Bachelor's Thesis

Bachelor's Programme in Science

[matrix] over Tor: A federated darknet instant messaging solution

Quy Anh Nguyen

January 14, 2025

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki, Finland

Email address: info@cs.helsinki.fi
URL: http://www.cs.helsinki.fi/

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Bachelor's Programme in Science

Tekijä — Författare — Author

Quy Anh Nguyen

Työn nimi — Arbetets titel — Title

[matrix] over Tor: A federated darknet instant messaging solution

Ohjaajat — Handledare — Supervisors

Prof. Petteri Nurmi, Dr. Xiaoli Liu

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Bachelor's Thesis	January 14, 2025	30 pages, 1 appendix pages

Tiivistelmä — Referat — Abstract

Instant messaging is, without a doubt, one of the most common and arguably important applications of the Internet. Unfortunately, this technology, so fundamental to our society, is in the control of a handful of tech corporations. The number of users of instant messaging apps is estimated to be over 3 billion globally, and yet of that figure, the two biggest messaging platforms, WhatsApp and Facebook Messenger, have already accounted for over 2.5 billion [1]. Virtually all of the most popular messaging platforms are developed based on a centralised software system architecture, whereby in order for any user to send a message to any other user, their message has to first go through a central backend. These backends, each of which might consist of one or multiple servers under the hood, are owned and operated exclusively by private for-profit tech companies, giving them the ability to read users' messages, monitor when users are online, observe who is talking to whom, misuse users' messages for intrusive advertising, or forward users' messages to government agencies for mass surveillance. This extends to even platforms that claim users' messages are end-to-end encrypted like WhatsApp, since their clients and servers are almost always closed-source, leaving users with little means to verify claims of encryption and proper data handling. In this paper, I present an alternative instant messaging solution based on the [matrix] communication protocol and Tor onion services. Compared to mainstream messaging platforms, my solution provides a verifiably secure, private, and anonymous real-time communication experience.

ACM Computing Classification System (CCS)

Networks → Network protocols → Application layer protocols

Networks → Network types → Overlay and other logical network structures

Security and privacy → Software and application security → Social network security and privacy

Avainsanat — Nyckelord — Keywords

matrix, tor, instant messaging, darknet, fediverse

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Contents

1 Introduction **1**

2 [matrix]: A Communication Protocol **3**

3 Tor Onion Services **7**

3.1 Standard Circuit Establishment 7

3.2 Onion Service Mechanism 10

4 [matrix] over Tor: A Proof of Concept **14**

5 Security Considerations **20**

5.1 Attacks on Tor 20

5.1.1 Correlation Attacks 20

5.1.2 Congestion Attacks 22

5.1.3 Denial of Service Attacks 22

5.1.4 Supportive Attacks 23

5.1.5 Revealing Hidden Services Attacks 24

5.2 Attacks on [matrix] 25

6 Conclusions **27**

Bibliography **27**

A Use of AI in the Thesis

1 Introduction

In recent years, the percentage of Internet users who access online messaging services has consistently been around 95% [2]. This figure indicates that instant messaging is one of the most important applications of the Internet. Contradictorily, this technology, despite being used by people across the globe, is not operated by the general public and in the general public's interest. Instead, a small number of private for-profit tech companies have commercialised the technology and turned the resulting market into an oligopoly. Indeed, the number of users of instant messaging apps is estimated to be over 3 billion globally; of that figure, the two biggest messaging platforms, WhatsApp and Facebook Messenger, which are both owned by the same company, have already accounted for over 2.5 billion [1]. Consequently, most popular instant messaging platforms follow a centralised modus operandi. Specifically, in order for one user to send a message to another, the message always has to be sent through a central server (or a central backend consisting of multiple servers under the hood) that is controlled by the owner company. This, in turn, grants a few companies the ability to read users' messages, monitor when users are online, observe who is talking to whom, misuse users' messages for intrusive advertising, or forward users' messages to government agencies for mass surveillance. This extends to even platforms that claim users' messages are end-to-end encrypted like WhatsApp, since their clients and servers are virtually always closed-source, leaving users with no way to verify claims of encryption or understand how their data is being handled.

Within this context, the challenge is to develop a secure, private, and anonymous instant messaging solution. I shall begin by laying out several requirements for such a solution. The solution needs to provide a familiar user experience, comparable to Facebook Messenger, a widely popular messaging platform. Specifically, I want users to be able to log in on any device and retrieve all their messages. For this, the solution requires servers, where messages can be stored. At the same time, I want the messages to be end-to-end encrypted, so that they are only readable by their intended recipients and not by any third party, not even the servers. I also want servers to be federated and open-source, so that users can choose any server to store their data or even self-host their server if they so wish, whilst still being able to communicate with any other user on any server. Naturally, I want clients to be open-source as well, so users know and can verify exactly what is happening to their data on their phones or computers. Finally, I want my solution to provide anonymity, such that no client or server would require any other client or server's IP address. To satisfy these

requirements, I need an open standard for federated communications and an anonymous overlay network. Correspondingly, I chose the [matrix] protocol and the Tor network for my solution. Chapters 2 and 3 explain these two technologies, respectively. Chapter 4 demonstrates how the two technologies are combined to produce my solution, in theory and in implementation. Finally, Chapter 5 discusses several potential security threats against my messaging system and corresponding defensive strategies, referencing previous security research on Tor and [matrix].

2 [matrix]: A Communication Protocol

This chapter describes characteristics of the [matrix] protocol that are relevant to my instant messaging solution, based on the specification by the Matrix.org Foundation [3].

[matrix] is a federated instant messaging protocol. Fig. 2.1 demonstrates the concept of federation.

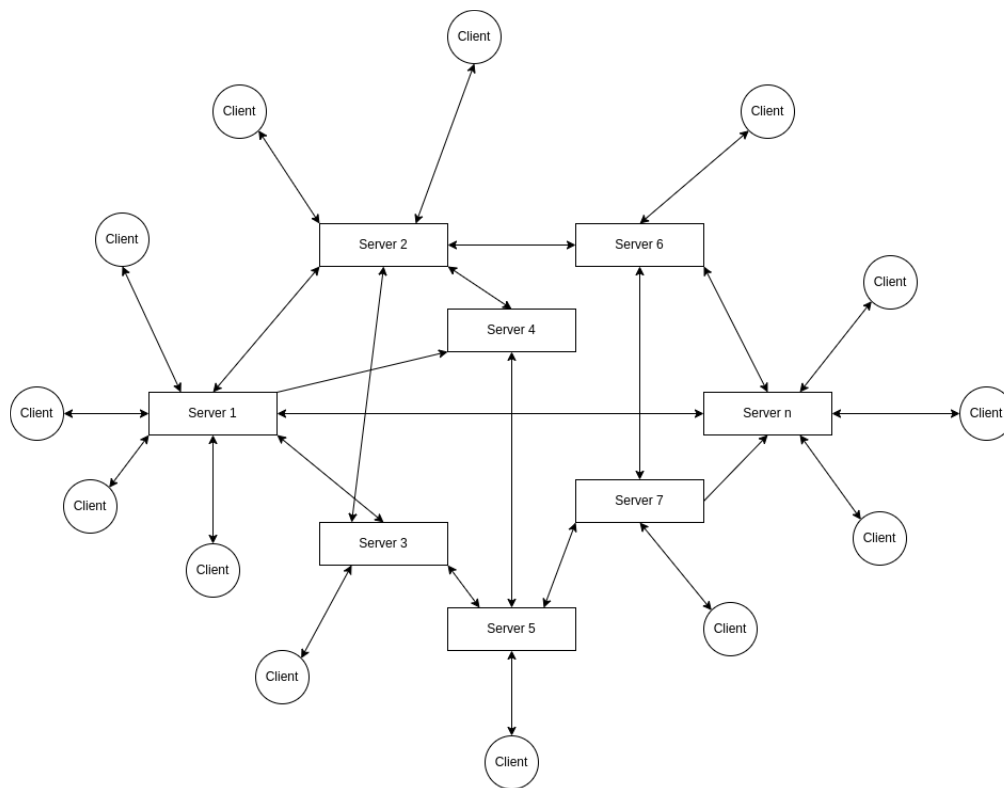

Figure 2.1: A sample federated network

A federated network consists of multiple servers and clients. There are direct bidirectional connections between a server and its clients as well as between servers. Each server provides access to the network to multiple clients, and the end-goal of the system is to enable communications between multiple clients that are connected to different servers. In Fig. 2.1, arrows denote the flow of data, as will be the case for figures throughout this paper.

In [matrix], servers are also called homeservers. Client-to-server and server-to-server commu-

nications take place over HTTPS, with Events being the sole units of data being exchanged over these communications. Before examining these communications, I shall first clarify a few concepts.

Each account registered on any [matrix] server has a globally unique `user ID`. This user ID is in the following format: `@localpart:domain`.

Whenever a user uses a client to log in to their homeserver with their account, a login session is created. Each login session is called a device, and within the scope of any account, each device has a unique `device_id`. A device does not have to be an actual physical entity; for instance, a mobile app and a browser tab both count as devices. Every device immediately sends a GET request to the `/sync` endpoint of the homeserver upon logging in, and the homeserver will keep this HTTP request indefinitely open so that real-time updates can be immediately sent to the device.

Each Event is a JSON object that has a `type` field. Events that serve the same purpose share the same type. For example, messages have the type `m.room.message`. A room is a channel for users to publish and subscribe to events. Each room has a globally unique `room ID` in the form of `!opaque_id:domain`.

Events are either message events or state events. Message events semantically represent one-off communication activities, e.g. a message being sent, an emoji reaction on a message is made, a file being uploaded, and so on. Message events normally have a `content` field. If End-to-End Encryption is enabled for a room, the content of the `content` field is encrypted inside a `ciphertext` field. This encryption is done using encryption keys stored locally on each device. State events, on the other hand, are persistent information about a room, e.g. room name, topic, avatar, member list, banned users, and so on.

In each room, a metadata field named `depth` is stored for each event. The first event in any room is always `m.room.create` and has a depth of 1. Every subsequent event must have a depth greater than that of any preceding event. [matrix] considers every event a node and every directed connection from an event to another event with higher consecutive depth an edge. This allows [matrix] to construct a Directed Acyclic Graph (DAG), whose topological order is the chronological ordering of the events. This graph is called the event graph of a room.

[matrix] synchronises the event graph and states of each room across all servers who have users that are members of the room in an eventually consistent manner. Eventual consistency

means that the servers collectively form a distributed data store which prioritises Availability and Partition Tolerance (see: CAP theorem by Eric Brewer [4]). This enables users to login to their homeservers and retrieve all their messages on any device, at any time. Let us now examine this synchronisation via an example. Suppose that Elliot wishes to send a message to his friend, Marcy. The message, in this case, would be an event of type `m.room.message`, as shown below:


```
{
    "content": {
        "body": "happy birthday!",
        "format": "org.matrix.custom.html",
        "formatted_body": "<b>happy birthday!</b>",
        "msgtype": "m.text"
    },
    "event_id": "$omzdsbgxjs216afc65qp7aux:matrix.elliot-at-helsinki
       .eu",
    "origin_server_ts": 1730660400000,
    "room_id": "!ikr5vw05oeasm4sf0ql1d7m8:matrix.elliot-at-helsinki.
       eu",
    "sender": "@root:matrix.elliot-at-helsinki.eu",
    "type": "m.room.message"
}
```


Notice that there is no field indicating that the event is to be sent directly to a user named Marcy. This is because in [matrix], 1:1 conversations and group chats are both conducted via rooms. In this case, Elliot and Marcy are exclusive members of the `!ikr5vw05oeasm4sf0ql1d7m8:matrix.elliot-at-helsinki.eu` room.

Elliot's device would send the content of the `m.room.message` event to his homeserver, `matrix.elliot-at-helsinki.eu`. The homeserver, upon receiving the event, would hash the event's content using SHA-256, append that hash to the event, sign the whole event using its private ED25519 key, append the signed event to the server's copy of the room's event graph, and forward the signed event to Marcy's homeserver. Marcy's homeserver would verify that the event hasn't been tampered with since it was sent by verifying both the hash and the signature, the latter using Elliot's homeserver's public key. Marcy's homeserver would also check various authorisation rules to ensure the event is allowed.

Upon successful verification and authentication, Marcy's homeserver would add the event to its copy of the event graph and send the new event to Marcy's device immediately as a

response for the `/sync` GET request. Marcy's device would immediately send another GET request to `/sync` upon receiving the message to await for any further real-time updates. This is known as long-polling. A chronological summary of the data flow is shown in Fig. 2.2.

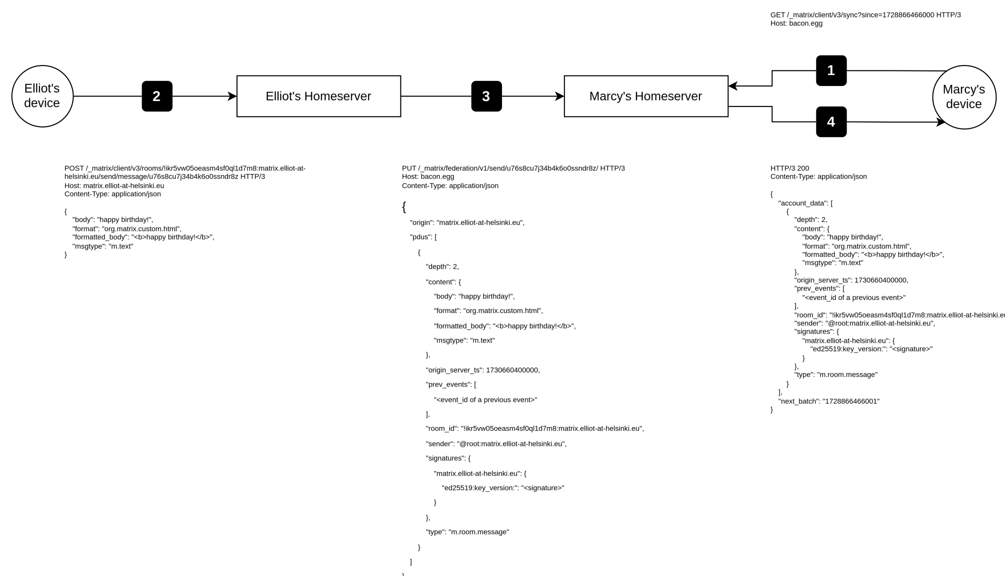

Figure 2.2: Sample Data Flow in [matrix]

In Fig. 2.2, `u76s8cu7j34b4k6o0ssndr8z` is a `transaction ID` generated by Elliot's device. This ensures that even if the HTTP request fails and gets retransmitted, the action does not get executed twice. In the PUT request from Elliot's homeserver to Marcy's homeserver, the depth of the message event is 2, as the first event would be `m.room.create`. There is also a `prev_events` field, which helps Marcy's homeserver determine where to insert the new event into the event graph.

3 Tor Onion Services

An onion service is a web server that provides various services over TCP [5]. However, unlike traditional web servers, an onion service is not accessible via an IP address, which reveals a host's actual geolocation. Instead, onion services can only be accessed anonymously via `.onion` top-level domains, e.g. `darkfailenbsdla5mal2mxn2uz66od5vtzd5qozslagrfzachha3f3id.onion`, which are not resolved into IP addresses using DNS [5]. This chapter details how clients can anonymously access onion services and how onion services remain anonymous when accessed by clients.

3.1 Standard Circuit Establishment

The Tor network consists, amongst other things, of relays, which are routers that simply forward packets of data from one Tor node to the next. A Tor node is any host that runs the Tor software, which can be a client, an onion service, or a relay. A Tor relay can also forward packets to a regular web server on the clearnet, in which case the relay is called an exit node. The IP addresses of relays are publicly available, allowing any Tor client to construct circuits, which are anonymous connection links between two hosts going through multiple intermediary relays. Circuits are the fundamental building blocks of the Tor network. This section explains how the standard 3-hop circuit is constructed and how clients use such circuits to anonymously access clearnet servers, i.e. servers with publicly available IP addresses. This standard 3-hop circuit is integral to the onion service mechanism that will be introduced in Section 3.2.

The IP addresses, public keys, ids, and roles of relays are stored in a consensus document, which in turn is distributed across authoritative servers officially deployed by the Tor developers [5]. There are 9 authoritative servers globally, the IP addresses of which are hardcoded into the Tor software, allowing clients to periodically retrieve the up-to-date consensus document [5]. The public keys that are used by relays to communicate with other Tor nodes will be referred to as onion public keys, to distinguish them from other kinds of public keys.

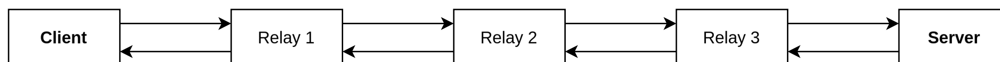

Figure 3.1: A standard 3-hop Tor circuit

Fig. 3.1 illustrates a standard Tor circuit. In this setup, the server and the relays all have publicly available IP addresses, but the server itself does not know the client's IP address.

The circuit construction process is as follows:

- Step 1: The client selects some random and distinct relays, typically three [6].
- Step 2: The client uses the curve25519 algorithm to generate a public-private key pair [7]. The generated public key is encrypted using the first relay's onion public key and sent to the first relay [6].
- Step 3: The first relay decrypts the payload using its onion private key to retrieve the public key generated by the client. The first relay also uses the curve25519 algorithm to generate another public-private key pair [7]. The generated public key is sent back to the client.
- Step 4: At this point, both the client and the relay have two public keys, one generated by the client and the other by the relay. Using these two keys, both parties calculate a shared secret, and using this shared secret, both parties calculate a set of symmetric encryption keys [7]. Subsequent data exchanged between the client and first relay will be encrypted using these symmetric keys.
- Step 5: Subsequent communications between the client and the second relay can now take place via the first relay. The circuit is said to have been extended to the first relay [6]. Steps similar to 2, 3, and 4 are taken to extend the circuit to the second relay [6].
- Step 6: Subsequent communications between the client and the third relay can now take place via the first and second relays. The circuit is said to have been extended to the second relay [6]. Steps similar to 2, 3, and 4 are taken to extend the circuit to the third relay [6].
- Step 7: The client can now send data to the server via the three relays.

The data that the client wishes to send to the server is encapsulated in payloads. When a circuit has been established, the client can send a payload to the server as follows:

- For each relay in the circuit, the client generates a unique `circuit_id` to identify the connection to that particular relay.
- The client adds the `circuit_id` corresponding to relay 3 and the IP address of the server as the next-hop address to the payload. This extended payload is encrypted using the symmetric keys shared between the client and relay 3.

- The client adds the `circuit_id` corresponding to relay 2 and the IP address of relay 3 as the next-hop address to the encrypted payload. This encrypted and extended payload is further encrypted using the symmetric keys shared between the client and relay 2.
- The client adds the `circuit_id` corresponding to relay 1 and the IP address of relay 2 as the next-hop address to the payload. This extended payload is encrypted using the symmetric keys shared between the client and relay 1.
- After multiple layers of extension and encryption, the client sends the payload to relay 1.
- Relay 1 decrypts the payload using the symmetric keys shared between it and the client. It discovers relay 2 as the next hop and so forwards the payload to relay 2. Relay 1 stores the IP address of the client, the IP address of relay 2, and the discovered `circuit_id` in memory.
- Relay 2 decrypts the payload using the symmetric keys shared between it and the client. It discovers relay 3 as the next hop and so forwards the payload to relay 3. Relay 2 stores the IP address of relay 1, the IP address of relay 3, and the discovered `circuit_id` in memory.
- Relay 3 decrypts the payload using the symmetric keys shared between it and the client. It discovers the server as the next hop and so forwards the payload to the server. Relay 3 stores the IP address of relay 2, the IP address of the server, and the discovered `circuit_id` in memory.

The above process was outlined in the original paper that introduced Tor [8]. The same paper also described how the server can send a payload back to the client:

- The server sends the payload to relay 3.
- Using the server's IP address, relay 3 searches its memory for the corresponding `circuit_id`. Relay 3 then knows it must forwards the payload to relay 2. Relay 3 encrypts the payload using the symmetric keys associated with that `circuit_id`, i.e., the symmetric keys shared between it and the client. The encrypted payload is then sent to relay 2.
- Using relay 3's IP address, relay 2 searches its memory for the corresponding `circuit_id`. Relay 2 then knows it must forwards the payload to relay 1. Relay 2 encrypts the payload using the symmetric keys associated with that `circuit_id`, i.e., the symmetric keys shared between it and the client. The encrypted payload is then sent to relay 1.

- Using relay 2's IP address, relay 1 searches its memory for the corresponding `circuit_id`. Relay 1 then knows it must forwards the payload to the client. Relay 1 encrypts the payload using the symmetric keys associated with that `circuit_id`, i.e., the symmetric keys shared between it and the client. The encrypted payload is then sent to the client.
- The client, being the only node in the setup to know the full path of the circuit, decrypts the payload layer by layer using the symmetric keys agreed between it and relays 1, 2, and 3, in that order.

3.2 Onion Service Mechanism

As previously mentioned, onion services do not have IP addresses, so they cannot be accessed via a standard circuit. Instead, onion services are accessed through special 6-hop circuits, which conceal the IP addresses of both clients and onion services from each other. The process of constructing a 6-hop circuit between a client and an onion service involves constructing multiple standard circuits under the hood. This section describes this process, based on the comprehensive analysis of V3 onion services by Wang et al. [5]. Throughout this section, any circuit other than the 6-hop circuit is to be understood as the standard circuit described in the previous section, although the number of hops might not necessarily be 3.

I shall first introduce a few concepts. An Introduction Point (IPO) is a relay acting as a reverse proxy for an onion service. The IPO is publicly accessible via an IP address like all relays and forwards incoming requests to the onion service via a circuit [5]. A Rendezvous Point (RPO) is a relay that is connected to both a client and an onion service, via one circuit for each of them [5]. A Hidden Service Directory (HSDir) is a relay that, apart from forwarding packets, also stores information related to onion services and makes this information publicly available for querying. Information about onion services is stored across HSDirs as a Distributed Hash Table [5].

When a Tor node first connects to the network after configuring itself as an onion service, the node generates a public-private key pair and computes a new `.onion` address using the public key of the pair [5]. The onion service then selects three random and distinct relays as IPOs, and constructs a circuit to each IPO, as shown in Fig. 3.2 [5].

After setting up the circuits to the IPOs, the onion service creates a descriptor, which is a data structure that contains, amongst other things, the generated `.onion` address, the corresponding public key, and the IP addresses of the three IPOs [5]. Then, via circuits, the

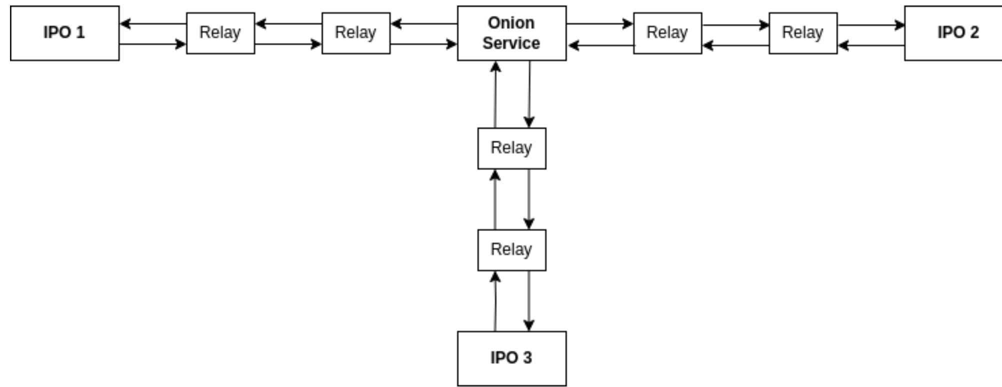

Figure 3.2: Onion service sets up circuits to 3 IPOs.

onion service uploads this descriptor to HSDirs, up to six diffent ones [9].

Suppose a client has obtained the `.onion` address via some out-of-band channel and now wishes to connect to the onion service. Based on the `.onion` address, the client calculates the `descriptor ID` of the onion service's descriptor [5]. Based on the `descriptor ID`, the client can calculate the `ID`s of HSDirs that store the descriptor, and look up the IP addresses of those HSDirs in the consensus document [5]. Then, the client can, via circuits, query the HSDirs for the descriptor using the `descriptor ID` and obtain the onion service's IPOs' IP addresses [5].

Next, the client selects a random relay as the RPO and establishes a circuit to this RPO, as shown in Fig. 3.3 [5].

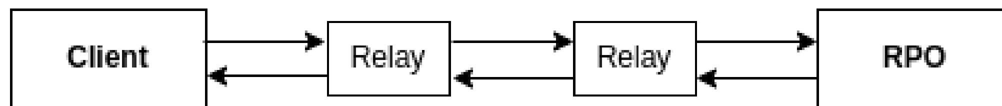

Figure 3.3: Client sets up a circuit to an RPO.

Using this circuit, the client creates an authentication cookie and sends it to the RPO [5]. The client also establishes another circuit to any of the three IPOs, as shown in Fig. 3.4 [5]. Using this circuit, the client sends the IP address of the RPO, the authentication cookie, and the public key of a generated curve25519 key pair to the onion service [7][10]. This data is encrypted using the onion service's public key before being sent [10].

The onion service decrypts the data using its own private key to retrieve the curve25519 public key, authentication cookie, and the RPO's IP address. Afterwards, the onion service establishes another circuit to the RPO, as shown in Fig. 3.5 [5]. The onion service also generates its own curve25519 key pair [5]. Using the public key of this pair and the public

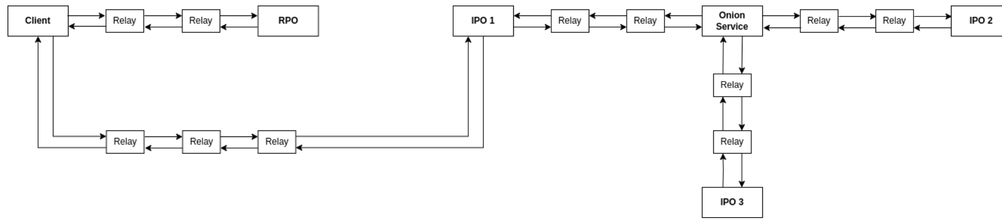

Figure 3.4: Client sets up a circuit to an IPO.

key generated by the client, the onion service can calculate the shared encryption keys. Via this circuit, the onion service sends the cookie and the public key of the generated pair back to the RPO [5][10].

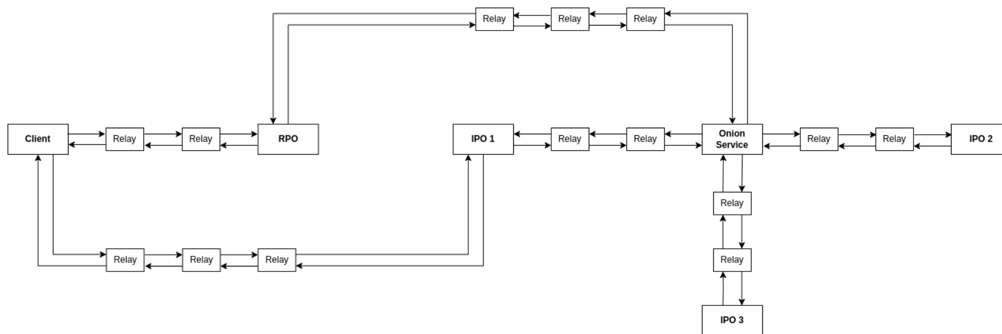

Figure 3.5: Onion service sets up a circuit to an RPO.

The RPO, after receiving the same cookie from two circuits, understands that the two hosts from both ends (i.e. the client and the onion service) want to communicate with each other. The RPO then bridges the two circuits to create one big 6-hop circuit between the client and the onion service, as shown in Fig. 3.6 [5]. The RPO then acts as a regular relay, and TCP connections between the client and the onion service can take place via the 6-hop circuit. The RPO also forwards the curve25519 public key generated by the onion service back to the client, so that the client can calculate the shared encryption keys [10].

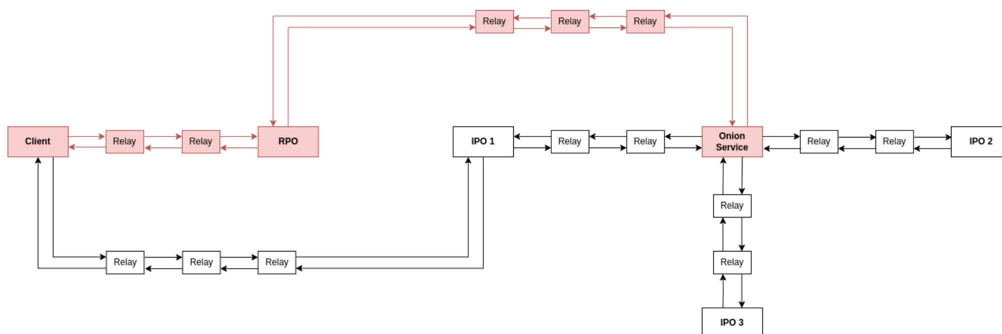

Figure 3.6: The 6-hop circuit is complete.
The client, the relays that make up the circuit, and the onion service are highlighted in red.

I already discussed how data is encrypted and decrypted throughout a 3-hop circuit in the previous section. Based on that process, we can logically deduce the order of encryption and decryption operations in a 6-hop circuit, since a 6-hop circuit is just the concatenation of two 3-hop circuits. Specifically, when the client wishes to send data to the onion service, it encrypts the data using the encryption keys shared between it and the onion service, the RPO, and the two relays preceding the RPO, in that order. In the first three hops, each hop decrypts its layer of encryption and forwards it to the next hop. When the data reaches the RPO, the RPO decrypts its layer of encryption, leaving only one layer of encryption left: the encryption done using the keys shared between the onion service and the client. The RPO then sends this data, still encrypted with one layer, to the next relay. Each of the last 3 relays subsequently adds their own layer of encryption. The onion service then decrypts all four layers of encryption to retrieve the original data. As such, Tor provides end-to-end encryption between a client and an onion service.

4 [matrix] over Tor: A Proof of Concept

In my theoretical instant messaging solution, there are multiple [matrix] homeservers globally, each being run as an onion service. This way, clients can anonymously connect to their homeservers, and homeservers can anonymously communicate with each other. This chapter provides a technical Proof of Concept for my solution.

For the Proof of Concept, I used a basic setup, consisting of two Synapse homeservers and two Element clients, as shown in Fig. 4.1. Element[†] is a popular [matrix] client, while Synapse[‡] is a popular [matrix] homeserver implementation. In my setup, both Synapse homeservers are onion services. Thus, for each Element client to communicate with their respective Synapse homeserver, or for the two Synapse homeservers to communicate with each other, the 6-hop circuit shown in Fig. 3.6 must be established. I hosted each homeserver on a separate EC2[§] instance running Debian, provided by Amazon Web Services.

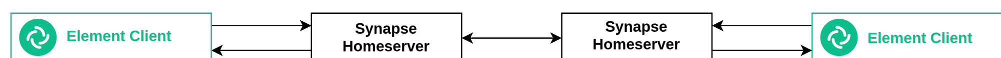

Figure 4.1: Proof of Concept Setup

First, I SSHed into the EC2 instance and installed Tor:

```
sudo apt update && sudo apt install tor
```

I then configured an onion service by editing the `/etc/tor/torrc` file and adding the following lines:

```
AutomapHostsOnResolve 1
DNSPort 53

SocksPort 9050
SocksPolicy accept *

HiddenServiceDir /var/lib/tor/synapse
```

[†]https://github.com/element-hq/element-desktop
[‡]https://github.com/element-hq/synapse
[§]https://aws.amazon.com/ec2/

```
HiddenServicePort 80 127.0.0.1:8448
HiddenServicePort 443 127.0.0.1:8448
HiddenServicePort 8008 127.0.0.1:8008
HiddenServicePort 8448 127.0.0.1:8448
```

Upon restarting tor with `sudo systemctl restart tor`, I found the `.onion` address of my onion service at `/var/lib/tor/synapse/hostname`:

```
root@ip-172-31-32-92:~# cat /var/lib/tor/synapse/hostname
zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion
```

The `.onion` address would be my Synapse server name. Next, I needed to ensure that outbound connections from my Synapse homeserver are routed through Tor. Here, a complication arised: Synapse only supports HTTP proxies, but the Tor network is only accessible via a SOCKS5 proxy. To overcome this complication, I set up an HTTP proxy that would forward incoming requests to the SOCKS5 proxy of Tor. Specifically, I set up the proxy so that all requests to `http://localhost:8118` would be forwarded to `socks5://localhost:9050`, which would in turn forward the request to the Tor network. I did this using privoxy*:

```
sudo apt install privoxy && \
sudo echo """forward-socks5t / localhost:9050 .
forward-socks5t .onion 127.0.0.1:9050 .""" >> /etc/privoxy/config &&
    \
sudo systemctl restart privoxy
```

The [matrix] protocol only supports using HTTPS for server-to-server communications. TLS certificates, on the other hand, are not usually available for `.onion` addresses [11]. To overcome this, I used self-signed certificates and disabled certificate verification for server-to-sever communications. I generated the self-signed certificate and the corresponding private key as follows:

```
DOMAIN="zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.
   onion"

mkdir -p /etc/synapse/certs/$DOMAIN

openssl genpkey -algorithm RSA -out /etc/synapse/certs/$DOMAIN/
   privkey.pem -pkeyopt rsa_keygen_bits:2048
```

*https://www.privoxy.org/

```
openssl req -new -key /etc/synapse/certs/$DOMAIN/privkey.pem -out /
   etc/synapse/certs/$DOMAIN/csr.pem \
  -subj "/C=US/ST=Tor/L=Hidden/O=Synapse/OU=Matrix Homeserver/CN=
     $DOMAIN"

openssl x509 -req -days 365 -in /etc/synapse/certs/$DOMAIN/csr.pem -
   signkey /etc/synapse/certs/$DOMAIN/privkey.pem \
  -out /etc/synapse/certs/$DOMAIN/fullchain.pem
```

Next, I set up the Synapse homeserver. Following the official documentation*, I installed the dependencies:

```
sudo apt install -y lsb-release wget apt-transport-https
sudo wget -O /usr/share/keyrings/matrix-org-archive-keyring.gpg
   https://packages.matrix.org/debian/matrix-org-archive-keyring.gpg
echo "deb [signed-by=/usr/share/keyrings/matrix-org-archive-keyring.
   gpg] https://packages.matrix.org/debian/ \$(lsb_release -cs) main
   " |
    sudo tee /etc/apt/sources.list.d/matrix-org.list
sudo apt update
sudo apt install matrix-synapse-py3
```

When prompted for the server name, I entered `zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion`. I then configured the homeserver by modifying the `/etc/matrix-synapse/homeserver.yaml` file to the following:

```
pid_file: "/var/run/matrix-synapse.pid"
listeners:
  - port: 8008
    tls: false
    type: http
    x_forwarded: true
    bind_addresses: ['::1', '127.0.0.1']
    resources:
      - names: [client]
        compress: false
  - port: 8448
    tls: true
    type: http
    bind_addresses: ['::1', '127.0.0.1']
```

*https://element-hq.github.io/synapse/latest/setup/installation.html

```
    resources:
      - names: [federation]
        compress: false
database:
  name: sqlite3
  args:
    database: /var/lib/matrix-synapse/homeserver.db
log_config: "/etc/matrix-synapse/log.yaml"
media_store_path: /var/lib/matrix-synapse/media
registration_shared_secret: "7cYGl_IF=D,DqeK-EHcjf0uCg6-ycW,
   oT4CdwwPyhea;l9YHV&"
signing_key_path: "/etc/matrix-synapse/homeserver.signing.key"
trusted_key_servers:
  - server_name: "matrix.org"
    accept_keys_insecurely: true
allow_public_rooms_over_federation: true
use_insecure_ssl_client_just_for_testing_do_not_use: true
federation_verify_certificates: false
serve_server_unknown: true
tls_certificate_path: "/etc/synapse/certs/
   zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion/
   fullchain.pem"
tls_private_key_path: "/etc/synapse/certs/
   zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion/
   privkey.pem"
```

Most importantly, I specified the TLS certificate and private key paths, set the client and federation ports to `8008` and `8448` respectively, enabled TLS for the federation port, and disabled certificate verification for federation communications.

Next, I configured Synapse to use `http://localhost:8118` for outgoing traffic. This was done by adding the following to `/etc/default/matrix-synapse`:

```
HTTP_PROXY=http://localhost:8118
HTTPS_PROXY=http://localhost:8118
```

I finally started the homeserver with the specified configurations by running `sudo systemctl restart matrix-synapse`.

The above steps were performed on both EC2 instances to set up two homeservers. After-

wards, I tested that federation over Tor indeed worked.

For the test, I first created two users, `@admin:zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion` and `@admin:3s5wmhttc772irrsr4xcxykyxcf5sm7zrfzpyg56veimhghr7bsmclqd.onion`, by running the following command on both EC2 instances:

```
sudo register_new_matrix_user -c /etc/matrix-synapse/homeserver.yaml
    http://localhost:8008 -u admin -p jNCH8ZNZ)4HVvMw#
```

I then installed the `tor` and `element-desktop` packages on both laptops:

```
sudo apt update && \
sudo apt install -y tor wget apt-transport-https && \
sudo wget -O /usr/share/keyrings/element-io-archive-keyring.gpg
   https://packages.element.io/debian/element-io-archive-keyring.gpg
    && \
echo "deb [signed-by=/usr/share/keyrings/element-io-archive-keyring.
   gpg] https://packages.element.io/debian/ default main" | sudo tee
    /etc/apt/sources.list.d/element-io.list && \
sudo apt update && \
sudo apt install element-desktop
```

Afterwards, I opened the Element client on two separate machines and proxied the client over `socks5://localhost:9050`, i.e., over Tor. On one machine, I logged in as `@admin:zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion` by specifying the homeserver as `http://zmnuzfqefxf6gaogejlg3jduk2pcjf3upqmt6opv37wtxraiqomupzid.onion:8008/`, the username as `admin`, and the password as `jNCH8ZNZ)4HVvMw#`. I logged in as the other user on the other machine by repeating the same steps except with a different `.onion` address.

The two users, logged in to two onion service homeservers, were able to communicate successfully, as shown in Fig. 4.2.

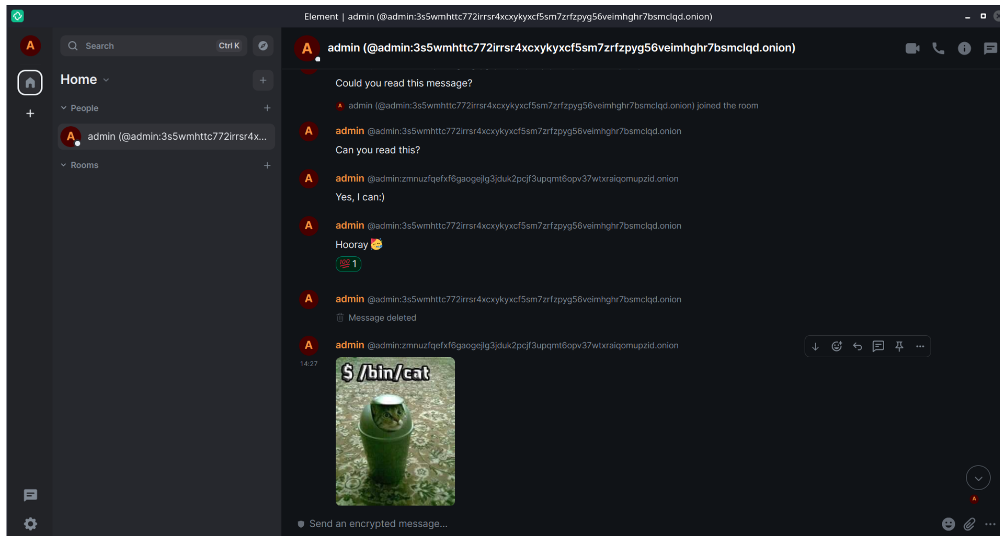

Figure 4.2: Successful Federation over Tor

5 Security Considerations

My system is made up of [matrix] homeservers being run on top of the Tor network. As such, vulnerabilities in either the [matrix] protocol or the Tor network would affect my system. In this chapter, I discuss potential attacks against my system, evaluate the threats they pose, and suggest some defence strategies. I based this chapter on previous security research on Tor and [matrix].

5.1 Attacks on Tor

Evers et al. identified seven main categories of attacks on the Tor network [12]. Amongst these seven categories, five appear to be able to partly compromise my system in one way or another and therefore deserve examination in further details.

5.1.1 Correlation Attacks

In correlation attacks, the attacker is assumed to be in control of both the entry node and the exit node in a circuit between a client and a server [12]. In Section 3.1, I already mentioned that when a client uses Tor to anonymously connect to a server on the clearnet, a 3-hop circuit is established between them. Particularly, in Fig. 3.1, relay 1 is the entry node, and relay 3 is the exit node. If the traffic entering the entry node correlates with the traffic exiting the exit node, the attacker can reasonably conclude that the client sending data to the entry node is trying to communicate with the server that the exit node is trying to send data to. For instance, if, in a short period of time, packets being sent to the entry node have roughly the same size and volume as the packets being sent from the exit node, the attacker can make such a conclusion.

Before continuing, it is helpful to note that in Tor, the basic unit of data that is exchanged between any two relays is a cell, which is a packet of data with various header fields and a body [13]. While the body is encrypted during transmission, the header fields are not. Each cell has a `Command` header field, whose value represent an `Identifier` like `PADDING`, `CREATE`, `RELAY`, or `RELAY_EARLY` [13].

In my setup, the homeservers are both onion services, so there are no exit nodes, and a classic

correlation attack as described above would not work against my system. However, variants of the correlation attack, like the Relay Early Traffic Confirmation Attack (RETCA), might [12]. Recall that when a client needs to connect to an onion service for the first time, it must construct a circuit to an HSDir and request the onion service's descriptor from that HSDir. In RETCA, the attacker needs to compromise this circuit and be in control of both the circuit's entry node and the HSDir, as shown in Fig. 5.1. Being in control of the HSDir means the attacker can intercept incoming requests for descriptors of onion services, and the job of the attacker is to figure out which client is requesting a particular onion service's descriptor. The attacker can do this by manipulating the HSDir to send out cells via the circuit back to the client such that the values of the `Command` header of those cells appear in a specific sequence. This sequence would encode the requested `.onion` address in a manner konwn only to the attacker. Then, if the entry node detects the same sequence, it can decode it to determine the onion service that is being requested by the client.

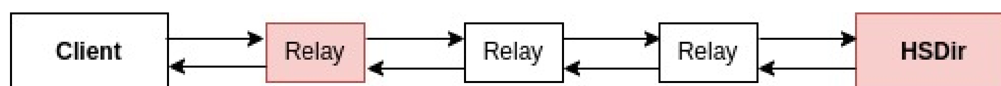

Figure 5.1: A client-HSDir circuit in the Relay Early Traffic Confirmation Attack. Compromised nodes are highlighted in red.

One successful RETCA attack, or any correlation attack, for that matter, would not post a significant risk to my system, since it would only reveal that a client is attempting to connect to a chat server (whose real IP address and location would still be unknown), nothing more. Neither would a successful correlation attack on the circuit between two homeservers provide any useful information to the attacker, since it would only reveal that two chat servers are communicating with each other and nothing about their users. To determine a potential connection between two users, at least three successful correlation attacks are required: one to prove the link between one user's client and their homeserver, another to prove the link between two homeservers, and another to prove the link between the other homeserver and the other user's client. Still, even if all three attempts were successful, the hypothetical connection between two clients would only be proven to be possible but could not in any way be proven to exist. Specifically, even if one can prove that client A is communicating with homeserver A, homeserver A is communicating with homeserver B, and homeserver B is communicating with client B, it does mean that they have in any way proven that client A is explicitly communicating with client B, since each homeserver might itself serve multiple clients. I therefore conclude that correlation attacks on their own pose little threat to my system.

5.1.2 Congestion Attacks

Congestion attacks attempt to determine which relays make up a circuit constructed by a particular Tor client. The last known successful congestion attack on Tor was designed and demonstrated by Evans et al. in 2009 [14]. The attack has been patched since its revelation [12]. Still, in the future, if a congestion attack could be successfully mounted, it would serve as a starting point to further compromise my system. Suppose a threat actor with unlimited resource like a nation-state can mount a congestion attack against the circuit between a client and a homeserver. Upon discovering the IP addresses of the relays in the circuit, the actor can attempt to wiretap the physical cables between the RPO and the relay immediately succeeding it (see: Fig. 3.6). This is the weakest link in my 6-hop circuit, since the first three layers of encryption have been removed, and only two layers of encryption remain before the attacker can read messages in plaintext: one layer is the encryption done using the encryption keys shared between the Tor client and the onion service, and the other layer is the encryption done by [matrix]. Even if the threat actor were unable to immediately break these two layers of encryption, they could still log the encrypted messages indefinitely until more powerful computers that can decrypt the messages are invented. This surveillance strategy is called Harvest Now, Decrypt Later (HNDL) [15], and a congestion attack could potentially serve as the prerequisite for HNDL.

5.1.3 Denial of Service Attacks

DoS attacks typically aim to flood a targest host with traffic so as to slow down the victim's ability to respond to legitimate traffic or to make the web services provided by the victim unavailable entirely.

A Distributed Denial of Service typically floods a target with UDP packets from multiple sources via the use of a botnet, but since Tor only supports TCP, my onion service homeservers are immune to this classic UDP-based DDoS attack [12]. However, various TCP-based DDoS attacks, as identified by Khan et al., could be employed against my homeservers [16]. The SYN attack, for instance, floods a host with SYN segments that are used to initiate the three-way handshake. The attacker subsequently never completes the three-way handshake by not responding with ACK upon receiving SYN-ACK from the targetted host, leaving these connections half-open indefinitely and wasting the memory of the victim. Another TCP-based attack is ECE flood, where the attacker floods a server with segments with the ECE flag set. ECE stands for Explicit Congestion Notification-Echo, so in fact the attacker is falsely signalling to the server that there is network congestion when there is none, causing

the server to needlessly limit its transmission rate and slow down its services. To mitigate these TCP-based DDoS attacks, I can split each Synapse homeserver into multiple processes called workers that can scale independently and then use a tool called Onionbalance as a reverse proxy to distribute incoming requests to those workers [17][18]. Also, on the Onionbalance reverse proxy, I can enable SYN cookies and disable ECN. This way, most malicious SYN and ECE segments should be blocked by my reverse proxy before they reach the homeserver processes, and even if some malicious traffic does reach my homeserver processes, each process can be made to scale automatically as required, allowing my system as a whole to absorp much heavier traffic.

Up until February 2013, Tor was also vulnerable to the Sniper Attack, an attack discovered by the U.S. Naval Research Laboratory that allowed an attacker to disable any Tor relay with minimal costs [19]. To shut down a particular relay, the attacker could construct a circuit to a destination host controlled by the attacker with the target relay as the entry node. Afterwards, the attacker would have their client stop receiving data from the entry node while at the same time use the destination host to flood the circuit with traffic. The traffic would travel through the circuit in reverse until it reaches the entry node. Since the attacker's client was not receiving data, the traffic would gradually occupy more and more space in the buffer of the first relay until it had to shut down the `tor` process due to the buffer being full. If a similar attack were to be discovered in the future, an attacker could selectively disable the IPOs of a homeserver or the HSDirs holding the descriptors for those IPOs, thereby making homeservers unavailable to users.

It is worth noting that in my solution, homeservers would form a federated network whose true size would be unknown. Therefore, one successful DDoS attack would not prevent users from accessing the service, since they can simply switch to another functional homeserver. To shut down my system entirely, a threat actor needs to somehow locate all the onion services that are [matrix] homeservers and successfully DDoS all of them.

5.1.4 Supportive Attacks

Supportive attacks are, as their name implies, supportive in nature. That is, these attacks themselves do not deanonymise onion services or disrupt the Tor network but rather serve as prerequisites for such attempts. With regards to my instant messaging system, the most relevant supportive attack is the Sybil attack.

Recall that in Section 5.1.1, to perform a RETCA attack, an attacker needs to be in control

of a circuit's entry node and the HSDir responsible for the descriptor of an onion service. Since the entry node is chosen at random by the client and the attacker does not know which onion service the client is trying to access at the beginning, meaning they cannot reasonably calculate which HSDir is responsible for the onion service's descriptor, the attacker cannot predict the entry node and HSDir in advance to somehow take control of them by e.g. hacking or physically seizing the relays. Instead, what the attacker has to do is to deploy a large number of relays to increase the chances of the entry node and HSDir being two relays in their control. This is what a Sybil attack does: deploying a large number of relays to gain disproportionate influence over the network [12]. There is no way to completely prevent Sybil attacks on Tor, as Tor rely on volunteers to run relays and so there is always the possibility of a threat actor deploying their own malicious relays. However, for my instant messaging solution specifically, I can consider using the Lokinet overlay network instead of Tor to mitigate Sybil attacks.

According to its whitepaper, Lokinet is an anonymous overlay network where clients, relays, and servers communicate anonymously via an onion routing protocol, just like Tor [20]. In Lokinet, SNApps are anonymous web servers accessible only via a special `.loki` address, similar to Tor's onion services. However, unlike Tor, Lokinet mitigates Sybil attacks by imposing a financial constraint on attackers: running a relay requires the operators to deposit actual money. Specifically, an operator must purchase an anonymous cryptocurrency and time-lock their funds upon deploying a relay, i.e. forfeiting the right to use those funds for a certain amount of time. Only by behaving honestly as a functional and high-performance relay would the time-lock be lifted and the operator be rewarded with extra cryptocurrencies. On the other hands, relays that perform poorly or maliciously can be flagged by other nodes, and being flagged by enough nodes would both prevent the malicious relays from earning their rewards and extend the time-lock on their collateral funds. Thus, deploying enough relays to mount an effective Sybil attack would be prohibitively expensive. Therefore, I can consider running my homeservers as SNApps instead of onion services and federating them over Lokinet instead of Tor to mitigate the Sybil attack.

5.1.5 Revealing Hidden Services Attacks

These attacks attempt to reveal the real location of an onion service, and due to their critical nature, they tend to be patched as soon as they are discovered. Prior to the introduction of guard nodes, i.e. highly reliable and trusted relays that are consistently chosen by clients as entry relays when building circuits, Overlier and Syverson were able to reveal the IP address of any onion service if they controlled only one relay in the circuit between a client and that

onion service [21]. If a flaw in Tor's design enabling an attack of this severity were again discovered, a threat actor with legal power, i.e. a government, might locate a homeserver's actual location and seize it by force. They would nevertheless still have to break [matrix]'s end-to-end encryption before they could read that homeserver's users' messages.

5.2 Attacks on [matrix]

As noted by Albrecht et al., various vulnerabilities have been discovered in Synapse and Element [22]. Synapse is the reference implementation of the [matrix] homeserver, while Element is the reference implementation of the [matrix] client.

With respect to Synapse, CVE-2021-41281 is a vulnerability that allows attackers to trick certain homeservers of versions lower than 1.47.1 into downloading random files, albeit in random directories [23]. If an attacker exploited this vulnerability to cause an affected homeserver to download a large number of files, the attacker could cause the homeserver's host to run out of storage, thereby affecting the normal functionality of the homeserver. Another related vulnerability is CVE-2022-31052, which allows an attacker to kill the Synapse process on a particular host by exploiting an unbounded recursion bug in the URL preview feature of Synapse homeservers of version 1.61.1 or below [24]. Both these vulnerabilities could be exploited with few resource requirements, meaning an attacker would find it relatively easy to shutdown a homeserver via these exploits than by performing a Denial of Service attack, as I have discussed in Section 5.1.3. Other discovered vulnerabilities, such as CVE-2021-39163 and CVE-2021-39164, allowed attackers with access to a particular `room ID` to retrieve various metadata of that room, e.g. name, avatar, topic, and member list [25][26].

Regarding Element, two discovered vulnerabilities are rather noteworthy. Namely, CVE-2022-23597 is a vulnerability that, while requiring certain user interactions, enabled an attacker to execute any binary on the victim's computer [27]. In theory, if an attacker could somehow download a certain file onto the victim's computer, either by tricking the victim to download the file themselves or via some other exploits, the attacker would then be able to run any code on the victim's machine. This in turn means the attacker would be able to perform any malicious action, e.g. extract the decryption keys on the victim's device, decrypt their messages, and send the messages to the attacker. Nevertheless, to the best of Element's developers' knowledge, this scenario was never observered in the wild prior to the vulnerability's fix [27]. The other notable vulnerability is CVE-2024-47771, whereby the client sometimes leaked its access token to third parties, who would then be able to authen-

ticate with the homeservers and receive the user's messages, although those messages would still be encrypted [28].

The above vulnerabilities have demonstrated that as with any software, [matrix] clients and servers are not immune to flaws. Such flaws have to be constantly detected and corrected in order to provide users with a reliable, secure, and private messaging experience. Researchers interested in discovering such flaws could, for a start, refer to a previous penetration testing attempt on Synapse's implementation of the [matrix] Server-Server API by Hivron Stenhav [29]. While the attempt did not prove to be successful, Stenhav identified 14 potential but untested vulnerabilities. Interested researchers could verify those vulnerabilities and perform similar penetration testing attempts on the Client-Server API.

6 Conclusions

In this paper, I presented a federated darknet instant messaging solution that provides an anonymous, private, secure, and user-friendly real-time communication experience. I explained the two technologies that underlie my solution, the [matrix] federated communication protocol and Tor onion services. I explained how my solution would work in theory and demonstrated the solution's feasibility via a proof of concept. Additionally, I discussed potential security threats to my proposed system, referencing previous research on the security of Tor and [matrix]. The federated nature of my proposed system, as enabled by the [matrix] protocol, allows users to control exactly where they wish to store their messages, while the anonymous nature of the chat servers in my proposed system, as enabled by the Tor protocol, allows users to discreetly access the service. In my proposed system, two layers of end-to-end encryption, provided by [matrix] and Tor respectively, further secure users' messages against eavesdroppers. As such, I have reasonable confidence to believe that my instant messaging solution is superior to mainstream centralised instant messaging platforms, with regards to protecting users' privacy and anonymity. Still, as Chapter 5 has shown, no system is invulnerable. As such, continuous efforts to secure the Tor and [matrix] protocols as well as their implementations are required in order to keep my instant messaging solution reliably secure, private, and anonymous.

Bibliography

[1] David Curry. *Messaging App Revenue and Usage Statistics (2024) - Business of Apps*. 2024. URL: https://www.businessofapps.com/data/messaging-app-market/ (visited on 12/02/2024).

[2] Statista. *Share of internet users accessing online chat and messenger services monthly worldwide from 1st quarter 2022 to 1st quarter 2024*. 2024. URL: https://www.statista.com/statistics/1489440/chat-and-messenger-service-usage/ (visited on 12/10/2024).

[3] The Matrix.org Foundation. *Matrix Specification*. 2024. URL: https://spec.matrix.org/v1.12/ (visited on 12/02/2024).

[4] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *SIGACT News* 33.2 (June 2002), 51–59. ISSN: 0163-5700. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[5] Chunmian Wang et al. "A Comprehensive and Long-term Evaluation of Tor V3 Onion Services". In: *IEEE INFOCOM 2023 - IEEE Conference on Computer Communications*. 2023, pp. 1–10. DOI: 10.1109/INFOCOM53939.2023.10229057.

[6] The Tor Project. *Creating circuits - Tor Specifications*. 2024. URL: https://spec.torproject.org/tor-spec/creating-circuits.html (visited on 12/02/2024).

[7] The Tor Project. *CREATE and CREATED cells - Tor Specifications*. 2024. URL: https://spec.torproject.org/tor-spec/create-created-cells.html (visited on 12/02/2024).

[8] Roger Dingledine, Nick Mathewson, and Paul Syverson. "Tor: the second-generation onion router". In: *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*. SSYM'04. San Diego, CA: USENIX Association, 2004, p. 21.

[9] David Goulet et al. *Hidden-service statistics reported by relays*. 2015. URL: https://research.torproject.org/techreports/hidden-service-stats-2015-04-28.pdf (visited on 12/02/2024).

[10] The Tor Project. *The introduction protocol [INTRO-PROTOCOL] - Tor Specifications*. 2024. URL: https://spec.torproject.org/rend-spec/introduction-protocol.html (visited on 12/02/2024).

[11] The Tor Project. *Tor Project | HTTPS for your Onion Service*. 2024. URL: https://community.torproject.org/onion-services/advanced/https/ (visited on 12/02/2024).

[12] B. Evers et al. *Thirteen Years of Tor Attacks*. 2019. URL: https://github.com/Attacks-on-Tor/Attacks-on-Tor/blob/master/Thirteen%20years%20of%20Tor%20Attacks.pdf (visited on 12/02/2024).

[13] The Tor Project. *Cells (messages on channels) - Tor Specifications*. 2024. URL: https://spec.torproject.org/tor-spec/cell-packet-format.html (visited on 12/02/2024).

[14] Nathan S. Evans, Roger Dingledine, and Christian Grothoff. "A practical congestion attack on tor using long paths". In: *Proceedings of the 18th Conference on USENIX Security Symposium*. SSYM'09. Montreal, Canada: USENIX Association, 2009, pp. 33–50.

[15] David Ott, Christopher Peikert, and et al. "Identifying Research Challenges in Post Quantum Cryptography Migration and Cryptographic Agility". In: *CoRR* abs/1909.07353 (2019). arXiv: 1909.07353. URL: http://arxiv.org/abs/1909.07353.

[16] Khan Zeb, Owais Baig, and Muhammad Kamran Asif. "DDoS attacks and countermeasures in cyberspace". In: *2015 2nd World Symposium on Web Applications and Networking (WSWAN)*. 2015, pp. 1–6. DOI: 10.1109/WSWAN.2015.7210322.

[17] Synapse developers. *Workers - Synapse*. 2024. URL: https://element-hq.github.io/synapse/latest/workers.html (visited on 12/02/2024).

[18] The Tor Project. *Intro - The Onion Services Ecosystem*. 2024. URL: https://onionservices.torproject.org/apps/base/onionbalance/ (visited on 12/02/2024).

[19] Rob Jansen et al. "The Sniper Attack: Anonymously Deanonymizing and Disabling the Tor Network". In: *Network and Distributed System Security Symposium*. 2014. URL: https://api.semanticscholar.org/CorpusID:6697362.

[20] Kee Jefferys et al. "Loki Private transactions, decentralised communication". In: 2020. URL: https://api.semanticscholar.org/CorpusID:232884269.

[21] L. Overlier and P. Syverson. "Locating hidden servers". In: *2006 IEEE Symposium on Security and Privacy (S&P'06)*. 2006, 15 pp.–114. DOI: 10.1109/SP.2006.24.

[22] Martin R. Albrecht et al. *Practically-exploitable Cryptographic Vulnerabilities in Matrix*. Cryptology ePrint Archive, Paper 2023/485. 2023. URL: https://eprint.iacr.org/2023/485.

[23] Common Vulnerabilities and Exposures. *CVE - CVE-2021-41281*. 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-41281 (visited on 12/09/2024).

[24] Common Vulnerabilities and Exposures. *CVE - CVE-2022-31052*. 2022. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-31052 (visited on 12/09/2024).

[25] Common Vulnerabilities and Exposures. *CVE - CVE-2021-39163*. 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39163 (visited on 12/09/2024).

[26] Common Vulnerabilities and Exposures. *CVE - CVE-2021-39164*. 2021. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-39164 (visited on 12/09/2024).

[27] Common Vulnerabilities and Exposures. *CVE - CVE-2022-23597*. 2022. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23597 (visited on 12/09/2024).

[28] Common Vulnerabilities and Exposures. *CVE - CVE-2024-47771*. 2024. URL: https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-47771 (visited on 12/09/2024).

[29] Hivron Stenhav. "Security evaluation of the Matrix Server-Server API". MA thesis. KTH Royal Institute of Technology, 2023. URL: https://kth.diva-portal.org/smash/record.jsf?pid=diva2%3A1845152&dswid=5986.

Appendix A Use of AI in the Thesis

Throughout the writing of this thesis, I occasionally used ChatGPT to search for relevant academic papers and sources. According to the Guidelines, "AI tools can be used for information retrieval when the tool is reliable for searching scientific information". Having verified the sources returned by ChatGPT afterwards, I believe this usage is legitimate. Additionally, in a previous version of this thesis, I incorrectly mentioned that I used ChatGPT to "explain certain concepts". This might appear to contradict with the Guidelines, which state that "AI tools may not be used to explain topics or summarize source materials". I therefore deem it necessary to clarify that I used ChatGPT to explain technical errors that occurred when developing my proof of concept in Chapter 4. In this sense, ChatGPT was used as a debugging tool in software development. I did NOT use ChatGPT to explain or summarise any of the source materials, which would have conflicted with the learning objectives. Additionally, I hereby declare that no part of this paper was written or edited, in whole or in part, by ChatGPT or any other LLMs.