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 2010 Information Theory and Coding course (not taught this year) 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

Acronyms

OOB Out Of Band

DOM Document Object Model

UML Unified Modelling Language

Acknowledgments

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 [18]. Encrypted Facebook is a Firefox extension which aims to protect privacy though the use of a broadcast encryption scheme when sharing certain content via the Facebook platform.

1.1 Background

Since its inception, Facebook has come under criticism in relation to online privacy [7]. As social netorks mimic real life interactions members are often willing to reveal more private details than they otherwise would, resulting in social networks accumulating a large repository of sensitive information [14]. 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 and its employees, [2] acquiescent disclosure to government authorities and unlawful access through phishing scams and security exploits [8] [20] [29].

Encrypting content ensures its secrecy in any eventuality, provided the encryption key itself is kept safe. Tools exist for performing encryption manually - some are even partly integrated into the Facebook user interface [10]. However, studies have suggested that software which requires manual management of cryptographic keys is not usable enough to provide effective security for most users [32]. In addition, interaction over

[1] Facebook was recently forced to update its privacy controls due to growing pressure from the public and media [26].

[2] As was the case in 2010 when personally identifiable Facebook user details were sold to data brokers by third party developers [9].

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

Solutions targetting 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 scaleable.

Attempts have been made at creating new social networks which use encryption and decentralised hosting to provide better privacy protection [25] [6]. 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 userbase. [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.
- **Useable** and therefore accessible to the typical Facebook user.
- **Scaleable** given the vast size of Facebook's userbase.
- **Incrementally deployable** in order to avoid some of the problems of network effects.

1.3 Limitations

Some objectives we consider non-goals, or outside the scope of the project:

- Securing the actual structure of the social graph.
- 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 [28] [16].

- Protection against middleperson attacks (see section 2.7.1).
- Guaranteeing availability of content. Completely severing reliance on Facebook would be tantamount to builing a new network.
- Concealing the existence of content itself or concealing the fact that it is encrypted. Steganography is employed but not for this reason (see section 3.4.3).
- Full cross-platform and/or cross-browser support (see section 5.4).

1.4 Existing work

Many applications for encrypting online data exist, most in the form of browser extensions [10]. There are also several applications which target Facebook specifically.

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 reciepients.
`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 look at the specific problems associated with encrypting images and discuss some additional security related concerns. Finally, we derive a concrete set of requirements and outline an appropriate software development methodology and testing strategy.

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 [15]. Arguably, however, privacy is only poorly preserved due to problems of inference control [2]. [1] Incremental deployment is also not possible since non-users will only ever see fictitious profiles [22].

[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, FaceCloack 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 scalebility. 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 targeting advertising to members based on the structure of the social graph [17]. 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 [30]. 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 [5].

By ensuring we use securely generated keys that are only ever stored on the user's device(s) we trade usability for better privacy protection. A criteria is given in section 2.7.3.

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 [22].

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 method would compromise the principle of secret key security.

2.2 Broadcast encryption

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

2.2.1 Existing solutions

Existing proposals tackle this problem in the following ways:

- 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.
- If a third-party is able to perform computation a technique called proxy re-encryption can be used, as with flyByNight [21]. Here the server changes the key under which the content may be decrypted on demand, without ever being able to read the cleartext itself [3].

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

- Distributing keys over OOB channels can permit one-to-many communication. A FaceCloak user, for example, shares a single decryption key OOB among friends [22].

None of these approaches are compatible with design principles stated in section 2.1.

2.2.2 Proposed scheme

Appendix XXX gives a full introduction to broadcast encryption. In general, schemes can be characterised by the size of each users private and public key and the amount of transmission overhead that must be sent with each message, given the number of recipients. The scheme presented here has a 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 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. A more advanced scheme which doesn't require key-updates is proposed in appendix XXX and discussed in section 5.4

Let U be the set of users with Encrypted Facebook installed and $R \subseteq U$ be a set of intended recipients.

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 and a message m, generates a session key k using scheme Q and broadcasts a message $b = (b_1, b_2)$ where:
  - b_1 is the list of pairs (u, k_u) such that $u \in R$, where k_u is the session key encrypted under pub_u.
  - b_2 is m encrypted under a session key k.
- A user $u \in U$ runs DECRYPT$(b, u, priv_u)$ that will:
  - If $(u, k_u) \in b_2$, extract the session key k from k_u using $priv_u$. m is obtained by decrypting b_2 using k
  - DECRYPT fails, if $(u, k_u) \notin b_2$ or if, equivalently, $u \notin R$.

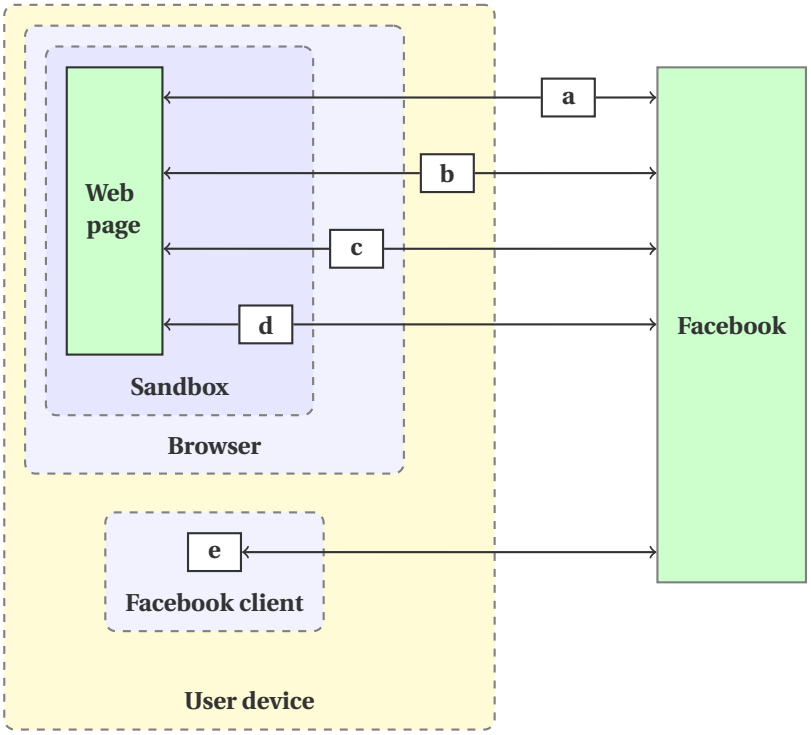

Figure 2.1.: *Possible deployment strategies for intercepting interaction with Facebook.*

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. [3] 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.

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 worlds most popular browser (as of January 2011). Manipulating 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 5.3.

Firefox extensions are written in JavaScript with partial support for binding library code written in Python or C/C++. Performing cryptography in JavaScript is possible but comes with severe performance difficulties [21]. 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 [4] version; supporting multiple versions is possible but non-trivial [**xpcom**].

[3] Flash applications, for example, are restricted but can provide basic filesystem access [**flash-sbox**].

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

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.*

- Loading native libraries with `js-ctypes`. Introduced in Gecko 2.0 [**js-ctypes**].
- 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 is used to load native code. This restricts 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.

2.5 The Facebook platform

Facebook represents all objects (e.g. users, messages, photos) 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.

Users may interact with Facebook in many different ways. We make the generalisation that interaction 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 XXX.

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. We therefore aim to encrypt only those most frequently used. From the data available these are Comments, Messages, Images and Posts (Table 2.1).

Broadcast encryption adds size overheads to any content that will be encrypted. Without relying on a third-party server all overheads must be stored on Facebook itself, in some form or another. Images and blog-style notes are obvious targets for storage utilisation due to their high capacity limits (see 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 XXX.

Each user's profile has a "Bio"[5] field with a character limit of XXX. With no other capacitous attribute that can be easily queried from a user ID, this is an obvious place to store a user's public key.

[5]Sometimes referred to as the "About me" field.

Activity	Frequency (per second)	Limitations	Notes
Comment	8,507	8,000 chars.	
Message	2,263	10,000 chars.	
Image	2,263	720 × 720 pixels	3-channel 8-bit colour. JPEG compressed (see section XXX).
Friend request	1,643		Social graph structure.
Status update	1,320	420 chars.	
Wall post	1,323	1,000 chars.	
Event invite	1,237		Social graph structure.
Photo tag	1,103		Social graph structure.
Link	833		
Like	unknown		Social graph structure.
Note	unknown	65,536 chars.	Used for blog-style posts.

Table 2.1.: *Facebook objects, their limitations and approximate frequency of creation [27]*

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 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 of sharply at around 250 [18] [13]. Marlow et al suggest that, regardless of the number of friends, communication is only ever between a small subset [4].

The Dunbar number is a theoretical cognitive limit to the number of peo-

ple a user can maintain relationships with and has been applied to social networks as well as face-to-face interactions. Exact estimates range from around 150 to 300 [**xxx**] [**xxx**], suggesting that the average degree of nodes within a social graph like Facebook's is unlikely to increase dramatically as Facebook expands further.

We this in mind we make it a requirement that the extension operates with recipient group sizes of up to 400.

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 content to non-useful content: typically spam, but in our case this could be transmission overheads as part of the broadcast encryption scheme or even the encrypted content itself since a user without Encrypted Facebook installed will be unable to see the cleartext.

In order to permit incremental deployment any system must ensure that its users can coexist with non-users. We therefore make it a requirement to limit the extensions impact on the signal-to-noise ratio of feeds.

Figure 2.3.: *News feed as presented on login.*

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.

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 [6] after the following GET request:

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

Listing 2.1: Authentication GET 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 2.2: Publishing POST request

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

<——> <————————————

Probably move this to Appendix ————————->

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 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 programatically 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.

<——————————————————————————————>

2.6 Storing data in images

As well as being one of the most frequently used types of content, images have been highlighted as a prime privacy concern on social networks [**fb-images**]. None of the existing work supports image encryption, though flyByNight discusses it as a possible extension.

The main problem is that, regardless of the input format, Facebook encodes all uploaded images using lossy JPEG. [7] This means we require some form of JPEG-immune coding to store the binary output of encryption in an image, such that even after undergoing compression we can exactly recover the original bytes.

We being by describing Facebook's JPEG compression process and evaluating naive attempts at encoding data in images, before proposing two more advanced approaches.

2.6.1 JPEG compression process

Information is lost at several stages:

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.

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

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 2.2.: *Quantisation matrix used by Facebook for luminance channel.*

Table 2.2 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. [8]

2.6.2 Naive data insertion

To motivate the need for a more complex scheme, we describe two naive methods of coding data in images.

One approach would be to encode directly into greyscale pixel values with enough redundancy to overcome the information lost during compression. Another approach might be to similarly encode data into DCT coefficients of 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 that don't map back to greyscale values in the range 0-255.

Figure XX graphs the bit error rate for both methods over a range of quality factors, using `libjpeg` for compression. For encoding in DCT coefficients a special bitmask, generated from the quantisation matrix, is used in an attempt to reduce the incidence of capping (see appendix XXX for details). Modeling the compression process as a binary symmetric channel, Figure XX graphs the theoretical per-image capacity of each

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

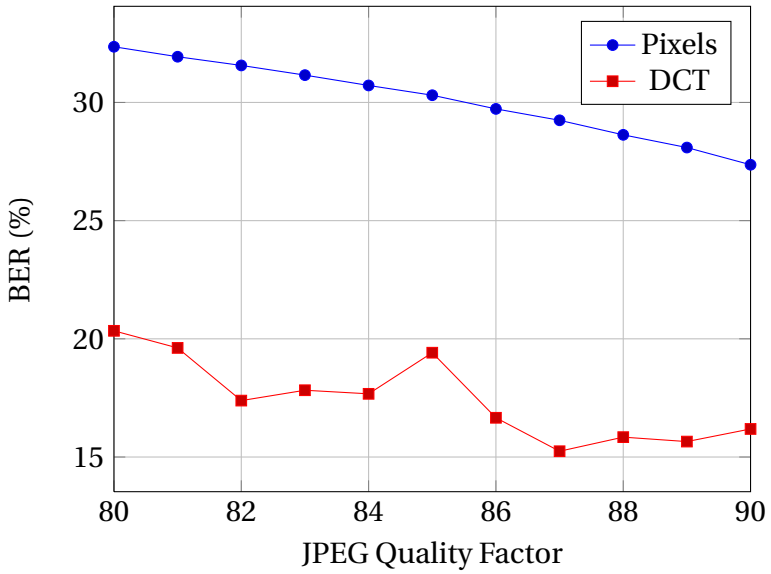

Figure 2.4.: *Bit error rate for varying quality factors.*

method for lossless data transmission (details for this calculation are given in Evaluation section XXX).

Given a recipient group size of 400 and ciphered session key size of 1024-bit (see section XXX), the transmission overhead required would be at least 50 KiB. This means neither of the naive approaches would be suitable, even given an error correction scheme which performs at or near the Shannon limit.

2.6.3 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 (i.e. losslessly) by using an integer lifting scheme. Xu et al demonstrate that data encoded in low-frequency

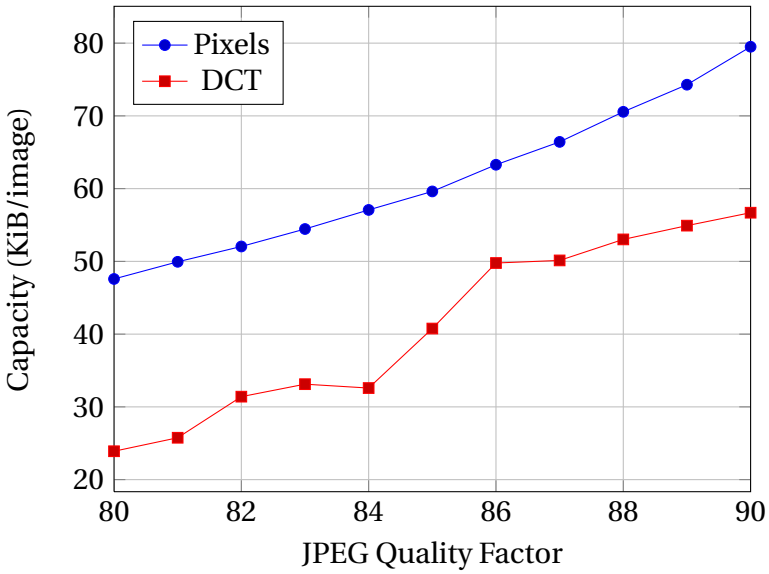

Figure 2.5.: *Per-image channel capacity (measured in KiB/image) for varying quality factors.*

HWT approximation coefficients can survive JPEG decompression when combined with an error correction scheme [33].

N-bit scaling method

This method maps the n-bit input space on to the 8-bit pixel space by scaling the input such that 0 corresponds to 0, and 2^n-1 corresponds to 255. The inverse process amounts to outputting the interval the pixel value lies in. This method works on the assumption that large changes in pixel intensity are unlikely.

Clearly both these schemes are sub-optimal and their exact properties (compression time and error rates) are unknown. We make 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 Further security considerations

In order to meet the goal of privacy protection we must consider proper security engineering practices. A particular concern is that of introducing new vulnerabilities into the browser platform. We perform a threat analysis and describe issue relating to the underlying cryptographic scheme and management of private keys.

2.7.1 Threat analysis

Full details of the threat and risk analysis are included in appendix XXX. We take an attack-centric approach, defining risk as (Vulnerability x Threat x Impact) according the methodology described by [**XXX**]. We identify several potential threats and possible security measures proportionate to the risk they pose:

Attack 1 Attacker breaks encryption scheme by brute force methods.

Measures Ensure key sizes are appropriate given the type of attacker. See section XXX.

Attack 2 Attacker gains access to user's computer by 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 users computer by injecting code into eval() which runs outside the browser sandbox.

Measures Wrap all calls to eval() with a SecureEval() function which sanitizes input and prevents malicious use.

Attack 4 Attacker gains access to Facebook account through browser code injection.

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

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

Measures: As stated in section 1.3 protection from this type of attack is a non-goal. However, informing the user whenever public keys are updated removes a large amount of risk.

Attack 6 Attacker causes DoS by creating a malicious object.

Measures Since the decryption process is built to fail gracefully (see section XXX) this need not be a large problem even in the worse case, provided some simple run time checks are included.

2.7.2 Underlying encryption schemes

We use AES and RSA as both schemes are approved by NIST standards [**nist**] and widely available via the Botan cryptography library. Ideally however, we would have liked to use a scheme based on elliptic curve cryptography since public key sizes are much smaller for the same amount of security than finite field or integer factorisation. This is important since the block size of the cipher depends on the public key, and this in turn determines the size of the session key once it has been encrypted. In our case, for example, a 256-bit AES session key requires 2048-bits of transmission overhead per recipient. Unfortunately ECC is less common in open libraries due to patent concerns [**XXX**].

Again, we make it a requirement that the project be modular enough to allow later use of a more suitable scheme.

2.7.3 Cryptographic keys

NIST(National Institute of Standard and Technology) provide recommendations on key size. Since 2010 NIST have recommended using at least 128-bits of security. This is equivalent to 3-TDEA (symmetric) or 2048-bit RSA (asymmetric). [**nist**]

NIST also give recommendations for acceptable psuedo-random number generators and hash functions when generating keys, which we adhere to. `/dev/random` is used as a source of entropy, an analysis of which is provided by http://www.pinkas.net/PAPERS/gpr06.pdf.

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

Specifically, key sizes will provide at least 128 bits of security according to the NIST standards described in section XXX.

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

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

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 XXX.

Requirement 5 To ensure the goal of usability is met, response times will meet the criteria described by `http://www.useit.com/papers/responsetime.html`.

Requirement 6 Should try not to introduce any security holes. Up to a point, given scope of project. We have already declared a threat model and testing strategy etc.

Requirement 7 The number of newsfeed 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 trad-offs associated with certain approaches to encryption and encoding data in images (and to a lesser extent error correction). 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 agile development style was adopted based on [**XXX**]. 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. Since the bulk of the project concerns encapsulating data in some layer of encoding the natural approach was 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 guidelines detailed here [12] 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.

2.10 Testing plan

General testing followed the agile testing principles of "test driven" development over "test last". The feedback loop between testing and implementation was kept as short as possible, to maintain flexibility over the exact approach.

2.10.1 Security testing

Special attention was 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 XXX).

2.10.2 Usability testing

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 evaluation chapter XXX.

2.10.3 Testing framework

In order to expedite testing often the Facebook image compression process was simulated locally. We used libjpeg at quality factor 85 since this most closely matches the compression signature of Facebook images.

The C++ module was developed and unit tested offline independently. Testing codes was built into extraneous library functions and `gprof` was used for profiling and debugging.

Once the JavaScript module was built integration testing was performed. ChromeBug, FireBug and FireUnit were used for testing, profiling and debugging.

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 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 describing the page interception and parsing process. We describe how keys are create, uploaded and managed. We then step through the process of encoding encrypted textual data and image data, detailing the relevant parts of the library as they are used.

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 XXX) the main body of the application is made up of five components (see Figure 3.1):

Toolbar XUL
: The toolbar XUL defines the toolbar interface using the Mozilla XML User Interface Language (XUL).

Page interception
: `pagecept` contains the HTML parser for extracting prospective decryption targets and inserting UI controls. Actual target processors and control event handlers are defined in `efb`. Updates due to changes in the Facebook web site and DOM injection security exploits are isolated to this component. Component re-use (e.g. in an extension for another browser) is also facilitated.

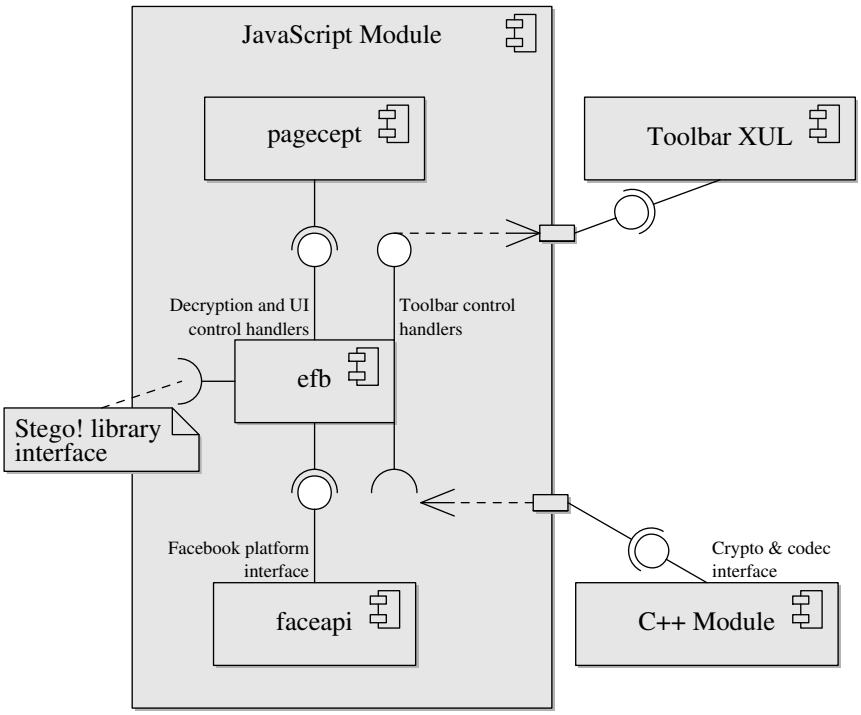

Figure 3.1.: *UML component diagram for the extension.*

Main extension component
`efb` defines the handlers for the toolbar and integrated UI controls. It contains handlers for decryption events and contains the plaintext cache data structures. It also contains 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 section XXX.

C++ Module
Primarily contains codec algorithms and cryptographic functions.

3.1.2 C++ module structure

The C++ module contains a library instance which implements the `IeFBLib` interface. The exposed behaviors 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 fascillitates 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 [1].

The library is built around the abstract factory pattern described in [11]. This allows us to encapsulate families of complimentary subcomponents since some interdependence exists between them. [2] Figure 3.2 outlines the pattern structure with an example concrete subclass `HaarWTConduitImage`.

[1] Since only one set of components currently provides a feasible solution, see evaluation section XXX

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

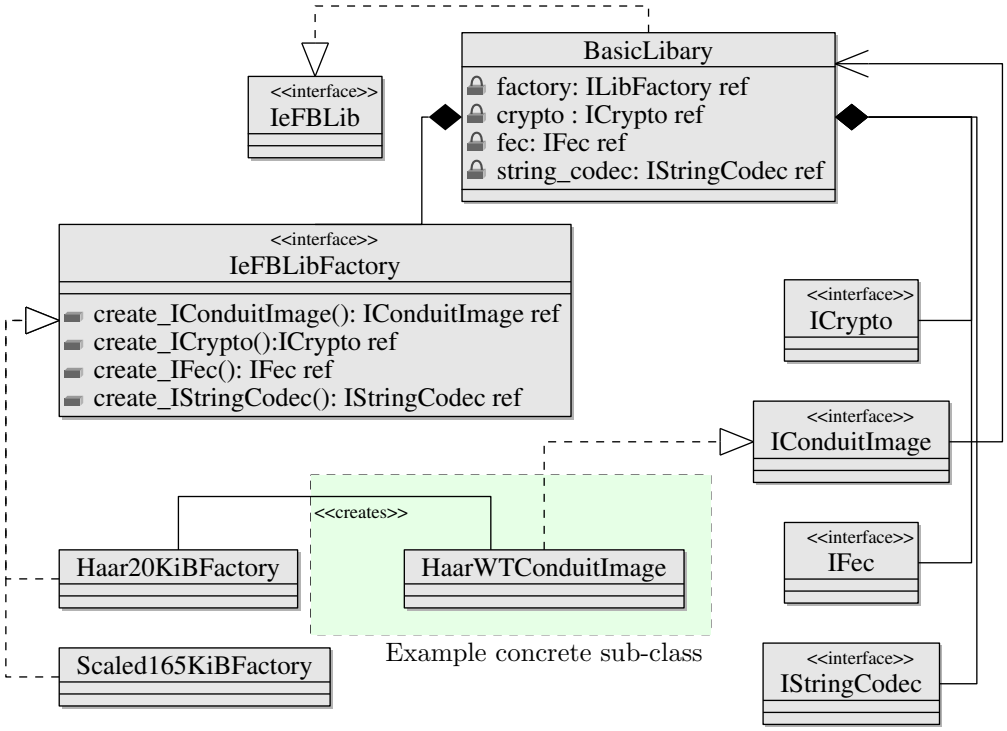

Figure 3.2.: *UML class diagrams for the library and its sub-components.*

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` button 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 page load events.

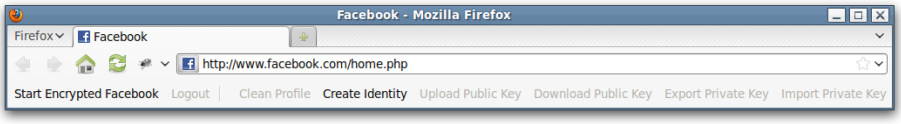

Figure 3.3.: *Toolbar before login.*

The toolbar buttons are disabled by default and become activated when appropriate. On first install only the `Create Identity` button will be enabled. Figure 3.3 shows the toolbar with an existing identity, before the user has logged in.

3.2 Integrating with the Facebook UI

The `pagecept` parser checks if a page is within the Facebook domain, then inserts additional UI controls and identifies potential decryption targets.

The parser is triggered whenever the Document Object Model (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

3.2.1 Inserting submission controls

There are four types of submission control, each of which can appear in multiple places within a single page. These are for general posts (status updates and wall posts) comments, private messages and image uploads. Each control is associated with an input field. Initially the parser identifies any control-field pairs using regular expressions.

Once a pair has been identified we generate an alternative encrypted submission control and place it beside the normal one (see Figure 3.7). A handler is generated 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. Elements within the control are populated with the user's names and profile pictures by performing queries through `faceapi`. Optionally, Encrypted Facebook will check if local public keys are up to date with online keys. The user is given the option of updating an out of date key,

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

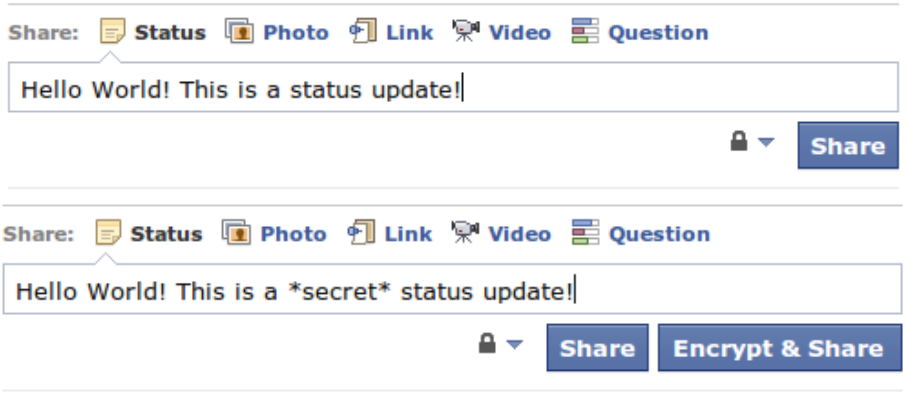

Figure 3.4.: *A control-field pair before (top) and after (bottom) parsing.*

and is informed of the tradeoff between vulnerability to middle person attacks and potential non-availability.

On submission, the selected list of friends and the input from the input field is gathered and processed by `efb` (detailed in section XXX). The result is a replacement input that can be either be uploaded via `faceapi` directly or inserted back into the input field.

3.2.2 Identifying decryption targets

Regular expression are also 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 case we take a best effort approach to filtering since a malicious user could easily create fake text tags and most Graph API images won't be encrypted. 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 an XmlHttpRequest triggered though `efb`. A

[4] Currently, since all images are 720 × 720 pixels images are filtered on dimension. The case for uploading variable sized images and its effect on filtering are discussed in section XXX.

Figure 3.5.: *Friend selector window.*

handler is attached to the request so that on completion, the cache can be updated appropriately. If an entry exists then several actions may be appropriate. If a valid plaintext 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.

<————- probably put a UML diagram here ——->

Figure 3.6.: *A newsfeed excerpt before (top) and after (bottom) parsing.*

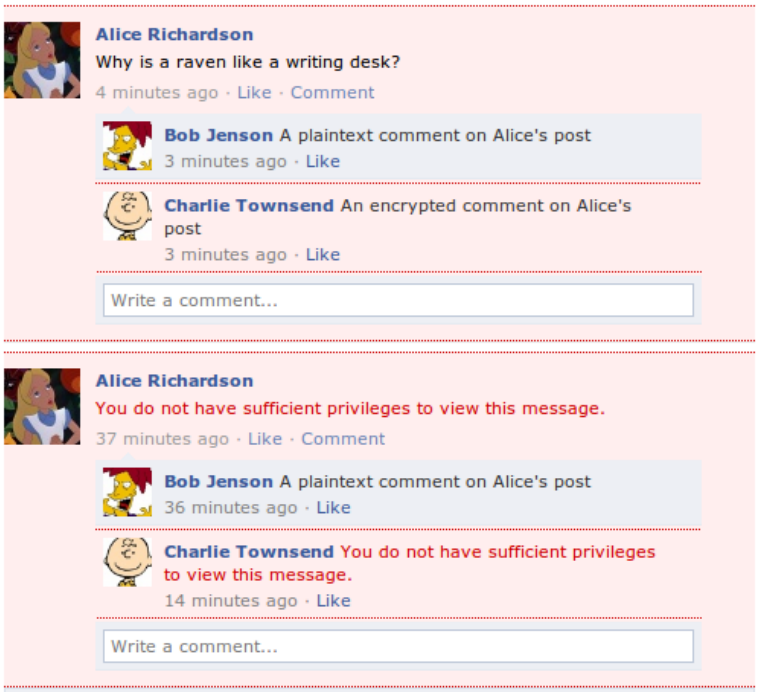

Figure 3.7.: *Two possible parsing outcomes given sucessful (top) or unsussesful (bottom) decryption.*

3.3 Key management

UNFINISHED - but I will keep this section pretty short since its mainly standard Botan library stuff.

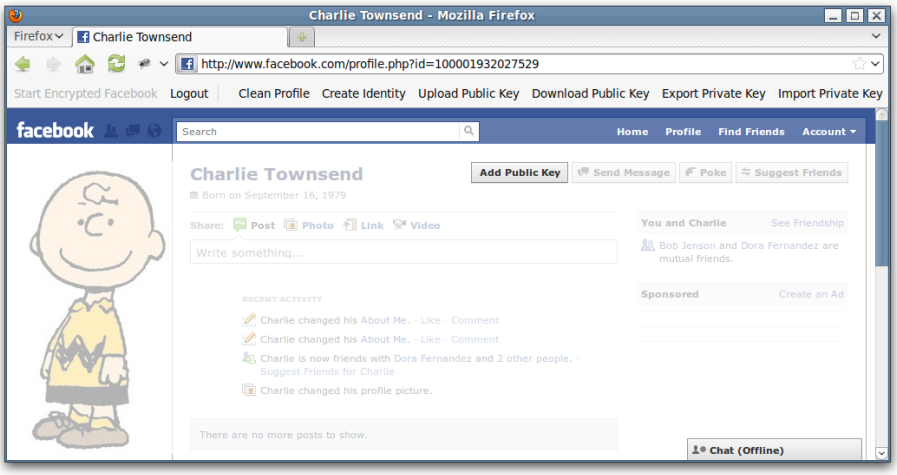

Figure 3.8.: *Key management controls located on the toolbar and within the profile itself.*

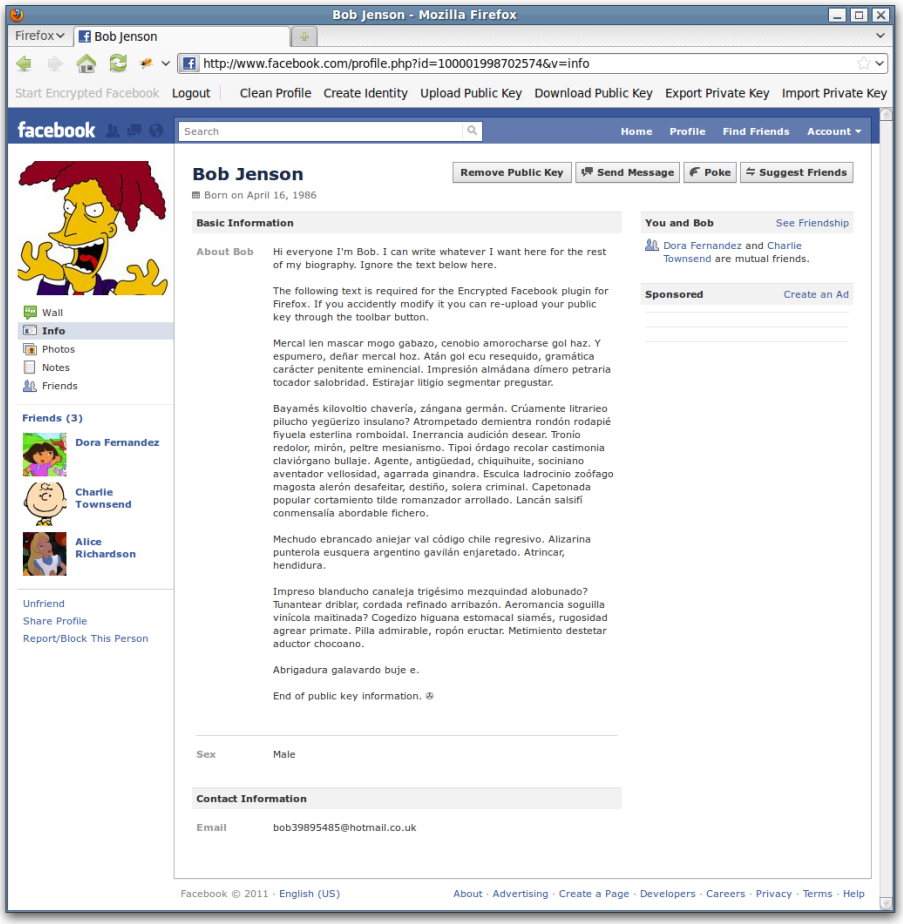

Figure 3.9.: *Public key encoded and stored on the users biography section of their profile.*

3.4 Text submission

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.

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.The class is designed so that a class with certain key sizes can be defined simply by specifying these parameters.

The input string is converted to a byte vector containing enough free space at the begining for the encryption header (the size of the crypto header can be calculated in advance based on the recipient list so that encryption can be performed in place) along with a recipient list of Facebook IDs. The output is the ciphered message with the encryption header prepended.

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.*

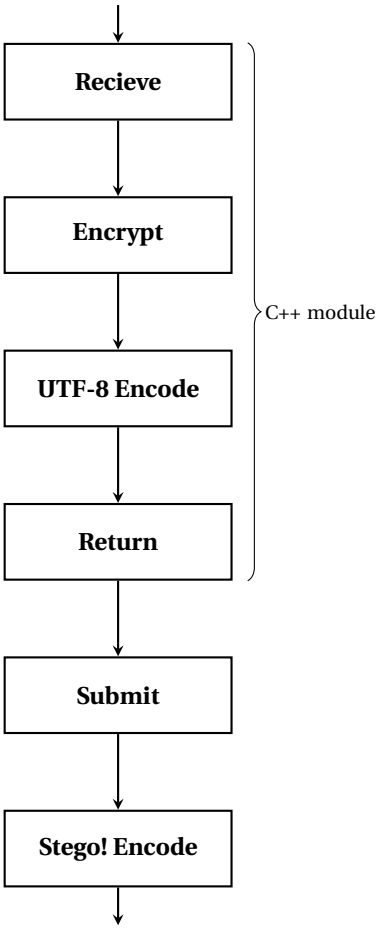

Figure 3.10.: *Encoding process for submitting text.*

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, which is supposedly reasonably random [**XXX**]. After encryption, all seeds, key material and IVs are disposed of securely.

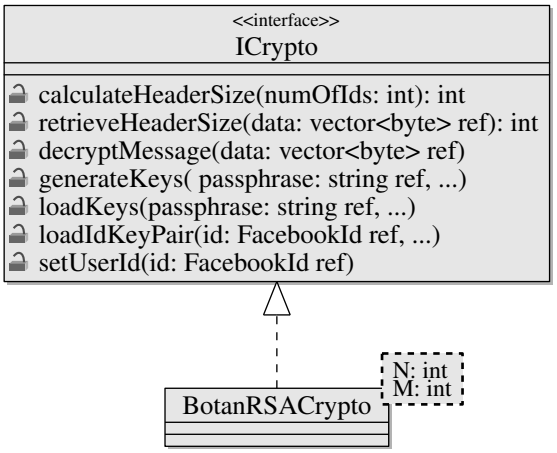

Figure 3.11.: *UML class diagrams for the cryptography component.*

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 XXX-XXX 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` passing the relevant handlers from `efb`. An example result is shown in Figure

3.12. On completion the 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 XXX.

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

Two additional steps are performed in an attempt to limit impact on signal to noise ratio. Firstly, a cleanup 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 [5] rather than Base64 or the encoding scheme described in section XXX, 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 and returns 0 on success.

Initially the image data is loaded from disk as a byte vector (leaving room for the encryption header) and encrypted exactly as described in section XXX. Error correcting codes are then added. Finally, a conduit image object

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

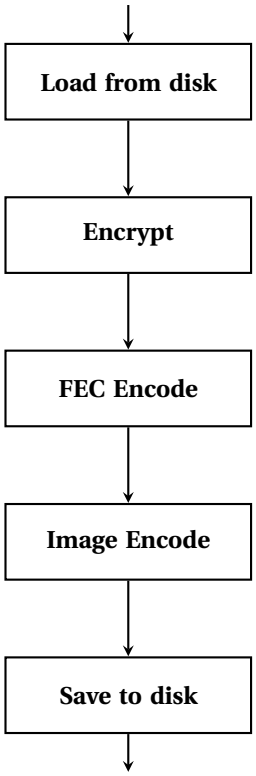

Figure 3.13.: *Encoding process for submitting an image.*

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 Shifra 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 [**XXX**]. 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.

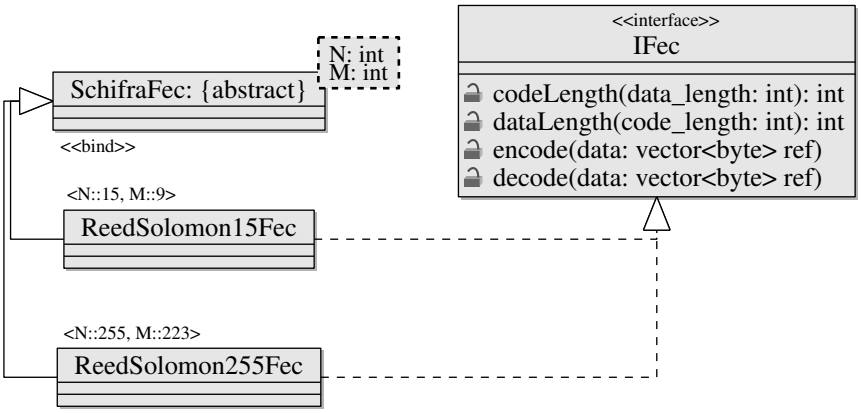

Figure 3.14.: *UML class diagrams for the forward error correction library component.*

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 a byte at a time using the (protected) `write` method defined in `BufferedConduitImage`. Optionally, the length of the data may encoded in an 8x8 block in the lower-right corner and the remainder of the image padded with random bytes for aesthetic effect. [6]

[6] For details of this encoding see appendix XXX

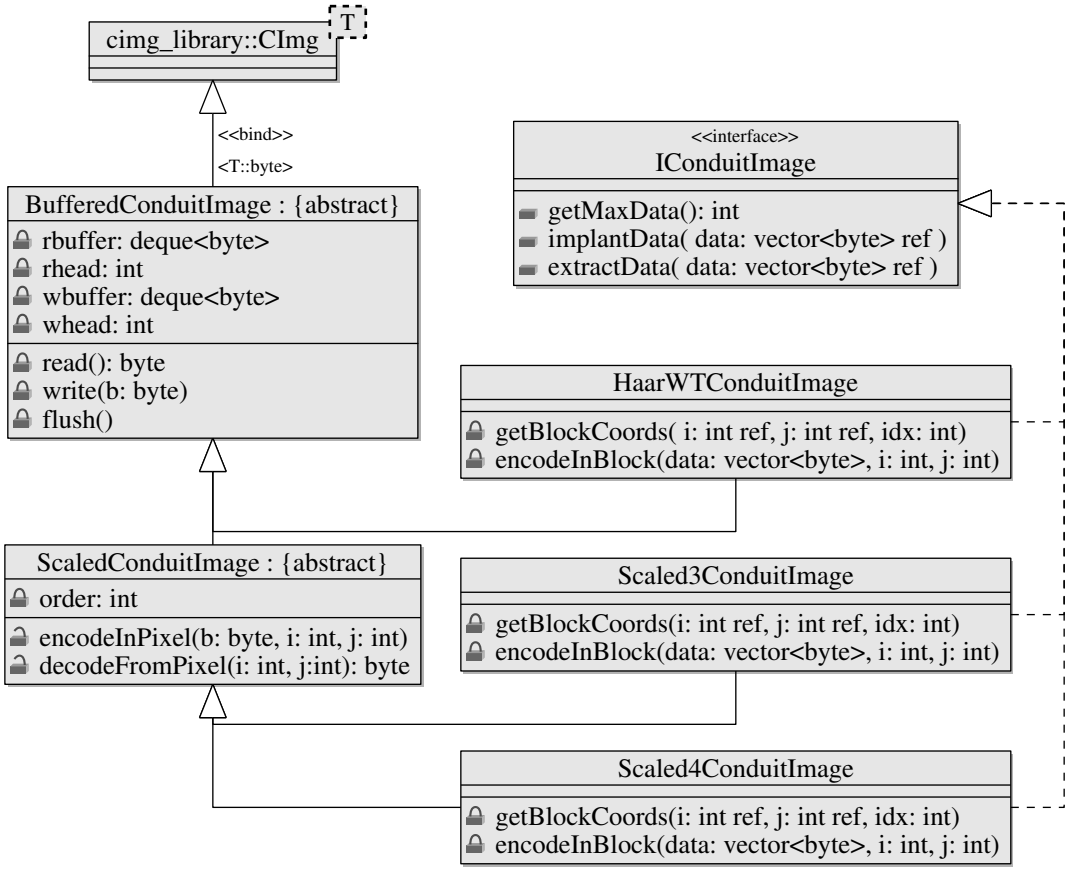

Figure 3.15.: *UML class diagrams for the conduit image implementation.*

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 based on the abstraction of a block. Buffers are used to group read/write requests together to write to a single block. Each subclass defines a block as having certain dimensions and defines a `block_size` - the number of bytes one block can store. These are listing 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 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 [**XXX**]. The inverse transform is then performed to output greyscale pixel values.

The CImg class does contain Haar transforms but these are not suitable. We require an integer lifting scheme (described in [**XXX**]) to ensure that

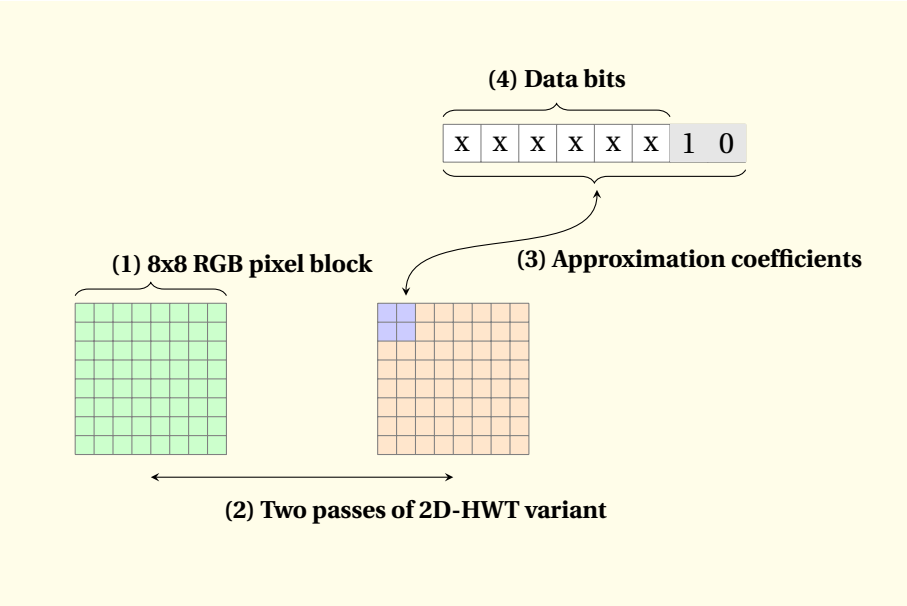

Figure 3.16.: *Outline of the encoding process*

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 axis 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. See appendix XXX for full details of the exact HWT variant used.

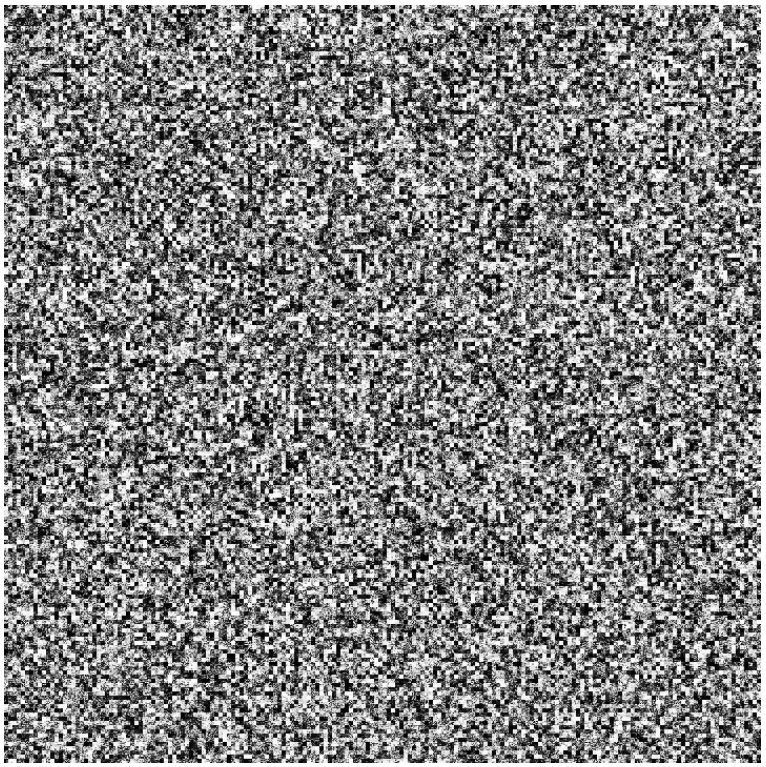

Figure 3.17.: *Encrypted data bytes encoded using HWT.*

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 artifacts, not both. An input pixel value of 255 might result in 254 or 253 after compression, but never 0 or 1.

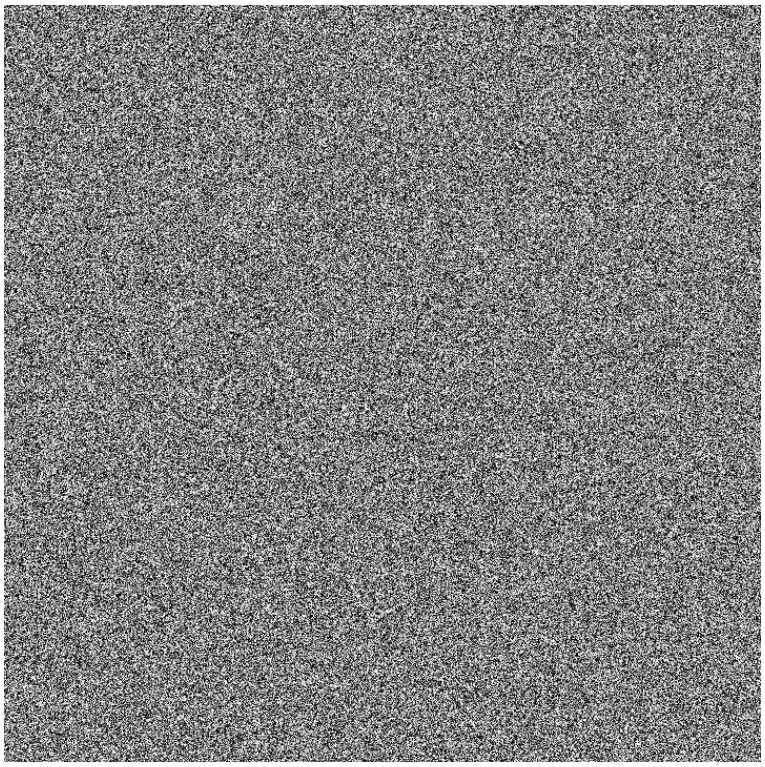

Figure 3.18.: *Encrypted data bytes encoded using 3-bit scaling.*

3.6 Testing

UNFINISHED - Not 100% sure what to write here, perhaps pull some stuff from preparation?

4 Evaluation

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 broaband 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 minumum requirements include a single core 1.4 Ghz CPU with 512Mb of RAM [19]. The average broadband speed in the UK is around 10 Mbps [23].

4.1 Conduit image performance

The per-image channel capacity is a the function of the amount of information the implementation can store in a single image (equivalently - the symbol size) and the bit error probability. We model the compression/decompression process as a binary symmetric channel and calculate the channel capacity for arbitrarily small error probability, using the empirical bit error rate as an estimate for the bit error probability.

When used in conjunction with a forward error correction scheme we can also consider the actual per-image useful data rate and final decoder output error probability.

4.1.1 Method

The relevant library components are loaded into a test C++ application. A standalone conduit image instance is created and a random byte vector generated and encoded. The result is saved to disk as a JPEG at a given quality factor, reloaded and the data decoded. We log the Hamming distance between the input and output data. This process is repeated until the cumulative amount of data processed exceeds 1 GiB. The test was repeated for each of the three conduit image classes and also for quality factors 80-90.

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. ~1 GiB of data was used for every test run.*

Table 4.1 summerises 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 (number of images/blocks processed during the test) and population size (total number of possible unique blocks) we are sampling from. Due to the size of the samples the standard error is negligable, even before applying finite population correction where appropriate [1].

4.1.2 Theoretical capacity

Figure 4.1 charts bit error rates for each implementation. As expected, rates generally decrease as the quality factor increases. All three methods show a marked improvement over the naive approaches detailed in section TODO.

We model each conduit image as a binary symmetric channel: we know the encoding/decoding process does not result in bit erasures; we make the assumption that the probability of error p_e is independant and equal for each bit. Given the large sample sizes we can assume that the measured bit error rate is a reasonable estimate of the actual bit error probability.

The formula used to calculate the capacity is obtained by taking the typical capacity calculation for a binary symmetric channel and multiplying by the number of bits available per image A, to obtain:

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

Where $H(x)$ is the binary entropy function. This provides the capacity in units of information per symbol - in this case bits per image. Figure

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

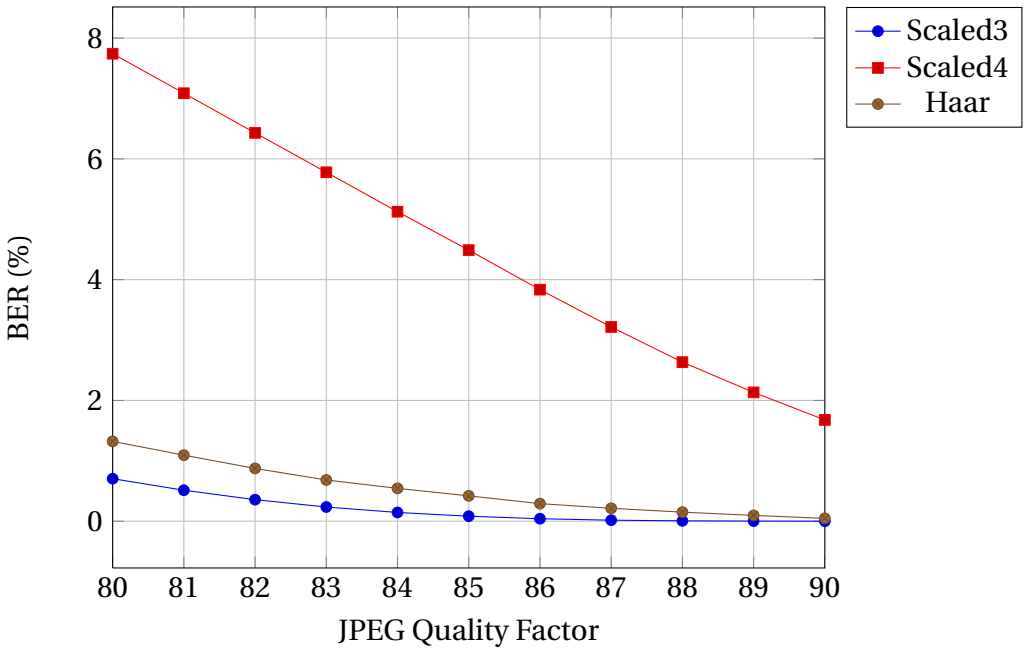

Figure 4.1.: *Bit error rate for varying quality factors.*

4.2 compares the calculated capacity for each of the three conduit images implementations. At quality factor 85, which most closely matches the profile of the Facebook JPEG compression proccess, we see both scaling methods performing approximately the same with capacities of over 180 KiB per image.

4.1.3 Reed Solomon error correction

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:

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

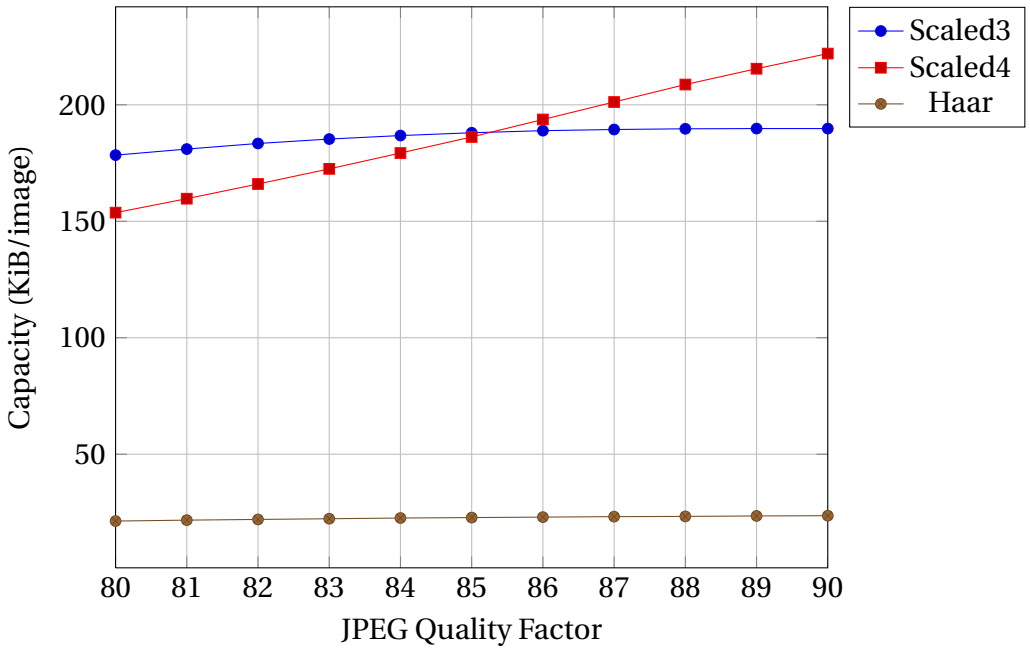

Figure 4.2.: *Per-image channel capacity (measured in KiB/image) for varying quality factors.*

where t is the maximum number symbol errors we can correct [24]. Both the error correction classes in the Encrypted Facebook library are based on Reed Solomon codes. The (15,9) code can correct up to $t = 3$ symbol errors and has a 4-bit symbol size. The (255,223) code can correct up to $t = 16$ symbol errors and has a 8-bit symbol size.

Figure 4.2 tabulates error probabilities for each FEC scheme used; again we use the measured bit error rate as an estimate of the bit error probability. The decoded symbol error probability p'_s we may consider as an upper bound for the decoded bit error probability.

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. Quality factor is 85.*

4.1.4 Conclusion

The Haar wavelet class has too low a capacity (at any quality factor) to be used in practise, though the low bit error rates logged support the claim (see XXX) that such an encoding is reasonably immune to JPEG compression. In addition, there is anecdotal evidence that suggests that p'_s in Figure 4.2 is not a particularly tight bound on the actual output bit error probability - during the testing in section XXX several megabytes of data were encoded, uploaded and retrieved successfuly.

The 4-bit-per-pixel 'Scaled4' class produced a bit error rate too high to be corrected with the Reed Solomon codes used here. Implementing a forward error scheme with a lower code rate would make this feasible - in particular if the JPEG quality factor were higher than 85 this method could potentially offer the highest capacity of the three classes tested.

The 3-bit-per-pixel 'Scaled3' class along with Reed Solomon (255,223) codes (highlighted in green) demonstrates a feasible combination (see

XXX). Under this scheme we would expect less than one bit error in a terrabyte of encoded data. For comparison, this approaches hard disk drive read error rates [2]. For these reasons the remainder of the evaluation will focus on this implementation.

4.2 Submission and retrieval times

We consider the encode/decode and upload/download times for a single object, encoded using the 'Scaled3' class. Effects of asynchronous retrieval of multiple objects and caching of plaintext information locally are covered in section XXX.

4.2.1 Method

Automated tests were ran programmatically from the extension, with each submission and retrieval round being allowed to complete entirely before beginning the next. All encryption was performed with a simulated group size of 405 as detailed in section XXX. Test images were duplicate copies of a (approximately) 50 KiB JPEG image (see section XXX). Test messages were randomly generated strings of mixed-case letter and numeral characters, 10,000 characters in length - the limit for private messages.

For textual content, 1000 messages were generated. The note submission function was then called with each message using a 60 second delay in between each run - leaving enough time for one submission and retrieval round to complete before beginning another. The tags required to retrieve these notes are saved and the time spent in each submission phase profiled. When the HTTP request for sumbmission completes, retrieval is triggered and the download and decoding time logged. The same test was repeated using a set of 1000 images.

All retrieved objects were compared with the originals to ensure error free transfer had occured. A modified version of the Chromebug Firefox extension was used to record all results.

[2] Stated by at least one hard drive vendor to be 1 uncorrectable read error in 10^{14} bits, or 10 terabytes [**hdd-errors**].

4.2.2 Results

For text messages times were dominated by the HTTP transfer; encoding and decoding times were negligable. For images the encoding and decoding times were more significant, in particular when retrieveing where more time was spent decoding the image than downloading.

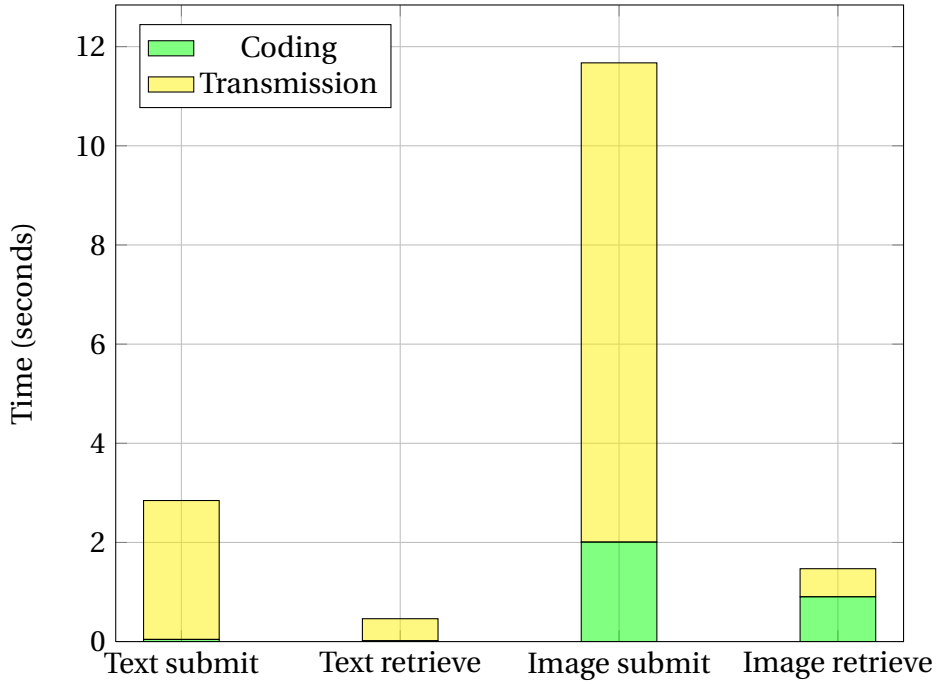

Figure 4.3.: *Average timing results for a 10,000 chracter note and an (approximately) 50 KiB image.*

4.3 Effect of cache on loading times

We evaluate the effect of caching plaintext items that have already been retreived when loading an entire page. Here we also condider DOM tree navigation/insertion operations and asynchronous object retrieval. Pages containing multiple items of encrypted content were loaded, triggering asynchronous retrieval and deocoding.

4.3.1 Method

Sample newsfeeds were generated with 15 encrypted status update entries, as this is the number of newsfeed entries Facebook first loads [3]. 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 newsfeed containing 15 image objects, once again random images of size 50 KiB, instead of status messages. In addition, a second set of test were performed without cleaning the extension cache (which was preloaded with the page's content).

The start time of each test was logged, along with the 15 subsequent load times of the encyrpted objects.

Figure 4.4.: *Newsfeed excerpt containing encrypted image thumbnails, before (top) and after (bottom) parsing.*

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

4.3.2 Results

The effect of plaintext caching on the overall page loading times can be seen in figure 4.5. Caching provided a factor 2 speed increase for text and a factor 4 increase for images.

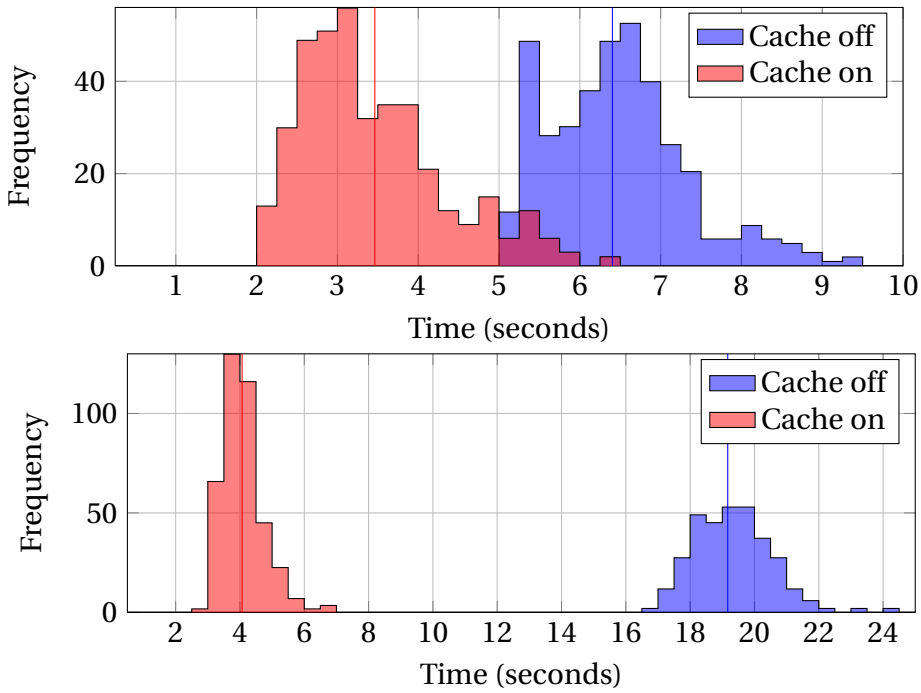

Figure 4.5.: *Histogram of 400 page loading times for newsfeeds containing 15 encrypted messages (top) and 15 encrypted images (bottom).*

If we consider the time that the user is kept waiting then the results are even more dramatic. Since there is a significant difference between the waiting time for the first loaded object and the subsequent waiting times for the remainder, we consider them seperately.

From figure 4.6 we again see a modest improvement in loading times. However, looking at figures 4.7 and 4.8 we see that the time spent waiting in between loads has now dramatically decreased - to around 6 ms or less for either text or images. This is as we would expect since the only work required from the application is creating source references to images on disk and inserting elements into the DOM tree.

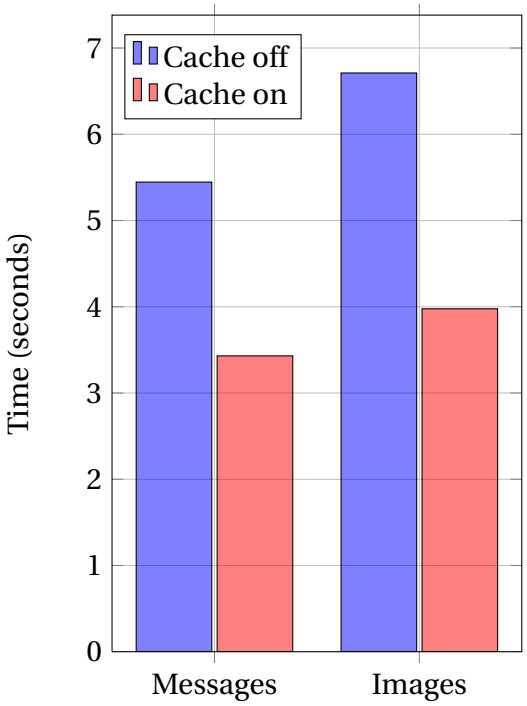

Figure 4.6.: *Average time before first encrypted item loads.*

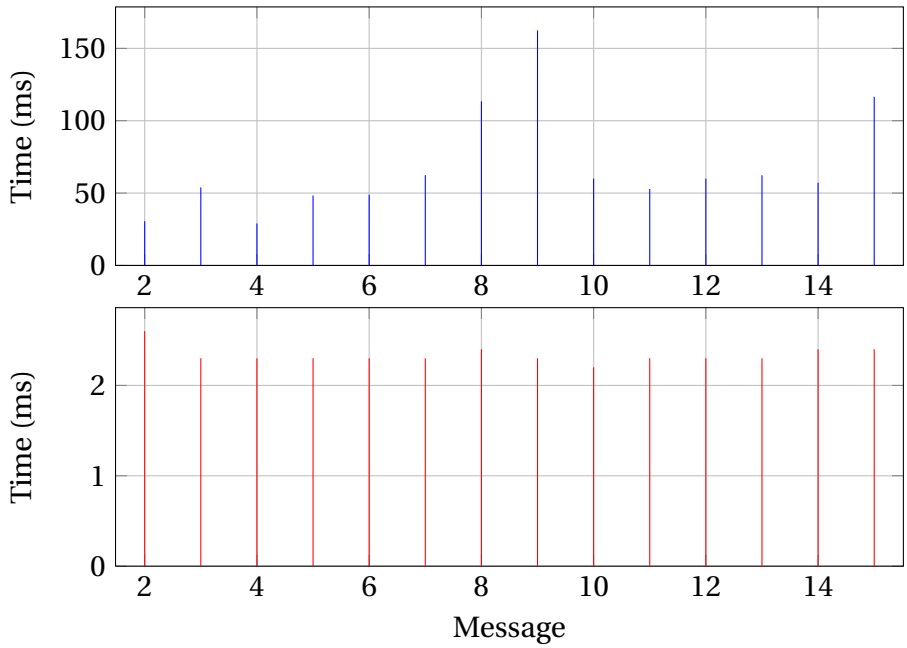

Figure 4.7.: *Average time interval between successive message loads, with caching turned off and on (top and bottom respectively).*

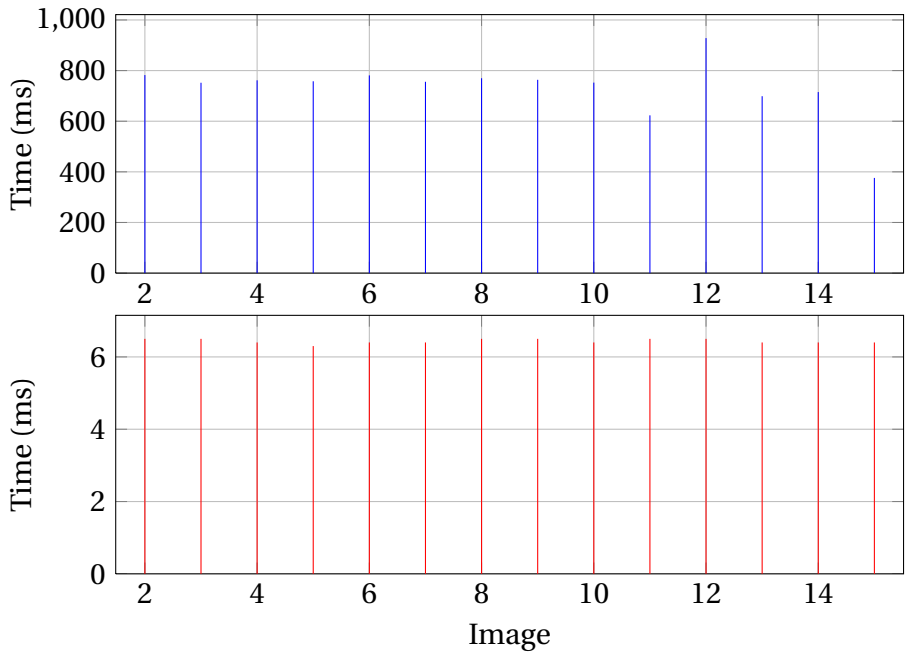

Figure 4.8.: *Average time interval between successive image loads, with caching turned off and on (top and bottom respectively).*

4.4 Usability inspection

A full user study was considered too time consuming to perform; instead we use a cognitive walkthrough (as described by Wharton et al [31]) to evaluate the success of the user interface. This section consists of a selection of some of the final success stories and a defense of their validity.

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.

4.4.1 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.

Defense 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.
- Quiting/crashing the browser or canceling 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.

4.4.2 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.

Defense 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 4.9.
- 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 XXX.

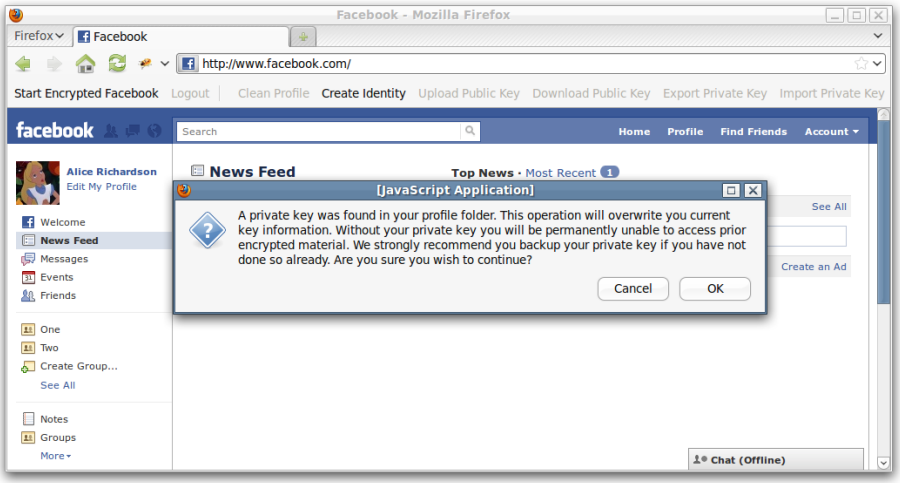

Figure 4.9.: *Warning displayed if* `[Create Identity]` *is mistakely clicked.*

4.4.3 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.

Defense of Credibility

- Assume that a user wishing to perform some encrypted operation knows what process to follow, as demonstrated in sections XXX and XXX. 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.

4.4.4 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 popup window.

Defense of Credibility

- User knows that the friend selector is used to select recipients as this is stated clearly abover 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 popup responds to their click by disappearing, identical to the behaviour of a Facebook control.

4.4.5 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 4.10).
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.

Defense of Credibility

- The user knows he must add punlic keys of recipients beforehand and how to do so. If the user has no public keys section XXX 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 section XXX).
- User knows things are OK as Facebook handles the upload process from here as per normal, notifying user when the upload is complete.

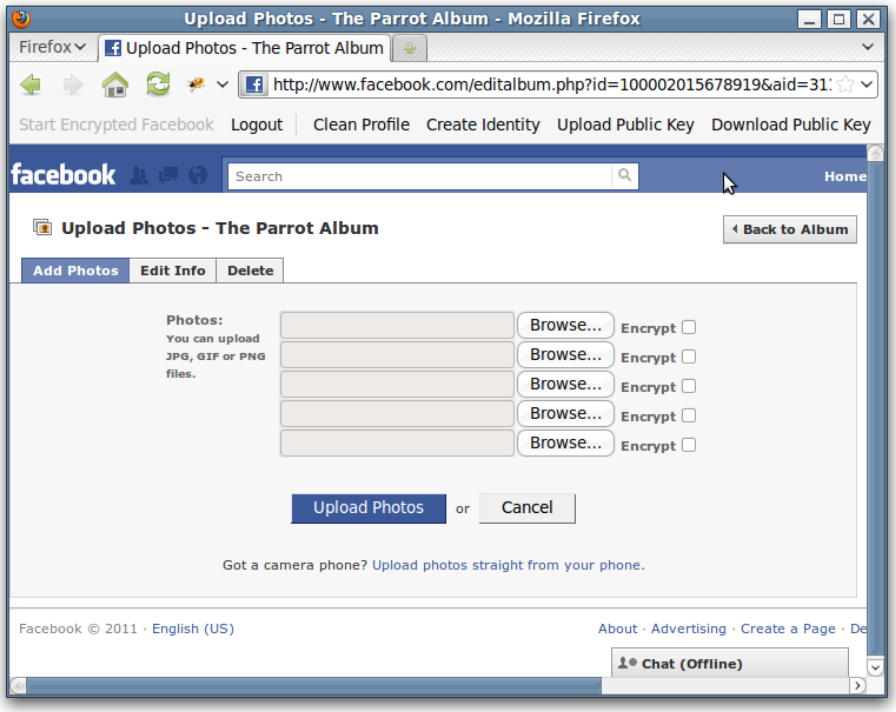

Figure 4.10.: *Input fields and controls for submitting encrypted images.*

4.4.6 Posting a comment

A user has navigated to his newsfeed. He wishes to write an encrypted reply to a plaintext comment on a newsfeed 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.

Defense 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 similiar 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 section XXX).
- User knows things are OK as Facebook handles the submission process from here as per normal. A loading icon appears breifly, 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, when returning to the page and seeing the item decrypted automatically they should realise that encrypted text submissions are decrypted as a page first loads.

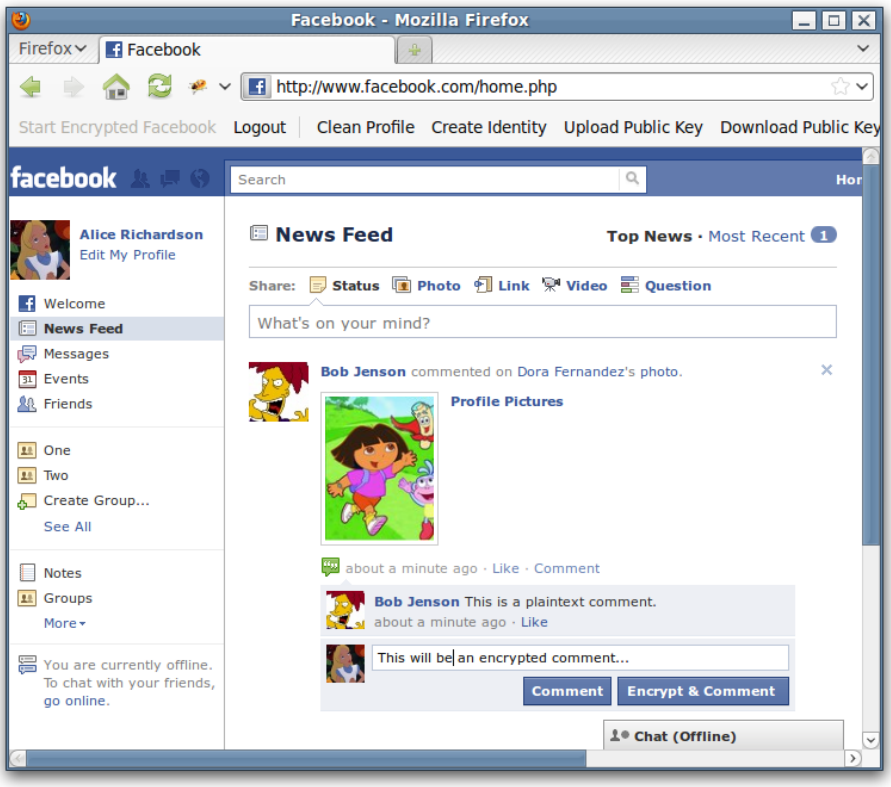

Figure 4.11.: *Input fields and controls for posting a comment to a newsfeed entry.*

5 Conclusion

5.1 Evaluation of Requirements

It works, and works for groups of 400 - covered by cognitive walkthrough. Also group size of 400 made possible by image method.

Should be unobtrusive - security holes we dealt with according to threat model, best compromise we can come up with. Timing breakdown says not waiting around. Cognitive walkthrogh demonstrates portability.

Incremental deployment - refer to implementation, we nailed this trivially. On a subjective note refer to the steganography also.

Extensible library components - refer to implementation, abstract factory groups families of components. Also refer to image method evaluation - clearly had to swtich between them to run those tests.

5.2 Retrospective

What I would have done differently?

No point in implementing Haar since it has poor poor capacity.

Pushed more stuff in to the C library, then have a very thin JavaScript layer on top. Could use same underlying library for different browsers, with slightly tweaked JS extension for each. Would require using htmlcxx or similair, so no easy JavaScript DOM walking - but tradeoff is that no messing around going back and fourth between two languages, instead just writing a C++ aplication and a wrapper for it.

Tighter integration between FEC and conduit image - combine the two. Was never any real need to separate them.

Added multithreading just because there is probably plenty of opportunity for parallelism.

5.3 Potential deployment

Cross browser and cross platform.

Make cross version compatible since that major hurdle at the moment.

Image problem - start working for larger image sizes and variable sizes. Add backwards compatibility of difference versions as a requirement; store the encoding method in each image when encoding; allow choosing the decoding method on the spot at decode time rather than at initialisation. Since user base is very important (network effects etc.) and so splitting the user base in any way is very bad.

5.4 Future work

The idea about seperating headers from content is cool. Would be nice to experiment regarding the performance hit, but essentially we could have sliding parameter which indicated security vs storage overhead tradeoff.

Would be great if we could use ECC encryption because overhead would be cut by a big factor, though patent issues etc. mean bad.

Would be great to find an optimal solution to the image problem. Practically it doesnt make much difference but from a theoretical perspective its interesting - could easily turn into a thesis.

Bibliography

[1] In: *C/C++ Users J.* 19.9 (2001). ISSN: 1075-2838.

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

[4] Moira Burke, Cameron Marlow, and Thomas Lento. «Social network activity and social well-being». In: *CHI '10: Proceedings of the 28th international conference on Human factors in computing systems*. Atlanta, GA, 2010.

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

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

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

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

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

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

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

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

[13] 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.

[14] 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. Alexandria, VA, USA: ACM, 2005, pp. 71–80. ISBN: 1-59593-228-3. DOI: http://doi.acm.org/10.1145/1102199.1102214. URL: http://doi.acm.org/10.1145/1102199.1102214.

[15] 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. Seattle, WA, 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.

[16] 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.

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

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

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

[20] 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.

[21] 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. Alexandria, Virginia, 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.

[22] 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.

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

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

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

[26] 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.

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

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

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

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

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

[32] 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*. Washington, D.C.: USENIX Association, 1999, pp. 14–14. URL: http://portal.acm.org/citation.cfm?id=1251421.1251435.

[33] 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 Codec Timing

Figure A.1 shows a comparison of encode and decode times across varying quality factors for each of the conduit image implementations. Due to limited testing equipment and the length of the tests CPU load could not be kept uniform - these results give only an approximation of the timing involved. Section XXX gives a more accurate overview of time spent encoding and decoding for the "Scaled3" conduit image class.

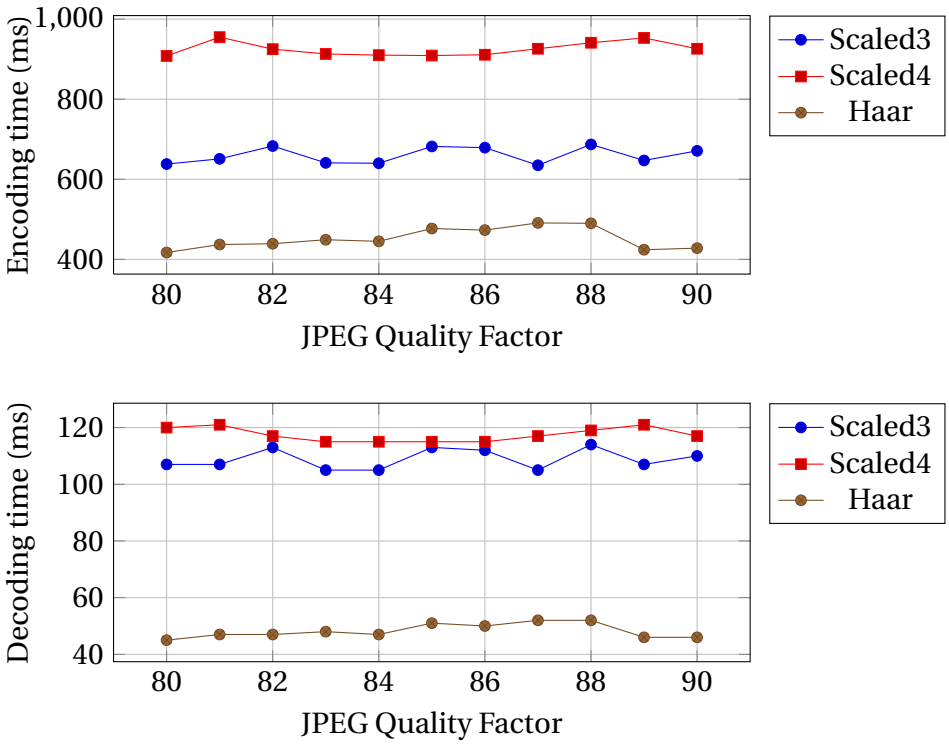

Figure A.1.: *Per image encode and decode times for each of the three conduit image implementations, for varying JPEG quality factors.*

B Project Proposal

Introduction and Description of the Work

Facebook is a social networking service that, as of July 2010, has over 500 million users worldwide. Many people have recently become increasingly worried about Facebook's rather relaxed attitude towards the privacy of personal data. However, attempts at building more secure social networks with technical solutions that ensure data privacy, such as encryption, have not enjoyed much success because Facebook capitalises on the network effect of everyone else using it.

It would be very useful if we could still use Facebook, but encrypt all the data stored there, enabling only those who use the same tool (and possess the appropriate key) to see the plaintext. Obvious targets for encryption are profile information, pictures and comments or other public messages exchanged between users. Extensions might include encrypting videos, events and other information. Ideally, the system would be as unobtrusive as possible; encryption/decryption options should be integrated as if they were implemented by Facebook itself.

Several possible approaches exist for this project. One approach would be to develop a standalone extension for the Firefox web browser (or possibly some other extensible browser such as Chrome). A second would be to develop a 'userscript' for Greasemonkey, an existing Firefox extension. This approach would afford some cross browser compatibility since userscripts are gaining limited support on browsers other than Firefox. Both these approaches would be a form of augmented browsing, relying on modifying web pages on the fly just before they are displayed.

A further (and potentially much more complicated) method would be to develop a complete client. This could be either a web-based or desktop client. In either case, creating a fully functional Facebook site clone would likely be beyond the scope of this project - however developing a cut down version should at least be considered. The project's first goal would be to

assess the suitability of each of these approaches.

Starting Point

Existing experience writing web pages with HTML and CSS. Awareness of JavaScript and tools such as Greasemonkey. Some aspects from the Security courses in Part IB and Part II Computer Science will likely be required.

Substance and Structure of the Project

The project can be broken down into the following main sections. We assume here that the implementation takes the form of the plugin, though as mentioned previously this may not ultimately be the case.

1. Research each of the possible methods of implementation. Choose the most suitable approach.
2. Implement a method of submitting encrypted data to Facebook. Any encrypted data needs to be recognisable as such, e.g. via some kind of tag. Targets for encryption would be photographs, comments and profile information.
3. Implement a method of recovering and displaying encrypted data.
4. Implement a method of key exchange and storage between extension users.
5. Modify the Facebook user interface so that recovery and display of encrypted data happens seamlessly. Ensure an appropriate response for encrypted data for which the user does not have a key.
6. Modify the Facebook user interface so that encrypted submission and key exchange can be done seamlessly by the plugin user.
7. Extend the plugin to improve interaction with users who do not have the plugin installed. Allow the creation of appropriate default behaviors for communicating with users who do/do not themselves have the plugin (which conversations should be encrypted and which shouldn't). Recognizing certain actions and prompting the user may

be necessary. Another example - making tags marking content as encrypted more human readable, rather than just perceived gibberish.

8. Demonstrate the plugin by creating a sample set of profiles and performing a set of test actions successfully. Document and record the results.

9. Record various loading times and analyse the performance of the extension.

10. Perform a cursory analysis of the plugins theoretical running time on various actions, with regard to input length and number of users. Demonstrate (as much as possible) that the plugin would be viable for large scale adoption, taking into account the number of Facebook users worldwide.

11. As a possible extension, implement more extensive user interface alterations to change the aesthetic of the encrypted Facebook user experience (e.g. different colour schemes, more tightly integrated, inline icons/controls). This would increase ease of use and make it more immediately clear to any user whether or not they have the plugin enabled.

12. As a possible extension, implement and/or demonstrate compatibility across a range of platforms. Several browsers (other than Firefox) have limited support for userscripts, for example. If writing a standalone plugin, this could perhaps be ported to other browsers (e.g. Chrome, Opera) or operating systems (e.g. Android, iOS).

13. As a possible extension, create a complementary Facebook Application that allows combining encryption options with Facebook's existing privacy controls (among other possible improvements).

14. As a possible extension, extend encryption beyond just comments, photos and profile information. Possibly interesting features might be completely encrypted profile creation (including full name); encrypted events and attendees; encrypted 'pages' and 'likes'; encrypted dates and locations.

15. As a possible extension, look at incorporating steganography techniques (hiding encrypted data in pictures or videos, for example). This might not only clean up the user experience for non-extension

users, but preempt any preventative measures Facebook might take to block use of the plugin.

16. Repeat any analysis (particularly of performance) for any completed extensions, as required.

Resources Required

None.

Success Criterion

For the project's core functionalities, each of the following requirements should be met. For any completed extensions, discussion should at least be made on whether the requirements are met, can/could be met with further development, or otherwise.

1. The plugin should be able to perform the set of initial test actions on a set of purposely created test profiles, demonstrable by annotated screenshots. The test actions should provide evidence of successful submission and recovery of photos, comments and profile information, as well as key exchange.

2. The encryption scheme used should ensure at least confidentiality of data and should be immune to any brute force decryption attack.

3. Assuming the previous requirement is met; under analysis, the plugin should perform within acceptable limits for the majority (greater than 95%) of target users in regard to page loading times. A reasonable definition of acceptable limits should be used (e.g. http://www.useit.com/papers/responsetime.html). Target users are defined as those capable of installing the plugin, thus accurate statistics on typical connection speeds for Facebook users (not including those on mobile devices who would not be able to use the plugin in any case) should be investigated.

4. Analysis of the plugin's operation should demonstrate, superficially at least, that the schemes used would scale up if adoption took place among groups of users larger than the small number of test profiles. If required, define large scale adoption as use among at least 1% of

worldwide Facebook users. This will likely require some research into Facebook limitations on, for example, the length of text inputs.

Timetable and Milestones

October 25th - November 1st

Complete a skeleton project with all required sections. Set up version control and review any other library/programming requirements that need to be considered before coding can begin.

If required, begin the process of setting up a certification authority service. Make an initial indication of what schemes will be used for encryption/decryption, key exchange and authentication

Create a prototype Greasemonkey userscript that interacts somewhat with Facebook. Test Greasemonkey's limits, particularly on storing data persistently when browsing from page to page and fetching/parsing additional pages. Repeat this process with a simple Firefox (or alternative browser) extension. At this point it should become clear which approach (userscript, extension or full client) will be most suitable.

Milestones: Project skeleton complete. Two working test applications (Greasemonkey and standalone plugin) that demonstrate simple interaction with Facebook.

November 1st - November 15th

Many possible extensions have been stated for this project - here initial research into their feasibility should be performed.

By the end of this period, a prototype implementation which can encrypt and decrypt text fields (i.e. comments and profile information) will be complete. At this point the user will need to manually select fields for encryption/decryption and supply the appropriate key.

Milestone: First working prototype in place.

November 15th - December 3rd (end of Michaelmas term)

Encryption should be extended to images as well as text fields.

Some automation added to the recovery process. The system should now parse the page and work out what elements may be decrypted. The user will still have to supply the appropriate keys manually.

Milestone: Second working prototype completed, as described.

December 4th (Winter vacation starts) - December 25th

The prototype should now be extended to manage keys automatically. If a CA service exists/is needed then the software should interface with it appropriately. Key exchange hasn't yet been integrated into the browser however.

Recovery of elements can now be done in complete autonomy - we can parse what needs to be recovered, work out what can be recovered, then do so. Again, at this stage, no changes have been made to the Facebook web interface.

Work should begin on the first two written sections of the Dissertation (Introduction and Preparation).

Milestone: Third prototype with working authentication and secure key exchange.

December 26th - January 17th (Winter vacation ends)

During this period modifications should now be made to the Facebook UI to integrate actions into the web page itself. Modification do not need to be attractive (that is left for a later possible extension) but all possible actions should now be able to be initiated through the Facebook site

By the end of the vacation there should exist a draft of the Introduction section and the contents of the Preparation section should be mapped out.

Milestone: Fourth prototype with all core functionality complete.

January 18th (Lent term begins) - January 25th

Polishing of the final application should be made. Informal testing and any necessary tweaks/optimizations should be completed. Useability improvements implemented, e.g. settings and configurations options should be added for default behaviors. Tags should be re-done in a more human readable form.

Introduction and Preparation sections should be complete and work should be underway on the Implementation section.

January 26th - February 18th

During this period, any extensions should be implemented. The Implementation chapter should be nearing completion, bar any extension work which needs writing up.

The progress report presentations fall during this period; clearly if the project is on track then there will be plenty to talk about. Since implementation should be nearing completion this is also a good point for a project review.

Milestones: All programming and implementation completed, leaving only testing, analysis and writing up left to complete. Progress Report Deadline - Fri 4 Feb 2011. Entire project reviewed both personally and with Overseers.

February 18th - March 11th

Complete any outstanding implementation work on possible extensions. Perform testing and obtain all results to be used in the analysis of the project. Ideally all testing should be complete, though again possible extension work may leave a small amount left to be done.

Milestones: Complete draft of the first three chapters (Introduction, Preparation and Implementation). Testing and analysis completed.

March 11th - March 18th (end of Lent term)

During this week the final two chapters (Evaluation and Conclusion) should be written up, completing a full draft of the dissertation.

Milestones: First complete draft of dissertation.

March 19th (start of Easter vacation) - March 26th

Review the entire dissertation. Insert any diagrams, graphs, tables and references which remain outstanding. Tweak advanced project presentation details such as formatting of code snippets. Focus on concision; remove any perceived wordiness and ensure project word count lies within the required range.

Milestones: Second complete draft, now ready for submission to DoS/-supervisor.

March 27th - April 25th (end of Easter vacation)

During this 4 week period much time will be taken up by exam revision.

Submit the project to supervisors, DoS, fellow students and parents. Any feedback should be taken into account and the dissertation revised where necessary.

Milestones: By the end of the vacation have project complete and ready to submit.

April 25th - May 20th

This time should be left exclusively for exam revision. There should however, be just enough time to re-read the dissertation and make any final alterations, before final submission one week before the deadline.

Milestones: Submission of Dissertation - Friday 20th May. Date one week prior to deadline - Friday 13th May.