

# 1 Introduction

Facebook is most common social network with X million users and Y daily hits. Currently all communication in plaintext. Single sentence description of project and what it does. Single sentence definition of broadcast encryption.

## 1.1 Background

- General context - recent media coverage of Facebook, privacy and security concerns. Data (user passwords) stolen from web sites by hackers. Privacy a big issue for the 21st century?
- Problems with confusing users and complicated interfaces (e.g. Facebook defaults). Problems with intrusiveness (e.g. SSL and Google, Facebook). Demand for privacy/security solutions that are UNOBTRUSIVE and TRANSPARENT.
- New social networking services being launched (Diaspora etc.). Some have excellent security properties. Problem: networking effects and Facebook user base make it hard to compete.
- Idea : keep using Facebook, add encryption layer on top, seamlessly integrate in to normal Facebook experience.

## 1.2 Limitations

What the application does not do:

- Does not hide the social graph. Arguably this is Facebook's real asset. But that's another story...

- Does not ensure integrity of data. Facebook employees could swap your messages.
- Does not ensure availability. Facebook could easily wipe your notes or images.
- Does not ensure authenticity or non-repudiation. Not from the employees of Facebook anyway.
- Threat model not comprehensive due to project limitations. You can turn off images, which haven't been audited.

## 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](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 cryptographically 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 key space.
    - \* Locally.
  - Where are public keys stored and how are they distributed?
    - \* 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.
- Objects and their sizes:
  - Status update, 144 chars
  - Post
  - Comment
  - Note, 66,000 chars or so
  - Image, 720x720 3 channel 8 bit colour. Not really because of chrominance subsampling.

Clearly images and notes are the most information dense so we use these. To summarise 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/0611/06111137v1.pdf](http://arxiv.org/PS_cache/cs/pdf/0611/06111137v1.pdf) says average is 170 and distribution 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.
  - 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 support 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). Obviously 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 similar? 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 entropy.
- 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.  $\text{Threat} = \text{Agent} \times \text{Mechanism} \times \text{Asset}$ .
  - 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 execute arbitrary 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 execute arbitrary 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.  $\text{Risk} = (\text{Vulnerability} \times \text{Threat} \times \text{Impact}) / \text{Security Measures}$ .
  - 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

vulnerability is high. Thus we take time to sanitize before we inject into the browser.

- Denial of service is low impact, but high vulnerability since the user need not do anything to initiate the decoding process other than 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 elaboration. From the above we want:
  - Testing of secureEval. Override 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 Professional practice stuff

- Software development methodology. Iterative prototyping. Work plan spells out which prototypes with what functionality should be completed when.
- Coding conventions, const correctness 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
- 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
  - First - try and use eval(). Ensure you get an error.
  - Next - test secureEval(). Should only decode JSON objects. Should only do so from Facebook 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 - 0x10FFFFFF (out of range)
    - \* 0x10F001 - 0x1FFFFFF (out of range)
  - We also throw an exception for valid UTF8 sequences when:
    - \* They have an overlong form i.e. the same scalar value can be represented using a shorter byte sequence.
    - \* They have scalar value 0x10F0000 (padding character) but this has already been seen during decoding.
    - \* They have scalar value 0x10F0000 (padding character) but the final decoded byte sequence (before padding removal) has length less than 2.
    - \* The final decoded byte sequence has length less than 1.
    - \* They are longer than 4-bytes.

- Naturally we reject any (invalid) UTF8 byte sequences with:
  - \* Unexpected continuation bytes when we expect a start character.
  - \* A start character which is not followed by the appropriate amount of valid continuation bytes - including start characters right at the end of a sequence.
- Public key downloader. Simply limit size, don't use exact size since other implementation might use different key sizes.



## 4 Evaluation

### 4.1 Comparative analysis of conduit image implementations

- First test Haar wavelet method WITH gray codes. Calculate max capacity from implementation. Create X bytes of random noise split accross N files of max capacity. Encode in JPEG, decode from JPEG. Repeat for all files. Log BER. Also log per byte encode/decode time. Repeat for different levels of JPEG compression, 80 through 90. Plot BER against compression level.
- Repeat this test for Scaled4 and Scaled3 methods, again both WITH gray codes. Use X bytes of noise again, but this time split across different number of files.
- Model as a binary symmetric channel. Calculate the capacity (refer to MacKays books), multiply by the number of bits they can effectively store per image. This is the effective, per image, capacity. Compare.
- An  $(n, k)$  Reed Solomon code will always decode a block correctly providing the number of symbol errors is less than  $t = 1 + \lfloor (n-k)/2 \rfloor$ . When we get  $t$  or more symbol errors the decoder either fails or decodes the wrong output sequence with some non-zero probability - we will henceforth use the term 'unsuccessful decode' to refer either of these events. In the worst case this will occur with only  $t$  bit errors in the decoder input, one in each symbol. Therefore, the probability of an unsuccessful decode fo a single block is bounded by:

$$\sum_{i=t}^{8n} \binom{8n}{i} p^i (1-p)^{8n-i} \quad (4.1)$$

where  $p$  is the bit error probability. Using our measured BER as an approximations of the bit error probability, we calculate an upper bound probability of an unsuccessful decode for each of our conduit image methods.

Should be millions of Gb. This NASA document [http://ipnpr.jpl.nasa.gov/progress\\_report/42-84/84F.PDF](http://ipnpr.jpl.nasa.gov/progress_report/42-84/84F.PDF) shows what they used for the Voyager space probe. Also this approaches hard drive read/write error rates. Calculate the final capacity of each method.

- Conclude - Scaled3 gives highest capacity, however you look at it. Effective error rates might be better for other methods, but who cares since as we have discussed, beyond  $10^{-13}$  no one cares.

## 4.2 Cognitive walkthrough

Use Upsampled3 since we've shown its the best

For one user, X:

- Create a crypto identity.
- Migrate profile information.

Now create 15 more users, friends of user X. (group A). Also have one user who is not a recipient (group B). And one more user who doesn't have application at all (group C). Repeat encryption headers 28 times so we simulate group of size 400. Also repeat entries in UI controls.

- Public key management - add keys of group A to user X.
- Text submission - from X to group A.
- Image submission - from X to group A.
- Text and image retrieval - for user X.
- Text and image retrieval - for one member of group A.
- Text and image retrieval - for group B.
- Text and image retrieval - for group C.

## 4.3 Profiling submission and retrieval operations

- How long does the C++ code take? Gives lower bound on overall times.
- Profile JavaScript functions.
- Actually we can probably profile both at once.
- Compare what's taking the longest, bottlenecks etc.



# 5 Conclusion

## 5.1 Evaluation of Requirements

It works, and works for groups of 400 - covered by cognitive walk-through. 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 walkthrough 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 switch 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 similar, 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++ application 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 different 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 doesn't make much difference but from a theoretical perspective it's 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 separating headers from content is cool. Would be nice to experiment regarding the performance hit, but essentially we could have a sliding parameter which indicated security vs storage overhead tradeoff.

# Bibliography

- [1] Özge Aksin et al. ?Effect of immobilization on catalytic characteristics of saturated Pd-N-heterocyclic carbenes in Mizoroki-Heck reactions? In: *J. Organomet. Chem.* 691.13 (2006), pp. 3027–3036.





# **Appendix**



# A Graph API

Lorem ipsum at nusquam appellantur his, ut eos erant homero concludaturque. Albucius appellantur deterruisset id eam, vivendum partiendo dissentiet ei ius. Vis melius facilisis ea, sea id convenire referrentur, takimata adolescens ex duo. Ei harum argumentum per. Eam vidit exerci appetere ad, ut vel zzril intellegam interpretaris.

Errem omnium ea per, congrue populo ornatus cu, ex qui dicant nemore melius. No pri diam iriure euismod. Graecis eleifend appellantur quo id. Id corpora inimicus nam, facer nonummy ne pro, kasd repudiandae ei mei. Mea menandri mediocrem dissentiet cu, ex nominati imperdiet nec, sea odio duis vocent ei. Tempor everti appareat cu ius, ridens audiam an qui, aliquid admodum conceptam ne qui. Vis ea melius nostrum, mel alienum euripidis eu.

## A.1 Appendix Section Test

Ei choro aeterno antiopam mea, labitur bonorum pri no. His no decore nemore graecis. In eos meis nominavi, liber soluta vim cu. Sea commune suavitate interpretaris eu, vix eu libris efficiantur.

Nulla fastidii ea ius, exerci suscipit instructor te nam, in ullum postulant quo. Congue quaestio philosophia his at, sea odio autem vulputate ex. Cu usu mucius iisque voluptua. Sit maiorum propriae at, ea cum primis intellegat. Hinc cotidieque reprehendunt eu nec. Autem timeam deleniti usu id, in nec nibh altera.

## A.2 Another Appendix Section Test

Equidem detraxit cu nam, vix eu delenit periculis. Eos ut vero constituto, no vidit propriae complectitur sea. Diceret nonummy in has,

no qui eligendi recteque consetetur. Mel eu dictas suscipiantur, et sed placerat oporteat. At ipsum electram mei, ad aequae atomorum mea.

Ei solet nemore consecetur nam. Ad eam porro impetus, te choro omnes evertitur mel. Molestie conclusionemque vel at, no qui omittam expetenda efficiendi. Eu quo nobis offendit, verterem scriptorem ne vix.

# **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. Usability 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.