Using Identity based encryption with the Twitter API

James Judge
B.A.I. Computer and Electronic Engineering
Final Year Project  April 2012
Supervisor: Dr. Hitesh Tewari
Declaration

I hereby declare that this project is entirely my own work and that it has not been submitted as an exercise for a degree at this or any other university.

________________________________________  __________
Name                                Date
Permission to lend

I agree that the library and other agents of the college may lend or copy this thesis upon request.

_________________________________________ 2012

James Judge
Acknowledgements:

I would like to take this space to acknowledge my supervisor, Dr. Hitesh Tewari, for being very supportive and enthusiastic throughout the project. I also would like to thank Karl Reid, a postgraduate student, for the bountiful help and time he gave me. Having you both made this project much easier than it may otherwise have been.

I’d also like to thank my father for reading over this report, my mother for the cups of tea while I coded, and my friends for keeping spirits high and pretending to listen while I arranged my thoughts.
Abstract

Identity Based Encryption (IBE) is an encryption technique which removes the need for prior communications between two bodies wishing to communicate privately. Twitter is an increasingly popular and very widely used social networking site. This project aims to apply IBE to Twitter.
Contents

1. Introduction 3
   1.1 - Project Aim 3
   1.2 - Motivation 4

2. Background 6
   2.1 - Encryption 6
      2.1.1 - Public-Key Encryption 6
      2.1.2 - Identity Based Encryption 12
   2.2 - Google Chrome Extensions 13
   2.3 - Twitter API 15
      2.3.1 – Twitter API 15
      2.3.2 – Oauth 15

3. State of the art 17
   3.1 – Public Key Encryption 17
      3.1.1 – Initial Work 17
   3.2 - Identity Based Encryption 17
      3.2.1 – Initial Work 17
      3.2.2 – The Boneh/Franklin Scheme 19

4. Design 21
   4.1 – Security 22
   4.2 - The Chrome Extension 24
   4.3 - The MIRACL Library 25
      4.3.1 – The MIRACL PKG 26
      4.3.2 – The MIRACL Client 26
4.4 - The Twitter Server

5. Implementation
   5.1 – Compression
   5.2 - Manifest
   5.3 - Background Page
   5.4 - LocalStorage
   5.5 - XMLHttpRequest
   5.6 - OAuth Library

6. Further Work

7. Conclusion
Chapter 1 - Introduction

1.1 – Project Aim

The aim of this project was to create an extension for the Google Chrome browser which would allow users to access Twitter and use Identity Based Encryption (IBE) to encrypt their tweets where desired. The extension offers the users the option to encrypt tweets they are posting and to decrypt tweets they are reading.

The two primary aims of the project were to:

1. Offer a confidential and secure method of communication for Twitter users.
2. Use IBE to simplify the encryption process.

Identity Based Encryption is built upon Public-Key Encryption. It aims to improve upon traditional Public-Key Encryption by reducing the amount of communication required with the Public-Key Infrastructure.

This project offers a use-case for IBE whereby its advantage over traditional Public-Key Encryption is obvious; as Twitter users generally communicate with large numbers of other users the ease of starting IBE communication with a new user greatly simplifies the communication process.
1.2 – Motivation

Twitter

Twitter is a social networking site used worldwide by over 350 million users. Twitter users may post short messages, called “tweets”, of up to 140 characters on their own wall or on other users’ walls (“@” another user). It has been used on numerous occasions to aid in the organisation of protests and revolutions; these are termed “Twitter revolutions”. Examples of these Twitter revolutions are: The Egyptian Revolution 2011, Tunisian Revolution 2010-2011, Iranian Election Protests 2009-2010, Moldova Civil Unrest 2009, and more are likely in years to come.

As it has been used in this way numerous times it is clear that Twitter is a medium of communication which appeals to protestors. Users may easily find people who they are looking for on Twitter but given the open nature of tweets users may find themselves desiring a more secure and confidential method of communicating. Twitter offers a direct messaging option whereby users may send private messages to other users; however an encrypted tweet could be preferred over such methods if users desire other users to be aware of whom they are communicating with, even if they cannot read the tweet itself. Also users organising protests in particularly powerful countries may distrust the security on the private messages especially if the information is very sensitive. Users may therefore prefer to be responsible themselves for the confidentiality of their tweets rather than risk the possibility that Twitter may be coerced into exposing their private messages.

Governments are clearly aware of the potential to revolutionaries that social networking sites have as Iran blocked Twitter and Facebook prior to the presidential elections in 2009. Similarly Twitter was blocked by the Egyptian government during the 2011 revolution.

On occasions Twitter users have been arrested due to their tweets.
Identity Based Encryption

With tradition Public-Key Encryption to find the code (Public-Key) needed to encrypt a message to a recipient the sender must communicate with a Public-Key Infrastructure to obtain this code. With IBE a global Public-Key is used for all users and an individual’s Public-Key can be formed by hashing this global key with their username. This means that Public-Keys can be formed on the fly and there is no need to contact servers to obtain them.
Chapter 2 - Background

2.1 – Encryption

2.1.1- Public-Key Encryption

Public-Key Encryption is an encryption technique whereby every user has a unique Private-Key and a Public-Key. These 2 keys are related to each other mathematically. The user must keep their Private-Key secure and distribute their Public-Key. Users who wish to send a message to another user will make use of the recipient’s Public-Key to encrypt their message. The recipient can then use their Private-Key to decrypt the message. Even though the Private-Key and Public-Key are mathematically linked, neither can be derived from the other.

The following diagram shows how when one user wishes to send a message to another user the sender encrypts the message using the recipient’s Public-Key; the recipient upon receiving the cipher-text can then decrypt it using their Private-Key.
Due to the public and private key being related, it is possible to also not only ensure that only the recipient can decrypt the message, but also to sign the message so that the recipient can be certain only the declared sender could have composed this message. Say we have the case of Alice sending a message to Bob with an eavesdropper Eve.
Without encryption Eve may intercept the message and read it plainly. With encryption when Eve intercepts the message she may only read the cipher-text which she has no means to decrypt.
However if Eve were to intercept the message and change the text Bob would be unaware that the message was not the original.

To account for this it is possible for Alice to put a signature on her message so that when Bob checks the signature only Alice could have made such a signature. Alice can use her Private-Key to make such a signature. Generally the signature is made by taking a digest of the message (as using the whole message is computationally expensive and superfluous) and signing that. Upon receiving a message Bob uses Alice’s Public-Key to retrieve the digest of the message, and uses his own Private-Key to retrieve the message itself. He can then compare the 2 digests to ensure that the message is indeed from Alice.

As every user in this system has a unique Private-Key and Public-Key, before communicating with another user the sender must first obtain the Public-Key of the recipient. This means every time a user wishes to communicate with a new recipient the sender must first make a communication with some form of Public-Key Infrastructure to obtain the recipients Public-Key. In the long run this is costly and can prevent communications if the Public-Key Infrastructure is not online.
Distribution of keys

As every user in the system generates their own individual Private-Key and Public-Key, there is no central body governing the creation of the keys. This leads to the sharing of keys becoming a problem. In practice a Public-Key Infrastructure (PKI) must be created and agreed upon before the system is implemented. After a user has generated their Public-Private-Keypair, they may upload their Public-Key to the PKI. Whenever a sender wishes to retrieve the recipient’s Public-Key from the PKI they must access the PKI and request a certain users Public-Key, therefore they must trust the PKI to give them the correct users Public-Key.

As Public-Keys are used not only for encryption but for verifying a sender there are 2 reasons why it is very important the PKI is deemed secure.

- Firstly, in a less dangerous situation, Alice will not be able to send Bob a message which he can decrypt if the PKI gives Alice a Public-Key which is not Bob’s.
- Secondly and much more dangerously is in the case of a malignant user, namely Eve, we must be aware of the consequences should Eve manage to place her Public-Key in the PKI where Bob’s Public-Key should be. Eve could then possibly be decrypting messages intended for Bob, and replying to them, even signing them with what is allegedly Bob’s Private-Key, whereas in reality it is hers. Alice in this situation could have no idea that it is in fact Eve and not Bob with whom she is communicating.

These are threats which are not present in IBE, however IBE does have its own dangers. The security of the system’s infrastructure is crucial in both traditional Public Key Cryptography and Identity Based Encryption, although a security compromise in the case of IBE is much more severe (as will be explained in the following section). However provided the system is secure there are strong
advantages to IBE as it offers a solution to the problem of distributing users Public-Keys.
2.1.2 – Identity Based Encryption

IBE offers a solution to the problem of sharing keys which is inherent in traditional Public-Key Encryption.

IBE proposes there is one element, the Private Key Generator (PKG) which holds a master Public-Key and a master Private-Key. The PKG can distribute the master Public-Key openly, however the master Private-Key must never be revealed. In order to obtain a recipient’s Public-Key the sender may obtain it by combining the master Public-Key with the user’s identity. They may then encrypt the message as normal by using this Public-Key which it just created for that individual. In order to decrypt the message users must use their individual Private-Key which can be obtained from the PKG. The user’s Private-Key is created by combining the user’s identity with the master Private-Key. Users must authenticate with the PKG in order to obtain this individual Private-Key. The general setup is shown in the following diagram.
Benefits:

Due to the use of a Global Public-Key this means that after one communication with the PKG the user is capable of communicating with any user who has authenticated with this PKG. This brings communications with the Public-Key Infrastructure down to a once off.

If the system is at capacity and every user has obtained their Private-Key from the PKG then the PKG may even be removed from the system. This removes the danger of the Master Private-Key being exposed and also means that the system is more sustainable as there is no PKI to maintain.

Concerns:

As any users Private-Key can be generated through the Master Private-Key it is pivotal that the PKG is secure so that the master Private-Key is never exposed. This is another reason why the removal of the PKG if possible can be very advantageous.

The removal of the PKG would however mean that users would not be able to get their Private-Key again if they somehow lost theirs.

2.2 - Google Chrome Extensions

Google Chrome is a widely used web browser with over 200 million users worldwide [2]. Google Chrome offers users the option to add extensions to the browser; these extensions are features which add additional functionality to the browser, thereby customising Google Chrome to suit the user. Importantly the extensions may be JavaScript code, and in the case of this project, HTML code which runs from a pop-out screen from a browser button (as shown in the following figure). The concept of extending a browser to offer additional functionality has been around since quite early into the history of web browsers, with Internet Explorer 5 (released in 1999) being the first edition of Internet
Explorer and most likely the first browser to offer the ability to be easily extended.

Google Chrome extensions may also run background pages. This is useful for the purpose of a Twitter account whereby users are required to sign in. The sign in process requires redirection to a webpage; this webpage may then pass credentials to the background page and finally back to the pop-out page itself (as it is not possible to redirect to a pop-out page). Credentials may also be stored and accessed through the background page while Chrome is in use, and in local-storage to prevent the need to sign into Twitter each time the browser is restarted. All of this proves ideal for a project such as this one.
2.3 – Twitter

2.3.1 - Twitter API

Like many other online services Twitter offers a documented Application Programming Interface (API) for developers to include Twitter in their applications or on their websites.

This project’s application communicates with the Twitter API through JavaScript HTTP requests. To allow the application to communicate with Twitter on a user’s behalf the user must authorize the application with Twitter.

2.3.1 - OAuth

Twitter uses OAuth for authorizing applications to post on their user’s behalf. OAuth is an open standard to allow secure communication between 2 parties without having to give confidential details across to one of the parties. In the case of this project the user never has to give their Twitter login username and password to the Chrome extension. The OAuth process removes the need for this. The OAuth process happens as follows.
Interaction with Twitter API

Chrome Extension

- Extension performs a GET Request_Token
- Extension redirects user to Twitter sign in Page with Request_Token to give the Extension permission to their account

Twitter Server

- Twitter provides a Request_Token
- An OAuthVerifier is granted upon a successful login
- The OAuthVerifier is given to Twitter
- Twitter sends an OAuth_Token, OAuth_Token_Secret, Screen_Name, and UserID

The OAuth_Token and OAuth_Token_Secret are used to create valid requests using the Twitter API.
Chapter 3 – State Of The Art

3.1 – Public Key Encryption

3.1.1 – Initial work

Public Key Encryption is often also termed asymmetric key cryptography; this is due to the fact that the two communicating parties do not use the same key, as is the case in Symmetric Key Cryptography.

The mathematical basis for Public Key Encryption relies on one way functions and the difficulty of factorization; this was first discussed by William Stanley Jevons in 1874. In 1976 the concept of Public Key Encryption was first announced by MIT researchers Ron Rivest, Adi Shamir and Leonard Adleman [10]. They developed the RSA algorithm which was later published in 1978.

However, years later, in 1997, the British Government Communications Headquarters (GCHQ) released documents which showed they had discovered the principals behind Public Key Encryption in 1970 [11], and discovered a method similar to RSA in 1973. In 1974 the GCHQ had a cryptographic system very similar to that released 2 years later; they did not publicise it for national security reasons.

3.2 – Identity Based Encryption

3.2.1 – Initial work

IBE was first proposed in 1984 by Adi Shamir [1]. While the method of use for Identity Based Encryption in this project is not as IBE was first proposed, the basis for its use is very similar. Due to the introduction of the internet and its widespread use, the practical applications for IBE have since tended to a more online aspect.

It was initially proposed as smart-card system whereby the PKG is an offline system which users must receive their smart-card from. The card was
proposed to have a microprocessor, an I/O port, RAM, ROM containing the secret key, programs for encrypting and decrypting messages, and generating and verifying signatures.

It highlighted the following points as being relied upon for the overall security of the scheme:

a) The security of the underlying cryptographic functions.

b) The secrecy of the privileged information stored at the key generation centres (in our context these are the PKG)

c) The thoroughness of the identity checks performed by the centres before they issue cards to users.

d) The precautions taken by users to prevent the loss, duplication, or unauthorized use of their cards.

However this paper could only offer an Identity Based Signature method and could not offer an Identity Based Cryptosystem. Although it conjectured that such cryptosystems do exist it was unable to prove that breaking the scheme was equivalent to solving a well-known computational problem.

The paper proposed a signature scheme whereby the following condition exists:

\[ s^e = i \cdot t^{f(t,m)} \pmod{n} \]

Where \( m \) is the message, \( s \) and \( t \) are the signature, \( i \) is the user’s identity, \( n \) is the product of 2 large primes, \( e \) is a large prime which is relatively prime to \( \varphi(n) \), and \( f \) is a one way function.

The parameters \( n, e \) and the function \( f \) must be generated the same for all users by the PKG. The factorization of \( n \) should be kept private but the value of \( e \) and function \( f \) can be made public. The difference between users is their identity, \( i \), and their secret key, \( g \), which corresponds to \( i \) as follows:

\[ g^e = i \pmod{n} \]

There are a large number of \((s, t)\) signatures for each message but their density are low enough that a random search is extremely unlikely to discover any of them. To set one of \((s, t)\) to a random value and solve for the other variable
requires the extraction of modular roots which is believed to be an exceedingly difficult computational task.

To sign the message the user chooses a random number $r$ and computes

$$s = g \cdot r^{f(t,m)} \pmod{n}$$

### 3.2.2 – The Boneh/Franklin Scheme

Dan Boneh and Matthew Franklin proposed a cryptosystem for IBE in 2001 [3]. The scheme used the application of pairings over elliptic curves and finite fields.

An elliptic curve is the set of solutions $(x,y)$ to an equation of the form $y^2 = x^3 + ax + b$, together with an extra point $O$ which is called the point at infinity [4]. For cryptography purposes we consider finite fields of $q$ elements.

There are 4 stages to the scheme, Setup, Extract, Encrypt and Decrypt. The maths behind the scheme is beyond the scope of this project.

**Setup:**

The setup is a once off occurrence. It creates the variables required for the PKG to use. During the setup the Master-Private-Key, Master-Public-Key and other more technical system parameters are generated. The Setup also accepts a security parameter.

**Extract:**

The extract phase is run when a user requests their private key. In practice a user must somehow verify their identity to authenticate with the PKG in order to execute this phase but this authentication process is outside of the bounds of IBE protocol.
To create a public key for user ID in the Extract phase the PKG uses the Master-Private-Key, the users Identity and the aforementioned parameters from the Setup phase.

**Encrypt:**

The encrypt phase uses the parameters created in the Setup phase, the message which is required to be encrypted and an identity ID

**Decrypt:**

The decrypt phase uses the cipher-text, a private key and the parameters from the setup phase.
Chapter 4 – Design

The system which this project proposes makes use of 4 bodies.

Physical System Layout

Firstly there is the Chrome Extension itself, which is the user's primary point of interaction. Everything the user is required to do should be possible through the extension which is run on their browser.

The Multiprecision Integer and Rational Arithmetic C/C++ Library (MIRACL) client should be run locally on the user's system. This software will perform the IBE encryption, decryption, signing and verifying signatures. The user should not be aware of the communications between the Extension and the MIRACL client.

The PKG server which is responsible for the generation of user's Private-Keys should be run remotely from users. Users only need to correspond with this body once unless they lose their Private-Key.

The Twitter Server is hosted and run by Twitter. The Chrome Extension communicates with the API offered by this server.
4.1 – Security

In section 3.1.1 we highlighted the 4 security risks to a system implementing the smart-card system of IBE as outlined in Shamir’s 1984 publication as:

a) The security of the underlying cryptographic functions.
b) The secrecy of the privileged information stored at the key generation centres (in our context these are the PKG)
c) The thoroughness of the identity checks performed by the centres before they issue cards to users.
d) The precautions taken by users to prevent the loss, duplication, or unauthorized use of their cards.

As the PKG will be responsible for the generation of keys, the security of the cryptographic functions used in the process is tied with the secrecy of the privileged information within the PKG. Therefore in short we have the following considerations while planning our system.

1) The security of the PKG.
2) Authorization of the user to the PKG.
3) The security of the Private-Key stored on the user’s computer.

1) The Security of the PKG.

In some cases it may be preferable amongst a group of users to run the PKG in an offline scenario, such as was initially intended at the inception of IBE, whereby users must physically go to the PKG to obtain their Private-Key. This of course would only be fitting for smaller scale groups who are most likely located in close proximity of each other. As this is not the case for the majority of potential users, this report envisions the PKG as being run on an online server. For the purpose that this project outlines it is suggested that the PKG be run on a server which has an SSL certificate and is physically remote from users’ machines.

In the case where there are a known limited number of users it is possible to shut down the PKG once all users have received their private key. This would remove the threat of the PKG’s parameters or the Master Private-Key becoming exposed. However the benefits of this come at a cost. Should the
PKG be shut down and a user loses their key there is no way of re-introducing this user back into the system. This means worried users may resort to duplicating their keys which could prove to add a security threat greater than that posed by the PKG remaining online.

2) Authorization of the user to the PKG.

Authorizing the user with the PKG is a problem with a few solutions.

As the secrecy of the users tweets depends on the security of the PKG it can be assumed that users would have no objections towards using their Twitter OAuth credentials to authenticate (Note: these credentials do not contain the user’s password).

I will put forward two suggestions which are based upon this method.

The first suggestion is simpler but has the prerequisite that a user must have been mentioned in at least one tweet. The idea is that the user will pass their OAuth credentials, which they have received from Twitter, along with who they are claiming to be to the PKG. The PKG will then use these credentials to communicate with the Twitter API. The PKG will request the authenticated user’s (i.e. the user who has authenticated with Twitter and received these credentials) mentions from Twitter. Upon receiving them successfully the PKG can examine the contents to see if the username matches with that of the user who is requesting their Private-Key.

Having to be mentioned is a reasonable prerequisite as mentioning other users is extremely common on Twitter, and the user will most likely have been mentioned in tweets if they are to engage in IBE. It should not hinder their ability to decrypt tweets from other users using this application as the sender must have mentioned the user to encrypt a tweet to the user. If the user wishes to encrypt a message it would be a minor once off hindrance to have to mention themselves in a tweet to obtain their Private-Key.

An alternative method which does not have the prerequisite of the user needing to be mentioned would be to allow the PKG to perform the final steps of the OAuth process for the user. The user could perform the first steps of the OAuth process, and upon receiving their OAuthVerifier (see OAuth under section 2.3) they could pass this to the PKG and the PKG could finish the process. The final step of this process involves the return of the authenticated users Screen_ID, i.e. their username. The PKG could then return the user their Private-Key along
with their OAuth credentials which the PKG received in return for their OAuth_Verifier.

The hindrance to this approach is that it requires the user to log out of the Twitter Chrome extension if they were already logged in. It also places more responsibility on the PKG as they must pass more values. However, as it is only a once off occurrence it is considered acceptable.

As the nature of online software is constantly adapting these solutions may not work in the long run should the Twitter API or OAuth process change, hence the reason why two solutions are offered.

3) The security of the Private-Key stored on the user’s computer.

The Private-Key stored on the user’s computer is the user’s responsibility to keep safe. It will be passed securely through SSL connection from the PKG to the Chrome Extension. From there the Extension may pass it to the MIRACL client which can store it locally. Therefore it is pivotal that, on initialising the MIRACL client, the user is made aware of the importance of protecting this key.

4.2 – The Chrome Extension

Purpose

The extension is written for Google Chrome and is predominately JavaScript code. The aim of the code is to provide the user with an intuitive and simple interface with which they may communicate with the Twitter API, the locally running MIRACL client and potentially with the PKG server, although the PKG server may also be run offline if so desired.

Distribution

The extension should be distributed in a packaged extension file (".crx" file type) which is a type of zipped file with a special header that includes a signature. When an extension is packaged the extension is assigned a unique
key pair with the extension’s ID based on a hash of the public key. The private key can be retained by the creator of the extension and used to sign future versions [5]. Google Chrome users can simply execute this file to install it.

Content

The extension runs a pop-out HTML page upon opening it. The page first requires the user to initiate the sign in process. Upon pressing the login button the extension runs JavaScript code to request a Request_Token through the Twitter API. The OAuth steps are then followed through as explained in section 2.3.1. The application stores the user’s OAuth credentials in local storage for future use [6]. The local storage is used for retaining the credentials so that the user may close the browser or shut down the computer. A background page is required to be run by the extension for the purpose of authentication as the pop-out page was not possible to re-direct to. Twitter passes the OAuth_Verifier to the authorization page, which is essentially a page purely to pass the information on to the background page. The background page is then accessed by the pop-out page to complete the OAuth process.

The extension itself offers users options to find users’ walls or their own timeline, and to encrypt tweets they are posting or to decrypt tweets they have been mentioned in. The buttons have JavaScript code associated to them which makes HTTP requests via Twitter’s JavaScript OAuth library when contacting Twitter or via regular XMLHTTPRequest for communicating with the MIRACL client. The PKG server would be communicated with via HTTP requests to the SSL server hosting it.

4.3 – The MIRACL Library

The MIRACL library is a C++ library for cryptographic purposes. Karl Reid, a postgraduate student who had used this library in his final year project [7], directed me to the software as although the library came with IBE examples, he had successfully implemented an IBE wrapper interface which was very suited to my needs.

The MIRACL library contains both the PKG and the functions to encrypt, decrypt, sign, and verify messages.
4.3.1 – The MIRACL PKG

Setting up the PKG is a simple process that involves simply running a few commands from the terminal.

The command “make all-x64” on 64 bit Linux creates a 64 bit version of the PKG but a 32-bit version of the code also exists. Then “./pkgTool64” is run to start the 64 bit version of the PKG.

Once started the PKG prompts the user to input a username to make a private key for, and asks for the filename that it should output the keypair to. This interface is required to be made accessible to all users who authenticate correctly (see section 4.1).

Although the MIRACL library contains both the client and the PKG there is no security threat as the PKG object created is dependent upon random variables and a timestamp which could not be re-created. This means that every user could potentially set up another IBE system with a new PKG (which would of have different Master Private and Public Keys) for use with other parties who trust them with the Master Key, should they desire to do so.

There is no requirement to distribute the PKG aspect of the library amongst users.

4.3.2 – The MIRACL Client

The MIRACL client is the part of the MIRACL library that would be essential to distribute amongst users. The MIRACL client must be started by terminal by a command of the format

```
./server.sh <identity><keyfile><port>
```

where the keyfile is that which was received from the PKG. For example:

```
./server.sh JamesJudge JamesJudge.keyfile 8080
```

The port may be any acceptable value chosen by the user and is changeable in the Chrome extension also.

The program was accessed by the Chrome extension via http requests to localhost. For example a request to a client running on port 8080 to encrypt a message of “Hello” to TwitterUser would be of the format:
And a similar decrypt request upon a message received of “cipher-text” (note: the actual cipher-text is a very long number with vertical bars in specific locations) would be:

http://localhost:8080/decrypt?cipher-text=cipher-text

There is no user identity specified in the request as you may only attempt to decrypt messages with your own private key which was already used upon initialising the client.

4.4 – The Twitter Server

The Twitter API was initially released in 2006. The API used for the purposes of accessing user’s timelines and posting tweets is the Twitter REST API. This is distinct from the Twitter Streaming API which is used for long lived connections across different architectures, providing “near real-time high-volume access to Tweets in sampled and filtered form” [8]. The REST API server is run by Twitter and we need not concern ourselves with the technicalities of its hosting. We just need to know that it is prone to changes; on the 14th May this year Twitter plans to remove the deprecated methods [9], while the methods used in this project are at present not deprecated there is a possibility certain methods may become deprecated in the future.
In conclusion, for a real world application of this project the general communication for a first time user of IBE would be as follows.

Real World Application

For future use the steps up to the blue dashed line would not need to be repeated. The OAuth steps would generally not need to be re-done unless the user logged out. Communication with the PKG should never need to be repeated unless the user deleted their private key; in such a case it would be repeated and the keypair re-passed to the MIRACL client.
Chapter 5 – Implementation

As there are some aspects to the project which need to be explained to understand how the extension works, this chapter explains some aspects of the code and methods used in the implementation.

5.1 - Compression

After sending a message to be encrypted the MIRACL client returns cipher-text of at least 322 characters back. These characters consist of 11 different symbols, those being the numbers from 0-9 and a vertical bar “|”. As the message and username increase in length so does the cipher-text. A 140 character tweet will reach approximately 660 characters. As this is 4 and a half times the length of the original tweet, this is rather bulky and would flood a user’s timeline with so many separate tweets; therefore a compression technique is used.

For simplicity Lempel-Ziv-Welch (LZW) encoding was implemented through JavaScript. LZW encoding is a lossless compression technique, this means that it decreases the amount of bits required for the data, while still allowing the full data to be recovered. It is widely used and known as a good lossless compression algorithm. LZW makes use of a “dictionary” system, and systematically adds new symbols to the dictionary; this means that any repeated sequences of characters can be compressed. It was used after receiving the cipher-text from the MIRACL client, and it was LZW decoded prior to sending to the MIRACL client for decryption.

After LZW encoding a 30 character tweet was reduced from ~400 characters to about 200 characters. This is an improvement however it is still quite far from perfect.

As there are only 10 unique symbols output by the MIRACL client, this means 4 bits per symbol is sufficient as:

\[
\log_2 11 = 3.459432. \text{ This rounds up to 4 bits.}
\]

4 bits per symbol * 450 average number of symbols = 1800 bits as the absolute minimum number of bits required to transmit the average amount of text data.

As Twitter offers 140 unicode characters, each of which has the potential to hold 2 bytes of data, this means a tweet can contain 2240 bits of data. From
this we can see that there is certainly room for improvement on the LZW algorithm for our purposes, however LZW does perform to a reasonable standard.

As the cipher-text is posted as separate tweets, before sending the extension numbers the tweets (it does this as it splits up the long encoded cipher-tweet), and appends the number to the end of the tweet. Upon receiving the tweets the user may then identify the tweets to decrypt, and the extension will know how to put the cipher-text back together by using the tweet number of each encrypted tweet.

### 5.2 – Manifest

Every Chrome Extension has a manifest file. This contains various details about the extension such as its name and description, but most significantly it names any background pages used and lists permissions for external sites which the extension requires to communicate with such as “http://api.Twitter.com”.

### 5.3 – Background Page

The background page is run the entire time the extension is enabled on Google Chrome. This means that unless you disable the extension, the background page will be constantly running until you close Google Chrome. The background page’s purpose is described in section 4.2 and 2.2; here we will simple demonstrate its use.

The background page is simply a JavaScript script inside an HTML page. For this extension the page contains some variables and some functions. Variables and functions in the Background page can be simply accessed from other pages within that extension by the following.

```
Chrome.extension.getBackgroundPage().my_variable=1
```

Or

```
Chrome.extension.getBackgroundPage().myfunction()
```
5.4 – LocalStorage

So the user does not have to re-authenticate with Twitter every time they re-open their browser, LocalStorage is used to store their OAuth credentials. LocalStorage allows for very easy storing and retrieving of data by the following commands.

```javascript
var localStoreKey="fyp2012"
localStorage.setItem(localStoreKey,Data);
var retrieved_data = localStorage.getItem(localStoreKey)
```

5.5 – XMLHttpRequest

XMLHttpRequest objects were used for communication with the MIRACL client. They were initially planned on being used for the communications with the Twitter API but a JavaScript OAuth library was used instead.

The HTTP request objects are created as follows:

```javascript
var req = new XMLHttpRequest();
```

The GET requests can be created, for example the following decrypt request.

```javascript
Req.open("GET","http://localhost:"+portnum+"/decrypt?cipher-text="+cipher-text,true)
```

After this object is created it can be sent by

```javascript
req.send()
```

Once the request has been sent and replied to, the returned data will be within

```javascript
req.responseText.
```
5.6 – OAuth Library

The “jsOAuth-1.3.3” JavaScript library was used. The OAuth object used to perform requests was created by:

```javascript
Var oauth = OAuth(options);
```

Where the variable “options” contains the callbackURL, consumerKey and consumerSecret for the program. The consumerKey and consumerSecret are related to the Twitter application itself, and are obtained when the application is registered on Twitter.

Further use of the OAuth object for GET requests is of the format:

```javascript
OAuth_object_name.get(url, function_to_perform_on_succeed(), function_to_perform_on_a_failure() )
```

So for example

```javascript
oauth.get(url, function(data){console.log(data)}, function(data){console.log(data)});
```

This expression uses the oauth variable to make a get request to the web address stored in the variable “url”. Upon a successful or unsuccessful response the data returned is logged to the console.
Chapter 6 – Further Work

The following areas are open to further work.

Seamlessness

The extension could potentially be run in-page rather than through a pop-out menu with the encryption/decryption options being set into the Twitter page. The user would be far less aware of the fact that they are using an extension in this case and the ease of use of the application would be better. The pop-out menu would be kept for the user to login/logout and for changing the port which the MIRACL client is run on. For first communication with the PKG the pop-out would also still be required. Automatic decryption of the encrypted tweets while running the application would also improve seamlessness and would be relatively easily introduced by using the time of their post along with the ciphertweet# (described in section 5.2). However it would be time-consuming to implement and could cause problems if the tweets were posted via a bad internet connection and the user posted two separate encrypted tweets within a short amount of time, or if the user posted tweets from different machines simultaneously.

Facebook

If I were given the opportunity to put further work into this project I would like to extend the application to work with Facebook. Facebook also offers an API for developing applications so it is suitable for this means. As a Facebook user I am aware of the fact that my private messages are stored and are plainly readable to anybody in the event that I forget to logout. I would like to make a similar application to this Twitter one for Facebook chat. Users could communicate through the pop-out menu and conversations entered in the chat screen could be encrypted and every encrypted message received would be automatically decrypted.

Facebook uses a different authentication method to Twitter however the underlying principles to writing such an application are essentially the same.
Long Tweets

At present, the encrypted tweet ends up having to be posted as multiple tweets as the JavaScript lossless compression presently in use does not compress the encrypted tweet to less than 140 characters. The application uses an implementation of LZW encoding [12]. It would therefore be useful to find an online text storage service such as Twishort [13] with an API suitable for this. The tweet could be posted on such a site and the link to the held text posted to the recipient Twitter user, such practice is common on Twitter. The recipient’s application could seamlessly retrieve the cipher-text from the hosted site, decrypt it and display it on the pop-out page. The compression algorithm appears to be unsuited to the task as while it does bring the tweet down to a feasible size it is still quite high.

Windows

As Microsoft Windows occupies 86% of the Operating System market share [14] it would greatly increase the market of this application if the MIRACL library could be ported to Windows.

Other Browsers

Versions of this application could also be extended to other internet browsers such as Mozilla Firefox and Internet Explorer. Internet explorer is still the most popular browser worldwide (34.8%), with Chrome being second most (30.87) and Mozilla Firefox being third most widely used (24.98) [15]. If offered for these browsers over 90% of internet users worldwide could use this extension.

Graphical User Interface (GUI)

A very simple GUI for the MIRACL client would make setting up the client much easier for users who are not be comfortable using command line. The GUI would help the user change the keyfile they are using and the port they wish to run the program on.
Online PKG

A public online PKG server could be created for users to use at their own risk, or an SSL certificate could be obtained. This could offer users the option to set up their own groups and members could be added by the group leader. Consequently this online PKG would essentially have different Master Private-Keys for different groups. It could potentially be self-sufficient through advertising funding, or charge members for its use to sustain it.
Chapter 7 - Conclusion

In conclusion the project succeeded its objective of applying Identity Based Encryption to Twitter. In relation to section 1.1, the extension offered a confidential and secure method of communication for Twitter users and used IBE to simplify the encryption process.

As is the nature of such a project there is of course room for improvement. Some of the suggestions outlined in section 6 could be implemented, or indeed might have been implemented had I been familiar with the technologies involved in the project from the beginning.

On a personal level my knowledge of JavaScript, HTTP requests, Google Chrome Extensions, HTML, XML, CSS, JSON, OAuth, encryption, compression, the Ubuntu OS and terminal, and C++ have all greatly improved as a result of this project.
Bibliography


