Chapter 1

Introduction

Loopix [**?**] is an anonymous communication system that provides medium-latency, low-bandwidth communication with strong anonymity. Loopix is traffic analysis resistant to both active attackers and global passive adversaries. This is achieved by utilising cover traffic and Poisson mixing which adds brief independent message delays.

I have successfully created a Java implementation of the Loopix client that is compatible with the original Python implementation. I also provide an analysis of my client and Loopix by measuring bandwidth and latency overhead and discussed the viability of Loopix on mobile devices.

1.1 Motivation

Anonymous communication systems such as Tor [**?**] allows users to communicate without revealing the identities of users to third parties. These are becoming increasingly important in the current age of pervasive data collection, surveillance and censorship, allowing for privacy and anonymity when parties are trying to collect as much data as possible on individuals, either for commercial exploitation or government surveillance. Such systems enable citizens in oppressive regimes to access censored material. Whistleblowers use anonymous communications to securely communicate with media organisations to protect their identity. Non-governmental organisations such as human rights activists can use such systems to avoid persecution while conducting activities. There are many use cases for anonymous communication systems, and these are just a few of possible uses.

However, the most widely used anonymous communication system, Tor, is vulnerable to attacks such as traffic correlation by a global passive adversary and corrupt nodes performing active attacks to deanonymise users. A global passive adversary is an adversary that is able to observe all traffic traversing all communication channels. This ability allows such an adversary to correlate the sender's and recipient's traffic patterns and linking the two together. Government agencies such as the National Security Agency (NSA) and Government Communications Headquarters (GCHQ) have already demonstrated the

ability to deanonymise a small fraction of Tor users [**?**]. Alternative systems that are not vulnerable to a global passive adversary tend to be high-latency, low-bandwidth, which severely limits possible applications due to the anonymity trilemma of choosing two out of strong anonymity, low bandwidth overhead, or low latency [**?**].

The medium-latency property Loopix should allow for communication applications such as instant messaging and email which are not very latency sensitive. A useful feature of Loopix is the ability to store messages for offline clients. Clients running mobile devices tend to have sporadic connections, which means such clients are not always connected to the network.

1.2 Related Work

Anonymous communication systems can be broadly categorised into either connection-based onion routing systems or message-based mix networks. The most popular architecture is onion routing as used in Tor, with various systems extending this paradigm. Message-based mix networks such as Mixminion [**?**], which Loopix is loosely based upon, on the other hand, are currently unfashionable due to the high-latencies involved.

The plethora of anonymous communication system designs shows how difficult it is to design an anonymous communication system with acceptable trade-offs for many different applications. Some require many clients to provide anonymity, which is not ideal as users would only use the system if it provides anonymity. Loopix attempts to cater for medium- to high-latency applications such as instant messaging and email, and not target very low-latency applications such as web browsing. Loopix's parameters can also be adjusted according to the number of participating users to maintain anonymity even when there are very few users.

The Loopix authors have already completed a rigorous analysis of Loopix and have a working implementation in Python. This meant I was able to use similar evaluation methodologies to the original paper [**?**], and their open source Python implementation[1] was crucial in developing my own implementation.

Loopix has since been worked on and improved by the PANORAMIX project, and the end result is a more robust mix network Katzenpost[2]. Katzenpost is based on Loopix, and adds a consensus-based public key infrastructure, reliable message delivery, and single-use replies.

[1] `https://github.com/UCL-InfoSec/loopix`

[2] https://katzenpost.mixnetworks.org/

Chapter 2

Preparation

This chapter begins by providing Loopix's threat model, providing a high-level overview of how Loopix is structured, a background into some of the cryptography, and details about the development environment and software engineering techniques used.

2.1 Threat Model

Loopix's threat model assumes adversaries that are well-resourced and concerned with linking users to their communications, or their partners. Such adversaries are envisioned to have three capabilities:

- **Global Passive Adversary (GPA)** A GPA is able to observe all network traffic in the system, and thus all information flowing between nodes.
- **Corrupt Mix Nodes/Providers** The adversary has the ability to compromise and corrupt mix nodes, or run malicious mix nodes. Such a mix node may tamper with messages by dropping or delaying messages. It may also inject messages into the system. The adversary also has access to all of the secrets associated with the nodes.
- **Insider** The ability to participate in the system as a user, and may deviate from the protocol. This is extended to assume the adversary is able to participate in a conversation with an honest user, and thus become a conversation insider.

Loopix attempts to provide the following security properties:

- **Sender-receiver third-party unlinkability** An adversary should not be able to infer that two users are communicating.
- **Sender online unobservability** An adversary should not be able to infer that a sender is actively communicating.
- **Sender anonymity** An adversary should not be able to distinguish between two possible senders communicating with a receiver.

- **Receiver unobservability** An adversary should not be able to infer that a receiver is actively communicating.
- **Receiver anonymity** An adversary should not be able to distinguish between two possible receivers communicating with a sender.

Loopix provides these properties against GPAs and corrupt mixes. However, a corrupt provider will be able to determine if a receiver registered under it is actively sending and/or receiving messages.

2.2 Structure of Loopix

In this section, I describe the Loopix network architecture and the Poisson Mix theory. An overview of the network architecture is shown in Figure 2.1.

2.2.1 Network Architecture

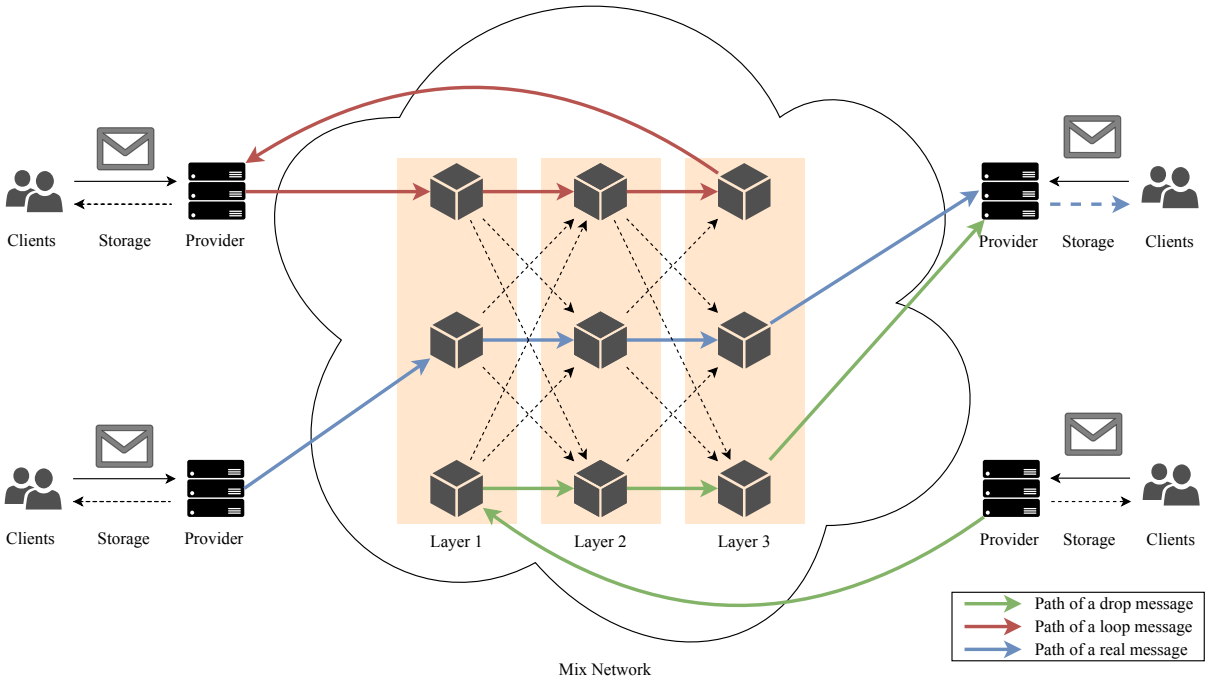

Figure 2.1: Overview of the Loopix network architecture. Clients pass messages to their providers, which are responsible for injecting the message into the mix network. The received messages are stored in inboxes at providers and retrieved by clients when they come online.

The Loopix network is composed of three parts: clients, mix nodes and providers. A client can communicate through the Loopix network and can act as a sender and receiver of messages. Each entity in the Loopix network has a unique public-private key pair that

is used to encrypt and decrypt messages. The mix nodes are separated into layers, with each layer forwarding messages to the next layer.

For a sender to send a message to a receiver, the sender needs to know the receiver's Loopix network location, that is, the IP address of the receiver's provider, an identifier of the user, and the receiver's public encryption key. The sender also needs to know the network locations of intermediate mix nodes as Loopix is source routed and thus the sender is responsible for selecting the route through the network.

2.2.2 Poisson Mix

Loopix employs a strategy called the Poisson Mix to prevent observers from learning the correspondences between incoming and outgoing messages at a node, therefore guarding against a global passive adversary performing traffic analysis attacks.

When a mix packet arrives at a mix node, the mix node decodes and extracts the subsequent mix packet to forward on. The decoded message includes a delay parameter which specifies how long to delay the forwarding of the packet. The source of the message determines the delay parameter. Honest clients choose this delay by sampling from an exponential distribution with a parameter λ that is assumed to be public and the same for all mix nodes.

Since honest nodes generate cover traffic, loop traffic, and real traffic following a Poisson process, aggregating these traffic streams at the input of a mix node produces another Poisson process with a rate λ_m dependent on the number of mix nodes and clients.

As this input process is a Poisson process, and each message is independently delayed using an exponential distribution with parameter λ, a Poisson Mix can be modelled as an $M/M/\infty$ queueing system since both input and output of the mix node are Poisson processes. As a result of the memoryless property of such a system, messages are indistinguishable from each other, since messages are emitted with equal probability regardless of the amount of time they have been waiting in the queue.

This provides the strong anonymity properties of the system, including a defence against a global passive adversary.

2.3 Requirements Analysis

The project requires that an implementation and evaluation of Loopix be produced. Java is chosen as the language of choice for the library, and graph generation will be done with Python. The project can be broken down into four parts:

- **Sphinx library** The packet format used by Loopix. There was an existing Scala implementation[1] created as part of a Part II project, but that implementation was incompatible with the Python library.
- **Loopix client** The client handles communication with the network and generating traffic using the Poisson mix strategy.
- **Demonstration application** Any application that uses the client for communication, such as instant messaging or email.
- **Test framework** A framework for testing my libraries, with respect to correctness and performance.

As there is already an implementation of Loopix, and binary compatibility is required, I have to refer to the behaviour of the implementation rather than the paper. This is because the paper does not describe sufficient implementation detail.

The choice of Java is a result of the aim of getting Loopix on Android devices. Java is the primary language used on Android. Android supports running native code such as C++, which would result in a more portable library. However, Java was chosen as a pure-JVM library would be easier to integrate into an Android application.

2.4 Cryptography

Loopix makes use of elliptic curve cryptography for exchanging keys using the Diffie-Hellman protocol and the LIONESS block cipher for encrypting messages.

2.4.1 Elliptic Curves

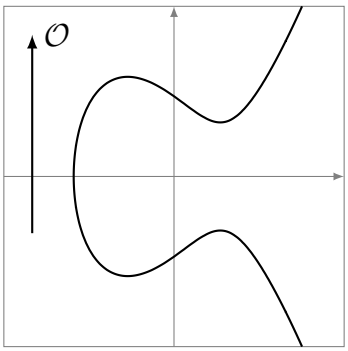

Figure 2.2: An elliptic curve with parameters $A = -2$ and $B = 2$, with the neutral element $\mathcal{O}$ at infinity.

Elliptic curve cryptography (ECC) is used in public-key cryptography. ECC is based on elliptic curves groups, which are sets of 2D coordinates (x, y) of the form $y^2 = x^3 + Ax + B$

[1] https://github.com/ndavison21/ScalaSphinx

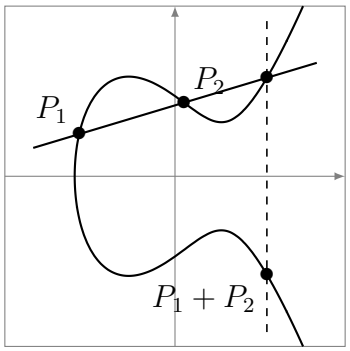

Figure 2.3: The addition of points P_1 and P_2.

and an additional point $\mathcal{O}$ at infinity with an example shown in Figure 2.2. The key group operation addition $P_1 + P_2$ is defined as: draw a line through P_1, P_2, and the line should intersect the curve at a third point P_3. Negate the y coordinate of P_3 to get $P_1 + P_2$. The addition operation is illustrated in Figure 2.3.

Each curve used in ECC has a base point which is the group generator G. Private keys are a random integer x, and the corresponding public key is given by

$$G^x = \underbrace{G + G + \cdots + G}_{x \text{ times}}$$

.

ECC can be used to perform Diffie-Hellman key exchange to share secrets over a public communication channel without revealing the secret to an observer.

Consider two parties, Alice and Bob. Both Alice and Bob generate a public-private key-pair PK_A, SK_A and PK_B, SK_B respectively and share their public keys.

Bob receives Alice's public key PK_A, which is of the form $PK_A = G^{SK_A}$. Bob generates a shared key by exponentiating Alice's public key with his private key $(G^{SK_A})^{SK_B}$. Alice does the same with Bob's public key $(G^{SK_B})^{SK_A}$. From the associativity of multiplication in a finite field, these two are equal, that is:

$$(G^{SK_A})^{SK_B} = (G^{SK_B})^{SK_A} = G^{SK_A * SK_B}$$

The Diffie-Hellman protocol is illustrated in Figure 2.4.

For an attacker to recover $G^{SK_A * SK_B}$ from only G^{SK_A} and G^{SK_B}, the two messages that are sent over a public channel and known to adversaries requires solving the computational Diffie-Hellman problem, which is believed to be computationally hard.

Loopix generates a random *blinding factor* b for establishing a shared secret key between two parties. The blinding factor is treated as the private key of one party, and G^b is the public key that is sent over the network.

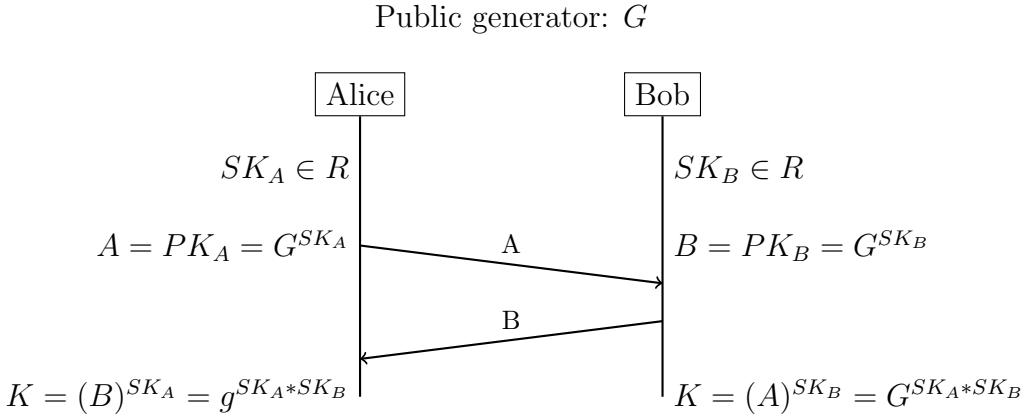

Figure 2.4: The Elliptic-Curve Diffie-Hellman protocol.

2.4.2 LIONESS Wide Block Cipher

LIONESS is a provably secure wide block cipher [**?**]. It is parametrised with a stream cipher (AES-128-CTR) and a hash function (SHA-256). LIONESS supports arbitrary sized blocks, such that the whole message is treated as a single block. This ensures that any change in the ciphertext would prevent the decryption of the ciphertext. This prevents the tagging of encrypted messages, where an attacker is able to modify the ciphertext and observe the same modification in the plaintext. The tagging attack can be used to recognise specific traffic in the network.

Internally, LIONESS is a four round unbalanced Feistel cipher, the structure of which is shown in Figure 2.5. LIONESS is supposed to take as inputs 4 rounds keys K_1, K_2, K_3, K_4, each the length of the hash function output. However, this project uses a variant of LIONESS that takes a single key K and derives from K and the plaintext the required round keys.

2.5 Development Tools

A good development environment is essential to productivity. For Java development, the IntelliJ IDEA IDE is used, with the Gradle build system and JUnit for unit tests. Gradle handles dependency management as well. Travis CI is a service that automatically builds projects and is used to run tests on commit to the Git repository. This ensures code committed to the Git repository can be compiled and passes all tests. Visual Studio Code is used for writing any Python and shell scripts. Both IDEs provide syntax highlighting, code auto-completion, and code analysis to increase productivity.

Revision control is managed using Git, with remote repositories hosted on GitHub, Bitbucket, and my server to serve as backups. Secondary backups are done by periodically syncing the project repository onto Dropbox and Google Drive.

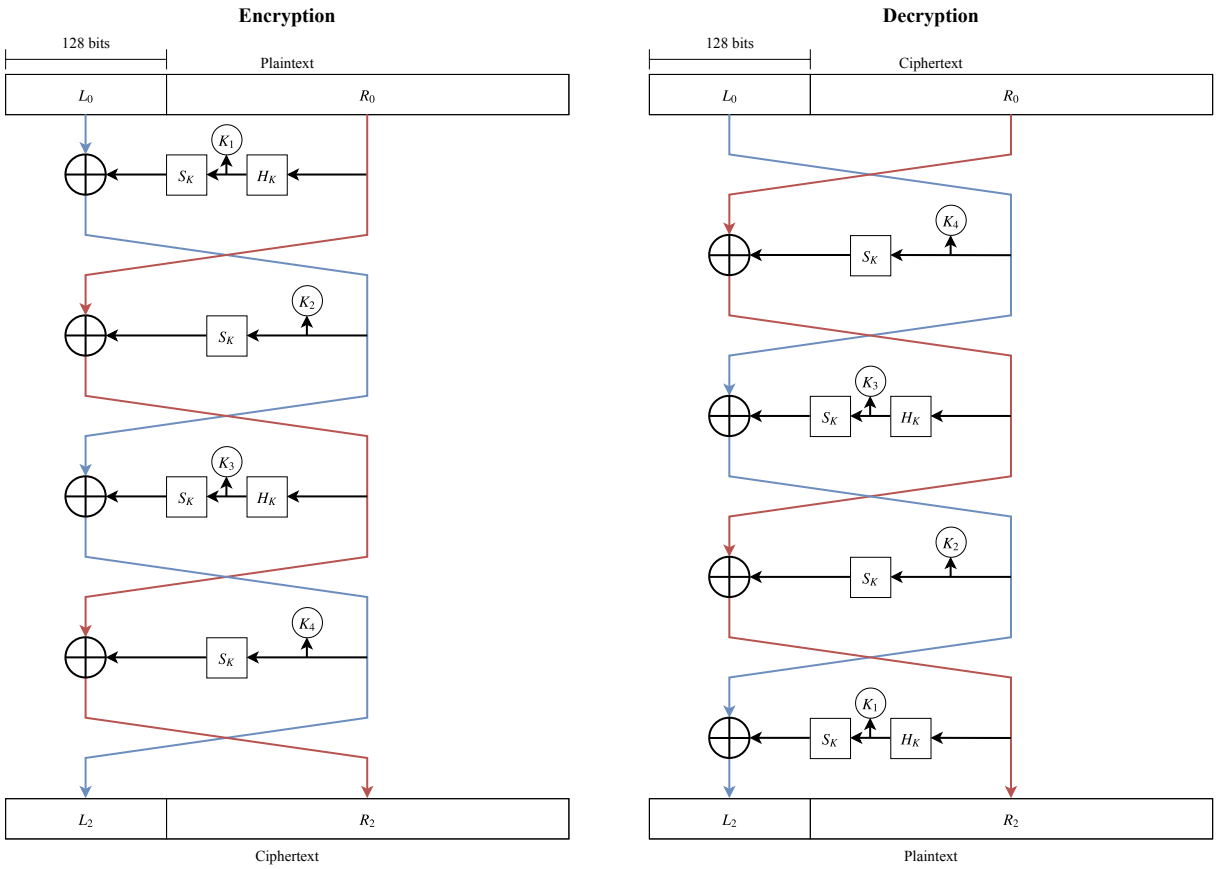

Figure 2.5: Feistel network of LIONESS encryption and decryption operations. S_K is a stream cipher keyed using K, and H_K is a hash function keyed using K.

Docker containers are used to package both Java and Python Loopix applications. As I need the ability to run the network both on my computer and a well-equipped server, Docker is used to enable reproducibility. This is because the containers contain all the dependencies necessary to run the applications and will behave identically on different machines regardless of the environment. Docker also simplifies orchestration of running a test network, as each container is isolated from each other. This is particularly important with the Python implementation, as each instance requires a separate working directory which is not necessary when encapsulated inside a container. It is simple to swap out a different implementations into the network to test by simply replacing an argument to `docker run`.

2.6 Existing Libraries

Many existing libraries are used to avoid reimplementing code that is already available, and some are dependencies as a result of the original implementation using a particular package.

- **Python Loopix and Sphinx libraries** - The existing `loopix` Python package is referenced for implementation details. It is also used for running parts of the test

network, such as providers, mix nodes and clients. This package also makes use of the `sphinxmix` package, which is also referenced when implementing my Sphinx library. `loopix` and `sphinxmix` are licensed under the LGPL-3.0 license.

- **Bouncy Castle** - Bouncy Castle is a Java library that provides a cryptography API, with support for various primitives such as AES, SHA, HMAC, and elliptic curves. Bouncy Castle was chosen for easier portability between clients which may not have the official Java cryptography extension installed. Reusing a well-maintained cryptography library is crucial, as implementing my own cryptographic functions properly is difficult, as vulnerabilities such as timing attacks are very easy to introduce. Bouncy Castle is licensed under the MIT license.
- **MessagePack** - MessagePack (`msgpack`) is an efficient binary serialisation format that is extensively used by the Python Loopix and Sphinx libraries. The Java library `msgpack-core` is used for my project. Unlike other serialisation libraries, it does not support serialisation of Plain Old Java Objects, which adds complexity to the usage of the library. `msgpack-core` is licensed under the Apache-2.0 license.
- **Apache MINA** - Apache MINA is an event-driven asynchronous networking library. This is used to simplify working with network sockets since the library handles threading and state management and exposes a simple send and receive interface. MINA is licensed under the Apache-2.0 license.
- **JUnit** - JUnit is a testing library for Java. It is used to support the test-driven development method by allowing the writing and execution of unit tests. JUnit is licensed under the Eclipse Public License.
- **SQLite** - `sqlite-jdbc` is a library for interacting with SQLite databases, which are used in Loopix to hold network information. It is licensed under the Apache-2.0 license.

2.7 Software Engineering Techniques

As binary compatibility is important, test-driven development is adopted to ensure my implementation is generating identical output to the Python implementation. This is done by generating test cases using the Python code and using unit tests to compare the output from the Java code. The test cases also include decoding output from the Python implementation. Changes are then made until the tests pass. This also ensures that any changes would cause regressions will be caught by the unit tests.

Chapter 3

Implementation

In this chapter, I will describe how I implemented my project, describing challenges faced along the way. The implementation consists of the following:

- A library for creating and processing Sphinx packets
- A library for sending and receiving messages on the Loopix network
- A command-line chat client that broadcasts messages to all clients on the network
- A testing framework for starting a test network

As Loopix uses the Sphinx packet format, I have implemented my own Java library for Sphinx. As such, the project is split into two libraries, with the low-level cryptographic elements and packet format implemented in Sphinx, and the high-level network decisions such as path selection and routing implemented in Loopix. The Loopix library provides a basic event-driven API for sending and receiving messages.

3.1 Sphinx Library

The Sphinx library provides an API for creating and processing Sphinx packets as generated by the `sphinxmix` Python package. Since there was little documentation for `sphinxmix`, significant efforts were required to reverse engineer functionality. This effort was necessary to meet the project goal of binary compatibility with the existing Python Loopix implementation. `sphinxmix`'s implementation deviates in several places when compared with the algorithm described in the Sphinx paper, and the paper does not specify every implementation detail.

3.1.1 Sphinx Parameters

The `SphinxParams` class encapsulates all of the cryptography and the various parameters for a Sphinx packet, such as the security parameter (key size) κ, header size, body size,

various hash functions, the LIONESS block cipher, and operations on an elliptic curve group. The security parameter in this implementation is defined to be $\kappa = 128$ bits. The header and body size are parameters that can be chosen to fit certain applications. By default, Loopix uses a header and body size of 1024 bytes each.

I used the BouncyCastle Java library for cryptographic primitives. These are used to implement the following functions:

- π: Used to encrypt the payload. Implemented using LIONESS, with AES-128-CTR as the stream cipher and SHA-256 as the hash function.
- π^{-1}: Used to decrypt the payload. This is the inverse of π, which is the LIONESS decryption function.
- μ: A Message Authentication Code (MAC). Implemented using HMAC-SHA256, truncated to be κ long.
- ρ: Used to encrypt the header. Implemented using AES-128-CTR.
- `deriveAesKeyFromSecret`: Used to derive the AES key for payload encryption from a secret elliptic curve key. Implemented using SHA-256, truncated to be κ long.
- Various hash functions:
  - h_π: Used to key π
  - h_μ: Used to key μ
  - h_ρ: Used to key ρ
  - h_τ: Used to identify previously seen group elements
  - h_b: Used to compute blinding factors

  These are implemented by encrypting a block of all zeroes using AES-128-CTR, with an initialisation vector unique to each function.[1] For example, the pseudocode for h_ρ is given below:

```
public byte[] hrho(byte[] key) throws CryptoException {
    return AES_ENCRYPT(key, ZERO_BLOCK, iv="hrhohrhohrhohrho");
}
```

Operations on an elliptic curve group such as exponentiation are encapsulated inside the `GroupECC` class. The curve chosen is the National Institute of Standards and Technology `P-224` curve. There are concerns that the NIST curves are backdoored by the NSA [**?**]. The coefficients of the curves are generated from an unexplained "random" seed value. As the seed values are unexplained, an attacker can freely generate curves until finding a curve vulnerable to a secret attack. Due to the need for binary compatibility with `sphinxmix`,

[1] The `sphinxmix` package deviates from the paper here as the paper describes using SHA-256 for these hash functions.

it is not possible to use a more secure and efficient curve such as `Curve25519`.[2] This design allows other implementations of cyclic groups to be used in place of the current implementation with ease. As such, modifying the library to use a new curve would be trivial.

3.1.2 Packet Creation

A Sphinx packet has two parts, the header and the body. The header contains information to process the packet correctly, and the body contains the encrypted payload. The packet has the following structure:

```
public class SphinxPacket {
    public SphinxHeader header;
    public byte[] body;
}
```

3.1.2.1 Sphinx Packet Header

The header has the following structure:

```
public class SphinxHeader {
    public ECPoint alpha;
    public byte[] beta;
    public byte[] gamma;
}
```

- `alpha`, α is an elliptic curve group element ($\alpha = g^b$) used to derive the per-hop shared secret required to authenticate and process the rest of `SphinxHeader`. It is also used to decrypt a layer of the Sphinx body payload.
- `beta`, β is a list of per-hop routing commands and padding that is encrypted in a nested manner. Each layer contains routing commands that is defined and processed by a Sphinx node.
- `gamma`, γ is a HMAC-SHA256 message authentication code tag that covers `alpha` and `beta`.

Header creation is handled in the `SphinxClient` class. Input is a sequence of ν mix nodes $\{n_0, .., n_{\nu-1}\}$ and their corresponding public keys $\{y_0, ..., y_{\nu-1}\}$, and outputs a `SphinxHeader` object and a list of per-hop shared secrets $\{s_0, ..., s_{\nu-1}\}$. The function signature for `createHeader` is given below.

[2] `sphinxmix` has recently been updated with a variant called Ultrix that uses `Curve25519` and has better performance. However, Loopix has not been updated.

```
private static SphinxHeaderData createHeader(SphinxParams params,
    List<byte[]> path,
    List<ECPoint> keys,
    byte[] destination) throws CryptoException
```

Header creation first starts with the derivation of key material for each hop. This key material will be used to encrypt and authenticate the rest of the header, and the packet payload.

Initially, a random initial blinding factor $b \in_R \mathbb{Z}_q^*$ is chosen. A sequence of ν tuples $(\alpha_0, s_0, b_0), ..., (\alpha_{\nu-1}, s_{\nu-1}, b_{\nu-1})$ is computed as follows:

- $\alpha_0 = g^b,\ s_0 = y_0^b,\ b_0 = h_b(s_0) * b$
- $\alpha_1 = g^{b_0},\ s_1 = y_1^{b_0},\ b_1 = h_b(s_1) * b_0$
- ...

Each tuple (α_n, s_n, b_n) is defined as:

- α_n is the group element used in the Elliptic-curve Diffie-Hellman key exchange (ECDH) to generate a shared secret. α_n can also be viewed as a public key.
- s_n is the shared secret for hop n. As the public key y_n is of the form g^{x_n}, then $s_n = (g^{x_n})^{b_{n-1}} = g^{x_n b_{n-1}}$. The shared secret is then passed into a key derivation function to produce an AES encryption key. The key derivation is done by passing the binary representation of the shared secret into the SHA-256 hash function and truncating the output to the security parameter κ.
- b_n, the blinding factor to be used for the next hop. This is computed by passing s_n into the hash function h_b.[3] The blinding factor allows both senders and mix nodes to compute α_{n+1} from the shared secrets. This is necessary to avoid transmitting every group element unaltered throughout the path, as this would lead to linkable messages.

At the end of this derivation process, a list of (α_n, s_n, b_n) tuples, each corresponding to a header for a hop in the path is generated. `header.alpha` is set as α_0.

The next step is to generate filler strings ϕ_i that serves as encrypted padding for β, the routing information block. This is done by repeatedly encrypting the padding block with each hop's shared secret. The filler strings are to ensure that the header block has a constant length as it is processed by nodes. This prevents mix nodes from learning their position in the message's path.

The routing commands β and MAC γ are generated next. Starting from the last hop, a sequence of headers $M_{\nu-1}, M_{\nu-2}, ..., M_0$, where $M_i = (\alpha_i, \beta_i, \gamma_i)$ is computed as follows:

[3] The `sphinxmix` package deviates from the paper here as the paper describes $b_n = h_b(\alpha_n, s_n)$, but the actual implementation ignores α_n and does $b_n = h_b(s_n)$.

- $\beta_{\nu-1} = \rho(h_\rho(s_{\nu-1}), \{\text{destination-identifier} \| \phi_{\nu-1}\})_{[0..\text{header size}]}$
- $\beta_i = \rho(h_\rho(s_i), \{|n_{i+1}| \| n_{i+1} \| \gamma_{i+1} \| \beta_{i+1}\})_{[0..\text{header size}]}$ for $0 \leq i < \nu - 1$
- $\gamma_i = \mu(h_\mu(s_i), \beta_i)$

β_i consists of either the destination identifier and the encrypted padding $\phi_{\nu-1}$, or the next hop in the path n_{i+1} prefixed with its length $|n_{i+1}|$, a MAC γ_{i+1} and a encrypted routing command block β_{i+1} to forward to the next node. β_i is then truncated to the header size.

`header.gamma` is set to γ_0, and `header.beta` is set to β_0. The final header structure when $\nu = 4$ hops is illustrated in Figure 3.1. At the end of this process, a `SphinxHeader` object with the necessary data M_0 and a list of per-hop shared secrets are created. The list of shared secrets will be later used for encrypting the payload.

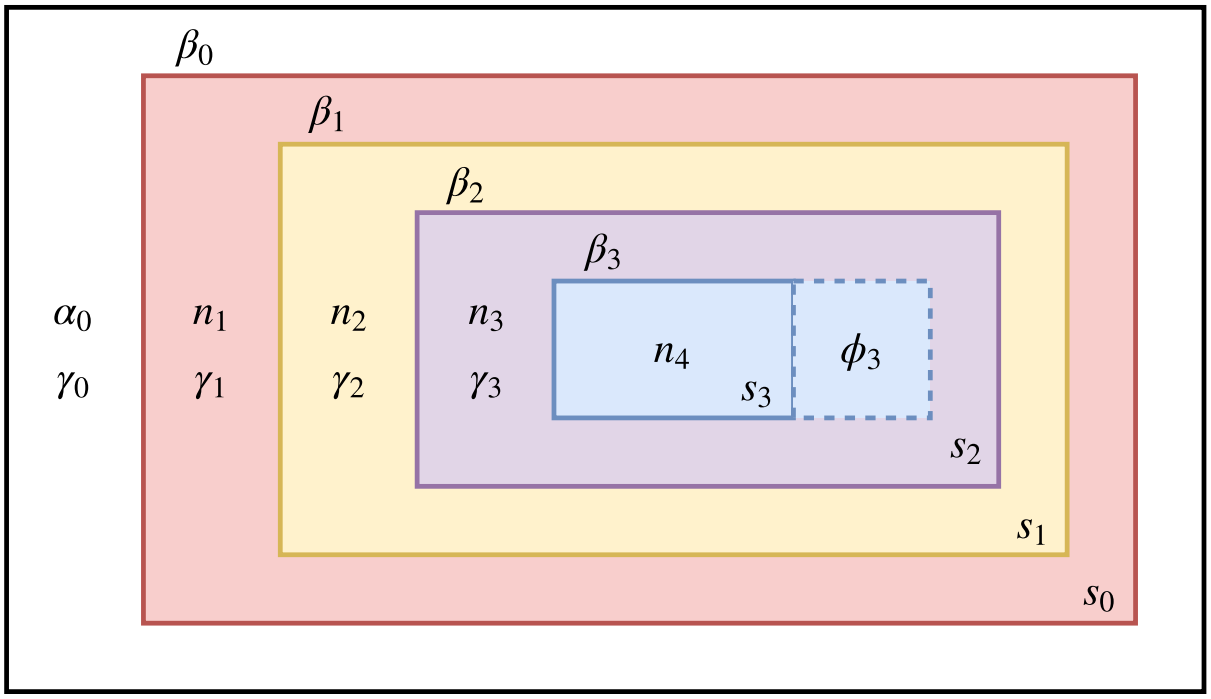

Figure 3.1: Sphinx header for a packet with 4 hops

3.1.2.2 Sphinx Forward Packet Body

Body creation is handled in the `SphinxClient` class. The function signature for creating an entire `SphinxPacket` is given below.

```
public static SphinxPacket createForwardMessage(SphinxParams params,
    List<byte[]> path,
    List<ECPoint> keys,
    Value destination,
    byte[] message) throws SphinxException
```

`createForwardMessage` takes as inputs a sequence of mix nodes and their corresponding public keys as per the header generation section above, and a message m of type `byte[]`. It performs the following operations:

1. `message` is concatenated with the destination identifier and serialised using `msgpack`, then padded to a fixed size.
2. A call is made to `createHeader` to generate a `SphinxHeader` and the list of shared secrets $\{s_0, ..., s_{\nu-1}\}$ to encrypt `message` with.
3. Starting from the terminal hop, `message` is repeated encrypted using π with a key derived from the hop's shared secret. That is,
   - $\delta_{\nu-1} = \pi(h_\pi(s_{\nu-1}), m)$
   - $\delta_i = \pi(h_\pi(s_i), \delta_{i+1})$

   and as implemented:

```
byte[] delta = paddedBody;
for (int i = path.size() - 1; i >= 0; i--) {
    delta = params.pi(
        params.hpi(header.keys.get(i)),
        delta
    );
}
```

The output of this process is the tuple (M_0, δ_0), which should be forwarded to the mix node n_0. The wire protocol for transmitting the tuple is determined by the consuming application.

3.1.3 Packet Processing

The packet processing API is exported as the following:

```
public class SphinxNode {
    public static SphinxProcessData processSphinxPacket(
        SphinxParams params,
        BigInteger secret,
        SphinxHeader header, byte[] delta)
        throws SphinxException, CryptoException
}
public class SphinxProcessData {
    public byte[] tag;
    public byte[] routing;
    public SphinxHeader header;
    public byte[] delta;
}
```

The `processSphinxPacket` function performs the majority of the per-hop packet processing. It handles authentication, decryption, and modifying the packet prior to forwarding it to the next node.

The mix node receives messages of the form $(M, \delta) = ((\alpha, \beta, \gamma), \delta)$. To decode the message, the mix node n, with private key x_n, performs the following steps:

1. Calculate the hop's shared secret s and replay tag τ.
   - $s = \alpha^{x_n}$
   - $\tau = h_\tau(s)$
2. Derive the AES key required for further processing of the header and payload.

   ```
   byte[] key = params.deriveAesKeyFromSecret(sharedSecret);
   ```

3. Validate the header by calculating the MAC and comparing the result with γ.

   ```
   if (!Arrays.equals(header.gamma,
       params.mu(params.hmu(key), header.beta))) {
       throw new SphinxException("MAC mismatch.");
   }
   ```

4. Pad β with 2κ zero bytes such that the total length is the header size. This is done to keep the length of β' which is forwarded to the next node the same. The padding is done before decryption to produce random-like padding after decryption.
5. Decrypt β using ρ.

```
byte[] B = params.rho(params.hrho(key), betaPadded);
```

6. Extract the routing command by using the length prefix from β. γ' is also extracted from β. The rest of the β is to be forwarded to the next hop as β'.
7. Compute the blinding factor and thus the group element for the next hop.

```
BigInteger b = params.hb(key);
ECPoint alpha = group.expon(header.alpha, b);
```

8. The payload δ is decrypted. If this is the last hop, then this results in the decrypted message.

At the end of this process, the tuple $(\tau, n, (\alpha', \beta', \gamma'), \delta')$ is produced. It is then up to the consuming application to check if it has seen τ before[4], parse the routing commands in n, and process the packet accordingly.

3.1.4 Challenges Faced

One of the goals was binary compatibility with the original Python implementation. As such, my implementation is based on the Python implementation. However, this introduced several challenges, mainly dealing with verbosity.

There are a lot of byte array manipulations, for example concatenation and slicing, which is easily done in Python as there are built-in operators. Another Python feature that is heavily used by `sphinxmix` is the ease of encoding strings as byte arrays. For example, the following code computes the first encryption round of the LIONESS cipher in Python:

```
k1 = self.hash(message[self.k:]+key+b'1')[:self.k]
c = self.aes_ctr(key, message[:self.k], iv = k1)
r1 = c + message[self.k:]
```

whereas in Java:

[4]In the original implementation of Loopix, there is a bug where the replay tag is not checked, losing some security in the process.

```
byte[] k1 = Arrays.copyOf(
    hash(
        Arrays.concatenate(
            Arrays.copyOfRange(message, this.k, message.length),
            key,
            "1".getBytes(Charset.forName("UTF-8")))),
        this.k);
byte[] c = aesEncrypt(key, Arrays.copyOf(message, this.k), k1);
byte[] r1 = Arrays.concatenate(c,
    Arrays.copyOfRange(message, this.k, message.length));
```

Sphinx also makes use of `msgpack` for serialising data. The Python API allows for simple serialisation of tuple data structures and dynamic objects which are widely used in `sphinxmix`. However, the Java API is harder to use for such data structures. For example, the following is a function that is used to encode mix node information in Python:

```
def encodeNode(idnum):
    return encode((Relay_flag, idnum))
```

and the same in Java:

```
public static byte[] encodeNode(int idnum) throws IOException {
    Packer packer = Packer.getPacker();
    packer.packValue(new ImmutableArrayValueImpl(new Value[] {
        new ImmutableBinaryValueImpl(new byte[]{RELAY_FLAG}),
        new ImmutableLongValueImpl(idnum)
    }));
    return packer.toByteArray();
}
```

The Python implementation also lacked documentation, which hampered efforts in understanding the implementation details. Python's dynamic types meant that it was not clear at first glance what kind of arguments were being passed around. It also used OpenSSL libraries under the hood, which has a very different API compared to BouncyCastle or the Java Cryptography Architecture. For example, the cryptographic library `sphinxmix` uses is `petlib`, which simply defined the curve used with `nid=713`. It was not immediately obvious which curve is used, and required finding the OpenSSL `nid.h` header file to map the unique ID into a curve.

3.2 Loopix Client Library

The Loopix client library provides APIs for sending and receiving messages. It is a thin wrapper around the Sphinx packet format. It provides the public key infrastructure, handles path selection, extends Sphinx to support delayed forwarding, and handles protocol details such as sending cover traffic. Extensive understanding of the `loopix` Python package was required since binary compatibility was a goal.

3.2.1 Exported Interface

The library exposes a `LoopixClient` class that handles message sending and receiving. A `LoopixClient` can be created from files generated by the Python implementation. This was done to simplify configuration management between the Java and Python nodes. Combined with the Docker-based architecture, it is possible to easily switch between the two. The configuration files contain the following parameters:

- `EXP_PARAMS_PAYLOAD`, `EXP_PARAMS_DROP`, `EXP_PARAMS_LOOPS`, `EXP_PARAMS_DELAY`: Scale parameters β for the exponential distributions for message delays and message send rates. β is the inverse of the widely used rate parameter $\lambda = 1/\beta$. Defined in terms of seconds.
- `PATH_LENGTH`: Number of mix nodes to select for a path. 3 is used.
- `DATABASE_NAME`: Path to the database containing information for nodes on the network.
- `TIME_PULL`: Time each message retrieval poll from the provider.
- `MAX_RETRIEVE`: Number of messages to return to client each poll.
- Client parameters such as the client's ID, provider ID, public key, and private key.

```
public class LoopixClient {
    public LoopixClient(String name, String host, short port,
        String providerName, ECPoint publicKey, BigInteger privateKey,
        Config config)
    public static LoopixClient fromFile(String configPath,
        String publicPath, String privatePath);
    public void run();
    public void setMessageListener(LoopixMessageListener listener);
    public void addMessageToQueue(String clientName, byte[] data);
    public void setMessageBuilder(LoopixMessageBuilder builder)
}
```

`LoopixClient` also exposes `setMessageListener` for applications to process incoming messages. As Loopix provides sender anonymity, only the message is available.

```
public interface LoopixMessageListener {
        void onMessageReceived(LoopixClient client, byte[] message);
}
```

`LoopixClient` exposes two methods for sending messages. `addMessageToQueue` adds a message to the send queue. `setMessageBuilder` allows applications to be asked for a message whenever a real message needs to be sent. This allows applications to perform optimisations such as batching multiple messages intended for a single destination into a single packet, reducing the number of packets needed. This is used by the latency measurement tool described later in the paper to accurately measure processing overheads without including the time a message sits in the queue.

```
public interface LoopixMessageBuilder {
        boolean isEmpty();
        ClientMessage getMessage();
}
```

3.2.2 Public Key Infrastructure

Sphinx, and by extension Loopix requires public key infrastructure to distribute knowledge of network nodes, such as their IDs, IP address, port number and public keys in order to route packets. This is achieved by distributing an SQLite database that is populated with information on every node that will be on the network. The SQLite database is also used in the Python implementation, allowing for reuse of existing code to generate the database. The database contains data of the following forms:

```
public class LoopixNode {
    public String host;
    public short port;
    public String name;
    public ECPoint publicKey;
}
public class MixNode extends LoopixNode {
        public int groupID;
}
public class User extends LoopixNode {
    public String providerName;
}
public class Provider extends LoopixNode { }
```

Each node has the following fields: a network location in the form of `<host>:<port>`, a `name` that uniquely identifies the node, and an elliptic curve `publicKey`. A mix node has an additional `groupID` which indicates the layer of the network topology it is located in. A user has an additional `providerName` that identifies the user's provider. The term `User` is used instead of `Client` to prevent confusion with the `LoopixClient` class.

Preferably, the hostname would be replaced with an IP address. This would avoid any possible impact to anonymity due to the use of DNS. However, for a test network, the ability to use hostnames simplifies configuration. The database can be populated with hostnames that are resolved at run time, instead of having to statically allocate the IP addresses of every node in advance.

The `ECPoint` data type is encoded using the scheme described in Section 3.2.3.2 and stored as a binary blob in the database.

Reading from the database is done using the `sqlite-jdbc` library and the corresponding Java Database Connectivity (JDBC) API built into Java. On start, the library will read and cache information on all mix nodes and the user's provider in memory to speed up lookups for path selection. SQL queries are used to lookup destination users and their provider. Optimally information on all users and providers are cached in memory as well, but this fails to scale as the number of users and providers are expected to be much larger than the number of mix nodes.

3.2.3 Loopix Protocol

This section describes the Loopix protocol. This includes packet formats, path selection and cover traffic details.

3.2.3.1 Packet Format

Loopix mainly uses the Sphinx packet format as described earlier, to provide end-to-end encryption and prevent intermediate nodes from learning additional information other than some routing details. However, there are control commands that do not use Sphinx, which may be considered bugs in the original implementation, as these packets reveal information that can break the Loopix security model. These have been maintained for backwards compatibility, but it could be easily fixed.

Loopix extends Sphinx by introducing its own routing commands. A routing command in Loopix contains the next hop's ID, hostname, port number, the number of seconds to delay forwarding the message, and a drop flag. The drop flag tells the processing node to drop the packet.

Loopix messages should be indistinguishable from each other. Loopix has the following types of Sphinx messages:

- **Real messages**

- **Loop cover messages** - Sent to ourselves with the magic number `HT` in the payload.
- **Drop cover messages** - Sent to a random user with the drop flag set in the routing command for the provider and a random payload.
- **Dummy messages** - Sent by providers to clients in response to a `PULL` command, with the magic number `HD` in the payload. In the original implementation, this is not a Sphinx packet but rather an unencrypted packet similar to the `PULL` and `SUBSCRIBE` commands. This modification was made to match behaviour described in the Loopix paper. Without the modifications, it is possible to distinguish dummy messages from real and loop cover messages when a client retrieves them from the provider, and analysis of bandwidth usage would be flawed. Although modifications were made to the Python implementation to support this, my implementation is still compatible with the original implementation.

Having magic numbers in the payload is not ideal, as having a type flag that is visible only to the destination node would be simpler. However, as the goal is compatibility with the Python implementation, this is not implemented.

Creation and processing of these messages are handled by the `ClientCore` class, which abstracts away the details above and exposes functions such as `createRealMessage`, `createDropMessage`, `createLoopMessage`, and `processPacket`. `ClientCore` is initialised with a `SphinxPacker` object that interacts with the Sphinx library to create packets. This design decision was made to reuse as much code as possible if I wanted to implement provider and mix node functionality. `SphinxPacker` exposes a generic `makePacket` function that is used to create any packet and handles encoding the various routing commands. As such, it also samples the delays for each hop from an $\text{Exp}(\mu)$ distribution, where μ is the inverse of the config parameter `EXP_PARAMS_DELAY` such that $\mu = 1/\beta_{\text{EXP\_PARAMS\_DELAY}}$. Sampling from an exponential distribution is implemented by sampling from a uniform distribution $U \sim U(0, 1)$ and using inverse transform sampling $X = -\ln(U)/\lambda = -\beta \ln(U)$:

```
public static double randomExponential(double scale) {
    return Math.log(random.nextDouble())*(-scale);
}
```

There are two control commands that clients can send to providers:

- `SUBSCRIBE` messages. These are tuples of the form `["SUBSCRIBE", name, hostName, port, publicKey]`. A provider will begin storing messages for this client in an inbox.
- `PULL` messages. These are tuples of the form `["PULL", name]`. A provider will send `MAX_RETRIEVE` messages back to the client, filling with dummy messages if there are less than `MAX_RETRIEVE` messages in the client's inbox.

3.2.3.2 Packet Serialisation

Sphinx requires Loopix to handle transmitting the Sphinx packet header and body to the next node. The approach taken by Loopix for serialising the packet is to encode the header and body using `msgpack`. That is, the tuple `(header, body) = ((alpha, beta, gamma), body)` is passed straight into `msgpack`. Loopix also uses `msgpack` for serialising the `SUBSCRIBE` and `PULL` control messages.

However, `msgpack` does not natively support packing the `ECPoint` type. This is done through the use of extension values in `msgpack`, which hold a type ID and a byte array payload. `ECPoint` is assigned type `0x02`, and the payload is encoded as per RFC5480 [**?**] using compressed ASN.1 encoding. The encoding is provided by BouncyCastle using `ECPoint.getEncoded()` and `ECCurve.decodePoint(byte[])` was used in the Java implementation.

3.2.3.3 Networking

Network communication in Loopix is done via UDP. The Apache MINA networking library was used to send and receive UDP packets. This was easier to use than Java's built-in `DatagramSocket`.

Upon starting the client, the client opens a UDP socket and listens for packets on a port listed in the public key infrastructure. The same socket is also used for sending packets. The only node that should be sending the client packets is the client's provider. Ideally, this means any packet from an unknown node is dropped. However, this is not implemented for two reasons: UDP spoofing can be used to bypass the check as there is no authentication between provider and client, and the use of hostnames instead of IP addresses complicates the check.

A scheduler using Java's built-in `ScheduledExecutorService` is started. The scheduler is used for scheduling four events:

- Sending real messages
- Sending loop cover messages
- Sending drop cover messages
- Sending provider pull requests

Sending provider pull requests is scheduled using the `scheduleAtFixedRate` function, running `retrieveMessages` every `TIME_PULL` seconds. `retrieveMessages` sends a `SUBSCRIBE` and `PULL` command to the provider.

Path selection. Messages are routed through 5 intermediate mix nodes, including providers. The path is: sender → ingress provider → layer 1 mix node → layer 2 mix node → layer 3 mix node → egress provider → recipient. Each mix node in the path is selected randomly. The recipient may be omitted for drop messages.

Sending messages. `sendRealMessage` is periodically called by the scheduler. It checks two sources of messages: an attached `LoopixMessageBuilder` and the send buffer. If there is a message, it will either ask for a message, or pop it from the buffer queue and sent. Otherwise, a drop message is generated and sent instead. This is shown in Figure 2.1 as the blue lines. Each time `sendRealMessage` is called, it samples a delay from an exponential distribution with parameter $1/\beta_{\text{EXP\_PARAMS\_PAYLOAD}}$, and the next call is delayed by the sampled value. This will cause a stream of messages to be emitted following the Poisson process $\text{Pois}(1/\beta_{\text{EXP\_PARAMS\_PAYLOAD}})$.

Cover traffic. The two cover traffic events occur similarly to the real message event. `sendLoopMessage` sends loop messages, which are messages that have their recipients specified as the sender. Loop messages are emitted following the Poisson process $\text{Pois}(1/\beta_{\text{EXP\_PARAMS\_LOOP}})$. `sendDropmessage` sends drop messages, which are messages with a random destination provider with a drop flag set such that the provider drops the packet instead of forwarding it. Drop messages are emitted following the Poisson process $\text{Pois}(1/\beta_{\text{EXP\_PARAMS\_DROP}})$. These are represented in Figure 2.1 by the red and green lines.

Processing messages. In response to a `PULL` request, a provider forwards messages held for a client, padding with dummy messages if necessary. The client first decodes and checks that the received packet has the correct `msgpack` format that is expected. The Sphinx library is used to process the decoded packet, which returns a `(header, body)` tuple. The client checks that the message is the final destination packet, and verifies that the destination is the client. Further processing is done to discard loop and dummy messages based on magic numbers in the payload. If it is a real message, then `LoopixClient` calls the attached `LoopixMessageListener` with the real payload.

3.2.3.4 Challenges Faced

Concurrency in the library introduced race conditions in the Sphinx library, as thread safety was never considered in the design phase and thus the library was not designed for concurrent operations. The client shared a single instance of the `SphinxParams` object, which used a single instance of the AES `Cipher` object. There are two operations that may run concurrently: receiving and sending messages. This may result in corrupted packets, as the encryption and decryption operations may overlap. This only manifested in the evaluation stage of the project, as before then the client was tested with low send and receive rates such that the race condition was never triggered. This was initially fixed by using the `synchronized` keyword to ensure concurrent operations never occur. A better fix was implemented by modifying the Sphinx library to be thread-safe by storing the `Cipher` object inside a `ThreadLocal` variable such that each thread has its own instance.

3.3 Chat Client

A basic command-line chat client was implemented using the Loopix library. This serves to fulfil the success criteria of a demonstration application, and can also be used to test the user experience of different network parameters, such as message delay, send rates and retrieval rates. It is also used for verifying that the network is working as expected.

It reads console input, prefixes a timestamp and name, and adds a magic header `CHAT` to the start of the payload to identify a chat message. The resulting message has the form `CHAT10:10:50 <client_name> <message>`. It is then broadcast to all other clients using the `addMessageToQueue` function. Upon receiving a payload, it checks for the `CHAT` prefix, strips the prefix and outputs the message to the console.

ANSI escape sequences are used to avoid clobbering and splitting console input. It is also used to erase console input from the terminal. This is done to provide a nicer output by separating the terminal into two regions for input and output.

3.4 Testing Framework

Both my implementation and the original Python implementation are bundled into Docker containers. This simplifies deployment of the network as all dependencies are already installed, at the cost of some initial time spent to configure the build process.

Running a test network is simply executing `docker run` for however many nodes are in the network. To run multiple instances of the original implementation without Docker containers would involve extra effort, as the original implementation requires a separate working directory for each instance. Containers are isolated from each other, and thus I can ignore such implementation quirks. Docker also provides an isolated network that helped with capturing network traffic from the system for evaluation.

The testing framework consists of a collection of shell scripts and Python scripts for:

- Building various applications and their corresponding Docker containers
- Setting up network configuration such as the public key infrastructure and network parameters
- Running a test network using Docker containers
- Automatically collecting data for evaluation, by varying network configurations and restarting the network
- Parsing collected data into graphs

Continuous integration was used to automatically build and run tests upon code commit. The Travis CI service was used to perform automatic builds. This ensured that any changes made did not introduce regressions and could still successfully build. This also

helped in practising test-driven development by ensuring changes passed all unit tests that were written and identify bugs introduced early.

3.5 Summary

In this chapter, I have discussed the implementation of the components of my system such as the Sphinx packet format and Loopix client libraries. I have described in detail both Sphinx and Loopix protocols. I have also discussed key design choices, and challenges faced with maintaining binary compatibility with poorly documented libraries.

Chapter 4

Evaluation

In this chapter, I evaluate my project in the context of the project's success criteria. Namely, I will show that my library is binary compatible with the existing Python library, and various metrics such as bandwidth and latency overheads are graphed. A theoretical and empirical analysis of the viability of Loopix on mobile devices is also presented.

The following evaluation demonstrates that I have achieved my success criteria that are listed below:

- I have completed a Java implementation of the Sphinx library.
- I have completed at the very minimum a client implementation of Loopix in Java.
- I have created a demo app that uses the library to send messages.
- I have produced an evaluation of my Loopix implementation, in terms of bandwidth throughput, bandwidth overhead, latency overhead, complexity overhead.

4.1 Functional Testing

The key functionality of the Loopix client library is the ability to connect to a Loopix network consisting of nodes running the original Python code. This is tested in two ways: unit tests and integration tests.

Unit tests. The Java Sphinx library uses the JUnit unit testing library. The output of the various keyed hash functions, key derivation functions, and LIONESS encryption and decryption is compared between the Java and Python implementations. The output from the creation of Sphinx packets is also compared. Random functions had to be stubbed out, as random padding and keys are generated and would prevent a byte-for-byte comparison between implementations. The Java test suite also attempts to decode Sphinx packets generated by the Python code. The same methodology is applied to the Loopix library. The results show that my library is able to generate the same bit strings as the Python code, and it is able to decode packets generated by the Python code.

Integration tests. A test network is initially setup consisting entirely of Python clients, providers and mix nodes. To test that the Java client is functional and compatible with the Python nodes, some of the Python clients are replaced with Java clients. Network behaviour is then observed to ensure correctness. An example of an issue found early in testing is the use of incorrect time units when generating delays. The Java code was producing packets with delays in milliseconds, which were then interpreted as seconds by the Python nodes. This resulted in Python nodes delaying packets $1000x$ longer than expected.

4.2 Performance

4.2.1 Experimental Setup

All experiments were run on the Amazon EC2 platform, on a single `m5.12xlarge` instance running 16.04 Ubuntu with 48 virtual cores on a 2.5 GHz Intel Xeon Scalable processor with 192GB of RAM. Experiments were run with two different network setups:

- 4 providers and 3 layers of mix nodes with 2 mix nodes in each layer for a total of 6 mix nodes.
- 8 providers and 3 layers of mix nodes with 6 mix nodes in each layer for a total of 18 mix nodes.

The Loopix paper uses the first setup with 6 mix nodes. However, bottlenecks were discovered and thus some experiments were repeated with more mix nodes in an attempt to scale the network.

The number of clients and network parameters are varied accordingly for each experiment. Experiments were run for 3.5 minutes with data collected after 30 seconds had passed to allow the network to initialise. Limited storage prevented longer running experiments.

4.2.2 Bandwidth

The bandwidth overhead of Loopix is evaluated by measuring the rate of outgoing messages, cover and real, against an increasing overall rate at which nodes send messages. Incoming messages are not measured, as it is fixed and predictable for clients. To measure bandwidth, packet traces were captured using `tcpdump`. Static private keys allowed the decryption of captured packets to determine the type of message. Values were averaged over 10 repetitions and the error bars are the standard deviation.

Each packet has a fixed size of 2079 bytes, with a Sphinx header and body size of 1024 bytes each. The large packet size exceeds the "safe" size of UDP packets of 508 bytes before the packet may be fragmented. This poses an issue where loss of a packet fragment results in the loss of the entire packet. The size parameter can be tuned for specific

applications, and as such results presented in this section are in terms of messages rather than bytes.

The delay parameter `EXP_PARAMS_DELAY` is set to 0.001 ($\mu = 1000$) such that the average delay per hop is 1 ms. There are 100 Java clients and 100 Python clients sending messages at rate λ each. λ is the sum of real, loop, and drop rates where $\text{Pois}(\lambda) = \text{Pois}(\lambda_R) + \text{Pois}(\lambda_L) + \text{Pois}(\lambda_D)$. The experiment was performed with parameters $\lambda_R = \lambda_L = \lambda_D = 0$ messages per second for a single client, increasing each iteration such that $\lambda_R = \lambda_L = \lambda_D = \{0, 0.02, 0.05, 0.08, 0.1, 0.2, ..., 2.9, 3.0\}, \lambda = \{0, 0.06, 0.15, 0.24, 0.3, 0.6, ..., 8.7, 9.0\}$. Mix nodes also send loop messages at rate λ_L.

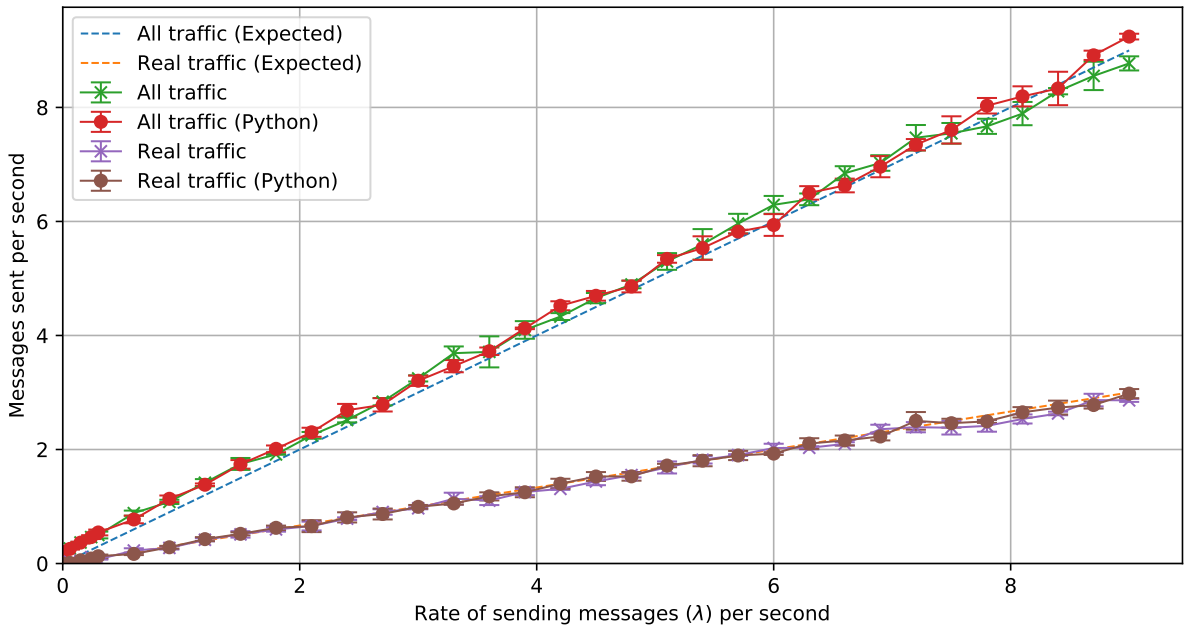

Figure 4.1: Overall sent traffic and real traffic per second for a single Java and Python client.

Figure 4.1 shows the sending rate of all traffic and rate of real traffic of both the Java and Python client versus the overall sending rate λ of a client. Observe that the Java client performs similarly to the Python client, and falls within the expected average rates predicted by a Poisson process. Therefore my implementation behaves similarly to the Python implementation and theory and there are no obvious issues.

It is important to note that the λ_R parameter determines the amount of real traffic in the network. Thus the parameter can be adjusted by a client depending on the amount of traffic in the network, with a potential loss of anonymity if there is insufficient cover traffic in the network.

Figure 4.2 shows the sending rate of all traffic and rate of real traffic versus the overall sending rate λ of a client at a single Python mix node. Messages sent by the mix node includes forwarded messages and cover traffic generated by the mix node. Observe that the throughput of the mix node increases linearly until around the 325 messages per second mark. Beyond this point, throughput stabilises before showing a small downward

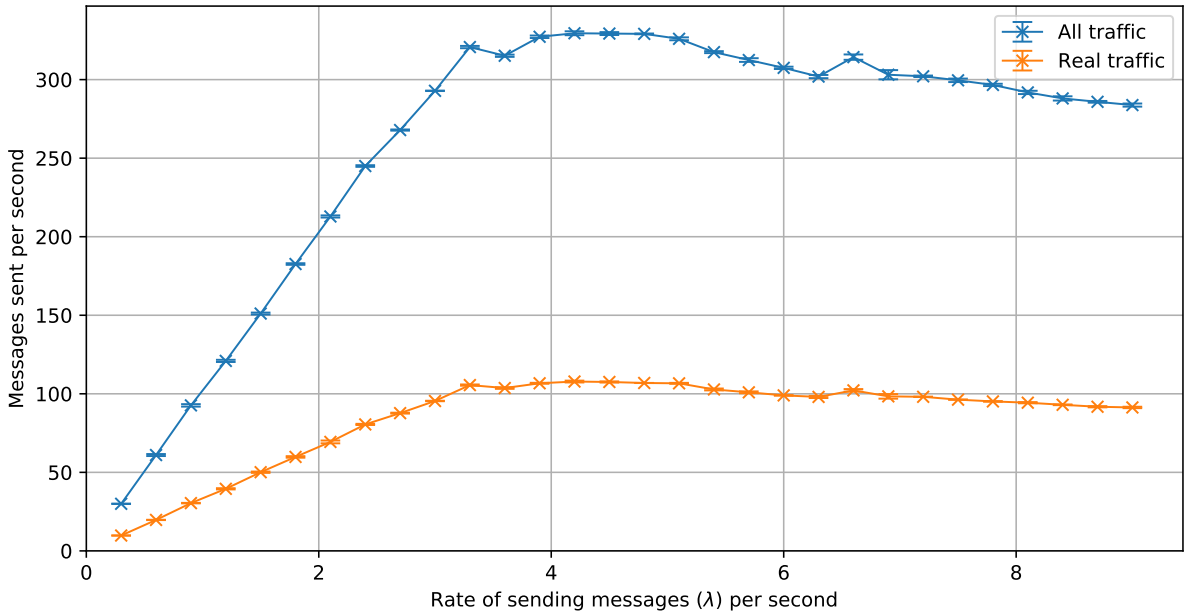

Figure 4.2: Overall sent traffic and real traffic per second for a single mix node. Network with 4 providers, 6 mix nodes.

trend as the configured sending rate of a single client increases. This deviates from results demonstrated in the Loopix paper [**?**], where throughput was linear until around the 225 messages per second mark, and a smaller growth was observed beyond that point instead of a downward trend. This may be attributed to a difference in experimental setup, as my entire network is run on a single machine, whereas they used multiple machines.

In an attempt to test the scalability of the system, the same experiment was repeated with 8 providers and 18 mix nodes. The results of which are shown in Figure 4.3. A very similar shape to the initial results is observed, with about half the throughput when the system shows a bottleneck. This implies that at higher send rates, there exists a bottleneck in my experimental setup.

4.2.3 Latency

Latency measurements were made by clients sending messages to themselves. Each message contains the timestamp when the message was created. Messages are created when the client needs to send a real message to remove the delay between the transmission of real messages. The client then records the timestamp when it passes a decrypted payload to the application, after receiving and processing the message. Sending looped real messages was done to ensure that time synchronisation issues would not impact measurements. As the experiments were conducted on a single host, network latency is negligible.

A modification was made to the Python provider to add a new configuration parameter `PUSH_MESSAGES`. When this option is set, providers will immediately forward messages

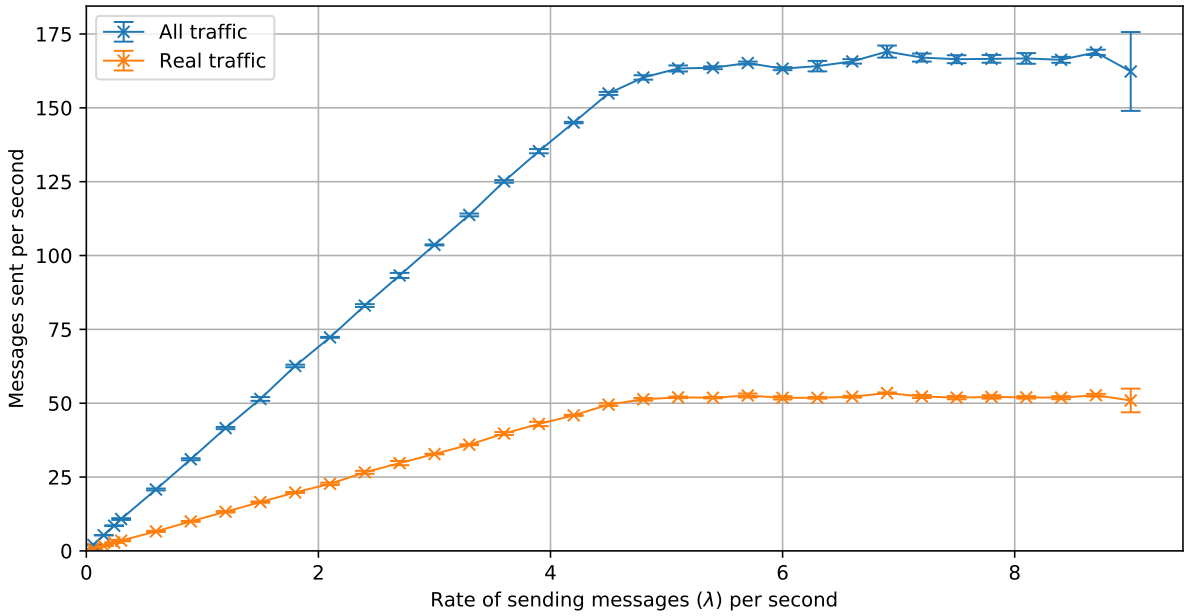

Figure 4.3: Overall sent traffic and real traffic per second for a single mix node. Network with 8 providers, 18 mix nodes.

to clients, instead of waiting for clients to send a `PULL` request. This was necessary to measure end-to-end latency solely as a result of system overhead.

The latency experiments were configured with $\lambda_R = \lambda_D = \lambda_L = 1.0$ messages per second, and the per-hop delay is set to 0.0 seconds. Values are measured by both Java and Python clients and averaged over 500 samples, and the error bars are the standard deviation.

The first experiment was conducted with `PUSH_MESSAGES` enabled. The number of clients is varied from 20 to 200 clients, increasing by 30 each iteration. The type of clients are evenly split between Java and Python clients. The results are shown in Figure 4.4.

With 20 clients, the latency measured by the Java and Python client was (18.09 ± 2.21) ms and (20.87 ± 1.36) ms respectively. This increases as the number of clients increases. At 200 clients, the latency increases to (27.44 ± 6.96) ms and (30.49 ± 7.44) ms. My Java implementation performs slightly better than the Python implementation.

My results differ significantly from the original paper [**?**], where the end-to-end latency was measured at 1.5 ms. This discrepancy may be caused by my experiment's significantly higher rates of sending messages or the use of different measurement techniques. The original paper uses $\lambda = 30$ messages per minute $= 0.5$ messages per second.

The latency behaviour changes dramatically when the system is overloaded. With 20 clients, the latency measured for the Java client was (19.35 ± 3.12) ms. This increases to (27.33 ± 7.50) ms with 50 clients. As the number of clients exceeds 50, a bottleneck appears and the latency overhead exceeds 1.5 s. This bottleneck also manifests in the mix node bandwidth results in Figure 4.2.

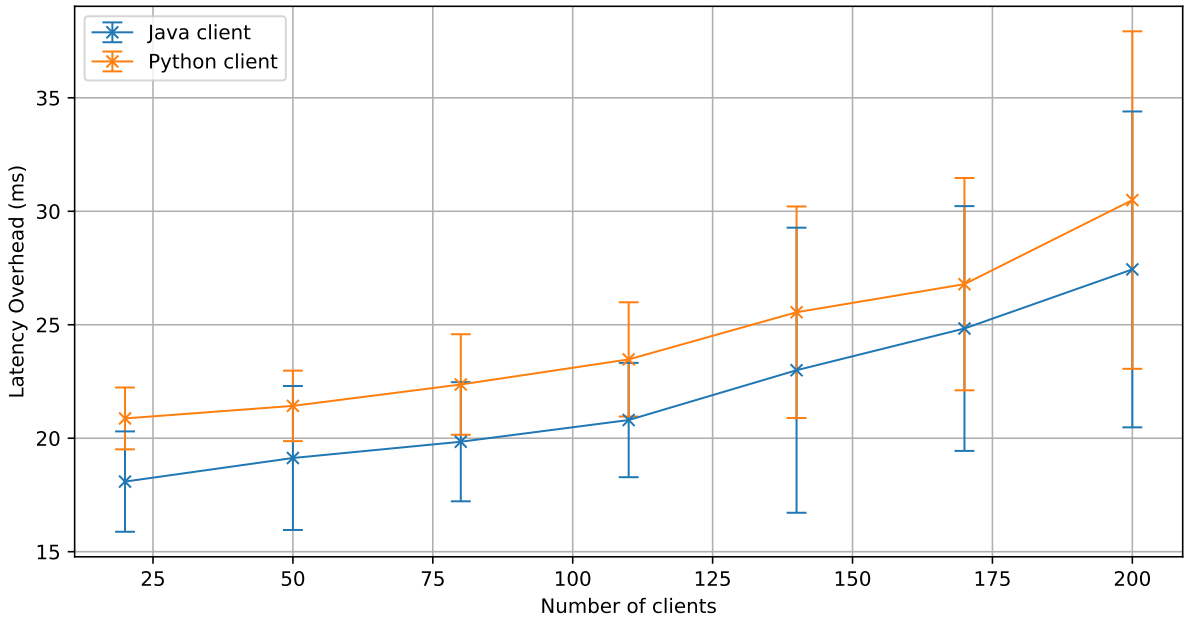

Figure 4.4: Latency overhead measured as the time taken to send a message through the Loopix network, with increasing number of clients with $\lambda = 3.0$ messages per second

The next experiment disables `PULL_MESSAGES`, and sets `TIME_PULL` and `MAX_RETRIEVE` such that the client will pull 30 messages every 5 seconds. The end-to-end latency measurement now includes time spent in the provider queue waiting for client requests.

Figure 4.6 plots the time a message is sent, the time the message is received, and the time taken to send and receive the message. Observe the periodic nature of the time taken plot, which is the result of periodically requesting messages from the provider. The time taken for messages should not and does not exceed 5 seconds in addition to a slight latency overhead.

When the system is overloaded, the end-to-end latency becomes less regular. Latencies of up to 25 seconds were observed, which is significantly higher than expected.

4.3 Viability on Mobile Devices

The key motivation for this project is to produce a library usable by mobile devices, specifically Android devices. Mobile devices are heavily resource constrained, especially with respect to cellular data usage and battery life. This section will explore the viability of Loopix on mobile devices in terms of energy use, data usage, and associated costs. The tuning of network parameters is also discussed.

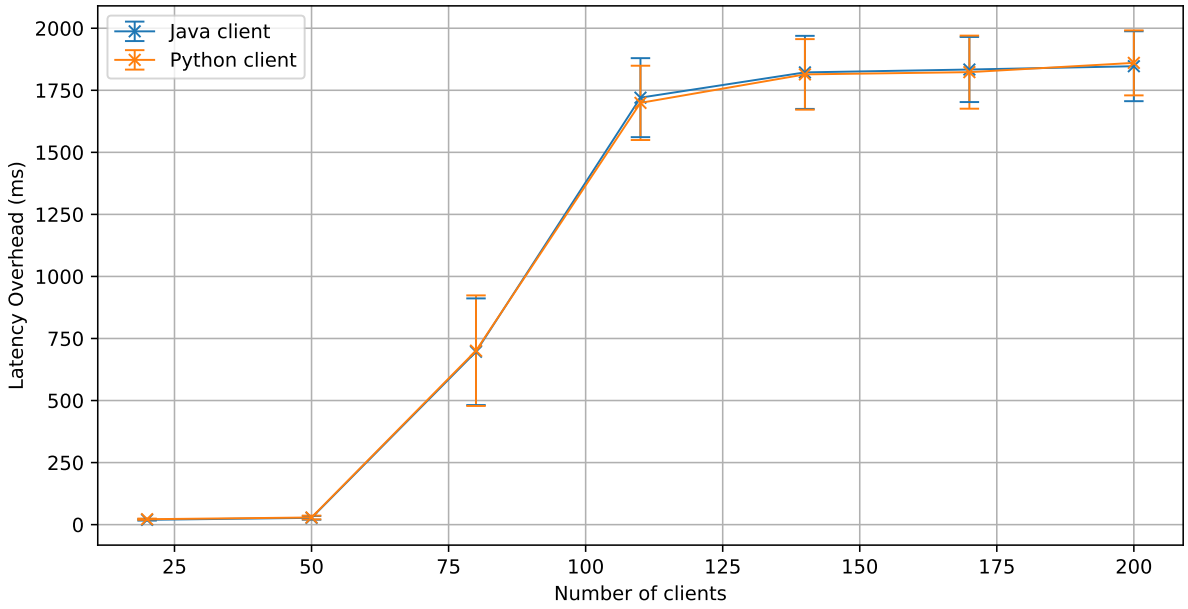

Figure 4.5: Latency overhead measured as the time taken to send a message through the Loopix network, with increasing number of clients with $\lambda = 8.0$ messages per second, showing signs of bottlenecking.

4.3.1 Bandwidth Usage

Analysis of the bandwidth usage of a single Loopix client is straightforward. The network message send rates $\lambda_R, \lambda_L, \lambda_D$ yield an expected average send rate $\lambda_{\text{send}} = \lambda_R + \lambda_L + \lambda_D$, and `TIME_PULL` and `MAX_RETRIEVE` characterise the rate of incoming traffic $\lambda_{\text{pull}} =$ `MAX_RETRIEVE/TIME_PULL`. As such, the total network traffic generated per second is $\lambda_{\text{total}} = \lambda_{\text{send}} + \lambda_{\text{pull}}$.

Suitable parameters need to be chosen first. A likely application of Loopix is for instant messaging. Assuming that for a usable experience, the total artificial delay $(4 * 1/\mu)$ introduced should be below 1 second. Adding network latency of about 200 ms per hop with 4 hops and a message retrieval period of 1 second yields an end-to-end latency of about 3 seconds which is usable for instant messaging. Loopix's authors suggest that $\lambda/\mu \geq 2$ to maintain strong anonymity. Adding the delay constraint of $4 * 1/\mu \leq 1$, the following constraints are obtained: $\lambda \geq 2\mu, \mu \geq 4$. Using $\mu = 4, 1/\mu = 0.25$ seconds, λ needs to be greater than 8 messages per second to maintain strong anonymity. λ_{pull} needs to be greater than the rate of loop messages to ensure that the provider inbox can be emptied, else the inbox would continuously grow. λ_{pull} is chosen to be λ_{send} to allow at least a real message stream from a single client to be successfully retrieved.

Using the parameters described, $\lambda_{\text{total}} = \lambda_{\text{send}} + \lambda_{\text{pull}} = 8 + 8 = 16$ messages/second. At 2079 bytes per message, this yields 16 messages/second $*$ 2079 B = 33264B/second = 2.68 GiB/day = 80.3 GiB/month. At £0.01 per MB, this would result in a monthly cost of £862.20, which is impractical.

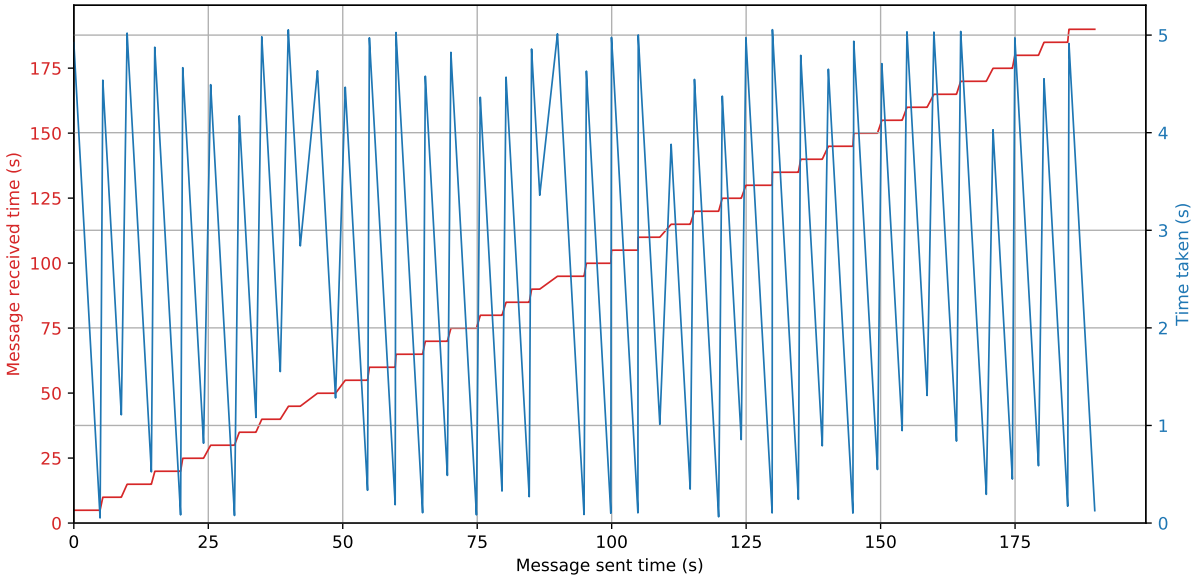

Figure 4.6: Plot of message sent and received times, and the time taken over a 3 minute period.

It is possible to reduce the size of a message to $\sim$ 1000 bytes by reducing the size of the header by using IP addresses instead of hostnames, and reducing the size of packet body as well. However, reducing bandwidth use by half would still cost £431.10, which still is impractical. Without sacrificing anonymity, the delay added needs to be increased to bring λ into reasonable values, however this would impact message responsiveness. A thorough exploration of bandwidth usage with different parameters is shown in Table 4.1.

4.3.2 Energy Usage

EnergyBox [**?**] is used in conjunction with the packet traces captured for bandwidth analysis to analyse energy use. EnergyBox was configured with energy models for a Nexus One connected to the TeliaSonera network. As the packet captures only contained 3 minutes of data, the output from EnergyBox was extrapolated to 24 hours. An assumption is made that because of the high message send rates, the radios are never able to power down. This approach was verified by using a 3 hour packet trace with a send rate of 1.5 messages/second. This yielded 6695 J and extrapolating to 24 hours yields 53 562 J.

In Figure 4.8, observe that the daily energy consumption stabilises around the 54 000 J mark, or 15 Wh. A Nexus One has a battery capacity of 5.18 Wh. This means there is insufficient battery capacity to run Loopix for a whole day. Note that the energy consumption values accounts only for energy used for network communication, and does not include the energy required to keep the device awake and generating Loopix messages. Even when the send rate is zero, 43 356.82 J was measured. This lower bound is determined by the message retrieval rate.

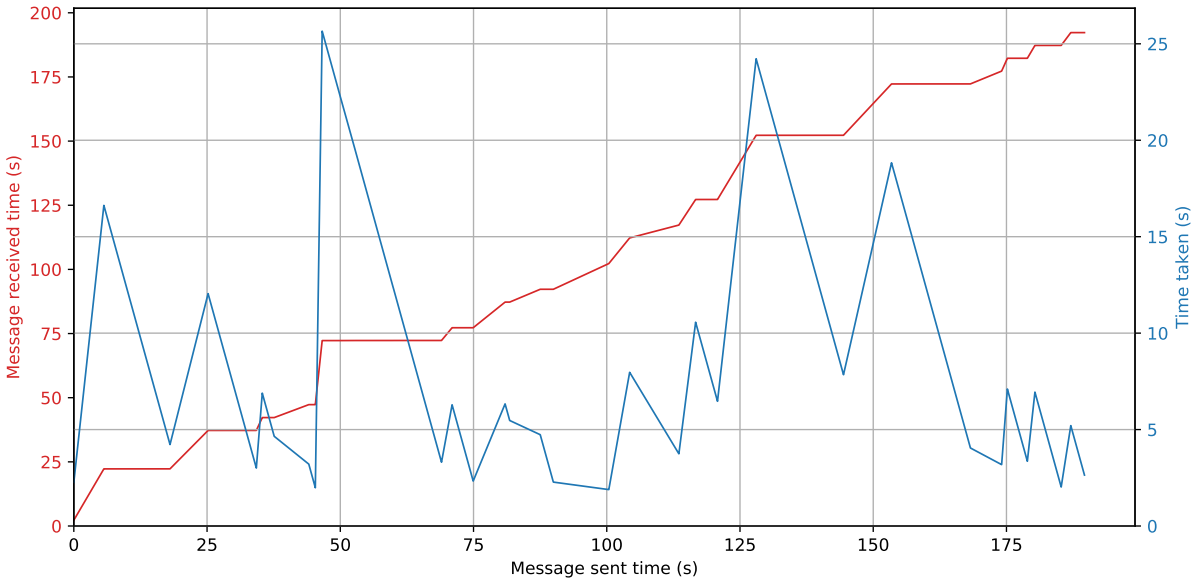

Figure 4.7: Plot of message sent and received times, and the time taken over a 3 minute period when the system is overloaded.

4.3.3 Parameter Tuning

In terms of data usage, working backwards from a target monthly data usage of 1 GiB, 1 073 741 824 B/2079 B = 516470 messages/month = 0.2 messages/second, which implies $\lambda_{\text{send}} = \lambda_{\text{pull}} = 0.1$ message/second. Maintaining strong anonymity would require a per-hop delay of $1/\mu \geq 2/\lambda_{\text{send}} = 2/0.1 = 20$ seconds, which gives a total average delay of 80 seconds. With such a high latency, applications such as instant messaging are no longer viable unless users do not expect near instantaneous communication. Looking at Table 4.1, it is clear that targeting low latency requires too high of a cost to maintain.

As providers store messages for clients, it is possible to use a strategy of sending traffic only when the application is in the foreground, and perform periodic background polling for messages from the provider. This would reduce the bandwidth and energy used as the Loopix client would only be active for short bursts, and the send rates could be set high and thus delays low. However, the potential impact to anonymity is a concern, as this introduces traffic patterns which may leak information.

4.4 Summary

In this chapter, I have shown that I have met all of the project's success criteria. I have tested the binary compatibility and correctness of my libraries, and analysed the performance of my libraries against that of the Python implementation. My implementation slightly outperforms the existing Python implementation in terms of processing overhead. I have also gone beyond the success criteria in producing a theoretical and empirical ana-

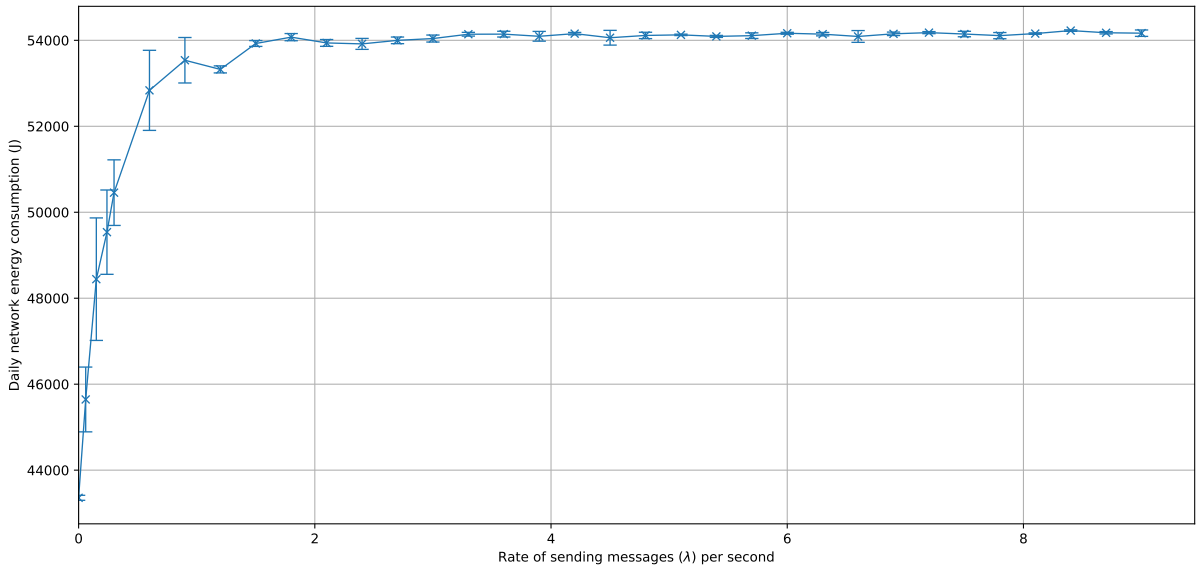

Figure 4.8: Daily energy consumption of a Nexus One device connected to the TeliaSonera network with varying message send rates.

lysis of the viability of the Loopix system on mobile devices in terms of bandwidth and energy consumption which are heavily constrained on mobile devices.

λ_{total} (msg/second)	Delay (s)	Monthly data usage (MiB)	Monthly data usage cost (£)
0.05	320	256.95	2.69
0.1	160	513.91	5.38
0.2	80	1027.82	10.77
0.5	32	2569.56	26.94
1	16	5139.12	53.88
2	8	10278.25	107.77
3	5.33	15417.38	161.66
4	4	20556.51	215.55
8	2	41113.03	431.10
16	1	82226.07	862.20

Table 4.1: Delays required to maintain strong anonymity and data usage for different λ parameters

Chapter 5

Conclusion

I have fulfilled all of my success criteria and produced an implementation and evaluation of Loopix in Java. I have produced a Sphinx library, a Loopix client library, a chat demo application, and a testing framework for running and measuring metrics of a test network. The libraries produced are binary compatible with the Python implementations.

The bandwidth overhead results reproduced similar scaling to the results in the Loopix paper, and I pushed the network even further, demonstrating the existence of bottlenecks and the behaviour of the system under load. The latency results showed that my implementation was slightly faster than the original implementation, but with differing results compared to the Loopix paper [**?**].

The mobile device viability analysis produced interesting results. Some bandwidth and energy overhead was expected, however when tuned for low latency Loopix is not viable for mobile devices, and perhaps even some desktop applications. This shows that even though Loopix was designed for a tunable trade-off between latency and bandwidth, the trade-offs may be too large for certain applications. Thus on mobile devices, Loopix is better suited for delay-tolerant applications such as email, notes/to-do lists, and peer-to-peer hosted blogs.

5.1 Further Work

Further work on Loopix has already started with the Katzenpost project[1]. Katzenpost is a mix network built on top of Loopix, which improves upon Loopix by introducing reliable transport through the use of acknowledgements, a consensus-based public key infrastructure, and the use of modern ciphers and hashes.

As Katzenpost suffers from the same issues of poor viability on mobile devices, further work could be done by analysing the anonymity impact of various approaches to conserving

[1] `https://katzenpost.mixnetworks.org`

resources on mobile devices, such as the strategy of only sending cover traffic when the application is actively running in the foreground of the device.

Further work could be done with dynamically adjusting the various network parameters such as cover traffic send rates based on the state of the network, as less cover traffic is needed when there are more users in the network. This would allow for the automatic optimisation of bandwidth overhead needed to maintain anonymity.