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 [2]. This project implements a Firefox extension which permits the use of broadcast encryption when sharing certain content via Facebook.

1.1 Background

As social netorks mimic real life interactions members are often willing to reveal more private infromation than they otherwise would []. Networks accumulate a large repository of sensitive user information which can then be exposed in any number of ways. Privacy controls may be misconfgured due to user error or poor design; Facebook was recently forced to update its privacy controls due to growing pressure from the public and media [**fb-priv**]. Error or neglect on the part of the social network may result in breach of privacy, as was the case in 2010 when personally identifiable Facebook user details were sold to data brokers by third party developers []. Unlike communications over telephone, email or post, Facebook content can ofen be openly intercepted and read by the authorities without requiring a warrant []. Deliberate attempts to unlawfully access information can also occur; when social application site RockYou was the victim of a malicious attack 35 million user passwords and account details were made public [].

Since its inception, Facebook has come under criticism in relation to online privacy [1].

Despite growing concern around issues of online privacy (and the related issue of security) even where counter measures are available adoption rates have often been slow. An example is HTTPS support being either disabled by default or unavailable completely from several prominent search, email and social network providers []. This is partly due to an unwillingness to add SSL latency overheads []. Clearly there is a demand for security solutions which operate transparently without detrimenting the typical user experience.

Recently there have been attempts to create alternatives to Facebook [**pidder**] which provide better privacy protection by encrypting shared content. Some propose to go further and decentralize hosting of the social network platform [**diaspora**]. Unfortunately, network externalities make it very difficult to compete [6] since the utility of a social network is linked to the size of its userbase [1].

The aim of this project is to provide enhanced privacy for existing Facebook users. By incorporating a broadcast encryption scheme via a Firefox extension we ensure confidentiality of shared content to a select set of recipients, without otherwise impacting browsing and with minal user supervision required.

1.2 Limitations

Some objectives we consider beyond the scope of the project:

- We do not attempt to hide or conceal any aspect of the social graph structure. This would be tantamount to creating our own social network.
- We do not ensure the integrity, authenticity or non-repudiation of communications, though in theory the public key scheme could be extended to do so. This would go beyond mere privacy control.
- We do not ensure availability of any communications since all content is stored on Facebook's servers. Storing content elsewhere is discussed in section XXX.
- Designing and implementing a security policy which comprehensively covers all aspects of the project's functionality, though essential for actual deployment, would be prohibitively time consuming. Section XXX discusses which functionality the threat model does cover. Operations which haven't been fully audited are disabled by default where possible and come with security disclaimers attached. Section XXX discusses possible deployment and extension of the security policy.

[1] Some suggest the value of a social network grows linearithmically, quadratically or even exponentially with the number of users [**metcalf**].

1.3 Existing work

- Symmetric only schemes:
  - FireGPG - http://blog.fortinet.com/encrypting-facebook/
  - CryptFire
  - TextCrypt - http://subrosasoft.com/OSXSoftware/index.php?main_page=product_info&cPath=210&products_id=207

  Lots of these around.

- Complete schemes:
  - uProtect.it
  - flyByNight - http://hatswitch.org/~nikita/papers/flybynight.pdf

2 Preparation

2.1 Approaches

- Can't we just store message on a server, authenticate and server will let you download? No. This just defers the problem (uProtect.it does this). We actually want cryptographicly secure communication - if the server contents were made public we would still be OK. So we MUST execute crypto functions locally (since homomorphic encryption is shit).
- Choices we have, based mainly on existing work:
  - Where are the private keys stored?
    - Rely on memorised passwords. Means we can store keys remotely, authenticate, download and unlock. Or, we could just hash the password or something. More portable BUT can't have big enough keyspace.
    - Locally.
  - Where are public keys stored and how are they distrubuted?
    - Locally, and distributed manually. Not really useable. Must be unobtrusive remember?
    - Through/from a trusted keychain. Nice but we would need to set something up, have a third party service which was registered.
    - With Facebook.
  - Where do we store messages?
    - Locally isn't feasible (unobtrusive, transparent).
    - On Facebook.
    - On a third party server. Can we scale this?

  - Where do we intercept Facebook interaction? This is the crucial question, can probably slim the content on the previous ones. Show a nice diagram.
    - Behind the browser (remotely) (e.g. a remotely hosted Facebook client running server side code)
    - Behind the browser (locally) (e.g. proxy server on localhost)
    - In the browser (inside the sandbox) (e.g. JavaScript, Java Applet (ughh))
    - In the browser (outside the sandbox) (e.g. Greasemonkey or extension. Maybe Flash aswell since can access local filesystem but Chrome sandboxes `http://blog.chromium.org/2010/12/rolling-out-sandbox-for-adobe-flash.html`)
    - No browser - custom built client

- The conclusion:
  - Minimal third party reliance - can't scale, reliability, complexity. Everything stored/performed on Facebook or locally. One exception is app itself.
  - If we want to store local state i.e. private keys, we can't use remote client, Greasemonkey or sandbox browser stuff. Custom client and proxy server just too complex. So extension all thats left.

2.2 Firefox and extensions

- Basics on how to build an extension.

- Due to JavaScript being shit we need some native code. Only choices are Python or C/C++. Speed is in our criteria. Do a side by side speed comparison (and of JavaScript as well). C is faster => we use C.

- Why we had to use Gecko 2.0 (I think it was to do with better file handling and better native code support).

- How to call C++ from JavaScript (binding, linking, compilation, marshalling). You should probably find out how this works. Note that we need Gecko 2.0 so we can avoid all that XPCOM nonsense.

- Conclusion: Firefox 4/Gecko 2.0 extension with C/C++ backend, avoiding XPCOM.

2.3 Facebook

- Activity we need to support. What the average Facebook user does http://www.onlineschools.org/blog/facebook-obsession/. Comments by far the most common activity. Messages and photos next most common. Then friends but this is social graph. Then status updates, wall posts. Invites and tags - again social graph. Finally links, not social graph but hard to integrate - doesn't matter too much since less popular than other usage forms. Can still share links manually anyway. Likes are probably on there somewhere - again we don't care.

  Clearly images and notes are the most information dense so we use these. To summerise we must support the most popular activities and will likely need to make use of the most dense objects.

- Deleting objects. Need to allow for INCREMENTAL DEPLOYMENT. Needs to be an opt in scheme, don't want to divide SNS. Non-users must be able to co-exist with users. By this we basically mean low signal to noise ratio, link somewhere about this. This might require deleting objects

- Communicating with Facebook. Now we know what we need to interact with, how do we do it? Facebook Graph API (JavaScript API etc.). What we can do. What we can't that we need to (writing to profiles, problems with album ids etc.) Why I didn't use the JavaScript SDK - poorly documented, working with images is a pain. For the workarounds - iframes and forms - see the implementation.

- Connectedness.
  - Facebook says average is 130. This paper http://arxiv.org/PS_cache/cs/pdf/0611/0611137v1.pdf says average is 170 and distrubtion drops sharply at 250. Cameron Marlow says we only speak to a core group of friends anyway.
  - Theoretical limits - Dunbar number of 150, others suggest higher at 300. So it perhaps won't increase with Facebook's expansion.

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. Frequency is measured in number of submissions per second.*

  - Conclusion - 400 if we can make it work, but anything as low as 100 we could probably get away with.

- Conclusion: we must be able to send objects X,Y and Z, and will probably need to make use of A,B and C. Three different ways needed to connect to Facebook to achieve this. No JavaScript SDK; must not significantly increase signal to noise ratio; must suport at least 400 friends/recipients.

2.4 Storing data in images

Images are the big thing. uProtect.it don't do images, FlyByNight talk about it as an extension.

- Description of Facebook/JPEG compression process (problem statement)
- Analysis of theoretical capacity
- Evaluation of naive implementations (MATLAB demonstrations). Obiously use gray codes to begin with.
  - Completely naive (encode in RGB values)
  - Slightly less so (encode in DST with bit masks)
  - Possible better schemes: Haar and Scale.
  - Optimal scheme? Dirty paper coding or similair? Its just too hard a problem but future work could be interesting.
- Conclusion: possible approaches : Haar and Scale. Modularity and extensibility useful though since we don't know which is best and both are sub-optimal.

2.5 Encryption schemes

- Threat model/analysis.
- Proxy re-encryption, like FlyByNite. Requires server side encryption so we leave it.
- Broadcast encryption.
  - Naive implementation.
  - More advanced schemes (and why I don't use them). Mainly due to no server side code, can't ask users to perform operations and expect a reply any time soon (or at all). We could reuse headers which would reduce a lot of size. In many ways this is a great idea, have a big linked list of notes containing links to headers, move frequently accessed entries to the front etc. However this means reusing message keys which is bad practice/NIST recommends against. Also, since we can only add and delete (not edit) notes (not through the API anyway, how fucking annoying) we might run into performance issues.
  - Underlying symmetric/asymmetric schemes. Maybe I could have improved the block size but ECC patents mean its not really found in open libraries.

- Conclusion : simple broadcast encryption using AES and RSA underneath. However, modularity and extensibility would be useful because there are improvements. In particular ECC would give dramatically smaller overheads.

2.6 Further security considerations

- 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.

2.7 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.

2.8 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.

2.9 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???

2.10 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 Project overview

- UML diagrams
- Class diagrams
- Orchestration diagrams
- Directory structure
- JavaScript extension structure

3.2 Encoding decoding data

- Encryption and decryption
- *Keeping key material safe*. Shredding RNG seeds and keys. Using SecureVector. Refer to NIST
- Forward error correction
- UTF8 encoding/decoding
- Text steganography. Keep this short. Stuff about it in testing anyway.

3.3 Storing data in images

- Abstractions
- Using the Haar wavelet transform
- Using upsampling

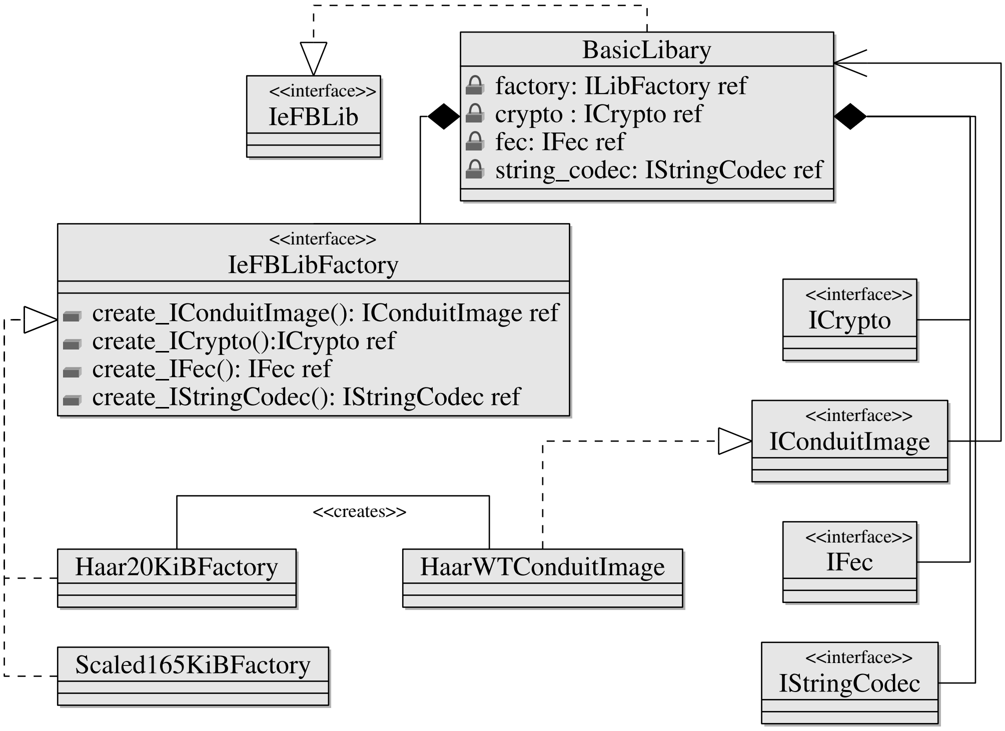

Figure 3.1.: *UML class diagrams for the library and its sub-components.*

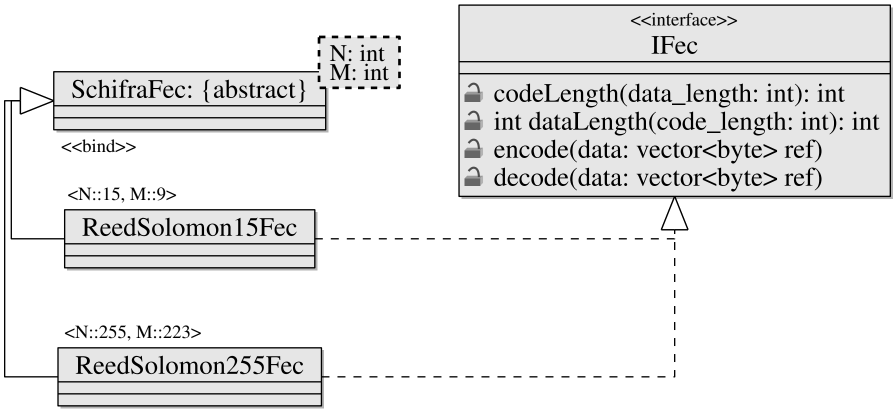
SchifraFec: {abstract}
N: int
M: int
<<bind>>
<N::15, M::9>
ReedSolomon15Fec
<N::255, M::223>
ReedSolomon255Fec
<<interface>>
IFec
codeLength(data_length: int): int
int dataLength(code_length: int): int
encode(data: vector<byte> ref)
decode(data: vector<byte> ref)

Figure 3.2.: *UML class diagrams for the forward error correction libary component.*

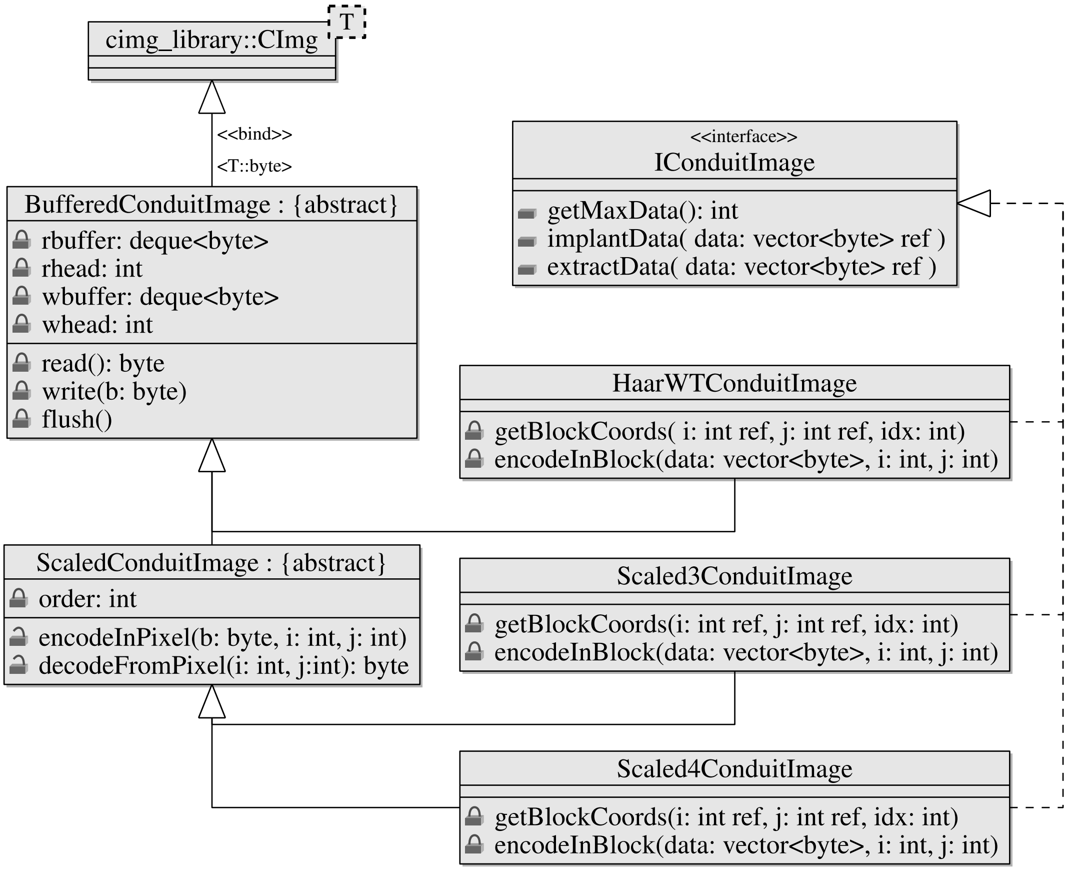
cimg_library::CImg
T
<<bind>>
<T::byte>
BufferedConduitImage : {abstract}
rbuffer: deque<byte>
rhead: int
wbuffer: deque<byte>
whead: int
read(): byte
write(b: byte)
flush()
<<interface>>
IConduitImage
getMaxData(): int
implantData(data: vector<byte> ref)
extractData(data: vector<byte> ref)
HaarWTConduitImage
getBlockCoords(i: int ref, j: int ref, idx: int)
encodeInBlock(data: vector<byte>, i: int, j: int)
ScaledConduitImage : {abstract}
order: int
encodeInPixel(b: byte, i: int, j: int)
decodeFromPixel(i: int, j:int): byte
Scaled3ConduitImage
getBlockCoords(i: int ref, j: int ref, idx: int)
encodeInBlock(data: vector<byte>, i: int, j: int)
Scaled4ConduitImage
getBlockCoords(i: int ref, j: int ref, idx: int)
encodeInBlock(data: vector<byte>, i: int, j: int)

Figure 3.3.: *UML class diagrams for the conduit image implementation.*

- Using bitmasks on DCT coefficients

3.4 Interfacing with Facebook

- Using the Graph API
- Obtaining access tokens
- Generating and submitting forms
- Through iFrames

3.5 Modifying the Facebook UI

- Inserting submission controls
- Retrieving content automatically

3.6 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.

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 [3]. The average broadband speed in the UK is around 10 Mbps [4].

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.

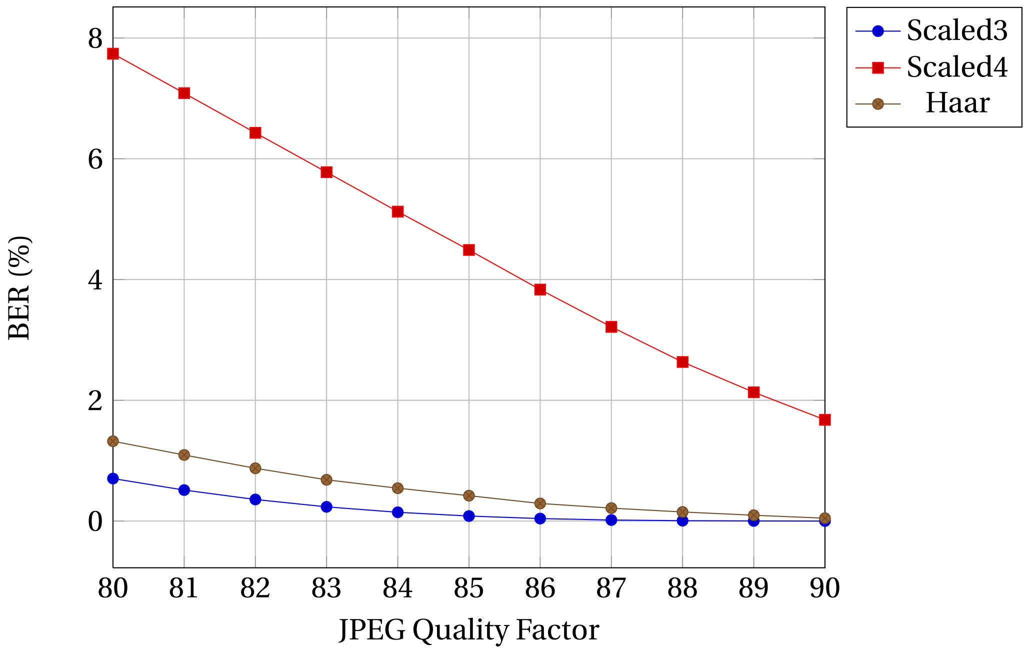

Figure 4.1.: *Bit error rate for varying quality factors.*

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:

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

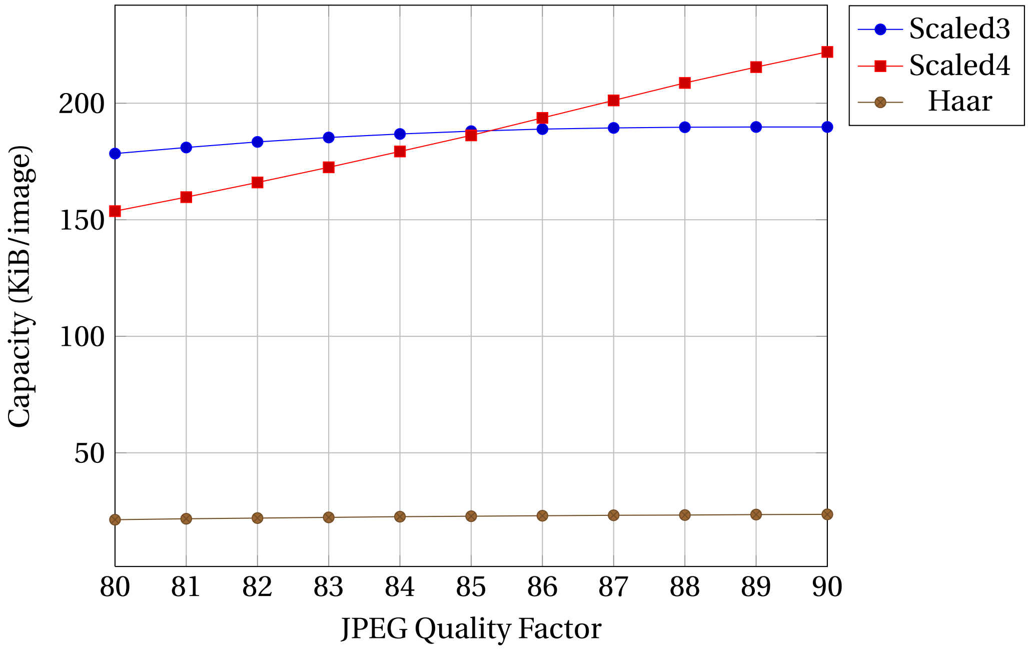

Figure 4.2.: *Per-image channel capacity (measured in KiB/image) for varying quality factors.*

where t is the maximum number symbol errors we can correct [5]. 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.

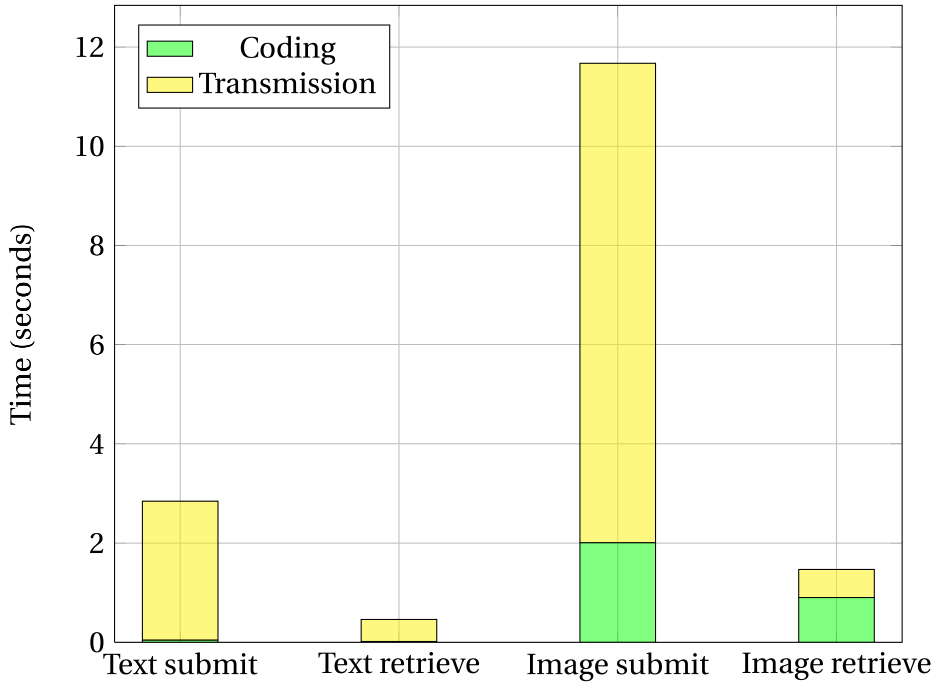

Figure 4.3.: *Average timing results for a 10,000 chracter note and an (approximately) 50 KiB image.*

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.

4.3.2 Results

The effect of plaintext caching on the overall page loading times can be seen in figure 4.4. Caching provided a factor 2 speed increase for text and a factor 4 increase for images.

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 4.5 we again see a modest improvement in loading times. However, looking at figures 4.6 and 4.7 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.

4.4 Useability inspection

A full user study was considered too time consuming to perform; instead we use a cognitive walkthrough (as described by Wharton et al [7]) to evaluate the success of the user interface. During development many passes of the walkthrough were perfomed and alterations made, until a

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

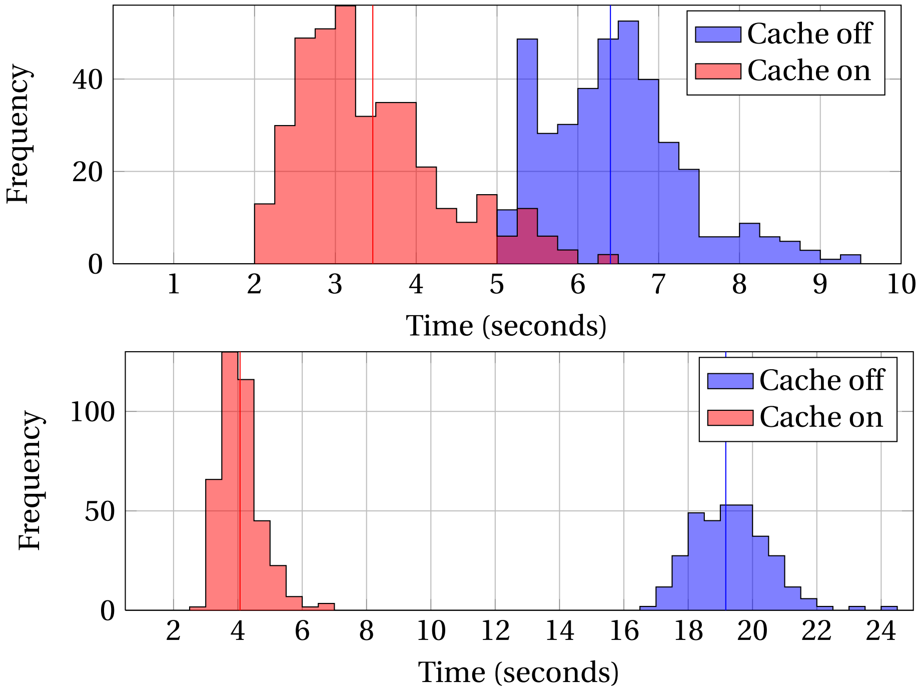

Figure 4.4.: *Histogram of 400 page loading times for newsfeeds containing 15 encrypted messages (top) and 15 encrypted images (bottom).*

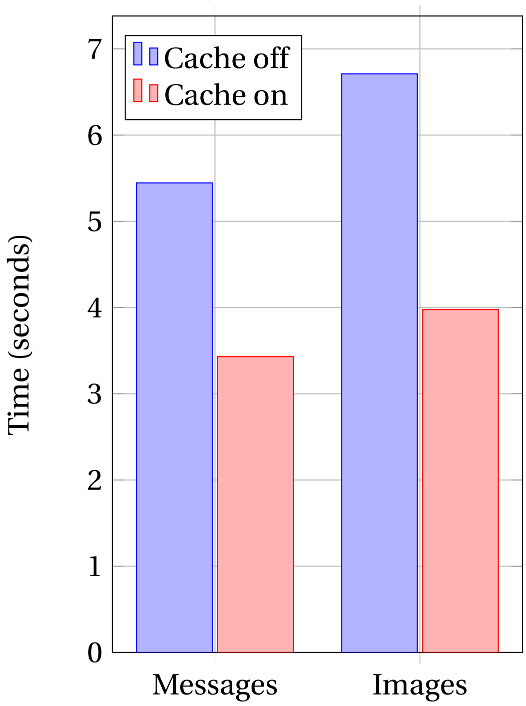

Figure 4.5.: *Average time before first encrypted item loads.*

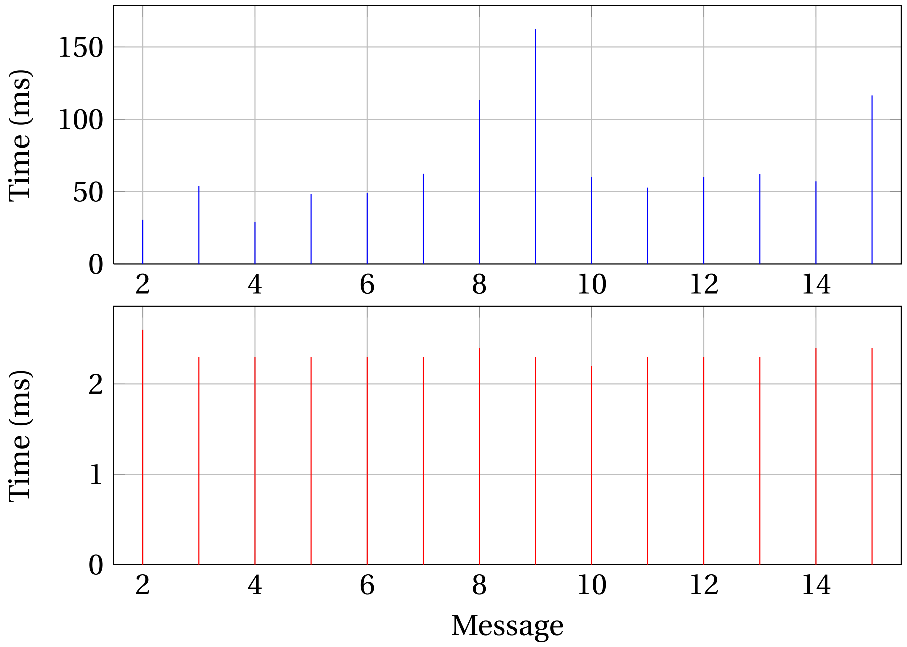

Figure 4.6.: *Average time interval between successive message loads, with caching turned off and on (top and bottom respectively).*

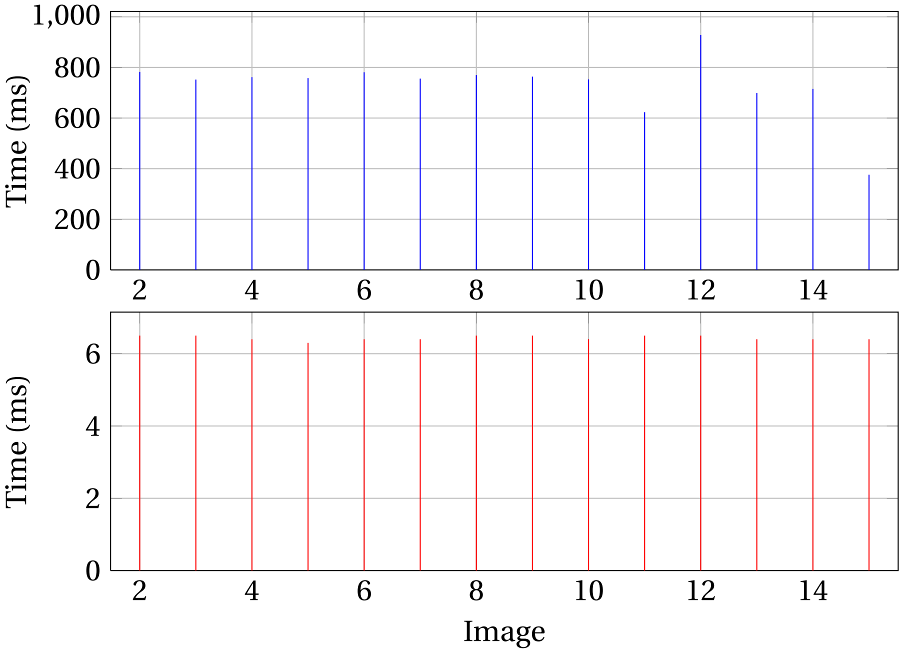

Figure 4.7.: *Average time interval between successive image loads, with caching turned off and on (top and bottom respectively).*

credible success story could be constructed for each task. 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 cancelling will result in a return to the initial state.
- User will either enter a valid password or be prompted by the restrictions on entering an invalid one, in which case he will then know how to enter a valid password and do so.
- User will know if he incorrectly re-enters password via an alert.
- User knows things are OK because an alert informs him the process was succesful.

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.
- 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.

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.
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.

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.

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.

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.

5.3 Future work

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.

5.4 Potential deployment

Obviously would need to expand threat model and deal with security holes. Also need to relax certain constraints i.e. more character support for languages other than English.

Main problem is Linux-only at the moment. Wouldn't take too much trouble to compile on Windows though. (as if).

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.

Bibliography

[1] 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`.

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

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

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

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

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

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

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.

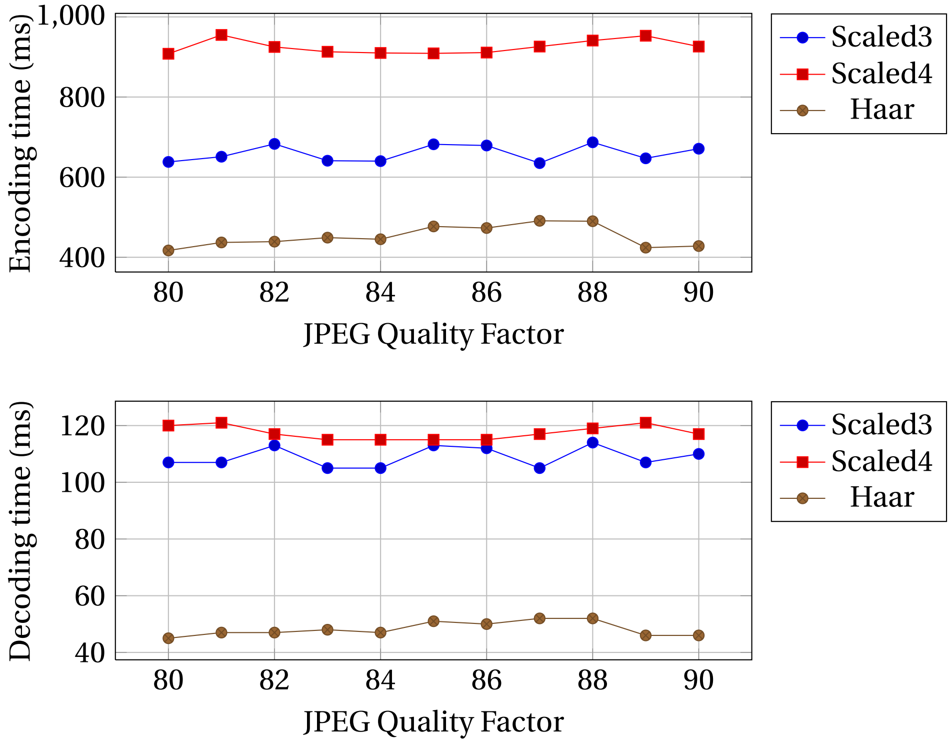

Figure A.1.: *Per image encode and decode times for each of the three conduit image implementations, for varying JPEG quality factors.*

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.