C. L. Harding

Encrypted Facebook

Computer Science Tripos Part II Project

Queens' College

March 2011

Proforma

Original aims

The aim of this project is to apply a broadcast encryption scheme to content shared over Facebook by means of an extension for the Firefox web browser.

Work completed

The extension can augment the Facebook web UI with encrypted submission controls. It is also capable of automatically retrieving and deciphering content as the user browses the Facebook site. Text content and images are supported. Resource heavy algorithms are contained in a shared object library written in C++.

Special difficulties

Elements of the Part II 2009/2010 Information Theory and Coding course (not fully taught in 2010/2011) were required.

Declaration

I, Christopher Louis Harding of Queens' College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Computer Science Tripos Part II Project, May 2011

C. L. Harding

Contents

List of Figures

List of Tables

Acknowledgements

Many thanks to my supervisor Jonathan Anderson, to Andrew Lewis and Markus Kuhn for their advice on JPEG image coding and to Andrew Rice whose general help has proved invaluable.

1 Introduction

Facebook, at the time of writing, is the world's most popular social network service with over 500 million users active in the last 30 days [30]. This project describes Encrypted Facebook, a Firefox extension which aims to protect privacy by applying broadcast encryption to content shared through the Facebook platform. As well as textual content, encrypted images are supported using JPEG compression immune coding.

1.1 Background

Since its inception, Facebook's weak and often seemingly indifferent approach to online privacy has come under criticism [11]. As social networks mimic real life interactions, members are inclined to reveal more private details than they otherwise would, resulting in the accumulation of a large repository of sensitive information [22]. There are a number of ways this information can then be exposed: users misconfiguring privacy settings,[1] malice, error or neglect on the part of the social network[2] and its employees [12], acquiescent disclosure to government authorities [27] and unlawful access through phishing scams [32] and security exploits [47].

Encrypting content ensures its secrecy in any eventuality, provided the encryption key itself is kept safe. Tools do exist for manually encrypting online exchanges [15]. However, studies have suggested that software which requires manual management of cryptographic keys is not usable enough to provide effective security for most users [50]. In addition, interaction

[1] Facebook was recently forced to update its privacy controls due to growing pressure from the public and media [43].

[2] As was the case in 2010 when personally identifiable Facebook user details were sold to data brokers by third party developers [13].

over social networks is typically one-to-many, unlike applications like e-mail which these tools are designed for.

Solutions targeting social networks specifically have been proposed. Some are still overly reliant on manual key management; others fail to fully protect the user's privacy or support the most popular forms of content (see sections 1.4, 2.1 and 2.5). Thus far all proposals have required the use of a third party in ways which are unlikely to be scalable.

Privacy conscious social networks have emerged with built-in encryption and support for decentralised hosting [42][10]. Unfortunately, network externalities make it difficult for newcomers to compete with Facebook since the utility of a social network is intrinsically tied to the size of its user-base. [3]

1.2 Objectives

The aim of this project is to provide an enhanced privacy solution for existing Facebook users which is:

- **Privacy preserving** given the scenarios presented in Section 1.1.
- **Usable** and therefore accessible to the typical Facebook user.
- **Scalable** given the vast size of Facebook's user-base.
- **Incrementally deployable** in order to avoid some of the problems of network effects.

1.3 Limitations

Some related objectives we consider to be outside the scope of an initial implementation:

- Securing the structure of the social graph itself.
- Ensuring the integrity, authenticity or non-repudiation of communications. In theory the public key scheme could be extended to do so but this would go beyond mere privacy control.

[3] Some suggest the value of a social network grows linearithmically, quadratically or even exponentially with the number of users [46] [25].

- Complete protection against middle-person attacks (see Section 2.7.1).
- Guaranteeing availability of content. Completely severing reliance on Facebook would be tantamount to building a new network.
- Concealing the existence of content itself, or concealing the fact that it is encrypted. Steganography is employed but purely for aesthetic effect (see Section 3.4.3).
- Full cross-platform and/or cross-browser support (see Section **??**).

1.4 Existing work

Many applications for generally encrypting online data exist, most in the form of browser extensions [15]. There are also several applications targeting Facebook specifically which we introduce below. Section 2.1 describes how they relate to our own approach.

uProtect.it Client side JavaScript application which inserts UI controls and intercepts content. Content is encrypted, stored and decrypted all on a third party server.
`http://uprotect.it`

FaceCloak Firefox extension which posts fake content to Facebook, using it as an index to the encrypted content on a third party server. Running your own server is possible, though only users on the same server can communicate.
`http://crysp.uwaterloo.ca/software/facecloak/`

flyByNight Content is submitted through a Facebook application, encrypted using client side JavaScript and passed via Facebook to a third party application server. Proxy re-encryption is used for sending to multiple recipients.
`http://hatswitch.org/~nikita/`

NOYB Content is stored in plaintext but profiles are anonymised. The mapping from real-to-fake profiles is known only to a user's friends.
`http://adresearch.mpi-sws.org/noyb.html`

2 Preparation

In this chapter we formulate the objectives presented in Section 1.2 into a set of design principles, drawing on existing work. We justify the use of broadcast encryption and define a suitable scheme. We describe possible deployment strategies and briefly review Firefox extension development and the Facebook platform. We also look at the specific problems associated with encrypting images and draft a security policy. Finally, we derive a concrete set of requirements and outline an appropriate software development methodology and testing plan

2.1 Design principles

We begin by outlining several key principles which were used to constrain the rest of the design and development process. We briefly describe how they enforce the stated aims of privacy preservation, scalability, usability and incremental deployment and how they relate to existing solutions. Note that here use of the term third-party excludes Facebook itself.

2.1.1 Encryption of shared content

It is possible to preserve privacy by encrypting or otherwise concealing the link between a real life user and their online identity. This is the basis of NOYB [24]. Arguably, however, privacy is only poorly preserved due to problems of inference control [2].[1] Incremental deployment is also hampered since non-users can't see the real names of the users they interact with [34].

[1] An obvious example is that many users will be easily identifiable simply from the photos they upload.

The alternative is to encrypt shared content itself in some way or another, restricting access to only those who possess the appropriate key even in the event of exposure.

2.1.2 Independence from third-party servers

In addition to using encryption, flyByNight, FaceCloak and uProtect.it opt to migrate content from the Facebook platform to an external third-party database.

We consider the practice of outsourcing content to conflict with our stated goal of scalability. Storing and delivering encrypted content requires at least the resources needed for storing and delivering the cleartext. Facebook are currently able to offer a free service by serving highly targeted advertising to members based on the structure of the social graph [29]. This revenue stream would be largely unavailable to any solution hosting a database of encrypted content.

Third-party servers can also be employed for performing encryption and/or decryption (as with uProtect.it and flyByNight) or as part of a public key infrastructure. Again, the resources required by the server would scale linearly in proportion to the amount of content exchanged.

2.1.3 Secret key security

Any encryption scheme will require some form of key whose secrecy is required.

It is possible to use a trusted third-party to store and distribute secret keys in a so-called key escrow arrangement. Key management can even be taken out of the users hands entirely, improving usability. This is the basis of uProtect.it [48]. However, confidentiality is only weakly assured since trust has simply been deferred from Facebook to the third-party and many of the scenarios raised in Section 1.1 still apply.

Another possibility is using secret keys derived from a password. flyByNight, for example, allows users to download a password-protected private key from their server. We could also store password protected keys in-band (i.e. on Facebook itself) or generate a key simply by hashing a password. Relying on the user to memorize a password rather than manage secret keys also improves usability. Unfortunately the entropy of user chosen passwords is far less than that of randomly generated keys [6].

By ensuring we use randomly generated keys that are only ever stored on the user's device(s) we trade usability for better privacy protection. Appropriate key lengths are discussed in Section 2.7.2.

2.1.4 Minimal use of OOB channels

Secure OOB (out-of-band) channels[2] can be used to transmit content, update messages, keys or other information as part of an encryption scheme. Since these channels are, by definition, external to the Facebook platform it can be hard to automate such exchanges and much is still required from the user. FaceCloak, for example, requires users to transmit messages over secure email when adding friends. The process is partly automated, however the user must set up an email client and install and configure PGP themselves [34].

By limiting the use of OOB transmission we mitigate usability concerns regarding manual key management. The exception is when installing a secret key across more than one device — since transporting a secret key by any other means would compromise the principle of secret key security.

2.2 One-to-many communication

Communication over social networks is typically one-to-many whereas cryptography traditionally considers one sender and a single recipient. We look at existing solutions and outline the broadcast encryption scheme we adopted for Encrypted Facebook.

2.2.1 Existing solutions

uProtect.it, flyByNight and FaceCloak tackle this problem in the following ways:

1. If content is both hosted and encrypted/decrypted remotely, as with uProtect.it, one-to-many support is trivial. The user simply authenticates with the server and is sent the cleartext.

2. If a third-party is able to perform computation a technique called proxy re-encryption can be used, as with flyByNight [33]. Here the

[2] Such as encrypted email or face-to-face exchange.

server changes the key under which the content may be decrypted on demand, without ever being able to read the cleartext itself [4].

3. Distributing keys over OOB channels can permit one-to-many communication. A FaceCloak user, for example, shares a single decryption key OOB among friends [34].

(1) and (2) are incompatible with the design principle of third party independence, whilst (3) violates the principle of limited OOB channel use.

2.2.2 Broadcast encryption

An alternative to the solutions in Section 2.2.1 is to use a broadcast encryption[3] scheme.

In general, schemes can be characterised by the size of each user's private and public key and the amount of transmission overhead that must be sent with each message, given the number of recipients. The scheme we use here has a relatively large transmission overhead, however unlike other more complex schemes it doesn't require the use of key-update messages. Transmitting key-updates OOB or though a third-party would violate our stated design principles. Transmitting them in-band would be possible, but since Facebook is a best-effort service synchronisation and lost updates would likely be a problem.

Let U be the set of users with Encrypted Facebook installed and $R \subseteq U$ be a set of intended recipients. For simplicity we describe broadcast encryption as a key encapsulation mechanism:

Definition 2.1. Given a suitable asymmetric encryption scheme P and a suitable symmetric scheme Q, we define our broadcast encryption scheme as the triple of algorithms (SETUP, BROADCAST, DECRYPT) such that:

- (SETUP) takes a user $u \in U$ and constructs their private key $priv_u$ and public key pub_u using scheme P.
- (BROADCAST) takes the list of privileged users R, generates a session key k using scheme Q and broadcasts a message b where:
  - b is the list of pairs (u, k_u) such that $u \in R$, where k_u is the session key encrypted under pub_u.

[3] Broadcast encryption is defined in Appendix **??**.

- A user $u \in U$ runs DECRYPT$(b, u, priv_u)$ that will:
  - If $(u, k_u) \in b$, extract the session key k from k_u using $priv_u$.
  - DECRYPT fails, if $(u, k_u) \notin b$ or if, equivalently, $u \notin R$.

[28] provides a proof that this scheme is at least as secure as the underlying encryption schemes P and Q. A more advanced scheme which also doesn't require key-updates is proposed in Appendix **??** and discussed in Section **??**. It was not pursued initially because it was thought that a working implementation couldn't be guaranteed.

2.3 Intercepting Facebook interactions

In order to encrypt content it must be intercepted before being submitted to Facebook. We describe the possible places at which this can occur (Figure 2.1) and briefly justify the approach we took with Encrypted Facebook.

2.3.1 Possible deployment strategies

a) On a remotely hosted proxy server. Provides easier multi-OS and multi-browser support.

b) On a proxy server running on localhost. Providers easier multi-browser support.

c) Within the browser, outside the browser sandbox. Extensions and plugins exist here and have elevated privileges over normal site code. Other examples include signed Java applets, ActiveX controls and to a lesser extent inline Flash and Silverlight applications.[4] FaceCloak takes this approach.

d) Within the browser, entirely inside the browser sandbox - using only JavaScript and HTML. uProtect.it and flyByNight both take this approach.

e) Outside the browser as part of a bespoke Facebook client application.

[4] Flash applications, for example, are restricted but can provide basic filesystem access [20].

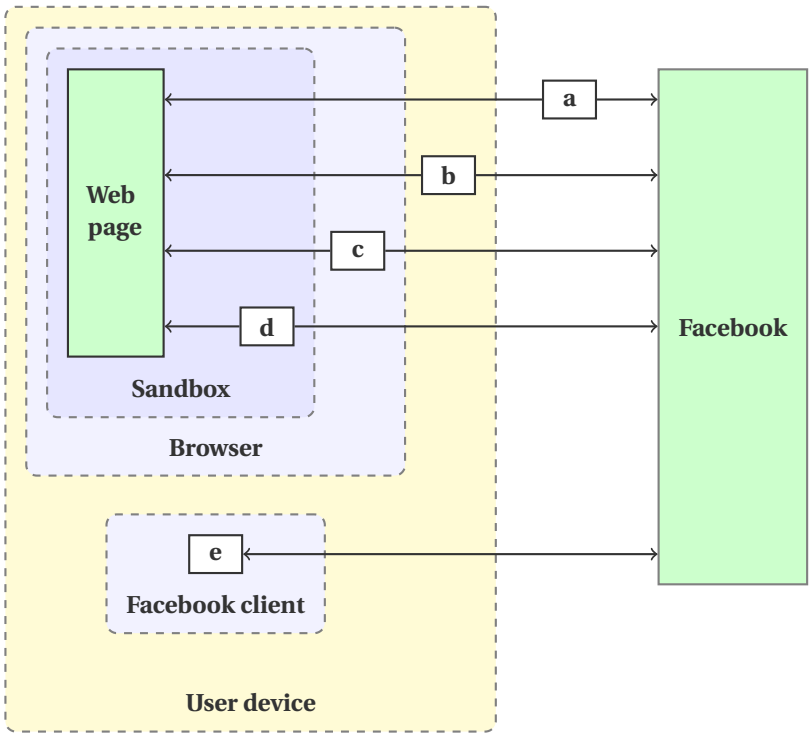


Figure 2.1: Possible deployment strategies for intercepting interaction with Facebook.

Approach (a) would conflict with the design principle of third party independence. The browser sandbox prevents local filesystem access, ruling out (d) if private keys are generated and stored securely. We took approach (c) since (b) and (e) are considerably more complex, at some cost to cross-platform compatibility.

2.4 Mozilla Firefox extension development

The project was developed as an extension for Mozilla Firefox, the world's most popular browser (as of January 2011). Manipulating web pages through the DOM (Document Object Model) is well supported by browser extensions since the browser interface chrome is often built on existing web technologies. Porting to other browsers is discussed in Section **??**.

Firefox extensions are written in JavaScript with partial support for binding library code written in Python or C/C++. Performing cryptography in

Language	Library	Time (ms)
Python 2.6.6	pycryptopp 0.5.17	1,220 ms
C++ 98	Botan 1.8.11	92 ms
JavaScript 1.6 (in Chrome 12.0.712)	JavaScrypt (last updated December 2005)	1,685,000 ms

Figure 2.2: Approximate time for 256-bit AES encryption of 1000 1.5 MiB random messages.

JavaScript is possible but comes with severe performance difficulties [33]. Table 2.2 compares approximate performance for each language based on some provisional tests.

Since long delays would hamper usability we opted to use C/C++ for computation intensive operations. Native code can be executed from within Firefox in three ways:

- Creating an XPCOM component. These are linked against a single Gecko[5] version; supporting multiple versions is possible but non-trivial [37].
- Loading native libraries with `js-ctypes`. Introduced in Gecko 2.0 [38].
- Using `nsiProcess` to invoke an external stand-alone application. Capturing output can be difficult.

Since building an entire XCPOM component would be excessive and give little advantage by way of multi-version compatibility, the newly introduced `js-ctypes` module was used to load native code. This restricted us to Firefox version 4.0 and above, however Gecko 2.0 also has the advantage of providing better support for working with the local filesystem and manipulating images in the DOM.

[5] Gecko is the layout engine used by Firefox.

2.5 The Facebook platform

Facebook represents all entities (e.g. users, messages, photos, events) uniformly as nodes in the social graph. Each node has a unique Facebook ID, several attributes including an object type, and one or more connections to other nodes.[6]

We make the generalisation that interaction with Facebook amounts to creating and retrieving objects in the social graph. Our goal then is to encrypt and decrypt the attributes of certain object types - the body of a message object, for example. We do not attempt to encrypt or otherwise conceal connections between objects; this is an non-goal as stated in Section 1.3.

We describe the most popular forms of object submitted and discuss issues relating to the connectedness of user nodes and the signal-to-noise ratio in network feeds. We then briefly look at interfacing with the Facebook platform.

2.5.1 Content types

Encrypting all possible types of object would be prohibitively complex. Ideally we should encrypt those most frequently used. From the data available these are Comments, Messages, Images and Posts (Table 2.1).

Without relying on a third-party server all broadcast encryption transmission overheads must be stored on Facebook itself, in one form or another. Images and blog-style notes are obvious targets for storage utilisation due to their large capacity (Table 2.1). In particular, the body of a note can contain over 120 KiB of information since each character represents one 16-bit Unicode code point. Images are subject to lossy compression which is discussed in Section 2.6.

Each user's profile has a `Bio`[7] field with a character limit of 64,536. With no other capacious attribute that can be easily queried from a user ID this is the obvious place to store a user's public key.

[6] This is actually a slight simplification. For full details refer to the Facebook Graph API reference documentation.

[7] Sometimes referred to as the `About Me` field.

Activity	Frequency (per second)	Limitations
Comment	8,507	8,000 chars.
Message	2,263	10,000 chars.
Image	2,263	720 × 720 pixels
Friend request	1,643	
Status update	1,320	420 chars.
Wall post	1,323	1,000 chars.
Event invite	1,237	
Photo tag	1,103	
Link	833	
Note	Unknown	65,536 chars.

Table 2.1: Facebook objects and connections, their limitations and approximate frequency of creation [45]

2.5.2 Connectedness

Since broadcast encryption has a transmission overhead proportional to the number of intended recipients, care must be taken to ensure the system works with large enough recipient groups. The number of recipients is bounded by the number of friends a user has, or equivalently by the degree of the user's node in the social graph. Empirical estimates for the average number of Facebook friends range from 130 to 170, with some evidence suggesting the distribution drops off sharply at around 250 [30][19].

The Dunbar number is a theoretical cognitive limit to the number of people a user can maintain relationships with and has been applied to both online social networks and to real-life interactions. Estimates range from around 150 to 300 [26] [35], suggesting that average node degree is unlikely to increase dramatically as Facebook expands further.

2.5.3 Signal-to-noise ratio

Activity within the social graph causes notifications to be posted to feeds. The news feed, for example, is shown to users on logging in (Figure 2.3 We define the signal-to-noise ratio of a feed as the proportion of useful items

Figure 2.3: News feed as presented on login.

to non-useful items.[8]

In order to permit incremental deployment any system must ensure that its users can coexist with non-users. Ideally the impact on signal-to-noise ratio should be limited.

2.5.4 Graph API

Facebook does provide a JavaScript SDK for interfacing with the Facebook platform, however it is poorly documented and doesn't allow uploading images - since most JavaScript applications are designed to run inside the browser sandbox without local filesystem access. Instead, we may use the Facebook Graph API directly.

Authentication is performed using the OAuth 2.0 protocol. Applications register with Facebook and present their application ID upon request. The user then confirms that the application may use the requested permissions and an access token is generated. This token is then used in future exchanges.

[8] By non-useful items we typically mean spam, but in our case this could be transmission overheads as part of the broadcast encryption scheme or even the encrypted content itself.

Unfortunately some actions are poorly supported by the Graph API, if at all. Some particular problem cases, and their solutions:

- When publishing images, getting a correct handle to the image is difficult due to JavaScript's poor support for working with local files. The workaround requires creating an invisible form with a file input element and extracting the file handle from there.
- Facebook currently uses two types of album ID, one which appears within web pages and one which can be used for publishing through the Graph API. An additional API query is required before uploading images to translate from one format to the other.
- Modifying the `Bio` attribute is unsupported entirely. The workaround requires creating an invisible iframe on the current page and manipulating a form on the Facebook site within.

Further workarounds and API subtleties are given in Appendix **??**.

2.6 Encrypting images

Regardless of the input format, Facebook encodes all uploaded images using lossy JPEG.[9] We therefore require some form of JPEG-immune coding for the binary output of encryption, so that after undergoing compression we can exactly recover the original bytes.

Appendix **??** describes Facebook's JPEG compression process. To motivate the need for a more complex scheme we evaluate two naive attempts at encoding data in images, before proposing a more advanced approach. Note that we only consider greyscale images.[10]

2.6.1 Naive data insertion

One approach would be to encode directly into greyscale pixel values with enough redundancy to overcome the information lost during compression. A second might be to similarly encode data into the DCT coefficients of

[9] Even if a file is already in the output format the compression process is repeated and information is lost.

[10] Chrominance subsampling means RGB images provide a less than 50% increase in capacity over greyscale. See Appendix **??**.

an existing JPEG image, since through the process of decompression and re-compression only a small amount of information is lost. The main source of information loss for the second approach is from capping DCT coefficients which don't map back to greyscale values in the range 0-255.

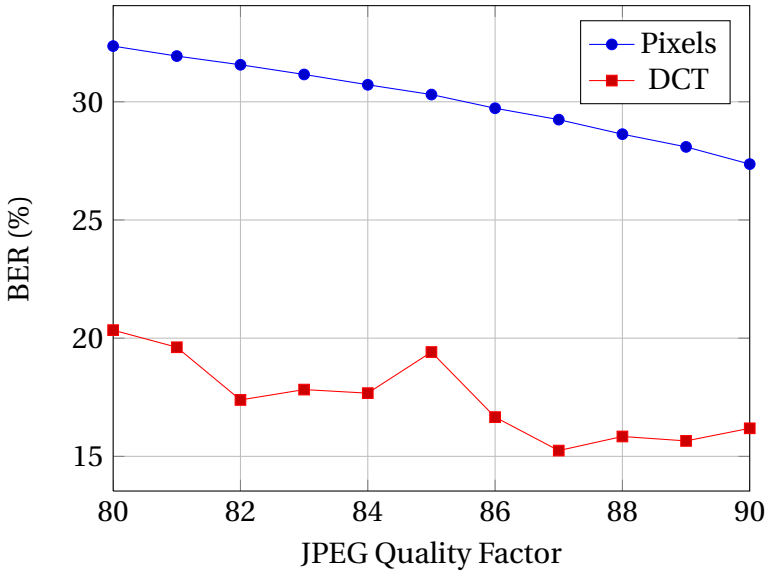


Figure 2.4: Bit error rate for varying quality factors.

Figure 2.4 graphs the bit error rate for both methods over a range of quality factors, using the IJG (Independent JPEG Group) `libjpeg` library for compression. Quality factor 85 most closely matches the compression signature used by Facebook (see Appendix **??**). For encoding in DCT coefficients a special bitmask, generated from the quantisation matrix, is used in an attempt to reduce the incidence of capping (details are given in Appendix **??**). Modelling the compression process as a binary symmetric channel, Figure 2.5 graphs the theoretical per-image capacity of each method for lossless data transmission (details for this calculation are given in Section 4.1.2).

Given a recipient group size of 400 and a ciphered session key size of 1024-bit (see Section 2.7.2) the transmission overhead alone would be at at least 50 KiB.[11] A 720×720 image encoded at low quality (IGJ factor 10) would still require around 25 KiB of storage space [23]. This means neither

[11] 50 KiB not including the list of recipient IDs, initialisation vector and any padding or size tags.

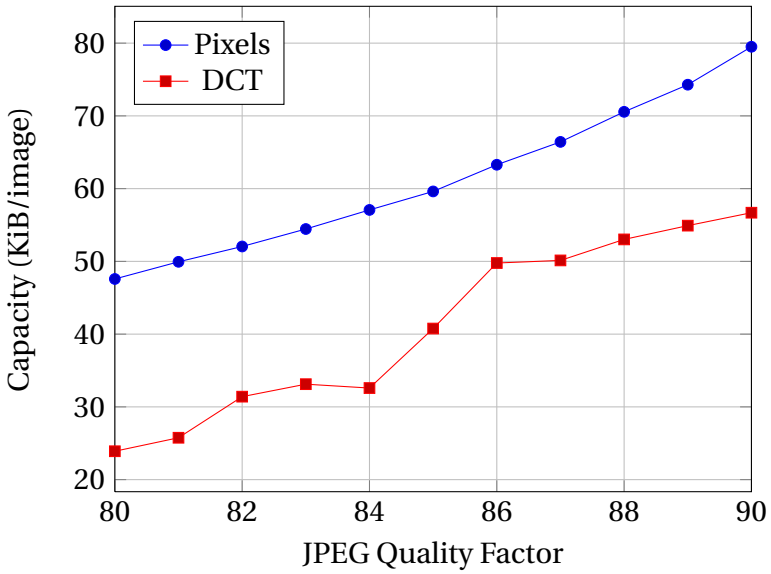


Figure 2.5: Per-image channel capacity (measured in KiB/image) for varying quality factors.

of the naive approaches would be suitable even given an error correction scheme which performs at or near the Shannon limit.

2.6.2 Advanced data insertion

One initial optimisation for any coding scheme would be to map binary data to appropriate length Gray codes to ensure that only single bit errors occur from erroneously outputting an adjacent codeword. In addition, we present two possible schemes, henceforth referred to as the HWT (Haar Wavelet Transform) method and the n-bit scaling method.

HWT method

JPEG DCT compression selectively quantises perceptually redundant high frequency components. Wavelet transforms allow us to embed data in the low frequency sub-band of the carrier signal and can be performed reversibly by using an integer lifting scheme. Xu et al demonstrate that data encoded in low-frequency HWT approximation coefficients can survive JPEG decompression when combined with an error correction scheme [52].

N-bit scaling method
This method maps the n-bit input space on to the 8-bit pixel space by scaling the input by a factor of $\frac{2^8-1}{2^n-1}$. The inverse process amounts to outputting which interval the pixel value lies in. This method works on the assumption that large changes in single pixel intensity are unlikely to occur during compression.

Both these schemes are sub-optimal and their exact properties (compression time and error rates) are unknown. We made it a requirement that the project should take a modular approach to conduit image implementations: firstly to ensure that both of the proposed schemes can be implemented simultaneously and their performances compared; secondly, to aid future development of a more optimal solution.

2.7 Security Policy

To truly protect privacy we must consider proper security engineering practices. Of particular concern is introducing new vulnerabilities into the browser platform. We perform a threat analysis and describe issues relating to the underlying cryptographic scheme.

2.7.1 Threat analysis

The threat analysis is included in its entirety in Appendix **??**. We take an attack-centric approach, defining risk as $Vulnerability \times Threat \times Impact$ according the methodology described by [36]. Below we summarise several potential threats and possible counter measures, proportionate to the risk posed:

Attack 1 Attacker breaks the encryption scheme by brute force methods.

Measures Ensure key lengths meet the appropriate standards. See Section 2.7.2.

Attack 2 Attacker gains access to a user's device through the download and execution of a file.

Measures Ensure only legitimate JPEG and public key files are downloaded. Public key files can be vetted on their character set and size, JPEG's by their extension.

Attack 3 Attacker gains access to a user's device by injecting code into `eval()`, which is then executed outside the browser sandbox.

Measures Wrap all calls to `eval()` with a `secureEval()` function which attempts to prevent malicious use.

Attack 4 Attacker gains access to a user's Facebook account through browser code injection.

Measures Sanitize all user inputs and ensure sanitisation can't be bypassed e.g. through the UTF-8 decoder[44]. Also sanitize output whenever inserting code into the DOM.

Attack 5 Attacker carries out a middle-person attack by intercepting and switching public keys.

Measures Informing the user whenever public keys are updated removes a large amount of risk. Complete protection would be impractical without use of OOB channels.

Attack 6 Attacker causes DoS (Denial of Service) by creating a malicious object.

Measures This need not be a large problem since decryption should be built to fail gracefully (see Section 3.2.2). Include some simple run time checks to limit iteration and special characters (combined with proper sanitisation) to prevent tags-within-tags.

2.7.2 Underlying encryption schemes

NIST (National Institute of Standards and Technology) publish guidelines on cryptographic schemes and protocols. Since 2010 NIST have recommended using at least 112-bits of security [3].

The ideal scheme would be based on elliptic curve cryptography since comparable strength public keys are much smaller than for finite field or integer factorisation methods (see Table 2.2). This is important since the block size of the cipher depends on the public key; this in turn determines the size of the transmission overhead. Unfortunately ECC is less common in open libraries due to patent concerns.

We opted to use RSA and AES due to their wide availability in libraries like Botan. Given the possible advantages of using ECC we made it a requirement that the cryptographic components of the library be extensible. This also accommodates for future increases in key lengths, perhaps due to revised recommendations.

Bits of Security	FFC	IFC	ECC
80	1024	1024	160-223
112	2048	2048	224-255
128	3072	3072	256-383
192	7680	7680	384-511
256	15360	15360	512+

Table 2.2: Table of public key length equivalences [3]

2.8 Requirements specification

Based on the contents of this chapter we derive a set of concrete requirements.

Requirement 1 The extensions will be able to broadcast-encrypt, submit, retrieve and decipher the following objects:

- Status updates
- Wall posts
- Comments
- Messages
- Images

Requirement 2 The size limits for encrypted text objects will be no smaller than the current limits Facebook imposes, given in Section 2.5.1.

Requirement 3 The size limit for encrypted images will be no less than 50 KiB. This is roughly equivalent to a 720×720 image encoded at medium quality (IJG factor 25) [23].

Requirement 4 To ensure the goal of scalability is met, all requirements will be met given recipient groups sizes up to 400, based on the discussion in Section 2.5.2.

Requirement 5 To ensure the goal of usability is met, all response times will lie within acceptable limits, in accordance with [40].

Requirement 6 To ensure privacy is protected, the project will adhere to the security policy described in 2.7.1.

Requirement 7 The number of news feed entries generated by encrypted submission will be no more than the number generated by normal content submission. This ensures that the effect on signal-to-noise ratio is kept to an absolute minimum - the only perceivable 'noise' is the encrypted content itself.

Requirement 8 There are uncertainties and/or trade-offs associated with certain approaches to encryption and image coding. It is also clear that in some cases the optimal approach is well beyond the scope of this project. Therefore, we make it a requirement to adopt a modular structure that facilitates switching between differing schemes and permits future extension.

2.9 Development

An approximately agile development style was adopted loosely based on [1]. Uncertainties in the optimal approach or combination of approaches for encryption, image coding and error correction meant that being flexible and able to respond to changes was crucial. Delivering functional prototypes at various stage in the development life-cycle also fits with the project's modularity requirement. In addition, since the bulk of the project concerns encapsulating data in some layer of encoding the natural approach would be to build successive prototypes, each implementing a new layer.

2.9.1 Development environment

Git was used for revision control and backup. The working repository is stored on a laptop and pushed twice-daily to a remote repository hosted by ProjectLocker.

Komodo edit was used as a development IDE, as recommended by Mozilla. C++ coding style followed the GeoSoft guidelines [17] and documentation was generated automatically using Doxygen.

Firefox 4.0 pre-beta was obtained through APT by adding the Mozilla Daily Build Team PPA. The build process consists of compiling and linking the C++ shared library, then packaging this (together with the rest of the extension code) and automatically installing to a number of Firefox development profiles.

GNU Octave (a free MATLAB-like application) was also used for provisional evaluation and to generate some of the results in this section.

2.9.2 Testing plan

We loosely adopt the agile testing principles of test driven development, rather than a test last approach [8]. Ensuring the feedback loop between testing and implementation is kept as short as possible allows us to maintain flexibility, which is useful given the uncertainties over the optimal approach.

Special attention needed to be made to security relevant testing such as input sanitisation and proper functioning of the UTF-8 decoder. Boundary-value analysis was used to generate sets of test scripts (included in Appendix **??**).

To help ensure the goal of usability was met, usability testing in the form of cognitive walkthroughs were performed. During development many passes of the walkthroughs were performed and alterations made, until a credible success story could be constructed for each task. Excerpts from final success stories are presented in Section **??**.

3 Implementation

We present the key aspects of the implementation. Some of the subtleties are omitted, in particular the specifics of integrating with the Facebook web UI and the workarounds required to correctly authenticate and interface with the Graph API. We begin with a general overview of the project structure, before outlining the page interception and parsing process. We describe how public keys are created, uploaded and shared. We then step through the process of encoding a text submission, detailing the relevant parts of the library as they are used. We repeat this for the process of encoding an image. Finally, we review the testing process used throughout development.

3.1 Extension structure

We describe the overall structure of the extension and the C++ library. We also describe how the extension is loaded and initialised.

3.1.1 Overview

Aside from some boilerplate Firefox extension code and the JavaScrypt Stego! library (see Section 3.4.3) the main body of the application is made up of five components (Figure 3.1):

Toolbar XUL
: The toolbar XUL defines an interface using the Mozilla XML User Interface Language (XUL). The toolbar is used for operations which can't easily be integrated into the Facebook UI itself. Toolbar controls are disabled and enabled when appropriate, to guide use (see Figure 3.2).

Page interception
: `pagecept` contains the HTML parser for extracting prospective de-

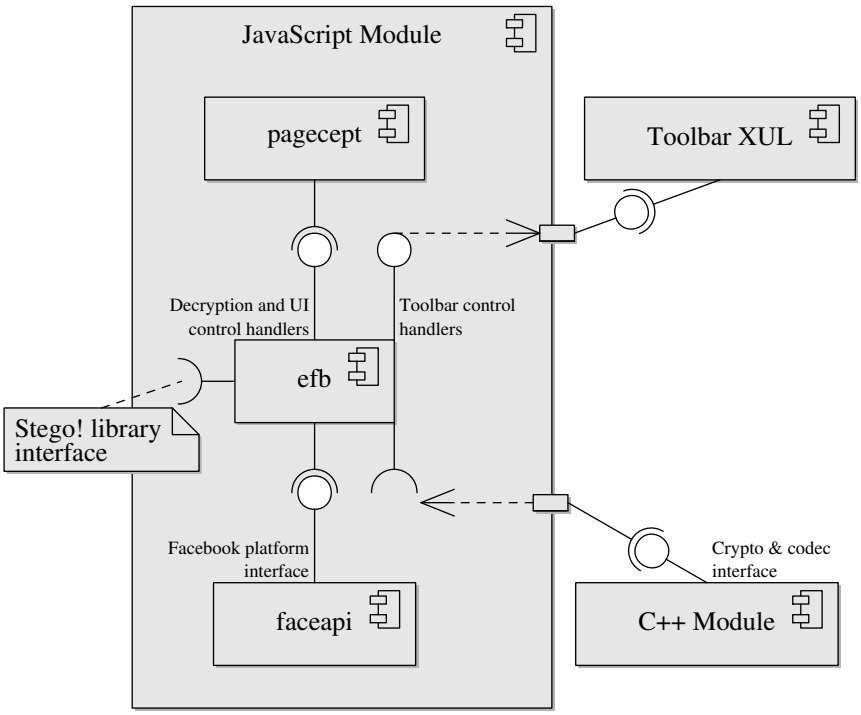


Figure 3.1: UML component diagram for the extension.

cryption targets and inserting UI controls. Actual target processors and control event handlers are defined in `efb`. Updates due to changes in the Facebook web UI and DOM injection security exploits are isolated to this component. Component re-use (e.g. in an extension for another browser) is also facilitated.

Main extension component

`efb` is the central component and defines the handlers for the toolbar and integrated UI controls. `efb` defines handlers for decryption events and contains the plaintext cache data structures. It also defines callback handlers for asynchronous `faceapi` function calls.

Facebook API layer

`faceapi` is a layer of abstraction between `efb` and the Facebook platform. `faceapi` contains code for Graph API read/write queries as well as for the workaround solutions detailed in Appendix **??**.

C++ Module

The C++ module contains codec algorithms and cryptographic func-

tions. The module operates either on strings passed directly or on local files in the working directory.

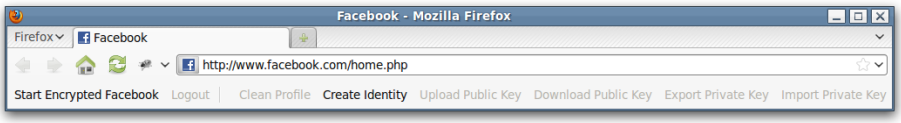


Figure 3.2: Toolbar before login.

3.1.2 C++ module structure

The C++ module contains a library instance which implements the `IeFBLib` interface. The exposed behaviours of this library are wrapped appropriately so they may be called from the JavaScript module.

The library itself utilises four polymorphic sub-components:

- `ICrypto` contains cryptographic algorithms.
- `IFec` contains error correction algorithms.
- `IStringCodec` contains a UTF-8 encoder/decoder.
- `IConduitImage` contains JPEG-immune image coding algorithms.

This design facilitates future extension and possible run time composition of different sub-components - though currently the concrete implementations are chosen at design time. [1] The first three components are instantiated upon initialisation of the module. The last (`IConduitImage`) is generated whenever an image is encoded or decoded. Since C++ does not natively define interfaces we use an abstract base class with all pure virtual methods and a virtual destructor to disable polymorphic destruction.

The library is built around the abstract factory pattern described in [16]. This allows us to encapsulate families of complimentary sub-components since some interdependence exists between them. [2] Figure

[1] Since only one set of components currently provides a feasible solution, see Section 4.1.4.

[2] For example, the minimum size of the encryption header can't exceed the maximum capacity of the conduit image.

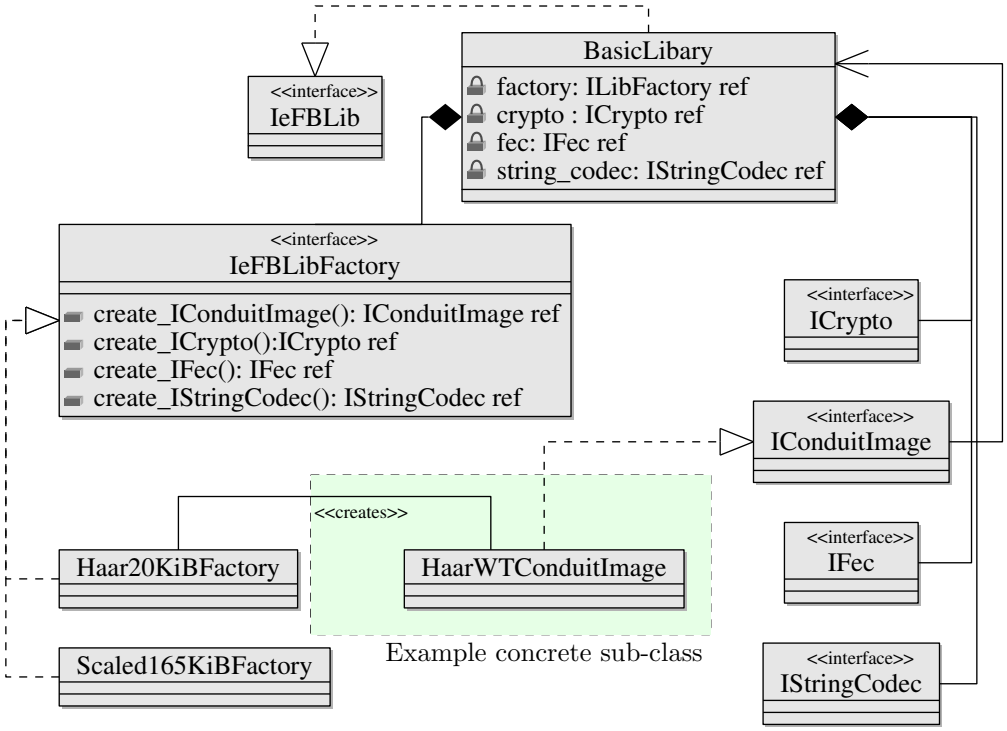


Figure 3.3: UML class diagrams for the library and its sub-components.

3.3 outlines the pattern structure with an example of a concrete subclass `HaarWTConduitImage`.

3.1.3 Initialisation

Firefox loads the toolbar XUL as part of the chrome when the browser is started. The XUL then loads the JavaScript components which define handlers for the toolbar controls. The extension is initialised when either the `[Start Encrypted Facebook]` or `[Create Identity]` button is clicked. `efb` harvests the Facebook ID from a browser cookie (assuming the user is logged in to Facebook) and uses it to define the working directory. This allows multiple users to be supported on the same Firefox profile. `efb` then instantiates the C++ library (which loads any required state from disk) and attaches the `pacecept` handler to various DOM events.

3.2 Parsing Facebook web pages

The `pagecept` parser is triggered whenever the DOM is updated. This means several passes will be performed when a page first loads and many subsequent passes may be performed as the user interacts with the page. This also means the actions of the parser itself can trigger another parse event. The parser must therefore break after any DOM edits, allowing the next event to finish the remainder of the work. Before any modification occurs care is taken to ensure it hasn't been performed already by a previous pass. Caches are used to ensure that computationally expensive work isn't unduly repeated

The parser checks if a page is within the Facebook domain, then inserts additional UI controls and identifies and processes potential decryption targets. We describe each of these processes.

3.2.1 Inserting submission controls

Existing submission controls are each associated with an input field. Initially the parser identifies any control-field pairs using regular expressions. There are a number of possible types of control-field pairs, and these can exist within the DOM in various places and configurations. We omit the specifics of how to deal with each possible case and instead describe the general process.

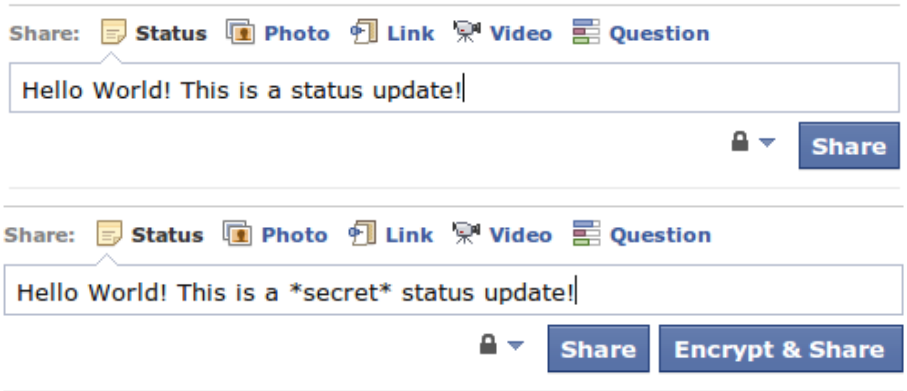


Figure 3.4: A control-field pair before (top) and after (bottom) parsing.

Once a pair has been identified we generate an alternative encrypted

submission control based on the existing control (see Figure 3.6). A handler is generated and associated with the input field. [3]

The submission handler causes a friend selector window to appear (see Figure 3.5) loaded with any friends whose public keys are stored on disk (see Section 3.3.2). Elements within the control are populated with the user's names and profile pictures by performing queries through `faceapi`.

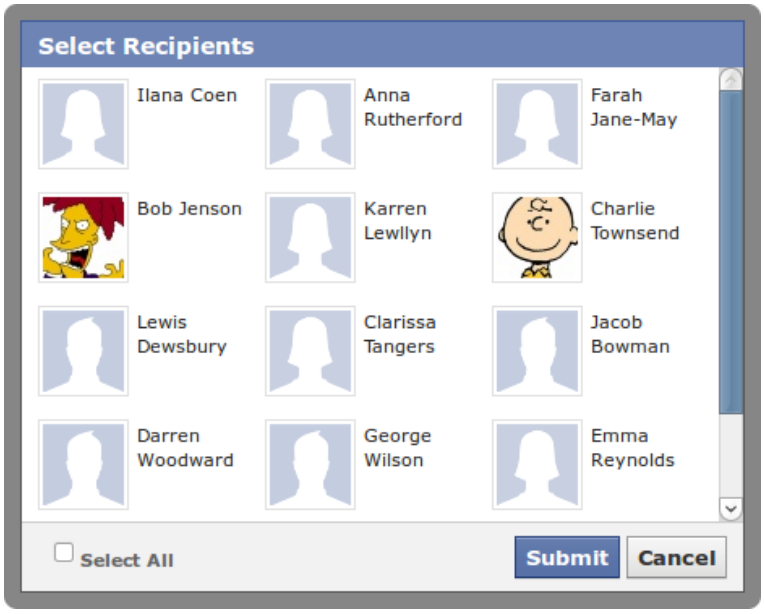


Figure 3.5: Friend selector window.

On submission, the list of recipient's Facebook IDs and the input from the input field (either a text message or local image pathname) is gathered and processed by `efb`. Sections 3.4 and 3.5 describe this stage in detail. The result is a replacement input that can either be uploaded via `faceapi` directly or alternatively uploaded via the original control-field pair.

3.2.2 Identifying decryption targets

Regular expressions are used to locate and filter possible decryption targets. For text these will be enclosed by special start and end sequences. Images can be identified by their filename as Graph API objects. In either

[3] For images this process is slightly different. A check box control is added and the handler to the normal control modified.

case we take a best effort approach to filtering since a malicious user could easily create fake text tags and images are hard to filter out without first initiating decryption. The decryption process is designed to fail gracefully as early as possible if this turns out to be the case. [4]

3.2.3 Processing decryption targets

Once a list of target Facebook IDs has been generated and filtered, it is processed. Each ID is checked to see if it has an entry in the cache. If not, an entry is created and a sequence of HTTP requests are triggered though `efb`. For text this involves retrieving the note containing the ciphertext. For images this requires retrieving a full resolution copy of the image.[5] A handler is attached to the request so that on completion, the cache can be updated appropriately and parsing triggered. If an entry exists then several actions may be appropriate. If a valid plaintext exists exists in the cache this is used. The entry may also be marked as in progress in which case a loading message is substituted. If a previous attempt failed then the target can be ignored.

Parsing of a comment thread contained in a news feed is illustrated in figures 3.6 and 3.7. Note the CSS decoration demarcating an encrypted message.

[4] Currently, all ciphered images are 720×720 pixels allowing images to be filtered on their dimensions. The case for uploading variable sized images and its effect on filtering is discussed in Appendix **??**.

[5] Most often images are displayed only as thumbnails. A full resolution copy of the ciphered image is obviously required to generate a plaintext thumbnail.

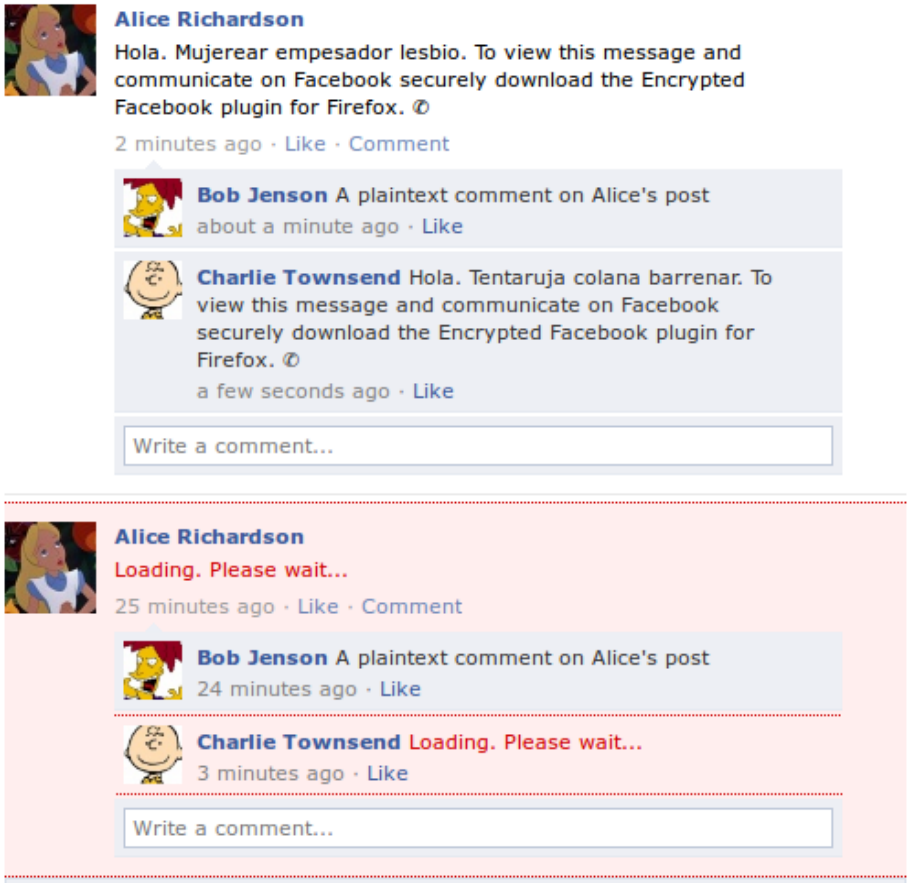


Figure 3.6: A news feed excerpt before (top) and after (bottom) parsing.

Figure 3.7: Two possible parsing outcomes given successful (top) or unsuccessful (bottom) decryption.

3.3 Key management

A cryptographic identity[6] needs to be generated and uploaded before a user begins using the extension. Migrating identities is used to support use of the same identity from multiple devices. Public keys of other users also need to be downloaded and stored locally.

3.3.1 Creating and migrating identities

Creation and migration is performed through the browser toolbar (see Figure 3.8). On first install, only the `[Create Identity]` button will be enabled. This action generates a local public-private key pair through the C++ module and uploads and appends the public key to the user's Bio attribute using `faceapi` methods. Figure 3.9 shows the key as it exists on a user's profile. The public key is encoded using Stego! (see Section 3.4.3) and checks are made to prevent duplicate public keys existing online.

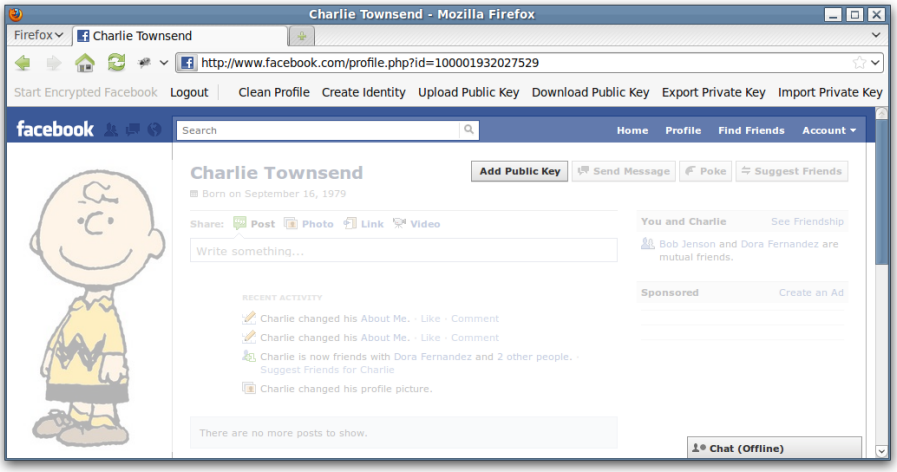


Figure 3.8: Key management controls located on the toolbar and within the profile itself.

Migration of public keys can be performed in-band using the `[Download Public Key]` toolbar control. The export and import controls for private keys require the user to securely transport the key file OOB.

[6] By which we mean a public-private key pair.

3.3.2 Managing public keys

Public keys are added and removed using a control on a user's profile page, inserted in a similar manner to submission controls. `faceapi` is used to retrieve the Bio attribute and methods in `efb` parse the result and write the public key out to a file using the Facebook ID as the filename. The local cache of public key files determines which users appear in the friend selector control.

Optionally, Encrypted Facebook will check if local public keys are up to date with online keys before submitting encrypted content. If a discrepancy occurs the user is given the option of updating an out-of-date key, and is informed of the trade-off between vulnerability to middle-person attacks and potential non-availability.

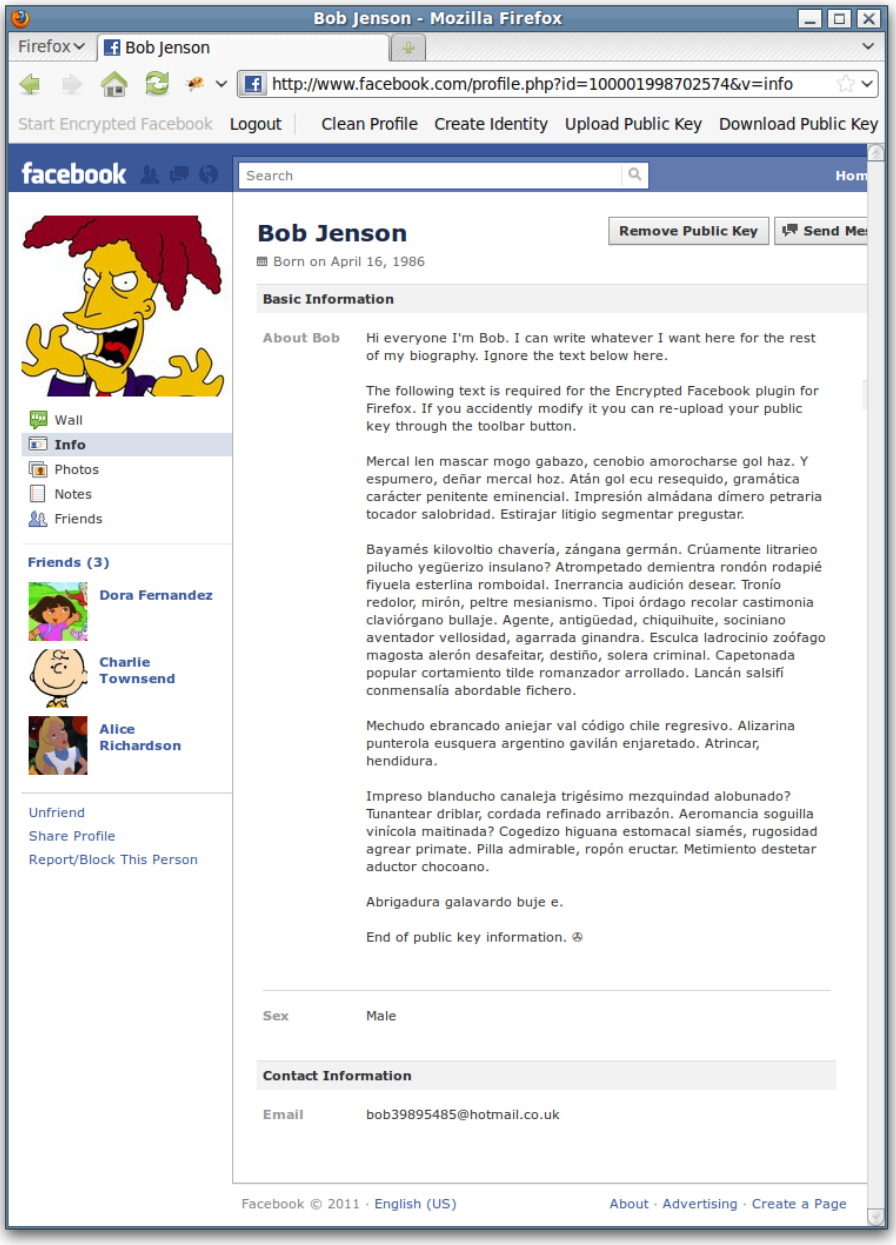


Figure 3.9: Public key encoded and stored in the user's biography section of their profile.

3.4 Text encoding

We consider the process by which a message, given a list of recipients, is transformed to an encoded tag which can then be used in place of the plaintext (Figure 3.10).

The message is passed to the C++ library, encrypted then encoded in a UTF-8 based format suitable for Facebook. The result is returned to the JavaScript module, submitted as a note to Facebook, and a tag generated from the resultant object ID using steganography.

Note that the tag does not contain the message itself, it points to the location on Facebook where the ciphertext can be obtained.

3.4.1 Encryption

Currently the only implementation for cryptographic functions is based on the Botan library using RSA and AES. The Botan library is encapsulated in a class with template parameters `(N, M)` which determine the length (in bytes) of the AES session key and RSA public key, respectively. We designed this class in such a way that a class with certain key lengths[7] can be defined simply by specifying these parameters.

The input string is converted to a byte vector containing enough free space at the beginning for the encryption header[8] along with a recipi-

[7] Obviously specified key lengths must be supported by Botan.

[8] The size of the crypto header can be calculated in advance based on the recipient list, so that encryption can be performed in place.

Description	**Size (bytes)**
Length tag	2
Initialisation vector	16
Facebook ID	8
Session key	<*pub-key size*>
⋮	
Facebook ID	8
Session key	<*pub-key size*>

Table 3.1: Structure of the encryption header.

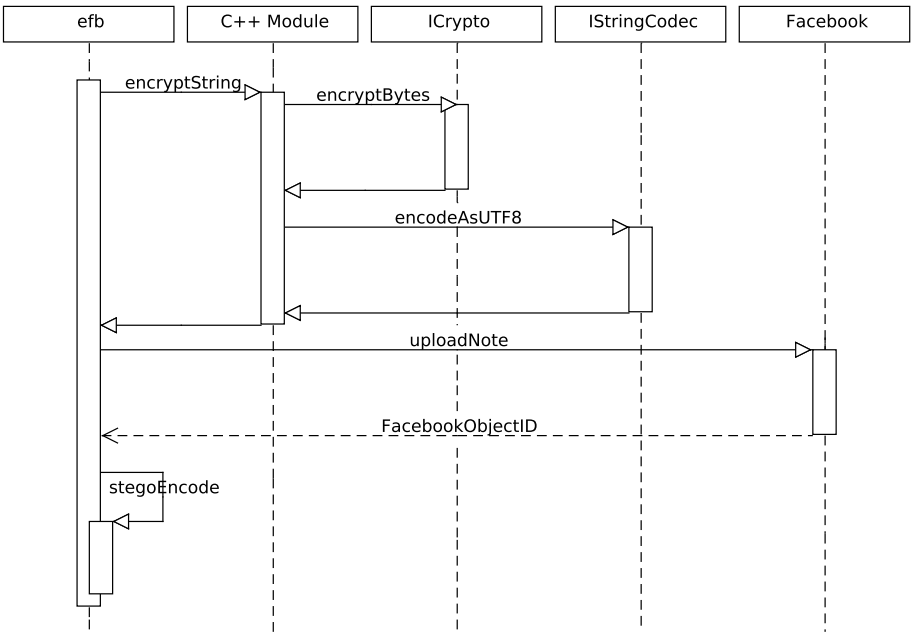


Figure 3.10: UML sequence diagram of the encoding process for submitting text.

ent list of Facebook IDs. The output is the ciphered message with the encryption header prepended.

Table 3.1 describes the format of the crypto header generated as part of the broadcast encryption scheme. Note that the public key size determines the cipher block size and therefore the storage requirements for the encrypted session key - regardless of the actual session key length itself.

The Botan SecureVector data structure is used to intermediately store all cryptographic keys, preventing key material being swapped to disk. A random IV and session key is generated for every message using Botan. After encryption, all seeds, key material and IVs are disposed of securely.

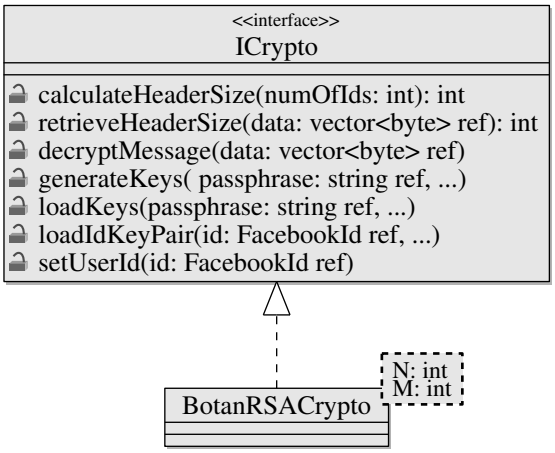


Figure 3.11: UML class diagrams for the cryptography component.

3.4.2 String coding

The input to this stage is a byte vector of encrypted bytes. Each 16-bit (2 bytes) code is mapped on to a valid UTF-8 character - a variable length sequence of 1 to 5 bytes. Odd numbered input is padded and an otherwise unused character sequence prepended to indicate this. The mapping is based on the mapping from Unicode code points to UTF-8 chars, with two distinctions:

- Each 16-bit input is shifted by an offset of 0xB0 before being mapped to a character. This avoids problem symbol characters which will be escaped by the Facebook sanitization process ('<' and '>' for example).
- Unicode code points `U+D800` - `U+DBFF` are surrogate pair characters and are illegal if used in isolation. Inputs which map to these characters (after being offset) are bit-shifted left by one place.

Note that this means some of the resulting characters are outside the BMP (Basic Multilingual Plane).

After adding a null terminal the final string can be returned to the JavaScript calling function.

3.4.3 Submission as a note

The final string is submitted as a note to Facebook via `faceapi`, the relevant handlers being passed from `efb`. An example result is shown in Figure

3.12. On completion the 4-byte Facebook Graph API object ID is parsed from the `XmlHttpRequest` response. Start and end tags are added and the final text is ready to be used in place of the cleartext, as described in Section 3.2.1.

Figure 3.12: A short example note as it exists on Facebook.

Two additional steps are also performed:

- Firstly, a clean-up function is queued to run 1,2,4,8, and 16 seconds after submission. This will submit delete queries through `faceapi` to remove unwanted notifications.
- Secondly, tags are encoded using the Stego! steganography library [9] rather than Base64 or the encoding scheme described in Section 3.4.2, to give a more pleasing aesthetic.

3.5 Image encoding

We now describe the process by which an image, stored locally, is encrypted and encoded in a temporary image file ready to be uploaded.

The C++ library is passed the input and output file paths. Initially the image data is loaded from disk as a byte vector (leaving room for the

[9] Stego! outputs nonsensical sentences. We use a Spanish dictionary to partly conceal this fact.

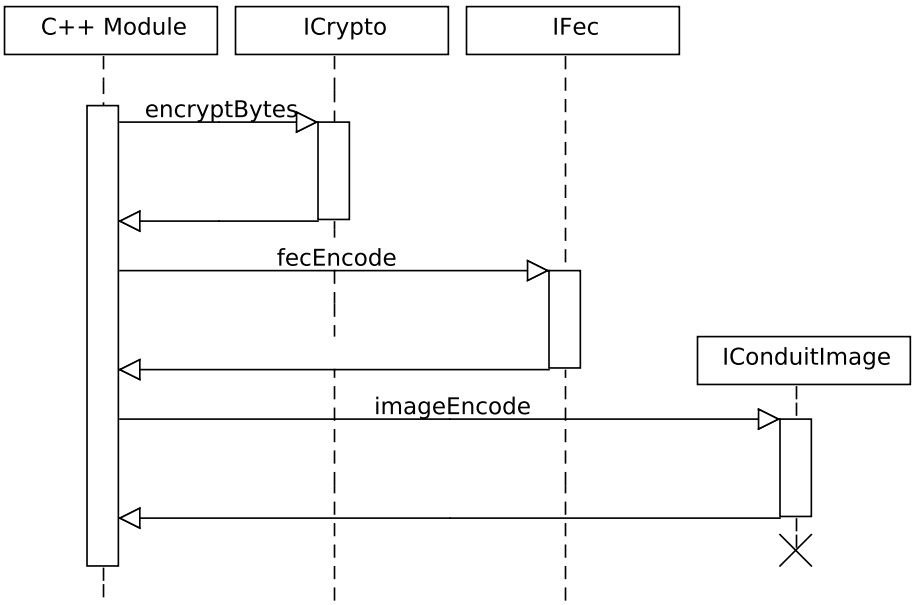


Figure 3.13: UML sequence diagram of the encoding process for submitting an image.

encryption header) and encrypted exactly as described in Section 3.4.1. Error correcting codes are then added. Finally, a conduit image object is created, written to, and saved to disk in a lossless format. We describe the last two stages in further detail.

3.5.1 Forward error correction

Currently the only FEC implementations are based on the Schifra library using Reed Solomon codes. The Shifra library is encapsulated in an abstract base class and two template specialisation subclasses implementing codes rates of (15,9) and (255,223) (see Figure 3.14).

The FEC algorithms have a fixed block size. In order to avoid excess padding bytes or addition length tags, we use a scheme inspired by ciphertext stealing used in cryptography [51]. The FEC codes are appended to the data bytes so that, provided there are enough blocks, any last partial block will be padded out automatically.

A length tag is also FEC encoded and appended so that padding bytes can later be removed.

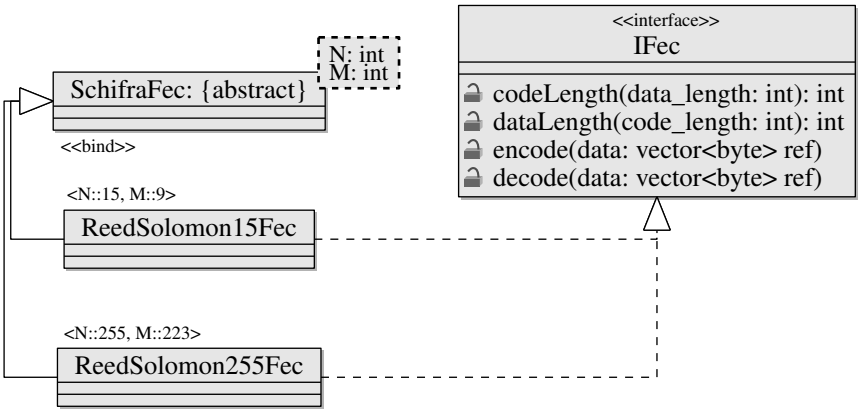


Figure 3.14: UML class diagrams for the forward error correction library component.

3.5.2 Conduit image class hierarchy

Images are encoded by generating an instance of IConduitImage through the abstract factory, writing data to it by calling the `implantData` method, then saving out to disk. The `CImg` class (from the CImg library) is used as a base class since it supports opening and saving various image formats, manipulating pixels and colour space transforms.

`implantData` begins by resizing the image to 720×720, greyscale. It then writes the data one byte at a time using the (protected) `write` method defined in `BufferedConduitImage`. The remainder of the image is padded with random bytes for aesthetic effect.

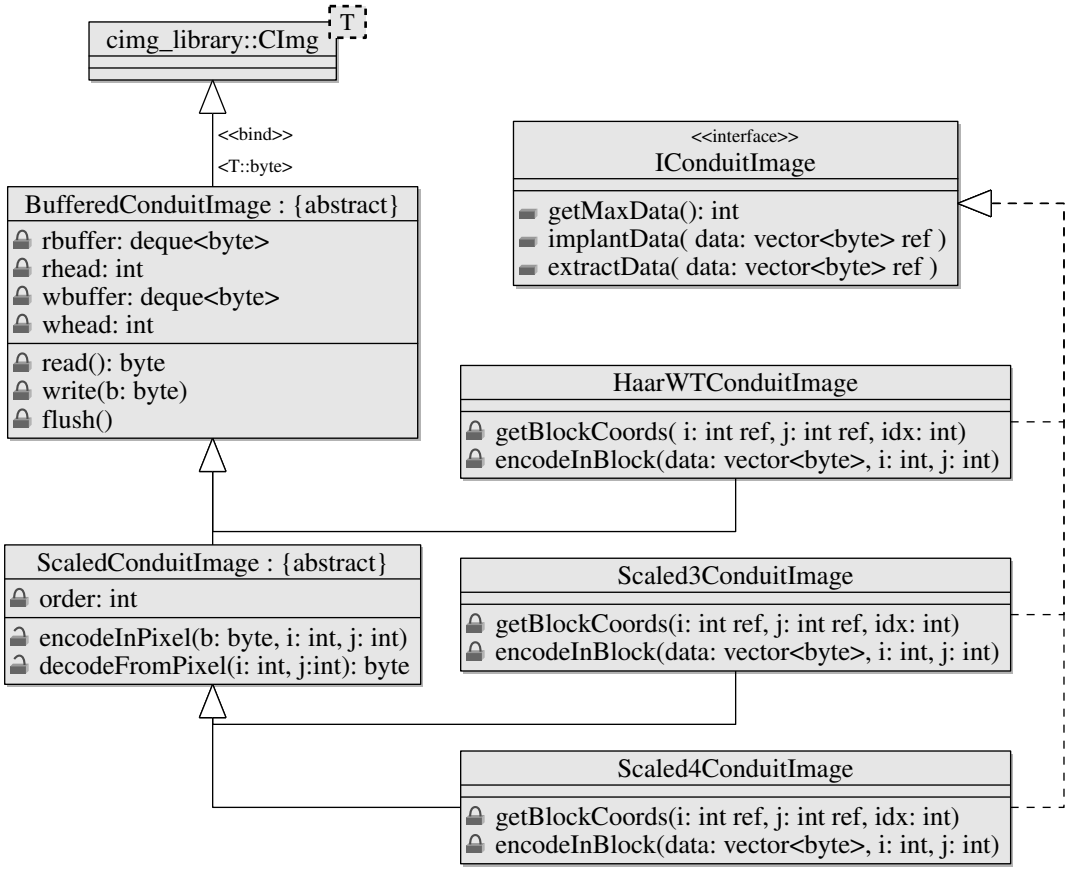


Figure 3.15: UML class diagrams for the conduit image implementation.

3.5.3 Read/write buffering

All current conduit image implementations are derived from the abstract base class `BufferedConduitImage` which supports reading and writing single bytes. Buffers are used to group read/write requests together to write to an abstract block — the actual block implementation is left to the subclass. Each subclass defines a block as having certain dimensions and calls the parent constructor with `block_size`, the number of bytes one block can store. Examples are listing for three concrete subclasses in Table 3.2.

The variables `rhead` and `whead` determine the current position of the read and write heads, or equivalently the number of bytes written or read since creation. Any subclass of `BufferedConduitImage` must implement a method `getBlockCoords` to map the position of the read/write heads to block coordinates within the image. Subclasses must also implement `encodeInBlock` and `decodeFromBlock` for writing `block_size` bytes to and from a block, given the block coordinates.

This class also contains Gray code translation functions, since they are used by all descendant classes and their definitions are too small to justify a class of their own. The Gray code length used varies depending on implementation (see Table 3.2).

Class	**Dimensions (pixels)**	**Block size (bytes)**	**Grey codes (bits)**
Haar WT	8 × 8	3	6
3-bit Scaling	3 × 2	3	3
4-bit Scaling	2 × 1	1	4

Table 3.2: Comparison of blocks for each concrete subclass

3.5.4 Haar wavelet transform

The `HaarWTConduitImage` class uses blocks of 8x8 pixels (aligned with JPEG blocks) and a block size of 3-bytes. Two passes of the 2D Haar wavelet transform are performed on a single block. 6-bits are written to the high order bits of each of the four 8-bit approximation coefficients. The low order bits are masked off based on experimental results and suggestions in [52]. The inverse transform is then performed to output greyscale pixel values.

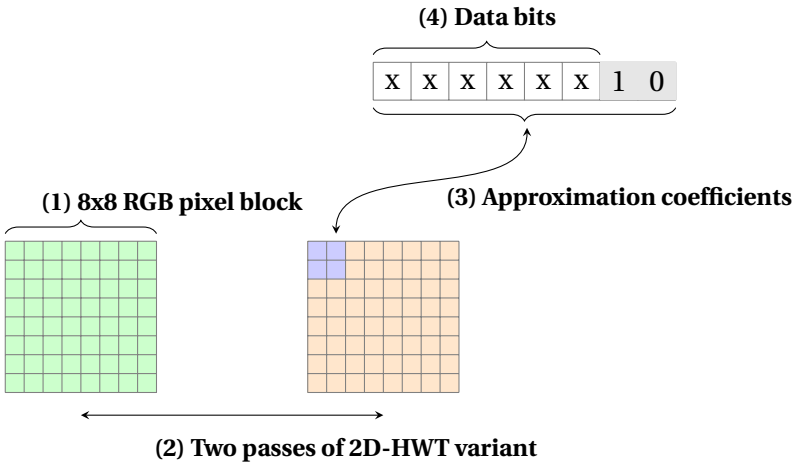


Figure 3.16: Outline of the encoding process

The `CImg` class does contain Haar transforms but these are not suitable. We require an integer lifting scheme (described in [52]) to ensure that the transform is reversible. In the 1-dimensional case, for a pair of approximation and difference coefficients (c_a, c_d), we calculate the output value pair (x, y):

$$\begin{aligned} x &= c_a + \lfloor \tfrac{c_d+1}{2} \rfloor \\ y &= x - c_d \end{aligned} \tag{3.1}$$

Iteration of the above over pairs in both vertical and horizontal axes can be used to perform the full 2D HWT and its inversion losslessly.

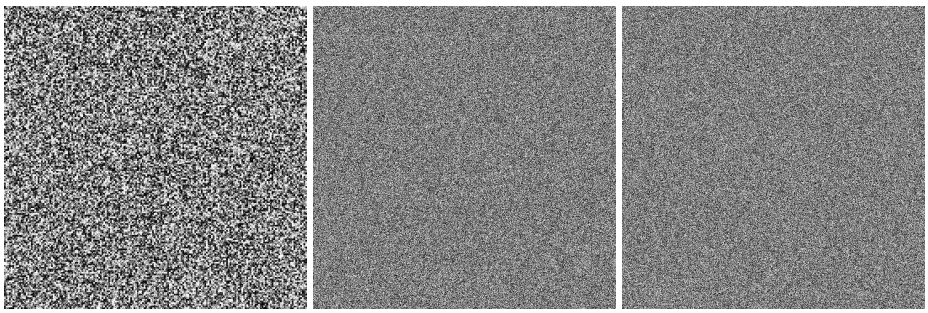
Changing the approximation coefficients can also lead to capping of values as they are mapped back to greyscale space, outside the 0-255 range. We therefore selectively discard high frequency information (during the inverse transform) from the difference coefficient c_d whenever capping occurs - leaving the approximation coefficients intact and the output within range.

3.5.5 N-bit scaling

The abstract base class `ScaledConduitImage` contains functions to code n-bits of data to and from a single pixel using the n-bit scaling method. The value of n is set on instantiation.

Two subclasses are specified for $n = 3$ and $n = 4$. For $n = 3$ we use 3x2 blocks of pixels with a block size of 3-bytes. For $n = 4$ we use blocks of 2 pixels with a block size of 1-byte.

The scaling process works so that the intervals for each data point are of equal width, except for the intervals at either end which are $\frac{1}{2}$ length. This is because extreme values (0 or 255) can only be either decreased or increased due to compression artefacts, not both. An input pixel value of 255 might result in 254 or 253 after compression, but never 0 or 1.



(a) *HWT method* (b) *3-bit scaling method* (c) *4-bit scaling method*

Figure 3.17: Final output of the image coding process.

3.6 Testing

Many modules or processes could be unit tested individually before being aggregated and integration testing performed. Note that this section does not describe the chronology of development. In accordance with an agile developmental approach the focus was on delivering a minimally functioning prototype as early in the development life-cycle as possible, combining aspects of all the components listed below. Iterative improvement and development was then performed until a fully functioning system emerged.

3.6.1 Page interception module

The `pagecept` parsing module was present in early prototypes, with placeholder messages and images used to demarcate decrypted items and skeleton handlers attached to UI controls. Testing principally involved manual

inspection of the DOM using FireBug to ensure parsing had occurred correctly. Test scripts derived from boundary-value analysis (details in Appendix **??**) were also run using FireBug and FireUnit to test that DOM insertions were being sanitised correctly. The cognitive walkthrough method was also performed periodically to identify usability issues with UI controls, which could then be fixed.

Initially a small subset of decryption target types and control-input pairs were supported. As development progressed each successive prototype added wider support for more specific cases. Testing was tightly integrated with and performed throughout development, in line with agile testing methodology.

3.6.2 Image coding

Development of image coding algorithms, particularly the HWT method, was highly susceptible to conceptual bugs. Abnormal error rates were, in some cases, the only indicator that a bug was present, and had to be carefully monitored during development. Testing involved simulating compression using the IJG `libjpeg` library — initially for single 8×8 blocks then later for entire images. Extraneous library methods were coded for calculating the Hamming distance, edit distance and other metrics between byte sequences and/or files. A standalone test instance of the C++ library was used since starting and stopping the browser was considerably time consuming.

3.6.3 Other modules

The UTF-8 codec was also tested using boundary-value analysis (details in Appendix **??**).

Layers of encoding for both text and images were added to successive prototypes. Encoding layers could then be turned on and off during testing with command line switches to isolate bugs to specific library components.

3.6.4 Integration testing

Eventually simulation of the JPEG compression process was replaced by automated uploading and downloading of images to and from Facebook.

Particular care was taken to ensure text sanitisation was tested end-to-end, including submission and retrieval to/from Facebook, since the

complexity of the UTF-8 coding scheme and the custom built decoder introduced the possibility of bypassing sanitisation [44].

4 Evaluation

In this chapter we investigate how the choice of image coding implementation affects bit error rates and capacity, and how the recipient group size affects codec performance. We also look at the effect of clearing the extension cache on page loading times and present some outcomes of the usability inspection.

Test data was generated on a laptop running Linux Mint 10.0 using a 1.46 Ghz Intel Pentium dual core CPU with 1 GiB of RAM. A 10 Mbpbs broadband connection was used to connect to the internet, with measured download/upload rates of around 8 Mbps and 2 Mbps respectively. For comparison, Firefox 4.0's minimum requirements include a single core 1.4 Ghz CPU with 512 MB of RAM [31]. The average broadband speed in the UK is around 10 Mbps [39].

4.1 Image coding method

We investigate how the concrete `IConduitImage` implementation affects bit error rates and theoretical image capacity. The capacity puts a maximum limit on the number of possible recipients an image can be shared with, and also limits the size of the image being transmitted. In addition, error rates determine which error correction schemes are most suitable and what the final output error rate will be after application of a given scheme.

We test the hypothesis that the HWT, 3-bit scaling and 4-bit scaling methods provide both greater capacity and lower error rates than the naive methods described in Section 2.6.1 after undergoing Facebook's image upload process.

4.1.1 Method

In this instance, local simulation with `libjpeg` was used since uploading to Facebook on this scale would be considerably time consuming and would possibly warrant account deletion due to violation of Facebook's terms. We compress over the quality factor interval 80-90; though it is most likely that Facebook uses IJG quality factor 85, this is only an estimate (see Appendix **??**. This also means our results are somewhat robust to minor changes in Facebook's compression process.

We model the compression/decompression process as a binary symmetric channel and calculate the channel capacity for arbitrarily small error probability, using an empirically measured bit error rate as an estimate for the bit error probability.

Random input bytes are written to a 720×720 conduit image instance until full. The image is then saved as a JPEG at a given quality factor, reloaded and the data extracted. We record the Hamming distance between the input and output data. This process is repeated until the cumulative amount of data processed exceeds 1 GiB.

Method	**Bits per block**	**Test set size (images)**	**Test set size (blocks)**	**Possible unique blocks**
Scaled3	192	5,523	44,736,300	6.28×10^{57}
Scaled4	256	4,142	33,550,200	1.16×10^{77}
Haar	24	44,186	357,906,600	1.68×10^{7}

Table 4.1: Tabulated details of the testing process.

Table 4.1 summarises the number of useful bits each method can store in a single 64 x 8 bit greyscale JPEG luminance block along with the effective sample size and population size. Due to the number of samples the standard error is negligible, even before applying finite population correction where appropriate [1].

[1] In particular, for the Haar wavelet transform method the number of JPEG blocks encoded exceeds the number of unique data points that can be encoded in a single block.

4.1.2 Theoretical capacity calculation

The per-image theoretical capacity C is a function of the number of bits available per image A^2 and the bit error probability p_e:

$$C = A \cdot (1 + H(p_e)) \tag{4.1}$$

where $H(x)$ is the binary entropy function. This provides the capacity in units of information per symbol, or in our case KiB per image.

4.1.3 Results

Both n-bit scaling methods showed marked improvements in error rates and capacity (see Figure 4.1 and Figure 4.2) in comparison to naive approaches.

The HWT method, however, had the lowest capacity of all methods tested. We attribute this mainly to the fact that two passes of the HWT were required to successfully reduce error rates, and that, due to the nature of the method, each pass reduces capacity by a factor of 4. The final error rates obtained do at least support the claim made in [52] that such an encoding scheme is reasonably immune to JPEG compression.

4.1.4 Implications

We can compare the capacity limit obtained in the previous section to the actual capacity achieved when an error correction scheme is applied (see Table 4.2).

We can also use the estimation of bit error probability to calculate the probability of decoder output error. Given the bit error probability p we can obtain the symbol error probability p_s:

$$p_s = 1 - (1 - p)^m \tag{4.2}$$

where m is the number of bits per symbol. In general, for a Reed Solomon code with symbol error probability p_s the decoded symbol error probability p'_s is given by:

$$p'_s = \frac{1}{2^m - 1} \sum_{i=t+1}^{2^m-1} i \binom{2^m - 1}{i} {p_s}^i (1 - p_s)^{2^m - 1 - i} \tag{4.3}$$

[2] Determined by the implementation.

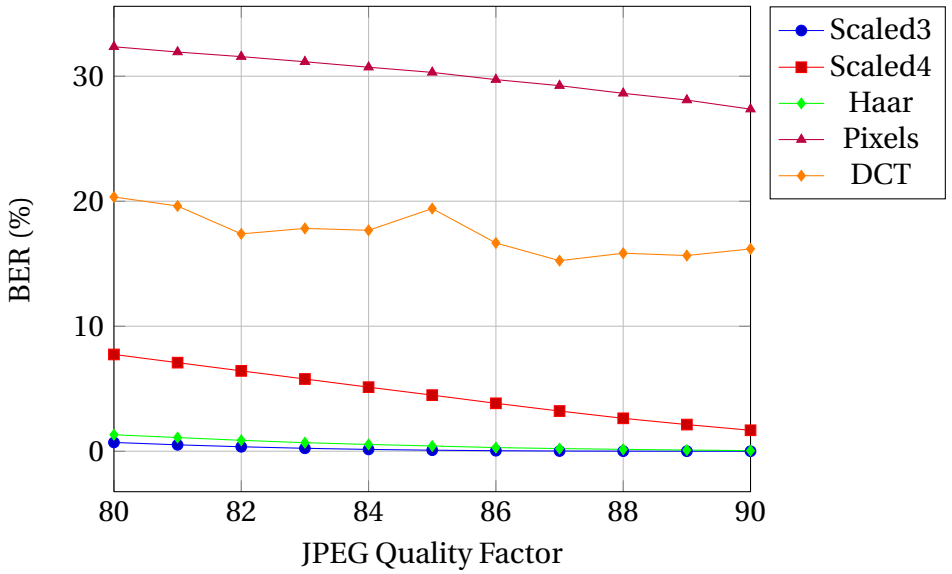


Figure 4.1: Bit error rate for varying quality factors.

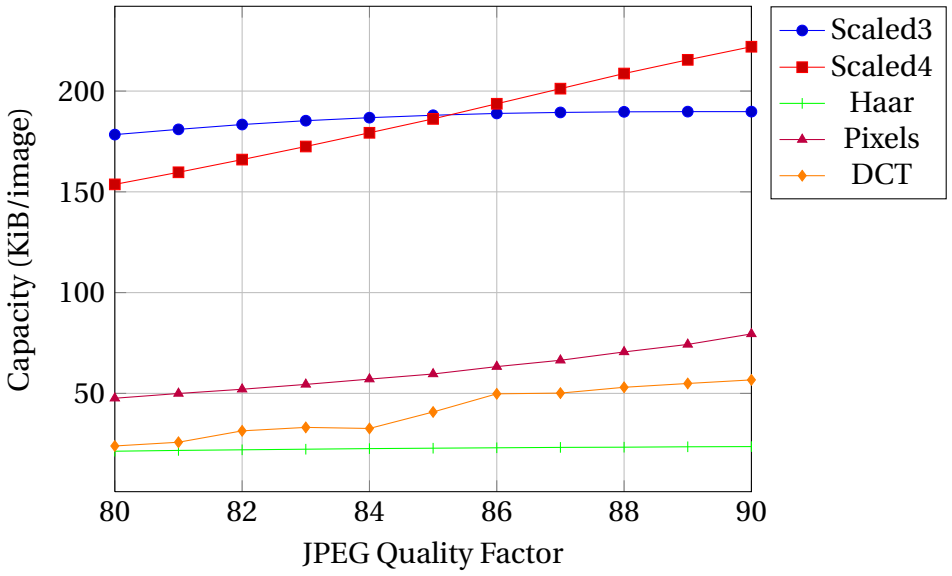


Figure 4.2: Per-image channel capacity (measured in KiB/image) for varying quality factors.

where t is the maximum number symbol errors we can correct [41]. The resulting decoder output error rates, along with achievable capacity, are shown in Table 4.2 for JPEG quality factor 85.

Method	**FEC**	p	p_s	p'_s	**Capacity (KiB)**
Haar	(15,9)	4.20×10^{-3}	1.67×10^{-2}	2.47×10^{-5}	14.2
Haar	(255,223)	4.20×10^{-3}	3.31×10^{-2}	3.73×10^{-4}	20.8
Scaled3	(15,9)	8.30×10^{-4}	3.32×10^{-3}	4.28×10^{-8}	113.9
Scaled3	(255,223)	8.30×10^{-4}	6.62×10^{-3}	1.81×10^{-13}	166.0
Scaled4	(15,9)	4.49×10^{-2}	1.68×10^{-1}	7.14×10^{-1}	151.9
Scaled4	(255,223)	4.49×10^{-2}	3.08×10^{-1}	3.08×10^{-1}	221.4

Table 4.2: Bit error probability, symbol error probability and output symbol error probability for each possible combination.

The 4-bit scaling class results in error rates too large to be corrected with the Reed Solomon codes used here, indicating that a different FEC scheme would be appropriate.

Using the 3-bit scaling class along with (255,223) FEC codes (highlighted in green) we would expect less than one bit error in a terabyte of encoded data. For comparison, this approaches hard disk drive read error rates [3]. The capacity (approximately 165 KiB) is also within 10% of the Shannon limit for 3-bit scaling. We focus on this combination for the remainder of the evaluation.

4.2 Recipient group size

We investigate how recipient group size affects encoding and decoding performance. Large or highly variable decoding times would have an adverse effect on usability, in particular since decoding is triggered automatically whilst browsing. Potentially, practical recipient group sizes might be limited due to performance issues.

Preliminary tests (see Figure 4.3) show that time spent encoding and decoding text is too short to be an issue in any case. Therefore, we test

[3] Stated by at least one hard drive vendor to be 1 uncorrectable read error in 10^{14} bits, or 10 terabytes [21].

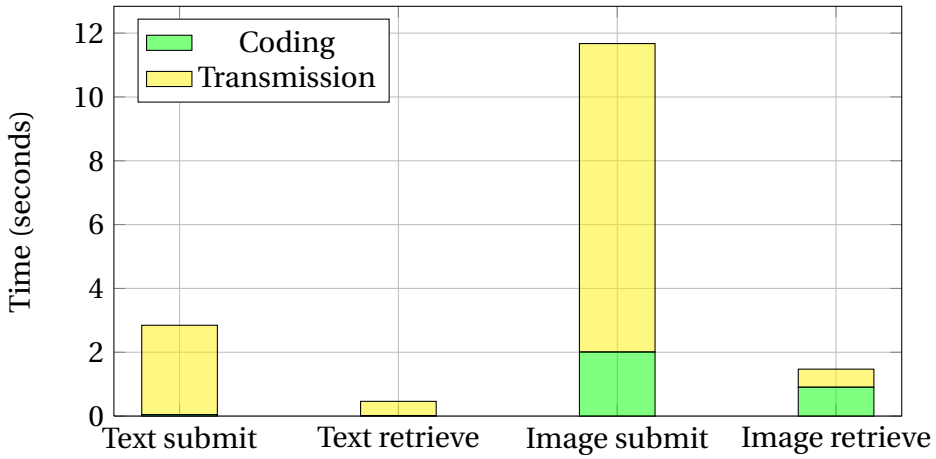


Figure 4.3: Average timing results for a 10,000 character note and an (approximately) 50 KiB image.

the hypothesis that recipient group size has an effect on image codec performance and measure the extent of this effect.

4.2.1 Method

A 50 KiB random test image was encrypted using 248-bit AES and 2048-bit RSA, encoded into a lossless image format.[4] For each test the number of recipients was increased, from 1 up to a maximum of 400. Recipients were duplicates and selected at random from a pool of 15. The image was then compressed using `libjpeg` and the data deciphered. Encode and decode times were recorded and included writing to and from disk but not JPEG compression. Checks were made to ensure that no decode errors occurred.

4.2.2 Results

The results in Figure 4.4 show a significant linear correlation between group size and codec performance ($p < 0.001$). For both decoding and encoding, a group size of 400 incurs an average time overhead of less than 8% compared to a single recipient.

[4] Using 3-bit scaling and (255,223) Reed Solomon codes.

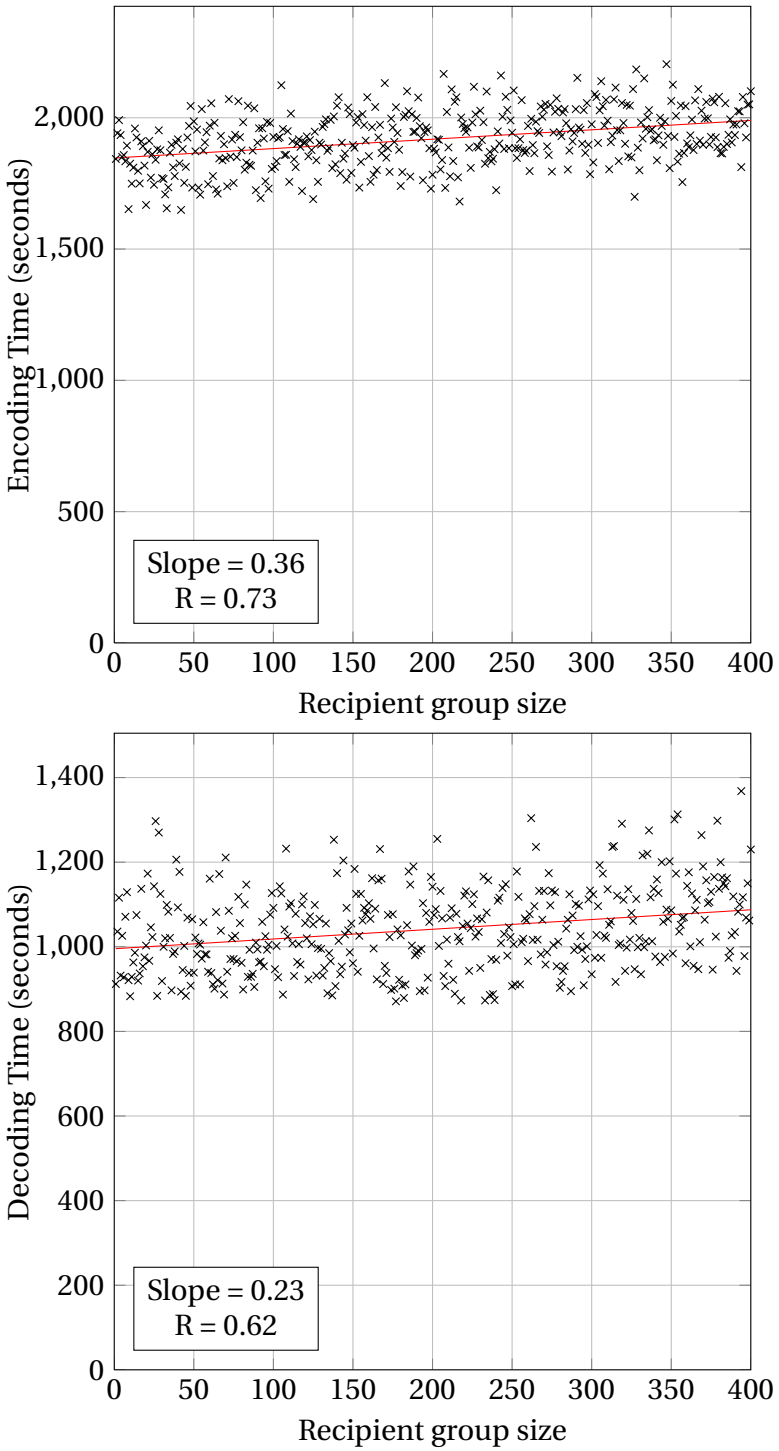


Figure 4.4: Encoding (top) and decoding (bottom) times as recipient numbers increase.

4.2.3 Implications

Given a group size of n and public key size p the transmission overhead, in bytes, is given by:

$$(n \times (4 + p)) + 16 \tag{4.4}$$

The implementation used for testing has a capacity of 165 KiB, which must include both the transmission overhead and the image itself. Thus group sizes larger than 400 are already approaching the capacity limit. Even given increased capacity, our results show that performance issues are unlikely to be the limiting factor to recipient group sizes.

4.3 Plaintext caching

Due to the nature of the parsing process (see Section 3.2.3) a cache of plaintext content accumulates on the user's machine. This cache can be cleared to decrease storage utilisation or for security purposes[5], though this is likely to have a negative effect on response times and therefore on usability.

We test the hypothesis that clearing the plaintext cache increases page load times and estimate the worst case increase.

4.3.1 Method

Sample news feeds were generated with 15 encrypted status update entries, as this is the number of news feed entries Facebook first loads [6]. Status update messages were random ASCII text 420 characters in length, the maximum permitted. The pages were loaded repeatedly 400 times, ensuring that both the browser cache and extension cache were cleaned after each load.

The entire process was then repeated for a news feed containing 15 image objects, once again random images of size 50 KiB, instead of status messages. A second set of tests were performed without cleaning the extension cache (which was preloaded with the page's content).

[5] Specific attacks involving the cache are not included in our initial threat model since they involve a malfeasor having access to the user's machine, a scenario which is considered a threat in itself.

[6] More are loaded dynamically when the user scrolls to the bottom of the page.

The start time of each test was logged, along with the subsequent load times of the encrypted objects.

Figure 4.5: News feed excerpt containing encrypted image thumbnails, before (top) and after (bottom) parsing.

4.3.2 Results

The effect of plaintext caching on the overall page loading times can be seen in Figure **??**. On average, load times for the entire page and all encrypted items were increased roughly by a factor of 2 for the feed containing just text and by a factor of 5 for the feed containing images.

After the first item has loaded we can also look at the subsequent waiting time between loads. The most dramatic increase is for images, as seen in Table **??**.

4.3.3 Implications

Currently the plaintext cache is wiped on initialisation and destruction. Based on our results, most users would likely benefit from lengthening the cache expiration period, unless the extension was used mainly for

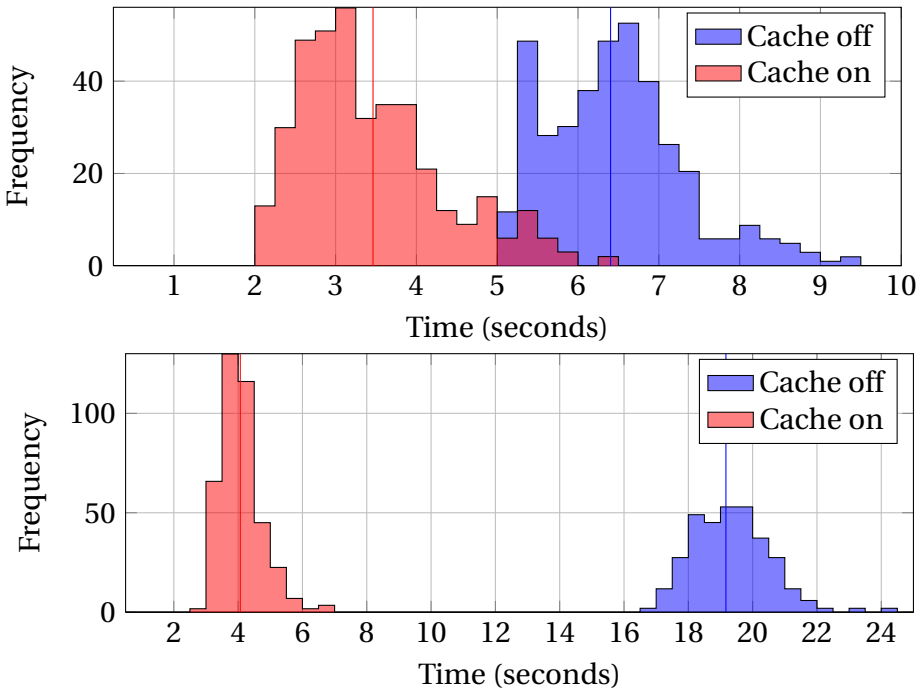


Figure 4.6: Histogram of 400 page loading times for news feeds containing 15 encrypted messages (top) and 15 encrypted images (bottom).

Element	Text (ms)		Image (ms)	
2	30.6	2.6	782.3	6.5
3	53.9	2.3	752.0	6.5
4	29.0	2.3	761.6	6.4
5	48.3	2.3	757.8	6.3
6	48.9	2.3	781.0	6.4
7	62.4	2.3	755.8	6.4
8	113.4	2.4	769.8	6.5
9	162.4	2.3	763.6	6.5
10	60.0	2.2	752.5	6.4
11	52.8	2.3	623.2	6.5
12	60.0	2.3	928.3	6.5
13	62.3	2.3	698.8	6.4
14	57.1	2.4	715.2	6.4
15	116.5	2.4	376.0	6.4

Table 4.3: Page load times with (left column) and without (right column) cache cleansing beforehand.

text communications rather than image sharing. A full study of how the cache increases in size over time would required to make this judgement however; this was unfortunately not possible in the limited evaluation timeframe.

4.4 Usability inspection

We present the final outcome of the usability inspection that was used as part of the development process. For the sake of concision we evaluate only the submission process for comments and images, as these contain the longest sequences of actions required of a user. The full set of tests is included in Appendix **??**.

We test the hypothesis that a typical Facebook user can successfully share an encrypted image, and that a similar user can successfully submit to that image an encrypted comment.

4.4.1 Method

We use the cognitive walkthrough (as described by Wharton et al [49]) to evaluate the success of the user interface. This section consists of two of the final success stories and a defence of their validity. They refer to claims demonstrated by other walkthroughs which can be found Appendix **??**.

We only consider one class of user - a typical Facebook user who is therefore familiar with the Facebook user interface. It is therefore assumed that actions such as "navigate to a given friend's profile" can be performed without additional aid. It is also assumed that the user will follow their usual actions when trying to perform an encrypted operation - when trying to upload an encrypted image they will, for example, simply follow the normal procedure for uploading an image rather than searching for some hidden option. Finally, we make the assumption that the extension has been installed and enabled and that the toolbar has not been hidden.

4.4.2 Uploading an encrypted image

A user wishing to send an encrypted image to 405 friends who also have the extension, does so.

Action Sequence

1. Add any public keys required.
2. Navigate to the image upload page for the required album (see Figure **??**).
3. Select the image to encrypted.
4. Check the `[Encrypt]` check box.
5. Click the `[Submit]` button.
6. Use the friend selector to select recipients and submit.

Defence of Credibility

- The user knows he must add public keys of recipients beforehand and how to do so. If the user has no public keys Appendix **??** demonstrates that he will be able to add one of his intended recipients. Since the process is reasonably simple - simply navigate to their page and click a clearly marked button, we assume that any user who has performed it once can do so again if they wish, without further instruction. The link between adding

public keys and being able to choose those friends as recipients should be fairly obvious; when the first key is added and encryption attempted again that same friend will appear as the only possible recipient.

- Facebook user will be familiar with the process of uploading an image. We assume a user trying to upload an encrypted image will be likely to try the method they are already familiar with.
- User knows things are OK since he sees the encryption check box option on arriving at the upload page. Check box leaves little room for confusion over whether an upload will or won't be encrypted.
- User knows to select an image to upload since the process is identical to uploading plaintext images.
- User knows to select `[Encrypt]` check box since uploading an encrypted image is the original task.
- User knows things are OK since the recipient selector pops up.
- User knows how to use the recipient selector (see Appendix **??**).
- User knows things are OK as Facebook handles the upload process from here as per normal, notifying user when the upload is complete.

4.4.3 Posting a comment

A user has navigated to his news feed. He wishes to write an encrypted reply to a plaintext comment on a news feed post of an encrypted photo. It is assumed he already possesses the public keys required and can decrypt the photo in question.

Action Sequence

1. Select the comment box below the plaintext reply.
2. Type comment in plaintext.
3. Click the `[Encrypt & Submit]` button.
4. Use the friend selector to select recipients and submit.
5. Refresh the page to review comment.

Figure 4.7: Input fields and controls for submitting encrypted images.

Defence of Credibility

- User knows to select the comment box. This is required for submitting a plaintext comment, we assume the user will try the method they are used to.
- User knows things are OK as the `[Encrypt & Submit]` appears once the textbox has focus.
- User knows to type in the comment - this is identical to submitting a plaintext comment.
- User knows to click `[Encrypt & Submit]`. The user is familiar with clicking a similar submit button during plaintext entry. The button is positioned beside the `[Submit]` for plaintext entry, is styled like the `[Submit]` button and is clearly labelled. Submitting an encrypted comment is part of the original task.
- User knows how to use the recipient selector (see Appendix **??**).
- User knows things are OK as Facebook handles the submission process from here as per normal. A loading icon appears briefly, then the comment itself appears.
- User knows that their submission was encrypted as this fact is stated as part of the comment encoding.
- User knows they must refresh the page to view their own comment. Even if this is their first encrypted submission, Facebook users will be familiar with the process of refreshing a page when something is not working or to view an update. If the user does not make the connection right away, they should realise that encrypted text submissions are decrypted as a page first loads when they return to the page and see the item decrypted automatically.

Figure 4.8: Input fields and controls for posting a comment to a news feed entry.

5 Conclusion

To conclude, we evaluate the project against the requirements in Section 2.8 and take a retrospective look at what would have been done differently given the benefit of hindsight. We finish with a discussion of the possible barriers to potential deployment and future work that might be relevant.

5.1 Evaluation of Requirements

We believe that all the requirements described in Section 2.8 were met in full:

Requirement 1 The extensions will be able to broadcast-encrypt, submit, retrieve and decipher the following objects:

- Status updates
- Wall posts
- Comments
- Messages
- Images

Defense Submission and retrieval of comments and images is demonstrated in Section **??**, and of status updates in Section 4.3. Section 3.2 describes how other encrypted items and submission controls are supported.

Requirement 2 The size limits for encrypted text objects will be no smaller than the current limits Facebook imposes, given in Section 2.5.1).

Defense Section 4.2 demonstrates submitting and decoding textual content 10,000 characters (the largest limit) in length. Since all content

types are stored in a note, the other limits in Section 2.5.1 are artificially imposed during sanitisation.

Requirement 3 The size limit for encrypted images will be no less than 50 KiB.

Defense Section 4.2 and Section 4.3 both demonstrate submitting and retrieving images of 50 KiB in length.

Requirement 4 All requirements will be met given recipient groups sizes up to 400, based on the discussion in Section 2.5.2.

Defense All the examples given in this section use recipient group sizes of 400 where applicable.

Requirement 5 All response times will lie within acceptable limits, in accordance with [40].

Defense All response times in Section 4.3 are under 10 seconds — the recommended limit for response times with no progress indicator. The one exception is the page load time for an entire news feed of non-cached encrypted images. Section 4.3 shows in this case the time between successive loads is less than one second on average. Intermittent loading of images may be considered an acceptable progress indicator [40].

Requirement 6 The project will adhere to the security policy described in 2.7.1.

Defense All results presented in Chaper 4 demonstrate the use of 256-bit and 2048 bit key lengths for AES and RSA respectively, which exceeds the required minimum (128-bit AES and 2048-bit RSA). Chapter 3 describes the implementation of remaining security measures.

Requirement 7 The number of news feed entries generated by encrypted submission will be no more than the number generated by normal content submission.

Defense Section 4.3 relies on the fact that only encrypted items themselves were posted to the news feed. Section 3.4.3 details the delete mechanism used to ensure this is the case.

Requirement 8 The application will adopt a modular structure that facilitates switching between different schemes and permits future extension.

Defense Section 3.1.2 describes the abstract factory pattern used for libary sub-components. Sections 4.1 demonstrates an example of polymorphic use of the IConduitImage sub-component.

5.2 Retrospective

In retrospect is would have been better not to implement the HWT coding method due to its poor capacity. One alternative might have been to explore encoding data in the low frequency coefficients of a lossless DCT-like transform such as that used in the upcoming JPEG XR format [9].

During the testing carried out in Section 4.1, provisional tests using multiple library instances were performed. Inspection of CPU utilisation and timing results suggests that parallelisation of the encoding/decoding process would offer considerable speed improvements. Ideally multi-threading capability should have been included as a initial requirement since this might be difficult to retrofit.

Facebook also recently increased their image size limits. A discussion of the modified approach required is included in Appendix **??**.

5.3 Barriers to deployment

Several issues need to be overcome to move the extension from being a dissertation project to something ready for deployment.

Given the network effects involved, any system should be available for as many platforms as possible. Currently the extension is only tested on Linux, though Firefox extensions are generally well suited to cross platform deployment. With regard to deployment on multiple browsers, the structure presented in Section 3.1.1 means that while both the Toolbar XUL and large parts of the `efb` module would have to be re-developed, the remainder of the project could be used largely unmodified.

Network effects also mean that splitting the user base would be highly undesirable. Ideally the extension should not only support a modular, extensible structure, but also backwards compatibility between newer and older versions. In particular this would require a canonical JPEG-immune

signature that describes what modules were used to encode an image. One possibility would be to use the n-bit scaling method, with $n = 1$.

The Botan, CImg and Schifra library components are distributed under the FreeBSD, CeCILL and GPLv2[1] licenses respectively, which are all GPL compatible [18]. Proper care would be required to ensure distribution adhered to the terms of the GPL.

5.4 Future work

A more advanced broadcast encryption scheme is presented in Appendix **??**, partly based on Boneh et al [5]. In brief, the scheme involves appending transmission overheads to the user's public key rather than to the message.[2] Session keys can then be re-used when transmitting to the same recipient group. The key expiration period[3] could provide a sliding parameter that trades lower amortised storage overheads for better security. In addition, by running cleanup routines periodically on the public key, old keys can be deleted both to save space and to give content a finite lifetime.

Finding an optimal approach to high capacity JPEG-immune image coding would be an interesting research-level problem. One promising avenue might be the use of a dirty paper coding scheme [7].

[1] Provided the use is non-profit.

[2] This would involve building a data structure (e.g. a linked list of Facebook Notes) to store the public key.

[3] NIST recommend key re-use periods from a few months up to 2 years, depending on the nature of the message. [3]

Bibliography

[1] Pekka Abrahamsson et al. *Agile software development methods*. Tech. rep. VTT Publications, 2002. URL: http://vtt.fi/vtt_show_record.jsp?target=julk&form=sdefe&search=44278.

[2] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. 1st. New York, NY, USA: John Wiley & Sons, Inc., 2001. ISBN: 0471389226.

[3] E. Barker et al. «NIST Special Publication 800-57: Recommendation for Key Management Part1 - General». In: (2007).

[4] Matt Blaze, Gerrit Bleumer, and Martin Strauss. «Divertible protocols and atomic proxy cryptography». In: *In EUROCRYPT*. Springer-Verlag, 1998, pp. 127–144.

[5] Dan Boneh, Craig Gentry, and Brent Waters. «Collusion Resistant Broadcast Encryption with Short Ciphertexts and Private Keys». In: Springer-Verlag, 2005, pp. 258–275.

[6] William E. Burr et al. *DRAFT i Draft Special Publication 800-63-1 Electronic Authentication Guideline*. 2008.

[7] M. Costa. «Writing on dirty paper (Corresp.)» In: *Information Theory, IEEE Transactions on* 29.3 (May 1983), pp. 439 –441. ISSN: 0018-9448. DOI: 10.1109/TIT.1983.1056659.

[8] Lisa Crispin and Janet Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. 1st ed. Addison-Wesley Professional, 2009. ISBN: 0321534468, 9780321534460.

[9] Francesca De Simone et al. «A comparative study of JPEG 2000, AVC/H.264, and HD Photo». In: *SPIE Optics and Photonics, Applications of Digital Image Processing XXX*. Vol. 6696. San Diego, CA USA, 2007.

[10] *Diaspora Alpha*. 2011. URL: https://joindiaspora.com/.

[11] Assistant privacy commissioner of Canada Elizabeth Denham. *Report of findings into the complaint filed by the Canadian internet policy and public interest clinic against Facebook Inc.* July 2009. URL: http://www.priv.gc.ca/cf-dc/2009/2009_008_0716_e.pdf.

[12] *Facebook employees know what profiles you look at.* 2007. URL: http://gawker.com/#!315901/scoop/facebook-employees-knowwhat-profiles-you-look-at.

[13] *Facebook uncovers user data sale.* 2010. URL: http://www.bbc.co.uk/news/technology-11665120.

[14] Amos Fiat and Moni Naor. «Broadcast Encryption». In: *Advances in Cryptology — CRYPTO' 93*. Ed. by Douglas Stinson. Vol. 773. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 1994, pp. 480–491. URL: http://dx.doi.org/10.1007/3-540-48329-2_40.

[15] *FireGPG.* 2011. URL: http://getfiregpg.org/s/home.

[16] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[17] *GeoSoft Coding Style Guide.* 2011. URL: http://geosoft.no/development/cppstyle.html.

[18] GNU. 2009. URL: http://www.gnu.org/licenses/license-list.html#PythonOld.

[19] Scott Golder, Dennis Wilkinson, and Bernardo Huberman. «Rhythms of social interaction: Messaging within a massive online network». In: *Proc. 3rd Intl. Conf. on Communities and Technologies*. 2007.

[20] Google. *Rolling out sandbox for Adobe Flash.* 2010. URL: http://blog.chromium.org/2010/12/rolling-out-sandbox-for-adobe-flash.html.

[21] Jim Gray and Catharine van Ingen. «Empirical Measurements of Disk Failure Rates and Error Rates». In: *CoRR* abs/cs/0701166 (2007).

[22] Ralph Gross and Alessandro Acquisti. «Information revelation and privacy in online social networks». In: *Proceedings of the 2005 ACM workshop on Privacy in the electronic society*. WPES '05. New York, NY, USA: ACM, 2005, pp. 71–80. ISBN: 1-59593-228-3. URL: http://doi.acm.org/10.1145/1102199.1102214.

[23] Independent JPEG Group. *JPEG FAQ*. URL: http://www.faqs.org/faqs/jpeg-faq/.

[24] Saikat Guha, Kevin Tang, and Paul Francis. «NOYB: privacy in online social networks». In: *Proceedings of the first workshop on Online social networks*. WOSN '08. New York, NY, USA: ACM, 2008, pp. 49–54. ISBN: 978-1-60558-182-8. DOI: http://doi.acm.org/10.1145/1397735.1397747. URL: http://doi.acm.org/10.1145/1397735.1397747.

[25] James Hendler and Jennifer Golbeck. «Metcalfe's law, Web 2.0, and the Semantic Web». In: *Web Semantics: Science, Services and Agents on the World Wide Web* 6.1 (2008). Semantic Web and Web 2.0, pp. 14–20. ISSN: 1570-8268. DOI: DOI:10.1016/j.websem.2007.11.008. URL: http://www.sciencedirect.com/science/article/B758F-4R6JP09-2/2/bc3ba99ebfcaf2fd3cf0077a55bd891d.

[26] R. A. Hill and R. I. M. Dunbar. «Social network size in humans». In: *Human Nature* 14 (), pp. 53–72. URL: http://www.liv.ac.uk/evolpsyc/Hill_Dunbar_networks.pdf.

[27] Matthew J. Hodge. «The Fourth Amendment and privacy issues on the "new" internet: Facebook.com and MySpace.com.» In: *Southern Illinois University Law Journal* 31 (2006), pp. 95–122. URL: http://www.law.siu.edu/research/31fallpdf/fourthamendment.pdf.

[28] Jeremy Horwitz. *A Survey of Broadcast Encryption*. Tech. rep. 2003.

[29] *How Facebook Makes Money*. 2010. URL: http://www.allfacebook.com/facebook-makes-money-2010-01.

[30] Facebook Inc. *The official Facebook.com statistics factsheet*. June 2011. URL: http://www.facebook.com/press/info.php?factsheet.

[31] Mozilla Inc. *Firefox 4.0 System Requirements*. 2011. URL: http://www.mozilla.com/en-US/firefox/4.0/system-requirements/.

[32] Tom N. Jagatic et al. «Social phishing». In: *Commun. ACM* 50 (10 Oct. 2007), pp. 94–100. ISSN: 0001-0782. DOI: http://doi.acm.org/10.1145/1290958.1290968. URL: http://doi.acm.org/10.1145/1290958.1290968.

[33] Matthew M. Lucas and Nikita Borisov. «FlyByNight: mitigating the privacy risks of social networking». In: *Proceedings of the 7th ACM workshop on Privacy in the electronic society*. WPES '08. New York, NY, USA: ACM, 2008, pp. 1–8. ISBN: 978-1-60558-289-4. DOI: http://doi.acm.org/10.1145/1456403.1456405. URL: http://doi.acm.org/10.1145/1456403.1456405.

[34] Wanying Luo, Qi Xie, and U. Hengartner. «FaceCloak: An Architecture for User Privacy on Social Networking Sites». In: *Computational Science and Engineering, 2009. CSE '09. International Conference on*. Vol. 3. Aug. 2009, pp. 26 –33. DOI: 10.1109/CSE.2009.387.

[35] C. Mccarty et al. «Comparing Two Methods for Estimating Network Size». In: *Human Organization* 60.1 (2001), pp. 28–39.

[36] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Professional, 2006. ISBN: 0321356705.

[37] Mozilla. *Bundling multiple binary components*. 2010. URL: https://developer.mozilla.org/en/Bundling_multiple_binary_components.

[38] Mozilla. *JavaScript Code Modules: ctypes.jsm*. 2010. URL: https://developer.mozilla.org/en/JavaScript_code_modules/ctypes.jsm.

[39] *Net Index by Ookla - household download index, UK*. 2011. URL: http://www.netindex.com/download/2,4/United-Kingdom/.

[40] Jakob Nielsen. *Usability Engineering*. San Francisco, California: Morgan Kaufmann Publishers, Oct. 1994. ISBN: 0125184069.

[41] J. P. Odenwalder. *Error Control Coding Handbook*. Linkabit Corporation, 1976.

[42] *Pidder*. 2011. URL: https://www.pidder.com/en/index.html.

[43] The Washington Post. *From Facebook, answering privacy concerns with new settings*. May 2010. URL: http://www.washingtonpost.com/wp-dyn/content/article/2010/05/23/AR2010052303828.html.

[44] Bruce Schneier. *Crypto-Gram Newsletter: July 15, 2000*. July 2000.

[45] Online Schools. *Obsessed with Facebook*. 2011. URL: http://www.onlineschools.org/blog/facebook-obsession/.

[46] C. Shapiro and H. Varian. *Information rules*. Harvard Business School Press, 1998. URL: http://www.utdallas.edu/~liebowit/palgrave/network.html.

[47] MG Siegler. *TechCrunch - RockYou hacked*. 2009. URL: http://techcrunch.com/2009/12/14/rockyou-hacked/.

[48] *uProtect.it - Take Back Control From Facebook*. 2011. URL: http://uprotect.it/.

[49] C. Wharton et al. «The Cognitive Walkthrough Method: A Practitioner's Guide». In: (1994).

[50] Alma Whitten and J. D. Tygar. «Why Johnny can't encrypt: a usability evaluation of PGP 5.0». In: *Proceedings of the 8th conference on USENIX Security Symposium - Volume 8*. Berkeley, CA, USA: USENIX Association, 1999, pp. 14–14. URL: http://portal.acm.org/citation.cfm?id=1251421.1251435.

[51] Hongjun Wu and Di Ma. «Efficient and secure encryption schemes for JPEG2000». In: *Acoustics, Speech, and Signal Processing, 2004. Proceedings. (ICASSP '04). IEEE International Conference on*. Vol. 5. May 2004, DOI: 10.1109/ICASSP.2004.1327249.

[52] Jianyun Xu et al. «JPEG Compression Immune Steganography Using Wavelet Transform». In: *Information Technology: Coding and Computing, International Conference on* 2 (2004), p. 704. DOI: http://doi.ieeecomputersociety.org/10.1109/ITCC.2004.1286737.

Appendix

A Broadcast Encryption

Broadcast encryption was originally conceived for applications such as satellite TV subsription services and DVD content protection. Information is broadcast to a large number of recipients, a small subset of whom are 'revoked users' and thus can't decipher the information [14].

We describe here a more recent variation where only a small subset of 'priveledged users' can decipher the message, since this is more applicable to social networks [28]. We begin with a general definition and outline two basic schemes (one of which is a variant of that used in Encrypted Facebook). Section **??** details a more advanced scheme.

A.1 Definition

For simplicity we describe broadcast encryption as a key encapsulation mechanism. A broadcast encryption scheme solves the general problem of transmitting a session key k to any subset R of priviledged users from a universe U in such a way that a coalition of non-priviledged users cannot learn the key [14]. Further communications can then performed securely between the broadcaster and priveledged set under the shared session key.

We use the following definition adapted from [14] and [5]:

Definition A.1. A broadcast-encryption scheme is a triple of algorithms (SETUP, BROADCAST, DECRYPT) such that:

- The setup algorithm (SETUP) takes a user $u \in U$ and constructs that receiver's private information $p_u \in P$ (for some set P associated with the scheme).
- The broadcast algorithm (BROADCAST) takes the list of priveledge users R and the session key $k \in K$ and outputs a broadcast message $b \in B$ (for some message space B associated with the scheme).

- A user $u \in U$ runs the decryption algorithm DECRYPT(b, p_u, u) that will:
  - Compute the k associated with b, if $u \in R$.
  - DECRYPT fails, if $u \notin R$.

A.2 Two simple schemes

We begin by describing two simple broadcast encryption schemes. We assume the existence of a secure public key infrastructure which can be used to establish shared private symmetric keys between the broadcaster and any other user.

Scheme 1.

- SETUP generates a private key $p_u \in P$ for each $u \in U$.
- BROADCAST outputs b, a concatenation of pairs (u, k_u) for each $u \in R$. k_u is the session key k encrypted under the private key p_u.
- If $u \in R$ then DECRYPT obtains the session key k by decrypting k_u. If $u \notin R$ then DECRYPT fails.

Scheme **??** requires each recipient to store $O(1)$ private keys and the broadcaster to store $O(|U|)$ keys. Equivalently, we could use a public key infrastructure with one fixed length private and public key for each user. Each broadcast message b is $O(|R|)$ in length.

Scheme 2.

- SETUP generates a private key p_S for every subset of users $S \subseteq U$. Each user u's private information p_u is the concatenation of all (S, p_S) such that $u \in S$.
- BROADCAST outputs b, a pair (R, k_S). k_S is the session key k encrypted under the private key p_S, where S is the set of priveledged users $R^{\complement}$.
- If $u \in R$ then u will posses k_S as part of their private information p_u and DECRYPT will decrypt k_S to obtain the session key k. If $u \notin R$ then DECRYPT fails.

Scheme **??** requires each recipient to store $O(2^{|R|})$ keys and the broadcaster to store $O(2^{|U|})$ keys. Equivalently, we could use a public key infrastructure with public and private keys of length $O(2^{|U|})$. Each broadcast message b is $O(\log|R|)$ in length.

A.3 Variants

Broadcast encryption attempt to find a suitable middle-grounds between the **??** and **??** In general, schemes can be classified by the following properties:

- The size of the public key, usually in relation to the number of recipients.
- The size of the private key, again usually in relation to the number of recipients.
- The size of the broadcast message, sometimes called the transmission overhead.

Schemes also vary in their complexity of implementation and computational requirements. Some schemes require users to remain connected to receive key-update messages. Others are optimised for recipient sets which remain reasonably constant.

Encrypted Facebook uses a variation of Scheme **??**. Though transmission overheads are high the scheme is still feasible given the connectedness of Facebook users. The principle advantages are that Scheme **??** has been proven to be secure given a secure underlying encryption scheme [28]. The scheme is also reasonable simple to implement compared to more advanced variants, and doesn't rely on key-update messages which would be considerably complex to implement without a third party server.

A.4 A more advanced scheme

We propose a possible improvement to the scheme used by Encrypted Facebook, based partly on Boneh et al [5]. Recall that the tranmission overhead is a list of pairs (u, k_u) where k_u is the session key k encrypted under a user specific key p_u. Equivalently, using public key cryptography, k_u is the session key k encrypted under a user specific public key $k_u.pub$, which may be decrypted with $k_u.priv$. Instead of storing the $O(R)$ transmission overhead with the message the sender can publish it as part of their public key and append a fixed size index to the message. Now messages and private keys are fixed size, whereas public keys are $O(AR)$, where R is the number of messages sent so far.

All else being equal this scheme requires the same amount of storage overheard — session keys are now simply obtained through another layer of indirection. However, we may re-use session keys when sharing content with the same recipient group. This allows us to trade security for lower storage overheads by changing the key usage period. [1] In addition, by running cleanup routines periodically on the public key, old keys can be deleted both to save space and to give content a finite lifetime.

[1] NIST recommendations for usage periods range from serveral weeks up to two years, depending on the nature of the message [3].

B Graph API

Objects can be read simply by making GET requests to `https://graph.facebook.com/ID` and parsing the return result as a JSON object. For none public objects we will also require an access token. Facebook uses the OAuth 2.0 protocol. An access token can be obtained by handling the page redirect [1] after the following GET request:

Listing B.1: Authentication request

```
https://www.facebook.com/dialog/oauth?
    client_id=APP_ID&
    redirect_uri=REDIRECT_URL&
    scope=SCOPE_VAR1,SCOPE_VAR2,SCOPE_VAR3&
    response_type=token
```

Content can also be published in a similar way. An example request might look like:

Listing B.2: Publishing request

```
https://graph.facebook.com/ID/feed?
    access_token=ACCESS_TOKEN&
    message=MESSAGE
```

There are several caveats:

- When publishing images, although the operation is supported, getting a correct handle to the image is difficult due to JavaScript's poor

[1] To ensure authentication can only occur in client side code the access token is passed in a URI fragment.

support for working with local files. The workaround requires creating an invisible form on the current page with a file input element and extracting the file handle from there.

- Images have to be published to an album. Facebook currently uses two types of album ID, one which appears within web pages and one which can be used for publishing through the Graph API. An additional API query is required before uploading to translate from one format to the other.
- It certain cases, though publishing through the Graph API is possible, it is more convenient to programmatically manipulate form controls. An example is when submitting a comment and triggering the click handler for the submit button.
- Modifying the "About Me" section of a users profile is unsupported entirely. The workaround requires creating an invisible iframe on the current page and manipulating a form on the Facebook site within.

C Compression process

Information is lost at several stages in the compression process:

1. Colour images are subject to a lossy colour space transform from RGB to YCrCb.
2. The chrominance components Cr and Cb are subsampled at a rate half that of the luminance channel.
3. The discrete cosine transform is applied to each 8x8 block using finite arithmetic.
4. DCT coefficients are quantised according to values in a quantisation matrix.

Note that chrominance subsampling means that colour images only provide a 50% increase in capacity over a grayscale image of the same resolution. For simplicity we will only consider grayscale images. This leaves quantisation as the principle step at which information loss occurs.

Table **??** displays the quantisation matrix used for a grayscale JPEG image downloaded from Facebook. Using this and several other compression parameters our best guess is that Facebook is using the `libjpeg` library for compression, with a quality factor setting of 85. [1]

[1] Based on the output of the JPEG Snoop application for Windows.

Quantisation Matrix							
5	3	3	5	7	12	15	18
4	4	4	6	8	17	18	17
4	4	5	7	12	17	21	17
4	5	7	9	15	26	24	19
5	7	11	17	20	33	31	23
7	11	17	19	24	31	34	28
15	19	23	26	31	36	36	30
22	28	29	29	34	30	31	30

Table C.1: Quantisation matrix used by Facebook for luminance channel.

D DCT bitmask coding

The DCT bitmask encoding method works by masking off both high and low order bits from DCT coefficients.

- Low order bits are masked off according the quantisation matrix, obtained from a sample JPEG of the correct quality factor. For a quantisation coefficient c, $\lceil \log_2 c \rceil$ bits are masked off.
- High order bits are masked off according to some global constant, in an attempt to reduce incidence of capping.

The process of extracting a mask using quality factor 85 is given in the following code sample:

Listing D.1: DCT bitmask method

```
# First read in a sample Facebook image
sam1 = imread('sample.jpg');
imwrite(sam1, 'sample2.jpg', 'Quality', 85);
sam = jpeg_read('sample2.jpg');

# Get the quant matrix for the luminance
q = sam.quant_tables{sam.comp_info(1).quant_tbl_no};

# Convert to bit mask
m = uint16( ceil( log2(q) ) );
m = 2.^(m+1);
m = bitor( m, 2*m );
m = bitor( m, 2*m );
m = bitand(m , ones(8)*0x007f );
```

Though error rates were succesfully reduced, Section 2.6.1 attests that this method does not offer improved theoretical capacity over encoding in RGB values.

E Threat analysis

E.1 Identification of threats

We ignore threats that would be present anyway (e.g. if HTTPS isn't used when browsing) and only consider additional threats introduced by the application. We take an attack centric approach and identify a number of possible threats:

Attack 1 Attacker breaks the encryption scheme by brute force methods.

Attack 2 Attacker gains access to user's computer through the download and execution of either a JPEG file or a public key file.

Attack 3 Attacker gains access to user's computer by injecting code into `eval()` which runs outside the browser sandbox.

Attack 4 Attacker gains access to Facebook account through browser code injection, outside the sandbox. Typically this could involve cross-site scripting.

Attack 5 Attacker carries out middle-person attack by intercepting and switching public keys.

Attack 6 Attacker causes DoS by creating a malicious object. An example could be a self referential object, or an overly large object.

E.2 Risk analysis

We define risk as $Vulnerability \times Threat \times Impact$ based on [36].

Attack 1 Medium impact (loss of privacy) but low vulnerability provided proper key lenths are used and proper protocols observed. Low risk.

Attack 2 Executing arbitrary code on the user's machine is the highest impact possible. However, vulnerability is low since getting code to be executed once downloaded would be difficult. Medium risk.

Attack 3 Again, executing arbitrary code on the user's machine is the highest impact possible. Vulnerability is medium. We identify this as a high risk attack.

Attack 4 This is medium impact (loss of privacy) but high vulnerability since code injection of user defined input is happening automatically, without mandate from the user. We identify this as a high risk attack.

Attack 5 Impact is medium (loss of privacy) and vulnerability is low since Graph API calls use HTTPS. Low risk.

Attack 6 Impact is low (loss of availability) but vulnerability is high, again due to automated decryption. Medium risk.

E.3 Proposed measures

We propose a set of measures that are proportionate to risk.

Attack 1 Ensure key sizes are appropriate given the type of attacker. See Section 2.7.2.

Attack 2 Ensure only legitimate JPEG and public key files are downloaded. Public key files can be vetted on their character set and size, JPEG's by their extension.

Attack 3 Wrap all calls to `eval()` with a `secureEval()` function which attempts to prevent malicious use.

Attack 4 Sanitize all user inputs and ensure sanitisation can't be bypassed e.g. through the UTF-8 decoder[44]. Also sanitize output whenever inserting code into the DOM.

Attack 5 Informing the user whenever public keys are updated removes a large amount of risk. Complete protection would be impractical without use of OOB channels.

Attack 6 This need not be a large problem since decryption is built to fail gracefully (see Section 3.2.2). Include some simple run time checks to limit iteration and special characters (combined with proper sanitisation) to prevent tags-within-tags.

E.4 Security testing plan

We outline a partial plan for testing some of the proposed measures during development.

- Testing of secureEval should ensure that only content in the repose text from the Facebook domain is parsed, and that the contents is a JSON object only.
- Testing of sanitisation and UTF-8 decoder can be performed using boundary variable analysis.
- Attempt at creating malicious objects can be made to test decoder.

F Cognitive walkthough

F.1 Creating a cryptographic identity

A user who is not logged in but who knows the extension is installed, wishes to make use of some of the extensions functionality and his actions result in the creation of a (prerequisite) cryptographic identity.

Action Sequence

1. Log in to Facebook.
2. Click on the `[Create Identity]` button.
3. Enter a password according to the restrictions.
4. Re-enter password.

Defence of Credibility

- The user knows to click the `[Create Identity]` button. We know the user wants to use the extension in some way and is aware that the plugin is installed - it is reasonable to assume that he will look at the toolbar. All other buttons are greyed out, and the process of creating some form of identity/account/profile before using a service is familiar to anyone who has used Facebook, or any other site which requires membership.
- If the user isn't logged in to Facebook, he knows he must first do so since he will be informed of this on clicking `[Create Identity]`. User will know how to log in to Facebook.
- User will know how to cancel process as it is clearly marked.
- Quitting/crashing the browser or cancelling will result in a return to the initial state.

- User will either enter a valid password or be prompted by the restrictions on entering an invalid one, in which case he will then know how to enter a valid password and do so.
- User will know if he incorrectly re-enters password via an alert.
- User knows things are OK because an alert informs him the process was successful.

F.2 Logging in to a cryptographic identity

A user who is not logged in but has created an identity, wishes to make use of some of the extensions functionality and logs in.

Action Sequence

1. Log in to Facebook.
2. Click on the `[Start Encrypted Facebook]` button.
3. Enter user password.

Defence of Credibility

- User knows to click `[Start Encrypted Facebook]` . We know the user wants to use the extension in some way and is aware that the plugin is installed - it is reasonable to assume that he will look at the toolbar. All other buttons are greyed out apart from `[Create Identity]` which he has used before. If he does mistakenly click `[Create Identity]` he will be informed that he already has an identity and that attempting to overwrite it will result in irrevocable data loss. See figure **??**.
- If the user isn't logged in to Facebook he knows he must first do so since he will be informed of this on clicking `[Start Encrypted Facebook]`. User will know how to log in to Facebook.
- User will know his password and be able to enter it. In the case that the user doesn't know his password he will know he must create a new one and do so as described in Section **??**.

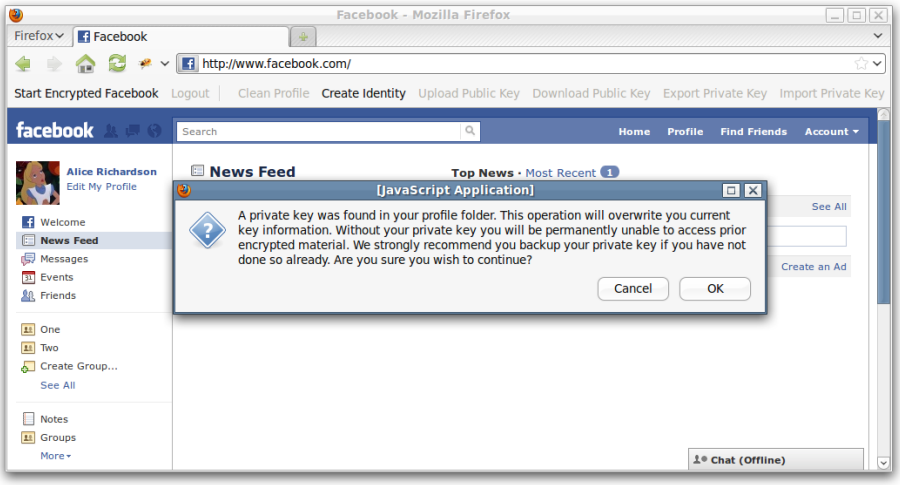


Figure F.1: Warning displayed if `[Create Identity]` *is mistakenly clicked.*

F.3 Adding public keys

A user with no public keys wishes to submit encrypted content. His actions lead to him obtaining one of his intended recipient's public key.

Action Sequence

1. Navigate to a friends profile.
2. Click the `[Add Public Key]` button.

Defence of Credibility

- Assume that a user wishing to perform some encrypted operation knows what process to follow. This is demonstrated in the walkthroughs for each respective action. Attempting this process with no public keys will result in a prompt informing the user that they must navigate to a friends profile and click the `[Add Public Key]` button in order to use them as a recipient.
- Facebook user will be familiar with the process of finding a friends profile and clicking on a button therein.
- User will know where the `[Add Public Key]` button is since it is clearly labelled and positioned at the top of the page next to several normal Facebook buttons.

F.4 Using the recipient selector control

A user presented with the recipient selector control wishes to use it to select a large subset of his friends for broadcast encryption, and does so. The user has 406 friends available to select and wants to choose 405 of them.

Action Sequence

1. Click the `[Select all]` check button
2. Deselect the friend who is to be excluded.
3. Click the `[Submit]` button in the pop-up window.

Defence of Credibility

- User knows that the friend selector is used to select recipients as this is stated clearly above the control.
- Users know generally how to select recipients by clicking on them one-by-one and also that they can deselect by clicking an already selected item, since this process of selecting recipients is familiar to them from general Facebook use and the UI control itself is based on Facebook's own.
- User knows to click `[Select all]`. The button is visible and clearly labelled; common sense would dictate that selecting all then deselecting one item would be quicker than selecting all 405 items individually.
- User knows things are OK. After clicking `[Select all]` all items will switch to looking selected in a manner consistent with other controls used by Facebook.
- User knows how to deselect a friend. Again based on the familiarity with the control from Facebook we assume they are capable of scrolling through the alphabetical list to find their the friend in question.
- User knows to click `[Submit]`. No button other than cancel is visible on the control, the button is clearly labelled and its appearance and position is based on Facebook's own controls.
- User knows things are OK as the pop-up responds to their click by disappearing, identical to the behaviour of a Facebook control.

G Boundary-value analysis

For the UTF-8 decoder we derived test cases using boundary value analysis. Below is a complete specification of accepted and invalid inputs.

- We accept any valid, non-overlong, UTF8 byte sequences, maximum length 4-bytes, with scalar value:
  - 0xB0 - 0xD7FF
  - 0xE000 - 0x100AF
  - 0x1B000 - 0x1BFFE (would-be surrogate pairs)
  - 0x10F0000 (indicates a padding byte was added, only one allowed per decode)
- We therefore must throw an exception whenever a valid UTF8 byte sequence is presented with scalar value:
  - 0x0 - 0xAF (out of range)
  - 0xD800 - 0xDFFF (surrogate pair characters)
  - 0x100B0 - 0x1AFFF (out of range)
  - 0x1BFFF - 0x10EFFFF (out of range)
  - 0x10F001 - 0x1FFFFF (out of range)
- We also throw an exception for valid UTF8 sequences when:
  - They have an overlong form i.e. the same scalar value can be represented using a shorter byte sequence.
  - They have scalar value 0x10F0000 (padding character) but this has already been seen during decoding.
  - They have scalar value 0x10F0000 (padding character) but the final decoded byte sequence (before padding removal) has length less than 2.

  - The final decoded byte sequence has length less than 1.
  - They are longer than 4-bytes.

- Naturally we reject any (invalid) UTF8 byte sequences with:
  - Unexpected continuation bytes when we expect a start character.
  - A start character which is not followed by the appropriate amount of valid continuation bytes - including start characters right at the end of a sequence.

For sanitisation, we only permit alphanumeric characters along with the a small selection of symbols `,.?!()'*&%$£`. Boundary-values can be easily computed from the ASCII character set.

H Image size increase

Facebook recently increased their minimum size limits from 720 × 720 pixels to 2048 × 2048.

After adapting the extension to work with the new sizes, preliminary testing shows that response times increase well beyond acceptable limits described in [40]. This is principally because the current implementation makes use of the entire image capacity regardless of the input size. This was justifiable practice with the old limits, as borne out by Chapters 4 and **??**..

The new limits call for the ability to create variable size images based on the input length. Some method of indicating any padding used would be required. One possibility would be storing a length tag in some canonical JPEG-immune format. An example might be to use the n-bit scaling method with $n = 1$. This could be combined with the signature scheme described in Section **??**

Another possibility would be to combine the IFec and IConduitImage components into one. This would allow a length tag to be encoded after error correction codes have been applied.

One problem with variable length images that was identified during preliminary testing, was that images can no longer be filtered on their dimensions. This causes a large number of extraneous Graph API image retrieval requests. The effect of this on performance would have to be taken into account.