Arcana: Private tweets on a public microblog platform

Anirudh Narasimman

Submitted to the graduate degree program in Electrical Engineering & Computer Science and the Graduate Faculty of the University of Kansas School of Engineering in partial fulfillment of the requirements for the degree of Master of Science

Thesis Committee:

Dr. Bo Luo: Chairperson

Dr. Luke Huan

Dr. Prasad Kulkarni

Date Defended

The Thesis Committee for Anirudh Narasimman
certifies that this is the approved version of the following
thesis:

Arcana: Private tweets on a public microblog platform

Committee:

Chairperson

Date Approved

Abstract

As one of the world’s most famous online social networks (OSN), Twitter now has 320 million monthly active users. Accompanying the large user group and abundant personal information, users increasingly realize the vulnerability of tweets and have reservations of showing certain tweets to different follower groups, such as colleagues, friends and other followers. However, Twitter does not offer enough privacy protection or access control functions. Users can just set an account as protected, which results in only the user’s followers seeing the tweet. The protected tweet does not appear in the public domain, third party sites and search engines cannot access the tweet. However, a protected account cannot distinguish between different follower groups or users who use multiple accounts. To serve the demand of the user so that they can restrict the access of each tweet to certain follower groups, we propose a browser plug-in system, which utilizes CP-ABE (Ciphertext Policy Attribute based encryption), allowing the user to select followers based on predefined attributes. Through simple installation and pre-setting, the user can encrypt and decrypt tweets conveniently and can avoid the fear of information leakage.

Acknowledgements

First and foremost, I would like to thank Dr.Bo Luo who is my advisor in this project. Professor Luo motivated me immensely during the course of the project. He always suggested new ideas whenever we encountered a bottleneck. He challenged me continuously during the course of the project which made be a better computer scientist. His advice, focus and careful planning helped me immensely. I learnt a lot from Professor Luo during the course of the project and can confidently assert that if it was not for him I would have never been able to achieve anything out of this project.

I would also like to thank the other professors in my committee Dr.Prasad Kulkarni and Dr.Luke Huan for being supportive. I would also like to thank Qiazohi Wang, Satya Narayanan and Ranjith Sompalli for their invaluable advice while implementing parts of the architecture. I would like to thank the EECS department and all the professors who I have worked with for their constant support. I would like to thank my family and friends who have supported me during the course of my education. Lastly I would like to thank my grandmother who has been a great source of inspiration and a big motivating factor in my life.

Contents

List of Figures

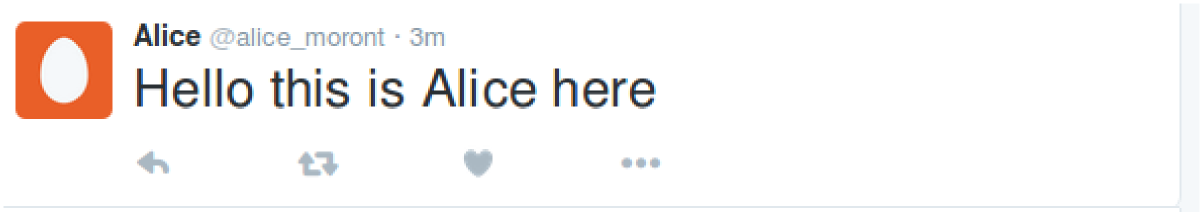
Chapter 1

Introduction and Motivation

1.1 Introduction and Motivation

Communication through a social forum i.e. the internet has evolved exponentially with the introduction of Online Social Networking Sites (OSN).The rapid increase in OSN has resulted in the people communicating and sharing information with other people through OSN rather than face to face communication. As of January 2014, 74% of the internet users use social networking sites [1]. This has increased over the years since it was just 8% of the users as of February 2005 [1].Twitter is one such OSN. Twitter estimates its monthly active users to be close to 310 million. As of September 2014 it is estimated that 23% of the adult population uses Twitter. Twitter poses a unique bottleneck to user in terms of a security perspective. Unlike other OSN such as Facebook any tweet (information) posted by the user is accessible to all other users. This prevents the user from sharing private information or puts user at risk of unwanted followers of his account being privy to information that was originally not intended for them. The user does have an option of sending this information in the form of private message via Twitter, this however would be a very tedious process in which the user every time has to manually select the user group to whom he or she wishes to share the information with Figure 1.1

is an example of data and communication over Twitter. Figure 1.1 is an example of a tweet message that a user (Alice) broadcasts over to the public.

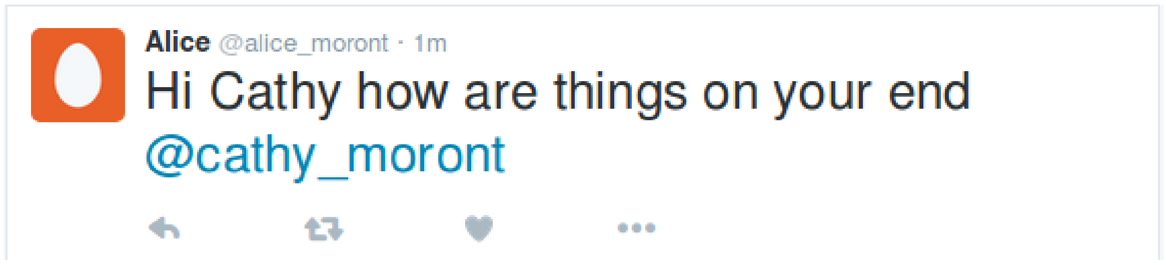


Alice @alice_moront · 3m

Hello this is Alice here

Figure 1.1. Twitter - Tweet Example

Figure 1.2 shows how a private tweet created by Alice which was originally intended for her friend Cathy but is now accessible to someone who is viewing her Twitter account.

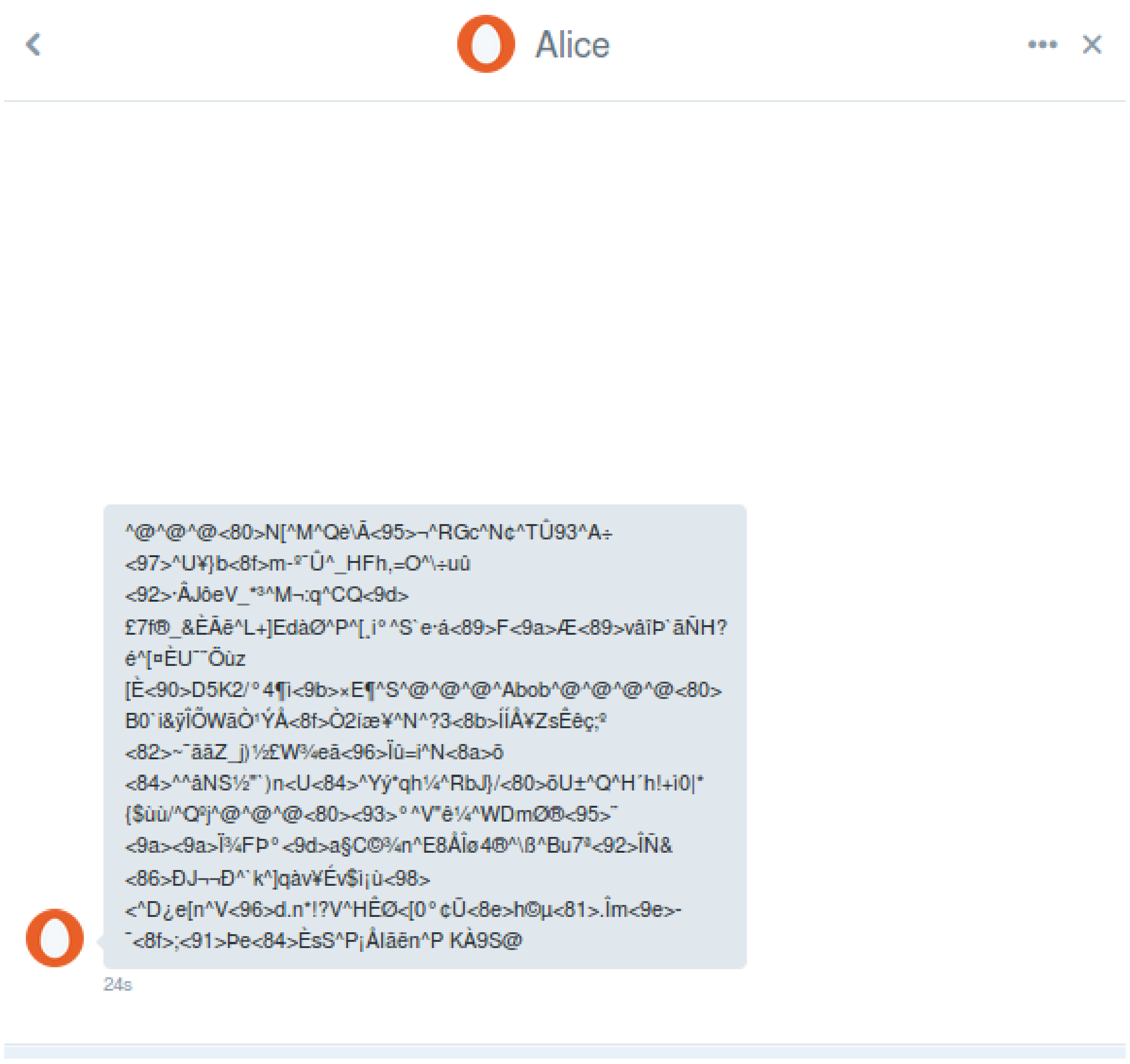


Alice @alice_moront · 1m

Hi Cathy how are things on your end @cathy_moront

Figure 1.2. Twitter – Tweet public example

Figure 1.3 illustrates a private message posted by Alice to Bob. In this example the private key is what Alice has shared with Bob; we will talk more about this later. Observe that every time the user has a new target group in mind he has to manually add all the users who will be privy to that information i.e. every time Alice wishes to share something with a bunch of users apart from Bob where the information she wishes to share with each user is different she then needs to manually send a message to each of these users.



```
^@^@^@<80>N[^M^Qè\Ã<95>¬^RGc^N¢^TÛ93^A÷
<97>^U¥}b<8f>m-º¯Û^_HFh,=O^\÷uû
<92>·ÂJôeV_*³^M¬:q^CQ<9d>
£7f®_&ÈÃë^L+]EdàØ^P^[¸i° ^S`e·á<89>F<9a>Æ<89>vâÏÞ`ãÑH?
é^[¤ÈU¯¯Õùz
[È<90>D5K2/° 4¶i<9b>×E¶^S^@^@^@^Abob^@^@^@^@<80>
B0`i&ÿÎÕWãÒ¹ÝÅ<8f>Ò2iæ¥^N^?3<8b>ÍÍÅ¥ZsÊêç;º
<82>~¯ääZ_j)½£W¾eã<96>Ïû=ï^N<8a>õ
<84>^^âNS½"`)n<U<84>^Yý*qh¼^RbJ}/<80>õU±^Q^H´h!+i0|*
{$ùù/^Qºj^@^@^@<80><93>° ^V"ê¼^WDmØ®<95>¯
<9a><9a>Ï¾FÞ° <9d>a§C©¾n^E8ÅÎø4®^\ß^Bu7ª<92>ÎÑ&
<86>ÐJ¬¬Ð^`k^]qàv¥Év$i¡ù<98>
<^D¿e[n^V<96>d.n*!?V^HÊØ<[0° ¢Ü<8e>h©µ<81>.Îm<9e>-
¯<8f>;<91>Þe<84>ÈsS^P¡Åläën^P KÀ9S@
```

Figure 1.3. Twitter – Private message example

From a security standpoint it is critical that only the intended users can view the information tweeted by the user unless the tweet is actually intended for the entire public. For example, Alice a Twitter user decides to throw a college graduation party and posts the details of the party via a tweet. Assume that the tweet was originally intended for the classmates of Alice. A random stranger viewing Alice’s profile on Twitter can actually view this information. Depending on the sensitivity of the information that the user tweets the consequences can be extremely dire. Mining this public data from Alice also gives third parties more information about her which they can use to their advantage.

For example if Alice is a computer scientist, and a company X who mines Twitter public data realizes that she is graduating from college. The company X is in need of computer scientists so based on the mined information from the tweets and additional user information they start contacting Alice as they feel she might be a good fit. What if Alice already has a job or is not interested in what company X does. Then company X contacting her can be very frustrating for her and a breach of her privacy. The root cause of this problem which Alice probably does not know is the innocent tweet she posted regarding her graduation party.

A cryptographic based solution would be ideal for protecting information in a distributed setting [2]. A perfect scenario would be where the user decides the policy decisions that is who can or cannot see the tweets. There are two issues for encryption while dealing with an OSN such as Twitter. The first is to restrict the information available to third party applications as precisely as possible, so that individual organizations are not entrusted with large volumes of personal information. The second issue is to restrict the information shared with followers who can also be considered as friends of the users to what might be appropriate. Any follower of the user in Twitter is considered as a friend [2]. These two issues as stand-alone problems are not particularly difficult to handle. The challenge lies in the user having the ability to share information to a restricted set of users on an OSN without taking the aid of any trusted third party application. An example of this scenario would be the ability of a user to tweet “called in sick to work” without telling co-workers [2].

Traditionally, this type of expressive access control is enforced by employing a trusted server to store data locally. The server is entrusted as a reference monitor that checks that a user presents proper certification before allowing him to access records or files. However, services are increasingly storing data in a distributed fashion across many servers. Replicating data across several locations has advantages in both performance and reliability. The drawback of this trend is that it is increasingly difficult to guarantee the security of data using traditional methods; when data is stored at several locations, the chances that one of them has been compromised increases dramatically. For these reasons we would like to require that sensitive data is stored in an encrypted form so that it will remain private even if a server is compromised [3]. To provide such functionality without any third party support requires some kind of cryptographic keying.

The underlying idea behind this research is to come up with a simple application that gives the user the ability to have access based control without the use of any third party application. The cryptographic mechanism used here to achieve this is Ciphertext-Policy Attribute-Based Encryption (CPABE). In traditional public key cryptography a message is encrypted for a particular receiver using the receiver's public key [4]. Identity based encryption (IBE) went one step further by generating public key of the user based on some unique identifier of the user. Attribute based encryption (ABE) dove one step further by defining identities not as standalone identifiers but a set of attributes. CPABE is a variant of ABE where the public key is identified by a set of policies. Another challenge that was encountered was how to share the encrypted information on an OSN. If third parties (e.g. Hackers) know that the information is encrypted then the tweet might be subjected to attacks making the system vulnerable. The plug-in (chrome extension) was built keeping all of these issues in mind and was able to handle all of these issues fairly successfully.

Chapter 2

Background and Related Work

2.1 About Twitter

Every OSN has its own convention and terminology for providing communication. The OSN that we deal with in this research is Twitter. Twitter is a popular OSN which has registered 310 million active users [1]. All user accounts are public by default in Twitter. Twitter is indexed with the popular search engine like Google, Bing, making every user is accessible to the public. Twitter communication is primarily through tweets - a word set that can contain up to 140 characters. Users can tweet about any topics, such as trending news or also personal information such as birthdays, anniversaries etc. There is no restriction on the number of tweets a user can post [5]. The user also has the ability to post an image or media. We have used this feature to our advantage while creating the extension.

The following conventions enhance our understanding of Twitter better for this research.

Twitter doesn't have a concept of Friends. "Follow" concept is followed in Twitter, where we just have to follow any person we want. Hence, to map the friends list, we used the intersection list of the followers (people whom the user follows) with followees (people who follow the given user) as friends list.

A tweet starting with RT is called Re-Tweet. This means that the user has re-tweeted someone else's tweet. Most of these tweets are about the general events and happenings.

The "@" - tag is used to tag a user. @ - followed by a user name is the convention used to tag a user to the message. If a user is tagged to a message like "Happy birthday mother", the probability of that user being a mother is high. Hence, we mine this @ - tag information to look for family details. We also ignore the @ - tag because our primary objective is to hide information and the @- tag would make it obvious that the user has some relation with the message tweeted.

Direct Messages are the private side of Twitter. You can use Direct Messages to have private conversations with Twitter users about Tweets and other content. You can start a private conversation or create a group conversation with anyone who follows you. Anyone in a conversation can send Direct Messages to the group. Everyone in a group can see all messages, even if everyone doesn't follow each other. In group conversations, anyone in the conversation can add other participants. Newly added participants won't see the prior history of the conversation. Some accounts, particularly businesses on Twitter, have enabled a setting to receive Direct Messages from anyone. You can send a Direct Message to these users even if they don’t follow you. In both group and one-on-one conversations, you cannot be in a conversation with someone you block [6].

Twitter also has an application management system. The application management system generates particular key value pairs for a particular application. They are Consumer Key (API Key), Consumer Secret (API Secret), Access Token and Access Token Secret. These values are arbitrarily large strings that ensure that only users with these valid credentials can make API calls to the particular Twitter account.

2.2 Preliminaries

2.2.1 Twitter API

In order to collect Twitter data, we used the Twitter open-source API - Tweepy [7]. Tweepy is a python based API that allows us to post tweets or private messages through Twitter. The API class provides access to the entire Twitter RESTful API methods. Each method can accept various parameters and return responses. When we invoke an API method most of the time returned back to us will be a Tweepy model class instance. This will contain the data returned from Twitter which we can then use inside our application. Tweepy tries to make authentication as simple as possible. In order to communicate with the users account in Twitter the user needs to pass Consumer Key (API Key), Consumer Secret (API Secret), Access Token and Access Token Secret to the OAuth handler present in Tweepy. The API has other simple calls to update media and download images.

2.2.2 Python Image Library

The Python Imaging Library (PIL) adds image processing capabilities to your Python interpreter. This library provides extensive file format support, an efficient internal representation, and fairly powerful image processing capabilities. The core image library is designed for fast access to data stored in a few basic pixel formats.

2.2.3 Chrome Extension

An extension is a small program that adds a new dimension to your browser in terms of functionality. The Google Chrome browser is the world’s most popular browser [8]. An extension running on the Chrome browser is a Chrome Extension. Chrome Extensions are written in web technologies such as JavaScript, HTML and CSS. Extensions have little to no user interface. Extensions bundle all their files into a single file that the user downloads and installs. This bundling means that, unlike ordinary web apps, extensions don't need to depend on content from the web [9].

2.2.3.1 Chrome Extension Architecture

An Extension is a bundle of zipped files comprising of HTML, CSS, JavaScript, image files etc. Extensions are essentially web pages, and they can use all the APIs that the browser provides to web pages, from XML

HTTP-Request to JSON to HTML5. Extensions can interact with web pages or servers using content scripts or cross-origin XMLHTTP-Requests. The components of the Chrome Extension we are going to deal with here are the manifest file, HTML, content script and background script [9].

The manifest file, called manifest.json, gives information about the extension, such as the most important files and the capabilities that the extension might use [9]. Shown below is the manifest file that is used for the creation of the extension. Figure 2.1 illustrates an example of a manifest file. What is included in the manifest file includes the name of the extensions. What HTML pages and JavaScript files would the extension interact with i.e. these are the files which are bundled along with the extension. There are a set of permissions and browser action keys in the manifest which set certain permissions and tell the extension how to behave on loading.

Background pages defined by background.html can include JavaScript code that controls the behavior of the extension. There are two types of background pages: persistent background pages, and event pages. Persistent background pages are always open. Event pages are opened and closed as needed. Extensions can contain ordinary HTML pages that display the extension's UI. For example, a browser action can have a popup, which is implemented by an HTML file. Any extension can have an options page, which lets users customize how the extension works [9].

```
{
  // Extension ID: knldjmfmopnpolahpmmgbagdohdnhkik
  "key":
"MIGfMA0GCSqGSIb3DQEBAQUAA4GNADCBiQKBgQDcBHwzDvyBQ6bDppkIs9MP4ksKqCMyXQ/A52JivHZKh4YO/9vJsT3oaYhSpDCE9RPocOE
QvwsHsFReW2nUEc6OLLyoCFFxIb7KkLGsmfakkut/fFdNJYh0xOTbSN8YvLWcqph09XAY2Y/f0AL7vfO1cuCqtkMt8hFrBGWxDdf9CQIDAQAB",
  "name": "Native Messaging Example",
  "version": "1.0",
  "manifest_version": 2,
  "description": "Send a message to a native application.",
   "background": {
    "persistent": false,
    "scripts": ["main.js"]
  },
 "browser_action": {
                      "default_title":"Encryption and Decryption",
                      "default_popup":"main.html"
  },
  "permissions": [
    "nativeMessaging"
  ],
  "content_scripts":[
    {
      "matches": ["https://Twitter.com/*"],
      "js": ["jquery-2.2.3.min.js", "myscript.js"]
    }
  ] }  }
```

Figure 2.1 Extension – Manifest File

Content scripts are JavaScript files that run in the context of web pages. By using the standard Document Object Model (DOM), they can read details of the web pages the browser visits, or make changes to them. The Background script on the other hand can only interact with the extension. Unlike the Content script which interacts with browser the Background script forms a bridge between the HTML (popup HTML) and the intended action of the extension.

2.2.3.2 Native Messaging

The CPABE algorithm was implemented on the system, so we needed a way for the extension to communicate with the algorithm. Extensions and apps can exchange messages with native applications using an API that is similar to the other message passing APIs. Native applications that support this feature must register a native messaging host that knows how to communicate with the extension. Chrome starts the host in a separate process and communicates with it using standard input and standard output streams [10].

In order to register a native messaging host the application must install a manifest file that defines the native messaging host configuration. Figure 2.2 is a manifest file that shows how the native messaging host configuration must be set.

```
{
  "name": "com.google.chrome.example.echo",
  "description": "Chrome Native Messaging API
Example Host",
  "path": "HOST_PATH",
  "type": "stdio",
  "allowed_origins": [
    "chrome-
extension://knldjmfmopnpolahpmmgbagdohdnhkik/"
  ]
}
```

Figure 2.2   Extension – Manifest File II

2.2.4 CPABE Toolkit

The CPABE toolkit provides a set of programs implementing a ciphertext-policy attribute-based encryption scheme. It uses the PBC library for the algebraic operations. The toolkit provides four command line tools used to perform the various operations of the scheme. They are designed for straightforward invocation by larger systems in addition to manual usage. The toolkit contains simple commands for encryption, decryption and key generation.

2.2.5 Webserver

"Web server" can refer to hardware or software, or both of them working together. On the hardware side, a web server is a computer that stores a website's component files (e.g. HTML documents, images, CSS stylesheets, and JavaScript files) and delivers them to the end-user's device. It is connected to the Internet and can be accessed through a domain name. On the software side, a web server includes several parts that control how web user’s access hosted files, at minimum an HTTP server. An HTTP server is a piece of software that understands URLs (web addresses) and HTTP (the protocol your browser uses to view webpages). At the most basic level, whenever a browser needs a file hosted on a web server, the browser requests the file via HTTP. When the request reaches the correct web server (hardware), the HTTP server (software) sends the requested document back, also through HTTP. The webserver that is dealt with here is a static webserver. consists of a computer (hardware) with an HTTP server (software). We call it "static" because the server sends its hosted files "as-is" to your browser. Figure 2.3 shows a simplified diagram of a webserver architecture [18].

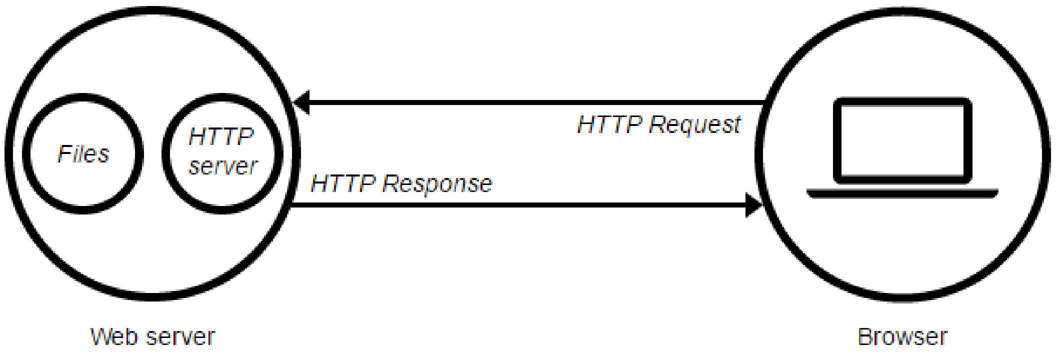


Figure 2.3 Webserver-Architecture

2.3 Related Work

The foundation of the CPABE lies in attribute based encryption which can be thought as a predecessor of CPABE. The need for attribute based encryption aroused when there was a rapid increase in storing of sensitive user data on third party applications through the internet. This made user data highly vulnerable if the third party application was subjected to any kind of attack. One of the solutions to this problem was to store data on these third party applications in an encrypted format. One disadvantage of encrypting data is that it severely limits the ability of users to selectively share their encrypted data at a fine-grained level. Suppose a particular user wants to grant decryption access to a party to all of its Internet traffic logs for all entries on a particular range of dates that had a source IP address from a particular subnet. The user either needs to act as an intermediary and decrypt all relevant entries for the party or must give the party its private decryption key, and thus let it have access to all entries. Neither one of these options is particularly appealing [12].

Sahai and Waters came up with a solution to this problem by introducing Attributed-Based Encryption (ABE). In an ABE system, the keys and cipher texts are labeled with a particular set of descriptive attributes. Decryption only occurs when the attributes of the key match with the cipher text [13]. Vipul Goyal, Omkant Pandey, Sahai and Waters came up with a richer approach where in each cipher text is labeled by the encryptor with a set of descriptive attributes. Each private key is associated with an access structure that specifies which type of cipher texts the key can decrypt [12].

Another paper that was critical while considering the implementation was “Decentralizing Attribute-Based Encryption” by Allison Lewko and Brent Waters. The paper proposed a multi authority based encryption scheme in which control does lies with one particular user but any user who has all the requirements at any given point of time. The major drawback of this simple engineered approach of ABE is that it requires a designated central authority [14]. This authority must be globally trustworthy, since its failure will compromise the entire system. If we aim to build a large or even global scale system, this authority will become a common bottleneck. Spreading a central authority’s keys over several machines to alleviate performance pressures might simultaneously increase the risk of key exposure.

Chase [15] came up with an interesting solution wherein she used the concept of using a global identifier as a “linchpin” for tying users’ keys together. Her system relied on a central authority and was limited to expressing a strict “AND” policy over a pre-determined set of authorities. Therefore a party encrypting would be much more limited than the existing ABE approach. Muller, Katzenbeisser, and Eckert [16, 17] give a different system with a centralized authority that realizes any LSSS access structure.

Allison Lewko and Brent Waters proposed a solution where a new multi-authority Attribute-Based Encryption system is in place. In this system any party could become the main authority and there was no need of any global certification provided initial reference parameters were set. A party can simply act as an authority by creating a public key and issuing private keys to different users that reflect their attributes. Different authorities need not even be aware of each other. They used Chase [15] concept of global identifiers to “link” private keys together that were issued to the same user by different authorities. A user can encrypt data in terms of any boolean formula over attributes issued from any chosen set of authorities [14]. The lack of a central authority improves the overall performance of the system as every process does not need any validation of a central authority. The paper by Allison Lewko and Brent Waters was critical while considering the implementation of the extension as we also wanted to have an ideal situation where each user has the ability to share the encrypted message without the blessing of a centralized authority which would greatly reduce the latency present in the system.

2.3.1 CPABE Overview

The paper by Bethencourt, Sahai and Waters is one of the founding papers for CPABE. The cpabe toolkit which was used as part of the extension was also built based on the conclusions of this paper [3]. Their system at a higher level consisted of a user’s private key being associated with an arbitrary number of attributes expressed as strings. Encryption of a message over a system requires parties to specify an access structure over a set of attributes. A user will only be able to decrypt a cipher text if that user’s attributes pass through the cipher text’s access structure. At a mathematical level, access structures in our system are described by a monotonic “access tree”, where nodes of the access structure are composed of threshold gates and the leaves describe attributes.

The primary drawback of the Sahai-Waters [13] threshold ABE system is that the threshold semantics are not very expressive and therefore are limiting for designing more general systems [3]. Similarly the biggest drawback in Key Policy Attribute Based Encryption (KPABE) [12] was that encryptor exerts no choice on who has access to the data he encrypts except for the choice of what attributes he chooses to encrypt the data. He must put his trust in key issuer hoping that he does not distribute the wrong key. In other words, the “intelligence” is assumed to be with the key issuer, and not the encryptor. In CPABE, the encryptor must be able to intelligently decide who should or should not have access to the data that he encrypts [3].

Another significant milestone when dealing with CPABE is its ability to fend off collusion attacks. If multiple users collude, they should only be able to decrypt a cipher text if at least one of the users could decrypt it on their own. A conventional approach to mitigate this problem was achieved by using a secret-sharing scheme and embedding independently chosen secret shares into each private key. Because of the independence of the randomness used in each invocation of the secret sharing scheme, collusion-resistance followed [13]. In the paper proposed by Bethencourt, Sahai and Waters a novel private key randomization technique that uses a new two-level random masking methodology. This methodology makes use of groups with efficiently computable bilinear maps [3].

2.3.2 CPABE Model Construction

A cipher text-policy attribute based encryption scheme consists of four fundamental algorithms: Setup, Encrypt, KeyGen, and Decrypt. In addition, the implementation allows for the option of a fifth algorithm Delegate [3]. Shown below is the five stage algorithm:

Setup: The setup algorithm takes no input other than the implicit security parameter. It outputs the public parameters PK and a master key MK.

Encrypt(PK, M, A): The encryption algorithm takes as input the public parameters PK, a message M, and an access structure A over the universe of attributes. The algorithm will encrypt M and produce a cipher text CT such that only a user that possesses a set of attributes that satisfies the access structure will be able to decrypt the message. We will assume that the cipher text implicitly contains A.

Key Generation(MK, S): The key generation algorithm takes as input the master key MK and a set of attributes S that describe the key. It outputs a private key SK.

Decrypt(PK, CT, SK): The decryption algorithm takes as input the public parameters PK, a cipher text CT, which contains an access policy A, and a private key SK, which is a private key for a set S of attributes. If the set S of attributes satisfies the access structure A then the algorithm will decrypt the cipher text and return a message M.

Delegate(SK, P). The delegate algorithm takes as input a secret key SK for some set of attributes S and a set P ⊆ S. It output a secret key $\widetilde{\mathrm{SK}}$ for the set of attributes P.

The CPABE toolkit was built on the model construction shown above. Bethencourt, Sahai and Waters have an in-depth mathematical proof on the validity of the process however a discussion on that is beyond the scope of this report.

2.3.3 Similar Approaches

The paper “Persona: An Online Social Network with User-Defined Privacy” had a very interesting approach with regard to policy based encryption. In their model each user generated an asymmetric key pair and distributes the public key to other users whom he wishes to share the information with. The model allows users to choose groups and share information to members of that group. Users control access to personal data by encrypting to “groups.” Restricting data to specific groups allows users to have fine-grained control over access policy, which permits exchanging data with more restrictions. Groups created by one user do not affect the groups that can be created by another [2]. A very similar approach was used when designing the architecture of the extension.

Another paper that is closely related to this discussion is the paper by Jahid, Mittal and Borisov. They came up with a novel way to prevent information leakage during policy based encryption. The architecture design of their system would prevent information leakage in a dynamic group. The use this by using a minimally trusted proxy. The basic idea is, a social contact who wants to decrypt a data, takes a part of the cipher text (CT) to the proxy. The proxy uses its key to transform CT into such a form that contains enough information that an unrevoked user can combine with his secret key mathematically, and successfully perform decryption, whereas a revoked user cannot do so. Upon each key revocation, the user rekeys her proxy with the latest revocation information. The biggest disadvantage in this architecture is that is the possible latency delay while the user tries to communicate with the proxy. The other issue here is that proxy needs to be a trusted source and should not be subjected an attack [19].

To conclude our background research we look into the paper by Lee, Chung and Hwang. The paper talks about the attribute based encryption models with access control schemes over the cloud. The paper judges the encryption scheme on six criterion namely fine-grained access control, data confidentiality, scalability, user accountability, user revocation, and collusion resistant. The paper indicates that CPABE all of the above criterion except scalability i.e. ability of the system to work efficiently when number of authorized users increases. The extension that was developed mitigates this issue as it gives each user at any given point of time complete autonomy on sending encrypted data to its user group i.e. it is not dependent on an external cloud server for validating the authenticity. This greatly improves the scalability of the system and also reduces latency [20].

Chapter 3

The Problem

3.1 Problem Definition

We want to develop a mechanism for encrypting and decrypting data on Twitter such that only the intended users are able to see the data. We also have to ensure that the entire system is decentralized such that every user is granted atomicity with regard to whom he wishes to share the data with. The system implemented must be scalable i.e. not falter if the number of users wishing to encrypt increases. Care should be taken to minimize the latency in the system so that the encryption and decryption process looks almost spontaneous.

3.2 Challenges

The problem definition posted many unique challenges which are listed below.

3.2.1 What encryption mechanism must be used?

This challenge was probably the most fundamental challenge while trying to come up with a solution for the problem statement. So we needed to choose an encryption mechanism. The problem statement states that the information flow is from many to many, i.e. every user decides who he wishes to send the data to. Conventional private key, public key cryptography can be completely ruled out as key management becomes extremely messy. Let us take an example to illustrate this say Jake wishes to share some information via a tweet to Alice, Cathy and Bob. Jake would have to encrypt the message separately with Bob’s public key, Alice’s public key and Cathy’s public key. Moreover he has to retweet the same information three times after encrypting the information with each of their public keys. Jake will also be responsible for the distribution of the private keys initially. If Jake wishes to send a sensitive tweet to N users he would be managing roughly 2N keys such a scenario is clearly undesirable.

Another encryption mechanism that was considered was symmetric key encryption. In symmetric key encryption Jake would encrypt the tweet with a key and send the same key to Alice, Cathy and Bob who would use the key to decrypt the encrypted text. The biggest issue here is that Jake has no way to revoke the key. Assuming that Jake initially sends a the encrypted tweet and symmetric key to Alice, Cathy and Bob but for the next tweet he only wishes to send it to Bob, Alice and Cathy can still decrypt this tweet as Jake cannot revoke their keys. After performing these analyses we realized that classical cryptographic approaches may not be ideal for this problem.

The next obvious choice was to shift to attribute based encryption. ABE despite many glowing advantages needed a middleman for i.e. a designated central authority for all authentications. Apart from this ABE did not offer fine-grained access control and ability to revoke users i.e. remove their key privileges [20]. Keeping these aspects in mind ABE was not considered.

After lot of deliberation CPABE was chosen as the encryption mechanism for solving the problem. CPABE had many advantages namely fine-grained access control, data confidentiality, user accountability, user revocation, and collusion resistance [20]. These advantages made us favor CPABE. The only disadvantage with CPABE is that it is not very scalable. However with carefully designing our system in a particular manner that is described below we can circumvent the problem.

3.2.2 How should the system be designed?

This was the next stage of the problem, we needed a system which requires minimum user effort while encrypting and decrypting tweets. An average user needs you to use the system without really understanding the process behind it. Moreover the user should not go outside the comfort of his workstation to retrieve any decrypted text or send an encrypted text i.e. no call should be made to any central authority. This is an understandable design constraint as no user would like to leave the comfort of his domain to retrieve information for he may feel that the location he or she is routed to might be subjected to an attack of some sorts. Another design paradigm that was imposed was that the system should be extremely light weight on the users system. The idea behind this being users want to access tools which are convenient to use and do not use too much of their system resources.

Keeping all these constraints in mind the research group decided to go with a chrome extension to be the interface that carries out the encryption and decryption process. Chrome extensions are essentially plugins that are run on the client side browser. Extensions are easy to use for the user and as it is run on the browser extremely light on the users system. Another advantage with extensions is that it is extremely interactive, the user has to click buttons rather than type commands to carry out the encryption and decryption process. The content scripts present in the chrome extension have an advantage with regard to storing the decrypted tweet which we will see later. The biggest advantage in my opinion of what the extension offers is that it runs on the browser. Our focus here is to send encrypted information through Twitter. It would be extremely laborious for the user if the system that does the same is stored in some other location of the user's host.

3.2.3 How to tweet encrypted messages?

The next challenge after we decided to use an extension was how do to tweet the encrypted message. This problem posed two unique challenges. The first challenge was we wanted to tweet the encrypted text in Twitter in such a way that any unwanted third party who is viewing the users profile cannot make out whether the tweet is encrypted or not. The second challenge was to minimize the size of the encrypted text. Twitter has a 140 character restriction with regard to the length of a tweet. All encrypted texts which even have singular policies result in encrypted texts of over 140 characters. Another trivial problem was that the encrypted text was in Unicode which Twitter does not support, so the encrypted text needs to be converted to some other format before tweeting.

A basic solution that was suggested was to use some kind of encoding mechanism to encode the encrypted text and then tweet the encoded encrypted text. Base91 was considered for the encoding. Base91 is an advanced method for encoding binary data as ASCII characters. It is similar to Base64, but is more efficient. The overhead produced by Base91 depends on the input data. It amounts at most to 23% (versus 33% for base64) and can range down to 14%, which typically occurs on 0-byte blocks [21]. So we encoded the encrypted text with Base91. This resulted in texts ranging from 720-2000 characters. So if we had to tweet the encoded text we would roughly need a minimum of 6 tweets. Apart from this while decrypting we would have combine these tweets together to decrypt them. There are lot of flaws in this process, first and foremost 6 arbitrary tweets in a weird format with definitely attract the attention of account eavesdroppers who would begin to get curious with what is being conveyed. Another issue which arises is that encrypted text usually is an arbitrary format with large amount of spaces. Using multiple tweets can corrupt the original format of the encrypted text making it impossible to decrypt. Factoring all these inferences the first approach was rejected.

The next approach that was considered was reducing the size of the encoded text to 140 characters by running various compression algorithms and then rephrasing the resultant text into something more meaningful. The smallest size of the encoded tweet is 720 characters, if that has to be converted to 140 characters you need a compression algorithm that runs at 6: 1 ratio. Though there are many powerful compression algorithms they usually work well for structured text data, the encrypted text is unstructured and arbitrary which is what we actually want it to be as random as possible. When we ran various compression algorithms which are suggested in [22] [23] the results obtained were not what was expected. Other popular techniques such as gzip were also implemented but the results obtained were not satisfactory.

Based on the results of the previous attempts we decided to take advantage of hiding the encrypted text in images. An image is essentially a matrix with each pixel corresponding to a particular intensity value. We know any text on the computer is essentially a binary number represented as a character encoding set. What we decided to do was to convert the encrypted text to a binary format and then add to the image matrix. Adding binary values to various pixels would not change its value by much as only 1's and 0's are added. Twitter gives as an option of tweeting media where we can tweet the image containing the encrypted text. This solution solves both of our problems the encrypted text is well hidden preventing eavesdroppers from realizing that the user has sent an encrypted text. Hiding the encrypted text in an image also means that we do not need to worry about 140 character limit present in Twitter. What is even better is that we can encrypt plain text over 140 characters and still send it via Twitter if the above mechanism is used.

Under normal circumstances it would be impossible to send a tweet that is over 140 characters.

3.2.4 How to distribute Keys?

We had to come up with a simple mechanism where in the user does not have to manage multiple keys. Also keys need not be generated multiple times if the sender is communicating with the same user group.

The solution to this problem was that the sender creates private keys for the users he wishes to shares the information with and sends the private keys to them via Twitter private messaging. After this the extension while trying to decrypt private messages uses the key present in the receiver’s message inbox. The sender can easily revoke the key by including something like a timestamp in his policy while encrypting the text. If this timestamp defers from the one in the receiver’s private key then it can be said that particular key of the receiver is not valid and is revoked. Since private messaging in Twitter is encrypted there is no fear of information leakage. Another advantage of using Twitter messaging system is that there is no need of any separate of key management system and all the decryption work can happen on the browser at the extension level.

3.2.5 How to display the decrypted text?

After the encrypted text is retrieved form the image and decrypted with the help of private key the resulting text still needs to be displayed. Storing the decrypted text on the users system is not ideal as there is a possibility of multiple users using the same computer and these users while accessing the computer can look at the user's decrypted text. Another issue is that from a user interface perspective the user would like to see the decrypted text on top of the image which had the encrypted text. This is desired because it makes life easier for the user.

The solution to this problem was achieved by using content script present in the extension. Content scripts give the user the ability to change the HTML page present in the browser locally. What we did was we made the extension's content script pull the decrypted text from the system and display it on top of the intended image in Twitter. By doing this we ensured that no local copy of the decrypted text is maintained and also the user knows what images in Twitter have information that was intended for him. This also improves user experience. Another interesting aspect about content scripts is that because all actions by the script will always have a local consequence the actual image posted by the sender does not undergo any undesired change. This is also a desired trait as we don't the receiver to post the decrypted text back onto Twitter as then it would not be privileged information.

Chapter 4

Architecture

4.1 Overview

Figure 4.1.A and 4.1.B gives the overview of our solution. Shown below is the architectural flow of our system design which is represented by the two figures. In the following discussion we will have an in-depth of each component in the system.

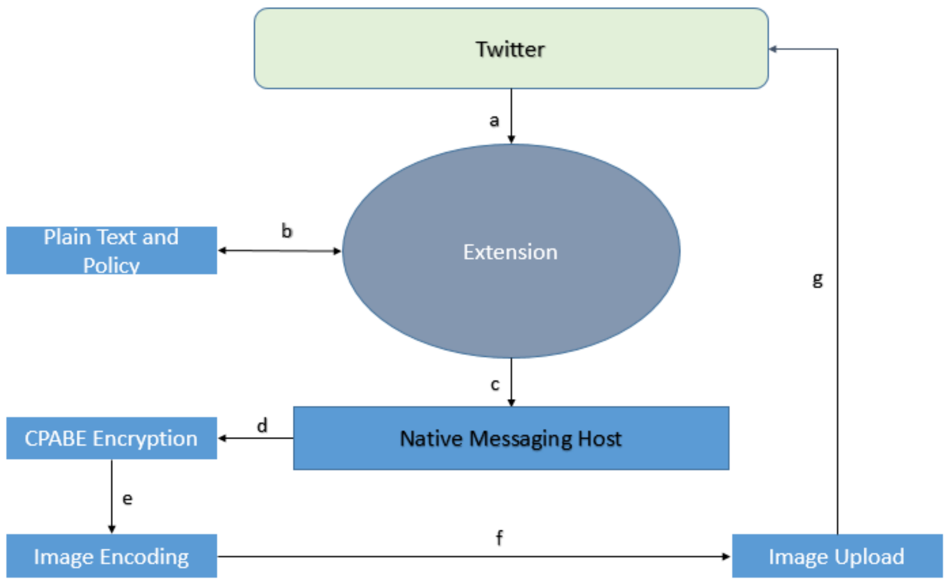


Figure 4.1.A Encryption Algorithm overview

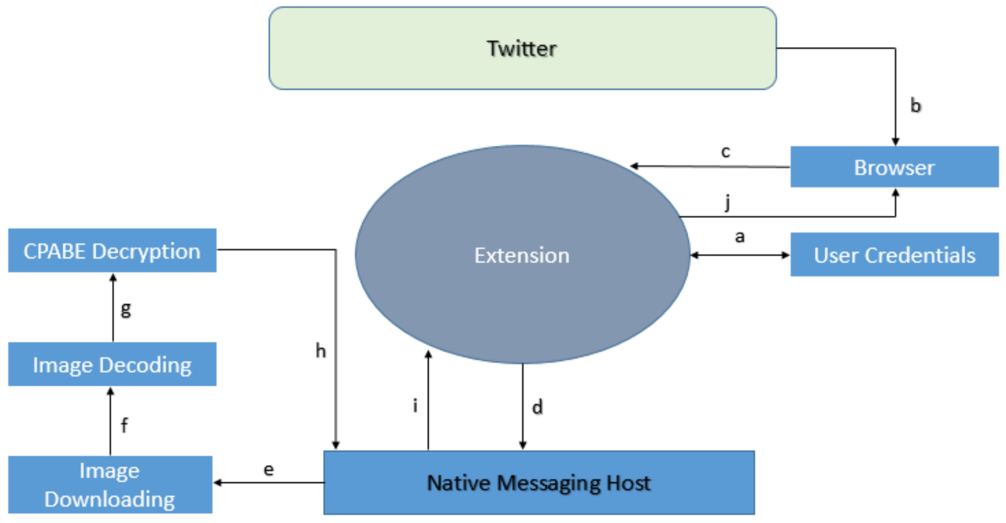


Figure 4.1.B Decryption Algorithm overview

In the following paragraphs we are going to give a quick overview of the encryption and decryption process. The encryption process begins with the user logging into Twitter which is shown in step a in figure 4.1.A. After doing this the user sends the plain text he wishes to encrypt along with a policy of his liking as indicated by step-b. The extension after step b is complete establishes connection with the native messaging host to carry out the encryption process which is indicated by step c. The native messaging host then encrypts the user's plain text along with the policy with the help of CPABE encryption as indicated in step d. The encrypted text is then encoded into an image as indicated in step e. The encoded image is then posted on the users Twitter account which is indicated by steps f and g respectively.

The decryption process overview is indicated in figure 4.1.B. Initially the user has to enter his credentials to the extension to validate he is the desired user for the decryption process as indicated in step a. This is done to guarantee that only the intended user decrypts the message. The user then logs in to his or her Twitter account via the browser as indicated in step b and step c. The extension then establishes connection with the native messaging host as indicated in step d. The native messaging host then downloads all images from the users Twitter feed which may be possible encrypted images as indicated in step e. The images then are decoded to extract the encrypted text as indicated in step f. After this CPABE decryption happens as indicated in step g. The results of the decryption are sent to the extension via the native messaging host as indicated in step h and step i. The extension then posts the result on the browser so that only the user can see this as indicated by step j.

4.2 Chrome Extension Encryption Architecture

4.2.1 User interface - Encryption Process

Shown below in Figure 4.2 is the pop up that appears when the user clicks the chrome extension. The chrome extension consists of a simple pop up HTML which pop's up upon clicking the extension. The pop up HTML in essence is the primary and the only user interface when the user wishes to interact with the extension. The popup HTML also has the background JavaScript present it which does all other actions intended for the extension.

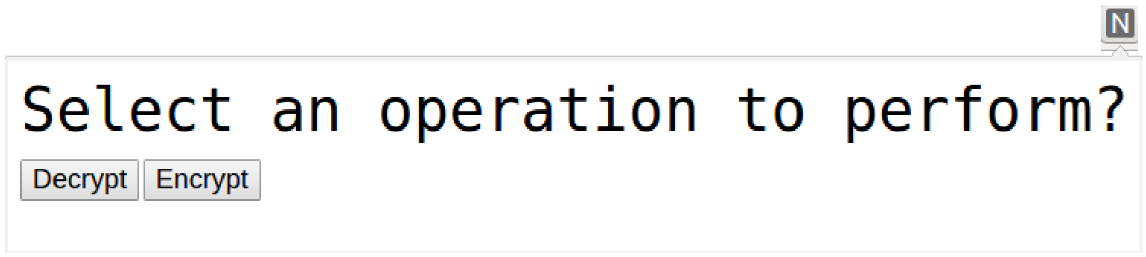
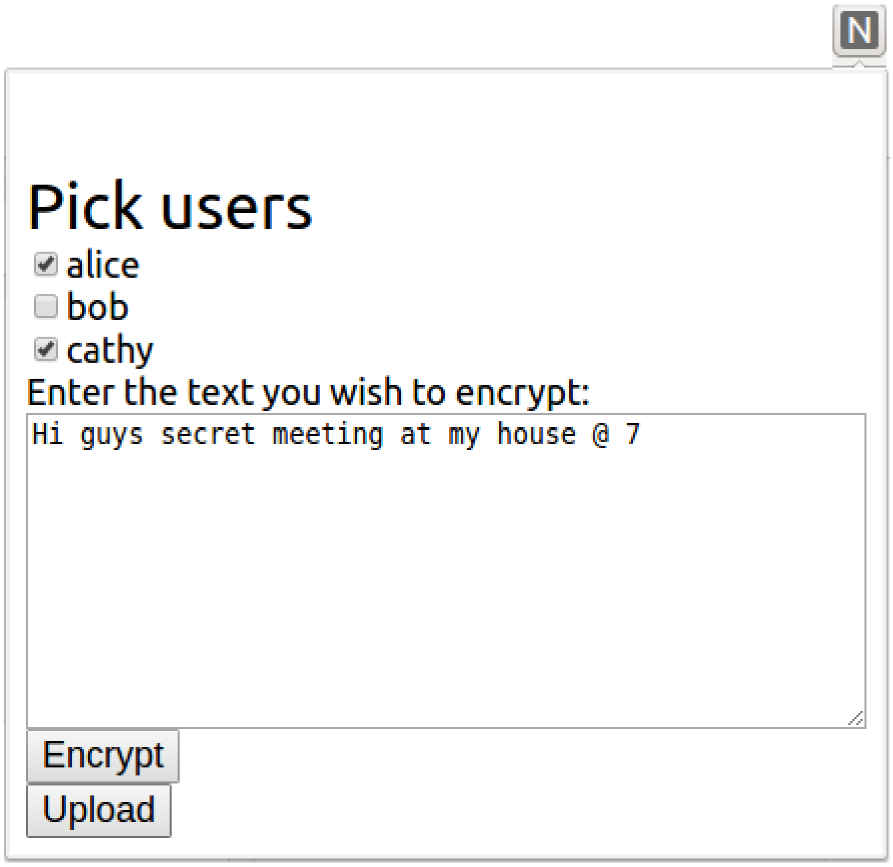

Select an operation to perform?

Figure 4.2. Chrome Extension

The initial HTML popup gives the user two choices one is sending an encrypted text i.e. sending a message via Twitter and the other is decrypting the existing messages. We will discuss both these mechanisms in depth in this chapter.

Upon clicking the encrypt button the screen changes to what is show in Figure 4.3. The interface comprises of three users who the sender can send the information to. The users are Alice, Bob and Cathy. Though ideally the user would like to share this information with a circle of friends we have simplified this to three people for a better user experience. Following that is a textbox where the user can enter the text he wishes to encrypt. Whoever the users chooses for the checkbox are the users who can decrypt the encrypted message. The page also has two buttons, the encrypt button which encrypts the text based on the attributes (check box values) and the upload button which uploads the encrypted image to Twitter. There are JavaScripts that run in the background that help us achieve the same. We will talk about this in detail in the following paragraphs.


Pick users

[✓ alice]
bob
[✓ cathy]

Enter the text you wish to encrypt:

```
Hi guys secret meeting at my house @ 7
```

Figure 4.3. Encryption User Interface

4.2.2 JavaScript – Encryption Process

Once the user hits the encrypt button what actually happens is that the background JavaScript file which is part of the extension is called. What must be observed is that the encryption and decryption algorithm is present in our local host as we make advantage of the CPABE toolkit. JavaScript is a client side tool, which can only run on the browser. It is impossible for JavaScript to access any program on a local host. To circumvent this problem we have taken advantage of native messaging feature present in chrome extension.

Chrome starts each native messaging host in a separate process and communicates with it using standard input (stdin) and standard output (stdout). The same format is used to send messages in both directions: each message is serialized using JSON, UTF-8 encoded and is preceded with 32-bit message length in native byte order. The maximum size of a single message from the native messaging host is 1 MB, mainly to protect Chrome from misbehaving native applications. The maximum size of the message sent to the native messaging host is 4 GB [10].

When a messaging port is created using runtime.connectNative Chrome starts native messaging host process and keeps it running until the port is destroyed. When a message is sent usingruntime.sendNativeMessage, without creating a messaging port, Chrome starts a new native messaging host process for each message. In that case the first message generated by the host process is handled as a response to the original request, then whenruntime.sendNativeMessage is called. All other messages generated by the native messaging host in that case are ignored [10].

Sending and receiving messages to and from a native application is very similar to cross-extension messaging. The main difference is that runtime.connectNative is used instead of runtime.connect, and runtime.sendNativeMessage is used instead of runtime.sendMessage. These methods can only be used if the "nativeMessaging" permission is declared in the extension's manifest file [10].

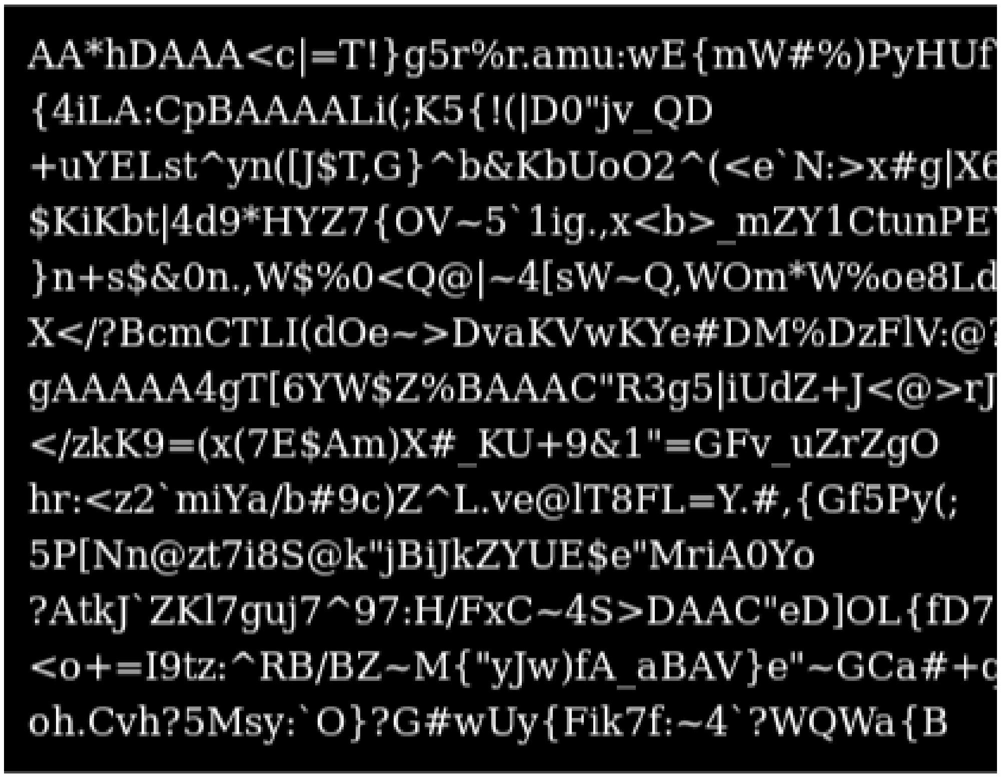
What happens upon clicking the encrypt button is that it initially calls the background.js which in turn downloads the value present in the check boxes and also the text that needs to be encrypted and stores it locally on the local host as a text file. Upon clicking the upload button the encrypted text that is present in the image is uploaded onto Twitter.

4.2.3 Native Messaging Host – Encryption Process

Upon clicking the upload button background.js makes a call to the native messaging host manifest file. The native messaging host manifest file is already preinstalled onto the local host. The manifest file essentially sets the permissions to what programs or script the extension can have access to. The native messaging manifest file points to a bash script. This script reads the text files downloaded by the encrypt button.

4.2.3.1 CPABE Encryption – Encryption Process

After the bashscript reads these files it encrypts the text that needs to be encrypted based on the checkbox values. To do this the bashscript calls the CPABE toolkit that is present and then passes the checkbox values present as parameters and encrypts the text file. The resultant text is then stored as result.cpabe. Figure 4.4 shows a sample output of an encrypted text.



```
AA*hDAAA<c|=T!}g5r%r.amu:wE{mW#%)PyHUf
{4iLA:CpBAAAALi(;K5{!(|D0"jv_QD
+uYELst^yn([J$T,G}^b&KbUoO2^(<e`N:>x#g|X6
$KiKbt|4d9*HYZ7{OV~5`1ig.,x<b>_mZY1CtunPE
}n+s$&0n.,W$%0<Q@|~4[sW~Q,WOm*W%oe8Ld
X</?BcmCTLI(dOe~>DvaKVwKYe#DM%DzFlV:@
gAAAAA4gT[6YW$Z%BAAAC"R3g5|iUdZ+J<@>rJ
</zkK9=(x(7E$Am)X#_KU+9&1"=GFv_uZrZgO
hr:<z2`miYa/b#9c)Z^L.ve@lT8FL=Y.#,{Gf5Py(;
5P[Nn@zt7i8S@k"jBiJkZYUE$e"MriA0Yo
?AtkJ`ZKl7guj7^97:H/FxC~4S>DAAC"eD]OL{fD7
<o+=I9tz:^RB/BZ~M{"yJw)fA_aBAV}e"~GCa#+
oh.Cvh?5Msy:`O}?G#wUy{Fik7f:~4`?WQWa{B
```

Figure 4.4 Encrypted Text

4.2.3.2 Base91 Encoding – Encryption Process

The bashscript then encodes the encrypted text with Base91. The encrypted text is originally in unicode and a lot of systems cannot process unicode due to previous user settings. To simplify matters we converted unicode into a more readable form with the help of Base91. Base91 was chosen because it had a fairly good compression ate when compared to other encoding schemes i.e. in does not make the resultant text too long.

4.2.4 Image Encoding – Encryption Process

The encoded text then needed to be appended to an image. To achieve this python was used. An image was selected by the extension and the image was converted into a binary matrix form with the help of python. Upon achieving this the encrypted text was also converted to a binary format. The encrypted text was then added to the existing image matrix. Upon doing this a binary tag was added to the end of the image to indicate that it is an encrypted image. Shown below in Figure 4.5 are code snippets for the same.

```
binary = str2bin(message) + binary tag
img = Image.open(filename)
img_matrix = img.getdata()
for item in img_matrix:
   if (digit < len(binary)):
      newpix =
encode(rgb2hex(item[0],item[1],item[2]),binar
y[digit])
      if newpix == None:
         newData.append(item)
      else:
         r, g, b = hex2rgb(newpix)
         newData.append((r,g,b,255))
         digit += 1
   else:
      newData.append(item)
img.putdata(newData)
img.save(filename, "PNG")
return "Completed!"
```

Figure 4.5 Image Encryption

4.2.4 Image Upload – Encryption Process

Once the image is encrypted the next stage of the process is to post the encrypted text to Twitter. To do this we utilize Tweepy. A python script was used which uses the senders credentials to post the image onto Twitter. After this is achieved the cached data such as the original plain text and the sender list are removed from the local host. Shown below in Figure 4.6 is a code snippet of the upload process.

```
import sys
import tweepy
import os

CONSUMER_KEY = '****************************'
CONSUMER_SECRET = '****************************'
ACCESS_KEY = ****************************'
ACCESS_SECRET = '****************************'

auth = tweepy.OAuthHandler(CONSUMER_KEY, CONSUMER_SECRET)
auth.secure = True

auth.set_access_token(ACCESS_KEY, ACCESS_SECRET)

# access the Twitter API using tweepy with OAuth

api = tweepy.API(auth)

fn =
os.path.abspath('/home/anirudh/Downloads/host/test.png')

#UpdateStatus of Twitter called with the image file

api.update_with_media(fn)
```

Figure 4.6 Image Upload

Figure 4.7 shows the overall flow of the encryption part of the extension which includes all the processes that are carried out by the bashscript that talks to the native messaging host.

```
FILE="/home/anirudh/Downloads/input.txt"
cd /home/anirudh/Downloads/
#check if the input file exists
if [ -f "$FILE" ];
then
   echo "File $FILE exist."
   cd /home/anirudh/Downloads/
   mkdir ola
   cp input.txt /home/anirudh/CPABE/cpabe
   cp attributes.txt /home/anirudh/CPABE/cpabe
   # removing cached file
   rm input.txt
   rm attributes.txt
   cd /home/anirudh/CPABE/cpabe
   # run a shell file that runs the encryption algorithm
   python shell_run.py
   cp input.txt.cpabe /home/anirudh/Downloads/base91-0.6.0-
linux-i386/bin
   # encode the encrypted text
   cd /home/anirudh/Downloads/base91-0.6.0-linux-i386/bin
   ./base91 -o encrypt.txt input.txt.cpabe
   cp encrypt.txt /home/anirudh/Downloads/host
   rm encrypt.txt
   python --version
   cd /home/anirudh/Downloads/host
   cp /home/anirudh/Downloads/image/test.png
/home/anirudh/Downloads/host
   /usr/bin/python2.7 ./hide.py -e test.png
   python –version
  # upload the tweet
   /usr/bin/python3.4 ./update.py
```

Figure 4.7 Program flow

4.3 Chrome Extension Decryption Architecture

4.3.1 User interface - Decryption Process

The next aspect of the extension process we are going to deal with is the decryption process. The user clicks the decrypt button present in the popup HTML. Upon hitting the decrypt button a screen similar to the one in Figure 4.8 shows up. The page requires you to enter your Twitter credentials so that your private key can be recovered from your messages. The idea of this extra authentication is to guarantee that only the intended user has access to the private key.

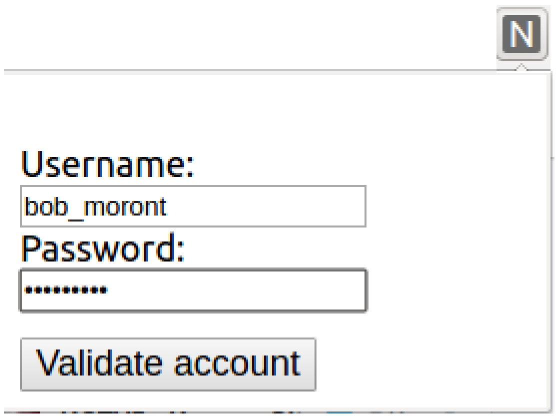


Figure 4.8 Decryption User Interface

4.3.2 Content Script - Decryption Process

After this is achieved all the images in the users Twitter page are downloaded. This is done through the content script. The content script has a function which basically crawls through the page and returns all possible encrypted image candidates. This then is stored in the form a text file on the local host. The text file is basically contains the links of all the images on the users Twitter feed. Figure 4.9 shows a code snippet of the same. The script works as a scroll down function i.e. when the user scrolls down.

```
var s = "";
var count = 0;
$('.stream').find('img').each(function( index ) {
  if(count < 10 &&
$(this).attr('src').indexOf('media') > -1 &&
$(this).attr('src').indexOf('pbs') > -1 &&
$(this).attr('src').indexOf('.png') > -1){
     s += $(this).attr('src') + '\n';
     count++;
}
});
console.log(s);
var textFileAsBlob = new Blob([s],
{type:'text/plain'});
var fileNameToSaveAs = "images.txt";
var downloadLink = document.createElement("a");
downloadLink.download = fileNameToSaveAs;
downloadLink.innerHTML = "Download File";
downloadLink.href =
window.URL.createObjectURL(textFileAsBlob);
downloadLink.click();
```

Figure 4.9 Image Download Links

After this is achieved the next stage is when our content script calls the native messaging bash script. This is achieved through chrome.runtime.sendMessage.

4.3.3 Native Messaging Host - Decryption Process

The method chrome.runtime.sendMessage sends a single message to event listeners within your extension/app or a different extension/app. Similar to runtime.connect but only sends a single message, with an optional response. If sending to your extension, the runtime.onMessage event will be fired in each page, or runtime.onMessageExternal, if a different extension. The extensions cannot send messages to content scripts using this method.

We use chrome.runtime.sendMessage to make an asynchronous call with the background.js file. The background.js file calls the bashscript that does the decryption process for all the images. Figure 4.10 show the same.

```
# In the content script
chrome.runtime.sendMessage({
from:    'content',
subject: 'showPageAction'
});

# In the background script
chrome.runtime.onMessage.addListener(function (msg, sender) {
//alert('Reached here 1');
//alert(msg.from);
//alert(msg.subject);
// First, validate the message's structure
if ((msg.from === 'content') && (msg.subject === 'showPageAction')) {
// Enable the page-action for the requesting tab

//chrome.pageAction.show(sender.tab.id);
//alert('Reached here 2');
chrome.extension.connectNative('com.google.chrome.example.echo');
}
});
```

Figure 4.10 chrome.runtime.sendMessage

Once the native messaging bashscript is called by background.js. It downloads all the images of the users Twitter feed. Figure 4.11 is a code snippet for the same.

```
import urllib
file = open('images.txt', 'r')
count = 0
for line in file:
        m = str(count)
        filename = line[28:47]
        print(filename)
     urllib.urlretrieve(line,
"/home/anirudh/Downloads/img/"+filename)
        count = count + 1
```

Figure 4.11 Image download

4.3.3.1 Image Decoding- Decryption Process

Once the images are downloaded we need to extract the encoded text present in them. What we do is check if every downloaded image has the binary tag that was put in it by the sender. If the binary tag is present then we download the binary contents of the image and convert to a more readable form like ascii. Figure 4.12 shows a code snippet of the same.

```
def retr(filename):
     img = Image.open(filename)
     binary = ''

     if img.mode in ('RGBA'):
          img = img.convert('RGBA')
          datas = img.getdata()

          for item in datas:
               digit = decode(rgb2hex(item[0],item[1],item[2]))
               if digit == None:
                    pass
               else:
                    binary = binary + digit
                    if (binary[-16:] == binary tag):
                         with open('output.txt', 'wb') as filer:
                              filer.write(bin2str(binary[:-16]))
                         return bin2str(binary[:-16])

          return bin2str(binary)
     return "Incorrect Image Mode, Couldn't Retrieve"
```

Figure 4.12 Retrieve Encrypted Text script

4.3.3.2 Image Decryption- Decryption Process

Once decoding is accomplished what is left is to try and decrypt the encrypted text. To do this the bashscript that talks to the extension retrieves the encrypted text from all the images it then tries to decrypt these images with the help of the private key it has downloaded. Figure 4.13 is a code snippet that shows the same. It is important to note that decryption in our case is a two way process which comprises of decoding converting to Unicode from base91 and then decryption.

```
#!/bin/bash
cd /home/anirudh/Downloads/
cp user_login.txt /home/anirudh/CPABE/cpabe
cp images.txt /home/anirudh/Downloads/host/
rm images.txt
rm user_login.txt
cd /home/anirudh/CPABE/cpabe
python private_key.py
cd /home/anirudh/Downloads/host/
/usr/bin/python2.7 download.py
cd /home/anirudh/Downloads/img/
paths="/home/anirudh/Downloads/img/"

count = 0
# Retrieves the encrypted text from the images
for file in /home/anirudh/Downloads/img/*.png
do
    alias python='/usr/bin/python2.7'
    #echo $m
    /usr/bin/python2.7  retreive.py -d $file
    #/usr/bin/python2.7 ola.py -d Cf4ciQKWQAA7bv1.png
done
cd /home/anirudh/Downloads/texter/
for txt in /home/anirudh/Downloads/texter/*.txt
do
              #decrypts the encrypted text
       myfile=$(basename $txt)
       echo $myfile
       #echo ${txt:28}
       cp $txt /home/anirudh/Downloads/base91-0.6.0-linux-i386/bin
       cd /home/anirudh/Downloads/base91-0.6.0-linux-i386/bin
       ./base91 -d -o result.cpabe $txt
       cp result.cpabe /home/anirudh/CPABE/cpabe
       cd /home/anirudh/CPABE/cpabe
       ./cpabe-dec pub_key priv_key_Twitter result.cpabe
       cp result /home/anirudh/Downloads/texter/
        rm result
        rm result.cpabe
       cd /home/anirudh/Downloads/texter/
        rm $txt
       mv result $myfile
done
```

Figure 4.13 Decryption Process

4.3.4 Image mapping - Decryption Process

After decryption is done we need to group the right decrypted text with the correct image this is done by creating a JSON file where each image is mapped to the corresponding decrypted text if it exists. This is illustrated by Figure 4.14.

```
import glob, os,json
os.chdir("/home/anirudh/Downloads/texter
/")
dict = {}
# Get all the encrypted text
for file in glob.glob("*.txt"):
    m = file[0:15]
    print(m)
    first_line = ''
    with open(file, 'r') as f:
        first_line = f.readline()
    print(first_line)
    dict[m] = first_line
print(dict)
#Map image with decrypted text
with open('data.json', 'w') as fp:
    json.dump(dict, fp)
```

Figure 4.14 Decrypted text in JSON Format

4.3.5 Displaying Results - Decryption Process

After mapping is completed now it is a question of our how do we display back this result to the Twitter page, to do this we use the help of the content script again. Before that we create a local webserver to location of our results. This is done by creating a local HTPPS server. The code snippet for the server is shown in Figure 4.15.

```
import BaseHTTPServer,
SimpleHTTPServer
import ssl

httpd =
BaseHTTPServer.HTTPServer(('loca
lhost', 4443),
SimpleHTTPServer.SimpleHTTPReque
stHandler)
httpd.socket = ssl.wrap_socket
(httpd.socket,
certfile='./server.pem',
server_side=True)
httpd.serve_forever()
```

Figure 4.15 HTTPS Server

The content script then makes an AJAX call to server retrieves the decrypted texts and places them on top of their respective images. Figure 4.16 shows a code snippet of the same.

```
# Displays the respective text on top of the decrypted
images
Display_text = function(){
$.ajax({
   url : "https://localhost:4443/data.json",
   type : "GET",
   success : function(result){
          //console.log(result.Cf5rW8yW4AAju4P);
          $.each(result, function(i,j){

     $('.stream').find('img').each(function(index){
               if($(this).attr('src').indexOf(i) > -1){
                    $(this).before(j);
               }
          });
          });
          }
```

Figure 4.16 Content Script to Display Results

In Figure 4.17 we show an example of the same where the user @twitertest23 tweets a set of encrypted images. He then distributes the private keys to Cathy and Bob. When Cathy and Bob try to decrypt the images they can only see those images which there private key can decrypt in other words decryption occurs only if the policy in the encrypted text matches the key.

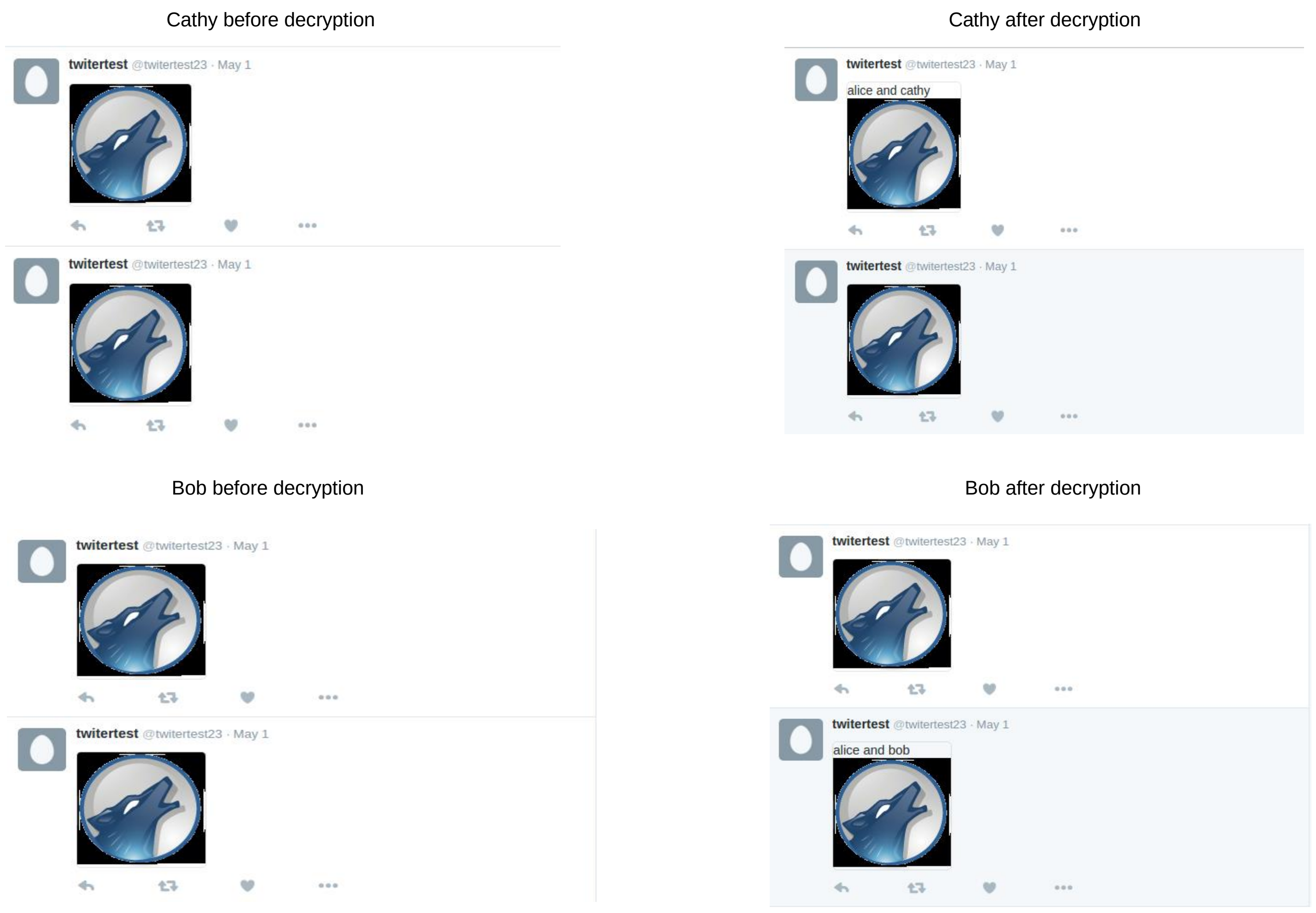

Cathy before decryption
twitertest @twitertest23 · May 1
twitertest @twitertest23 · May 1
Cathy after decryption
twitertest @twitertest23 · May 1
alice and cathy
twitertest @twitertest23 · May 1
Bob before decryption
twitertest @twitertest23 · May 1
twitertest @twitertest23 · May 1
Bob after decryption
twitertest @twitertest23 · May 1
twitertest @twitertest23 · May 1
alice and bob

Figure 4.17 Decryption with two different accounts

Figure 4.17 shows the decryption process for two users Cathy and Bob. The user has encrypted the images in such a way that the first image only Alice and Cathy while the second image only Alice and Bob can see. The top images on the left hand side of Figure 4.16 show Alice and Bob’s Twitter feed before decryption while the ones on the right so their respective Twitter feeds after decryption. Observe that Cathy is only able to decrypt the first image while Bob is only able to decrypt the second image.

Chapter 5

Results

In the following chapter we are going to discuss the results and the performance of the chrome extension. The chrome extension was able to encrypt and decrypt messages with CPABE as the encryption protocol in a fairly efficient manner. The extension is very scalable and can be implemented across multiple systems without any user gaining complete authority. This was tested out by implementing the system across various systems. As every single user is given complete authority during the encryption and decryption process and all the process are done locally at a browser level it is not possible for any user or external system to gain control at any point of time.

The idea of using a chrome extension worked to our advantage. The extension was easy to use for the user and as it is run on the browser it did not consume may system resources. Another advantage with our extension was that it is extremely interactive, the user had to click buttons rather than type commands to carry out the encryption and decryption process. The easy of usability is a critical factor while trying design an encryption platform for an OSN like Twitter.

The use of images rather than text in the tweet gave us an advantage with regard to time complexity. Though traditionally image processing is slower than text processing when we deal with encryption it yields better result. When we deal with encrypted text the tweet limit becomes a serious issue, as Twitter allows only a 140 character limit for any tweet. This would have led us to have close to 6 tweets even for very small encrypted messages. The time taken to group these encrypted tweets would be extremely large, given that encrypted texts have random spacing, moreover even while grouping these encrypted texts lot of things can go wrong which would result in the decryption algorithm breaking, this is highly undesirable so the shift towards an image based encryption approach worked in our favor.

Another aspect that was a success with regard to the extension was key distribution. The sender creating the private keys for the users he wishes to share the information with and then distributing the private keys via Twitter private messaging resulted in no need of any separate of key management system and all the decryption work can happen on the browser at the extension level. After this the extension while trying to decrypt private messages used the key present in the receiver’s message inbox. The sender was easily able to revoke the key by including something like a timestamp in his policy while encrypting the text. The timestamp deferring from the one in the receiver’s private key resulted in the key being revoked. Since private messaging in Twitter is encrypted no additional encryption was needed. Ideally in a plugin the policy would be entered in a textbox rather than checkboxes. But we have included textboxes for easy of usability. This implies that the user should include the timestamp in as part of the plain text that needs to be encrypted. The plugin initially does a quick comparison of the timestamp with the existing timestamp of the user’s private key and they match then the decryption process begins. This variation of the implementation was carried out for ease of user usability.

Another advantage with the system implemented is that it cannot have encoding issues. All texts after encryption are encoded to Base91, this guarantees that any system would be able to pick the encrypted text and read it. Otherwise what might have happen is certain systems might only support certain type of encoding this regarding other types. This reduces any conflict in case the user has set the default encoding to a particular format.

Storing the decrypted text on the users system is not ideal as there is a possibility of multiple users using the same computer and these users while accessing the computer can look at the user’s decrypted text. The extension ensures that this does not take place by using the help of content scripts. As discussed earlier in the previous chapter what the extension does is that it decrypts the text for a group of images which it downloads from the user Twitter feed. It then creates JSON file with the image and decrypted text as the corresponding key value pair. After that the content script makes an AJAX call with the localhost and pulls data from the JSON file. It then does a one to one mapping where the decrypted text if it exist is placed on top of the image which had its original encrypted text. To make sure that the AJAX call is not made before all the images are decrypted we introduce a delay in the content script before the AJAX call is made. This delay is essentially the time taken by the extension to carry out the decryption. The AJAX call is made when the user scrolls down his Twitter page.

Figure 5.1 is a graph that shows the encryption and decryption process based on the number of images crawled from the users Twitter feed. The time taken for the encryption and decryption process will obviously change also based on the size of the plaint text that is encrypted. Images that have the binary tag will not be decrypted as far as the plugin is considered they are plain images, but however these images will still be downloaded.

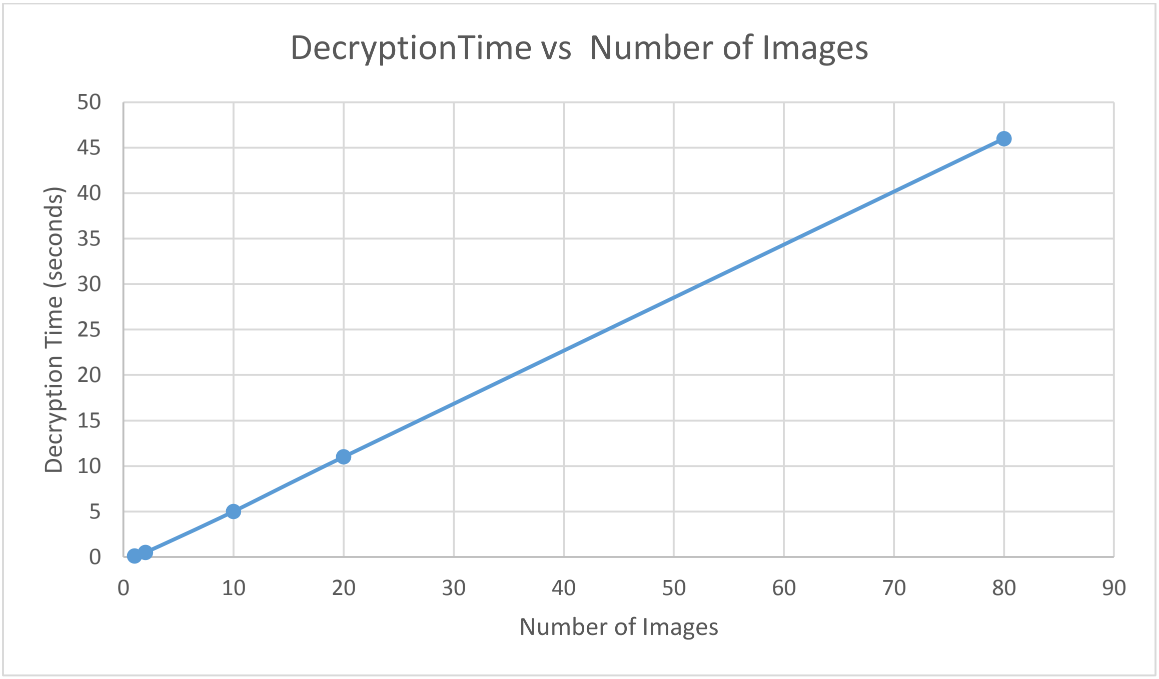


Figure 5.1 Encryption Time vs Number of Images

We observe that the encryption time linearly increases if the extension has to decrypt more images. This is what we would conclude as the time taken to decrypt the image is nearly a constant, and so if the number of images increases we expect a linear increase in encryption time.

Another interesting capability of the extension is that now using it we can send text of over 140 characters via Twitter. For very large amount of characters the size of the image needs to be increased so that it can handle the additional textual data. Figure 5.2 and Figure 5.3 shows the time taken to encrypt the plain text and the time taken to append the encrypted text to an image for plain texts of various sizes.

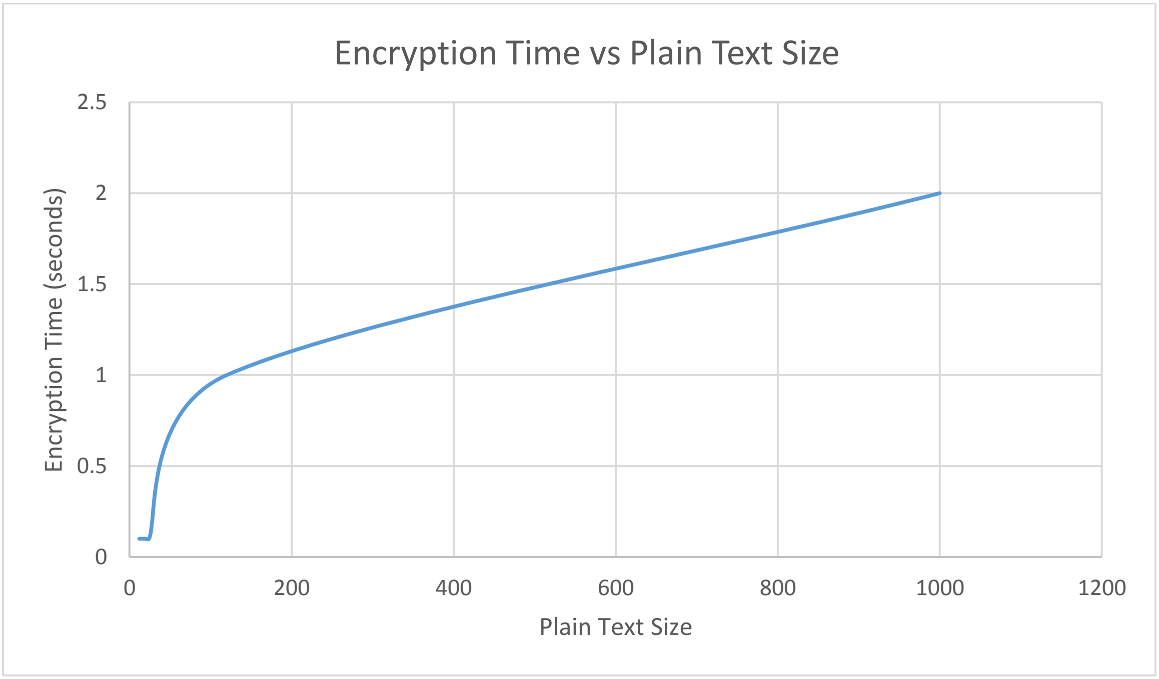


Figure 5.2 Encryption Time vs Plain Text Size

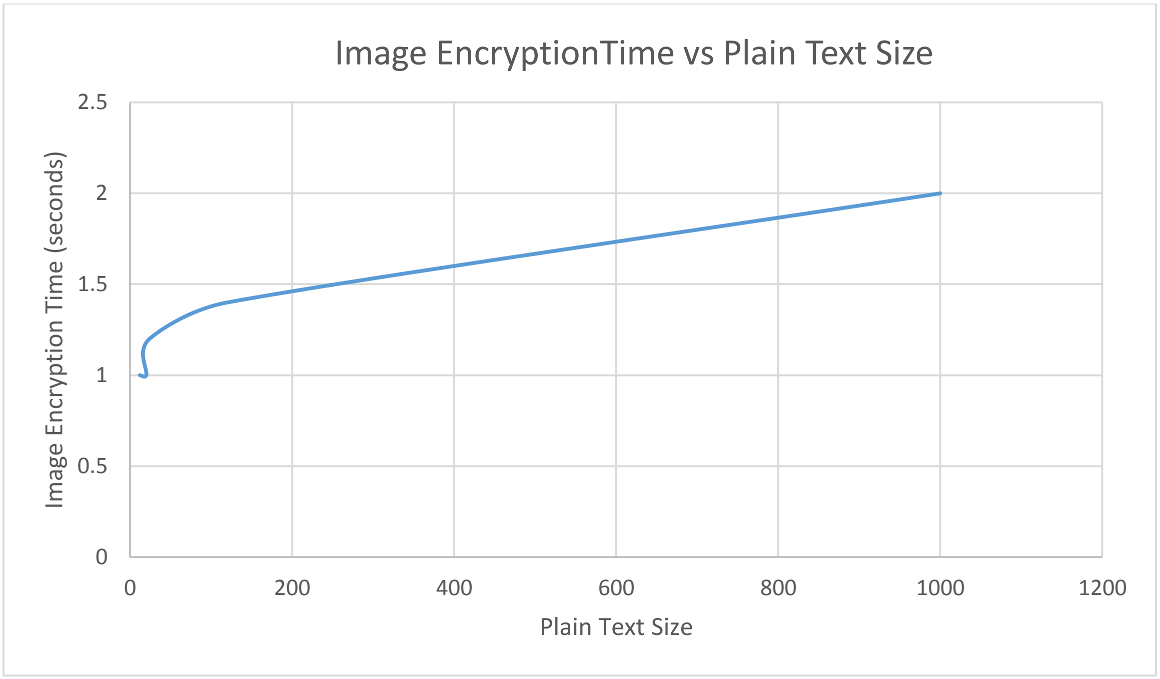


Figure 5.3 Image Encryption Time vs Plain Text Size

As we can see from Figures 5.2 and 5.3 the encryption time and the image encryption time increase if the size of the plain text increases. It nearly is a constant for plain texts of a particular size but as the size of the plain text increases to larger values the encryption time of image as well as the plain text increases. This is what is expected as the encrypted text size increase more columns in the image matrix need to be accessed this increase the encryption time of the image. With regard to the encryption of the plain text lot of other factors play in such as your public key size, policy size etc.

Chapter 6

Conclusion and Future Work

We were able to develop a mechanism for encrypting and decrypting data on Twitter in a way that only the intended users is able to see the data. Each user is granted complete atomicity with whom he wishes to share the data. The system implemented is scalable.

We have also taken care to minimize the latency in the system so from a user's perspective things look nearly spontaneous. The biggest take away from the system design is that most of the work of the system design is carried out by the browser extension. Novice users who have no prior knowledge with command line interfaces and other command line technologies only need to use the extension for encryption and decryption. Most of the conditions of our problem statement were met by implementing a thorough architectural design.

To make this project more complete in the future what a user can do is try to implement the CPABE toolkit in JavaScript rather than using its current form as a C based application. By doing this we can bring down latency to nearly zero seconds as all process will be carried out by the browser. Another extension that can be added to the application is to give the user an option by which he can send the encrypted image to a group of users of a particular type rather than selected users. By this what we mean is the user has the ability to send it to all people working in his company rather than handpicking the users via the checkbox. Another extension can be that once an image has been decrypted the plugin can prevent the user from performing any other action on that tweet. This prevents the user from accidentally sharing the decrypted text with any unwanted people.

References

[1] Social Networking Fact Sheet Washington DC: Pew Internet and American Life, 2013.

[2] Baden, Randy, et al. "Persona: an online social network with user-defined privacy." ACM SIGCOMM Computer Communication Review. Vol. 39. No. 4. ACM, 2009.

[3] Bethencourt, John, Amit Sahai, and Brent Waters. "Ciphertext-policy attribute-based encryption." Security and Privacy, 2007. SP'07. IEEE Symposium on. IEEE, 2007.

[4] Paterson, Kenneth G., and Geraint Price. "A comparison between traditional public key infrastructures and identity-based cryptography." Information Security Technical Report 8.3 (2003): 57-72.

[5] Gopal, Jamuna. I Know Your Family: A Hybrid Information Retrieval Approach to Extract Family Information from Microblogs. Diss. University of Kansas, 2014.

[6] Twitter, https://support.Twitter.com/, Twitter support pages

[7] Twitter Open-Source – API, http://docs.tweepy.org/en/v3.5.0/api.html

[8] James Timcomb, Google Chrome passes Internet Explorer as most popular browser to end Microsoft's 18-year reign, Telegraph, May, 2016.

[9] Google, Chrome Developers, https://developer.chrome.com/extensions

[10] Google, Chrome Developers, https://developer.chrome.com/extensions

[11] Bethencourt, J., A. Sahai, and B. Waters. "Advanced crypto software collection: the cpabe toolkit." 2011-04-01). http://acsc.cs.utexas.edu/cpabe (2011).

[12] Goyal, Vipul, et al. "Attribute-based encryption for fine-grained access control of encrypted data." Proceedings of the 13th ACM conference on Computer and communications security. ACM, 2006.

[13] Sahai, Amit, and Brent Waters. "Fuzzy identity-based encryption." Advances in Cryptology–EUROCRYPT 2005. Springer Berlin Heidelberg, 2005. 457-473.

[14] Lewko, Allison, and Brent Waters. "Decentralizing attribute-based encryption." Advances in Cryptology–EUROCRYPT 2011. Springer Berlin Heidelberg, 2011. 568-588.

[15] Chase, Melissa. "Multi-authority attribute based encryption." Theory of cryptography. Springer Berlin Heidelberg, 2007. 515-534.

[16] Müller, Sascha, Stefan Katzenbeisser, and Claudia Eckert. "Distributed attribute-based encryption." Information Security and Cryptology–ICISC 2008. Springer Berlin Heidelberg, 2008. 20-36.

[17] Müller, Sascha, Stefan Katzenbeisser, and Claudia Eckert. "Distributed attribute-based encryption." Information Security and Cryptology–ICISC 2008. Springer Berlin Heidelberg, 2008. 20-36.

[18] Mozilla, Mozilla Developer Network, https://developer.mozilla.org/en-US/Learn

[19] Jahid, Sonia, Prateek Mittal, and Nikita Borisov. "EASiER: Encryption-based access control in social networks with efficient revocation." Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security. ACM, 2011.

[20] Lee, Cheng-Chi, Pei-Shan Chung, and Min-Shiang Hwang. "A Survey on Attribute-based Encryption Schemes of Access Control in Cloud Environments." IJ Network Security 15.4 (2013): 231-240.

[21] Base91-encoding Open-Source API, http://base91.sourceforge.net/

[22] Burrows, Michael, and David Wheeler. "A block-sorting lossless data compression algorithm." DIGITAL SRC RESEARCH REPORT. 1994.

[23] Dheemanth, H. N. "LZW Data Compression." Volume 3 Issue 2–February 2014: 22.