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 [6]. Encrypted Facebook is a Firefox extension which permits the use of broadcast encryption when sharing certain content via the Facebook platform.

1.1 Background

Since its inception, Facebook has come under criticism in relation to online privacy [1]. 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 [4]. 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 [2] [**fb-gov**] [7] [11].

Encrypting content ensures its secrecy in any eventuality, provided the encryption key itself is kept safe. Tools exist to perform encryption manually, some are even partly integrated into the Facebook user interface [**firegpg**]. However, studies have suggested that security software which requires manual management of cryptographic keys is not usable enough to provide effective security for most users, putting them beyond the reach of the masses [**johhny**]. In addition, interaction over social networks is

[1] Facebook was recently forced to update its privacy controls due to growing pressure from the public and media [8].

[2] As was the case in 2010 when personally identifiable Facebook user details were sold to data brokers by third party developers [3].

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 and **??**). Thus far all proposals have required the use of a third party in ways which are unlikely to be scaleable (see section xxx).

Attempts have been made to create new social networks which protect privacy by encrypting content and in some cases even decentralizing hosting of the entire platform [**pidder**] [**diaspora**]. 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 [10] [5].

- Protection against middleperson attacks (see section XXX).
- 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 XXX).
- Full cross-platform and/or cross-browser support (see section XXX).
- Designing and implementing a security policy which comprehensively covers all aspects of the project's functionality (see section XXX).

1.4 Existing work

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

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

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 profile 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 **??** into a set of design principles, drawing on existing work. We describe possible deployment strategies. We briefly review Firefox extension development, broadcast encryption schemes, the Facebook platform and other security related concerns. We also look at the specific problems assosciated with encrypting images and possible solutions. Finally, we consider an appropriate testing strategy and software development methodology and derive a concrete set of requirements.

2.1 Design principles

We describe the design principles of Encrypted Facebook; how they enforce the stated aims of privacy preservation, scaleability, usebility and incremental deployment and how they compare with existing approaches. Note that 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, as with NOYB [**noyb**]. Arguably, however, privacy is only poorly preserved due to problems of inference control [**ross**]. An example: many users will be easily identifiable simply from the photos they upload. Incremental deployment is also not possible since non-users will only ever see fictitious profiles [**facecloak**].

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 release.

2.1.2 Independency from third-party servers

In addition to encrypting content, flyByNight, FaceCloack and uProtect.it all opt to migrate content from the Facebook platform to an external third party database.

Encrypted Facebook is designed to avoid outsourcing content due to scaleability concerns. Storing and delivering encrypted content requires at least the resources needed for storing and delivering the cleartext. Facebook's monthly bandwidth overheads alone are in excess of two million dollars [**fb-costs**]. They are able to offer a free service by serving highly targetting advertising to members based on the structure of the social graph [**fb-ads**]. This revenue stream would be largely unavaible to any solution hosting a database of encrypted content. A subscription based service would also be unlikely to scale, since the majority of Facebook users would be reluctant to pay [**fb-pay**].

Third-party servers can also be employed for performing encryption and/or decryption, as with uProtect.it and flyByNight. Encrypted Facebook is designed to work without requiring a third-party for such computation, again since the resources required by the server would scale linearly in proportion to the amount of content exchanged.

Another use of third-party servers is to store and distribute cryptographic keys, perhaps as part of a public key infrastructure. Arguably issues of scale here are not so severe. In any case Encrypted Facebook does not rely on such a service due to either conflicts with other design principles, or in some cases due to a limited percieved benefit given the additional complexity of impementation.

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. This is the basis of uProtect.it [**uprotect**]. Key management can even be taken out of the users hands entirely, greatly improving usebility. However, confidentiality is only weakly assured: trust has simply been defered from Facebook to the third-party and many of the scenarious 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 to generate a key by hashing the password. Again, relying on the user to memorise a password rather than manage secret keys improves usebility. Unfortunately the entropy of user chosen passwords is far less than that of randomly generated keys [**password**].

Encrypted Facebook is designed to only ever generate and store secret keys on a user's device(s), trading usebility for better privacy protection.

2.1.4 Minimal use of OOB channels

Secure **OOB!** (**OOB!**) channels (such as encrypted email or face-to-face exchange) can be used to transmit update messages, keys or other information as part of an encryption scheme (see section XXX). 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 themselves and ensure that the email is sent securely (by using PGP, for example).

Encrypted Facebook is designed to limit the use of **OOB!** transmission due to usebility concerns regarding manual key managament. The exception is when installing a secret key across more than one device - since transporting a secret key by any other method would comprimise the principle of secret key security.

2.2 Broadcast encryption

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

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 authenticates with the server and is sent the cleartext.

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

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

None of these approaches are compatible with Encrypted Facebook's design principles. Instead we use a form of broadcast encryption.

2.2.2 Proposed scheme

Given a suitable assymetric encryption scheme P and a suitable symmetric scheme Q, we use the broadcast-encryption scheme defined by the triple of algorithms $(\text{SETUP}, \text{BROADCAST}, \text{DECRYPT})$ such that:

- The setup algorithm (SETUP) takes a user $u \in U$ and constructs that receiver's private kev $priv_u$ and public key pub_u using scheme P.

- The broadcast algorithm (BROADCAST) takes the list of priveledged 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 the decryption algorithm $\text{DECRYPT}(b, u, priv_u)$ 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, $u \notin R$.

2.3 Intercepting Facebook interactions

In order to encrypt content it must be intercepted before being submitted to Facebook. We describe the possible stages at which this may occur (Figure 2.1):

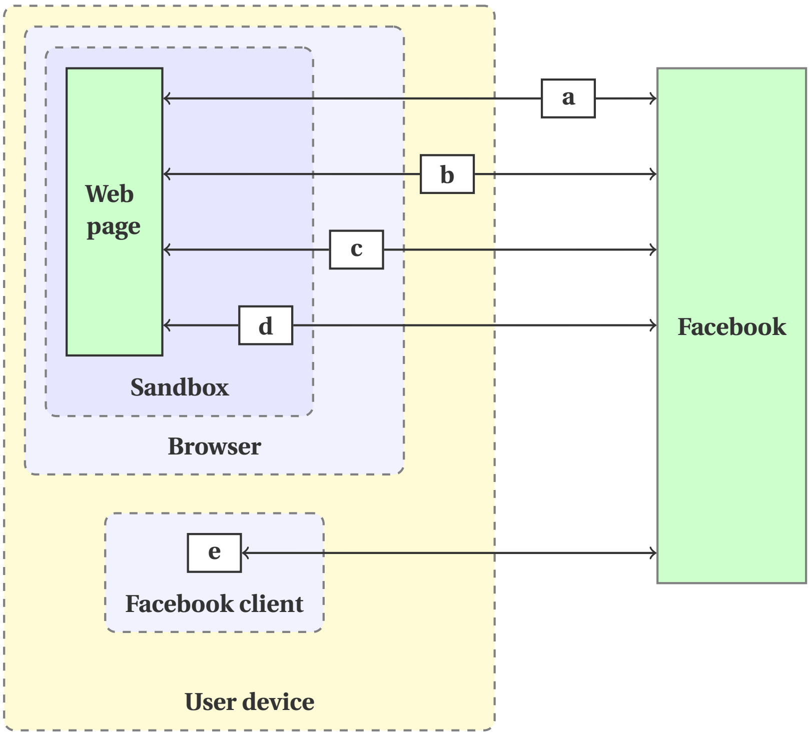

Figure 2.1: *Possible deployment strategys for intercepting interaction with Facebook.*

(a) A remotely hosted proxy server. Supports multiple platforms.

(b) A proxy server running on localhost. Supports multiple browsers.

(c) Within the browser, outside the browser sandbox. Extensions and plugins operate here and have elevated priveledges over normal site code. Other examples include signed Java applets, ActiveX controls and to a lesser extent inline Flash and Silverlight applications. [1] 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.

[1]Flash applications, for example, are restricted but can provide basic filesystem access `http://blog.chromium.org/2010/12/rolling-out-sandbox-for-adobe-flash.html`.

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

(a) conflicts with the design goal of no server-side computation. The browser sandbox prevents local filesystem access, ruling out (d) if private keys are to be kept securely. Encrypted Facebook takes approach (c) since (b) and (e) are considerably more complex, at some cost to cross-platform compatability.

2.4 Mozilla Firefox extension development

Encrypted Facebook is developed for Mozilla Firefox, the worlds most popular browser (as of January 2011). Manipulating the **DOM!** (**DOM!**) is well supported by browser extensions since the browser interface chrome is often built on existing web technologies. This allows Encrypted Facebook to integrate functionality into Facebook's own user interface which mitigates usebility issues concerning key management. Porting to other browsers is discussed in section XXX.

Firefox extensions are written in JavaScript with partial support for Python and C/C++ [**ffox-lang**]. Performing cryptography in JavaScript is possible but comes with severe performance difficulties [**flybynight**]. Table 2.2 compares approximate performance for each language (full details given in appendix xxx).

Since long delays would hamper usebility Encrypted Facebook uses C/C++ for computation intensive operations. Native code can be executed from within Firefox in at least three ways:

- Creating an XPCOM component. These are linked against a single

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.*

Gecko [2] version; supporting multiple versions is possible but non-trivial [**xpcom**].

- Loading native libraries with `js-ctypes`. Introduced in Gecko 2.0 [**js-ctypes**].
- Using `nsiProcess` to invoke an external application. Capturing output can be difficult.

Encrypted Facebook is designed to work on the latest version of the Firefox web browser, Firefox 4.0, which runs on Gecko 2.0. Since building an entire XCPOM component would be excessive and give little advantage by way of backwards compatibility, the newly introduced `js-ctypes` module is used to load native code.

Gecko 2.0 also provides better support for working with the local filesystem and manipulating images in the **DOM!**.

2.5 The Facebook platform

Each Facebook user has a 'news feed' which contains posts of recent updates from their friends in the network. A seperate 'wall feed' also exists for each user containing that user's recent activity. All users also have a profile where they may fill out personal details and submit status updates and upload images. Such actions create notifications in the relevant feeds. These notifications can then be commented on, and these comments themselves will also appear in the feed.

2.5.1 Activities and content types

Table 2.1 lists the most popular Facebook activites in order of frequency of submission (where available). Popular forms of textual submission (comments, messages, status updates and wall posts) are also supported. No attempt is made to encrypt or otherwise conceal structural elements of the social graph, such as 'likes' or 'tags' since this is a non-goal, as stated in section XXX.

The broadcast encryption scheme described in section XXX adds size overheads to any content that will be encrypted. Without relying on a third-party server all overheads must be stored on Facebook itelf, in some form

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

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 [9]*

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, notes 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 an "About Me" section with a character limit of XXX. With no other obvious attribute that can be easily queried from user ID and is large enough, Encrypted Facebook uses this store the users public key.

2.5.2 Connectedness

Since Encrypted Facebook's broadcast encryption scheme has a transmission overhead proportional to the number of intended recipients, care must be taken to ensure the system works with large enough recipient groups. The number of recipients is bounded by the number of friends a user has has (since content can normally only be shared with friends anyway), which is equivalently 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 [6] [**fb-connectedness**]. Marlow et al suggest that, regardless of the number of friends, communication only occurs with a small core group [**burke2010social**].

The Dunbar number is a theoretical cognitive limit to the number of people 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.

2.5.3 Signal-to-noise ratio

The signal-to-noise ratio refers to the amount of useful content in relation to junk content. By junk content, we typically mean spam but in our case this could be transmission overheads as part of the broadcast encryption scheme, or even the encrypted content itself 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. Any solution should therefere take measures to limit the impact it has on the signal to noise ratio of non-users.

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 use the Facebook Graph API directly.

Graph API has objects and connections between them. Every object has a unique ID. Make HTTP requests, return JSON objects.

- Authentication. Authentication and authorisation is done using client-side OAuth 2.0 protocol.

```
https://www.facebook.com/dialog/oauth?
  client_id=YOUR_APP_ID&
  redirect_uri=YOUR_URL&
  scope=email,read_stream&
  response_type=token
```

- Reading. https://graph.facebook.com/ID.
- Publishing.

```
https://graph.facebook.com/arjun/feed?
access_token=XXX&
message=XXX
```

  Returns the ID of the new object.

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 currenly 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 tranlate from one format to the other.
- It certain cases, though publishing through the Graph API is possible, it is more conveniant 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

Encrypted Facebook is capable of encrypting images: not only are images one of the most popular forms of shared content, images have also been highlighted as a prime privacy concern on social networks [**fb-images**]. We describe the lossy stages of Facebook's JPEG compression and evaluate naive attempts at encoding data in images, before describing some more advanced approaches.

2.6.1 JPEG compression process

Regardless of the input format, Facebook encodes all uploaded images using lossy JPEG. [3] 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 lumuninance channel.

3. The discrete cosine transform is applied to each 8x8 block using floating point arithmetic.

4. DCT coefficients are quantised according to values in a quantisation matrix.

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.

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. [4]

---

[3] Even if a file is already in the output format, the compression process re-encodes and information is still lost.

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

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

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

2.6.2 Naive data insertion

We encode data in an image and compress/decompress at quality factor 85, using libjpeg. We then examine the Hamming distance between the output and the original data and compute the bit error rate.

For each DCT coefficient with corresponding quantisation coefficient c we know that:

- $\lceil \log_2 n \rceil$ low order bits will be lost during quantisation.
- Setting higher order bits is undesirable due to clipping. Large magnitudes are more likely to produce values which fall outside the 0-255 interval when performing the inverse DCT.

Figure XXX graphs the error rates for naive insertion in to RGB values and insertion into DCT coefficients using a bitmask. Figure XXX also shows the per image capacity calculated by modelling the compression process as a binary symmetric channel (details for this are given in Evaluation section XXX).

2.6.3 Advanced data insertion

We map binary data on to appropraite length gray codes to ensure that only single bit errors occur from erroneously an adjacent codeword.

- JPEG DCT compression selectively quantises high frequency components since these are less perceptually salient. 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 [**haar**] data encoded in the low-frequency approximation coefficients can survive JPEG decompression when combined with an error correction scheme.

- What will from now on be refered to as a n-bit scaling method. We map the n-bit input space on to the 8-bit pixel space by scaling and shifting the input such that 0 corresponds to 0, and 2^n corresponds to 256. The inverse process amounts to outputting which interval the data lies in.

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 to ensure that a) both of the proposed schemes can be implemented simulatnaously and their performances compared and b) to aid future development of a more optimal solution.

2.7 Further security considerations

2.7.1 Threat model

We describe an attack centric threat model based on the methodoloy of [**XXX**]

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 eliptic 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 due to patent concernse [**XXX**]. Again, we make it a requirement that the project be modular enough to allow later insertion of more optimal scheme.

2.7.3 Key management

- Key management and size (NIST recommendations).
- Message key and IVs, don't reuse. Ensure good source of entrophy.
- Private key policy. Find a good reference, but basically we just mimic SSH and the like.
- Public key policy. Good idea to warn the user of the risks when they add public keys, check SSL is enabled etc.

2.8 Testing plan

- What kinds of testing will I use?
  - Unit testing , anything else??
  - Cognitive walkthrough - does this count as usability testing?
  - Security testing, since potential for exploits and project is security based - important enough to warrant its own section.
- How can I make these tests possible? Test bed or framework that needs to be in place?
  - Need a method of simulating the Facebook JPEG compression process. Use libjpeg since it most closely matches the compression signature (table of compression signatures). Show coefficient table.
  - Need a BER (bit error rate) calculator. Again coded as a C function.
  - FireBug and FireUnit for unit testing and profiling JavaScript functions.
  - gprof for profiling C/C++ functions.

2.9 Security testing

Loosely based on methodology here `http://mtc-m18.sid.inpe.br/col/sid.inpe.br/ePrint%4080/2006/12.20.12.15/doc/v1.pdf`. Must compromise since full security audit beyond scope of project. Look only at text retrieval process and public key management. We ignore images, and general attacks (e.g. setting up a spoof Facebook site). We also ignore threats that would be present ANYWAY e.g. if you haven't got SSL on. As an extension expand threat model.

- Threat analysis. Threat = Agent x Mechanism x Asset.
  - Facebook user creates a tag, which when decryption is attempted, causes denial of service (by locking up resources).
  - Facebook user creates a tag which when decrypted injects script in to page, gains control of users browser, can exectute arbitraty scripts within the Facebook domain (XSS) gains access to Facebook cookies.
  - Facebook user exploits UTF8 encoder/decoder to smuggle illegal characters past sanitization, gains control of users browser, can exectute arbitraty scripts within the Facebook domain (XSS) gains access to Facebook cookies.
  - Facebook user injects text which is run by JavaScripts eval() function, can execute arbitrary JavaScript outside the sandbox. Very Very bad!
  - Facebook user creates public key which, when parsed, creates a malicious file on the users local system.
- Risk analysis. Risk = (Vulnrability x Threat x Impact) / Security Measures.
  - Highest impact is running code outside the sandbox. True it maybe unlikely so long as we aren't stupid, but still. Basically we ban use of the eval function except for when we need it (retrieving JSON objects) then we replace it by a secureEval() which only allows valid Facebook object things.
  - Access to Facebook cookies can impact our security guarantees (since they could then change the public key). Also vulnrability is high. Thus we take time to sanitize before we inject into the browser.
  - Denial of service is low impact, but high vulnrability since the user need not do anyting to initate the decoding process other browsing to a site with a malicious post. So, test UTF8 decoder a lot, ensure that UTF8 decode, FEC decode, decrypt, all fail gracefully. Not image decode since out of scope, as mentioned above.

    - Public keys we can limit to Base64 characters of a certain length. Done.

- Test plan elabouration. From the above we want:
    - Testing of secureEval. Overide or otherwise ban eval().
    - Testing of text sanitiser.
    - Testing of UTF8 de/codec. Complicated given the large range of i/o.
    - Testing of public key downloader.

2.10 Proffesional practice stuff

- Software development methodology. Iterative prototyping. Work plan spells out which prototypes with what functionality should be completed when.
- Coding conventions, const correctedness etc.
- Version/source control. Git and project locker.
- Performance bounds. Of what???

2.11 Requirements analysis

- Encryption should be available on the most commonly used tasks (apart from those otherwise ruled out in section XXX). The user must therefore be able to broadcast-encrypt, submit, retrieve and decipher the following objects.
    - Status updates
    - Wall posts
    - Comments
    - Messages
    - Images

  Specifically, encryption should ensure confidentiality of data with at least 128 bits of security.

- All requirements should be met with recipient groups of size up to 400, which is a reasonable number - refer to discussion.

- Should be unobtrusive (refer to introduction) i.e. must not negatively affect browsing/Facebook experience of users. From this we derive the following:

  - 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.

  - Retrieval and submission times should be within acceptable limits. Define acceptable as `http://www.useit.com/papers/responsetime.html`.

  - Must not confine users to one computer. Should be portable. Securely transporting private keys is up to the user however.

- User activity should not negatively affect the activity of non-users (because of XXX refer to rest of preparation). We know there has to be some increase due to, for example, broadcast encryption overhead and status update's tiny length. Lets say maximum of twice number of objects generated compared to a normal user for the same activity.

- There are uncertainties and/or tradoffs 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, it is highly important that we adopt a modular structure that fascillitates switching between differing schemes and permits future extension. This need not extend to simultaneously supporting different schemes - this would introduce much redundant complexity.

3 Implementation

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, shown on Figure 3.1:

Toolbar XUL
: The toolbar XUL defines the toolbar interface using the Mozilla XML User Interface Language (XUL). The component is loaded as part of the browser chrome when Firefox is started and is responsible for loading the JavaScript components.

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`. This layer of abstraction means updates due to changes in the Facebook web site are isolated to this component. Component re-use (e.g. in an extension for another browser) is also fascillitated.

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. During the login process `efb` attaches the `pagecept` HTML parser to page loading events and

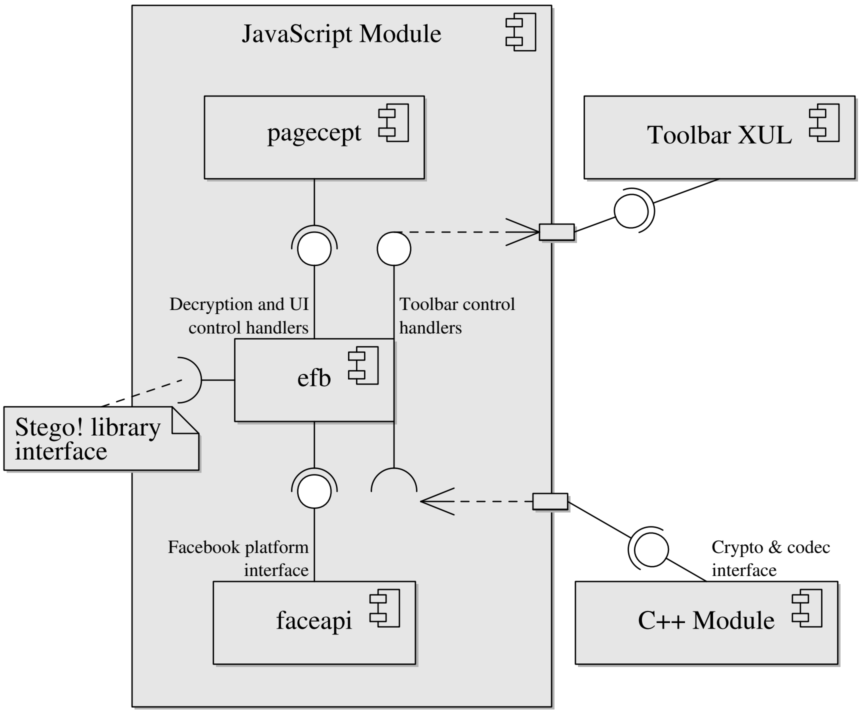

Figure 3.1: *UML component diagram for the extension.*

initialises the C++ module, binding the native methods to JavaScript wrapper functions.

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 libary 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 classe with all pure virtual methods and a virtual destructor to disable polymorphic destruction [**interfaces**].

The library is built around the abstract factory pattern described in [**dpatterns**]. This allows us to encapsulate groups of complimentary sub-components since some interdepenence exists between sub-components. [2] Figure 3.2 outlines the pattern structure with an example concrete sub-class HaarWTConduitImage.

3.1.3 Initialisation

Firefox loads the toolbar XUL as part of the chrome when the browser is started. The XUL then loads the javascript components which define handlers for the toolbar controls. The extension is initialised when either the start button or generate 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. The C++ library will then be instantiated and the pacecept handler attached to page load events.

---

[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.

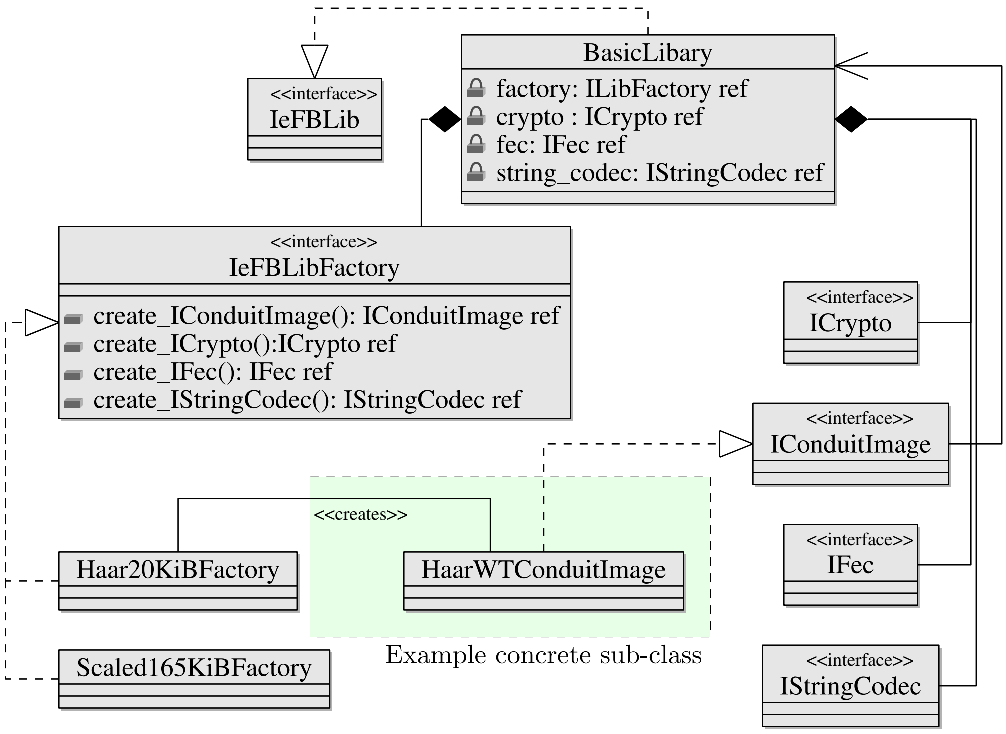

Figure 3.2: *UML class diagrams for the library and its sub-components.*

3.2 Modifying the Facebook UI

- Inserting submission controls
- Retrieving content automatically

3.3 Text submission

We consider the process by which a plaintext message, given a list of recipients, is transformed to a steganography encoded tag which points to the ciphered message stored online as a note. This tag can then be used in place of the plaintext.

The string 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 and a tag generated from the resultant ob-

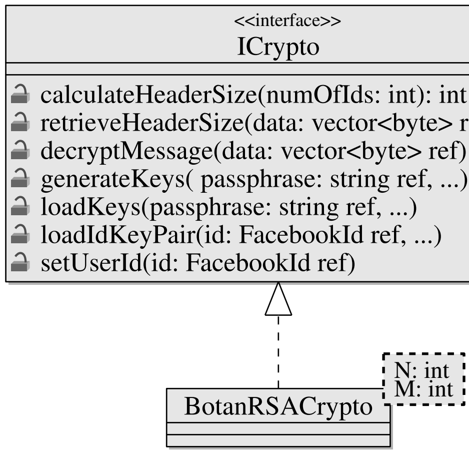

Figure 3.3: *UML class diagrams for the cryptography component.*

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.*

ject ID using steganography. We describe each step of this process in detail.

3.3.1 Encryption

Currently the only implementation for cryptographic functions is based on the Botan library using RSA and AES. The Botan library is encaspulated 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 paratameters.

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.

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 encyrpted 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.

3.3.2 String coding

The input is a byte vector of random bytes from the encryption stage. 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 a 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 code points are outside the BMP (Basic Multilingual Plane) but are still well defined as UTF-8 characters and supported by Facebook.

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

3.3.3 Submission as a note

The final string is submitted as a note to Facebook via `faceapi` passing the relevant handlers from `efb`. On completion the Facebook Graph API object ID is parsed from the XmlHttpRequest response and encoded using the steganography functions provided by the JavaScrypt Stego! library. 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.

3.4 Image submission

We now describe the process by which an image, stored locally, is encryped 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 is created, written to, and saved to disk. We describe the last two stages in detail.

3.4.1 Error correction

Currently the only implementations are based on the Shifra library using Reed Solomon codes. The bulk of 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 3.4).

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 crytography [**XXX**]. The FEC codes are prepended to the data so that, provided there are enough blocks, any last partial block will be padded out automatically.

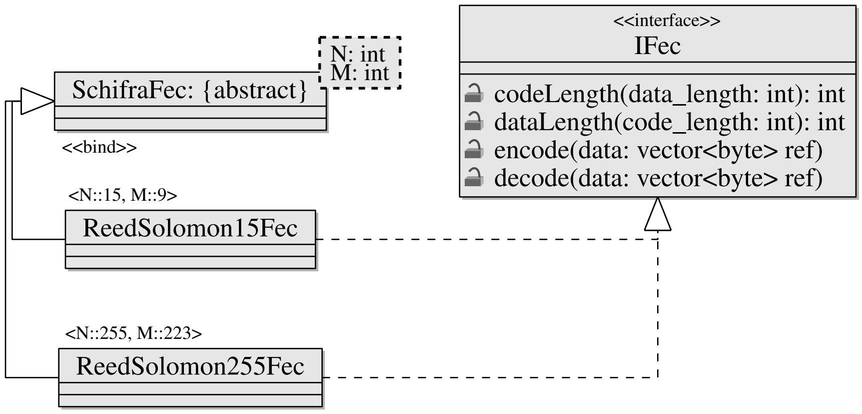

Figure 3.4: *UML class diagrams for the forward error correction libary component.*

Figure 3.5: *Error correction self-padding scheme.*

3.4.2 Read/write buffering

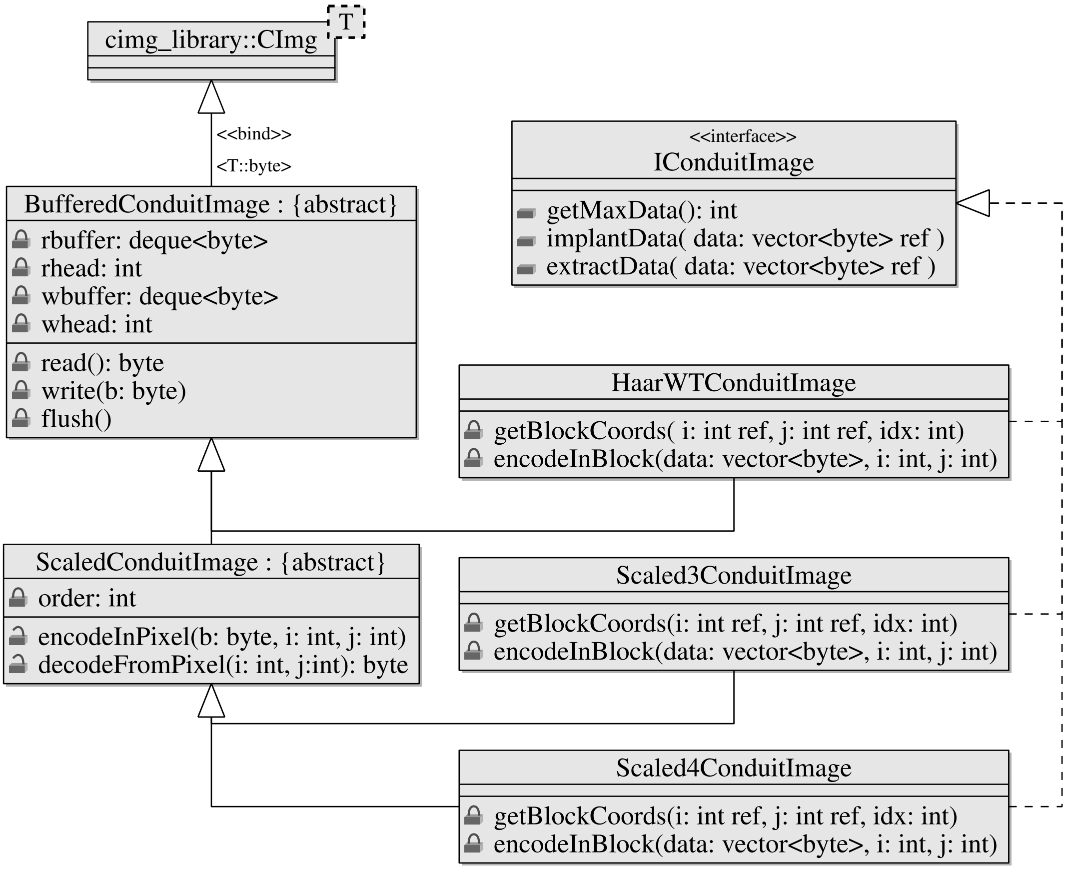

Figure 3.6: *UML class diagrams for the conduit image implementation.*

3.4.3 Haar wavelet transform

3.4.4 N-bit scaling

3.5 Testing

- Unit
- Regression
- Black box
- White box
- Integration
- Security/penetration testing?

Security tests

- Use of the eval() and secureEval() functions
  - test secureEval(). Should only decode JSON objects. Should only do so from Facbook API requests.
- Insertion of text in to page. Easy since we can use JavaScript and RegExp.
  - We allow all uppercase lowercase letters and numerals. Also allow .,?!(). That's it, better safe than sorry. Means no linking to malicious pages. Fully test all boundary cases etc etc.
- UTF-decoder. Slightly harder since have to look at bytes not characters. Using the following rules we conformance test, test all boundaries etc etc. Put list of test inputs in appendix.
  - We accept any valid, non-overlong, UTF8 byte sequences, max length 4-bytes, with scalar value:
    * 0xB0 - 0xD7FF
    * 0xE000 - 0x100AF
    * 0x1B000 - 0x1BFFE (would-be surrogate pairs)
    * 0x10F0000 (indicates a padding byte was added, only one allowed per decode)

- We therefore must throw an exception whenever a valid UTF8 byte sequence is presented with scalar value:
  * 0x0 - 0xAF (out of range)
  * 0xD800 - 0xDFFF (surrogate pair characters)
  * 0x100B0 - 0x1AFFF (out of range)
  * 0x1BFFF - 0x10EFFFF (out of range)
  * 0x10F001 - 0x1FFFFF (out of range)
- We also throw and exception for valid UTF8 sequences when:
  * They have an overlong form i.e. the same scalar value can be represented using a shorter byte sequence.
  * They have scalar value 0x10F0000 (padding character) but this has already been seen during decoding.
  * They have scalar value 0x10F0000 (padding character) but the final decoded byte sequence (before padding removal) has length less than 2.
  * The final decoded byte sequence has length less than 1.
  * They are longer than 4-bytes.
- Naturally we reject any (invalid) UTF8 byte sequences with:
  * Unexpected continuation bytes when we expect a start character.
  * A start character which is not followed by the appropriate amount of valid continuation bytes - including start characters right at the end of a sequence.

• Public key downloader. Simply limit size, don't use exact size since other implementation might use different key sizes.