensure the previous

message was successfully sent. This script was let to run in its entirety, and the output was then processed further. The time duration between the END and START timestamps was extracted, and this was written in a CSV file.

This test was performed only on `Firefox 74.0`, as the main bottleneck for encryption times is GPG and Icelos, and times were very similar between the browsers.

The average time for sending a message without encryption (control) was 2.262ms (Figure 5.3). When encryption was used, the average time was 272.744ms. The p-value of the t-test between the two data sets is $p = 2.2 * 10^{-16}, p < 0.05$, making the difference statistically significant.

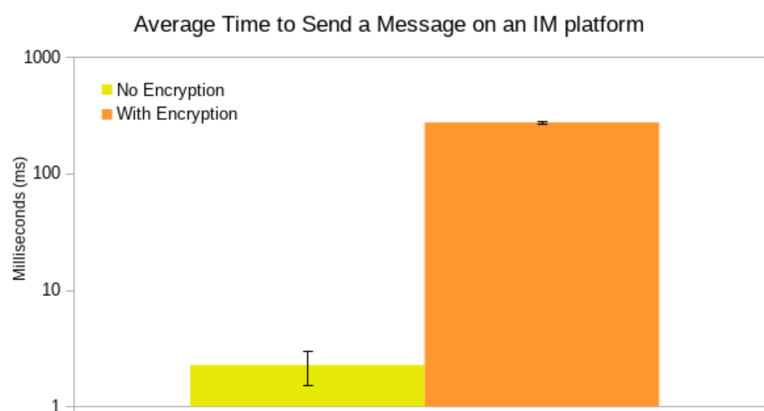


Figure 5.3: Average time taken to send a message over an IM-P, from the user input event to the IM-P sending the network packet. The error bars represent standard error. Note that the y-axis is on a logarithmic scale.

5.1.2.4 Decryption

Finally, an experiment was held to measure the impact of message decryption when receiving messages. A control was not deemed necessary as the time taken from receiving a message to someone observing said message would essentially be instantaneous, whereas the real interest is how long it takes for a message to be decrypted in order for someone to observe the said message. The hypothesis was that **decryption of messages would not take on average more than 500ms** so that the software stays within the performance barrier set by the Non-Functional Requirements.

This test was carried out by again modifying the Morpheus source code to output a START time stamp closest to the nearest millisecond when Morpheus sees a message, and an END time stamp when the plaintext is displayed in the message body.

Morpheus was modified to send fixed messages of given size periodically. More specifically, there were three different types of message size that decryption was tested on:

- Few words
  - This is simply a message containing the string ‘Message Test’
- 1KB
  - The first 1024 characters from the beginning of the GNU GPLv3 License[1].

[1] `https://www.gnu.org/licenses/gpl-3.0.txt`

- 32KB
  - The first 32768 characters from the beginning of the GNU GPLv3 License.

These messages were to simulate typical sizes of instant messages sent by humans, and probe the limits of encryption of larger content via Icelos and PGP. While 32KB is not a lot of characters for a computer to process, it is a substantial amount of characters to write in a single instant message.

The test was again performed only on `Firefox 74.0` due to the reasons stated previously.

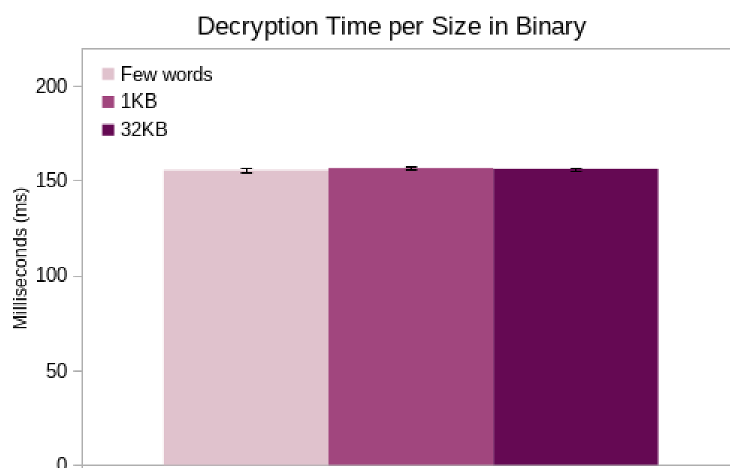


Figure 5.4: Average time taken to decrypt messages received on an IM-P, with text content of given size. The error bars represent standard error.

The average times of decryption were 155.346ms, 156.546ms, and 155.98ms — respectively for 'Message Test', an 1KB message and a 32KB message (Table 5.4).

The p-values for the t-tests can be found in Figure 5.5. All of the values in the table $p > 0.05$, which means there is no statistical significance between the datasets.

	Few words	1KB	32KB
Few words	n/a	0.1217	0.4078
1KB		n/a	0.4488
32KB			n/a

Figure 5.5: P-values for the t-tests of each combination of the datasets.

5.2 Security

This section discusses in detail all of the security aspects of Morpheus, as well as potential attack vectors and abuse cases of the program.

5.2.1 Perfect Forward Secrecy

The same way as with e-mail discussions, PGP does not have perfect forward secrecy. This means that any encrypted messages sent some time in the past can be decrypted if the private key gets compromised. This can be a problem if message history is retained in the IM-P. If the adversary can steal the private key, they will have access to *all sent messages* in the past.

This can be resolved by using one-time ephemeral keys generated either per discussion, or per message. After the discussion/message is deemed useless (e.g. if read), the ephemeral private key is destroyed, and a new one is generated. These keys are signed with the master key that is long-term and kept very private.

This project does not explore this option as it is mainly a PGP concern to set up a keying system with ephemeral key rotation. While it is fair to perhaps consider PGP as not the best method for performing encryption/decryption in communications such as this, Morpheus is fortunately not strictly tied to PGP as a form of encryption. Future work of either implementing OTR[2] or the newer Signal protocol on the framework of sending/receiving messages via Morpheus is a possibility. Using either of these contains a session-based ephemeral key setup where losing a key is not such a large issue any longer[3].

5.2.2 Abuse Cases

5.2.2.1 Malicious IM-P

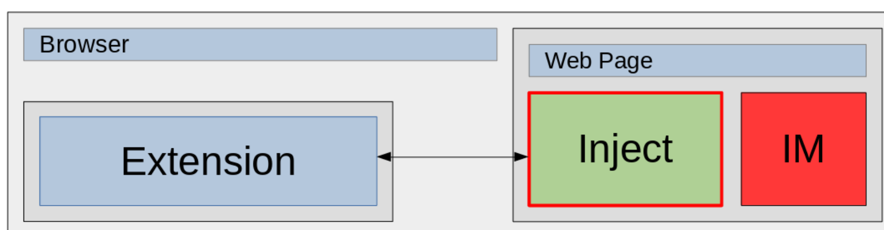


Figure 5.6: Malicious Instant Messaging Platform or Webpage that Morpheus is used on. The red outline indicates parts of the program that are compromised.

Morpheus directly accesses and edits the Data Object Model of the target web application in order to display text. It clones elements and retains the originals in order to affect the functionality of the IM-P as little as possible. However, as the DOM is shared between extensions and the website scripts, it is possible that a malicious particularly targeted website script can target the cloned elements and read their decrypted contents, as the principle of separation is violated in current implementations of DOM. That is, the decrypted plaintext could be retrieved when the DOM updates.

This can be mitigated in using the 'Advanced Mode' described in Section 4.5.2. This means that the messages are left untouched, but decrypted in the extension and displayed there. The

[2]See Off-The-Record Messaging (OTR) in Chapter 2
[3]See Future Work in Chapter 6.2 for more information.

website will not have any information as all encryption/decryption happens in the extension and its own closed container, as seen in Figure 5.6.

The reason why the advanced mode is not enabled by default is mainly in line with the motivation of the rest of the project. We are assuming there is not necessarily a malicious adversary on the IM-P — the encryption/decryption is done mainly because of concerns on data retention. Were a platform *truly malicious in intent* and willing to create a *Morpheus-specific* attack, we need not only to enable the advanced mode, but also to reconsider the priorities (mainly why this IM-P / IM-S is used in the first place).

5.2.2.2 Malicious Operating System

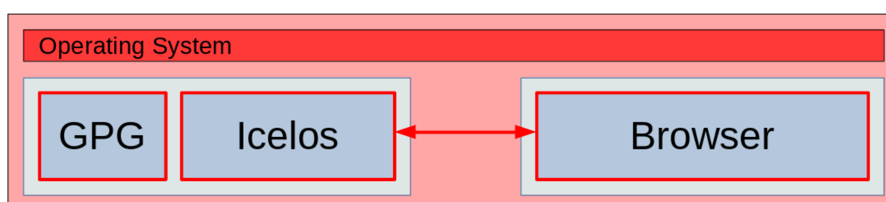


Figure 5.7: Malicious Operating System (OS) that the entire system is running on. The red outline indicates parts of the program that are compromised. The red arrows indicate intercepted traffic.

In case an operating system has been infected with a virus, or has other malicious proprietary software running on it, the security benefits of the entire Morpheus system can be irrelevant.

If the underlying system is compromised, **even the private keys handled by GPG** can be extracted regardless of GPG running in a container or on the host system. That is, no part of the program is secure, as the OS has access to all of them (Figure 5.7).

Using a trusted and open-source Operating System (OS) will eliminate the possibility of this abuse case.

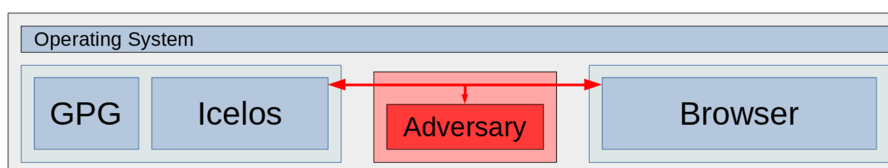


Figure 5.8: Malicious executable running on the host system. The red arrows indicate intercepted traffic.

If the system is not compromised, but an offending binary has access to the system, it can potentially sniff on HTTP packets sent to and from Icelos and thus gain access to any plaintext before encrypting, and plaintext after decrypting (See Figure 5.8). This sniffing can be mitigated by using Transport Layer Security (TLS) to communicate between Morpheus and Icelos, which can be enabled from Morpheus given that the certificates are generated correctly.

5.2.2.3 Malicious Browser

Even though the Morpheus web extension is run in a private and secure container, a malicious browser would be detrimental to the security of the encrypted and decrypted messages. Thus, the

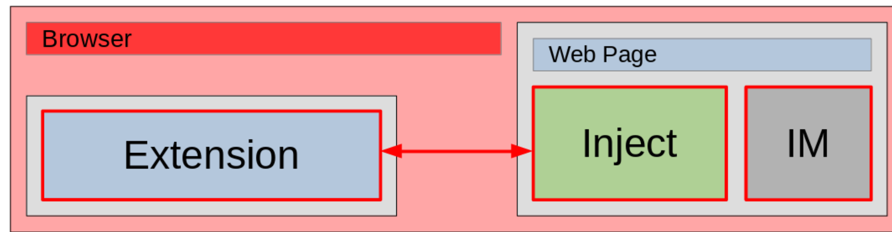


Figure 5.9: Malicious browser executable that Morpheus Browser Extension is running on. The red outline indicates parts of the program that are compromised. The red arrows indicate intercepted traffic.

entirety of the browser execution would be compromised, as seen in Figure 5.9.

The user's private keys would remain safe, as neither Morpheus nor Icelos has access to them; by design only GPG handles the keys. However, any ciphertext decrypted by Morpheus, regardless whether in Advanced Mode or Basic Mode, could be intercepted by a malicious browser. The same applies to plaintext that is encrypted using Morpheus. Using a well-known and open-source browser which has been properly audited for security will eliminate these issues.

5.2.2.4 Malicious user of Icelos

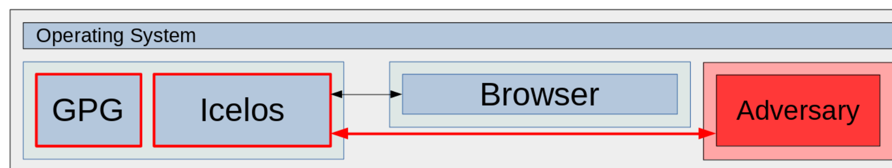


Figure 5.10: Malicious executable running on the host system that poses itself as Morpheus and communicates with Icelos. The red outline indicates parts of the program that are compromised. The red arrows indicate intercepted traffic.

An adversary could encrypt and decrypt messages on behalf of someone, *if they get access to the host machine*. This would be done by crafting requests that appear to be coming from Morpheus, and sending them to Icelos over HTTP (Figure 5.10). While the adversary would not gain access to the user's private keys, they would be able to encrypt and decrypt any number of messages as Icelos has direct access to GPG.

A way to mitigate this attack is to use modern firewall software on the client. Morpheus and Icelos should also have SSLTLS enabled to encrypt all communication between the processes. Morpheus could then be modified to employ more ways of authentication between the browser extension and Icelos.

5.2.2.5 MitM on Icelos

Some adversary could craft a bogus Icelos process to run on the host. If the process does not talk to GPG, it could not decrypt encrypted messages, but could still see plaintext before it gets encrypted (Figure 5.11). If the process does talk to GPG, it could successfully act as Icelos, brokering messages to and from GPG, but additionally keep log of the plaintexts.

In order for this to happen, the host machine would already need to be compromised.

This can be again mitigated by using SSL/TLS with trusted keys, and perhaps another layer

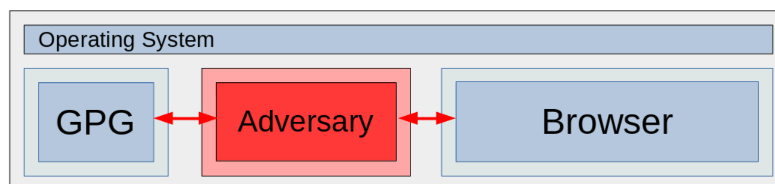


Figure 5.11: Malicious executable running on the host system that performs a MitM attack on traffic. The red arrows indicate intercepted traffic.

of authentication to ensure the browser is talking to the correct Icelos.

5.2.2.6 Unattended Computer

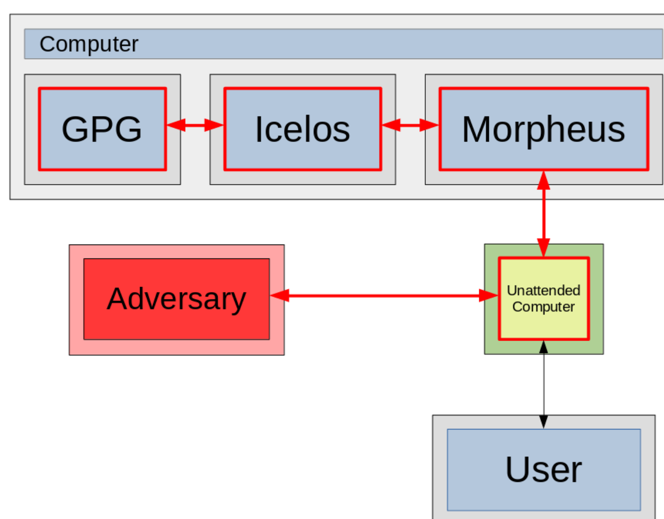


Figure 5.12: Adversary taking advantage of an unattended computer. The red outline indicates parts of the program that are compromised. The red arrows indicate intercepted traffic.

An adversary can gain access to the decryption mechanism of the GPG Agent if it has been recently used. By default, GPG allows to cache the passphrase to the private key in order to subsequently encrypt/decrypt/sign multiple messages. The default time window for this is 10 minutes — that is, the first message needs a password to decrypt, and the password is then cached until 10 minutes of inactivity.

This means that if a person were to use Morpheus to decrypt messages and leave their laptop unlocked, an adversary could decrypt messages if they use the computer in the last 10 minutes, potentially compromising the system, as seen in Figure 5.12. A fix for this is to change the default cache lifetime to something shorter, remove caching altogether and practice secure computing by locking the computer after usage.

5.2.2.7 Over the Shoulder

An adversary could wait and look at plain text messages that are decrypted by the victim, as seen in Figure 5.13. The best way to prevent this is to decrypt messages only in a private space. By default, Morpheus does not automatically decrypt messages — the user has to prompt message decryption by either:

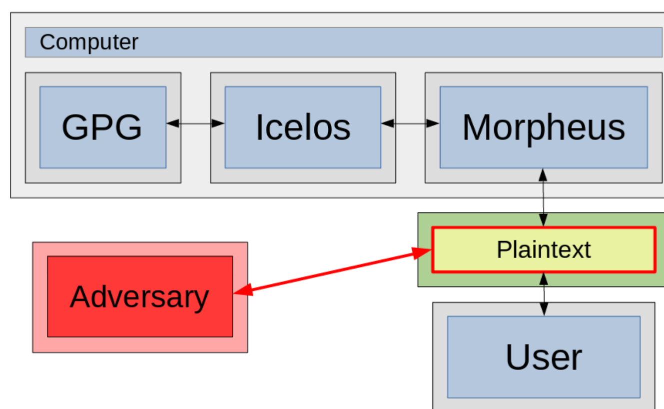


Figure 5.13: Adversary physically intercepting plaintext by looking over the shoulder of a user. The red outline indicates parts of the program that are compromised. The red arrows indicate intercepted traffic.

- Clicking on the lock icon next to an encrypted message to decrypt that single message.
- Enabling automatic decryption and thus decrypting all messages.

5.3 Testing

The business code was tested vigorously using all three methods of testing described below. Some of the UI elements were **unit tested** as well when creating the component, and then later **integration tested** manually.

Encryption/decryption speeds were tested to keep a baseline of how fast the automatic encryption/decryption works (see Section 5.1.2).

5.3.1 Component Testing

Component testing happens via **unit testing frameworks** in both Go and JavaScript. For both languages, each individual component is tested individually against a select set of test cases. **Table-driven testing** ensures that test code stays concise and good coverage is reached. All the tests create an average test coverage of 92% across the modules of Icelos. Morpheus Browser Extension components were also unit tested extensively where possible. Any regressions (previous bugs) are also individually tested here to ensure they do not re-emerge.

5.3.2 System Testing

Even though there are no full integration tests with the back-end and front-end working together, the system has integration tests for both of the ends separately. The back-end is tested that it runs and communicates with GPG, and the front-end is tested to ensure proper communication between modules.

5.3.3 Manual Testing

Finally, there is a certain number of test cases that are gone through in manual testing in order to increase confidence that the software is working as expected. These manual tests are done in a so-called ‘playground’ which provides a very basic ‘IM-P’ to test against. This is the same testbed that was used in the performance evaluation in Section 5.1.2.

The extension was tested on a few online platforms where I would ask a friend to install Morpheus using the instructions of the user manual (Appendix A), and we would establish a secure connection over the platforms[4]. I gained valuable feedback on what the pitfalls were in both installing and using the system.

5.4 Known Issues

5.4.1 Store Availability

The extension is currently not on any common browser extension ‘stores’. This is mainly due to time constraints of the project. There is a code review process that goes into releasing browser extensions on the stores, and all major browsers have their own store systems.

Including Morpheus on all of the stores is not a goal of this project, and would potentially take up a lot of development time as a ‘request for code and quality review’ would need to be submitted individually for each of these stores. As Morpheus is largely a prototype, the cost of re-installing the applet via an unconventional route while developing is not as high.

5.4.2 No separation of Own Keys from Recipient Keys

Morpheus needs to know the recipients’ own key in order for them to decrypt their conversation history later. This was implemented in such a way that the own key simply becomes an additional recipient to the encrypted message. The decision was made to simplify logic when encrypting; while this is true, Morpheus can run into some edge cases:

If all recipient keys are missing in a group chat, an encrypted message would still be sent. However, this message would only be encrypted to the sender — practically useless as the only person who is able to decrypt the message has already seen it (being the one writing the message).

This can be fixed by either having Icelos handle Own Keys (by probing the list of GPG secret keys, for example), or specify the two different key types when encrypting a message.

5.4.3 Difficulty of Creating Modules

Even when the modules are relatively simple and definitely concise, it requires quite a lot of technical JavaScript knowledge to develop a module to Morpheus. It would be great to develop a solution where users could simply point to an encrypted message and decrypt it at will. Similarly, they could point to an input to ‘secure’ that input, regardless of the website.

This would tremendously extend the use cases of Morpheus, but isn’t unfortunately as easy to do as it may intuitively sound. For example, given that web platforms have different sets of events that input fields (or not even input fields but `divs`) listen to. It would be a painstaking process for the software to *guess* which permutation of events to use for sending a message.

5.5 Fulfilment of Requirements

- Secure encryption of messages with PGP on an IM-P
  - Morpheus provides two ways of securely encrypting messages on an IM-P.
- Secure decryption of messages with PGP on an IM-P
  - Morpheus provides two ways of securely decrypting messages on an IM-P.

[4] PGP key exchange was handled separately, in person.

- Detection of recipient and their key using the name on an IM-P
  - Morpheus detects recipients based on Core Elements on a web page, and communicates with GPG via Icelos to fetch the correct recipient keys.
- Detection of errors when recipient name is defective, either:
  - if the name is ambiguous
    * Morpheus specifically informs when a name is ambiguous (Figure 4.14)
  - if the name cannot be found (i.e. when using a nickname)
    * Morpheus informs when a recipient name cannot be found.
- Ability to change the recipient's name manually if it is defective
  - Morpheus provides an input to change the key in the extension (Figure 4.11(a))
- Interfacing with at least three different IM-P
  - Morpheus has modules and can successfully encrypt and decrypt messages on the following platforms:
    * Discord
    * Facebook Messenger
    * Telegram Web
    * WhatsApp Web
    * Slack

5.5.0.1 Non-Functional Requirements

- Encryption/decryption should happen in a speed that is considered 'almost instantaneous' (< 500ms)
  - Encryption and decryption both happen in a time below 500ms (Section 5.1.2)
- Should use open-source software for security reasons
  - Morpheus is fully open-source, and the code is available on GitLab[5]
- Should keep platform-specific code at a minimum in order to improve code re-usability
  - The only platform-specific code is within Morpheus modules, and it consists of only around 50 lines of code per platform.
- Should allow users to interface (send encrypted messages) with their platform of choice
  - Morpheus supports a predefined set of platforms.
  - It is also possible to write a module for Morpheus relatively easily.
  - However, an Element Picker approach could also work here (See Future Work in Chapter 6)

[5] `https://gitlab.com/ilmikko/morpheus`

Chapter 6

Discussion

Over this chapter, we wrap up and discuss how the project went in its entirety. We discuss achievements, future work and learning opportunities that this project brought.

6.1 Achievements

6.1.1 Platform-Agnosticism

Overall, Morpheus works well as a truly platform-agnostic program. The modular design pattern allows Morpheus only to have around 50 lines of code for each individual web platform, and all the rest of the code stays generic. This makes it extremely easy to extend the functionality to different sites without modifying the existing program code (and potentially breaking other sites' functionality in the process).

Furthermore, because Morpheus uses the Extension API instead of targeting a single browser, it is available for many different browser setups (Chrome, Firefox, Opera...). The usage of ES8 (ECMAScript 2017) ensures that most browsers support the functionality natively, and the usage of Go for building the binary ensures maximum portability on most platforms[1].

The system fully supports UTF-8, and sending characters such as letters in the Latin alphabet or emoji works the same way as 'regular' text. Icelos and GPG also support encryption/decryption of binary data such as audio and images, but due to interfacing questions (see Section 6.2.4) wasn't fully realised in this project when interfacing with IM-Ps.

The usage of Docker for Icelos and GPG gives more portability to the client-side of Morpheus, allowing it to be installed on many major platforms.

6.1.2 Security

While it is left for other people to perform a rigorous security audit on the software, the software has always been built security in mind. Morpheus does not introduce any new encryption algorithms — instead, it uses well-known, trusted, and up-to-date software to perform encryption and decryption.

Additionally, users are free to experiment with their own setup. While sensible security is provided out-of-the-box, obviously different people have different use cases and thus the configuration is easily available.

[1] See Section 3.2 for description about the choice of languages.

6.1.3 Respect for peoples' individual setups

As well as being indifferent to the platform that people are using Morpheus on, many steps were taken to ensure it would cater to as many setups as possible. The usage of GPG allows for encryption/decryption using any setup, including a standard encrypted/unencrypted local private key, usage of smart keys / smart cards, etc. Thus the usage of master- and sub-keys is abstracted away by GPG and Morpheus can enjoy the full feature set of the program.

The usage of Docker for the Icelos instance allows complete independence of the host platform, meaning Morpheus can be installed on Windows, Mac, Linux or any OS that can run Docker[2].

Morpheus also does not force users on the receiving side to use Morpheus. Users could use any software that encrypts/decrypts PGP to read the messages, or even develop their own. When sending a message, Morpheus gives a link to a webpage[3] that helps people use the software on different platforms, or even decrypt messages manually without the help of the software.

6.1.4 Performance

Despite having large keys and a lot of messages, Morpheus is still fast to use. The usage of Promises in JavaScript code, and a concurrent backend in Go, both enable the application to be fast in encrypting/decrypting messages. Any expected delays are clearly displayed to the user: for example, when decrypting a group of messages, they all change to show a loading message, and then be procedurally decrypted in sequence of importance.

The IM-P remains usable even when Morpheus is decrypting or encrypting parts of it. Any new messages that appear are caught by the Message Observer and marked for potential decryption. Any message is only ever touched once, which means the website is not repeatedly scanned in its entirety. This brings down the complexity of the message marking algorithm to $O(n)$ for n messages.

The performance details can be observed in the testing results in Section 5.1.2.

6.1.4.1 Page Load Times

Looking at page loading speeds alone, Morpheus adds only around 25 milliseconds (on Chrome) to 100 milliseconds (on Firefox) to page load time (Figures 5.1 & 5.2). As this increase is already after the visual content (DOM, CSS) and the rest of the remote content has been loaded, the perceived delay can be considered even less intrusive.

This includes time taken to initialise the script, set up relevant Message Observers, load up a module for the given IM-P, mark messages for decryption and secure any Bind Inputs.

6.1.4.2 Encryption

Sending messages without encryption is obviously faster than encrypting the same messages. The encryption experiment shows that Morpheus adds around 270ms of time for encryption before a message can be sent (Figure 5.3). This falls well within the acceptable boundary set in the Core Requirements, and does not create a perceivable difference in IM-P usage.

[2]Windows nor Mac were unfortunately not tested during the course of the project due to equipment restrictions and unforeseen relocation circumstances due to COVID-19.

[3]https://5x.fi/morpheus/help

6.1.4.3 Decryption

Decryption of messages is a larger performance concern, as in a chat application context there may be 10-20 messages that are visible on screen. Morpheus decrypts all of these messages, some of which may be large in size, so it's important to be performant in handling decryption.

From the experiment, we can conclude that there is no conceivable difference in decryption speeds when dealing with IM messages of a traditional size. Even messages of over 30 000 characters are decrypted almost instantaneously in around 150 milliseconds, the same time taken for small messages (Figure 5.4).

For 20 messages, decrypting all of them without any parallelism would take around 3 seconds. As both Icelos and Morpheus are designed to work with concurrent requests, this time is taken down even further. Because of the Message Observer algorithm, decryption of new messages usually takes only a few hundred milliseconds. Again, because the DOM is updated while decrypting, the perceived delay is less.

6.2 Future Work

Clearly, Morpheus is more at a proof-of-concept stage than a finished product that can be distributed to all end-users. This section discusses improvements that future work could hold.

6.2.1 Usability & Commercialisation

There are some usability improvements that could be done to bringing Morpheus to a more general audience. Perhaps the biggest blocker for widespread usage of the tool is the difficulty of developing modules. This could be prevented by allowing people to choose the Core Elements using a pointer tool and simply clicking an element, similar to how you can select a specific element to block in AdBlock[4].

This allows users to use Morpheus on sites that do not have a module that needs to be developed by a developer, and requires virtually no programming experience.

In order to make the message writing system more reliable, there are other ways to perform artificial keyboard input on the client-side. This would mean that, instead of firing a complicated sequence of JavaScript events, the OS could send keystrokes to the browser using a framework such as Selenium. While this is a bit more complicated to implement (as sending keystrokes to an application is a very platform-specific task), it would make sending encrypted messages more robust and reliable.

6.2.1.1 Software as a Service

The extension should also be put into most common stores for Web Browsers to reach a wider audience. This includes some cost, and supporting major platforms always incurs a small maintenance cost as they will eventually change.

Morpheus could be provided as a Software as a Service — while it is free to download and install for a tech-savvy person, there could be a 'professional' version that includes support. This could provide a stable revenue stream to the developers while still keeping the project open-source for security and trust reasons. As a plus, buyers of the software can also feel assured that they can receive timely support for the service that they're paying for.

[4]https://adblockplus.org

Most of technical expertise required to use Morpheus is needed in the installation and module creation phase of the software cycle. Once Morpheus has been installed on the client machine, it is mostly maintenance-free. This means that the Software as a Service -approach can still be viable even with a huge customer base, as developer/support time is not required for the average user.

6.2.2 Key Exchange

One way to improve Morpheus would be to include logic of handling key exchange and trust networks. Instead of requiring users to perform key exchange over some secure platform (such as face to face), they could automatically exchange keys securely based on user trust. This could be done in a variety of ways, some discussed below.

6.2.2.1 Ephemeral Keys

Ephemeral keys between two users that already trust each other could be generated and cross signed. If ephemeral keys were generated for each conversation individually, there would be perfect forward secrecy for each discussion. Keys could then be destroyed after the conversation has taken place, and thus render the old conversation unreadable.

New keys could then be generated for new conversations, and this kind of key rotation could be automated by Morpheus.

6.2.2.2 Double Ratchet Algorithm

An example of an ephemeral key rotation algorithm would be the Double Ratchet Algorithm (DRA). DRA is designed to have full perfect forward secrecy, and also something called *post-compromise security*, a property which allows further secure communication even after an ephemeral key has been compromised, given that one subsequent ephemeral key is left uncompromised.

6.2.3 Message Protocols

Of course, the ephemeral key rotation does not have to be done manually if algorithms such as the OTR or Signal protocol are implemented into Morpheus instead of using PGP.

More researched protocols such as Signal already have implemented DRA [14]. While common providers of IM-Ss claim they are using the Signal protocol for end-to-end encryption[13, 5, 10, 4], much is left to be guessed about the exact implementations and details of the security of all forms of communication.

Tools such as Morpheus can effectively offload the concern of end-to-end encryption to the user, where they can configure and audit their own security and algorithm of choice.

For example, some users may only wish to use static passwords to encrypt their messages, perhaps in the lack of a better mechanism (e.g. if the recipient does not have a PPK pair), or because the recipient is not tech-savvy at all. For a human it’s much easier to say ‘the password is the name of the place where we first met’ than ‘you need to generate keys on your machine, then scan this QR code, and then we need to perform key exchange in a secure place...’ — again, it’s a matter of *how much* security a concerned user needs.

6.2.4 Message Formats

Currently Morpheus only fully supports encryption and decryption of text and binary content, but there are many other formats it could process.

6.2.4.1 Encrypted Binary Data

Binary data encrypted and decrypted using GPG is all the same, and encrypting an image, a voice message or a text message is all the same to the binary. The problem comes from integrating this functionality to the IM-P. Sending binary data as a file may work, but is hardly acceptable from the usability point of view.

For example, an image sent over a chat needs to be uploaded into the server, and this process happens entirely within the browser for security reasons. While the binary data of the image could be encrypted, it would hardly be sent as an 'image' to the IM server.

There could be a way to encrypt an image's *contents*, keeping its format still an image. Yet, when decrypting this encrypted image, it would have to pass through either complicated JavaScript or be downloaded as browsers cannot write files on the host machine.

This is the same with audio; unless simply sending encrypted 'files' that actually are audio and images, there is no easy way to integrate with the functionality that plays a voice message directly back when received.

6.2.4.2 Steganography

Another interesting idea would be to embed text into seemingly harmless images. Morpheus could be set up to choose an image from a pool of pictures, and process this image and embed the secret encrypted message into the pixels. This could be a way to achieve security through obscurity, as image content is of abundance and isn't usually suspected of containing more data than meets the eye.

6.2.5 Combating Metadata

Because users talk to each other on single IM-P, there is a lot of metadata being exchanged. It is short-sighted to discuss about encryption of data without mentioning the metadata, and there is a lot of it. For example, IM-Ps have all the metadata of who is talking to whom, when are messages being sent, the tags of a photo, and so on.

As Morpheus already interfaces with the IM-P, it could incorporate some noise to the communication. The software could send a bogus encrypted message every 5 seconds, and if there are any actual messages that the user wants to send, they would be all sent at once in place of one of those bogus messages. If the user on the other side was using Morpheus, the program could skip all the bogus messages and remove them from the DOM to keep the chat clean for actual messages. This would not disclose when the user has sent the message, and if Morpheus is left to run for prolonged periods of time, can disguise communication as noise.

6.2.6 Mobile

A big improvement to Morpheus would be to allow it to work on Mobile. The problem with this is the changed concept of Core Elements - while they can be identified in the structure of the UI, similar querying and modification mechanisms such as DOM are not readily available. Worse, there isn't unified application suites for encryption and decryption, and the hardware support of cryptographic keys and cards for the devices is not quite there. The mobile platform does offer more interesting ways of authentication, such as fingerprint scanners and face authentication, but only the future will show whether these get unified in any sound manner. Any Morpheus-like applications would have to be heavily targeted for a single platform, as technologies differ greatly

even within the same mobile operating systems.

6.3 Learning Opportunities

This project had a lot of learning opportunities to take in. From a software design side, it is difficult to predict perfectly which way a software is going to go, even if you think you have a 'full' plan. There were multiple discussions about the initial system design that I had with my friends, which made me realise once again how important white-boarding can be in designing a software system. Thankfully I did not require any major refactorings, but in the early stages I did have to change a few moving parts.

Obviously delving into implementation, even for a very simple design (See Figure 3.2) there are a lot of unforeseen considerations that need to take place. I had never developed a browser extension before, so it was an entirely new territory for me. It was never really clear how to interface with GPG until implementing it. And definitely, the User Interface took a lot of iterations to get to a comfortable level.

Having a weekly development cycle meant that I needed to carefully consider implementations that are feasible to do within that week. At first, I thought I was over-shooting my ambitions, but it turned out to be the opposite. Towards the end of the project, this switched around, and I realised I could not implement as much as I thought I could within a week. I think the take-home message for me here is that it is truly challenging, given an arbitrary feature, to predict precisely how long a software project will take. The agile approach did ensure I was always developing something, and I'm glad during the software phase I had a very short feedback loop from my supervisor.

If I was to start the project again, I would definitely spend more time thinking about each of the functionalities further, expand on the idea of Core Elements. I would definitely run through user stories to also understand different situations (Morpheus is on, but I want to send an unencrypted message; Someone has a nickname that's different from my PGP keys; etc...). For languages I would probably choose TypeScript as the front-end language as I was constantly running into typing issues when developing JavaScript.

Chapter 7

Conclusion

Years from now, I hope we will see ourselves in a future where Instant Messaging is more secure, robust and streamlined. Perhaps, if our technologies mature, we will see a unification of most common Instant Messaging Platforms into a common API, where clients could even write their own front-ends — similar to how e-mail currently works.

In this dissertation, we have shown that it is possible to create a platform-agnostic Public-Private Key (PPK) encryption/decryption wrapper on many of the modern Instant Messaging Platforms, even if those platforms do not have a source or an API available to modify upon. The fact that HTML is becoming more accessible, and people are making a group effort of unifying the APIs makes it perfect for prototyping, but still requires a lot of work to be an exact working solution across every single platform, including mobile.

Keen developers could even find a way to interface with IM-Ps on their own preferred platforms. While platform-agnostic encryption can be done in a simple way using web extensions, in theory, this should be extensible to any application given enough time and effort. Of course, the usage of encryption mechanism is not only limited to PGP, and when interfacing (free reading/writing) with Instant Messaging Platforms, any messaging protocol could potentially be used.

Evaluating the project at its end has shown that it is possible to interface with IM-Ps in a way that does not interrupt a user's workflow. This can be done very quickly in a few hundred milliseconds maximum, tying seamlessly to the rest of the platform. Of course the software is not ready to be published to all end-users, but acts as a proof-of-concept that this can be done. There is some exciting future work to be committed now that the framework has been laid out, be it more encryption methods (static password, Signal protocol, etc...) or solving problems with secure key exchange and trust networks automatically.

Software such as Morpheus enable users to not implicitly trust the platform — be it due to concerns about message encryption or the data retention — and instead use a wrapper to secure their messages on the fly without intervention. It separates the concern of end-to-end encryption from the Instant Messaging Service provider and shifts the power to the user to decide for the storage and retention of their own messages, allowing for a more secure and private conversation.

Appendix A

User Manual

A.1 Installing Morpheus

In order to install Morpheus, you will need to build the **browser extension** and **set up Icelos**.

The source code of Morpheus can be found here:

`https://gitlab.com/ilmikko/morpheus`

You can either clone the repository, or download a zip directly using this link:

`https://gitlab.com/ilmikko/morpheus/-/archive/master/morpheus-master.zip`

You will also need a working PGP setup (See Section A.1.3).

A.1.1 Building the Extension

This section describes the browser extension. If you already have Morpheus installed in your browser, please see Setting up Icelos in Section A.1.2.

You can install the browser extension as follows:

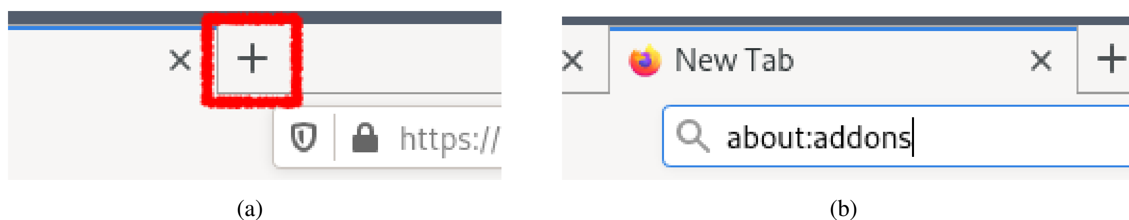
Go to the cloned Morpheus repository.

Open folder 'morpheus'.

Run 'make' in this directory. This creates a .zip file that you can open in your browser.

Open up your browser, and follow the steps for Chrome or Firefox.

A.1.1.1 Firefox



(a) (b)

Figure A.1

Create a new tab, and type in 'about:addons' to the address bar (Figure A.1).

This should take you to the 'Manage Your Extensions' page.

You could also press Ctrl+Shift+A or navigate to 'Addons' in the Firefox menu.

On the 'Manage Your Extensions' page, click the cogwheel icon on the top right corner (Figure A.2).

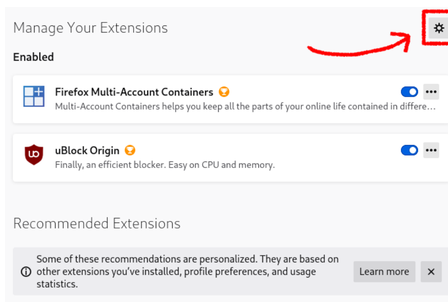


Figure A.2

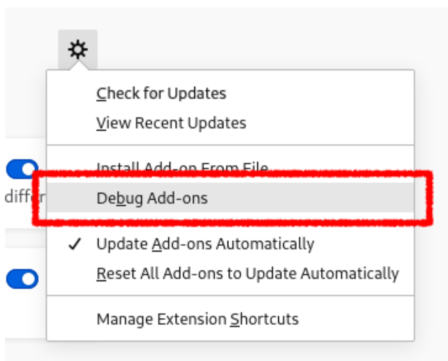


Figure A.3

A drop down menu appears.

Click on the item ‘Debug Add-ons’ (Figure A.3).

Do NOT click on ‘Install Add-on From File’ as this does not currently work.

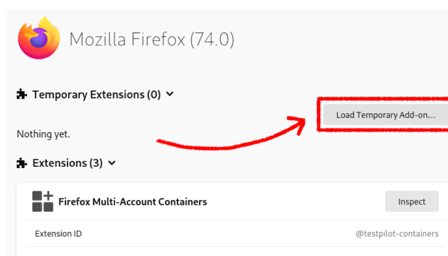


Figure A.4

After clicking ‘Debug Add-ons’, you should see a menu for Temporary Extensions.

Click ‘Load Temporary Add-on’ on the top right corner (Figure A.4).

Navigate to the zip file you created under Building the extension (Section A.1.1), and open it (Figure A.5).

Morpheus should now be installed in your browser (Figure A.6)!

A.1.1.2 Chrome

Create a new tab, and type in ‘about:extensions‘ or ‘chrome:extensions‘ to the address bar (Figure A.7).

This should take you to the ‘Extensions’ page.

Figure A.5

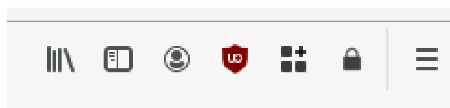


Figure A.6

You could also navigate to 'Extensions' in Chrome settings.

Enable 'Developer Mode' on the top right corner (Figure A.8).

This will display an additional menu on the top left.

Click 'Load unpacked' in the top left menu (Figure A.9).

Navigate to the extension folder you created under Building the extension (Section A.1.1), and open the folder (Figure A.10).

Morpheus should now be installed in your browser (Figure A.11)!

A.1.2 Setting up Icelos

A.1.2.1 Basic Docker Setup

The easiest way to set up Icelos is to install Docker on your computer.

You can then build the Dockerfile by doing as follows:

Go to the cloned Morpheus repository.

Run `docker-compose up` in this directory.

Icelos will start running on the specified port.

A.1.2.2 Advanced Setup

If your PGP setup is more elaborate, you can also have Icelos run directly on the host machine and connect to GPG.

This is usually required if your PGP setup greatly varies from the out-of-the-box setup, and/or if you're using smartcards or similar.

You should have Go and Make installed.

Simply go to the cloned Morpheus repository, open the folder called 'icelos', and build the binary using 'make'.

Afterwards, you can run Icelos directly any time you need to use Morpheus, or keep it running using the provided Systemd daemon.

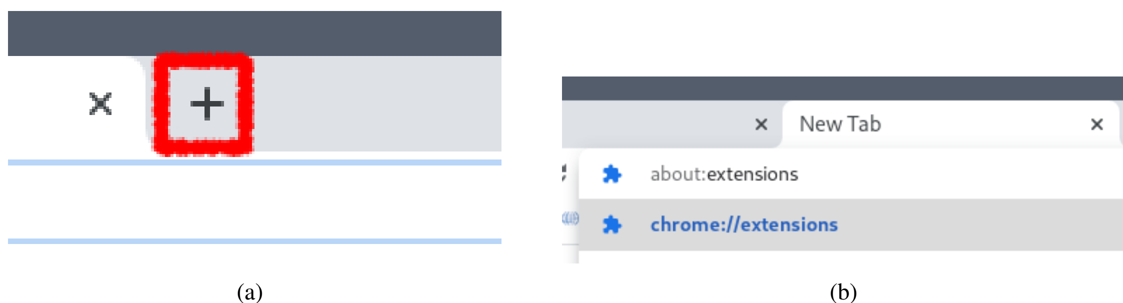
A.1.3 Setting up PGP

Morpheus uses GPG to encrypt and decrypt PGP messages.

There are many great resources online for GPG, for example you can follow this tutorial to get started: `https://www.gnupg.org/gph/en/manual/c14.html`

The most important point is that you should **never ever share your private key with anyone**.

Nobody should be telling you to use a private key they generated - this is very unsecure.



(a) (b)

Figure A.7

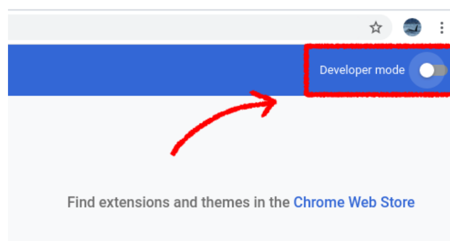


Figure A.8

Once you have GPG installed, you should generate a public-private key pair and keep the private key very private.

Other people will use your **public key** to encrypt messages to you. In order to tell people about your public key, you need to exchange the keys securely (see Section A.1.4).

Once you start receiving messages encrypted with your public key, you can then use your **private key** to decrypt them.[1]

A.1.4 Exchanging Keys Securely

This is something Morpheus cannot do due to the laws of cryptography.

If you want to exchange keys with someone, you should do that over a secure channel, or off-line.

The best way to securely exchange keys is to meet in person. This eliminates any man-in-the-middle attacks or impersonation attempts.[2]

A.1.5 Frequently Asked Questions

A.1.5.1 Morpheus encountered an unexpected error

For error messages, hover your mouse over the error icon for more information.

A.1.5.1.1 Connection refused to Icelos on port XXXX This means that Morpheus is failing to find Icelos that it requires to encrypt/decrypt PGP messages.

Please revise the steps in Setting up Icelos in Section A.1.2 to resolve the issue, paying close attention to the port numbers.

[1] For more information about the process, you can read this article: `https://www.khanacademy.org/computing/ap-computer-science-principles/the-internet/tls-secure-data-transport/a/public-key-encryption`

[2] You can read more about secure key exchange in `https://ssd.eff.org/en/module/deep-dive-end-end-encryption-how-do-public-key-encryption-systems-work#4`.

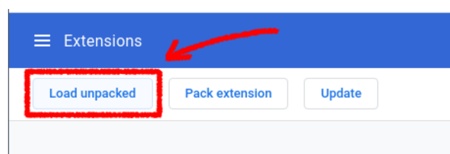


Figure A.9

Figure A.10

If the port number for Morpheus (1234) is not the same as the port number of Icelos (1234), then Morpheus will not be able to find Icelos.

A.1.5.1.2 Missing host permission for the tab This means that Morpheus cannot encrypt or decrypt messages on the current tab.

Morpheus may still work on other tabs, but there are some tabs (such as the extensions page) that cannot be modified by Morpheus.

A.1.5.2 I have received an encrypted message.

The message should look something akin to Figure A.12.

In order to decrypt a message encrypted with Morpheus, you don't need Morpheus (although it will make your life easier).

Follow the sections below depending on whether you want to decrypt manually, or using Morpheus.

A.1.5.2.1 Decrypting Using Morpheus If you do not have Morpheus installed, refer to Installing Morpheus first.

Once you have installed Morpheus, you should be able to decrypt any PGP or Morpheus messages automatically, or by pressing a lock icon next to the message.

If you get decryption errors, refer to the FAQ in Section A.1.5.

A.1.5.2.2 Decrypting Manually Morpheus messages are PGP encrypted. They follow a slightly different format due to format limitation of some chat platforms.

If you do not have an active PGP setup, then you should be suspicious - the message is probably not encrypted for you.

In order to set up PGP, please refer to Setting up PGP in Section A.1.3, and afterwards ask the sender to re-send their message using your public key.

The only difference is in the start and end tokens of the messages (Figure A.13).

Morpheus messages start with `Encrypted:[` and end with a single `]`.

The PGP data lays in the middle of these two tokens.

You can start by copying and pasting the message into your favourite text editor (Figure A.14).

Move the start and end tokens on their own lines so you won't accidentally chop off any important data (Figure A.15).

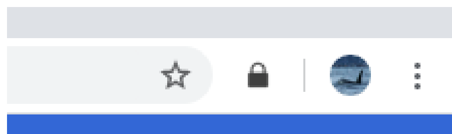


Figure A.11

Encrypted:[hQEMA0GZmT377BIBAQf/RBXaVyvk2cQGlIU4EXjHe30lPwGVhYOLCU4cSyGuuzGS
kImKq1t1CYA/CCstDF/2aE8ABFPeq36XGsW/W4qWu90ZCrPEuCvaNrimU/3gjLa1
22DkF8TMnQs4+WtYIyqD3AHtpOIuPR7f/J2Y9T6kLf9MC+FMJXp9j20laN4MWegA
amctX5xrNFWlBG16vDW3ApvFt4t4mmFBrinPrrxpVubutjn+jw11+DGaYn2bX5mB
B0vSiYzoPd/GXU2NiVLJVhiLuGRxPnklUo+NuPXAyZwz7jNCKTrkMOCCk8qf2Vag
LKP3XAAqlvT0qpPrbV8N6tLKxngm8HkCjNFhkIyW0tJAAcgTy05DYeqn9jFUBEb+
7fx9htn0p0TwsUpF1vEvmUrsAEL15uCSUR3uikmGsw3yVfiDeabv1Kl4v3qNodML
5g==
=5CE+]

Figure A.12

Then, replace the start token `Encrypted:[` with `-----BEGIN PGP MESSAGE-----`.
Replace the end token `]` with `-----END PGP MESSAGE-----`.
Take care as PGP is very particular about the format of these tokens (Figure A.16).
That's it, you now have a valid PGP message!
You can now, depending on your PGP setup, decrypt this message.
Below is a command-line example.

```
$ gpg --decrypt message.txt
gpg: encrypted with 2048-bit RSA key, created 2020-02-04
(y)
```

It is, of course, quite cumbersome to do this for every message that you receive.
If you end up doing this a lot you might want to look into installing Morpheus in Section A.1.

Encrypted:[hQEMA0GZmT377BIBAQf/RBXaVyvk2cQGlIU4EXjHe30lPwGVhYOLCU4cSyGuuzGS
kImKq1t1CYA/CCstDF/2aE8ABFPeq36XGsW/W4qWu90ZCrPEuCvaNrimU/3gjLa1
22DkF8TMnQs4+WtYIyqD3AHtpOIuPR7f/J2Y9T6kLf9MC+FMJXp9j20laN4MWegA
amctX5xrNFWlBG16vDW3ApvFt4t4mmFBrinPrrxpVubutjn+jw11+DGaYn2bX5mB
B0vSiYzoPd/GXU2NiVLJVhiLuGRxPnklUo+NuPXAyZwz7jNCKTrkMOCCk8qf2Vag
LKP3XAAqlvT0qpPrbV8N6tLKxngm8HkCjNFhkIyW0tJAAcgTy05DYeqn9jFUBEb+
7fx9htn0p0TwsUpF1vEvmUrsAEL15uCSUR3uikmGsw3yVfiDeabv1Kl4v3qNodML
5g==
=5CE+]

Figure A.13

Encrypted:[hQEMA0GZmT377BIBAQf/RBXaVyvk2cQGlIU4EXjHe30lPwGVhYOLCU4cSyGuuz
GS
kImKq1t1CYA/CCstDF/2aE8ABFPeq36XGsW/W4qWu90ZCrPEuCvaNrimU/3gjLa1
22DkF8TMnQs4+WtYIyqD3AHtpOIuPR7f/J2Y9T6kLf9MC+FMJXp9j20laN4MWegA
amctX5xrNFWlBG16vDW3ApvFt4t4mmFBrinPrrxpVubutjn+jw11+DGaYn2bX5mB
B0vSiYzoPd/GXU2NiVLJVhiLuGRxPnklUo+NuPXAyZwz7jNCKTrkMOCCk8qf2Vag
LKP3XAAqlvT0qpPrbV8N6tLKxngm8HkCjNFhkIyW0tJAAcgTy05DYeqn9jFUBEb+
7fx9htn0p0TwsUpF1vEvmUrsAEL15uCSUR3uikmGsw3yVfiDeabv1Kl4v3qNodML
5g==
=5CE+]

Figure A.14

Encrypted:[
hQEMA0GZmT377BIBAQf/RBXaVyvk2cQGlIU4EXjHe30lPwGVhYOLCU4cSyGuuzGS
kImKq1t1CYA/CCstDF/2aE8ABFPeq36XGsW/W4qWu90ZCrPEuCvaNrimU/3gjLa1
22DkF8TMnQs4+WtYIyqD3AHtpOIuPR7f/J2Y9T6kLf9MC+FMJXp9j20laN4MWegA
amctX5xrNFWlBG16vDW3ApvFt4t4mmFBrinPrrxpVubutjn+jw11+DGaYn2bX5mB
B0vSiYzoPd/GXU2NiVLJVhiLuGRxPnklUo+NuPXAyZwz7jNCKTrkMOCCk8qf2Vag
LKP3XAAqlvT0qpPrbV8N6tLKxngm8HkCjNFhkIyW0tJAAcgTy05DYeqn9jFUBEb+
7fx9htn0p0TwsUpF1vEvmUrsAEL15uCSUR3uikmGsw3yVfiDeabv1Kl4v3qNodML
5g==
=5CE+
]

Figure A.15

-----BEGIN PGP MESSAGE-----
hQEMA0GZmT377BIBAQf/RBXaVyvk2cQGlIU4EXjHe30lPwGVhYOLCU4cSyGuuzGS
kImKq1t1CYA/CCstDF/2aE8ABFPeq36XGsW/W4qWu90ZCrPEuCvaNrimU/3gjLa1
22DkF8TMnQs4+WtYIyqD3AHtpOIuPR7f/J2Y9T6kLf9MC+FMJXp9j20laN4MWegA
amctX5xrNFWlBG16vDW3ApvFt4t4mmFBrinPrrxpVubutjn+jw11+DGaYn2bX5mB
B0vSiYzoPd/GXU2NiVLJVhiLuGRxPnklUo+NuPXAyZwz7jNCKTrkMOCCk8qf2Vag
LKP3XAAqlvT0qpPrbV8N6tLKxngm8HkCjNFhkIyW0tJAAcgTy05DYeqn9jFUBEb+
7fx9htn0p0TwsUpF1vEvmUrsAEL15uCSUR3uikmGsw3yVfiDeabv1Kl4v3qNodML
5g==
=5CE+
-----END PGP MESSAGE-----

Figure A.16

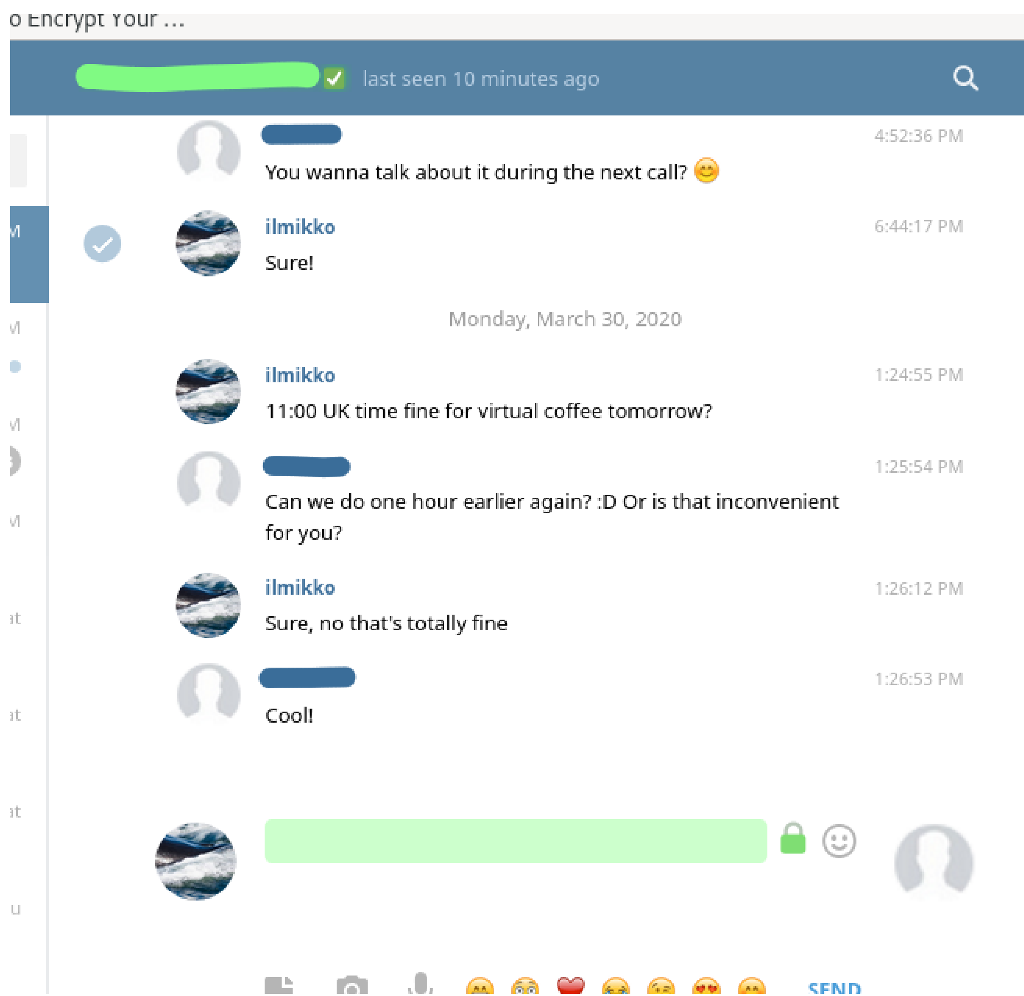


Figure A.17: Screenshot of Morpheus working on Telegram Web.

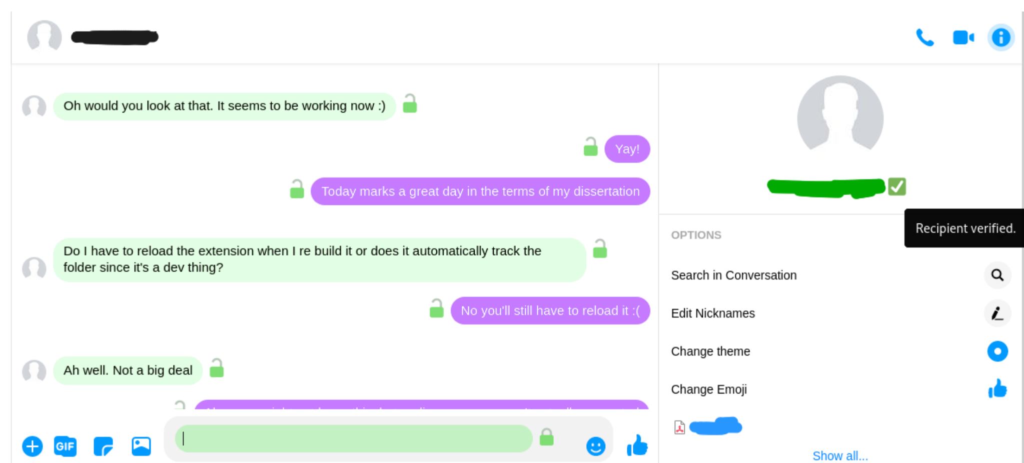


Figure A.18: Screenshot of Morpheus working on Facebook Messenger.

Appendix B

Maintenance Manual

B.1 Software Requirements

Morpheus requires the following software for best results:

- Golang 1.14.2
- Docker 19.03.8-ce, build afacb8b7f0
- Browser that supports the WebExtensions API, such as:
  - Google Chrome 81.0.4044.113
  - Mozilla Firefox 75.0

B.2 Installation

Please refer to Section A.1 for detailed installation instructions.

The source code of Morpheus can be found here: `https://gitlab.com/ilmikko/morpheus`

B.3 Implementing Modules

In order to implement a module for Morpheus, you will first need to open up an Instant Messaging Platform (IM-P) that you want to interface with.

Take note of the URL on the top of the page and copy this into `morpheus/common/site-detector.js`. Create a simple ID that site-detector will return, that doesn't clash with existing module IDs.

Then, create a new file into `morpheus/module/<ID>.js`. Inside this file, paste the following template:

```
if (SiteDetector.Current() == '<INSERT YOUR ID HERE>')
        browser.runtime.sendMessage({
                command: 'module',
                customCSS: ``,
                bindInputs: [
                        // TODO
                ],
                messageFeeds: [
                        // TODO
                ],
                messageElement: '', // TODO
                messageText: '', // TODO
                recipientHints: [
                        // TODO
                ],
                resetters: '' // TODO
        });
```

Now, identify each of the Core Elements (defined in Section 3.2.5) in the IM-P and note down their DOM query. You can easily do this by opening up the Browser Console using F12, and in the console typing in `document.querySelector("<YOUR QUERY>")` and seeing if this returns the expected element.

After filling all of the Core Elements, recreate the Morpheus Web Extension and check that you have the functionality you require. You may need to tweak some settings for best results (see other modules as example).

B.4 Project Tree

B.4.1 Root

```
how-to -- User Instructions
├── icelos -- Source code for Icelos
├── morpheus -- Source code for Morpheus Browser Extension
├── morpheus-test -- Testing and evaluation code
│   ├── index.html -- Contains a mock IM-P to test with
│   └── test-server.go -- Simple HTTP server for testing
├── README.md -- Information about the project
└── docker-compose.yml -- Instructions for installation on Docker
```

B.4.2 Morpheus

```
background -- Source code for the Background Script (Section 3.2.6.1)
│   ├── icelos.js -- Code for interfacing with Icelos
│   ├── morpheus.js -- Main logic for Morpheus Browser Extension
│   └── tabdata.js -- Logic for keeping data between Morpheus instances within
│       tabs separate
├── common -- Source code shared across scripts
│   ├── constants.js -- Constants such as port numbers
│   ├── convert.js -- Conversion between PGP and Morpheus encryptions (Section
│   │   4.2.8)
│   ├── debounce.js -- Debounce function[a]
│   ├── http.js -- HTTP into Promise wrapper (Section 4.2.2)
│   ├── polyfill.js -- Polyfill for nonconforming browsers (Section 4.2.3)
│   ├── send.js -- Code for unified message passing between modules
│   └── site-detector.js -- Code for detection of sites for modules (Section
│       4.2.5)
├── img -- Image content
├── inject -- Source code for the Inject Script (Section 3.2.6.3)
│   ├── element.js -- Code for manipulating DOM elements
│   ├── element_picker.js -- Code for DOM element queries
│   ├── element_store.js -- Code for storing/retrieving original and cloned
│   │   DOM elements (Section 4.2.7)
│   ├── guard.js -- Code to ensure a single run on multiple injects
│   ├── guard_end.js -- Ditto
│   ├── inject.css -- Styling rules for injected DOM elements
│   ├── morpheus.js -- Injected portion of Morpheus
│   └── recipient.js -- Code for keeping track of recipients
├── module -- Source code for all Modules
├── popup -- Source code for the Popup Script (Section 3.2.6.2)
│   ├── default -- Logic for when the extension icon is clicked
│   ├── settings -- Popup logic for the settings page
│   ├── status -- Popup logic for the status page (Section 4.5.2)
│   └── welcome -- Popup logic for the welcome/setup page
├── Dockerfile -- Instructions for Docker to build Morpheus
├── make.extension.sh -- Build script for creating the extension
├── makefile -- Build rules
└── README.md -- Information about the Morpheus Browser Extension
```

[a] https://medium.com/@jamischarles/what-is-debouncing-2505c0648ff1

B.4.3 Icelos

```
│ config -- Default configuration module
├── icelos -- Source code for the Icelos Go module
│   ├── gpg -- Source code for interfacing with GPG
│   │   ├── gpg.go -- Code for encrypt/decrypt using GPG
│   │   └── keys.go -- Code for key querying and retrieval using GPG
│   ├── encrypt.go -- Code for Icelos encryption
│   ├── decrypt.go -- Code for Icelos decryption
│   ├── icelos.go -- Code for HTTP server and Icelos module
│   └── keys.go -- Code for Icelos key querying
├── icelos.go -- Icelos Go main package
├── Dockerfile -- Instructions for Docker to build Icelos
├── icelos.service -- Icelos Systemd service file
├── makefile -- Build rules
└── README.md -- Information about Icelos
```

Bibliography

[1] Bernstein, D. (2016). A state-of-the-art diffie-hellman function. https://cr.yp.to/ecdh.html Last Accessed: 2020-04-09.

[2] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., and Stebila, D. (2017). A formal security analysis of the signal messaging protocol. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 451–466. IEEE.

[3] Ermoshina, K., Musiani, F., and Halpin, H. (2016). End-to-end encrypted messaging protocols: An overview. In *International Conference on Internet Science*, pages 244–254. Springer.

[4] Greenberg, A. (2016a). With allo and duo, google finally encrypts conversations end-to-end. https://www.wired.com/2016/05/allo-duo-google-finally-encrypts-conversations-end-end/ Last Accessed: 2020-05-01.

[5] Greenberg, A. (2016b). You can all finally encrypt facebook messenger, so do it. https://www.wired.com/2016/10/facebook-completely-encrypted-messenger-update-now/ Last Accessed: 2020-05-01.

[6] Herra-Vega, F. (2017). The new peerio: A technical deep dive. https://web.archive.org/web/20181107124811/https://www.peerio.com/blog/posts/the-new-peerio-a-technical-deep-dive/ Last Accessed: 2020-03-16.

[7] Jonathan Christensen, P. Z. (2019). Wire security whitepaper. https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf Last Accessed: 2020-04-09.

[8] Kobeissi, N. (2019). Cryptocat - security. https://web.archive.org/web/20190228095608/https://crypto.cat/security.html Last Accessed: 2020-03-16.

[9] Lee, M., Grauer, Y., Lee, M., and Grauer, Y. (2020). Zoom meetings aren't end-to-end encrypted, despite misleading marketing. https://theintercept.com/2020/03/31/zoom-meeting-encryption/ Last Accessed: 2020-04-09.

[10] Lund, J. (2018). Signal partners with microsoft to bring end-to-end encryption to skype. https://signal.org/blog/skype-partnership/ Last Accessed: 2020-05-01.

[11] Marshall, P. (2018). When is a back door not a back door? https://gcn.com/articles/2018/01/16/fbi-encryption-backdoor.aspx Last Accessed: 2020-03-16.

[12] McMahon, J. (2017). Why we should all ditch other messaging apps for signal. https://www.wired.com/story/ditch-all-those-other-messaging-apps-heres-why-you-should-use-signal/ Last Accessed: 2020-03-20.

[13] Metz, C. (2016). Forget apple vs. the fbi: Whatsapp just switched

on encryption for a billion people. https://www.wired.com/2016/04/forget-apple-vs-fbi-whatsapp-just-switched-encryption-billion-people/ Last Accessed: 2020-05-01.

[14] Perrin, T. and Marlinspike, M. (2016). The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf Last Accessed: 2020-05-01.

[15] Price, R. (2015). David cameron wants to ban encryption. https://www.businessinsider.com/david-cameron-encryption-apple-pgp-2015-1?r=UK Last Accessed: 2020-03-16.

[16] Robison, C., Ruoti, S., van der Horst, T. W., and Seamons, K. E. (2012). Private facebook chat. In *2012 International Conference on Privacy, Security, Risk and Trust and 2012 International Conference on Social Computing*, pages 451–460. IEEE.

[17] Sun, Y., Liu, D., Chen, S., Wu, X., Shen, X.-L., and Zhang, X. (2017). Understanding users' switching behavior of mobile instant messaging applications: An empirical study from the perspective of push-pull-mooring framework. *Computers in Human Behavior*, 75:727 – 738.

[18] Sutikno, T., Handayani, L., Stiawan, D., Riyadi, M. A., and Subroto, I. M. I. (2016). Whatsapp, viber and telegram: Which is the best for instant messaging? *International Journal of Electrical & Computer Engineering (2088-8708)*, 6(3).

[19] Vaas, L. (2018). Fbi: we don't want a backdoor; we just want you to break encryption. https://nakedsecurity.sophos.com/2018/03/12/fbi-we-dont-want-a-backdoor-we-just-want-you-to-break-encryption/ Last Accessed: 2020-03-16.

[20] Wagner, K. (2018). Here's how facebook allowed cambridge analytica to get data for 50 million users. https://www.vox.com/2018/3/17/17134072/facebook-cambridge-analytica-trump-explained-user-data Last Accessed: 2020-03-16.

[21] Weinberger, M. (2014). Matrix wants to smash the walled gardens of messaging. https://www.itworld.com/article/2694500/matrix-wants-to-smash-the-walled-gardens-of-messaging.html Last Accessed: 2020-03-20.

[22] Whited, S. (2019). Xep-0364: Current off-the-record messaging usage. https://xmpp.org/extensions/xep-0364.html#overview Last Accessed: 2020-04-09.

[23] Winder, D. (2019). Unsecured facebook databases leak data of 419 million users. https://www.forbes.com/sites/daveywinder/2019/09/05/facebook-security-snafu-exposes-419-million-user-phone-numbers/#39b5595a1ab7 Last Accessed: 2020-03-16.