A-LEVEL COMPUTER SCIENCE  
PROJECT

End-To-End Encrypted Chat Application

Matthew TRANMER (SIX 06 - WS)

28/06/2022

Candidate Number: 3251

Centre Name: Wolfreton School and Sixth Form Collage

Centre Number: 44207

Qualification Code:

# Contents

[Contents 1](#_Toc109774121)

[Analysis 2](#_Toc109774122)

[Introduction and Problem Definition 2](#_Toc109774123)

[Description of Current System 3](#_Toc109774124)

[Examples of similar products 4](#_Toc109774125)

[SMS 4](#_Toc109774126)

[WhatsApp 5](#_Toc109774127)

[End Users 5](#_Toc109774128)

[Interview with Mr Paffley 5](#_Toc109774129)

[Key Algorithms 6](#_Toc109774130)

[The Signal Protocol 6](#_Toc109774131)

[The Double ratchet algorithm 6](#_Toc109774132)

[Diffie Hellman 8](#_Toc109774133)

[Elliptic Curve Digital Signature Algorithm 12](#_Toc109774134)

[Extended Triple Diffie Hellman 13](#_Toc109774135)

[Objectives 15](#_Toc109774136)

[Objective 1 15](#_Toc109774137)

[Objective 2 15](#_Toc109774138)

[Objective 3 16](#_Toc109774139)

[Server Architecture Considerations 16](#_Toc109774140)

[Monolithic Architecture 16](#_Toc109774141)

[Microservice Architecture 17](#_Toc109774142)

[Architecture Conclusion 17](#_Toc109774143)

[Other Considerations 17](#_Toc109774144)

[Programming Language 17](#_Toc109774145)

[Database Software 18](#_Toc109774146)

[Modelling 18](#_Toc109774147)

[Login System 18](#_Toc109774148)

[Data Flow Diagram - Level 0 18](#_Toc109774149)

[Data Flow Diagram - Level 1 19](#_Toc109774150)

[Sending the first message 20](#_Toc109774151)

[Data Flow Diagram - Level 0 20](#_Toc109774152)

[Data Flow Diagram - Level 1 20](#_Toc109774153)

[Design 21](#_Toc109774154)

[Database – Objective x 21](#_Toc109774155)

[Explanation Of Each Table 22](#_Toc109774156)

[API design – objective x 24](#_Toc109774157)

[Overview 24](#_Toc109774158)

[API Endpoints 24](#_Toc109774159)

[Key Algorithms Design – objective x 26](#_Toc109774160)

[Elliptic Curve Functions 26](#_Toc109774161)

[Cryptographic Functions 27](#_Toc109774162)

[Merkle Damgard Construction Based Hash Function 28](#_Toc109774163)

[End To End Encryption Algorithms 29](#_Toc109774164)

# Analysis

## Introduction and Problem Definition

I will be developing an end-to-end encrypted chat application. This application will allow people to communicate securely over a network using an easy-to-use graphical user interface.

For the system to be end-to-end encrypted, the main message encryption needs to happen client-side only so that the server cannot decode the messages being sent, meaning that the server only acts as a message relay and data storage service. The data stored on the server should be useless to anyone that views it that is not authenticated to see the stored data, meaning if we have a data breach, the attackers will get no plaintext information, other than data stored in plaintext such as usernames. Data stored in plaintext will be public information and not sensitive. In the key algorithms section, I will describe the algorithms that will be used in order to provide end to end encryption.

The system will need to contain an account system to authenticate users, meaning I will need to take care in storing sensitive information like passwords. I will also need to make sure that user’s passwords are long, secure and are hard to break because breaking the user’s password will allow the hacker to read old messages and impersonate the user.

Security measures will be put in place to attempt to stop malicious actors from intercepting messages, performing man in the middle attacks, or engineering someone into believing that they are chatting with someone else.

The password will be used to derive two constants: an authentication code and an encryption key (known as the secret key). The secret key will never be sent to the server. The authentication code will be sent to the server, hashed and then saved in the database. The authentication code is an extra layer of security so if somebody finds the private keys then they won’t be able to impersonate the user because they don’t have the authentication code. The secret key will be used to encrypt all of the secrets that will be stored on the server. For example: the private keys and the old messages will be encrypted with the secret key so that the client does not need to keep a record of these values and so that the client can access their Messager from any client. If we didn’t do this, we would have to use the password as the authentication code and the encryption key, and if the server was impersonated, the server would be able to read our password in plaintext and then decode all of the sensitive data encrypted on the server. There will be no way to get the password from the authentication code or the secret key, and there will be no way to get the secret key from the authentication code.

Unfortunately, If the user forgets their password, they will be locked out of their account forever because their account data is encrypted using a key derived from the password. Users will still be able to delete their account if they have access to the email used to sign up, and if they can answer their security questions. Users will also be able to change their password, given that they know their current password.

When the finished solution is produced. Users should be able to communicate sensitive data over the network with full trust that the system will ensure complete privacy between the communicating parties.

## Description of Current System

Most widely use communication methods used do have encryption, but do not have end-to-end encryption. This means that the messages are encrypted, sent to the server, decrypted on the server, re-encrypted with the receiver’s key and sent to the receiver, creating a security vulnerability. Examples of these vulnerable communication methods are email and SMS.

If end-to-end encryption is not used, then service providers are able to read your messages in plaintext, and so are hackers. This makes it impossible to have a private conversation and makes it dangerous for people to communicate sensitive information using these services.

At Wolfreton school, we use email for communication between staff and students. I believe that messages between staff and students should not be end-to-end encrypted because of safeguarding issues – a system should be in place for inspecting messages and the staff and students should be warned that this is in place. Whilst these messages shouldn’t be end-to-end encrypted, messages between staff should be. Any communication of sensitive information, such as student records, exam results, or internal organization communication should be end-to-end encrypted.

Other end-to-end encrypted chat applications like WhatsApp and Signal require a phone number to use the application, and they require mobile phones to access the desktop application. This wouldn’t be suitable for a school environment where phones are not allowed to be used and aren’t seen as professional.

## Examples of similar products

### SMS



SMS is the system behind text messages on mobile phones and it stands for short message service. SMS is a worldwide standard allowing people with mobile phones to communicate through text. The problem with this system is that it is unencrypted and sent in plaintext.

SMS is open to attacks such as SIM swapping, where social engineering techniques are used to get mobile phone providers to port someone’s SIM card information to another phone, and interception where attackers inspect the packets being sent to the cell tower and read the data being sent in plaintext. Also, fake cell towers can be made to perform a man in the middle attack.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| International standard that most people are already familiar with | Unencrypted message communication |
|  | Susceptible to man in the middle attacks |
|  | Susceptible to SIM swapping. |

### WhatsApp



WhatsApp is a proprietary end to end encrypted messaging app that supposedly implements the signal protocol. Because WhatsApp is closed source, we do not know if it is actually end-to-end encrypted, we just hope that it is because meta (formerly Facebook) says that it is.

WhatsApp requires a phone to be used because if you go to WhatsApp web, you need to scan a barcode in the WhatsApp app on your phone.

|  |  |
| --- | --- |
| Advantages | Disadvantages |
| End-To-End encrypted | Do not know if it is end-to-end encrypted with 100% certainty |
| Popular | Requires a mobile phone |

## End Users

I would like my project to be used by the staff at Wolfreton school for internal communication. Currently, teachers use email for communicating GDPR protected sensitive information such as medical information.

To gather more information, I would like to setup an interview with Mr Paffley, who is the Lead Teacher for ICT at Wolfreton School, and a School Governor. This would help me understand more about what type of data is being communicated internally and how someone like Mr Paffley would benefit from using this system.

### Interview with Mr Paffley

**What program/system do you use to communicate with other members of staff and why was this system chosen?**

We use email as the main form of communication, sometimes using Teams to do so (i.e. chat or posts).

This is used as firstly it is a legacy system that everyone is familiar with. It is accessible on a range of platforms. It allows for a record of conversations as well as creating folders or marking messages to highlight them as required.

It is also useful to send messages, files and links. It integrates with OneDrive.

**Are there any problems with this system?**

High volumes of messages can be difficult to manage & messages can get lost in a large inbox. Junk mail is also an issue.

**Can you give some examples of when you would need to communicate with other members of staff?**

E.g. to send reminders, information about exams or coursework, share resources, updates on policy, to discuss specific issues as the arise.

**Do you communicate any sensitive information using this system?**

Yes, some messages will be sensitive e.g. containing information about a data subject or personal data.

**Are there any security risks with this system?**

The main risks are standard to Microsoft products as we use Office.com. However, there is a risk of Phishing, shoulder surfing or accidently putting emails onto a large screen in the classroom. A stolen mobile phone could be a risk. Also, an issue with sending an email to the wrong person.

**How would you improve the current system?**

It would be good to be able to filter conversions and set reminders for messages. Also, a way to check who it is being sent to so an error in recipient is less likely (i.e. differentiate between a student and a teacher).

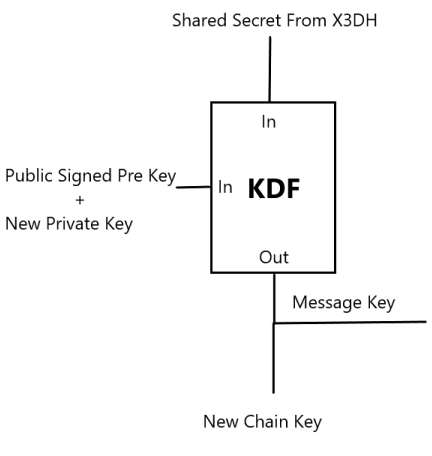
## Key Algorithms

### The Signal Protocol

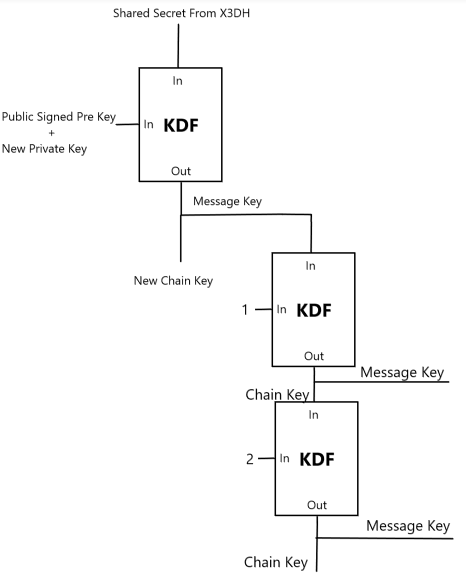
The signal protocol describes two algorithms/protocols that can be used to provide end-to-end encrypted communications: the ‘extended triple Diffie-Hellman key agreement protocol’ (X3DH for short) and the ‘double ratchet algorithm’. Each implementation of these algorithms builds on top of different algorithms, but the key points are that the X3DH protocol is a set of instructions on how to generate a shared secret value between two parties, with messages being relayed over an unsecure connection through a server, and the double ratchet algorithm generates encryption keys for each message.

### The Double ratchet algorithm

The double ratchet algorithm takes an input value that is used to derive all of the keys needed to encrypt each message. This input value needs to be secret and is obtained through the use of the X3DH protocol. The double ratchet algorithm creates a key derivation function (KDF) chain which takes a starting value, introduces an input variable to change the data then produces a chain key and a message key. When someone wants to start a conversation with another user, they perform a key exchange using the X3DH protocol and use this key as the root key. They then generate a key pair and combine the private component with the public signed pre-key of the other user using a Diffie Hellman (the public signed pre-key is given to the server beforehand) which gives a new value which is given as the input to the key derivation function.



The message key is then used the derive a new KDF chain with the inputs the message number so the full chain ends up looking like this:

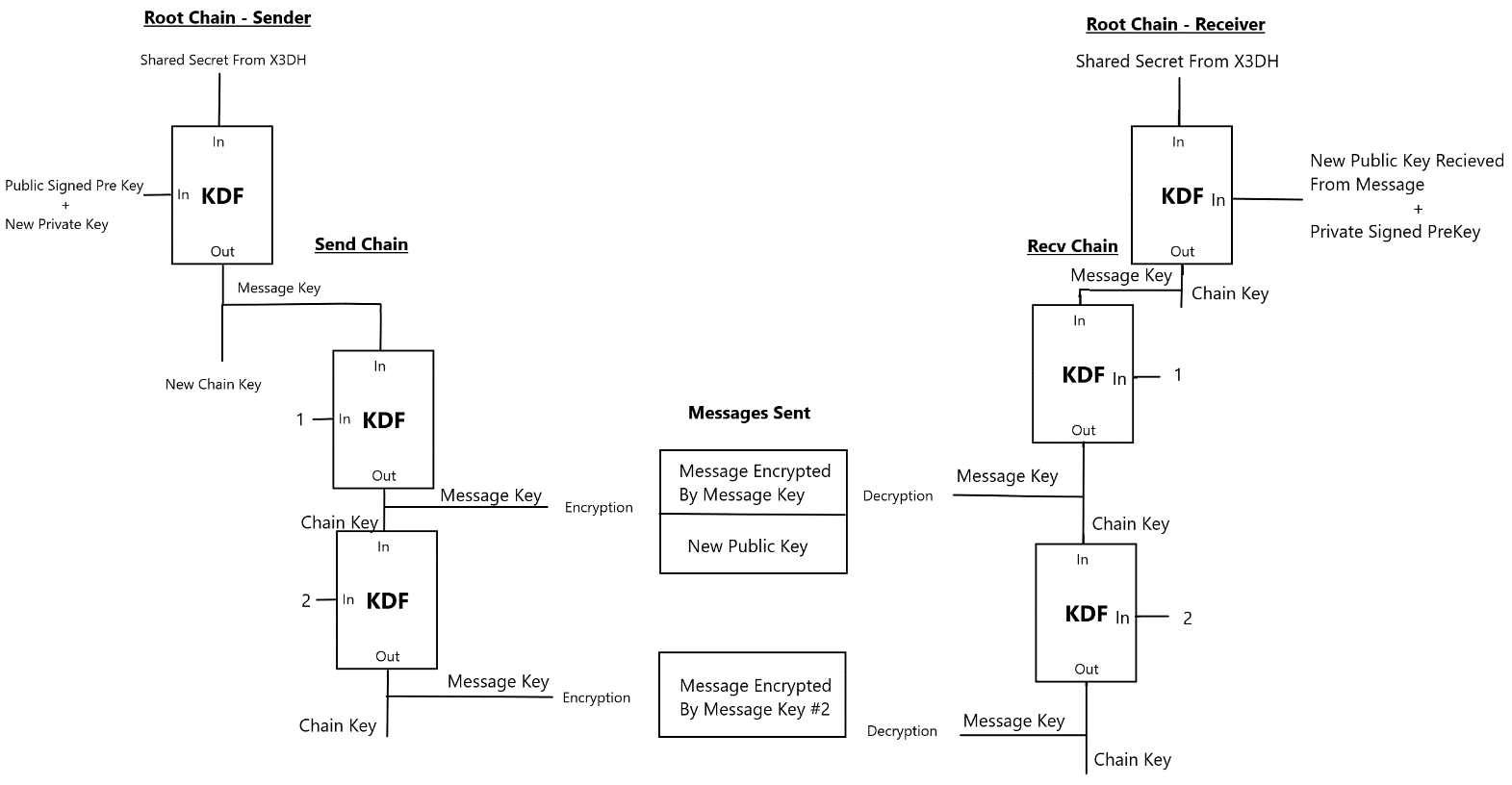


The chain on the left is known as the root chain, and the chain on the right is known as the send chain. A new KDF is added on the send chain if the user sends a message without a reply.

When the user sends the first message, they will send their public key as well as the encrypted message (encrypted by the message key). If the user keeps sending messages without a reply, a new KDF is added, and the input is increased by 1.

These chains are included to provide forward secrecy – if a hacker breaks the key found in the X3DH, they will need to break the keys used as an input in the root chain as well, and they would need to keep breaking all the keys used after this to decode every message.

Each send chain has a symmetrical receive chain that is used to derive the encryption keys. Below is an image that illustrates this entire process:

<https://signal.org/docs/specifications/doubleratchet/>

### Diffie Hellman

In the sender’s root chain, the public signed pre-key is combined with the new private key to produce a value and in the receiver’s root chain, the private signed pre-key is combined with new public key to produce the same value, but how is this done? The answer is a Diffie Hellman is performed.

A Diffie Hellman key exchange is a way of producing a shared secret by sharing public values through an unsecure channel. To simplify how a Diffie Hellman key exchange works: both parties produce a public private key pair. They then exchange these public values, so each party has: their public key, their private key, and the other person's public key. They then combine their private key with the other parties’ private key to produce a shared secret value that only they know. If an eavesdropper intercepted the communication, then they would only get both public components, meaning that they cannot calculate the shared secret.

This can be implemented in multiple different ways, but we will look at two in this document.

#### Diffie Hellman using Modular Exponentiation

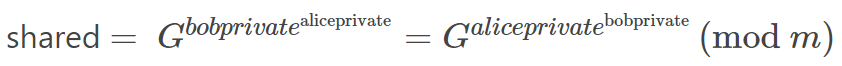
This implementation of Diffie Hellman uses the idea if we have the value v = Gx mod m, it is intractable to compute the exponent x, if large enough numbers are used. Note that the value of G should be a primitive root mod m so that for each value of x, an even distribution of value v is produced. This is known as the discrete logarithm problem, and it can be seen as a one-way function.

To start off the algorithm, both parties agree on some parameters, the value of G, and the value of m. Then each party generates a cryptographically secure random number for the value x – this will be their private key. To compute their public keys, they raise their private key to the base ‘G’ and get the remainder after division with the value m. They then share their public keys. They then raise their private key to the other parties’ public key and calculate the remainder after division with m. This new value is the shared secret.

Here is the algebraic proof that this works:









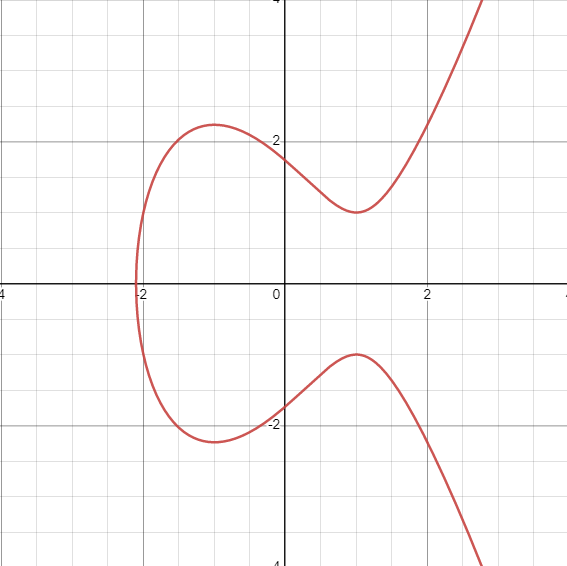
|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Easy to implement | Easy to compute |
| Easier to understand | Because it’s easy to compute, key sizes need to be exceptionally large (recommended 1024 – 2048 bits) |
|  |  |
|  |  |

<https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange>

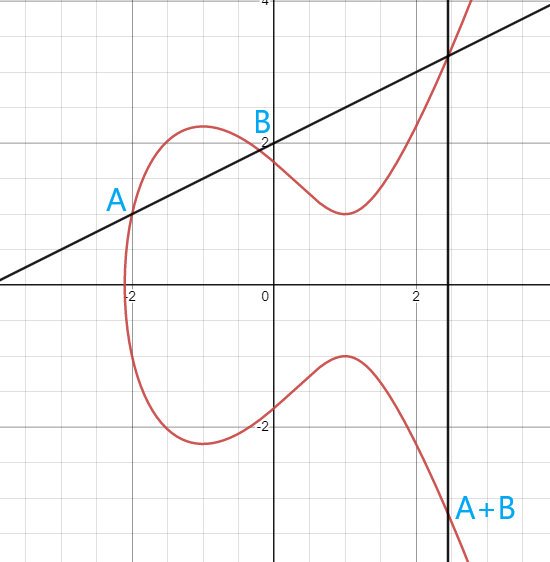
#### Diffie Hellman using Elliptic Curves

An elliptic curve is a graph that takes the form y2 = x3+ax+b (mod p). P is a prime modulus that defines the curve over a finite field (there can only be so many points on the curve).

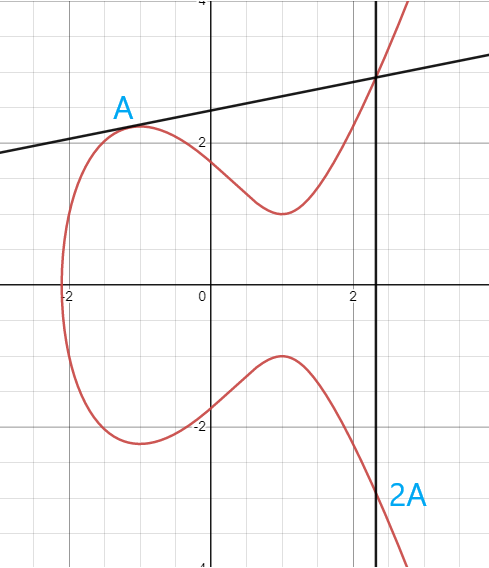
Here is an example of an elliptic curve with the curve parameters a = -3 and b = 3, but this is not defined over a finite field to make it easier to illustrate.



We need to define some functions that we can perform on the graph, the first one is point addition. Two points on an elliptic curve can be added. To add the two points, draw a line through both points and continue this line until it intersects the curve, then draw a line parallel to this point until it intersects the curve again. This point is defined as A + B. Here is an example of this:



The second function we need to define is point doubling. To double a point on the curve, draw a tangent to this point and continue it until it intersects the curve, then draw a line parallel to this point until it intersects the curve again. This point is defined as 2A (or A + A).



The third function we need to define is point multiplication. Point 7A is equal to ‘A’ added to itself 7 times (7A = A + A + A + A + A + A + A). This can be made more efficient by adding doubled points together (7A = A + 2A + 4A). Note that point multiplication is a one-way function – there is no point division so if you know A and Ax, you cannot find x without going through every point and checking if it is equal to Ax whilst keeping a count of the points checked. It is tractable to calculate an exceptionally large point because it has a big O notation of O (log n) where n is the number to multiply the point by, but it is intractable to find what a point was multiplied by to get a new point because you would need to brute force through every point on the elliptic curve. If a 160-bit curve is used, it will take on average 365,375,409,332,725,729,550,921,208,179,070,754,913,983,135,744 iterations to find the multiplier (2158).

Now to calculate a shared secret using elliptic curves, we first agree on the curve parameters, then both parties generate random numbers in the range 1 and n-1 where n is number of points on the curve (the order of the curve). They then produce public keys, which are points on the curve by multiplying the generator point by the private key (the generator point is just a point on the curve that can be multiplied to produce any of the valid points on the curve). They exchange these points and then they multiply the other parties public key point by their private key. This can also be proved algebraically:











|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Harder to compute | Hard to understand |
| Smaller key sizes (around 160 bits) | Harder to implement |

<https://en.wikipedia.org/wiki/Elliptic-curve_Diffie%E2%80%93Hellman><https://en.wikipedia.org/wiki/Elliptic_curve_point_multiplication>

#### Diffie Hellman Conclusion

I am going to use Elliptic Curve Diffie Hellman because it allows me to use smaller key sizes for the same amount of security, and due to the fact that I can also implement the digital signature algorithm (DSA) using elliptic curves.

### Elliptic Curve Digital Signature Algorithm

Digital signatures offer cryptographic proof that someone knows the private key used to make a public key. A piece of text can be signed by the controller of a private key to produce a signature. Another party can then verify the signature using the piece of text, the signature and the public key.

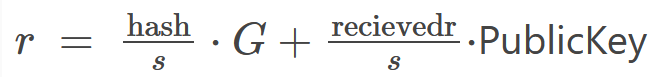
I will be using my implementation of the Digital Signature Algorithm within the X3DH key exchange which will use elliptic curves.

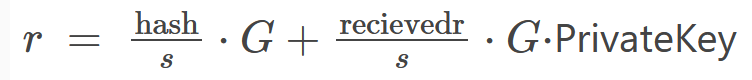
To generate a signature, the signer needs to produce a private key, a public key, a random integer (in the range 1 to n-1) and a hash of the message to sign. Note that the hash needs to be the same bit length as the curve order so a 256-bit hash would need to be truncated to the number of bits representing the curve order. The hash now needs to be converted to an integer. A signature consists of two values r and s – the significance of these two values will be shown later on.

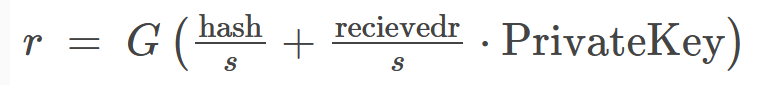
Once the signer has calculated these values, they can then calculate r which is the x coordinate of the generator point multiplied by the random number. After this, they can calculate the value of s which is the (hash + private key \* r) divided by the random number. The signer then gives the value of r, s, the public key and the message to the verifier.

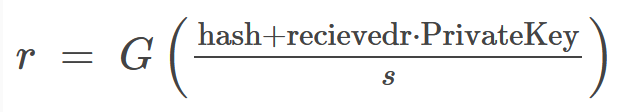
Once the verifier has received these values, they can go on to verify the signature. The verifier produces a hash of the message and truncates it like before. They then divide the hash by the value of s giving them the value u1. After this, they then divide the value of r by s giving them u2. They then calculate two points on the curve using point multiplication: point 1 is u1 multiplied by the generator point and point 2 is u2 multiplied by the public key point. These two points are then added together using point addition. Take the x coordinate of this point and compare it to the received value of r. If they are equal, then the signature is valid, if they are not equal then the signature invalid.

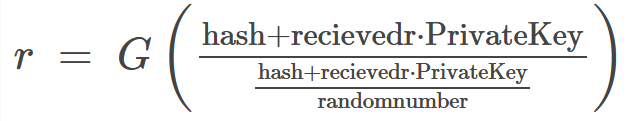
To explain why this works, we can do some algebraic proof (note all calculations are done mod n):

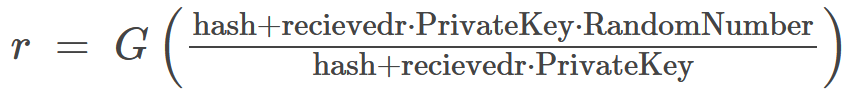














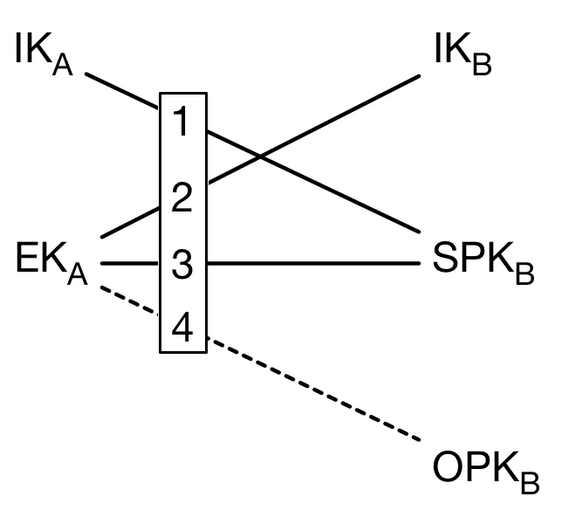
As you can see, if the values used to calculate ‘r’ and ‘s’ are the same then the first equation can be used to calculate r again. If our calculated value of r is not the same as the received value, then the private key that corresponds to the public key was not used to calculate ‘s’ (or another value was incorrect, which is unlikely, but true).

<https://en.wikipedia.org/wiki/Elliptic_Curve_Digital_Signature_Algorithm>

### Extended Triple Diffie Hellman

X3DH is a way of producing a shared secret value whilst avoiding a man in the middle attack. When an account is created, a user creates a key pair which are their identity keys. The user then publishes the public identity key to the server. This key will be used to verify the identity of the user. The user also generates a new key pair which are their signed pre keys. The user then produces a digital signature of the public signed pre key, called the pre key signature, using the private identity key. The public signed pre key and the pre key signature is then published to the server. The user then generates multiple key pairs and publishes the public components to the server – these are the ephemeral pre keys. The identity key is used to identify the user, the signed pre key and the ephemeral pre key is used to provide forward secrecy (if the attacker breaks one key, then they can’t read all of the messages).

When a user wants to start up a conversation with another user, they request a pre-key bundle from the server which contains the public identity key, the public signed pre key, the pre key signature and one of the public ephemeral pre keys. Note that the ephemeral pre-key is optional as it just makes the encryption even harder to break so we might as well get it if we can. The user should also create an ephemeral key pair. Before continuing, the user should verify the pre-key signature with the public identity key to make sure that the server hasn’t tampered with the signed-pre key. The user should also verify the identity key of the other user out of band to make sure that they are actually speaking to the correct person and a man in the middle attack isn’t being performed. Once these have been verified, the user should now preform four Diffie Hellmans, one with the public identity key of the other user and the private ephemeral key, another with the private identity key and the public signed pre-key, another one with the private ephemeral key and the public signed pre key and a final one with the private ephemeral key and the public ephemeral pre-key. The first two are used to provide mutual authentication, and the last two are used to provide forward secrecy. All these four values are then used to derive a single value, known as the X3DH shared secret.



Now that the user has generated the X3DH shared secret, they use that as the root key for the double rachet algorithm. The user then publishes a message to the server which contains their public identity key, their public ephemeral key, the id of the ephemeral pre-key that they used and the encrypted message. The server then contacts the recipient and when the recipient comes online, the server sends the message to the recipient and then they perform all of the Diffie Hellmans and decrypt the message.

<https://signal.org/docs/specifications/x3dh/>

## Objectives

### Objective 1

1. GUI should contain a login system
   1. First screen when loading up the GUI should be a login screen
      1. Should contain an input for the username
      2. Should contain an input for the password
      3. Button to submit data to server
   2. When login data is sent to the server, the server should authenticate the user
      1. Receive JSON data from client and parse into a C# object
      2. Query database with username and extract password hash and salt
      3. Hash and salt received authentication code then compare with stored hash
      4. If they are equal then generate a session token, save it in the database and send the session token to the client
      5. If they aren't equal, then return an error message
   3. GUI should display different items based on response from authentication server
      1. Parse JSON response into C# object
      2. Display error if necessary
         1. Extract error message from object
         2. Create red message box
         3. Set message text to extracted error message
         4. Display red message box on screen
      3. If authentication is successful, take user to main screen
         1. Close current form
         2. Open main screen form

### Objective 2

1. Server should authenticate requests
   1. When a client sends a command to the server, they should include their username and session key
   2. The server should check if these values are correct before performing the command
      1. Parse JSON into C# object
      2. Query database with username and extract session token with the expiration date
      3. Make sure token is valid
         1. Compare expiration date with current time, if the current time is greater than the expiration date then the token is invalid
         2. Compare the token sent with the token in the database, if they are equal then token is valid
      4. If token is valid, then perform command
      5. If token is invalid, then return an error message

### Objective 3

1. Messages between two people should be end-to-end encrypted
   1. When a new chat is created, create a shared secret key using X3DH
      1. Contact the server for a pre-key bundle
      2. Verify the pre-key signature
      3. Generate ephemeral key-pair
      4. Perform 4 Diffie Hellman key exchanges with pre-key bundle
      5. Generate shared key from the 4 key exchanges
      6. Send public ephemeral key, pre key bundle ID and public identity key to recipient
      7. Recipient follows this process in reverse to generate the same shared secret key
   2. When a message is sent, encrypt it using a KDF chain
      1. Generate a new keypair
      2. Perform Diffie Hellman with new private key and recipients signed-pre key (if first message, otherwise use last public key sent by the recipient)
      3. Take the X3DH and new Diffie Hellman values as inputs to the KDF function
      4. Use generated message key and message ID as input to KDF function
      5. Use new generated message key to encrypt the message
      6. Send the public key along with the message and the message ID to the recipient
   3. When a message is received, decrypt it using a KDF chain
      1. Extract public key from message
      2. If the message is the first message sent, perform Diffie Hellman with the received public key and the private signed pre-key, otherwise perform the Diffie Hellman with the received public key and the last generated private key
      3. Use the X3DH key and the new key as input to the KDF chain
      4. Use generated message key and message ID as input to KDF function
      5. Use new generated message key to decrypt the message

## Server Architecture Considerations

This application with be created using a client server architecture. This means that a machine will be dedicated to serving clients. Clients will make requests to the server which will then cause the server to perform a task for them.

There are multiple different ways to implement a client server architecture and I will analyse two.

### Monolithic Architecture

In this architecture, we produce one executable file for our server. This one program is responsible for performing all server side tasks.

This is a simple architecture and is great for small projects with a small amount of bandwidth. As development goes on and more features get added to the server, debugging gets more complex because the system will become more and more convoluted. If we start getting more and more clients, we will need more processing power and therefore we would need to scale the server. Using a monolithic architecture, we can only scale vertically (upgrade the computer components) because the server is one executable file. Vertical scaling is only possible up to a certain point, so this will limit the number of clients we can handle.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Easy To Implement | Can grow more and more convoluted |
| One Computer (simplicity) | Hard To Debug |
| One Program (simplicity) | If the program crashes, the whole system goes down |
|  | No Redundancy |
|  | Only Vertically Scalable |
|  |  |

### Microservice Architecture

In this architecture, multiple servers are created, and each server is dedicated to a certain task. For example, we could have a server handling web traffic, a server handling API requests, a server handling user authentication, a server handling account creation and so on. To allow these servers to be accessed through one IP address, a reverse proxy must be used to send traffic to the server that handles the received request.

Because multiple servers are being used, we have redundancy. For example, if the signup server goes down, people will not be able to create an account, yet people that already have an account will be able to use the service perfectly fine. This also means that we have added lots of complexity to the system as a whole but reduced the complexity of each server. This means that debugging should be easier because each server is simpler. It can be difficult to manage this architecture because there are many servers, and these servers need to be able to communicate with the other servers.

This architecture can be scaled both horizontally and vertically. Horizontal scaling means that we can add more computers instead of upgrading the computer we already have. To do this, we would create more instances of certain servers on other machines and load balance the requests between these new instances. This also means that we don’t have to scale each server because if we are getting lots of requests to the login microservice but getting few requests to the signup microservice, we can increase the number of instances of the login microservices whilst not scaling the signup microservice.

|  |  |
| --- | --- |
| **Advantages** | **Disadvantages** |
| Can be scaled vertically and horizontally | Difficult to manage |
| Can handle lots of clients | Difficult to implement |
| Can scale separate services whilst leaving other services untouched | Lots of different systems |
| Redundancy |  |
| Easy to debug each instance |  |
|  |  |

### Architecture Conclusion

Using monolithic cos microservices is too much work for less marks.

## Other Considerations

### Programming Language

C# because experience

### Database Software

MySQL because experience

# Modelling

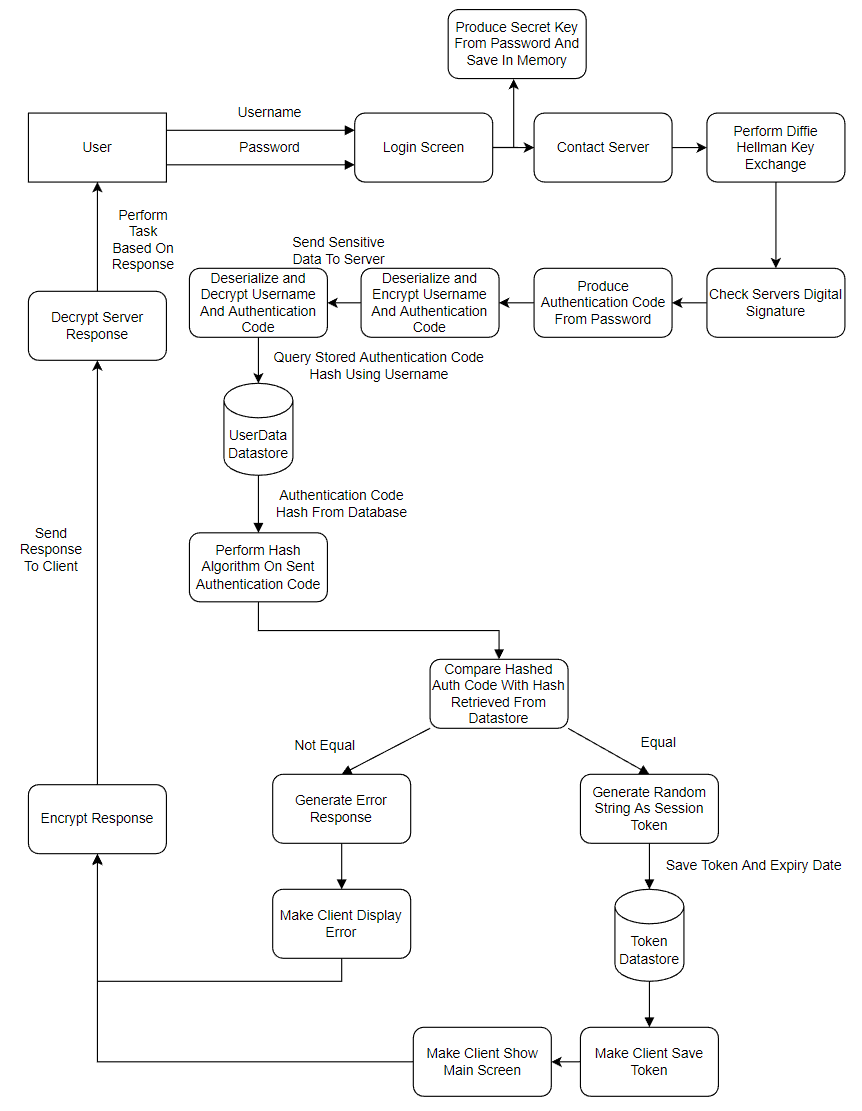
## Login System

### Data Flow Diagram - Level 0

Diagram

Description automatically generated

### Data Flow Diagram - Level 1



## Sending the first message

### Data Flow Diagram - Level 0

Diagram

Description automatically generated

### Data Flow Diagram - Level 1

Diagram

Description automatically generated

# Design

## Database – Objective x

Diagram

Description automatically generated

(Ill add the relationships in the future because they are unnecessary for implementation)

The reason why there are lots of tables in this relational database is because lots of values need to be stored in case a user goes offline and switches to another device. Some of these values need to be only known to the client therefore the client needs to encrypt them before sending them to the server.

### Explanation Of Each Table

#### Inactivated Accounts

This table stores accounts that have just been signed up. All of the information in this table (except for the ID, Expiration and Activation Code) will be copied into the Users table once the user has activated their account. The server will generate a random string as the activation code and send an email to the user with a link on it. This link will contain get parameters referencing the Activation Code, and the server will check this parameter with the code it has stored in the table. If they match, it will copy the data into a new row in the ‘Users’ table and get the user to generate the other necessary information.

#### Users

This table stores the account information of the user, and it holds the ‘UserID’ field that will be used by other tables to reference the user. The authentication key is used to authenticate the user server side and is derived from the password (look at introduction for more detail).

#### Handshake Values

These values can be seen as handshake constants because they are used in the X3DH handshake every time a user creates a new conversation with another user. The identity key is never changed but the signed-pre key could be changed every so often (for example, every 30 days).

#### Private Keys

This table holds the private components of the keys in the handshake values. They are all encrypted by the secret key.

#### Conversations

A conversation is a one-way chat by this definition. There is a source user and a destination user. A chat between two people is two conversations. The conversation ID is used so the server knows where to send the data. When a user sends an initial message, a conversation is made. When the recipient replies, a new conversation is made.

#### Messages

When a user sends a message to another user, it is stored in this table until they read it. It is then removed from the table and stored into old messages table. The conversation ID acts as an alias to the source and destination users. The sequence count holds the number of messages sent without a reply – this will be used as an input to the KDF chain allowing us to read messages sent out of order if an error occurs when sending happens. The Message Data column holds the contents of the message. If the message type is of type ‘image’ (noted in the ‘Message Type’ field), the message data will hold a path to the encrypted image on the filesystem. If the message is of type ‘text’, the message data will hold the message encrypted by the message key derived from the KDF chain. If the message is the first in the sequence, the public key field will be populated. This holds the public key used as input to the KDF chain.

#### Old Messages

After the initial message has been sent and read by the user, it is removed from the messages table and put into this table with a few modifications. The plaintext message data is re-encrypted using the secret key derived from the password. The Encryption Key ID tells us what secret key has been used to encrypt the data (in case of a password reset) – more on the Encryption Key ID below. The conversation ID tells us who sent the message and who the recipient is.

#### Encryption Keys

To allow the user to change their password, we need to store old secret keys that are used to encrypt the old messages. Because the user can store an unlimited amount of messages, re-encrypting these old messages may take a long time – and if the connection drops out whilst the client is in the middle of re-encrypting the messages, we would have a problem because the client would need to reconnect and a system would need to be put in place to get the old password and the new password from the client and then start the re-encryption process again – which wouldn’t be ideal. Instead, when we reset the password, we get the client to input their old password along with the new password. We then produce the secret keys from the passwords, encrypt the old secret key with the new secret key and store it in this table. If the user isn’t in the process of resetting their password, the encryption key field will be null, and the client will know to use the secret key in its program memory (the client will query the server for the row with a null encryption key field and will return the ID of that key so that the client knows when to use the secret key in main memory without querying the server again).

When the user loads the client up, the server will inform the client that it needs to re-encrypt the values. The client will then start re-encrypting the values one-by-one in the background and sending the re-encrypted values back to the server slowly. Once there is no reference to the old secret key in the encryption keys table anymore, the server will then delete this key from the table.

#### Tokens

This table is used in authenticating users. When a user logs in, a session token is created and saved in this table. The token is then sent whenever the client makes a request and is used to authenticate the client by querying this table. Each token has an expiration date – if the token’s expiration date is greater than the current date then it can no longer be used in authenticating the user.

#### X3DH Handshake

This table is used to store the values needed for the recipient to decode the first message and plays a key role in the X3DH handshake. When the user sends a first message, they will upload the values that will be stored in this table. When the recipient receives the first message, they contact the server for the values. The server will then find the conversation then query this table for the correct handshake values. It will then query the private pre key table and collect the encrypted private pre key along with the encryption key ID. It will then bundle up all these values and send them to the client so they can perform the handshake.

#### Management Requests

This table is used as a buffer for management requests that will be sent down the management tunnel when a user comes online. When the user starts a connection with the management tunnel, the server will query all of the buffered data and start sending it to the user. This is data like new messages and friend requests.

#### Chain Keys

This table holds the private chain keys in case a user goes offline and switches to another device. They are used in the double ratchet algorithm.

#### Public Pre-Keys

This table holds the ephemeral pre keys used in the X3DH handshake. The client must keep reuploading more pre-keys as they get used up.

#### Private Pre-Keys

This table holds the private components to the public pre-keys. They are encrypted by the secret key.

## API design – objective x

### Overview

### API Endpoints

#### Management endpoints

1. Management Tunnel (username, token) -> CONNECTION  
   URL: \api\management\management\_tunnel  
     
   A connection to this endpoint is maintained when a client is using the application. It is used to receive control messages from the server. The client will listen on this connection for any messages, and if a message is received, the client will decode the message and complete a task. Some example messages may be new conversation requests, new message notifiers and pre-key notifiers.
2. Request Account Creation (username, email, authentication code) -> None  
   URL: \api\management\request\_account\_creation  
     
   This endpoint creates a row in the inactivated accounts table and sends an email to the specified address. The email will contain a link with an identifier and when clicked it will complete the signup processes
3. Finalize Account Creation (username, activation code) -> None  
   URL: \api\management\finalize\_account\_creation  
     
   This endpoint will move the account from the inactivated account table to the users table if the correct activation code is given and if the expiration date hasn’t been past.

#### Authentication endpoints

1. Generate Token (username, authentication code) -> token  
   URL: \api\authentication\generate\_token  
     
   Used for creating session tokens. This session token will be used to identify the user whenever a request is made to the server.

#### Secret Storage endpoints

1. Retrieve Handshake Values (username, token) -> identity key, encryption ID  
   URL: \api\secrets\retrieve\_handshake\_values

This endpoint allows the client to retrieve their handshake values. These are private values, so they need to be decrypted to be used in the X3DH handshake.

1. Upload Initial Keys (username, token, private identity key, public identity key, public signed pre key, private signed pre key, pre key signature) -> none

URL: \api\secrets\upload\_initial\_keys

This endpoint allows the client to send the private keys to the server. They need to be encrypted before being sent so the encryption id should be sent in the JSON object. If the keys are already stored, nothing will happen.

1. Upload Prekey (username, token, public pre key, private pre key)  
   URL: \api\secrets\upload\_prekey

This endpoint allows the client to upload a new ephemeral pre-key when they are running low on them. Since a pre-key is used up every X3DH handshake, the client needs to make sure that they don’t run out.

#### Message Endpoints

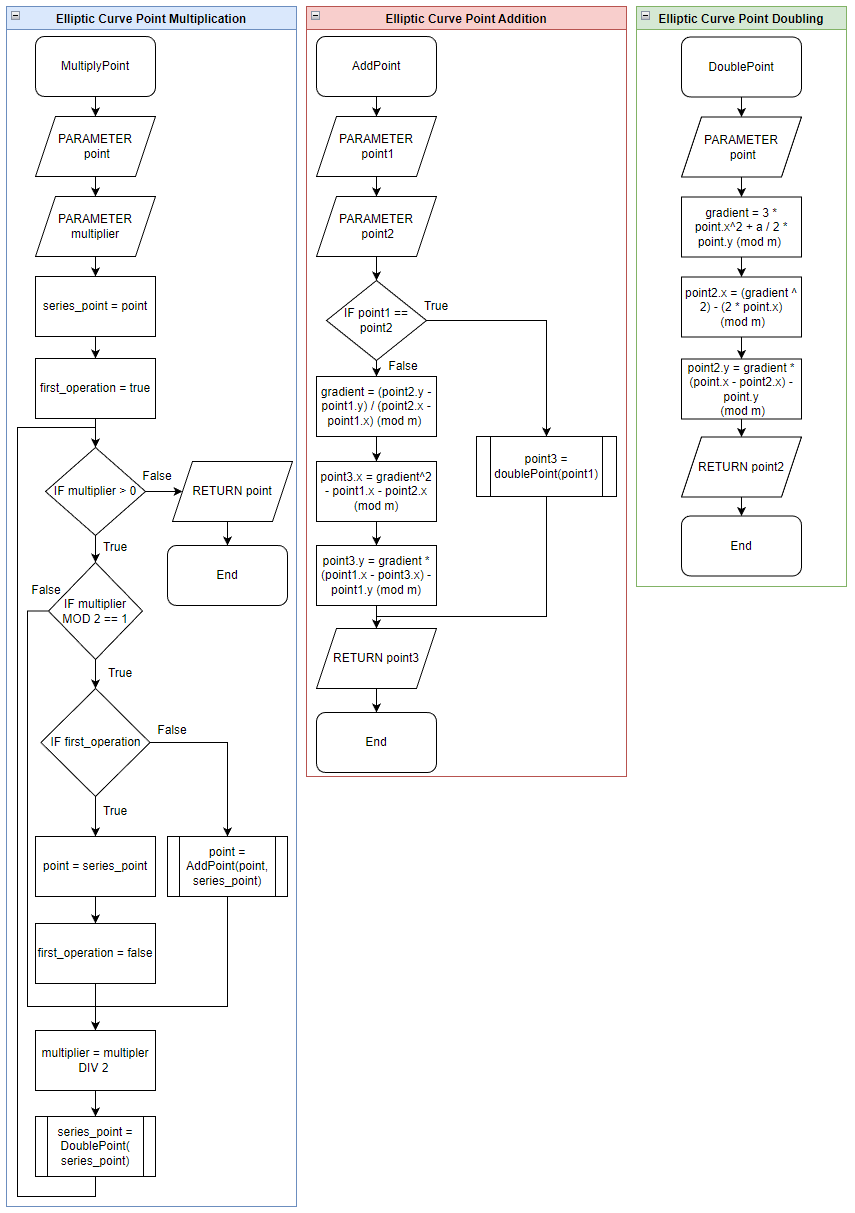
1. Send Message (username, token, recipient, message type, message data, sequence count, public key) -> None  
   URL: \api\message\send\_message  
     
   This endpoint is used to send encrypted messages to other clients.
2. Receive Message (username, token, sender) -> message type, message data, sequence count  
   URL: \api\message\recieve\_message  
     
   This endpoint is used to receive encrypted messages from other clients. This will be called when we get a message from the management tunnel telling us that we have received a message.
3. Upload Message (username, token, recipient, encryption ID, message data, message type) -> None  
   URL: \api\message\upload\_message

This endpoint is used to upload old messages to the server, encrypted by the secret key. This allows users to save message history.

1. Download Message (username, token, sender, index) -> message type, message data, encryption ID  
   URL: \api\message\download\_message  
     
   This endpoint is used to download old messages allowing users to see message history on multiple devices.
2. Get Last Message Index (username, token, sender) -> index  
   URL: \api\message\get\_last\_message\_index  
     
   This endpoint will be called every time a chat is viewed. It allows the client to get the index of the last message so they can view old messages in chunks at a time. This is because we do not want to receive all of the message history at once because that could be slow and wasteful.

## Key Algorithms Design – objective x

### Elliptic Curve Functions



### Cryptographic Functions

Diagram, schematic

Description automatically generated

### Merkle Damgard Construction Based Hash Function

Diagram

Description automatically generated

#### Permutation Diagram

Diagram

Description automatically generated

### End To End Encryption Algorithms

#### X3DH, Double Ratchet, KDF