Jun Siang Cheah

An implementation and evaluation of Loopix, an anonymous communication system

Computer Science Tripos – Part II

Christ's College

April 8, 2018

Proforma

Name:	**Jun Siang Cheah**
College:	**Christ's College**
Project Title:	**An implementation and evaluation of Loopix, an anonymous communication system**
Examination:	**Computer Science Tripos – Part II, July 2018**
Word Count:	**???**
Project Originator:	Dr Alastair Beresford
Supervisor:	Dr Alastair Beresford

Original Aims of the Project

Work Completed

Special Difficulties

[1]This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, Jun Siang Cheah of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

Chapter 1

Introduction

1.1 Motivation

Edward Snowden, Tor vuln to traffic analysis

1.2 Overview

1.3 Related Work

1.3.1 Mix Networks and Onion Routing

Chapter 2

Preparation (Due 20th April)

This chapter begins by providing a high level overview of how Loopix is structured, ...

2.1 Structure of Loopix (Due 23rd March)

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.

2.1.1 Network Architecture

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.

In order 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 the sender is responsible for selecting the route through the network.

2.1.2 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. This delay parameter is determined by the source of the message. 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.

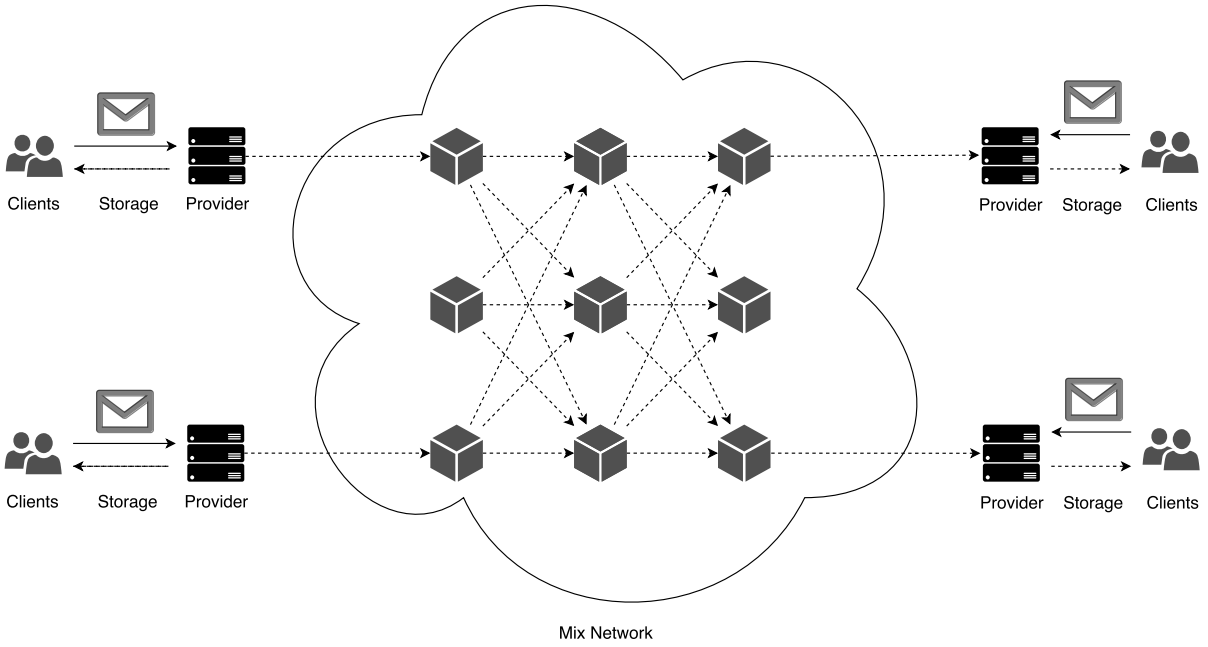

Figure 2.1: Overview of the Loopix network architecture. Clients pass messages to their providers, which are responsible for injecting the message into the mix network. The received messages are stored in inboxes at providers and retrieved by clients when they come online. TODO: Refine the figure with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

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.

2.2 Cryptography

2.2.1 Elliptic Curves

2.2.2 LIONESS Wide Cipher

2.3 Requirements Analysis

Sphinx, Loopix, Testnet, Evaluation

2.4 Development Tools

IDEA IntelliJ

Docker

2.5 Existing Libraries

2.5.1 Python Loopix Library

2.5.2 Bouncy Castle

Crypto (AES/SHA/HMAC/ECC)

Chosen for easier portability between clients which may not have the official java crypto extension installed, or have the neccessary ciphers.

2.6 Software Engineering Techniques

Test driven development. Existing library allows generation of test data to verify correctness and compatibility.

Chapter 3

Implementation (Due 6th April)

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 of 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, and is based heavily on the package. This distinction is 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.

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), header size, body size, various hash functions, and the LIONESS wide block cipher.

As mentioned in section 2.5.2, I used the BouncyCastle Java library for cryptographic primitives.

These are:

- AES-128-CTR
- SHA-256
- HMAC-SHA256

3.1.2 Packet Creation

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

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

Sphinx Packet Header

The header has the following structure:

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

- `alpha` 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` is a list of per-hop routing data and padding that is encrypted in a nested manner. Each layer contains routing commands that is defined and processed by a Sphinx node.
- `gamma` is a HMAC-SHA256 message authentication code tag that covers `alpha` and `beta`

Header creation is handled in the `SphinxClient` class. Header creation takes as input a sequence of 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 is given below.

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

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

- α_n is the group element used in the Ephemeral Elliptic-curve Diffie-Hellman key exchange (ECDHE) to generate a shared secret. α_n can be thought of as an ephemeral public key.

- s_n is the shared secret for hop n. As the public key y_n is of the form g^{x_n}, $s_n = (g^{x_n})^{b_{n-1}} = g^{x_n b_{n-1}}$. 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 a hash function h_b, which is implemented by using s_n as an AES encryption key and encrypting a block of all zeroes. [1]

At the end of this derivation process, a list of (α_n, s_n, b_n) tuples, each corresponding to 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, routing information block. This is done by repeatedly encrypting the padding block with each hop's shared secret.

—possibly add phi diagram—

β and γ are generated next. Starting from the terminal 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:

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

[1] 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 $b_n = h_b(s_n)$.

- ρ is implemented using AES-128-CTR.
- h_ρ is a hash function implemented using AES-128-CTR, by encrypting the string `hrhohrhohrhohrho` using the parameter to h_ρ as the encryption key.
- μ is implemented using HMAC-SHA256.
- h_μ is a hash function implemented using AES-128-CTR, by encrypting the string `hmu:hmu:hmu:hmu:` using the parameter to h_μ as the encryption key.
- β_i consists of either the destination identifier and the encrypted padding $\phi_{\nu-1}$, or a routing command n_{i+1}, a MAC γ_{i+1} and a encrypted routing command block β_{i+1} to forward to the next node.

`header.gamma` is set to γ_0, and `header.beta` is set to β_0. The final header structure when $\nu = 4$ 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.

—possibly add (beta, gamma) diagram—

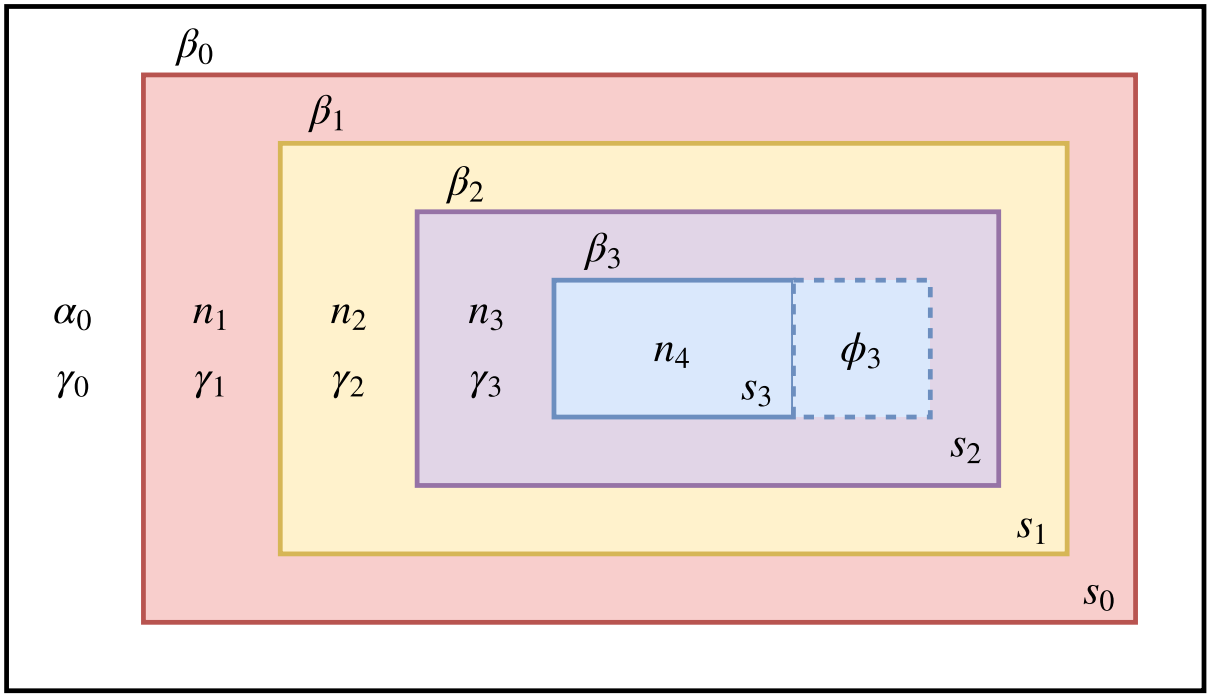

Figure 3.1: Sphinx header for a packet with 4 hops

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

3.1.4 Challenges Faced

Most of the challenges faced involves differences in language support for certain features between Java and Python.

There were a lot of byte array manipulations that primarily involved slicing arrays into different pieces, which is easily done in Python. However, in Java this caused the code to be very verbose and very hard to read.

The structure of how data is passed around also introduced readability issues, as the original implement made good use of tuple return types, which meant the creation of a lot of different classes in the Java codebase.

3.2 Loopix Client Library

3.2.1 Public Key Infrastructure

sqlite db

3.2.2 Path Selection

random selection, delay sampling

3.3 Chat Client

3.4 Testing Framework

3.4.1 Unit Tests

JUnit

3.4.2 Continuous Integration

Travis.CI (should this be under software engineering techniques in prep?)

3.4.3 Docker Containers

3.5 Challenges Faced

Chapter 4

Evaluation (Due 13th April)

4.1 Functional Testing

4.2 Performance

4.3 Viability on Mobile Devices

4.4 Security Evaluation

Chapter 5

Conclusion

Appendix A

Project Proposal

Bibliography