DCU
Ollscoil Chathair
Bhaile Átha Cliath
Dublin City University

SecureYAC
Technical Manual

Eryk Zygmunt Styczynski (21753851), Liucija Paulina Adomaviciute (21790411)

Supervisor: Geoff Hamilton

SecureYAC is a messaging application with a focus on encryption and security. The goal of the project is to provide a cryptographically secure communications channel that does not depend on third-parties, such as servers. It establishes a direct connection between users using a custom peer-to-peer networking protocol. The user identity creation and verification is handled by a key agreement protocol based on the Signal's X3DH specification. Messages between users are encrypted using Signal's Double Ratchet algorithm.

Dublin City University
School of Computing
May 2025

Table of Contents

1. Introduction

Due to data breaches becoming increasingly common and rising privacy concerns, there has been a significant interest in ways to protect personal data. Most everyday communication happens over the Internet, and we wish to explore ways to protect our communication channels. End-to-end encryption technology provides a solution. Our project aims to investigate the design and implementation of network and cryptographic protocols to understand their mechanisms of action and potential risks.

We are aware that due to security concerns, the standard and recommended practice is to use cryptographic libraries developed by professional cryptographers. By implementing the cryptographic protocols ourselves from the ground up, using native tools provided by our chosen programming language, we aim to gain a better understanding of how elliptic curve and public key cryptography work and to explore lesser-known and uncommon cryptographic protocols that do not

have widely used standard libraries. Due to the lack of experience in the field of cryptography, we do not guarantee cryptographic security or the correctness of our application.

Notation

A || B denotes a concatenation of byte arrays A and B.

2. Research

2.1 Encryption and Cryptographic Protocols

Algorithms used for identity generation and verification and message encryption are the focus of our application. While multiple cryptographic protocols were considered, we decided to use Signal's X3DH[2] and Double Ratchet[3] protocol specifications as the basis for our application's cryptography.

The X3DH key protocol requires Signal's signature scheme XEdDSA[4]. Double Ratchet specification includes integration of the X3DH key agreement protocol. Together, they cover identity generation, verification, and message encryption.

These protocols have been chosen for their non-standard implementation and expansion of more commonly used cryptographic protocols (Diffie-Hellman key exchange algorithm and EdDSA digital signature scheme). Currently, no programming language offers native implementation of these protocols, and the only available open-source implementation written in Java and Kotlin is developed and maintained by Signal.

While the specifications of these algorithms were designed and described by professional cryptographers, due to lack of experience in cryptography, our implementation does not guarantee correctness or security.

2.2 Elliptic Curve Cryptography

An elliptic curve is a smooth, projective, algebraic curve of genus one, on which there is a specified point O[5]. These curves have been successfully applied in elliptic curve cryptography (ECC) to provide robust security with smaller keys. ECC fundamentally relies on the Elliptic Curve Discrete Logarithm Problem (ECDLP)[6], where determining the scalar k from points P and kP on the curve is computationally challenging.

Curve25519

Curve25519 is an elliptic curve, where E is the curve equation $y^2 = x^3 + 486662x^2 + x$, and p is the prime number $2^{255} - 19$[7] used to define the finite field $\mathbb{F}p$.

Curve25519 offers a ~128-bit security level, and has the following properties:

Name	Definition
Montgomery curve equation	$v^2 = u^3 + A * u^2 + u$
A	486662
Prime number defining the prime field $\mathbb{F}p$ (p)	$2^{255} - 19$
Order of the prime-order subgroup	2^{252} + 0x14def9dea2f79cd65812631a5cf5d3ed
Cofactor	8
u-coordinate of base point P on a Montgomery curve (U(P))	9

v-coordinate of base point P on a Montgomery curve (V(P))	0x20ae19a1b8a086b4e01edd2c7748d14c923d4d7e6d7c61b229e9c5a27eced3d9

The shape of this curve and the use of compressed elliptic points (only x-coordinate) allow for fast point calculations[7].

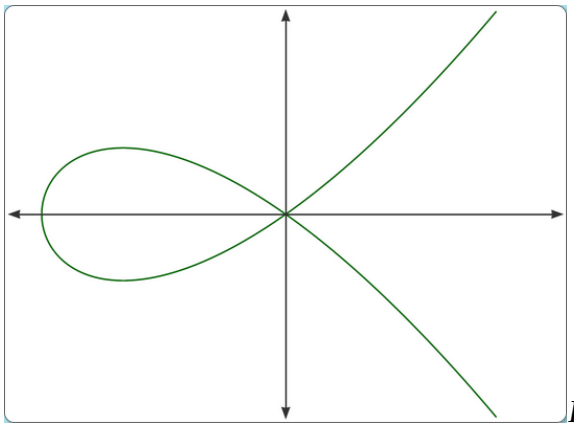


Figure 1: Curve25519 in real numbers [1]

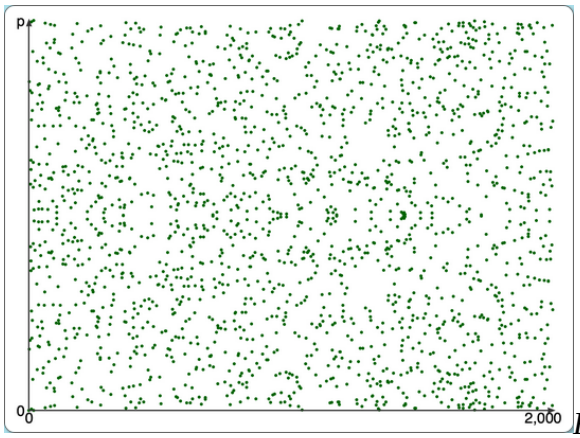


Figure 2: Curve25519 calculated in 𝔽p[1]

Ed25519

Curve25519 is birationally equivalent to a twisted Edwards curve, called Edwards25519[8]. This curve is used in EdDSA and XEdDSA signature schemes for fast creation and verification of signatures.

Ed25519 is EdDSA created using Edwards25519 elliptic curve and has the following parameters[8], [9], [10]:

Name	Definition
b	256
Hash (H)	SHA-512
Prime number defining the prime field 𝔽p (q)	$2^{255} - 19$

Curve448 and Ed448

Curve448 is an elliptic curve, where p is the prime number $2^{448} - ^{2224} - 1$[11], [12] used to define the finite field 𝔽p.

Curve448 offers a ~224-bit level of security and is 4-isogenous to the Edwards curve$x^2 + y^2 = 1 + d * x^2 * y^2$, called Ed448. Ed448 is also called “Goldilocks”.

X25519 and X448

X25519 and X448 are functions that performs scalar multiplication on Montgomery form Curve25519 or Curve448 elliptic curve for key generation. It takes a scalar (an array of randomly generated bytes) and a u-coordinate as inputs and produces a u-coordinate as an output. Using this function, each user generates 32-byte long keys. These keys are subsequently used in elliptic curve Diffie-Hellman (ECDH) calculations in the X3DH key agreement protocol. The values of these keys are congruent to the value modulo p and are stored in little-endian order[12].

To encode the output as X25519 key, the following bit clamping operation is performed: the three least significant bits of the first byte and the most significant bit of the last byte are set to 0, and the second most significant bit of the last byte is set to 1. This results in an integer of the form 2^{254} plus eight times a value between 0 and $2^{251} - 1$ (inclusive)[12].

The implementation of this function is designed to be fast and constant-time but does not guarantee protection from side-channel attacks[12].

2.3 XEdDSA

XEdDSA Elliptic Curves

XEdDSA is a signature scheme used to create EdDSA-compatible signatures using X25519 or X448 elliptic curve Diffie-Hellman functions[4]. XEdDSA allows the reuse of the same key pair for both ECDH calculations and signatures.

XEdDSA uses the following elliptic curve parameters[4]:

Name	Definition
B	Base point
I	Identity point
u	Curve25519 or Curve448 Montgomery public key u-coordinate
P	Edwards curve twisted point
p	Field prime
| p |	$ceil(\log 2(p))$
q	Order of base point (prime; q < p; qB = I)
| q |	$ceil(\log 2(q))$

Curve Conversion

The keys generated using X25519 and X448 are calculated using the Montgomery ladder (or equivalent algorithm) and each user’s key is a Montgomery u-coordinate. EdDSA (and XEdDSA) are defined on a twisted Edwards curve, where the key (twisted Edwards point P) consists of a twisted Edwards y-coordinate along with a sign bit (0 or 1). Point P from the Montgomery u-coordinate can be calculated with $y = (u - 1)/(u + 1)$[12]. Since Montgomery keys do not store the v-coordinate, it is impossible to determine the twisted Edwards sign bit. Instead, the sign bit is forced to 0[4].

Conversion of Montgomery private key k to Edwards private and public keys A, a is done by multiplying k by the twisted Edwards base point B, forcing the sign bit of A to 0 and adjusting a if necessary to produce a sign bit of 0. The following pseudocode for elliptic curve conversion is given in the XEdDSA specification[4]:

```
calculate_key_pair(k):
    E = kB
    A.y = E.y
    A.s = 0
    if E.s == 1:
        a = -k (mod q)
    else:
        a = k (mod q)
    return A, a
```

Hash Functions

XEdDSA uses a cryptographic hash function. The default hash function is SHA-512. *hash* is a function that applies cryptographic hash to an input byte sequence and returns a hashed byte sequence. To provide cryptographic domain separation, given curve constants p and b, a family of hash functions, indexed by non-negative integers i, such that $2^{|p|} - 1 - i > p$, is defined[4]:

```
hashi(X):
    return hash(2^b - 1 - i || X)
```

hash$_1$ changes the first byte of input data to 0xFE.

The Algorithm

All calculations in the algorithm are done modulo q. The XEdDSA signing and verification functions take the following inputs[4]:

Signing		Verifying	
Name	Definition	Name	Definition
k	Montgomery private key	u	Montgomery public key
M	Message to sign (byte array)	M	Message to verify (byte array)
Z	64 bytes of secure random data	R || s	Signature to verify

Signing of the Key

To sign a message M using Montgomery key k, first, k has to be converted to Ed25519 keys using calculate_key_pair(k) method defined above. First, a nonce r is calculated by applying *hash*$_1$ to a concatenation a || M || Z. A point R on twisted Edwards curve is calculated by multiplying r by the base point B. A hash h is made by applying *hash* (SHA-512) function to concatenation R || A || M and the signature is calculated by adding hash h multiplied by private key a to the nonce r.

The Signal's documentation provides the following pseudocode for XEdDSA signing function[4]:

```
xeddsa_sign(k, M, Z):
    A, a = calculate_key_pair(k)
    r = hash1(a || M || Z) (mod q)
    R = rB
    h = hash(R || A || M) (mod q)
    s = r + ha (mod q)
    return R || s
```

Verification of a Signature

To verify a signature, the algorithm performs a series of checks to determine the validity of values. If any of these checks fail, the function returns false.

First, the verification function checks if the inputs (Montgomery public key u, Edwards point R and the signature) are valid values. If they are, the Montgomery public key is converted to the Edwards public key (point on a twisted Edwards curve), and the function confirms that such a point exists on the curve.

If all validity checks succeed, a hash function is applied to the byte concatenation of R || A || M modulo q. A verification calculation is done by subtracting hash h multiplied by Edwards public key A from signature s multiplied by Edwards elliptic curve base point B. If the resulting value R_{check} equals the Edwards point R appended to the signature, the signature is valid[4].

The following proof breaks down the verification function:

Theorem: $R_{check} = R$.

Let:

pK = Edwards form private key.

epK = Edwards form ephemeral key used in XEdDSA signature.

random = 64 bytes of secure randomly generated data.

B = Ed25519 base point.

q = Ed25519 order.

PK = pK * B (mod q)

R = $hash_1$(pK, epK) * B (mod q)

hash = hash(R, PK, epK)

Sig = $hash_1$(pK, epK, random) + hash(R, PK, epK) * pK (mod q)

Proof:

$$R_{check}=Sig * B - hash * PK(modq)=$$
$$=(hash_1(pK, epK, random) + hash(R, PK, epK) * pK) * B - hash(R, PK, epK) * (pK * B)(modq)=$$
$$=hash_1(pK, epK, random) * B + hash(R, PK, epK) * pK * B - hash(R, PK, epK) * pK * B(modq)=$$
$$=hash_1(pK, epK, random) * B(modq)$$

Therefore, $R_{check} = R$.

Signal provides the following pseudocode for verification method[4]:

```
xeddsa_verify(u, M, (R || s)):
   if u >= p or R.y >= 2^|p| or s >= 2^|q|:
      return false
   A = convert_mont(u)
   if not on_curve(A):
      return false
   h = hash(R || A || M) (mod q)
   Rcheck = sB - hA
   if bytes_equal(R, Rcheck):
```

```
        return true
    return false
```

Security Considerations

Using a random nonce with each signature improves the resilience of the scheme, making it safe to reuse the same keys[4].

2.4 X3DH

Protocol Parameters

Extended Triple Diffie-Hellman (X3DH) is a key agreement protocol that establishes a shared secret key between two parties who mutually authenticate each other based on public keys. The protocol uses three parameters:

- curve (X25519 or X448),
- hash (SHA-256 or SHA-512),
- info (ASCII string).

In Signal's specification, three roles are defined: Alice, Bob and a server. Alice and Bob are users who want to exchange data and establish connections, and the server stores the key bundles of the users[2]. Due to our application's server-less nature, the server role is omitted from our implementation, and the following description of the protocol will reflect this change.

X3DH uses the following elliptic curve public keys for calculations[2]:

Name	Definition
IK_A	Alice's identity key
EK_A	Alice's ephemeral key
IK_B	Bob's identity key
SPK_B	Bob's signed prekey
OPK_B	Bob's one-time prekey

It is assumed that each public key has a corresponding private key. The protocol requires the following functions[2]:

- **Encode(PK) –** encoding function to encode an X25519 or X448 public key *PK* into a byte array;
- **DH(PK1, PK2)** – an Elliptic Curve Diffie-Hellman calculation function that returns a shared secret value from calculations involving public keys *PK1* and *PK2*, whose form depends on the curve parameter;
- **Sig(PK, M)** – XEdDSA[4] signature on the byte sequence *M* that is verified with a public key *PK* and was created by signing *M* with *PK's* corresponding private key;
- **KDF(KM)** – 32-byte output from HKDF algorithm[2].

The Protocol

Each user has a long-term identity key ($IK_{A/B}$). During the protocol run, Alice generates an ephemeral key pair (EK_A). Bob has a signed prekey pair and a set of one-time prekey pairs that are exported into a key bundle and exchanged with Alice before the beginning of the protocol. After the

calculations, Alice and Bob share a 32-byte secret key (SK), which is used as an initial value in the Double Ratchet protocol[2].

X3DH has three phases:

1. Bob exports a key bundle (IK_B, SPK_B, Sig(IK_B, Encode(SPK_B)), OPK_B) and exchanges it with Alice through a trusted communications channel.
2. Alice imports a key bundle and uses it to send an initial message to Bob.
3. Bob receives and processes Alice's initial message.

Sending Initial Message

When Alice imports a key bundle, she verifies the prekey signature. If the verification fails, Alice aborts the protocol. Otherwise, Alice generates a pair of ephemeral keys EK_A and performs the following calculations using the public keys[2] (see Figure 3):

$$DH1 = DH(IK_A, SPK_B)$$
$$DH2 = DH(EK_A, IK_B)$$
$$DH3 = DH(EK_A, SPK_B)$$
$$DH4 = DH(EK_A, OPK_B)$$
$$SK = KDF(DH1 \,||\, DH2 \,||\, DH3 \,||\, DH4)$$

After performing these calculations, Alice deletes her ephemeral keys and intermediate Diffie-Hellman values. Then she calculates additional data AD, that contains identity information of both users:

$AD = Encode(IK_A) \,||\, Encode(IK_B)$.

The initial message contains:

- Alice's identity public key IK_A,
- Alice's ephemeral public key EK_A,
- Bob's OPK_B,
- Initial ciphertext encrypted using post-X3DH encryption protocol.

Receiving Initial Message

Bob, after extracting keys and ciphertext from the message, performs the following calculations (see Figure 4):

$$DH1 = DH(IK_B, EK_A)$$
$$DH2 = DH(SPK_B, IK_A)$$
$$DH3 = DH(SPK_B, EK_A)$$
$$DH4 = DH(OPK_B, EK_A)$$
$$SK = KDF(DH2 \,||\, DH1 \,||\, DH3 \,||\, DH4)$$

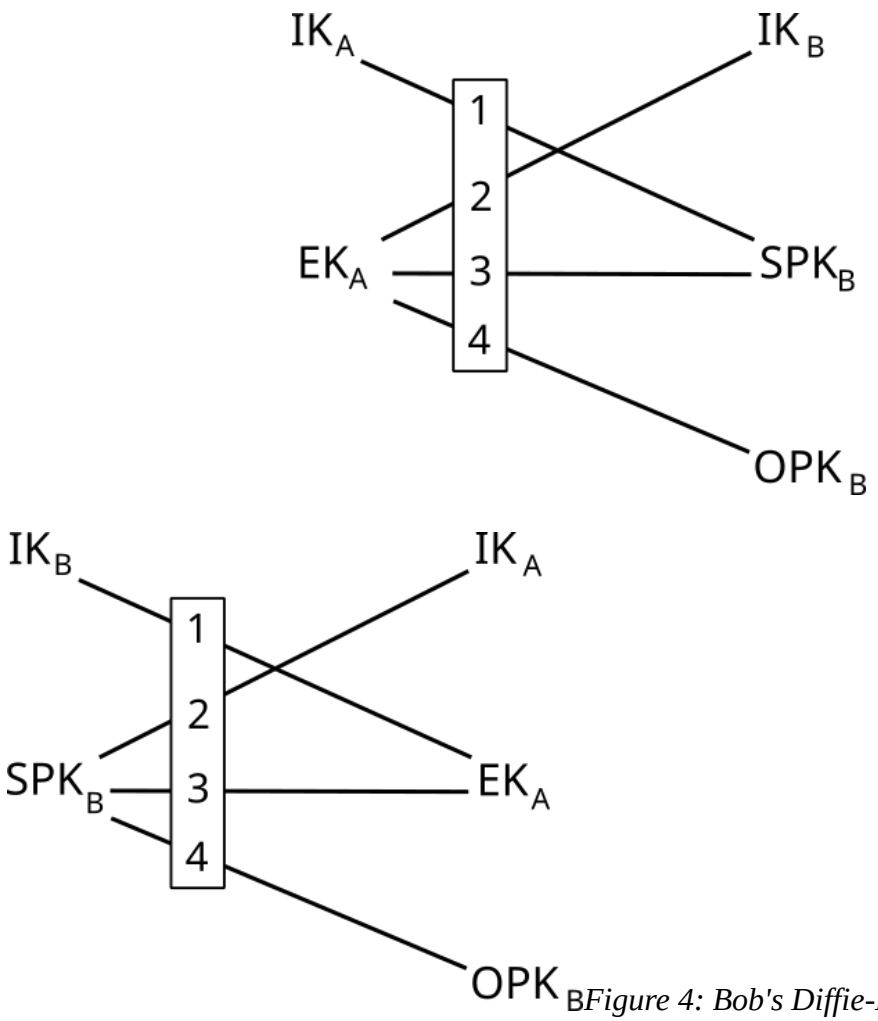


Figure 4: Bob's Diffie-Hellman calculations

Now, Bob and Alice have the same secret key and post-X3DH protocol can be initialised for message encryption and decryption.

Security Considerations

X3DH provides mutual authentication and forward secrecy, but does not perform identity verification. It does not give users publishable cryptographic proof of the fact that they communicated and the contents of that message exchange. It is necessary to randomise further encryption keys to prevent replay attacks and key reuse[2].

2.5 Double Ratchet

The Double Ratchet algorithm is used to exchange encrypted messages between two users. The users agree on a shared secret key using the key agreement protocol. In our implementation, the key agreement protocol used in X3DH. For each new message, new keys are derived, and the results of the Diffie-Hellman calculations are mixed into the key generation process. These steps prevent previous keys from being calculated from the prior ones and give the users some protection in the case of key compromisation[3].

Each user keeps track of the following variables for the Double Ratchet algorithm[3]:

Name	Definition
DHs	Diffie-Hellman ratchet “sending” key pair

DHr	Diffie-Hellman ratchet “receiving” key pair
RK	32-byte root key
CKr	32-byte “receiving” chain keys
CKs	32-byte “sending” chain keys
Ns	Message number for sending
Nr	Message number for receiving
PN	Number of messages in the previous sending chain
MKSKIPPED	Skipped-over message keys, indexed by ratchet public key and message number
MAX_SKIP	Maximum number of messages that can be skipped in a single chain

The algorithm requires the following functions[3]:

- **GENERATE_DH()**: generates a new Diffie-Hellman key pair;
- **DH(dh_priv, dh_pub)**: return the result of Diffie-Hellman calculations using private key *dh_priv* and *dh_pub;*
- **KDF_RK(rk, dh_out)**: return a 32-byte root key and 32-byte chain key as a result of applying KDF keyed by a *rk* to Diffiel-Hellman output *dh_out*;
- **KDF_CK(ck)**: return a 32-byte chain key and 32-byte message key as a result of applying KDF keyed by a *ck* to some constant;
- **ENCRYPT(mk, plaintext, associated_data)**: return AEAD encryption of *plaintext* using message key *mk*. *associated_data* is authenticated, but not included in the ciphertext;
- **DECRYPT(mk, ciphertext, associated_data)**: return AEAD decryption of *ciphertext* using message key *mk;*
- **HEADER(dh_pub, pn, n)**: generate a new message header, containing DH ratchet public key *dh_pub*, previous chain length *pn* and the message number *n*.

Key Derivation Function Chains

The Key Derivation Function (KDF) chain is a core concept in the Double Ratchet algorithm.

KDF is a cryptographic function that takes a key and some input data and returns output data that is indistinguishable from random.

KDF chain is when the output from some KDF function is used to replace the input KDF key[3].

A KDF chain has the following properties[3]:

- Resilience: output keys appear random without knowing the input KDF keys.
- Forward security: previous output keys appear random even if a KDF key is known.
- Break-in recovery: future output keys appear random even if a previous KDF key is known.

In a Double Ratchet session, KDF keys are used for three chains: root chain, sending chain and receiving chain.

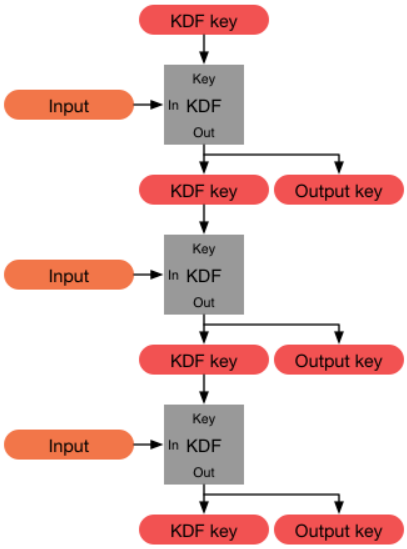


Figure 5: KDF chain[3]

Symmetric-Key Ratchet

Symmetric-key ratchet is a step, where a KDF chain is used with a constant input to produce a chain and a message key. The chain key is used as input for the next symmetric-key ratchet step, while the message key is used for encryption. Because the input to the KDF chain is constant, the symmetric-key ratchet does not provide break-in recovery[3].

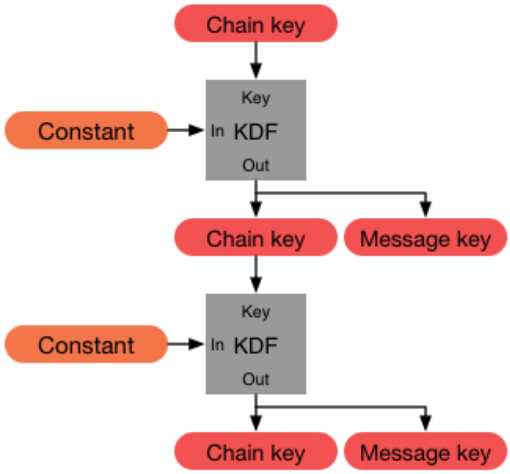


Figure 6: Two steps of the symmetric-key ratchet[3]

Diffie-Hellman Ratchet

To prevent message decryption in case of key compromise, the symmetric-key ratchet is combined with a Diffie-Hellman (DH) ratchet. DH ratchet updates the chain keys based on DH outputs.

Each party first generates a pair of keys, which become their ratchet key pair. The current public ratchet key is included in the message header. When a new public ratchet key is received, a DH value is calculated and a DH ratchet step is performed to replace the ratchet keys[3].

The DH ratchet initialisation steps between Alice and Bob (see Figure 6):

1. Alice receives Bob's ratchet public key.
2. Alice calculates shared secret DH value using Bob's public and her private ratchet keys.
3. Alice sends Bob a message containing her ratchet public key.
4. Bob receives Alice's ratchet public key and calculates shared secret DH value using Alice's public and his private ratchet keys.
5. Bob generates a new pair of chain keys.
6. Bob calculates a new DH value using Alice's ratchet public key and his new ratchet private key.
7. Bob sends a message encrypted using the new DH value and containing his new ratchet public key to Alice.
8. Alice repeats steps 1 & 2.

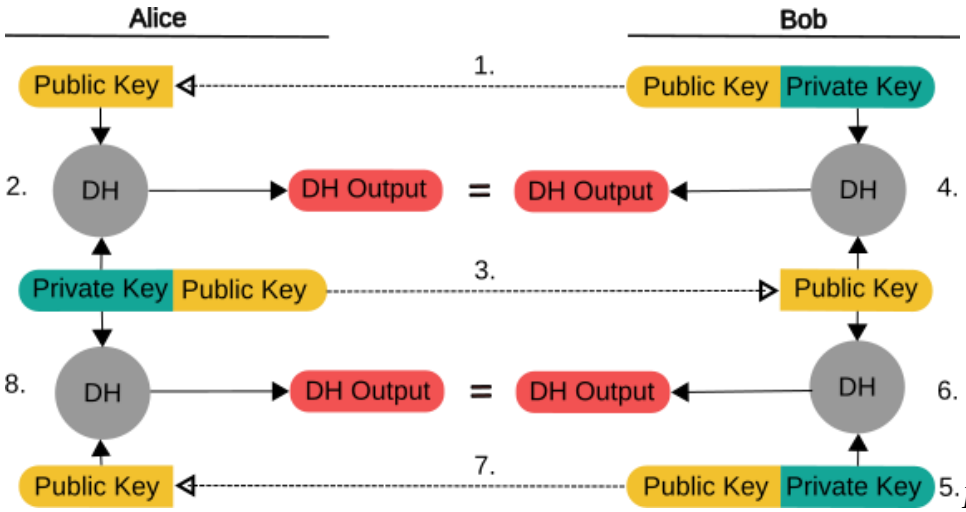


Figure 7: Diffie-Hellman Ratchet[3]

The DH outputs derived during each DH ratchet step are used to derive new sending and receiving chain keys, i.e. when Alice sends the first message at step 3, her DH output is used as an input to a root KDF chain, and KDF outputs from root chain are used to derive a sending chain key. When Bob receives Alice's public key and calculates his DH value in step 4, his DH output is used in the same way to derive new receiving chain keys, which are identical to Alice's sending chain keys. The DH ratchet and KDF chain integration is shown in Figure 7[3].

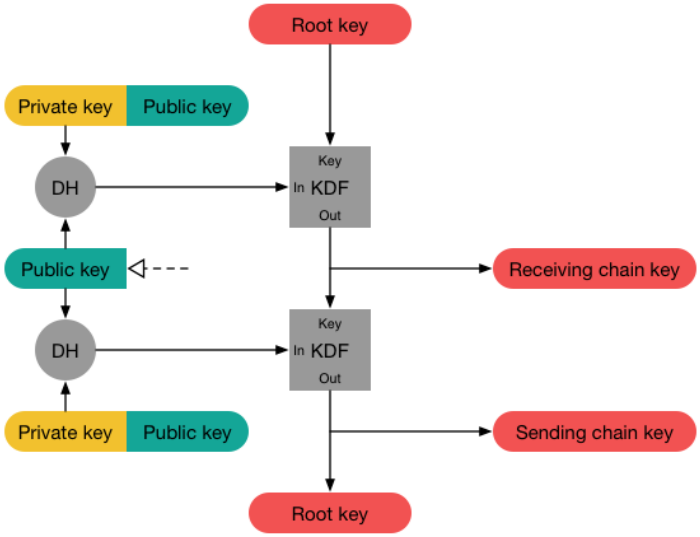


Figure 8: Diffie-Hellman ratchet with KDF chain[3]

Double Ratchet

The Double Ratchet is the combination of the symmetric-key ratchet and the Diffie-Hellman ratchet. When a message is sent or received, a symmetric-key ratchet is used to derive new message keys and when a message is received, a DH ratchet is used to replace the chain keys. X3DH can be integrated into Double Ratchet by using the shared secret value calculated by X3DH protocol as the initial root key and Bob's signed prekey pair as the initial root keys. Additional data (AD) used in X3DH is used in the Double Ratchet protocol as well[3].

The Double Ratchet protocol with X3DH key agreement protocol integration steps between Alice and Bob from Alice's perspective[3]:

1. Alice receives Bob's ratchet public key from his key bundle and initialises X3DH. The resulting shared secret value is the initial root key (RK).
2. Alice generates new ratchet key pair, and uses X3DH value to derive new root key (RK) and sending chain key (CK).
3. Alice uses symmetric-key ratchet to sending chain key to derive a new message key. She encrypts and sends her first message (A1), stores the new chain key and discards the message key.
4. When Alice receives a response from Bob (B1), she extracts the Bob's new ratchet public key and applies DH ratchet to derive new sending and receiving keys.
5. Alice applies symmetric-key ratchet to derive a new message key for the message decryption.

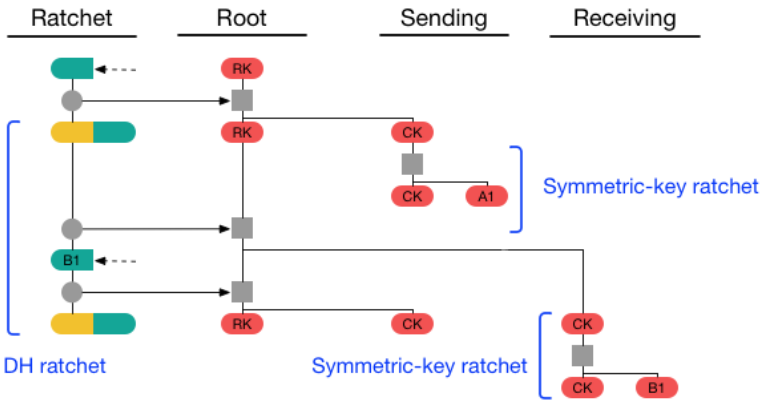


Figure 9: Alice's Double Ratchet[3]

Out of Order Messages

Double Ratchet stores the message number from the sending chain in the header, and the length of the previous sending chain. When a user receives a message that is out of order, they can skip message keys and store them for later. If the number of skipped-over message keys is equal to MAX_SKIP, the protocol throws an error and terminates.

2.6 Peer-to-Peer Network Protocol

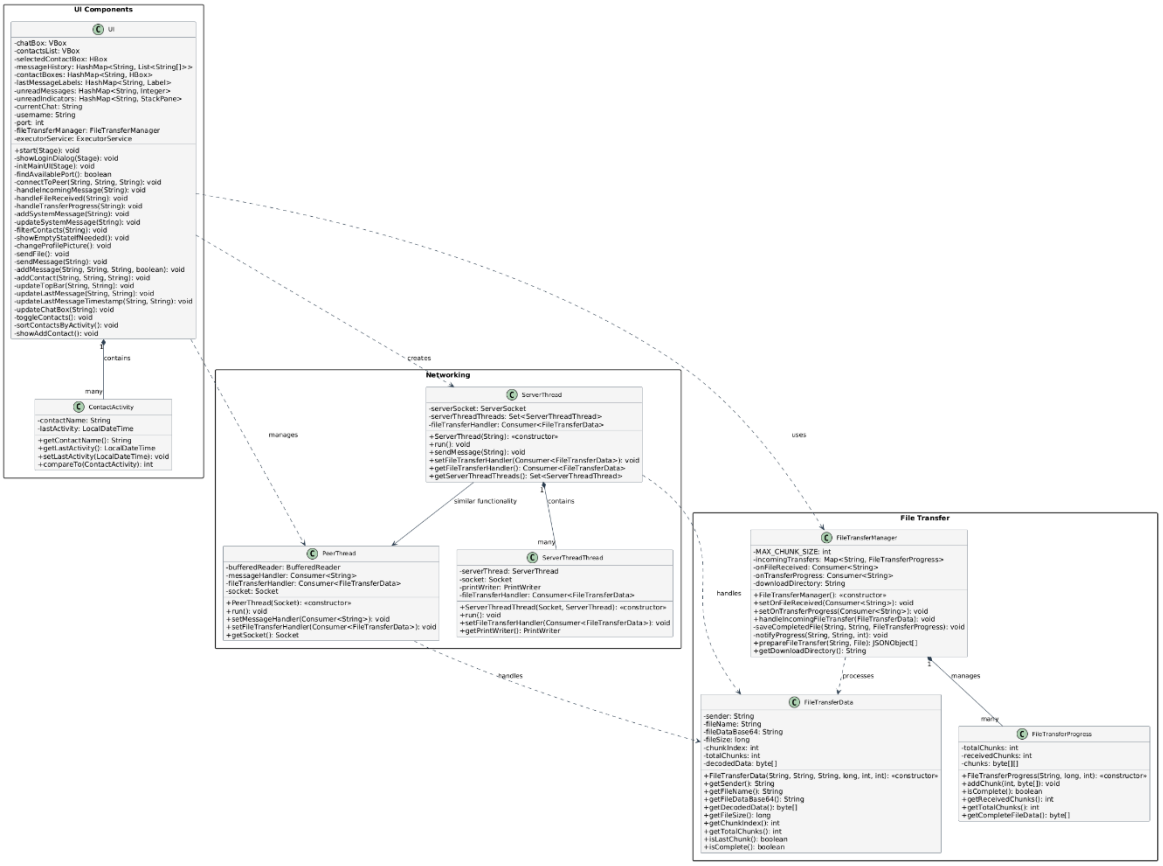
Initially our implementation of the P2P Networking Protocol was supposed to be based on the existing documentation outlined by the Tox Protocol. During the development process, a decision was made to abandon that idea, as the Tox Protocol has proven too complex for our needs and we decided to go with a simpler custom protocol.

3. Design

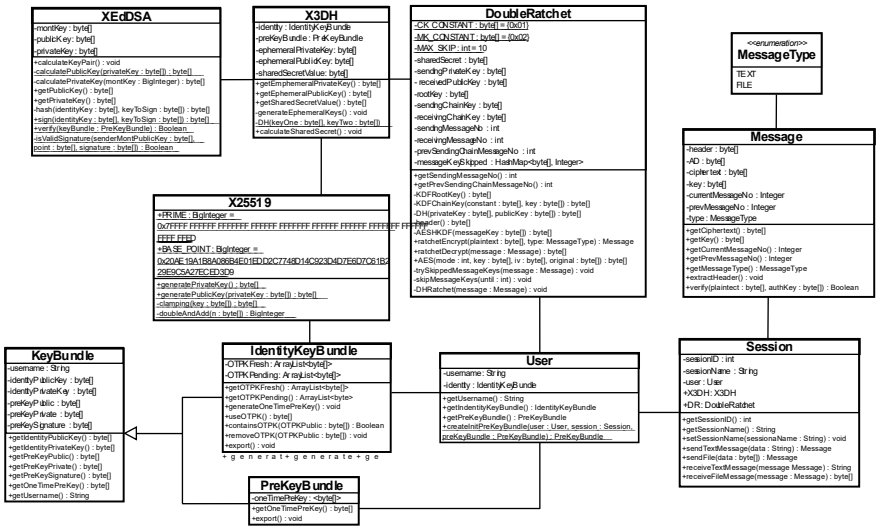
3.1 System Architecture

Networking and UI System architecture:

3.2



Cryptography architecture:



3.3 Cryptographic Protocols

Curve25519 vs Curve448

X3DH and Double Ratchet cryptographic protocols utilise elliptic curves for key generation, while XEdDSA depends on keys used in X3DH for signature creation and verification. In the specifications (see Sections 2.3 and 2.4), both Curve25519 and Curve448-based key generation functions X25519 and X448 were mentioned as possible parameters.

X25519 provides ~ a 128-bit level of security, while X448 provides ~ a 224-bit level of security. While X448, on paper, does seem to offer a higher level of security, it is unlikely that the level of security that X25519 offers will not be enough, as a brute-force attack would require a quantum computer, which would break all elliptic curve cryptography, and the trade-off between security and performance, in this case, is unnecessary.

From a signature point of view, Ed25519 is more widely supported than Ed448, and because XEdDSA converts Montgomery form signatures into equivalent Edwards form ones, selecting X25519 as the key generation curve for X3DH forced XEdDSA to utilise Ed25519.

Integration

The general design of cryptographic protocols is defined in their specifications, outlined in Section 2. The main concern was the integration. It was necessary to decide how the classes would interact with each other and what data would be passed between them. In Figure 10, dashed arrows indicate variables passed between the classes, while regular arrows – indicate methods called from the target class. The prekey bundle exported by the user becomes the central data passing mechanism

for the cryptographic protocols, as it stores the keys used both for X3DH and Double Ratchet algorithms. X25519 supplies freshly generated keys for X3DH and Double Ratchet, while XEdDSA verifies one-time prekey and user identity.

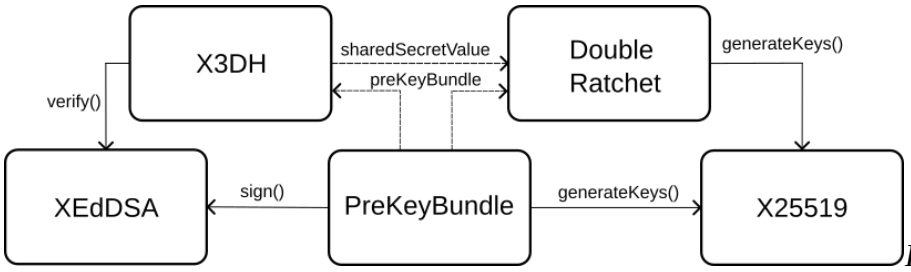


Figure 10: Cryptography Design High-Level Overview

3.4 Network Protocol

The network protocol design follows a peer-to-peer approach that eliminates the need for a central message routing server while maintaining user-friendly connection methods. Attempts have been made to implement Distributed Hash Tables and UDP Hole Punching methods mentioned in the Tox Reference, but they are not present in the final implementation.

Each application instance functions as both a server and a client:

1. **Server Component:**
    - Listens on a specific port for incoming connections
    - Accepts connection requests from other users
    - Creates a new thread for each connection to handle incoming messages
2. **Client Component:**
    - Connects to other users' server components
    - Sends messages to connected peers
    - Manages multiple outgoing connections

The application automatically searches for an available port during startup, which becomes part of the user's identity. This approach avoids port conflicts when multiple instances run on the same network and makes it easier for users to connect without manual port configuration

When a user sends a message, it is sent to desired connected peers. Each message is transferred in a JSON format to allow for easier message type identification and attach usernames to each message.

Socket Connections

The application uses Java's socket API to establish and maintain connections:

- ServerSocket for the server component
- Socket for client connections
- PeerThread and ServerThreadThread classes to manage connections

These components work together to provide message and file delivery between peers.

3.5 User Interface

The user interface is designed to provide an intuitive and familiar messaging experience while maintaining a focus on security and encryption.

Design Goals

1. **Simplicity**: Keep the interface simple and focused on messaging functionality
2. **Familiarity**: Use design patterns common in popular messaging applications
3. **Responsiveness**: Provide immediate feedback for user actions

Layout Structure

The main interface is divided into two sections. One of them is responsible for maintaining connections with other users while keeping them nicely listed out. The bigger part of the screen displays the conversation with the selected contact. This commonly used split-pane design allows users to easily switch between conversations while keeping it clear who they are chatting with at any given moment.

Visual Components

The UI employs several visual components to enhance usability:

- **Message Bubbles**: Different colours for sent and received messages
- **Profile Pictures**: Customizable user avatars
- **Unread Message Indicators**: Visual indicators for new messages
- **File Transfer Progress**: Visual feedback during file transfers

Responsive Design

The interface is designed to be responsive, with components that automatically adjust to different window sizes.

4. Implementation

4.1 Utility Class

Overview

Some operations are performed by multiple classes. To reduce the repeated implementations of the code that delivers the same functionality, we have exported these methods into a utility (Util.java) class. The utility class itself does not represent any components of the application and is merely a collection of shared methods between the classes.

Methods of the Class

- **changeEndian**

Changes the endianness of the byte array from big-endian to little-endian and vice versa by reversing the order of the bytes in the byte array. Used by multiple classes to enforce storing of data (keys, hashes, etc.) in little-endian order.

- **bigIntToByteArray**

Converts the BigInteger to byte array. Necessary due to Java's BigInteger.toByteArray() method returning byte arrays in the big-endian order along with sign bit. When changing the endianness of such a byte array, the original integer value is distorted and lost.

The bigIntToByteArray() method first converts the BigInteger into a binary string (see Figure 10). Then it ensures that the length of the binary string is divisible by 8 and of the specified length (expectedBits argument) by padding it with 0s if necessary. Once the final binary representation of the value has been obtained, the string is iterated through by 8 characters at the time and these characters are converted into bytes, which are then placed into a byte array in big-endian order. The changeEndian method is called to change the order of bytes into little-endian and the final byte array is returned.

```
/**
    Convert BigInteger into a byte array.
 *
 * @return byte array representation of BigInteger in little-endian.
 */
public static byte[] bigIntToByteArray(BigInteger bigInteger, int expectedBits) {
    String binary = bigInteger.toString(2);
    while (binary.length() < expectedBits || binary.length() % 8 != 0) {
        binary = "0" + binary;
    }
    byte[] array = new byte[binary.length() / 8];
    int arrPos = 0;
    for (int i = 0; i < binary.length(); i += 8) {
        String substring = binary.substring(i, i + 8);
        Integer numeric = Integer.parseInt(substring, 2);
        array[arrPos] = numeric.byteValue();
        arrPos++;
    }
    return Util.changeEndian(array);
}
```

Figure 11: bigIntToByteArray() Implementation in SecureYAC

- **byteArrayToString**

Converts the byte array into a hexadecimal string.

- **byteArrayToBigInteger**

Converts the byte array into Java's BigInteger by first changing the endianness of the byte array and then calling a native BigInteger constructor.

- **concatByteArrays**

Concatenates two byte arrays.

- **hash**

Hashes a byte array using a SHA-512 algorithm. Hashing method provided by the BouncyCastle's cryptographic library. The method enforces 512-bit length of the hash by padding with 0s if necessary.

- **HKDF**

Derives new keys using HKDF with SHA-512 algorithm. Algorithm provided by the BouncyCastle's cryptographic library.

- **HMAC**

Applies HMAC with SHA-512 function to input based on key provided. Algorithm provided by the BouncyCastle's cryptographic library.

- **loadFromFile**

Reads a file using Java's native Scanner implementation, and generates the appropriate KeyBundle based on the file's extension. The structure of both identity and prekey bundle files is similar – the first line will always contain the username and the second and third – keys. If the file ends in .id, an IdentityKeyBundle object is created and returned from the information read from the file. If the file is a prekey file (.pkb), the method reads two more lines and returns a PreKeyBundle.

```
public static KeyBundle loadFromFile(String fileName) throws Exception {
    KeyBundle keyBundle = null;
    File file = new File(fileName);
    Scanner reader = new Scanner(file);
    String username = reader.nextLine();
    byte[] firstKey = Util.bigIntToByteArray(new BigInteger(reader.nextLine(), 16), 256);
    byte[] secondKey = Util.bigIntToByteArray(new BigInteger(reader.nextLine(), 16), 256);
    if (fileName.contains(".id")) {
        keyBundle = new IdentityKeyBundle(username, firstKey, secondKey, 50);
    } else if (fileName.contains(".pkb")) {
        byte[] preKeySignature = Util.bigIntToByteArray(
                new BigInteger(reader.nextLine(), 16), 768);
        byte[] otpk = Util.bigIntToByteArray(
                new BigInteger(reader.nextLine(), 16), 256);
        keyBundle = new PreKeyBundle(
                username, firstKey, secondKey, preKeySignature, otpk);
    }
    reader.close();
    return keyBundle;
}
```

Figure 12: loadFromFile() Implementation in SecureYAC

- **writeToFile**

Writes data using Java's native Writer implementation into a file. This method is called by IdentityKeyBundle and PreKeyBundle classes to export their keys into files. This is a recursive method, as it first checks if a file of a given name already exists. If it does, it calls itself again, appending "_{fileCount}" to the file name and repeating the checks until a file name that does not exist yet is found. This allows the user to export the key bundles multiple times without causing any errors.

```
public static int writeToFile(String fileName, String fileType, String data,
        int fileCount) throws IOException {
    String fileFullName = fileName + fileType;
    if (fileCount != 0) {
        fileFullName = fileName + "_" + (fileCount + 1) + fileType;
    }
    File file = new File(fileFullName);
    if (!file.createNewFile()) {
        return Util.writeToFile(fileName, fileType, data, fileCount += 1);
    }
    file.createNewFile();
    FileWriter writer = new FileWriter(file);
    writer.write(data);
    writer.close();
    return 0;
}
```

Figure 13: writeToFile() Implementation in SecureYAC

4.2 X25519

Overview

The X25519.java class implements Curve25519 defined in Section 2.2. Its main purpose is to generate 32-byte long-byte arrays (keys) based on specification. To achieve this, only Java's BigInteger and SecureRandom classes are imported. It was decided against using BouncyCastle's cryptographic library's key generation implementation because some of the curve parameters are needed in further calculations in other classes and would not be available when using an external library.

All methods in this class are static, as the class itself does not store any dynamic data and therefore there is no need to initialise an object. The only data stored by this class are PRIME and BASE_POINT values used in key generation calculations.

Key Generation

As defined in specification[2], [12], X25519 keys are 32-byte length byte arrays stored in little-endian with certain bits set to specific values.

The private key is a 32-byte array filled with random bytes by Java's SecureRandom.nextBytes() method. Then, the bits in the first and last byte are clamped using the clamping() method.

The public key is generated by using the private key as an input and applying the double-and-add algorithm to it.

Clamping

```
private static byte[] clamping(byte[] key) {
    // Clamp bits 1, 2, 3 to 0.
    key[0] &= 0b11111000;
    // Clamp bit 255 to 1 and bit 256 to 0.
    key[31] |= 0b01000000;
    key[31] &= 0b01111111;
    return key;
}
```

Figure 14: clamping() Implementation in SecureYAC

Following the specification[2], [12], the bits in private key are clamped in this manner:

- The three least significant (6^{th}, 7^{th} and 8^{th}) bits oaf the first byte and the most significant (1^{st}) bit of the last byte are set to 0 by performing AND operation between bytes;
- The second most significant (2^{nd}) bit of the last byte is set to 1 by performing OR operation between bytes.

Double-and-add Algorithm

Double-and-add is an algorithm used to multiply elliptic curve points. The algorithm initialises the result variable as 0 (see Figure 14), multiplies the base point of the curve modulo prime of the curve p (as defined in Section 2.2) and stores it in a variable called addEnd. As each bit is iterated through, if the bit is 1, the addEnd value is added to the result modulo curve prime. The result is scalar (key) multiplied by the base point of the Curve25519.

```java
/**
    @param n scalar for multiplication.
    @return result of n * BASE_POINT computed using double-and-add algorithm.
 */
private static BigInteger doubleAndAdd(byte[] n) {
    BigInteger result = BigInteger.ZERO;
    BigInteger addEnd = BASE_POINT;
    for (int i = 31; i >= 0; i--) {
        for (int j = 0; j < 8; j++) {
            byte tmp = n[i];
            if (((tmp >> j) & 1) == 1) {
                result = result.add(addEnd).mod(PRIME);
            }
            addEnd = addEnd.multiply(BigInteger.valueOf(2)).mod(PRIME);
        }
    }
    return result;
}
```

Figure 15: Double-and-add Algorithm Implementation in SecureYAC

4.3 XEdDSA

Overview

XEdDSA.java implements the message signing and verifying scheme using the Ed25519 curve, as defined in Sections 2.2, 2.3, and 3.2. Because there are no libraries that support the implementation of this signature scheme, the implementation uses Java's BigInteger and SecureRandom classes, with BouncyCastle cryptographic library providing the necessary hashing algorithms (SHA-512) not supported natively by Java JDK21, which were moved into Util.java class to reduce repetitive code. This class converts Montgomery form Curve25519 keys into Edwards form, signs messages, and verifies signatures.

This class stores the Montgomery private key from which public and private Edwards form keys are derived, as well as Ed25519 base point coordinates and order (see Section 2.3).

Key Conversion

As defined in specification[4], the signing method uses twisted Edwards elliptic curve points as keys for signing. Because XEdDSA is integrated into X3DH, the input it receives is the Montgomery form key, as the same key is used for shared secret calculation and message signature creation. The key conversion method is implemented based on the pseudocode provided in Section 2.3. The private key modulo Ed25519 order has its most significant (sign) bit set to 0 (as it is impossible to recover the sign bit from Montgomery u-point, see Section 2.3).

The public key is calculated from the converted private key, by multiplying it by base point and performing a modulus order operation. The most significant (sign) bit is also set to 0.

```
private static byte[] calculatePublicKey(byte[] privateKey) {
    BigInteger edPrivateKey = Util.byteArrayToBigInteger(privateKey);
    byte[] edPublicKey = Util.bigIntToByteArray(
            edPrivateKey.multiply(BASE_POINT).mod(ORDER), 256);
    // Force sign bit to 0.
    edPublicKey[31] &= 0b01111111;
    return edPublicKey;
}

private byte[] calculatePrivateKey(BigInteger montKey) {
    byte[] edPrivateKey = Util.bigIntToByteArray(montKey.mod(ORDER), 256);
    edPrivateKey[31] &= 0b01111111;
    return edPrivateKey;
}
```

Figure 16: Montgomery to Edwards Key Form Conversion Implementation in SecureYAC

Signing

It was decided to make the signing function static, as XEdDSA only needs to provide the signature. Because it was not possible to implement a method that converts the public Montgomery key to a public Edwards key equivalent to the Edwards key used for signature creation that was derived from the private Montgomery key, it was chosen to prepend the public Edwards key to the signature itself.

The signature is created by calculating nonce + (hash * private key) modulo Ed25519 order. A concatenation of Edward's public key, point and key signature is returned as a signature.

Verifying

For signature verification, first, the function extracts data (signature, Edwards form public key and point on the curve) from the key bundle it has received. Afterwards, it checks if the extracted data is valid using a helper isValidSignature() method. By moving the validity check outside the main verification method, its length and complexity are reduced, and the code becomes more readable. Then, variable Rcheck is calculated and compared to the point, as detailed in Section 2.3. If any of the checks fail, the method returns false, and if Rcheck equals the point extracted from the key bundle, the signature is valid.

```
public static byte[] sign(byte[] identityKey, byte[] keyToSign)
        throws Exception {
    XEdDSA xEdDSA = new XEdDSA(identityKey);
    xEdDSA.calculateKeyPair();
    SecureRandom random = new SecureRandom();
    byte[] randomData = new byte[64];
    random.nextBytes(randomData);
    byte[] byteConcat = Util.concatByteArrays(
            Util.concatByteArrays(xEdDSA.privateKey, keyToSign), randomData);
    byte[] nonce = Util.bigIntToByteArray(
            Util.byteArrayToBigInteger(xEdDSA.hash(byteConcat, 1)).mod(
                    XEdDSA.ORDER), 256);
    byte[] point = Util.bigIntToByteArray(
            Util.byteArrayToBigInteger(nonce).multiply(BASE_POINT).mod(
                    XEdDSA.ORDER), 256);
    byteConcat = Util.concatByteArrays(
            Util.concatByteArrays(point, xEdDSA.getPublicKey()), keyToSign);
    byte[] hash = Util.bigIntToByteArray(
            Util.byteArrayToBigInteger(xEdDSA.hash(byteConcat, 0)).mod(
                    XEdDSA.ORDER), 256);
    // signature = nonce + (hash * privateKey) % ORDER
    byte[] signature = Util.bigIntToByteArray(
            Util.byteArrayToBigInteger(nonce)
                    .add(Util.byteArrayToBigInteger(hash).multiply(
                            Util.byteArrayToBigInteger(xEdDSA.privateKey)))
                    .mod(XEdDSA.ORDER), 256);
    return Util.concatByteArrays(
            xEdDSA.getPublicKey(), Util.concatByteArrays(point, signature));
}
```

```
public static Boolean verify(PreKeyBundle keyBundle)
        throws NoSuchAlgorithmException {
    XEdDSA xEdDSA = new XEdDSA(keyBundle.getIdentityPublicKey());
    byte[] bundleSignature = keyBundle.getPreKeySignature();
    xEdDSA.publicKey = java.util.Arrays.copyOfRange(bundleSignature, 0, 32);
    byte[] point = java.util.Arrays.copyOfRange(bundleSignature, 32, 64);
    byte[] signature = java.util.Arrays.copyOfRange(bundleSignature, 64, 96);
    if (!XEdDSA.isValidSignature(
            keyBundle.getIdentityPublicKey(), point, signature)) {
        return false;
    }
    byte[] byteConcat = Util.concatByteArrays(
            Util.concatByteArrays(point, xEdDSA.getPublicKey()),
            keyBundle.getPreKeyPublic());
    byte[] hash = Util.bigIntToByteArray(
            Util.byteArrayToBigInteger(xEdDSA.hash(byteConcat, 0))
                    .mod(XEdDSA.ORDER), 256);
    // Rcheck = (signature * BASE_POINT) - (hash * EdPublicKey)
    byte[] Rcheck = Util.bigIntToByteArray(
            (Util.byteArrayToBigInteger(signature).multiply(XEdDSA.BASE_POINT))
                    .subtract(Util.byteArrayToBigInteger(hash).multiply(
                            Util.byteArrayToBigInteger(xEdDSA.getPublicKey())))
                    .mod(XEdDSA.ORDER), 256);
    return java.util.Arrays.equals(point, Rcheck);
}
```

Figure 17: Signing and Verifying Methods in SecureYAC

4.4 X3DH

Overview

X3DH.java is the class containing the implementation of key agreement protocol X3DH. There is a need to initialise the class, as it stores unique information relating to sessions between users, such as prekey bundles and generated ephemeral keys that are used in shared secret calculations.

The class depends on XEdDSA.java, as the first operation after instance, initialisation is checking the validity of the prekey signature found in the given prekey bundle. The instance has two different "states" it could be initialised in, determined by a boolean "received" variable. If received is True, that means that the user has received an initialisation message and an associated key bundle. In this case, the object extracts the ephemeral keys from the prekey bundle before calculating the shared secret value.

If the user is the one initialising the session (received is set to False), then the object generates ephemeral keys, which will be stored in the instance.

Calculating Shared Secret

The main operation of shared secret value calculation is the Diffie-Hellman step (performed by the DH() method). This method multiplies two-byte arrays and was extracted from the main method to reduce code repetition. The first three calculations are performed the same, no matter if the user is initialising or receiving an initialisation message.

The fourth Diffie-Hellman value depends on the state of the instance. This is checked by comparing the one-time prekey included in the bundle with the user's one-time prekeys. If the prekey is found in the User object, then it is determined that the user is the one receiving the initialisation request and the first two resulting Diffie-Hellman values are concatenated in the reverse order.

If the one-time prekey is not found among the user prekey list, it is determined that the user is the one initialising the session and they concatenate the Diffie-Hellman values in order.

This check ensures that both users have the same shared secret value, as their first two calculations are reversed, depending on whether they are initialising or receiving an initialisation request (see Section 2.4).

```
public void calculateSharedSecret() throws NoSuchAlgorithmException {
    byte[] firstValue = X3DH.DH(this.identity.getIdentityPublicKey(),
            this.preKeyBundle.getPreKeyPublic());
    byte[] secondValue = X3DH.DH(
            this.ephemeralPublicKey, this.preKeyBundle.getIdentityPublicKey());
    byte[] thirdValue = X3DH.DH(
            this.ephemeralPublicKey, this.preKeyBundle.getPreKeyPublic());
    byte[] fourthValue = null;
    byte[] concatValues = null;
    byte[] OTPK = this.preKeyBundle.getOneTimePreKey();
    if (identity.containsOTPK(OTPK)) {
        fourthValue = X3DH.DH(this.preKeyBundle.getPreKeyPublic(), OTPK);
        concatValues = Util.concatByteArrays(
                Util.concatByteArrays(secondValue, firstValue),
                Util.concatByteArrays(thirdValue, fourthValue));
    } else {
        fourthValue = X3DH.DH(this.ephemeralPublicKey, OTPK);
        concatValues = Util.concatByteArrays(
                Util.concatByteArrays(firstValue, secondValue),
                Util.concatByteArrays(thirdValue, fourthValue));
    }
    this.sharedSecretValue = this.HKDF(concatValues);
}
```

Figure 18: X3DH Shared Secret Calculation Implementation in SecureYAC

4.5 Double Ratchet

Overview

The DoubleRatchet.java class is the main cryptographic class, as it connects all the previous protocols to encrypt and decrypt messages. As specified in Section 2.5, it keeps track of multiple variables and keys.

As this protocol is integrated with X3DH (which in turn uses XEdDSA signature scheme implementation), the shared secret calculated by the key agreement protocol is used as the initial root key.

X25519.java methods generate new sending keys in the Diffie-Hellman ratchet step.

HKDF and HMAC algorithms, provided by the BouncyCastle cryptographic library are used in this class for root, chain and message key derivation. They have been moved to the Util.java class to minimise the amount of code in the DoubleRatchet.java, as these methods only deal with instantiating the algorithms and producing appropriate output.

Encryption

For encryption, as recommended in the protocol specification[3], Advanced Encryption Standard (AES) algorithm initialised in cipher-block chaining (CBC) mode with PKCS7 padding is used. BouncyCastle is used as a security provider, as Java natively does not support the PKCD7 padding scheme. An HKDF function specifically for AES generates an initialisation vector and authentication key. While the specification mentions that additional data (AD) from the X3DH protocol becomes the initial AD in Double Ratchet, this recommendation is ignored and each encryption run generates additional data, which is added to the Message.

Decryption

For decryption, a message key is derived from the receiving chain key, the authentication key and the initialisation vector using the same AES HKDF method as for encryption. The authentication

key is used for message verification and, if the message is valid, it is decrypted. Otherwise, the method throws a general Exception.

```
public Message ratchetEncrypt(byte[] plaintext, MessageType type) throws Exception {
    this.sendingChainKey = this.KDFChainKey(CK_CONSTANT, this.sendingChainKey);
    byte[] messageKey = this.KDFChainKey(MK_CONSTANT, this.sendingChainKey);
    byte[] hkdfOut = this.AESHKDF(messageKey);
    byte[] authenticationKey = java.util.Arrays.copyOfRange(hkdfOut, 0, 32);
    byte[] iv = java.util.Arrays.copyOfRange(hkdfOut, 32, 48);
    this.sendingMessageNo += 1;
    byte[] AD = Util.HMAC(plaintext, authenticationKey);
    return new Message(type, this.header(), AD,
            AES(Cipher.ENCRYPT_MODE, messageKey, iv, plaintext));
}

public byte[] ratchetDecrypt(Message message) throws Exception {
    message.extractHeader();
    this.receivingChainKey = message.getKey();
    this.trySkippedMessageKeys(message);
    if (message.getPrevMessageNo() != this.prevSendingChainMessageNo) {
        this.skipMessageKeys(message.getPrevMessageNo());
        this.DHRatchet(message);
    }
    this.skipMessageKeys(message.getCurrentMessageNo());
    byte[] messageKey = this.KDFChainKey(MK_CONSTANT, this.receivingChainKey);
    byte[] hkdfOut = this.AESHKDF(messageKey);
    byte[] authenticationKey = java.util.Arrays.copyOfRange(hkdfOut, 0, 32);
    this.receivingMessageNo += 1;
    byte[] iv = java.util.Arrays.copyOfRange(hkdfOut, 32, 48);
    byte[] plaintext = AES(Cipher.DECRYPT_MODE, messageKey, iv, message.getCiphertext());
    if (message.verify(plaintext, authenticationKey)) {
        return plaintext;
    }
    throw new Exception();
}
```

Figure 19: Encryption and Decryption Methods in DoubleRatchet.java in SecureYAC

4.6 Network Protocol

The network protocol is implemented through several Java classes that work together to establish and maintain connections between peers.

ServerThread Class

The ServerThread class represents the server component of the application. It is responsible for initialising a server socket on a specified port and it accepts incoming connections and creates a new thread for each one

PeerThread Class

The PeerThread class manages individual peer connections It reads incoming data from the socket connection, parses JSON messages to determine message type, delegates message handling to appropriate handlers and handles exceptions and cleanup when connections terminate.

Automatic Port Assignment

The application automatically finds an available port during startup:

```
private boolean findAvailablePort() {
    int attempts = 0;
    while (attempts < MAX_PORT_ATTEMPTS) {
        int portToTry = NEXT_PORT.getAndIncrement();
        try (ServerSocket socket = new ServerSocket(portToTry)) {
            // Port is available, use it
            this.port = portToTry;
            return true;
        } catch (IOException e) {
            // Port is in use, try the next one
            attempts++;
        }
    }
    return false;
}
```

This method tries to bind to ports sequentially starting from a base port number and continues until it finds an available port or reaches the maximum number of attempts.

Connection Establishment

The application establishes connections to peers using their address and port:

```
private void connectToPeer(String host, String portStr, String peerAddress) {
    Socket socket = null;
    try {
        int portNum = Integer.parseInt(portStr);
        socket = new Socket(host, portNum);
        PeerThread peerThread = new PeerThread(socket);
        peerThread.setMessageHandler(this::handleIncomingMessage);
        peerThread.setFileTransferHandler(fileTransferManager::handleIncomingFileTransfer);
        peerThread.start();

        // Store the peer thread for later reference
        activePeers.put(peerAddress, peerThread);

        // Add the peer as a contact if not already added
        Platform.runLater(() -> {
            if (!messageHistory.containsKey(peerAddress)) {
                addContact(peerAddress, "Connected", APP_ICON_PATH);

                // New contact - add to activity tracker
                contactActivities.put(peerAddress, new ContactActivity(peerAddress, LocalDateTime.now()));
                sortContactsByActivity();
            }
        });
```

This method:

1. Creates a socket connection to the specified host and port
2. Initializes a PeerThread to handle the connection
3. Sets up message and file transfer handlers
4. Updates the UI to reflect the new connection
5. Handles error cases with appropriate user feedback

File Transfer Protocol

The file transfer functionality is implemented through the FileTransferManager class, which handles sending and receiving files over the network.

FileTransferManager Class

The FileTransferManager class manages file transfers. This implementation breaks large files into manageable chunks, encodes each chunk as Base64 to ensure reliable transmission as text. When the file is being transferred, it tracks progress of incoming file transfers, after which it reassembles the file and saves it on the recipient's computer.

FileTransferData Class

The FileTransferData class represents the data for a single chunk of a file transfer. By storing metadata about a file chunk and providing helper methods to determine if a chunk is the last one or if the transfer is complete, the FileTransferManager can transfer whole files without any issues.

4.7 User Interface

The UI was fully implemented using JavaFX. The biggest regret in the way it is implemented is that slightly too late into the project I realised splitting the UI class into multiple smaller chunk classes would be a good idea.

The first thing our user is greeted with is the Login Dialog, where they are able to set their username and profile picture, which will later be used to identify them in the contact lists of other people. In case their username is empty, they will be asked to input something. Upon successful identity creation, our user proceeds to the main UI with all the important interactable elements such as the contacts list and the chat area. showEmptyStateIfNeeded() is responsible for showing the user the welcome message view when they log in for the first time and have no contacts added.

4.8 Helper Classes

While DoubleRatchet.java unites the various cryptographic protocols, to facilitate data transmission between classes and instances of these classes, helper classes are utilised:

- **IdentityKeyBundle.java -** extends the KeyBundle class and stores information related to the user identity. May be exported into a file as an identity backup, as well as imported into the application to move user identity across devices. If the same IdentityKeyBundle is exported multiple times into the same directory, the method appends a suffix indicating that the file is a copy of an already existing file(s).
- **KeyBundle.java** – a general key bundle class that represents a general key bundle and data stored in it. Used to minimise the repetition and amount of code in key bundle classes that extend this class.
- **Message.java** – a class, whose instances are exchanged between Users during the Session. It stores the header, additional data, ciphertext and associated key. Two types of Messages exist – text and file. The main difference between types is the final format in which plaintext is represented after decryption – text message plaintext is converted into a String and file type Messages are kept as byte arrays. Message class can verify the message contents by performing HMAC on plaintext and

authentication key. If the output matches additional data (AD) within the Message, the validation passes.

```
public Boolean verify(byte[] plaintext, byte[] authKey) {
    byte[] AD = Util.HMAC(plaintext, authKey);
    return java.util.Arrays.equals(this.AD, AD);
}
```

Figure 20: Message Verification Method in Message.java

- **PreKeyBundle.java** – implements KeyBundle class and stores information needed to initialise a Session with the User who exported the key bundle. May be exported into a file to exchange with another user through a secure communications channel for Session initialisation. If the same PreKeyBundle is exported multiple times into the same directory, the method appends a suffix indicating that the file is a copy of an already existing file(s).

- **Session.java** – used to store information about a messaging session between two Users. This is the main class that facilitates the integration of the Double Ratchet algorithm and X3DH key agreement protocol on its initialisation.

- **User.java** – this class stores information related to the user, such as username and identity bundle containing identity keys.

5. Testing

5.1 Ad Hoc Testing

During the development, while the unit tests were written as the implementation benchmark, some ad hoc testing was needed to ensure the bare minimum functionality before more formal testing could be conducted. With this informal and unstructured testing, some issues were caught early in the development process.

Sign Bits and Coin Flip Verification

During the implementation of XEdDSA signature scheme, an odd bug was discovered – the verify() method returned True for valid signatures only 50% of the time. This meant that a valid signature may not be verified. To investigate this issue, first, the logic behind the verification calculations was investigated. After determining that the calculations are correct (since a valid signature would get verified half the time), the variables passed to the method were investigated next.

```
Signature: dd5516dcc3bcbd2b2413b10135a469178f04080072e5f42b3526b3372a77fa7, 32
Sign PublicKey: d64b7e2beaad2d30c1738c61489ca672713038ab1c2fb99d1712d7ff84ff2c, 32
Sign Point: d320082097217fe1ec8122fe65e6a19b0c543176c8a1633d4620e60f7ab50a3
Sign PreKey: 31774e95de33dd642ba01f8cab9a78c9794a9bf248ed63d2a3a3930f3ad9f3cd, 32
Sign hash: 820be110ea776eb20f917028c3ae737e92c40060480183f77fb4ec4ee5e9c3b, 32
Received Signature: dd5516dcc3bcbd2b2413b10135a469178f04080072e5f42b3526b3372a77fa7, 32
Verify PublicKey: d64b7e2beaad2d30c1738c61489ca672713038ab1c2fb99d1712d7ff84ff2c, 32
Verify Point: d320082097217fe1ec8122fe65e6a19b0c543176c8a1633d4620e60f7ab50a3
Rcheck: d9109095d2d8d8e9b53937603d20608fa47d4924bb76383755e47268c8e9ef6, 32
Verify PreKey: 31774e95de33dd642ba01f8cab9a78c9794a9bf248ed63d2a3a3930f3ad9f3cd, 32
Verify hash: 820be110ea776eb20f917028c3ae737e92c40060480183f77fb4ec4ee5e9c3b, 32
Signature Valid?: false
```

The values in both signing and verifying functions matched up (see Figure 20), but after the calculations, Rcheck was different than the Point. This eliminated the signing and verifying methods as the places where the bug was.

Since the verification failed only 50% of the time, this suggested that the issue may lie within the binary representation of these values, as they were stored as byte arrays. After printing out the same data in binary, the key conversion method was investigated next. It was found, that due to confusion related to endianness of the byte arrays, the method was clamping the wrong bit to 0, so the sign bit was not set as according to the specification. After fixing the key conversion method, the bug disappeared.

5.2 Unit Tests

During the development, each class had a set of unit tests using JUnit testing framework written for it. The tests were created to ensure that methods written in the classes produced expected output. The testing ranged from checking if the correct bits were clamped in X25519 key generation to asserting that decrypted ciphertext from Double Ratchet algorithm matches the original plaintext. For methods that rely on random byte generation, the unit tests were set to be repeatable. The unit tests performed for each class are:

- **DoubleRatchet.java**
  - *testEncryptAndDecryptByteArray()* - assert correct Double Ratchet encryption and decryption of a byte array;
  - *testEncryptAndDecryptString()* - assert correct Double Ratchet encryption and decryption of a String;
  - *testAES()* - assert correct and error-less AES encryption and decryption of a byte array.
- **IdentityKeyBundle.java**
  - *testInitOTPK()* - Class initialisation, assert a proper amount of keys is generated based on parameter;
  - *testGenerateOTPK()* - assert that when additional one-time prekey is generated, it is appended to the one-time prekey HashMap;
  - *testUseOTPK()* - assert that when a one-time prekey is used, it is moved from fresh one-time prekey HashMap into pending one-time prekey HashMap;
  - *testExport()* - test exporting IdentityKeyBundle into a .id file. Fails if the method does not create a .id file;
  - *testMultiExport()* - test exporting the same IdentityKeyBundle multiple times. Fails if less than 10 .id files are created by the method or if the export() method throws an Exception.
- **Message.java**
  - *testHeaderExtract()* - assert that the current and previous sending chain message numbers are extracted correctly from the Message's header;
  - *testVerify()* - assert that a Message can be verified with a correct authentication key;
  - *testBadVerify()* - assert that a Message fails verification with modified authentication key.

- **PreKeyBundle.java**
  - *testExport()* - test exporting PreKeyBundle into a .pkb file. Fails if the method does not create a .pkb file;
  - *testMultiExport()* - test exporting the same PreKeyBundle multiple times. Fails if less than 10 .pkb files are created by the method or if the export() method throws an Exception.
- **Util.java**
  - *testChangeEndian()* - assert that if a method is used on a byte array twice, that the final byte array is the same as the original one. Repeated test, as the method uses randomly generated values;
  - *testByteArrayToBigIntToByteArray()* - assert that if a byte array is converted to a BigInteger and back to byte array, that the resulting byte array is the same as the original. Repeated test, as the method uses randomly generated values;
  - *testConcatByteArrays()* - assert that the byte arrays are concatenated in the correct order;
  - *testHash()* - assert that the hash function produces output of the correct length and does not throw any Exceptions. Repeated test, as the method uses randomly generated values;
  - *testHKDF()* - assert that HKDF method produces the same outputs when given the same inputs;
  - *testHMAC()* - assert that HMAC method produces the same outputs when given the same inputs;
  - *testLoadFromIdFile()* - assert that the method for importing files can correctly read a .id file and create a corresponding IdentityKeyBundle with values placed in correct variables. Repeated test, as the method uses randomly generated values;
  - *testLoadFromPkbFile()* - assert that the method for importing files can correctly read a .pkb file and create a corresponding PreKeyBundle with values placed in the correct variables. Repeated test, as the method uses randomly generated values.
- **X25519.java**
  - *testKeyLength()* - assert that the byte arrays used to store public and private keys are 32-byte long. Repeated test, as the method uses randomly generated values;
  - *testClamping()* - assert that the necessary bits are always clamped to 0 or 1 as according to the specification. Repeated test, as the method uses randomly generated values.
  - *TestPublicKeyGeneration()* - assert that the same public key is generated when given the same private key. Repeated test, as the method uses randomly generated values.
- **X3DH.java**
  - *testSharedSecretCalculation()* - assert that two users obtain the same value after performing Diffie-Hellman calculations. Repeated test, as the method uses randomly generated values.
- **XEdDSA.java**

- *testKeyLength()* - assert that the Edward's form public key that was calculated from the Montgomery form private key is 32-byte long. Repeated test, as the method uses randomly generated values;
- *testKeySignBit()* - assert that the sign bit (the least significant bit) of the public key is set to 0. Repeated test, as the method uses randomly generated values;
- *testSignatureLength()* - assert that the signature generated by the scheme is 96-bytes long. Repeated test, as the method uses randomly generated values;
- *testSignAndVerify()* - assert that an unmodified prekey signature passes the verification. Repeated test, as the method uses randomly generated values;
- *testBadSignAndVerify()* - assert that a modified prekey signature fails the verification. Repeated test, as the method uses randomly generated values.

5.3 Regression Testing and CI/CD Pipeline

To ensure continued functionality of implemented features, the unit tests detailed in Section 5.1 were ran every time a commit was pushed into project's repository. This was achieved by creating a GitLab's pipeline script. The pipeline first runs a Gradle build job, which ensures that the code can be compiled successfully. If the first job is successful, the pipeline runs unit tests using Gradle test command.

Throughout the development, over 60% of the pipelines succeeded. By continuously running unit tests that have passed in the past, we were able to ensure that the application's functionality was preserved when new functionality was added.

5.4 Integration Testing

An integration test was written to simulate a message exchange between two users – Alice and Bob. This test ensures that the cryptographic protocols and message exchange are working properly and allowed us to catch issues and broken functionality before pushing the commit into the repository.

5.5 User Evaluation Testing

User Evaluation was done on a small group of 3 technically fluent individuals. Everyone was tasked with easy to complete sequences of actions: adding a specific contact, finding and messaging a contact, exchanging files with another user. They reported that despite its simplicity, they were somewhat satisfied with the UI design and the application's responsiveness. They all noticed the networking side's shortcoming in the form of added users appearing with names of format (ip:port), which they did not like. This highlights the need of improving the connection establishing process between peers so that the usernames appear in the contacts list immediately after connecting, as opposed to after exchanging a message. One user mentioned that they would appreciate the addition of adjustable colours schemes such as the night theme. Admittedly, our only available design is too bright for long sessions.

6. Results

The final version of this application has the main functionality described in the functional specification. The users are able to create a unique identity using minimalistic and easy-to-use user interface, import and export key bundles required to establish a connection, initialise a conversation with another user through the Internet and exchange encrypted messages and files.

The application passes designed unit and integrations tests, with the repository's pipeline finishing its execution within seconds without failure.

7. Future Work

In the future, more testing should be performed to ensure that the application is cryptographically secure and to identify potential faults in cryptography. More consideration is required for user input (ex. usernames, message text, etc.) and ways to secure application from potential malicious inputs. A file compression algorithm could be used to increase the speed of file transfer in peer-to-peer network. While the current user interface is sufficient for the purpose of this application, more user evaluation and feedback could be used to improve it even further. Furthermore, more work needs to be done to ensure cross-platform compatibility. In the future, we might consider expanding our network protocol to involve improved user discovery methods such as Distributed Hash Tables and UDP hole punching. Improved connection mechanism could be used to incorporate friend requests as opposed to requiring both users to add each other with their IP addresses.

8. References

[1] M. Driscoll, 'Hands-on: X25519 Key Exchange'. Accessed: Apr. 08, 2025. [Online]. Available: https://x25519.xargs.org/

[2] M. Marlinspike, 'The X3DH Key Agreement Protocol'. Signal, Nov. 04, 2016. Accessed: Mar. 25, 2025. [Online]. Available: https://signal.org/docs/specifications/x3dh/

[3] M. Marlinspike, 'The Double Ratchet Algorithm'. Signal, Nov. 20, 2016. Accessed: Mar. 25, 2025. [Online]. Available: https://signal.org/docs/specifications/doubleratchet/

[4] T. Perrin, 'The XEdDSA and VXEdDSA Signature Schemes'. Signal, Oct. 20, 2016. Accessed: Apr. 03, 2025. [Online]. Available: https://signal.org/docs/specifications/xeddsa/

[5] 'Elliptic curve', *Wikipedia*. Mar. 17, 2025. Accessed: Apr. 08, 2025. [Online]. Available: https://en.wikipedia.org/wiki/Elliptic_curve

[6] H. C. A. van Tilborg and S. Jajodia, Eds., *Encyclopedia of cryptography and security*, 2nd ed. in Springer reference. New York: Springer, 2011.

[7] D. J. Bernstein, 'Curve25519: New Diffie-Hellman Speed Records', in *Public Key Cryptography - PKC 2006*, vol. 3958, M. Yung, Y. Dodis, A. Kiayias, and T. Malkin, Eds., in Lecture Notes in Computer Science, vol. 3958. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 207–228. doi: 10.1007/11745853_14.

[8] D. J. Bernstein, N. Duif, T. Lange, P. Schwabe, and B.-Y. Yang, 'High-speed high-security signatures', *J. Cryptogr. Eng.*, vol. 2, no. 2, pp. 77–89, Sep. 2012, doi: 10.1007/s13389-012-0027-1.

[9] D. J. Bernstein, S. Josefsson, T. Lange, P. Schwabe, and B.-Y. Yang, 'EdDSA for more curves'. Cryptology {ePrint} Archive, Jul. 05, 2015. Accessed: Apr. 13, 2025. [Online]. Available: https://eprint.iacr.org/2015/677.pdf

[10] J. Brendel, C. Cremers, D. Jackson, and M. Zhao, ‘The Provable Security of Ed25519: Theory and Practice’, in *2021 IEEE Symposium on Security and Privacy (SP)*, San Francisco, CA, USA: IEEE, May 2021, pp. 1659–1676. doi: 10.1109/SP40001.2021.00042.

[11] M. Hamburg, ‘Ed448-Goldilocks, a new elliptic curve’. 2015. Accessed: Apr. 20, 2025. [Online]. Available: https://eprint.iacr.org/2015/625.pdf

[12] A. Langley, M. Hamburg, and S. Turner, *Elliptic Curves for Security*, Jan. 2016. Accessed: Apr. 08, 2025. [Online]. Available: https://www.rfc-editor.org/rfc/rfc7748