BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

GPG ENCRYPTED WEB PAGES
ČÁSTI WEBOVÉ STRÁNKY ŠIFROVANÉ POMOCÍ GPG

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR
AUTOR PRÁCE

Bc. JIŘÍ MATĚJKA

SUPERVISOR
VEDOUCÍ PRÁCE

Ing. LIBOR POLČÁK, Ph.D.

BRNO 2020

Department of Information Systems (DIFS)   Academic year 2019/2020

Master's Thesis Specification

22307

Student: **Matějka Jiří, Bc.**

Programme: Information Technology   Field of study: Information Technology Security

Title: **GPG Encrypted Web Pages**

Category: Web

Assignment:

1. Get familiar with the GnuPG project.
2. Learn how to write WebExtensions.
3. Design an extension for the Firefox browser that allows to decrypt parts of a web page. Propose a suitable approach that detects encrypted objects. Propose a solution that is able to detect interactive changes to the page performed employing XHR API, Fetch API, and Push API.
4. Develop the extension with support for at least GNU/Linux.
5. Test the extension.
6. Publish the extension on addons.mozilla.org, evaluate the results and propose possible future directions for further development of the extension.

Recommended literature:

- Mueller: Security for Web Developers: Using JavaScript, HTML, and CSS. ISBN 978-1-491-92864-6, O'Reilly, 2016.
- The GNU Privacy Guard. The GnuPG Project. Available online https://gnupg.org/.

Requirements for the semestral defence:

- Points 1 to 3 of the specification.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor: **Polčák Libor, Ing., Ph.D.**

Head of Department: Kolář Dušan, doc. Dr. Ing.

Beginning of work: November 1, 2019

Submission deadline: June 3, 2020

Approval date: January 14, 2020

Abstract

The aim of the thesis is to design and implement a solution to secure sensitive data on public servers or third-party servers. The thesis deals with an implementation of a web browser extension for Mozilla Firefox browser, that will be able to detect and decrypt encrypted objects on a web page. The extension must be able to process dynamic changes on the web page caused by the usage of XHR API, Fetch API or Push API. The last but not least, the thesis deals with the testing of the implemented solution and measuring its influence on the performance of the browser.

Abstrakt

Cílem této práce je navrhnout a implementovat způsob zabezpečení citlivých dat na veřejných serverech nebo serverech třetích stran. Práce se zabývá implementací rozšíření pro webový prohlížeč Mozilla Firefox, které bude schopno nalézt a dešifrovat zašifrované prvky webové stránky s využitím výstupů GnuPG projektu. Rozšíření musí být dále schopno zpracovat dynamické změny webové stránky způsobené použitím XHR API, Fetch API, či Push API. V neposlední řadě se práce zabývá testováním implementovaného řešení a měření vlivu rozšíření na celkovou dobu zpracování webových stránek prohlížečem.

Keywords

GPG, GnuPG, OpenPGP, cryptography, WebExensions

Klíčová slova

GPG, GnuPG, OpenPGP, kryptografie, WebExensions

Reference

MATĚJKA, Jiří. *GPG Encrypted Web Pages*. Brno, 2020. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Libor Polčák, Ph.D.

GPG Encrypted Web Pages

Declaration

I declare that I have carried out the Master's thesis independently under supervision of *Ing. Libor Polčák, Ph.D.* from *Brno University of Technology*, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or processional qualification.

....

Jiří Matějka

July 24, 2020

Acknowledgements

Contents

Chapter 1

Indroduction

The most popular information resource today is undoubtedly the internet. One of its key advantages is data availability. Frequently, data are stored on remote devices (commonly servers) and users can connect to these devices and access data they require. These data can contain private or secret information such as family pictures, passwords, bills or other sensitive content that need to be protected. Access to a server with this kind of sensitive information can be protected by some kind of authorization (for example with a login and a password). But even the most secure kind of authorization is not sufficient enough to secure data from an unauthorized access. For a user, to obtain any type of content on a remote device, data must be transferred. In the case of the internet, data are transferred over multiple devices on which data can be accessed or even modified by a potential attacker (without the knowledge of either side of a communication). The figure 1.1 shows such possible unauthorised data modification.
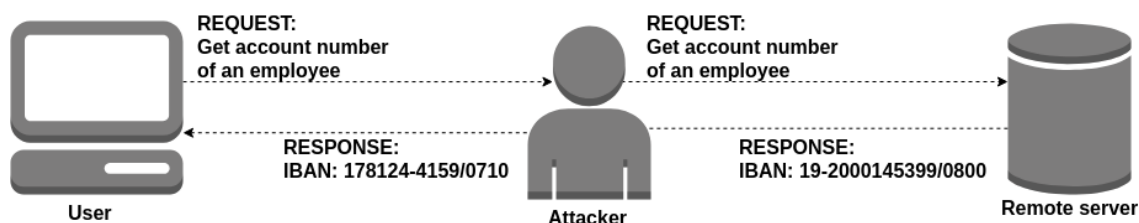
Figure 1.1: Example of unauthorised data access and modification during its transmission.

Usage of cryptography is the most frequently used solution for this problem. Data can be encrypted during the communication or encrypted data can be stored on servers and then transferred with or without further encryptions. This thesis will use the second approach, where data are already encrypted on a remote server. Under the term of data, you can imagine usual web page including not only a text, images or videos but JavaScript and CSS as well. Some elements of this page (or even whole page) can be encrypted using a symmetric cypher and different parts can be encrypted using a different key or some parts can be encrypted using multiple keys. These keys are encrypted with an asymmetric cypher and they are a part of encrypted content as well. The outcome of the thesis will be a web browser extension that will be able

to detect an encrypted content (images, videos, text, etc.) and decrypt it for a user using his available keys. With this encryption/decryption system, users can create web pages where different data can be accessible for different users without the need for authentication on a remote server.

The target platform will be GNU/Linux and browser extensions will be implemented for Firefox web browser. Data will be decrypted with the Linux command line application called gpg, that will be used not only for decryption but for key management as well.

Chapter 2

The GNU Privacy Guard Project

This chapter provides information about *The GNU Privacy Guard Project*. The text is divided into the following sections:

1. The GNU Privacy Guard Project,

2. OpenPGP standard,

3. GnuPG for Linux distributions,

4. Alternative Software.

Information provided by sections *The GNU Privacy Guard Project* and *GnuPG for Linux distributions* are a summarization of material published on *The GNU Privacy Guard Project* website [4]. The section dealing with the *OpenPGP standard* is a summarization of a large standard definition by RFC4880 [1].

2.1   The GNU Privacy Guard Project

The GNU Privacy Guard Project, also known as GnuPG or GPG, is a complete and free implementation of the *OpenPGP* standard as defined by RFC4880 [1]. The GnuPG offers encryption, decryption and signing both data and communication. It features a versatile key management system with access modules for many kinds of public key directories. Not only that the GnuPG is available for both Windows and Linux operating system, but also a wealth of applications and libraries are available.

The Linux implementation of the GnuPG is a command line tool with features for integration with other applications. The Windows version of the GnuPG is Gpg4win with a context menu tool, a crypto manager and an Outlook plugin to send and receive standard PGP/MIME mails.

2.2   OpenPGP standard

As mentioned earlier, the GnuPG is the implementation of the *OpenPGP* standard. The text of this section summarizes the *OpenPGP* standard definition provided by RFC4880 [1]. The *OpenPGP* combines symmetric–key encryption and public–key encryption to

provide confidentiality. First of all, the object is encrypted using a symmetric encryption algorithm. It is worth mentioning that each symmetric key is used only once for a single object. For each object, a new key is generated as a random number. This key is bound to the message and transmitted with it. Key is protected by encryption as well – the key is encrypted with the receiver's public key. The sequence is as follows (also described in the figure 2.1):

1. A message is created by the sender.

2. The sending *OpenPGP* generates a random number to be used as a session key for this message only.

3. The generated session key is encrypted using recipient's public key. This encrypted session key starts the message.

4. The sending *OpenPGP* encrypts the message using the session key, which forms the remainder of the message. Note that the message is also usually compressed.

5. The receiving *OpenPGP* decrypts the session key using the recipient's private key.

6. The receiving *OpenPGP* decrypts the message using the session key. If the message was compressed, it will be decompressed.
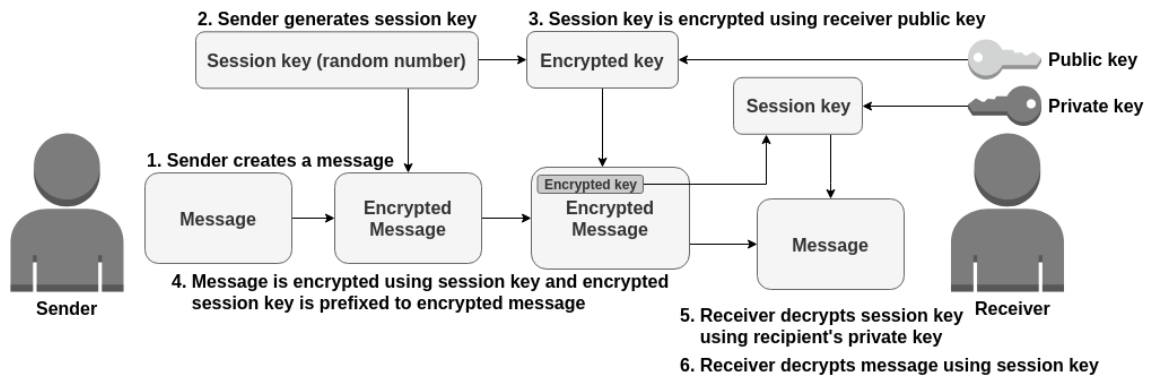
Figure 2.1: Schema of message encryption and decryption

The symmetric key, that is used for message encryption, can be derived from a passphrase (or different kind of shared secret), or a two–stage mechanism similar to the public–key method that was described above, in which a session key itself is encrypted with a symmetric algorithm keyed from a shared secret.

Authentication can be achieved using a digital signature. The digital signature uses a hash code or a message digest algorithm and a public–key signature algorithm. The sequence is as follows (also described in the figure 2.2):

1. A message is created by the sender.

2. The sending software generates a hash code of the message.

3. The sending software generates a signature by encrypting hash code of message using the sender's private key.

4. The binary signature is attached to the message.

5. The receiving software keeps a copy of the message signature.

6. The receiving software generates a new hash code for the received message and verifies it using the message's hash code obtained by decrypting signature with sender's private key.
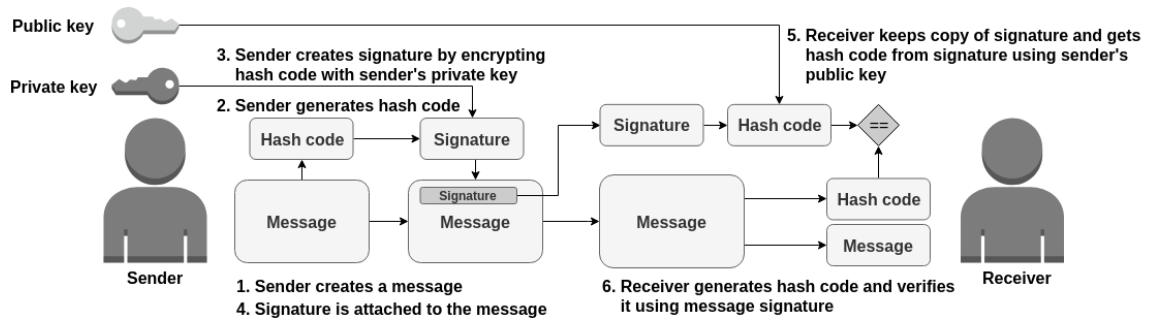
Figure 2.2: Schema of message signature

Both confidentiality and signature services may be applied to the same message. First, a signature is created and attached to the message. Then the message (including the signature) is encrypted using a symmetric session key. At last, the session key is encrypted using the receiver's public–key and prefixed to the encrypted message.

2.3 GnuPG for Linux distributions

As described above, there is a Linux application called gpg. It is important to have this application installed and configured, otherwise implemented software in this thesis will not work correctly, or will not work at all. The reason for this is a fact, that GnuPG is not only used for decryption of encrypted elements but solves problems of the key management as well. Furthermore, the gpg application can be used for encryption of web pages.

Installation of the GnuPG may differ on different operating systems. Some of the GNU/Linux distribution may already come with directly installable packages. However, it is worth considering installation from the source code because the version of these packages may be old. The list of different GnuPG packages, libraries, required tools, optional software, legacy versions of GnuPG or manual can be found on *The GNU Privacy Guard Project* web page [4].

2.3.1 Basic Key Management

Since objects are signed using the receiver's public key and decrypted with the receiver's private key, it is clear that the receiver must have this keypair. The keypair

is not only needed to encrypt the messages but also to sign them. With gpg installed on user's machine, the user can generate its own private and public keys. Basic key management will be described in these steps:

1. Generating a new pair of private and public key.

2. Importing and exporting of both private and public keys.

Generating a New Pair of Private and Public Key

To generate keypair with gpg, the user must complete several steps. The first step is to lunch gpg application with the argument *--gen-key*. To prevent problems with private key access permissions, the user can specify a directory, where keys will be stored with argument *--homedir*. Detailed instructions, how to manage user's access to private keys, are described in the article about GPG encrypted credentials by Fabian Lee [5]. Next, the user is asked for a name and an email address. From the name and the email address, gpg creates a User ID. The User ID is something like a name tag for the generated keypair and is also used to identify an owner of a public key.

```
xmatej52@merlin: ~$ gpg --gen-key
Real name: Jiří Matějka
E-mail address: xmatej52@stud.fit.vutbr.cz
You are using the 'utf-8' character set.
You selected this USER-ID:
    "Jiří Matějka <xmatej52@stud.fit.vutbr.cz>"

Change (N)ame, (E)mail, or (O)kay/(Q)uit?
```

Then, the user is asked to enter a passphrase. The passphrase is used to encrypt the private key so it is protected. If the passphrase is compromised, anyone who can access such private key will be able to decrypt owner's received messages and sign his messages as the owner of the private key. After the user enters the passphrase, gpg need to generate a lot of random bytes and ask the user to perform some other actions. After some time, a new keypair is finally generated.

```
Enter passphrase:
We need to generate a lot of random bytes. It is a good idea to perform
some other action (type on the keyboard, move the mouse, utilise the
disks) during the prime generation; this gives the random number
generator a better chance to gain enough entropy.

gpg: key 6C9359504F0C8F81 marked as ultimately trusted
gpg: revocation certificate stored as '/path/to/file'
public and secret key created and signed.

pub   rsa3072 2019-12-30 [SC] [expires: 2021-12-29]
      3252EA3A9A0F105E226BE7BF6C9359504F0C8F81
```

```
uid                           Jiří Matějka <xmatej52@stud.fit.vutbr.cz>
sub   rsa3072 2019-12-30 [E] [expires: 2021-12-29]
```

Importing and Exporting of Both Private and Public Keys

Export of public keys is an essential feature but in some cases, it is required to export a
private key as well. The GnuPG provides options for exporting both of the keys. While
the export of a public key is not a problem, the export of a private key may require
extra permissions (for example to run gpg via *sudo*), if the logged user does not have
permissions to access the private key. To see a list of accessible keys, use command
gpg --list-public-keys for public keys, and *gpg --list-secret-keys* for private keys.

```
xmatej52@merlin: ~$ gpg --list-secret-keys
/path/to/dir/pubring.kbx
--------------------------------
sec   rsa3072 2019-08-11 [SC] [expires: 2021-08-10]
      346222BC9BEF6635994EF30DC7C029E805D6F30A
uid           [ultimate] test <test@test.cz>
ssb   rsa3072 2019-08-11 [E] [expires: 2021-08-10]


sec   rsa3072 2019-12-30 [SC] [expires: 2021-12-29]
      3252EA3A9A0F105E226BE7BF6C9359504F0C8F81
uid           [ultimate] Jiří Matějka <xmatej52@stud.fit.vutbr.cz>
ssb   rsa3072 2019-12-30 [E] [expires: 2021-12-29]
```

The user ID is shown as *uid* in the list. The user ID can be used to specify which
key should be exported. The command *gpg --armor --export uid* will export the public
key specified by *uid*. To export the private key, use arguments *--armor* and
--export-secret-keys uid. Since private keys are encrypted, the user will be asked to
enter the passphrase that was entered while generating the keypair.

```
xmatej52@merlin: ~$ gpg --armor --export xmatej52@stud.fit.vutbr.cz
-----BEGIN PGP PUBLIC KEY BLOCK-----

mQGNBF4J/jYBDAC2xruHMLrLn2SwV7n2N5dBkjVO4Jxu//BGjHNqbylcyOOaUT7l
.....
-----END PGP PUBLIC KEY BLOCK-----
xmatej52@merlin: ~$ gpg --armor --export-secret-keys \
xmatej52@stud.fit.vutbr.cz
-----BEGIN PGP PRIVATE KEY BLOCK-----

nQWGBF00HzUBDADApYD+T6kx0Lb0h0MyB1CrWRmrINqHLQqQmzLiaABDLAYqvhgB
.....
-----END PGP PRIVATE KEY BLOCK-----
```

2.3.2 Encrypting, Decrypting, Signing, and Verifying Data

The GnuPG was created to provide cryptographic privacy and authentication for data communication. To achieve this goal, the GnuPG implements encryption, decryption and signatures algorithms. Instructions for basic usage of these algorithms will be described in the following steps:

1. Encrypting files and text,

2. decrypting files and text,

3. signing data,

4. verifying signed data.

Encrypting Files and Text

The GnuPG do not distinguish between the text and binary files and both can be encrypted using the same arguments – *gpg --recipient uid --encrypt [filename]* (*uid* specifies which public key will be used for encryption of generated session key). Regardless of the source file type, the output will be binary content. Though it is not a problem to use the binary representation of encrypted data for images, videos or other files, binary content cannot be used on web pages. To create encrypted text that can be used on web pages, specify the argument *--armor* as well.

```
xmatej52@merlin: ~$ gpg --armor --recipient \
xmatej52@stud.fit.vutbr.cz --encrypt
Hello world
-----BEGIN PGP MESSAGE-----

hQGMA9uyf/0+dawwAQwAgUvSRPDKKtBcVUMOU4Wna/UCaVARIQwlfQUm7hFJa1xp
.....
-----END PGP MESSAGE-----
```

Decrypting Files and Text

Data are encrypted with a symmetric algorithm. The session key used for encryption is randomly generated and the session key is encrypted using the receiver's public key. To decrypt the session key, it is necessary to access the receiver's private key. Because of it, decryption may require extra permissions (for example to run gpg via *sudo*), if the logged user does not have permissions to access the private key. To decrypt data, run command *gpg --decrypt [filename]*. If the private key is encrypted, the user will be asked to enter a passphrase.

```
xmatej52@merlin: ~$ gpg --decrypt
-----BEGIN PGP MESSAGE-----

hQGMA9uyf/0+dawwAQwAgUvSRPDKKtBcVUMOU4Wna/UCaVARIQwlfQUm7hFJa1xp
.....
```

```
-----END PGP MESSAGE-----
Enter passphrase:
gpg: encrypted with 3072-bit RSA key, ID DBB27FFD3E75A, created 2019...
       "Jiří Matějka <xmatej52@stud.fit.vutbr.cz>"
Hello world
```

Signing Data

Data are signed using the sender's private key so it is guaranteed that the sender is
the only one, who can create such data (unless the private key is compromised). As
a result of using the private key, signing may require extra permissions, if the logged
user is not authorized to access the private key. Data can be signed using the command
gpg --sign –local-user uid [filename] (*uid* specifies which private key will be used for
signing). To prevent binary content on output, the argument *--armor* can be specified
as well. If the private key is encrypted, the user will be prompted for the passphrase
he specified while generating the keypair.

```
xmatej52@merlin: ~$ gpg --armor --recipient \
xmatej52@stud.fit.vutbr.cz --encrypt
Hello world
Enter passphrase:
-----BEGIN PGP MESSAGE-----

owEB9QEK/pANAwAKAT3XKsKGzorWAcsSYgBeEPJkSGVsbG8gd29ybGQKiQHPBAAB
.....
-----END PGP MESSAGE-----
```

Verifing Signed Data

Given signed data, the user can both verify the signature and recover original data.
If user only wishes to check the signature, he can use *--verify* option. To verify the
signature and extract the original data, the user can use *--decrypt* argument. To verify
the signature, gpg need to access the sender's public key. Without the access, the
original data can be still recovered but cannot be verified.

```
xmatej52@merlin: ~$ gpg --verify
-----BEGIN PGP MESSAGE-----

owEB9QEK/pANAwAKAT3XKsKGzorWAcsSYgBeEPJkSGVsbG8gd29ybGQKiQHPBAAB
.....
-----END PGP MESSAGE-----
gpg: Signature made Sat 04 Jan 2020 21:15:37 CET
gpg:                using RSA key 3252EA3A9A0F105E226BE7BF6C9359504F0C8F81
gpg:                issuer "xmatej52@stud.fit.vutbr.cz"
gpg: Good signature from "Jiří Matějka <xmatej52@stud.fit.vutbr.cz>"
[ultimate]
```

```
xmatej52@merlin: ~$ gpg --decrypt
xmatej52@merlin: ~$ gpg --verify
-----BEGIN PGP MESSAGE-----


owEB9QEK/pANAwAKAT3XKsKGzorWAcsSYgBeEPJkSGVsbG8gd29ybGQKiQHPBAAB

.....
-----END PGP MESSAGE-----
Hello world
gpg: Signature made Sat 04 Jan 2020 21:15:37 CET
gpg:                 using RSA key 3252EA3A9A0F105E226BE7BF6C9359504F0C8F81
gpg:                 issuer "xmatej52@stud.fit.vutbr.cz"
gpg: Good signature from "Jiří Matějka <xmatej52@stud.fit.vutbr.cz>"
```

2.3.3   User Interface

KGpg is KDE's application providing a simple interface for the GnuPG. The application can help to set up and manage keys, import and export keys, view key signatures, trust status, expiry dates or encrypt/decrypt text or files. The KGpg is a free and open source software available for Linux and similar operating systems. Since the KGpg provides user interface, KGpg makes it easy to work with the gpg command–line application so the user does not have to remember all the gpg's commands, particularly those for the key management and the data encryption/decryption.

2.4   Alternative Software

Although the GnuPG is a very popular software and one of the most used implementations of the *OpenPGP* standard implementation, there are some alternatives for the GnuPG that also implements the *OpenPGP* standard. *OpenPGP.js* is one of such alternatives and was even used in this thesis for a prototype development.

OpenPGP.js

The *OpenPGP.js* is a project that aims to provide an open source *OpenPGP* JavaScript library so it can be used on most of the devices. While many other implementations of *OpenPGP* standard are aimed at using native code, the *OpenPGP.js* is meant to bypass this requirement so people are not forced to install gpg on their machines in order to use the library. The idea behind *OpenPGP.js* is to implement all the needed *OpenPGP* functionality in JavaScript library that can be reused in other projects that provides web browser extensions or server applications. The information provided about the *OpenPGP.js* is gained from *OpenPGP.js* project webpage [13].

The *OpenPGP.js* library was used for the implementation of the first prototype (Section 4.1). The implemented prototype was able to encrypt and decrypt elements using hardcoded private and public keys in the source code. Accessing the user's public and private keys was a serious problem. When a background script and a native applica-

tion were developed, this library was no longer necessary and was replaced with gpg (using gpg also solved the problem with private and public key access).

Chapter 3

Browser Extensions

This chapter summarizes basic knowledge about browser extensions that are needed to understand the software development process of the thesis. The emphasis will be on Firefox browser extensions because it is the target browser for the extension. The chapter is divided into the following sections:

1. WebExtensions API

2. Structure of a Firefox Extension

3. Native Applications

The *WebExtensions API* section provides an introduction into the browser extension development. The content of this section is a summarization of information provided by [2] and Firefox [7, 12] web pages. The following section, *Structure of a Firefox Extension*, describes the content of a browser extension's manifest file and the text of the section, and its subsections, summarizes the Firefox documentation [6, 9, 8, 11]. The last section of the chapter contains a brief description of native applications, and the content of the section is based on the Firefox documentation [10] as well.

3.1 WebExtensions API

The browser extensions are software programs that extend or modify the capabilities of a browser. They enable users to tailor web browser functionality and behaviour to individual needs or preferences. They are built on web technologies such as HTML, JavaScript, and CSS. The browser extension developed in this thesis is built using the *WebExtensions API*.

Extensions for the Firefox browser are built using the *WebExtensions API*. It is a cross–browser system for developing extensions. To a large extent, the system of the *WebExtensions API* is compatible with the extension API supported by Google Chrome, Opera and the W3C Draft Community Group.

Extensions cat take advantage of the same web APIs as JavaScript on a web page, but they can access to its own set of JavaScript APIs. Thanks to this, extensions can do a lot more than a developer can with code on a web page.

An extension must serve a single purpose that is narrowly defined and easy to understand. A single extension can include multiple components and a range of functionality, as long as everything contributes towards the defined purpose.

3.2 Structure of a Firefox Extension

An extension consists of a collection of files, libraries, packages and installation, yet there is the only file that must be present in every extension – *manifest.json*. The manifest contains basic metadata about the extension (name, version or the permissions it requires). It also provides a list of other files that are included in the extension.

The manifest can also include pointers to several other types of files:

1. Icons

2. Content scripts

3. Background scripts

4. Sidebars, popups, and option pages

5. Web–accessible resources

3.2.1 Icons

Icons are used to represent extension in components (for example in Add-ons Manager). Icons are represented in the manifest file as an object that consists of key–value pairs of image size (in px) and image path relative to the root directory of the extension.

It is recommended by Mozilla developers to supply at least the main extension icon, ideally in 48×48 px in size. However, it is possible to provide icons of any size and Firefox will attempt to select the best icon to display in different situations. Firefox also promises to consider screen resolution when choosing an icon so it is good practice to provide double–sized versions of all icons in order to deliver the best visual experience.

Firefox also supports the usage of icons in SVG format but it is necessary to specify *viewBox*. Nevertheless, a developer can use one file in the SVG format as an icon, he needs to specify various sizes of the icon in the manifest.

3.2.2 Content Scripts

Content scripts are used to access and manipulate a content of web pages. Content scripts are executed in the context of a particular web page and can access DOM APIs the same way, just like the scripts loaded by the web page.

Content scripts can only access a relatively small subset of the WebExtension API, but they can communicate with background scripts (Section 3.2.3) using a messaging system, thereby indirectly access the WebExtensions API.
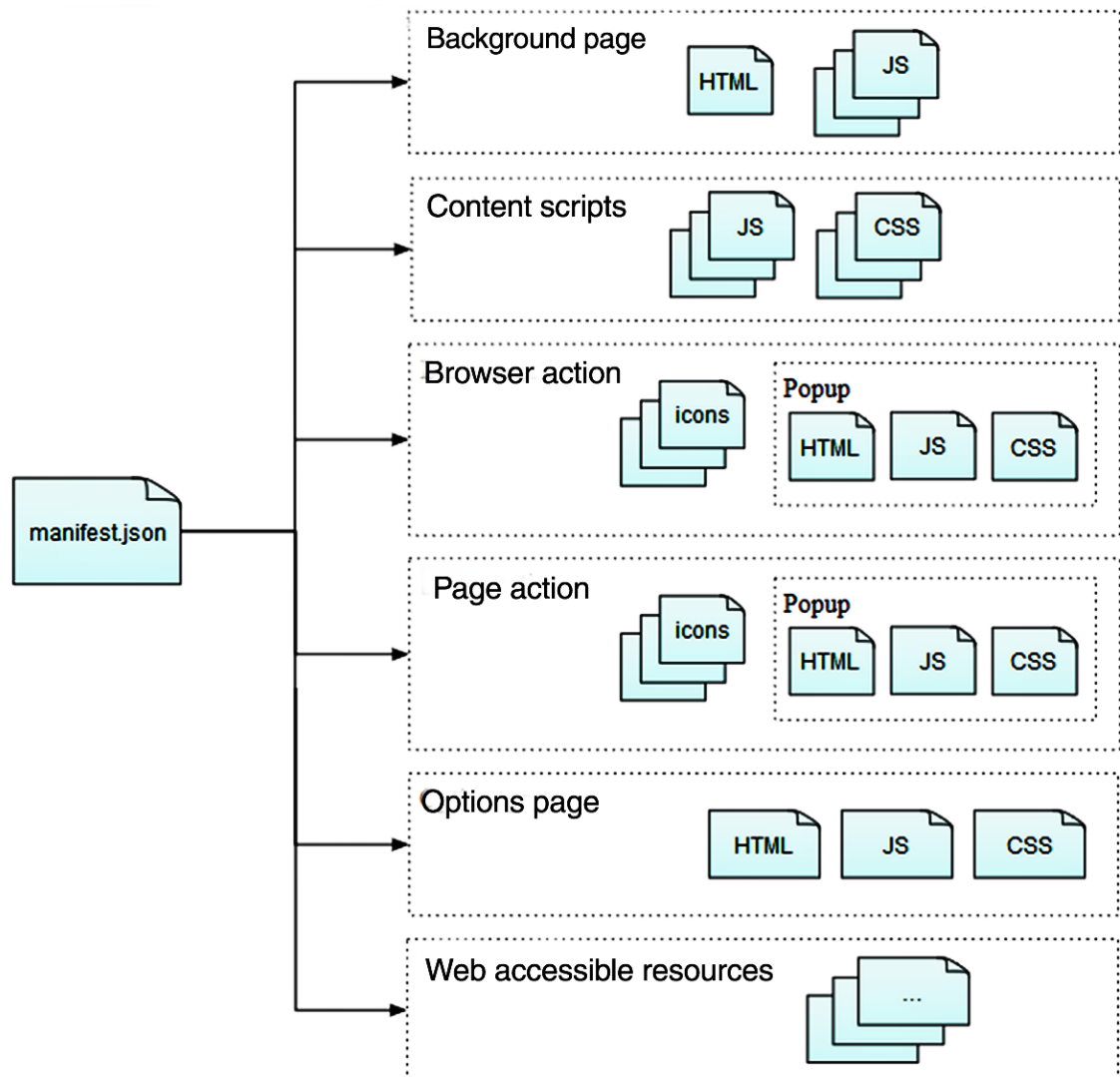
Figure 3.1: Anatomy of a browser extension [6].

Content Script Enviroment

As mentioned earlier, content scripts can access DOM APIs, therefore they are able to modify and to access DOM, just like normal page scripts can. Furthermore, they can detect and see any changes that were made to the DOM by page scripts. However, it is worth mentioning that content script does not share JavaScript variables defined by page scripts, and if page script redefines a built–in DOM property, the content script will be able only to access the original version of the modified property. The same is true in reverse – page scripts cannot see JavaScript property changes made by the content script.

Another feature content scrips have is the possibility of making requests using *window.XMLHttpRequest* and *window.fetch()* APIs. Content script shares the same cross–domain privileges with the rest of the extension. In case the extension has requested cross–domain access for domain using the permissions key in the manifest file, then its content script can get access to the same domain as well.

The possibility of communication with background scripts (Section 3.2.3) is another important feature of content scripts. There are two basic types of communication between content and background scripts – one–of messages and longer–lived connection. The One–off messaging is useful when none or only one response is expected to a message, and when only a small number of scripts listen to receive messages. On the other hand, it is recommended to use connection-based messaging, where multiple messages are exchanged, when the extension needs information about task progress, needs to be notified if a task is interrupted, or may want to interrupt a task that was initiated using messaging system.

3.2.3 Background Scripts

Background scripts are for maintaining long–term state or for performing long–term operations independently of the lifetime of any particular web page or browser window. Background scripts are loaded with extension installation and stay loaded until the extension is disabled or uninstalled. Developers can also profit from *WebExtension APIs* in the background scripts, although it is necessary to specify any needed permissions in the manifest file.

A background page is special page that provides a *window* global, along with all the standard DOM APIs provided by it. Since background scripts run in a context of the background page, they can access the *window* global.

The background scripts does not have direct access to web pages. But they are alowed to load content scripts into web pages and they can communicate with these scripts.

3.2.4 Sidebars, Popups, and Option Pages

Extensions can include various user interface components. Those components are defined with an HTML document. Components can be:

1. Sidebar – a pane displayed at the side of the browser window, next to the web page

2. Popup – a dialogue that is associated with a toolbar button or address bar button

3. Option page – a page that enables preferences that user can change

Every component is defined with its own HTML document and a point to it using specific property in the manifest file. Such HTML document can include both CSS and JavaScript files. JavaScript can use all privileged *WebExtension APIs* as background scripts (Section 3.2.3) and they can even directly access variables in the background page using a special method *runtime.getBackgroundPage().*

3.2.5   Web–Accessible Resources

Sometimes it is needed to provide resources (images, HTML, CSS, JavaScript, etc.) and make them available to web pages. This problem is solved with web–accessible resources. Resources, which are made web–accessible, can be referenced by page scripts and content scripts using special URI scheme.

Note that if a page is made web–accessible, any website may link or redirect to that page. Therefore, web–accessible pages should treat any input as if it came from an untrusted source.

3.2.6   Extension Pages

Extension pages are HTML documents in browser extension that are not attached to some predefined user interface components. Extension pages do not have entry in the manifest file unlike sidebars, popups or option pages. However, they also get access to the same privileged WebExtension APIs as the background script.

3.3   Native Applications

A native application is installed using the underlying operating system's installation machinery. A browser can communicate with the native application via a native messaging mechanism. This enables an extension to exchange messages with the native application installed on the user's computer. Thanks to this mechanism, native applications can provide services to extensions without needing to be reachable over the web. Another feature provided by the native application is that they can enable to access resources that are not accessible through WebExtension APIs, such as hardware.

Along with the native application itself, the developer needs to provide another manifest file called *host manifest* or *app manifest* and install it in a defined location on the user's computer. The host manifest file describes how the browser can connect to the native application. Note that the extension must also request permission *nativeMessaging* in its manifest file and the native application must include the extension's ID in the *allowed_extensions* field of the host manifest.

Once both extension manifest and host manifest are correctly set, the extension can exchange messages in JSON format with the native application. The native application receives messages on its standard input and sends them using its standard output. The extension uses a set of functions of the runtime API.
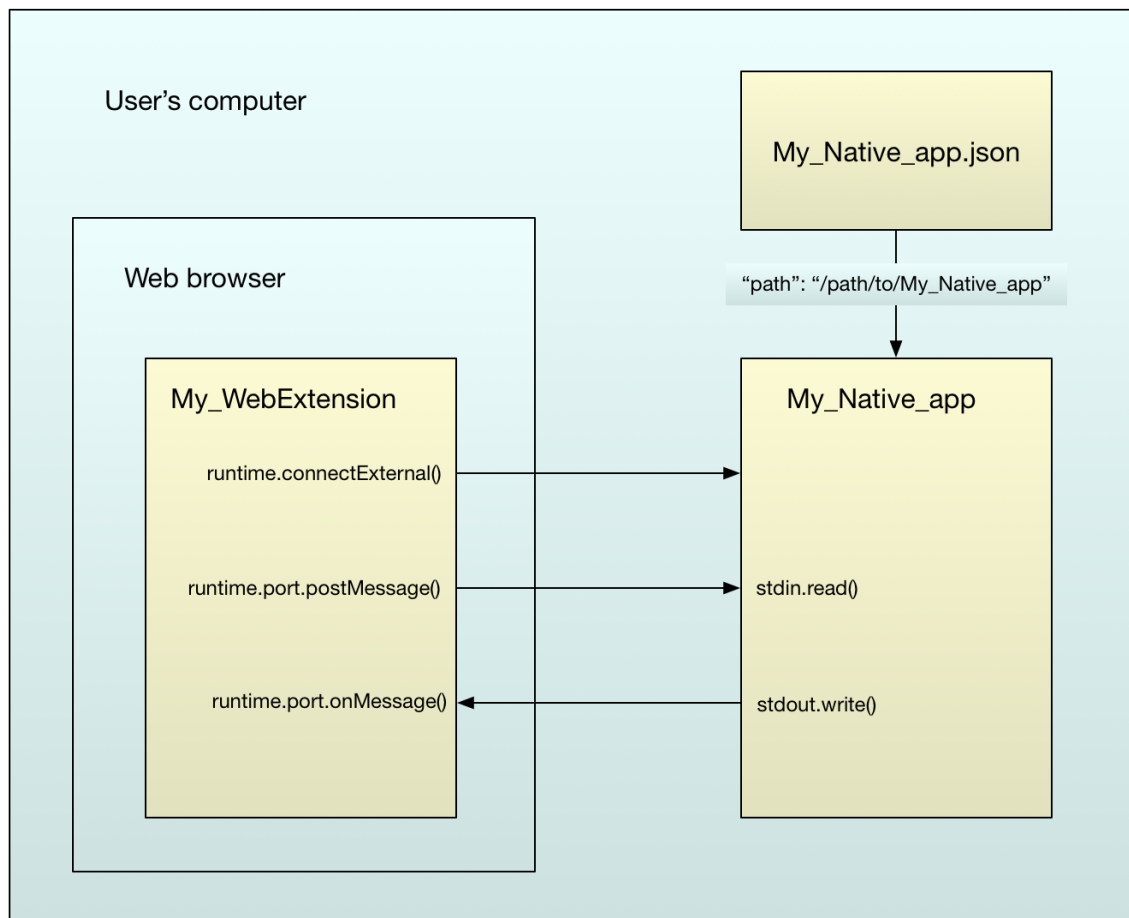
Figure 3.2: Native messaging scheme [10].

Chapter 4

Iterative Development

The goal of this thesis is to implement a browser extension that will detect encrypted elements on a web page and decrypt them. Developing a software is a complex and demanding process and it is not easy to deliver a project that correspondents with stakeholder's expectation. To fulfil all expectation, the bowser extension was developed in several iterations.

Each iteration consists of design, development, testing and review. The iteration were usually 3 – 4 weeks long and resulted into a prototype. The prototype were then disgusted with thesis supervisor – Ing. Libor Polčák, Ph.D.
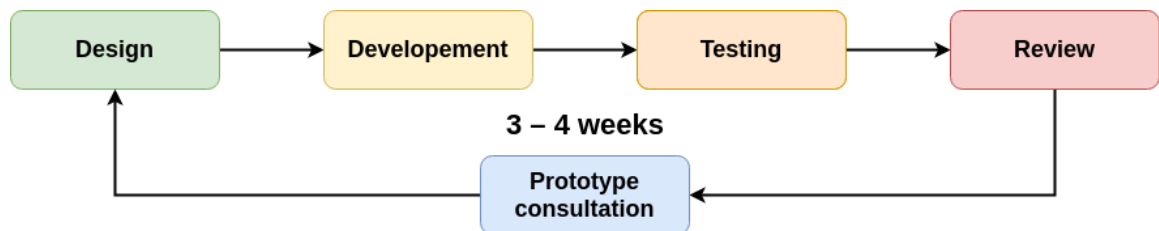
Figure 4.1: Software development model.

Each iteration is described in the thesis from various reasons. During prototype development, new problems that were missed in the software design can occur. Also, some of the used technologies and methods can lead to a dead–end or lose their purpose in the future prototypes or the final product. Such methods or technologies might not be important for the final product of the thesis, but can be important for some readers.

4.1 OpenPGP.js Prototype

The purpose of the first prototype is to learn how to develop a browser extension. The goal is to implement a simple browser extension that will be able to decrypt images using a hardcoded private key. Ideal library for this task was *OpenPGP.js* (Section 2.4) and since the Firefox browser has troubles to load the *OpenPGP.js* library, this prototype is implemented for the Google Chrome browser.

4.1.1 Design

No background script or native application is needed to develop so new script called *gnupg_decryptor.js* and the *OpenPGP.js* library can be loaded as content scripts. The designated schema of the web extension can be seen in the figure 4.2. The script *gnupg_decryptor.js* will detect encrypted image by file suffix in its source URL, downloads it, and tries to decrypt it using *OpenPGP.js* library. Once the content of the image is decrypted, *gnupg_decryptor.js* will create a new URL pointing to the decrypted image and replace the source URL of *img* element on the web page thus displaying the decrypted image to a user. The icon that represents the extension is taken from *The GNU Privacy Guard Project* web page [4].
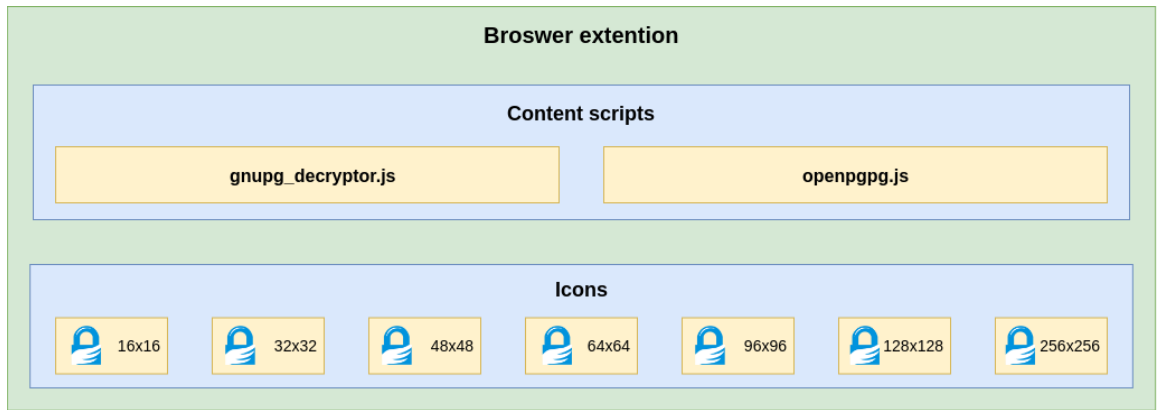
Figure 4.2: Structure of the *OpenPGP.js* prototype.

4.1.2 Implementation

Since some parts of this prototype serves as a foundation stone for next development, special care was taken for reusability of useful functions as well as for preparing testing web pages, a different set of encrypted data and also to prepare special scenarios, specifically another web extension that changes DOM of a web page. Simplified schema of the implemented prototype can be seen on the figure A.1 in the appendix section of the thesis.

4.1.3 Review

The implemented prototype is able to detect images on a web page by their source URL and then decrypt them using the *OpenPGP.js* library. The Prototype is able to use only hardcoded keys and can not access user's keyring on his computer. The prototype does not need any background scripts or a native application. The only browser, where is implemented prototype operational, is Google Chrome.

Some functions implemented in this prototype are included in future prototypes and in a final product of the thesis as well. The main benefits from the first iteration are implemented test pages, encrypted images, generated keypairs and gained experience with web extension development.

4.2 GnuPG_Decryptor Prototype

Hardcoded keys are one of the biggest problems of the previous prototype. A new prototype should have access to users keyring. In order to access it, it will be necessary to implement a native application that will be able to communicate with the gpg application. With the native application, the *OpenPGP.js* library will no longer be necessary and all decryption tasks will handle gpg itself.

4.2.1 Design

Content script (*gnupg_decryptor.js*) will start the communication with a background script. The background script will serve only as an intermediator and will resend messages from the content script to a native application. The native application will be implemented in Python language and will communicate with gpg using the *subprocess* library. Since a user must provide passphrase to a private key, the native application will have a user interface as well. The user interface will be implemented using the *tkinter* library. All used Python libraries are showned on the figure 4.3.
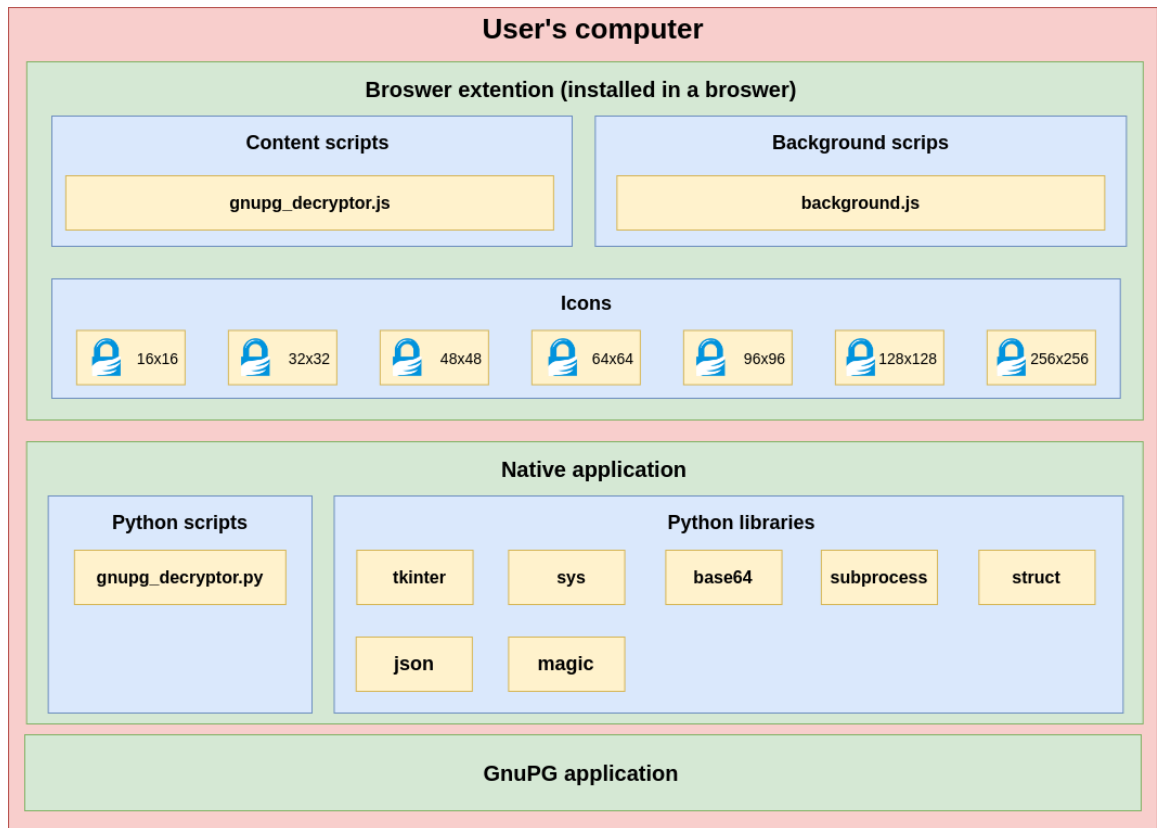

Figure 4.3: Anatomy of the GnuPG_Decryptor prototype.

Decoded content will be sent to the background script and it will resend it to the content script. The content script will then replace encrypted data with decrypted data the same way as in *OpenPGP.js* prototype (Section 4.1).

Messages must be in JSON format. All messages will have a mandatory field *type* that will describe the type of message. In this prototype, supported types will be *decryptResponse, decryptRequest, and debug*. A message with type *decryptRequest* will have mandatory fields *messageId* containing the ID of an encrypted element on a web page, *data* containing encrypted data, and *encoding* specifying which encoding was used to encode the content of field *data*. A message of type *decryptResponse* will have mandatory fields *success* specifing if the decryption was successful, *data* containing base64 string with decrypted content (or an empty string if failed), *message* containing an error message if decryption failed (or an empty string on success), and *messageId* containing the ID of *decryptRequest* message to which the response relates.

4.2.2 Implementation

Purpose of this prototype is the implementation of a messaging system that will be used in future prototypes and final product. The messaging system simplifies debugging of the native application by supporting debug messages and the messaging system is extensible and reusable although the messaging system is still incomplete.

The native application is implemented as simple as possible. Application has two dialogues to enter the passwords – password for sudo to access user's keyring and for a passphrase to decrypt the private key. The application does support decryption with using only one encrypted key and does not resolve errors reported by sudo or gpg itself.

4.2.3 Review

The implemented prototype can access to a certain extent user's keyring. Using the messaging system, the extension is able to exchange data between the gpg application installed on a user's computer and a web browser thus enabling usage of gpg for decryption. Through messaging system is incomplete, the GnuPG_Decryptor prototype offers sufficient basis for future development.

Future prototypes must complete messaging system – there is missing functionality dealing with the limited size of messages. The user interface provided by the native application is also insufficient and must be improved in order to use multiple keys for decryption and do not require sudo password, if not necessary. The native application also needs to deal with errors occurs during the data decryption and somehow informs the user about the error.

4.3 Large Content Support

In native messaging, the maximum size of a single message sent from the native application is 1 MB and the maximum size of a message sent from the background script is 4 GB. Because of these limits, it is necessary to send large content in separate messages. So it is necessary to make changes in the messaging system on both sides of the communication.

4.3.1 Design

To support the limited size of a single message, all types of messages will have mandatory elements *messageId* and *lastBlock*. If a message would exceed the maximum size, its data will be split into blocks and each block will be sent with separately with the same *messageId*. The field *lastBlock* will be set to value 1 in case of the last block otherwise to 0.

The maximum size of the *data* part of *decryptRequest* message will be set to a maximum of 75 % of a maximum size of the whole message (3 GB), so there is more than enough space for other attributes of the message. Dividing data into blocks will be handled by the content script, so the background script can only forward messages to the native application.

Since there is no other choice, large content sent by the native application will be split into blocks by the native application itself. As it is inthe content script, the maximum size of the *data* part of the *decryptResponse* message will be set to 75 % (750 KB) of the maximum size of the whole message. This will ensure that there will be enough space for the rest fo message attributes.

4.3.2 Implementation

The messaging system was updated as designed – attribute *messageId* is now mandatory as well as new mandatory attribute *lastBlock*. The size of the *data* part of *decryptRequest* message is computed as length of base64 string or armoured text (in both cases, one character has a size of one byte) and if it is necessary, the string is split. In the case of the native application, size of the *data* part of the *decryptResponse* message is computed as a number of bytes (and data are then divided if needed). An example of decrypting large content is shown on the figure A.3 in the appendix section of the thesis.

4.3.3 Review

The browser extension is now able to decrypt files of any size, thus allowing usage of large files like high-resolution images, audio tracks or even videos. But the extension is not able to detect any dynamic changes in DOM of a web page, so it is not able to appropriately react to changes made by other scripts or to work properly work with XHR API, Fetch API or Push API.

4.4 Interactive Changes Support

To support XHR API, Fetch API and Push API, it is necessary to somehow detect changes in the DOM tree of a web document. For this purpose, the Web API implements a *MutationObserver* interface. With the usage of the *MutationObserver* interface, the browser extension will be able to watch changes being made to DOM tree of a web document.

4.4.1 Design

The content script will be responsible for using the *MutationObserver*. The *MutationObserver* need to be connected to the root node of the DOM tree, so it will detect all the changes in the DOM tree of a web document. Once some change occurs, the content script needs to locate all changed or new elements and detects which of them (if any) contains encrypted data.

To ensure, that same elements will not be detected multiple times, this prototype will also stores detected elements and data, so same encrypted data are not sent to the native application for deduplication multiple times. To prevent possible high memory demands, this data will be stored in the content script.

4.4.2 Implementation

Both the *MutationObserver* connection and duplicities detection are implemented in the content script. Once a web page is loaded, the content script connects the *MutationObserver* to the *document.documentElement* (root element of the web document) to detect any changes in the content of elements, the tree structure of DOM or the *src* attribute of any element. If a new subtree is detected, the root of the subtree will be parsed by function, that will check the root itself and all of its descendants for encrypted content. The *MutationObserver* will also detect changes made by the browser extension itself and will test inserted content to the page as well. Thanks to this feature, the web browser extension is able to decrypt content, that was encrypted multiple times using multiple keys (or the same key multiple times). Each parsed content is stored in the associative array. Index to the array is either hash of encrypted content (in case of armoured text) or URL to an encrypted file.

In the case of armoured text, stored data are *status*, *type*, *data* and *elements*. The attribute *type* is always set to value `text`. If the value of the *status* is `decrypted`, the attribute *data* contains decrypted data. Otherwise, the value of *status* is `decryptRequest`, meaning that encrypted data were already sent to the native application and decrypted content is not available yet. The attribute *elements* contains a list of element ids, that uniquely identifies elements, which shares the same encrypted content. If the encrypted text (in the format of PGP armoured text) is detected, a hash is computed. Then it is tested if the hash is already used as string index into the associative array and if not, new data are inserted into the array and encrypted content is then sent to the native application. Otherwise, the content of the element is replaced (if encrypted data are already available), or appended to the list of elements with the same content for future processing.

Encrypted files are in the associative array stored similar way as the armoured text. Stored data are *status*, *type*, *url* and *elements*. The attribute *type* is always set to value `file`. If the value of the attribute *status* is `decrypted`, the attribute *url* contains URL to a decrypted file, otherwise, the *url* points to an encrypted file. Unlike in the case of armoured text, *status* can contain two more values – `creatingRequset` and `decryptRequest`, depending on a situation – if the file was already downloaded and forwarded to the native application (then the value is `decryptRequest`), or is being currently downloaded (then the value is `creatingRequset`). If an encrypted file is

detected, it is processed the same way as in the case of armoured text, except the URL is used as a key to access array, instead of a hash.

4.4.3 Review

The implemented prototype is able to work properly with XHR API, Fetch API, Push API or any other API or javascript module that can interactively change the content of a web page. The prototype is even capable of decrypting content, that is encrypted multiple times with multiple keys (as long as these keys are protected with the same password, or are not protected by password at all). Furthermore, a number of exchanged messages might be even smaller than in the previous prototype thanks to the new deduplication system.

But the newest prototype still lacks usable GUI. Current GUI is too simple and lacks support to enter different passwords for different keys, thus making this prototype still not suitable to be published on the Mozilla addons page.

4.5 Graphic User Interface

Missing or insufficient graphic user interface greatly reduces the usability of this browser extension. Current GUI is too simple and requires two passwords – sudo and the password of the key, both are mandatory. But the user interface also needs to collect necessary parameters for GnuPG application like homedir. User can also have multiple private keys and GUI needs to have entries for a password of every key, the user might want to use.

4.5.1 Design

The user interface will support multiple password entries for different private keys. There will be also entries for sudo and homedir parameter. Providing sudo a password will not be mandatory anymore since a user might not have permission to use it. Homedir will not be mandatory as well (GnuPG will use default location). A draft of a simple and sufficient user interface, that will be implemented in this prototype, is shown on the figure 4.4. The graphic user interface will be displayed once the user clicks on the icon of the extension in a web browser.

Figure 4.4: Draft of a user interface.

The previous graphic user interface was implemented using library *tkinter*. In this iteration, this library will be replaced by *PyQt5*. To make the implementation of the drafted user interface easier, a sketch was made, that is made only from *PyQt5* classes (expect classes *GnuPG_Decryptor_GUI*, *KeyList*, *KeyItem* and *Refresher* – but these classes only extend the *Widget* class from *PyQt5*). The created sketch can be seen on the figure 4.5. With the new module and changed libraries, the structure of the web browser extension was changed and new structure can be seen on the figure 4.6.

4.5.2 Implementation

The graphic user interface is implemented in the separated module named as *GnuPG_Decryptor_GUI.py*. The class *GnuPG_Decryptor_GUI* contains a pointer to object of class *GnuPG_Decryptor* (a class that implements native application), so GUI can communicate with the backend. Furthermore, class *GnuPG_Decryptor_GUI* contains references to two more classes – the *KeyList* class and the *Refresher* class. The *KeyList* contains all information about user keys. This information is held, displayed and can be edited by *KeyItem* class. The *KeyList* class can contain any number of objects of *KeyItem* class. The *Refresher* class contains information about the homedir parameter and usage of sudo. The *Refresher* class is also responsible for showing these settings to the user and enabling its editing.
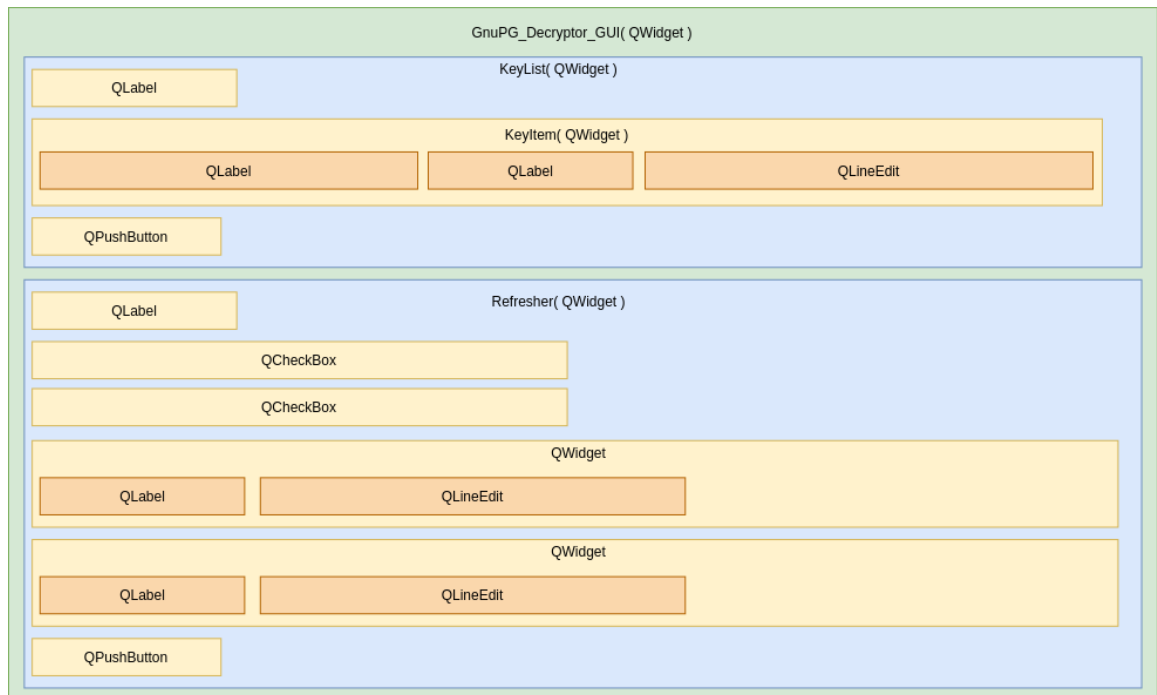
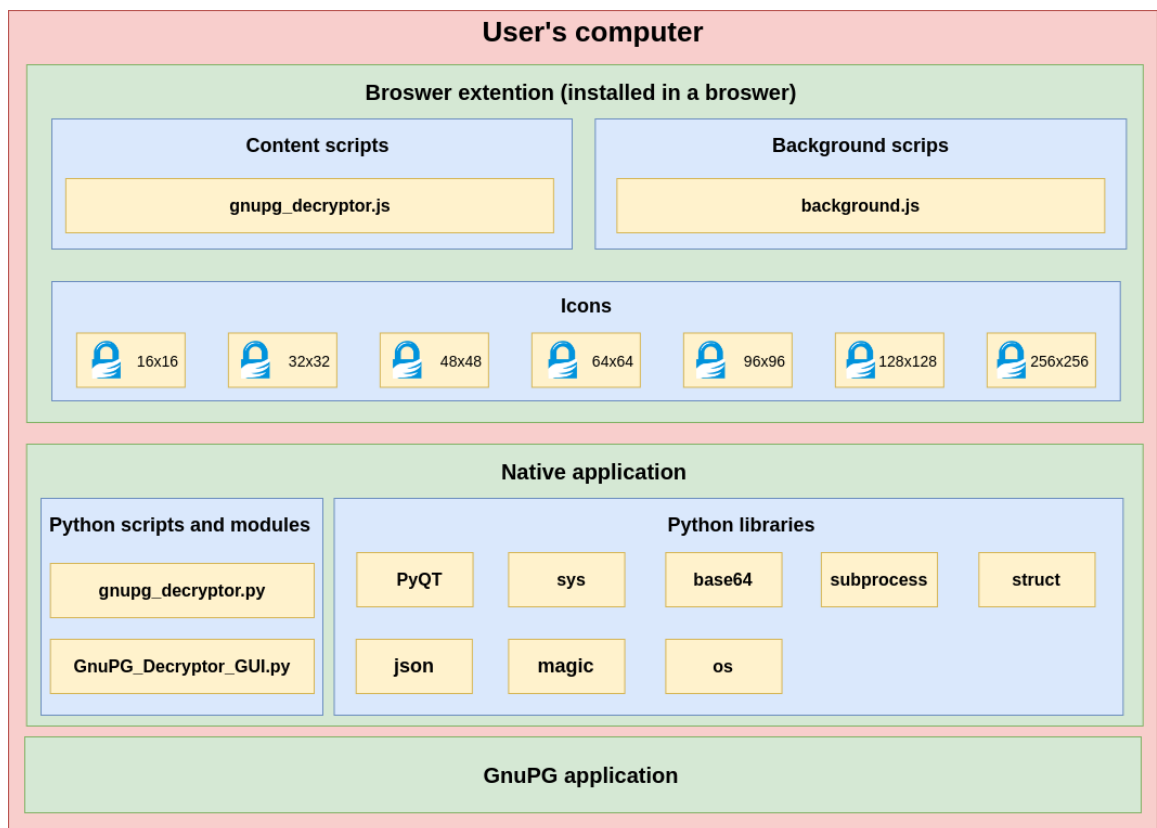Figure 4.5: Structure of the implemented user interface.

Figure 4.6: New structure of the web browser extension.

4.5.3 Review

With the new graphic user interface, this prototype is functional and was successfully published on the Mozilla addons page. To complete the web browser extension, it is necessary to run a number of tests. Once the web browser extension is debugged, optimized and tested, the online documentation needs to be created and it is also necessary to publish the native application on some public web page.

4.6 Final Iteration

The final iteration results in the final product of the thesis. Tests (Chapter 5) discovered two major bugs in the last implemented prototype – unable to detect encrypted files in *href* attribute and unable to work when the encrypted content is detected on more than one tab at the same tame. The final product must also support threads in the native application.

4.6.1 Design

The problem with the *href* will be fixed in the content script. The content script will detect encrypted files in the *href* attribute the same way as it detects files in the src attribute.

The problems with encrypted content in more than one tab at the same time is caused in the background script – the background script does not know, to which tab send the decrypted content. To resolve the problem, the content script will send to the background script a new *tabIdRequest* message and the background script will response with a *tabIdResponse* message, where the background script will provide the unique identification number for the content script. The number will be present in the mandatory attribute *tabId* of *decryptRquest* and *decryptResponse* messages. The messaging system will be also updated with thee new messages – *updateKeysRequest*, *getKeysRequest* and *getKeysResponse*. These messages will be used to exchange the settings of the web browser extension so it can be saved (without the passwords) and it is not necessary to set everything at every start of a web browser (unless he needs to use sudo).

Threading will be supported using the *threading* library for python. The main process will read the messages and once the decryption will be requested and possible, it will create and start a new thread. The thread will then decrypt the data and send decrypted data to the content script. The function, that sends the data to the content script will be protected with a lock to prevent unexpected behaviour.

The schema of the final product components can be seen on the figure 4.7. The example of the communication between the content script, the background script, the native application and the GnuPG application can be seen on the figure A.4 in the appendix section of the thesis.
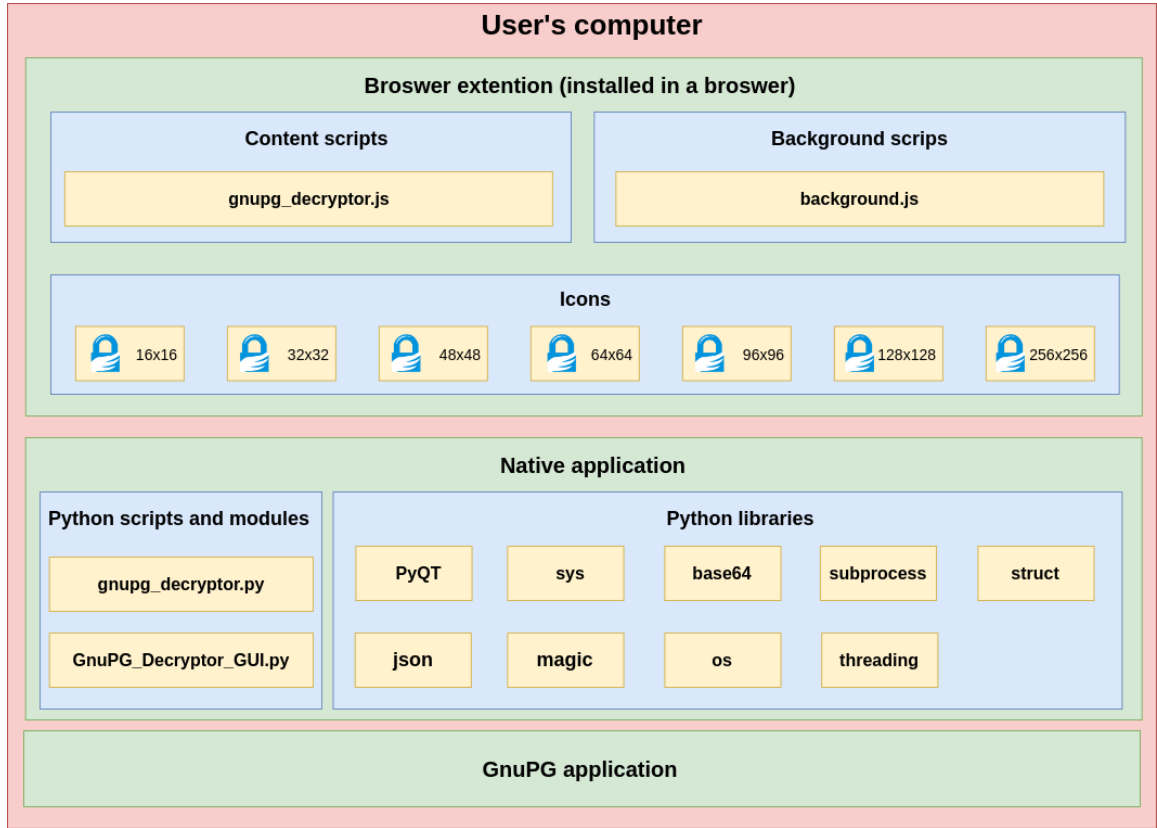
Figure 4.7: New structure of the web browser extension.

4.6.2 Implementation

All of the components of the web browser extension were modified. The native application was modified to support the threads and new messages (*updateKeysRequest* and *updateKeysResponse*). Once the native application is launched, it sends *updateKeysRequest* message to the background script. The background script loads settings from the storage and sends the settings back to the native application via *updateKeysResponse* message.

The *getKeysRequest* message has only attribute *type*. On the other hand, the *getKeysResponse* and *updateKeysResponse* messages have attributes *homedir*, *sudo* and *keys*. The *homedir* contains a string with *homedir* directory (or is not specified if the *homedir* parameter is not used), *sudo* can only contain values 1 or 0. If the sudo should not be used, the value is 0 or the *sudo* attribute is not specified. The attribute *keys* is a list of string uids of the keys and the attribute is mandatory.

The threads in the native application are used for data decryption and creation of *decryptResponse* messages. The main process prepares data for a thread (detects keys, join blocks, decode base64 string, ...) and then create the thread that will handle the rest. While the thread is decrypting the data, the main process can read new messages and react to them.

The content and background scripts were modified to support the *tabIdRequest* and *tabIdResponse* messages. Once the content script is loaded into the page, it sends the *tabIdRequest* message and waits for the *tabIdResponse* message from the background script. The *tabIdRequest* has one mandatory attribute – *type*. The *tabIdResponse* message has two mandatory attributes – *type* and *tabId*. The *tabId* contains the unique id for the content script that will be used, to determine to which tab the background script should forward the *decryptResponse* message.

4.6.3 Review

The implemented browser extension was tested in all implemented tests (Section 5) and passed all of them. During the time measurements, we measured only effect of the threads on the time it took to display decrypted elements on the page. More about the results of measurements and influence of the threads can be seen in the section about the performance of the web browser extension (Sections 5.3 and 5.3.1).

The implemented extension was named as *GnuPG Decryptor* and can be found on the Mozilla addon page[1]. Since it is possible to download only the extension on the Mozzila addon page and not the native application, a GitHub[2] repository was created, where the user can download the native application. All source codes and tests were also published on the GitHub page.

With this last iteration of the development process (the last one within this thesis), we created the web browser extension, that is able to detect and decrypt many encrypted elements on the page, such as text, HTML code, images, videos, audio tracks, styles, JavaScript or other pages and documents. The web browser extension is implemented for the Mozzila Firefox web browser and the native application is implemented for the GNU/Linux systems. In the future, the extension can be modified to support other browsers and the native application upgraded to support other operating systems.

---

[1]https://addons.mozilla.org/cs/firefox/addon/gnupg_decryptor/
[2]https://github.com/tehryn/GnuPG_Decryptor/

Chapter 5

Testing the Browser Extension

This chapter focuses on testing and debugging the web browser extension. The goal of testing is to debug and optimize the *Graphic user interface prototype* (Section 4.5). The tested and debugged prototype will result in the final product of the thesis (Section 4.6). The final product will be then uploaded as a new version of the *GnuPG Decryptor* extension on the Mozilla addons page.

We will test three aspects of the software and each aspect will be described in the separate section:

1. Basic Functions,

2. Advanced Functions,

3. Performance.

All the test were performed on the device *Acer Aspire VX15* with processor *Intel Core i5-7300HQ (2.5GHz, TB 3.5GHz)*. At the time of testing, the system installed on the computer was *Ubuntu 18.04.4 LTS* and version of the Mozilla Firefox was 78.0.2.

5.1   Basic Functions

In this section, we will test the basic functions of the web browser extension, like how is extension capable of detecting encrypted elements and decrypting them. The testing of the basic functions will be done in the following four steps:

1. Element Detection,

2. Basic Decryption,

3. Large Files,

4. Interactive Changes.

Since this section is about testing basic usage of the *GnuPG Decryptor extension*, there will be only one test file per step and the tests will be as much simple as possible. We plan to perform more complex tests in the section Advanced Functions (Section 5.2).

5.1.1 Element Detection

The first thing, the implemented web browser extension does, is the detection of the encrypted elements. In the first test, we will create a web page with several encrypted images, audio tracks and videos and test, if the extension is able to detect them all and send then to the native application. Here is a simplified HTML schema:

```html
<html><head>...</head>
  <body>
    <h1>Test 01</h1>
    <p>Detection test of 17 images, 4 audio tracks and 4 videos
       and 1 encrypted text.</p>
    <div width="320px">
      <img src="../encrypted/img/img1_test1.jpg.gpg" >
      <img src="../encrypted/img/img2_test1.jpg.gpg" >
      <!--15 more images follows-->
    </div>
    <div>
      <audio controls>
        <source
          src="../encrypted/audio/audio1-test1.mp3.gpg"
          type="audio/mpeg"
        >
        Error - browser does not support mp3.
      </audio>
    </div>
    <!--3 more audio tracks follows-->

    <div>
      <video width="320" height="240" controls>
        <source
          src="../encrypted/video/video4-test4.mp4.gpg"
          type="video/mp4"
        >
        Error - browser does not support mp4.
      </video>
    </div>
    <!--3 more video tracks follows-->
    <p>
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv+N72od1JiuvcybR9b7z1WxtShsRm0B9pHDWJc8zB+s3sk
...
-----END PGP MESSAGE-----
    </p>
```

```
    </body>
</html>
```

Once the web page was accessed by the browser with the *GnuPG Decryptor* extension installed, the extensions detected 26 encrypted files and sent them into the native application for decryption, thus the web browser extension is capable of detecting encrypted images, audios and videos on such simple page and sends all the encrypted elements into the native application only once.

Another element detection tests will be performed in the section where we will be testing interactive changes (Section 5.1.4) and in the section where recursive decryption will be tested (Section 5.2.3).

5.1.2   Basic Decryption

The next basic function, that will be tested is the decryption of small content – encrypted text and small images. To test this basic feature, we created a simple page, similar to the one used before, but only with 17 images and encrypted text (no video and audio tracks). We expect from the web browser extension to detect and decrypt 18 elements and then display them to the user. Here is a simplified HTML schema of the web document used for testing:

```
<html><head>...</head>
  <body>
    <h1>Test 02</h1>
    <p>Decryption test of 17 images and 1 encrypted text</p>
    <div width="320px">
      <img src="../encrypted/img/img1_test1.jpg.gpg" >
      <img src="../encrypted/img/img2_test1.jpg.gpg" >
      <!--15 more images follows-->
    </div>
    <p>
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv+N72od1JiuvcybR9b7z1WxtShsRm0B9pHDWJc8zB+s3sk
...
-----END PGP MESSAGE-----
    </p>
  </body>
</html>
```

All of the encrypted elements were detected, decrypted and displayed in the web browser, so the *GnuPG Decryptor* extension did as expected and we may proceed to test the decryption of larger data, like audio or video. To demonstrate, how the *GnuPG Decryptor* extension replaced encrypted elements, we appended the simplified HTML schema of the decrypted document. The bolded text has been updated by the *GnuPG Decryptor* extension.

```
<html><head>...</head>
  <body>
    <h1>Test 02</h1>
    <p>Decryption test of 17 images and 1 encrypted text</p>
    <div width="320px">
      <img
        id="GnuPG_DecryptorElemId-0"
        src="blob:https://www.stud.fit.vutbr.cz/a8e463c9-3fd8..."
      >
      <img
        id="GnuPG_DecryptorElemId-1"
        src="blob:https://www.stud.fit.vutbr.cz/5ce5b051-181f..."
      >
      <!--15 more images follows-->
    </div>
    <p id="GnuPG_DecryptorElemId-17">Testing....</p>
  </body>
</html>
```

5.1.3 Large Files

To test the decryption of large file, we created a similar HTML page as in the previous section, but instead of the images, we used video and audio elements. Both video and audio are larger than 1 MB, so they have to be split into blocks in order to be sent by the native messaging API. Since files are smaller than 4 GB, we set the maximum size of a single message in the content script to 4 MB. We expect that the implemented web browser extension will split data in the content script into blocks with a size of 3 MB (75 % of the set maximum size) and send them into the native application. The native application should then join those blocks and decrypt the files. Once it is finished, it will split the decrypted data into blocks of size up to 750 KB and sends them into the content script. The content script is expected to join those blocks and display decrypted content in the web browser. A simplified HTML schema of test document follows:

```
<html><head>...</head>
  <body>
    <h1>Test 03</h1>
    <p>Decryption test of 1 encrypted video and 1 encrypted audio</p>
    <div>
      <audio controls>
        <source
          src="../encrypted/audio/audio1-test1.mp3.gpg"
          type="audio/mpeg"
        >
        Error - browser does not support mp3.
```

```
      </audio>
    </div>
    <div>
      <video width="320" height="240" controls>
        <source
          src="../encrypted/video/video4-test4.mp4.gpg"
          type="video/mp4"
        >
        Error - browser does not support mp4.
      </video>
    </div>
  </body>
</html>
```

The web browser extensions detected both encrypted files, split them into blocks and sent them to the native application as expected. The native application correctly accepted the blocks, joined them into one buffer and forwarded the data into GnuPG application. Once data was decrypted, the native application split them into blocks and sent them back to the content script. The content script joined received blocks and correctly displayed the decrypted content for the user.

5.1.4 Interactive Changes

I this scenario, we will prepare a more complex test page than in previous tests. We prepared encrypted a CSS file with styles and a javascript code, that will load encrypted content into page after 4 seconds. We expect the extension to load the detect the encrypted CSS file, decrypt it and load the styles. Once javascript inserts encrypted content into the page, we expect the *MutationObserver* will detect the change, thus enabling the *GnuPG Decryptor* extension to detect new encrypted contents and decrypts it. Here is a simplified HTML schema.

```
<html>
  <head>
    ...
    <link rel="stylesheet" href="../encrypted/css/styles.css.gpg">
  </head>
  <body>
    <script>
    setTimeout( insert, 4000 );
    function insert() {
      document.getElementById( 'insertHere' ).innerHTML = '
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv/e++Sh8fknwHaSd92p4xUQ9L1FJwnyOIf4Pby2V6A1Qe7
...
-----END PGP MESSAGE-----
```

```
';
    }
    </script>
    <h1>Test 04</h1>
    <p>Iteration changes on a page.</p>
    <p id="insertHere">This content will be replaced.</p>
  </body>
</html>
```

The *GnuPG Decryptor* extension did not detect encrypted CSS file as expected. But the extension was able to detect the encrypted content, once JavaScript inserted the content into the page. After this test, we updated the content script to detect the *href* attribute the same way, as it detects the *src* attribute. Once the extension was updated, the content script was able to detect, decrypt and loads the styles into the web page.

5.2 Advanced Functions

In this section, we tested the advanced features of the web browser extension. We started with simple tests, but in the Recursive Decryption (Section 5.2.3), we tested, how is the web browser extension able to handle complex pages with multiple times encrypted data, encrypted javascript, CSS styles or even whole HTML pages. The tested features were:

- Duplicate Files,

- Multiple Receipments,

- Recursive Decryption.

5.2.1 Duplicate Files

To test the deduplication system, we created another test, that consists of 40 images and 5 encrypted texts. The images has different relative *src* URL but the URL still points to the same file. The text is the same as well but in different elements. We expect that the *GnuPG Decryptor* extension detects 45 encrypted elements but sends to the native application only two. A simplified HTML schema follows:

```
<html><head>...</head>
  <body>
    <h1>Test 05</h1>
    <p>Deduplication test - The page contains 40 same images
       with different relative path and same encrypted text
       in five different elements.</p>
    <div width="320px">
      <img src="././encrypted/img/img1_test1.jpg.gpg" >
      <img src="././././encrypted/img/img1_test1.jpg.gpg" >
      <!--15 more encrypted images follows-->
```

```
      </div>
      <p>
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv+N72od1JiuvcybR9b7z1WxtShsRm0B9pHDWJc8zB+s3sk
...
-----END PGP MESSAGE-----
      </p>
      <h1>
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv+N72od1JiuvcybR9b7z1WxtShsRm0B9pHDWJc8zB+s3sk
...
-----END PGP MESSAGE-----
      </h1>
      <!--3 more encrypted texts follows in tags <span>, <b> and <del> -->
   </body>
</html>
```

The *GnuPG Decryptor* extension detected 45 encrypted elements but sent to the native application only two messages, one as a request to decrypt an image and one as a request to decrypt the armoured text. Therefore the deduplication system works as intended on this simple example. We further tested the deduplication system in the following tests.

5.2.2   Multiple Receipments

The GnuPG application allows specifying more than one recipient while encrypting data. To decrypt such encrypted content, it is necessary to have access to at least one of the used keys. The native application should detect, that content can be decrypted with multiple keys and use those keys, that are available. We created a web page, that contains five images. Images were encrypted with 4 different keys – *test1*, *test2*, *test3* and *test4*. The *test1* key is not protected with a password and was used as recipient of the first two images, so we expect, those images will be decrypted without need for a password. The *test2* key was used as a recipient for the second, the third and the fourth image. Once we provide a password for the key, we expect that these images will be decrypted as well. The *test3* key can be used to decrypt the second, the fourth and the fifth image. The *test4* key is used as a recipient for all the images, except the fourth image. So to decrypt all the content, it should be necessary to provide passwords for only two of three keys, that are protected by a password. Follow simplified HTML schema of the test page

```
<html><head>...</head>
   <body>
      <h1>Test 05</h1>
      <p>Multiple receipments -- all content should be decrypted
```

```
          using only two of three keys, that are protected by
          password (test2, test3.test4).</p>
        <div>
          <img src="../encrypted/img/img12_test1_test4.jpg.gpg" >
          <img
            src="../encrypted/img/img16_test1_test2_test3_test4.jpg.gpg"
          >
          <img src="../encrypted/img/img14_test2_test4.jpg.gpg" >
          <img src="../encrypted/img/img11_test2_test3.jpg.gpg" >
          <img src="../encrypted/img/img17_test3_test4.jpg.gpg" >
        </div>
    </body>
</html>
```

We tested all combinations of two keys and the whole content of the page was correctly displayed all the times as expected. Therefore we can say, that the *GnuPG Decryptor* extension supports encrypted files for multiple recipients.

5.2.3 Recursive Decryption

In this section, we have tested decryption of the data, that was encrypted multiple times. We expect, that the web browser extension will detect encrypted data, decrypt them, display to the user and then detect new encrypted content, decrypt it and display to the user. Here is a simplified HTML schema of the web document:

```
<html><head>...</head>
  <body>
    ...
    <div class="content">
      <h1>Test 06</h1>
      <p>Recursive decryption -- data on the page were encrypted
        two times using different keys.</p>
      <div>
-----BEGIN PGP MESSAGE-----

hQGMA63SHuyrZ3NZAQv/WdlCq6Upb0198drAgHGFRHOAL+dyC5fRegdTuUEjsyQf
...
-----END PGP MESSAGE-----
      </div>
    </div>
  </body>
</html>
```

Encrypted content was successfully detected, decrypted and replaced with new encrypted content. Follows simplified HTML schema after the first decryption (notice the two new *div* elements that were inserted into the page):

```
<html><head>...</head>
  <body>
    ...
    <div class="content">
      <h1>Test 06</h1>
      <p>Recursive decryption -- data on the page were encrypted
        two times using different keys.</p>
      <div>
        <div>
-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv/Q8y2Vdbxo3Q54TPCVcRGhgE1HkSFZgeLn/dtTK52T9Pk
...
-----END PGP MESSAGE-----
        </div>
        <div>
-----BEGIN PGP MESSAGE-----

hQGMA9GHiHuARJ+MAQv9GabshtHDlYlQrAbiE0/yQnCJkG0n4DlezV8pD+ZEjiLE
...
-----END PGP MESSAGE-----
        </div>
      </div>
    </div>
  </body>
</html>
```

The new encrypted elements on the page were detected thanks to the *MutationObserver* and the web browser extension was able to process them and display to the user. Here is simplified HTML schema after the second decryption (notice, that there are still encrypted elements):

```
<html><head>...</head>
  <body>
    ...
    <div class="content">
      <h1>Test 06</h1>
      <p>Recursive decryption -- data on the page were encrypted
        two times using different keys.</p>
      <div>
        <div>
          <audio controls>
            <source
              src="../encrypted/audio/audio1-test1.mp3.gpg"
              type="audio/mpeg"
```

```
          >
            Error - browser does not support mp3.
          </audio>
        </div>
        <div>
          <video width="320" height="240" controls>
            <source
              src="../encrypted/video/video4-test4.mp4.gpg"
              type="video/mp40"
            >
            Error - browser does not support mp4.
          </video>
        </div>
      </div>
    </div>
  </body>
</html>
```

The encrypted elements were again detected by the *GnuPG Decryptor* extension
and then decrypted. In the next test, we tried to encrypt entire HTML documents. We
encrypted the whole HTML file and added a link to the encrypted file into the different
HTML document. Then we encrypted everything between the *html* elements:

```
<html>
-----BEGIN PGP MESSAGE-----


hQGMA4gRpLdh4RX7AQwAoeLD/t32FS/IBQeaHq3fdgwXRcgTmb5MTvLRLJvrr4pe
...
-----END PGP MESSAGE-----
</html>
```

The browser detected missing *head* and *body* elements, created the missing el-
ements and put encrypted text into the new *body* element. The extension then de-
tected encrypted content in body element and decrypted it. The browser then removed
from decrypted content *head* element, but the content remained. The CSS styles were
loaded, although they are in the *body* element of the document. The encrypted HTML
document was decrypted as well. Here is decrypted HTML document:

```
<html><head></head><body id="GnuPG_DecryptorElemId-0">
<meta http-equiv="Content-Type" content="text/html; charset=utf-8">
<meta name="author" content="Jiří Matějka (xmatej52)">
<link
  rel="stylesheet"
  href="blob:https://www.stud.fit.vutbr.cz/7f6aad30-ed93..."
  id="GnuPG_DecryptorElemId-1"
>
```

```
<div class="menu">
  <a href="../html/test06.html">Test 06</a>
  <a href="../html/test07.html">Test 07</a>
  <a href="../html/test08.html">Test 08</a>
  <a href="../html/test09.html">Test 09</a>
  <a href="../html/test10.html">Test 10</a>
  <a
   href="blob:https://www.stud.fit.vutbr.cz/1bc051cf-7c8b..."
   id="GnuPG_DecryptorElemId-2"
  >
  Periodic Change
  </a>
</div>
<div class="content">
  <h1>Test 07</h1>
  <p>Recursive decryption -- Encryption of entire HTML page.</p>
</div>
</body></html>
```

Once clicked into the link to the encrypted HTML document, the page was loaded with its content and displayed. But external CSS style and JavaScript was not loaded and the browser did not allow to use links as well. The file was successfully decrypted, but due to its representation as BLOB, it was not functional. As a result of this test, we recommend to encrypt text data only between *body* tags and to not encrypt entire HTML files. But it is still possible to encrypt external CSS styles.

The next test was performed to see, how the web browser extension and the browser will handle encrypted javascript files. We prepared document, that with encrypted external javascript code. The javascript code will then insert encrypted content into the page every 10 seconds. Here is a simplified HTML schema of the document:

```
<html><head>...</head>
    <body>
        <div class="menu">...</div>
        <div class="content">
            <h1>Test 08</h1>
            <p>Testing javascript support.</p>
            <p id="insertHere">This content will be replaced.</p>
            <script src="../encrypted/javascript/javascript.js.gpg">
            </script>
        </div>
    </body>
</html>
```

The encrypted javascript was successfully detected and decrypted, but the browser refused to load the javascript nor execute it. Although removing the element from DOM

and putting it back to the page did not help, creating a copy of the element worked.
So after updating the source code of the web browser extension, the *GnuPG Decryptor*
supports encryption of external JavaScript code. To test the execution of an encrypted
internal JavaScript source code, we prepared another test page. Here is a simplified
HTML schema:

```
<html><head>...</head>
  <body>
    <div class="menu">...</div>
    <div class="content">
      <h1>Test 08</h1>
      <p>Testing javascript support.</p>
      <p id="insertHere">This content will be replaced.</p>
      <div>
-----BEGIN PGP MESSAGE-----

hQGMA63SHuyrZ3NZAQv/bPyUy4Qott9+nQXm/Wn70b3POzuDurci6Dsvy5nH4ftN
-----END PGP MESSAGE-----
      </div>
    </div>
  </body>
</html>
```

After decryption of the encrypted content, the web browser extension inserted new
script element with decrypted JavaScript source code into the page. But the page did
not execute an internal JavaScript code.

As the last test, we prepared a document, where all implemented the features and
will be possible to demonstrate. Below are several steps in decryption with a detailed
explanation of the decryption process. Here is the encrypted HTML page:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="author" content="Jiří Matějka (xmatej52)">
    <link rel="stylesheet" href="../encrypted/css/styles.css.gpg">
  </head>
  <body id="body">
-----BEGIN PGP MESSAGE-----

hQGMA9GHiHuARJ+MAQwAtPLNJ+FIaAnnv+Jvx+p4AqrIV/FNJGfYqPFQHJRAe9a8
...
-----END PGP MESSAGE-----
  </body>
</html>
```

The *GnuPG Decryptor* extension detected two elements and sent then to the native
application for decryption. Since we did not provide the native application passwords

for keys, the GnuPG application asked for keys and then decrypted content. The decrypted content was then sent to the content script, that inserted decrypted content into the page:

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <meta name="author" content="Jiří Matějka (xmatej52)">
    <link
      id="GnuPG_DecryptorElemId-0"
      rel="stylesheet"
      href="blob:https://www.stud.fit.vutbr.cz/b9fff6c3-8d9a...">
  </head>
  <body id="body">
    <div>
-----BEGIN PGP MESSAGE-----

hQGMA9GHiHuARJ+MAQwAit/BxUPbYjrKvlqdhLArclTo/P9KmrGbZG22wmirV/G7
...
-----END PGP MESSAGE-----
    </div>
    <div class="menu">
-----BEGIN PGP MESSAGE-----

hQGMA63SHuyrZ3NZAQv+LJF6ktKj3q96sDGujGVoYOmvLmks2/Kg9nPGbnI2gccy
...
-----END PGP MESSAGE-----
    </div>
    <div class="content">
      <h1 id="header">-----BEGIN PGP MESSAGE-----

hQGMA63SHuyrZ3NZAQwAi2Yn006ZU0kuz/O+uSFd5SKj2Rm95lSPLMGzMdanluYM
...
-----END PGP MESSAGE-----</h1>
      <p>
-----BEGIN PGP MESSAGE-----

hQGMA1XcsG3hc6mnAQv+K3+uYJ6WKbO8SBr/M26vvSGXvPx627CP6C1wc03jw7Kw
...
-----END PGP MESSAGE-----
      </p>
      <div>
-----BEGIN PGP MESSAGE-----

hQGMA1XcsG3hc6mnAQv+IvfkpQBKlaoaH+VwGDOWbVlw0/c3QKiqwGN5CFyrhjkL
...
```

```
-----END PGP MESSAGE-----
      </div>
      <div>
        <audio controls>
          <source
            src="../encrypted/audio/audio1-test1.mp3.gpg"
            type="audio/mpeg"
          >
          Error - browser does not support mp3.
        </audio>
      </div>
      <div>
        <video width="320" height="240" controls>
          <source
            src="../encrypted/video/video4-test4.mp4.gpg"
            type="video/mp4"
          >
          Error - browser does not support mp4.
        </video>
      </div>
      <p id="insertHere"></p>
      <button onclick="setContent( 'body' );">Click me</button>
    </div class="content">
  </body>
</html>
```

As soon as the content script inserted the decrypted content, the *MutationObserver* detected the change in the *link* and *body* elements and these elements were given for further processing. During the process, new encrypted elements were detected in the *body* of the HTML page and sent for decryption.

```
<html><head>...</head>
  <body id="body">
    <div>
      <script src="../encrypted/javascript/javascript.js.gpg"></script>
    </div>
    <div id="GnuPG_DecryptorElemId-1" class="menu">
      <!--5 links leading to other test pages-->
      <a href="../encrypted/html/periodicContentChange.html.gpg">
        Periodic Change
      </a>
    </div>
    <div class="content">
      <h1 id="header">Test 10</h1>
      <p id="GnuPG_DecryptorElemId-2">
        Testing all the functionalities.
```

```
      </p>
      <div id="GnuPG_DecryptorElemId-3">
        <img src="../encrypted/img/img1_test1.jpg.gpg" >
        <!--4 more encrypted images follows-->
      </div>
      <div>
        <audio controls>
          <source
            src="blob:https://www.stud.fit.vutbr.cz/ba4c332a-c875..."
            type="audio/mpeg"
            id="GnuPG_DecryptorElemId-4"
          >
          Error - browser does not support mp3.
        </audio>
      </div>
      <div>
        <video width="320" height="240" controls>
          <source
            src="blob:https://www.stud.fit.vutbr.cz/9d0a4f51-f8aa..."
            type="video/mp4"
            id="GnuPG_DecryptorElemId-5"
          >
          Error - browser does not support mp4.
        </video>
      </div>
      <p id="insertHere"></p>
      <button onclick="setContent( 'body' );">Click me</button>
    </div class="content">
  </body>
</html>
```

Once again, the *MutationObserver* detected the changes and changed elements were further processed and new encrypted files were detected, decrypted and decrypted content was displayed. In the case of JavaScript, a copy of the *script* element was created and the original element was replaced with the copy, thus the javascript was loaded and executed.

```
<html><head>...</head>
  <body id="body">
    <div>
      <script
        id="GnuPG_DecryptorElemId-6"
        src="../encrypted/javascript/javascript.js.gpg"></script>
    </div>
    <div class="menu">
      ...
```

```
        <a
          href="blob:https://www.stud.fit.vutbr.cz/c3963a82-a393..."
          id="GnuPG_DecryptorElemId-7"
        >
          Periodic Change
        </a>
      </div>
      <div class="content">
        <h1 id="header">Test 10</h1>
        <p id="GnuPG_DecryptorElemId-2">
          Testing all the functionalities.
        </p>
        <div>
          <img
            id="GnuPG_DecryptorElemId-8"
            src="blob:https://www.stud.fit.vutbr.cz/35405d32-9421..."
          >
          <!--4 more decrypted images follows-->
        </div>
        <!--Here are decrypted audio and video files-->
        <p id="insertHere">-----BEGIN PGP MESSAGE-----

hQGMA4gRpLdh4RX7AQv/e++Sh8fknwHaSd92p4xUQ9L1FJwnyOIf4Pby2V6A1Qe7
...
-----END PGP MESSAGE-----</p>
        <button onclick="setContent( 'body' );">Click me</button>
      </div class="content">
    </body>
</html>
```

Soon after decrypting the whole page, javascript inserted another encrypted con-
tent to the page. This content was again detected thanks to the *MutationObserver*
and successfully decrypted. The last step, in this test, will be executing the JavaScript
function *setContent* by clicking the button at the bottom of the page. By doing so,
we replaced all the content in the page with encrypted data (whole *body* of the docu-
ment was replaced by an encrypted content again). But this time, no data were sent
to the native application. The decrypted content is already available for the content
script and the content script will now only find the encrypted elements, detect that
encrypted data were already decrypted and will replace the encrypted content with
decrypted data from its memory. Here is a simplified HTML schema after executing
the JavaScript several times:

```
<html><head>...</head>
  <body id="body">
    <div>
    ...
```

```
    <!--Here are decrypted menu items and script.-->
    ...
    <div class="content">
      <h1 id="header">Test 10</h1>
      <p id="GnuPG_DecryptorElemId-48">
        Testing all the functionalities.
      </p>
      ...
      <!--Here are decrypted audio, video and image files-->
      ...
      <p id="insertHere">
        <div class="frame">
          <div>
            <p>Zašifrovaný audio soubor</p>
            <audio controls="">
                <source
                  src="blob:https://www.stud.fit.vutbr.cz/..."
                  type="audio/mpeg"
                  id="GnuPG_DecryptorElemId-375"
                >
                Error - browser does not support mp3.
            </audio>
          </div>
        </div>
      </p>
      <button onclick="setContent( 'body' );">Click me</button>
    </div class="content">
  </body>
</html>
```

5.3 Performance

With all the functionalities and features tested, we may proceed to test the performance of the web browser extension. We tested the performance on pages with encrypted content and the pages without any encrypted content (to see, how the web browser extension influences the browser performance on those pages).

5.3.1 Pages with Encrypted Elements

Since to slowest part of the process is downloading the content, we measured the time between the time the content was downloaded and the time, it was replaced. The measures must be therefore performed in the content script. We also measured the time it took the native application and GnuPG application to decrypt the data and send them to the content application. For measurements, we used pages from the previous testing. The average size of used encrypted image is appropriately 100 KB. The sizes of

encrypted videos are 11.5 MB, 24 MB, 28.5 MB and 73.7 MB. And the sizes of encrypted audio tracks are 4.7 MB, 7.5 MB, 8 MB and 9.4 MB.

Test 01

This test was performed on the *Test 10* page (Section 5.1.1). The page contains 17 images, 4 audio tracks and 4 videos and 1 encrypted text. The tests were performed 15 times and between each test, the cache of a browser was cleared. The first decrypted element was the text. The detection of encrypted elements took about 25 ms. The text was displayed on average of 0.5 s from its detection and was decrypted on average of 0.4 s. Each of the images was decrypted in approximately 0.4 s and the last image was displayed on average of 6.5 s since it was downloaded. The smallest audio track was decrypted in approximately 0.7 s and was displayed on average of 2.8 s after its download, the largest audio track was decrypted in approximately 0.8 s and was displayed 6 3 ms after it was downloaded as a last audio track. The smallest video was decrypted in approximately 0.8 s and was displayed in a browser after approximately 6.2 s. The largest video was always decrypted between 2.8 s and 3.5 s and was displayed between 65 s and 78 s for the user.

Test 10

This test was performed on the *Test 10* page (last test is the section 5.2.3). The test was performed again 15 times and between each test, the cache of a browser was cleared. The page contains an encrypted video (8 MB), two encrypted audio tracks (same files of size 4.7 MB), encrypted external javascript, encrypted external CSS, encrypted external HTML page, several encrypted images and several encrypted HTML elements with content. The content on the page is encrypted multiple times.

The first detection of the elements did not take longer than 6 ms. The decrypted *body* of the document was displayed in 0.5 s after its detection and was decrypted in approximately 0.4 s. The external CSS styles and javascript were decrypted in 0.4 ms and it took them 0.5 s to be displayed on the page since they were downloaded. The menu of the page was decrypted in approximately 0.5 s and was displayed 0.8 s since its detection. Each of the images was decrypted in approximately 0.4 s and the last image was displayed on average of 3.7 s after its download. The audio was decrypted in 0.5 s and was displayed after in approximately 7.5 s since its availability. The video vas decrypted in 1 s and was displayed 6.3 s since its download. Once the button on the page was clicked (to replace all decrypted content with encrypted one again), it took only appropriately 35 ms to display decrypted content again.

Threads in the Native Application

The performed tests showed that which each encrypted element on a page, there is an increasing delay between the availability of encrypted content and its replacement by decrypted one. We thought, that one of the causes of the delay is, that the native application does not manage to decrypt as fast as the content is processed to it. Due to this assumption, we updated the native application with *threading* library and the decryption was processed with threads. But this modification had only effect on pages

Domain	Minimum time [ms]	Average time [ms]	Maximum time [ms]
youtube.com	88	175.5	329
en.wikipedia.org	10	44	84
twitter.com	5	19.8	34
facebook.com	126	76	33
amazon.com	32	58.4	136
yelp.com	177	115.5	79
reddit.com	88	63.6	35
imdb.com	106	74	42
fandom.com	63	56.6	47
pinterest.com	13	8.6	3
tripadvisor.com	22	15.6	11
instagram.com	36	23.2	15
walmart.com	148	82.4	54
craigslist.org	28	14.2	6
ebay.com	140	80.4	58

Table 5.1: Time statistics of scannig pages without encrypted content.

with large encrypted files and it only allowed to process the smaller content, while the large content was being processed. The main advantage is the responsibility of the application – the user can open the GUI window during the encryption (without threads, the window is opened once all content is decrypted).

The main problem, why there is the increasing time it takes to display decrypted content is the increasing number of message needed to be exchanged between the native application, the background script and the content application. In case of large files, the data must be split into small blocks and the blocks need to be sent separately and then joined. All of this has a negative influence on the processing time of decrypted content and is the reason for the time it took the decrypted content to be displayed.

5.3.2 Pages without Encrypted Elements

Since encryption will try to detect encrypted content on every page the user will visite, we made measured the time it took the extension to test a web page for encrypted content. We measured the most 15 visited pages [3]. We visited the page, browsed them and measured times it took the web browser extension to scan them fro encrypted content. The time statistics can be seen in the table 5.1 below.

The web browser extension scans every element in the page for encrypted content. The more element the page has, the more time it takes to scan them. Most of the pages listed in the table 5.1 use APIs that allows loading the content of a page separately. In these cases, we measured the time it took to process it took the extension to scan these changes in the web page, not the whole page.

Chapter 6

Conclusion

In this thesis, I developed a web browser extension that detects and decrypts elements on a web page. I started by studying the GnuPG project and the OpenPGP standard. I focused on the implementation of the Linux application *gpg* and the OpenPGP message format to learn how to use the *gpg* application and how to detect encrypted content in OpenPGP format.

After getting familiar with the GnuPG project and the OpenPGP standard, I focused on the WebExtensions API, that can be used to implement extensions or add-ons for the Mozilla Firefox web browser. Once I learned about the WebExtensions API, I used gained knowledge to create a detailed design of an extension for the Mozilla Firefox browser that allows decrypting parts of a web page. I proposed a suitable approach to detects encrypted objects. Last but not least, I came up with a solution that is able to detect interactive changes to the page performed employing XHR API, Fetch API or Push API.

Great effort was also dedicated to the development process of the software. The extension was implemented in six iterations. Each iteration was designated before the implementation. We also tested the implementation for possible bugs and examined it to find its lacks and possible improvements. In addition, we implemented web pages to test all of the functionalities and features in the last two iterations of the development. We also tested the implemented browser extension on the most popular pages to measure, how the extension influences the browser performance.

The thesis resulted in implemented web browser extension that is able is able to detect and decrypt various encrypted elements on a web page and detects any dynamic changes that are made to the DOM by JavaScript or specific APIs (XHR API, Getch API, Push API, etc.). Therefore the software implemented in the thesis can be used as one of the comfortable ways of protecting sensitive content (family pictures, source codes, passwords, etc.) that is stored on public or third-party servers.

The web browser extension is implemented for the GNU/Linux operating system and the Mozilla Firefox browser. The extension was reviewed by Mozilla developers and then published on the Mozilla addons page as well as on GitHub.

In the future, the extension can be modified for different browsers, like Google Chrome, Opera or Microsoft Edge. The new native application can be created to support more operating systems then just GNU/Linux systems.

Bibliography

[1] Atkins, D., Callas, J., Donnerhacke, L., Finney, H., Shaw, D. et al. *OpenPGP Message Format* [online]. IHTFP Consulting, Inc., 2007 [cit. 2019-12-26]. Available at: `https://www.ietf.org/rfc/rfc4880.txt`.

[2] Google Developers. *What are extensions?* [online]. chrome [cit. 2020-01-05]. Available at: `https://developer.chrome.com/extensions`.

[3] Joshua Hardwick. *Top 100 Most Visited Websites by Search Traffic (as of 2020)* [online]. Ahrefs Pte Ltd., 2020 [cit. 2020-07-21]. Available at: `https://ahrefs.com/blog/most-visited-websites/`.

[4] Koch, W., Ellmenreich, N., Ashley, M., Cappelletti, L., Shaw, D. et al. *The GNU Privacy Guard* [online]. The GnuPG Project, 2019 [cit. 2019-12-25]. Available at: `https://gnupg.org/`.

[5] Lee, F. *Linux: Using GPG encrypted credentials for enhanced security* [online]. Fabian Lee, 2018 [cit. 2020-01-04]. Available at: `https://fabianlee.org/2018/10/30/linux-using-gpg-encrypted-credentials-for-enhanced-security/`.

[6] Mozilla Contributors. *Anatomy of an extension* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Anatomy_of_a_WebExtension`.

[7] Mozilla Contributors. *Browser Extensions* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions`.

[8] Mozilla Contributors. *Content scripts* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Content_scripts`.

[9] Mozilla Contributors. *Icons* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/icons`.

[10] Mozilla Contributors. *Native messaging* [online]. Mozilla, 2019 [cit. 2020-01-06]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/Native_messaging`.

[11] Mozilla Contributors. *Web_accessible_resources* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/manifest.json/web_accessible_resources`.

[12] Mozilla Contributors. *What are extensions?* [online]. Mozilla, 2019 [cit. 2020-01-05]. Available at: `https://developer.mozilla.org/en-US/docs/Mozilla/Add-ons/WebExtensions/What_are_WebExtensions`.

[13] OpenPGP.js. *OpenPGP.js* [online]. ProtonMail, 2019 [cit. 2019-12-29]. Available at: `https://openpgpjs.org/`.

Appendix A

Sequence Digrams

This section of the thesis contains all sequence diagrams that were created during the design phase during iterative development. Note that schema does not show communication between objects, as usually a sequence diagram does, but it shows communication between the content script, the background script, the native application and the GnuPG application.

There were created three diagrams. The first diagram shows communication between the content scripts of the *OpenPGP.js* prototype (Section 4.1) and *OpenPGPjs* library (Section 2.4). The second diagram shows a simplified schema of communication between the components of the web browser extension in the implemented *GnuPG_Decryptor prototype* and can be seen on the figure A.2. Next diagram shows an example of decrypting too large content. This feature was implemented in the *Large content prototype* (Section 4.3) and can be seen on the figure A.3. The last diagram (Figure A.4) is an example of the communication between the components of the web browser extension after updating the messaging system in the last iteration of developement process (Section 4.6).
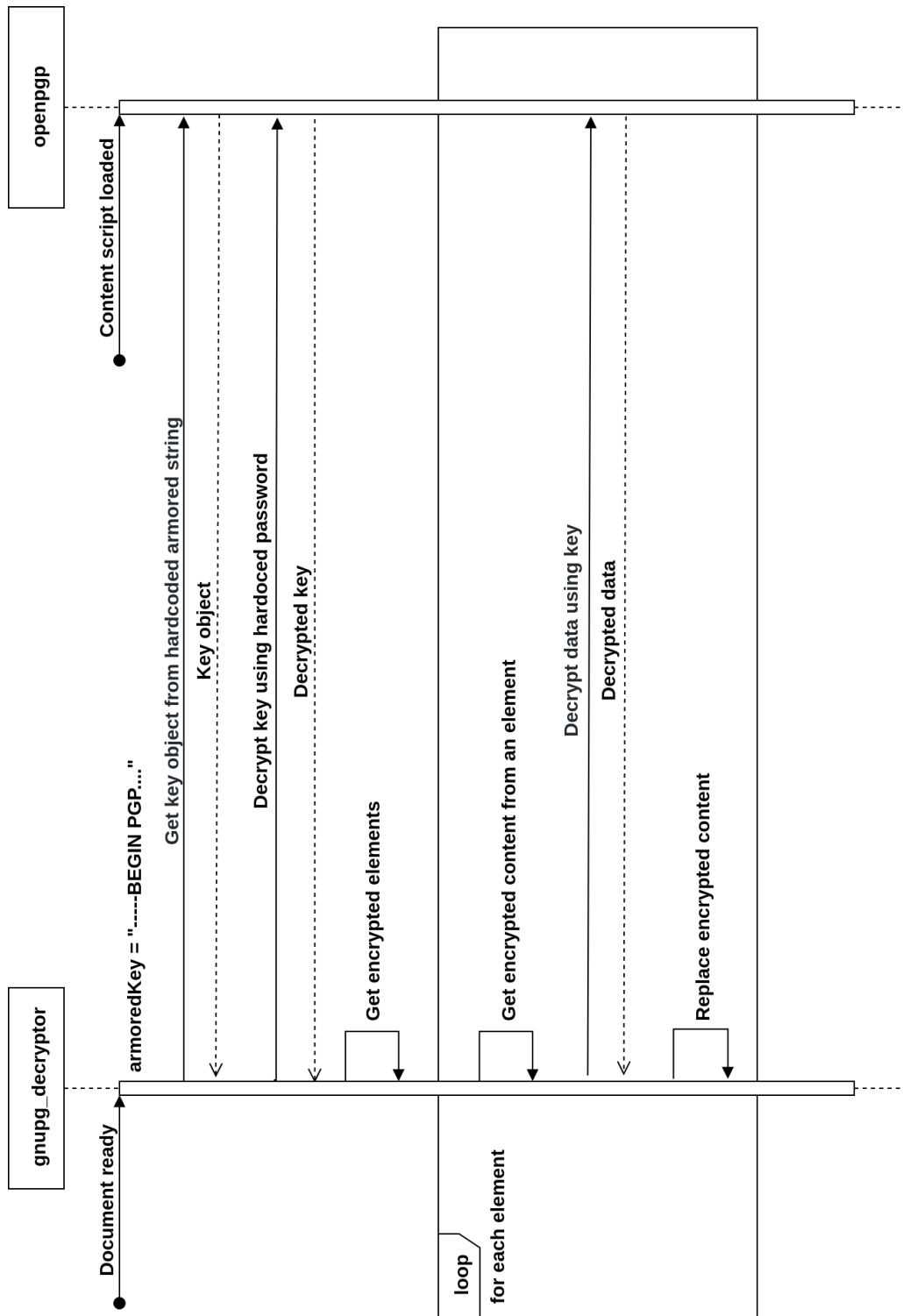
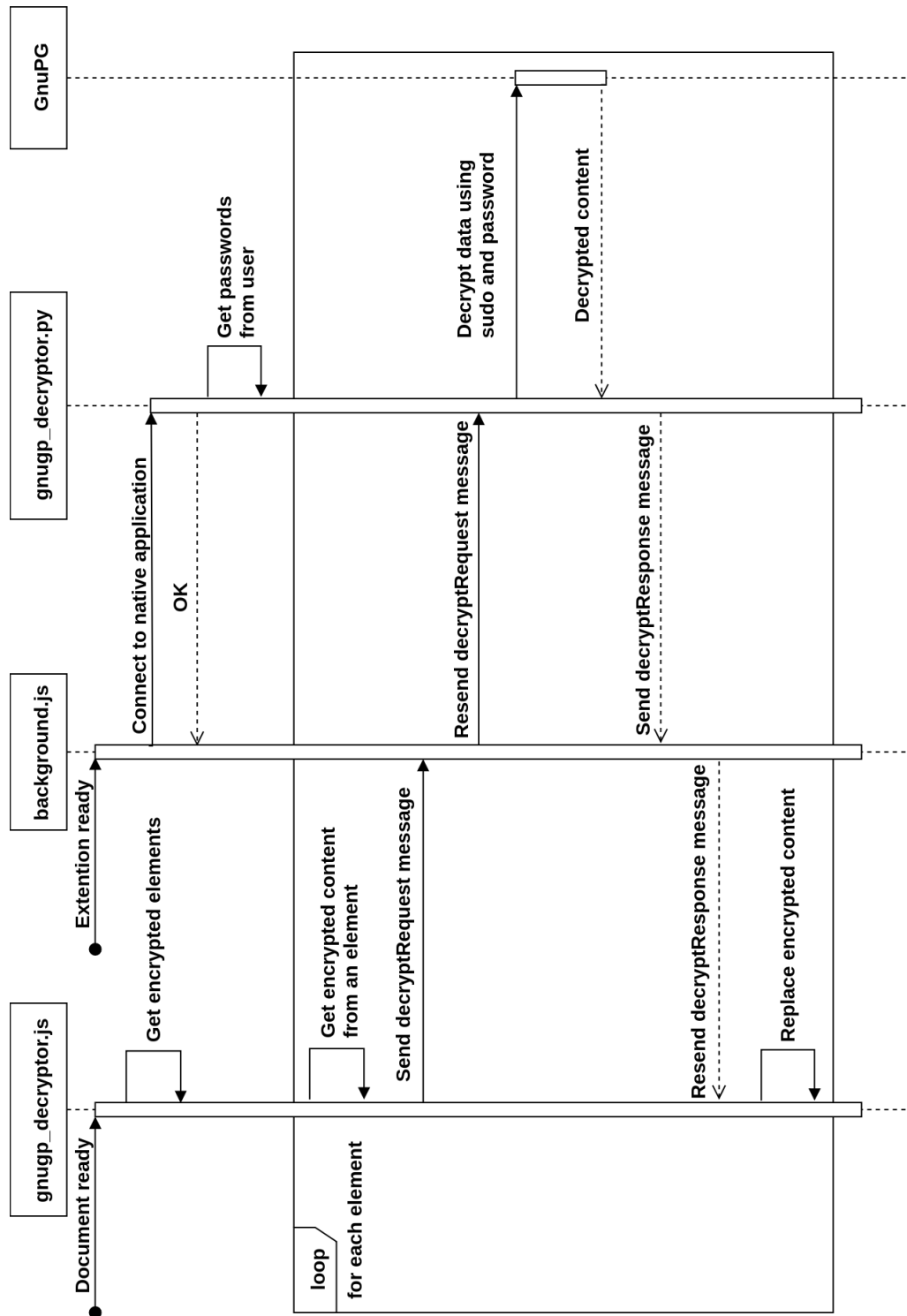Figure A.1: Communication between the content scripts of *OpenPGP.js* prototype.

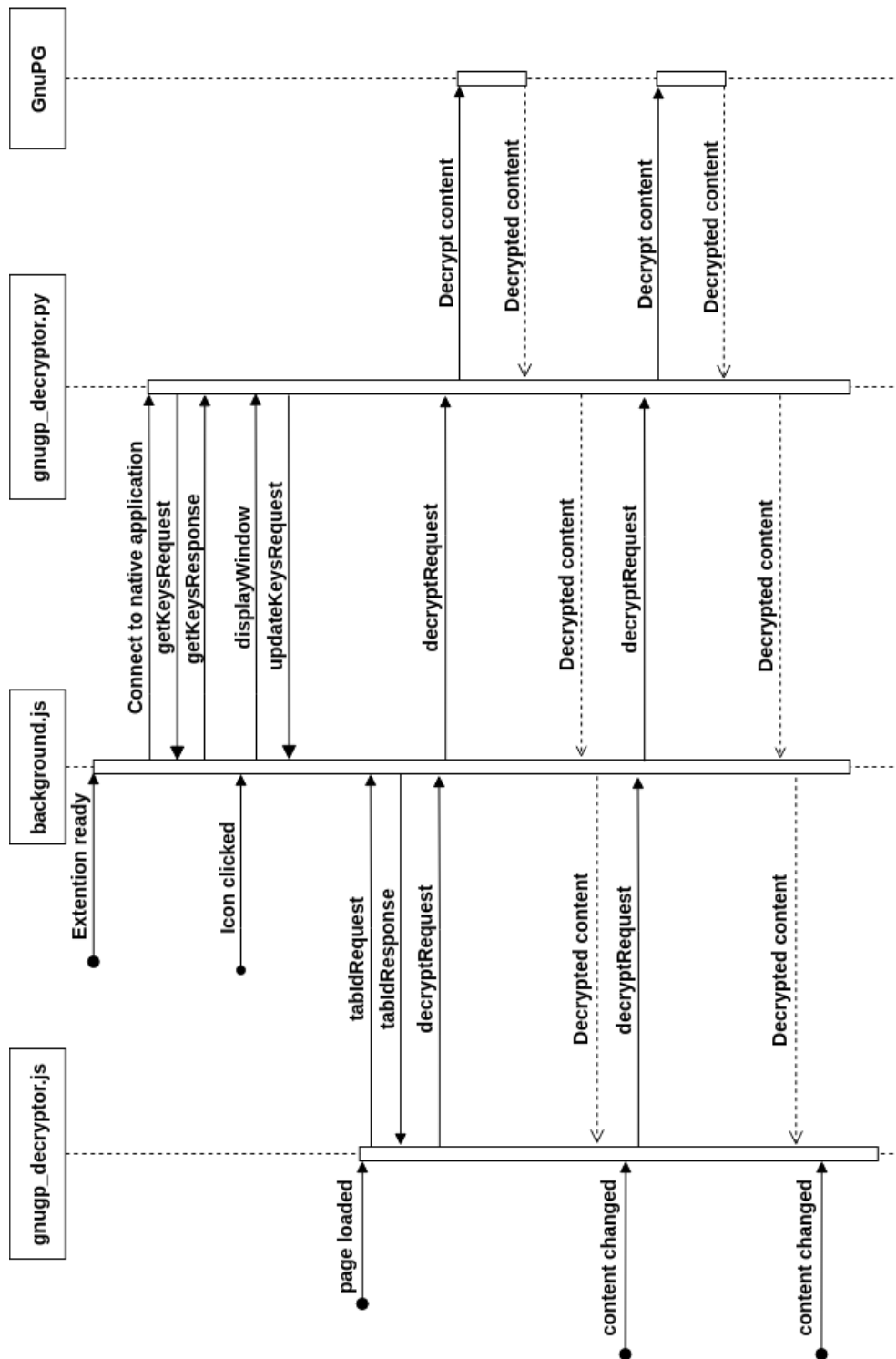Figure A.2: Communication between the web extension, the native application, and the GnuPG application.

Figure A.3: An example of decrypting too large content to be sent in a single message.

Figure A.4: An example of communication between the components of the web browser extension.

Appendix B

Contents of the Attached StorageMedia

On the attached storage media, the following files can be found:

- The digital form of the thesis,

- LATEXsource codes of the thesis,

- the GnuPG Decryptor extension,

- the native application for the GnuPG Decryptor extension,

- implemented prototypes,

- mplemented test pages (without encrypted audio and video files),

- GPG public and private keys,

- README file with password for keys and the description of the content.