Summary

In this thesis, we develop an Instant Messaging Simulation tool (IM-sim) to investigate the Deniable Instant Messaging protocol called DenIM.

DenIM is a hybrid messaging protocol that allows for piggybacking of deniable messages on regular instant messaging traffic, all in an effort to secure any metadata which might be useful to an adversary. DenIM is proven to be formally secure through a formal analysis by the creators of the protocol at Aarhus University, though not much research has gone into testing the practicality and security in real-life scenarios.

With IM-sim, user behaviour in Instant Messaging can be simulated while capturing protocol network traffic, in which the behavioural patterns were derived by looking at current literature surrounding the topic of human online behaviour.

To facilitate this, we implemented the DenIM protocol on top of last semester's Signal implementation to allow for evaluation of the DenIM protocol.

Having all the proper tools in place, several simulations were conducted with different scenarios and varying user counts to gather as much data on the protocols as possible. The trust assumptions of the DenIM protocol were also weakened in an attempt at trying to find vulnerabilities in the security guarantees.

Some initial success was achieved when searching for deniable contacts in scenarios with few users. But by increasing the user count of the simulation, it quickly became very hard to gain any information regarding deniable contacts.

The thesis, however, does undeniably reach its goals of creating a functional DenIM implementation and a general simulation tool with rich opportunities for further improvements and development.

Yet Another Privacy Protocol End-user Reveal

Master Thesis in Distributed Systems

Andreas Knudsen Alstrup,
Arthur August Osnes Gottlieb &
Martin de Fries Justinussen

Software, cs-25-ds-10-07, June 5, 2025

Department of Computer Science
Aalborg University
http://www.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Yet Another Privacy Protocol End-user Reveal

Theme:
Distributed Systems

Project Period:
Spring Semester 2025

Project Group:
cs-25-ds-10-07

Participant(s):
Arthur August Osnes Gottlieb
Andreas Knudsen Alstrup
Martin de Fries Justinussen

Supervisor(s):
René Rydhof Hansen
Danny Bøgsted Poulsen

Code Repository:
https://github.com/Deniable-IM

Number of Pages: 41

Date of Completion:
June 5, 2025

Abstract:

Metadata privacy has become an increasingly researched topic in recent years, with little testing in real-world scenarios. We developed a generalised simulation tool for Instant Messaging user behaviour that captures network traffic. The Deniable Instant Messaging (DenIM) protocol has been implemented to challenge the protocol's trust assumption that deniable behaviour does not affect regular user behaviour. By simulating the DenIM protocol, network traffic has been captured, and novel attacks using the traffic have been designed to reveal deniable communication between users.
The attacks deal with different pruning strategies, and these are evaluated empirically with the simulated network traffic we generate.

Preface

In an ever-evolving world, privacy is a rare commodity in online communication. Thus, it is important to investigate new avenues of metadata privacy to improve the individual's privacy when using instant messaging services. These avenues are still largely unexplored, and even fewer attempts to implement metadata privacy have been made. This thesis aims to mitigate the gap by faithfully implement a simulating tool to further analyse networking traffic in instant messaging protocols. To facilitate this, the Deniable Instant Messaging protocol has been implemented, which aims to hide metadata on the transport layer.

Use of generative AI

This thesis has utilised generative AI for minor parts of the project. GitHub Copilot has been used for added reviews of some pull requests, which has helped highlight minor problems in the code, e.g. a struct field not being initialised in the simulator.

Furthermore, ChatGPT has been utilised to generate nonsensical text strings formatted as a Go array. These strings have been used as the messages sent between clients in the simulation[1].

[1] These strings can be found in `messagemaker.go` on line 3

Acronyms

E2EE *End-to-End Encryption* 1

GUID *Global Unique Identifier* 13

IM *Instant Messaging* . . . . 1, 3, 14 f., 17, 19, 22, 37

IMD *Inter-Message Delay* . . . . 17 f., 20 f., 27 f.

KDC *Key Distribution Center* 5, 38

SDA *Statistical Disclosure Attack* . . . . 3, 22, 25 f., 28 ff.

Contents

1 Introduction

Instant Messaging (IM) has become a widespread form of communication during the past few decades, with 8.3 billion accounts existing in 2021 [1]. In an effort to ensure the privacy of users, most IM apps enable message confidentiality by implementing *End-to-End Encryption* (E2EE) on their services. Protocols such as the formally secure Signal Protocol are widely used to facilitate E2EE [2]. The Signal Protocol has resilience, forward security, and future secrecy as useful properties [3], but these properties do not prevent IM apps using the Signal Protocol from leaking sensitive information to adversaries through metadata and traffic patterns [4]. The leaked information may seem harmless in itself, but it is enough to identify administrators of group chats as well as other individuals [4]. An oppressive government can identify dissenters by analysing the metadata and crack down on the dissenters. Former CIA director Michael Hayden goes even further and states “We kill people based on metadata” [5]. Metadata should therefore be as protected as the message itself, to minimise the information an adversary can obtain and analyse. Signal Foundation has attempted to mitigate the leakage by implementing functionality to anonymise senders, called sealed sender, where the identity of the sender is hidden from the server [6]. However, sealed sender has been proven to be insufficient in its privacy guarantees, which means more complex solutions are required [7].

Several proposals for achieving metadata privacy exist, but none of them have been implemented in an IM app, as metadata privacy is a somewhat recent research topic. Deniable Instant Messaging, DenIM, is one such proposal, which seeks to preserve metadata privacy on both the transport layer as well as the application layer [8]. The provable metadata privacy makes the proposal interesting to investigate, as the work assumes user behaviour is not affected by the deniable messages.

Problem statement

According to the trust assumptions in Nelson et al. (2024), the deniable behaviour of users will not influence their regular behaviour. How resilient is DenIM to a leak of metadata when this trust assumption is weakened? Neither DenIM implementation nor a dataset for DenIM exists. From this lack of available data, the following problem arises:

> *How can we develop a tool that generates realistic data for evaluating the metadata leakage in Instant Messaging protocols, by capturing all network traffic between clients and adjusting user behaviour?*

Contribution

In order to facilitate the investigation of the DenIM protocol, there is first a need for a functional implementation of the protocol, as well as plenty of DenIM user communication data to enable an analysis of its security guarantees. This leads to the contributions of this thesis, which are the following:

1. An implementation of the basic functionalities of the DenIM protocol.
2. A tool for simulating user behaviour on an instant messaging platform.
3. Attacks on the DenIM protocol under weakened trust assumptions.
4. An analysis of the attacks and an evaluation of the data collected using the simulation tool.

The remaining part of this work will be divided into the following sections:

- Chapter 2 reviews the existing literature in deniable instant messaging as well as traffic analysis of secure instant messaging.
- Chapter 3 outlines the protocol, as well as defining the adversary used in this work.
- Chapter 4 outlines the implementation of DenIM, which is based on the Signal client-server infrastructure described in Alstrup et al. (2025) [9].
- Chapter 6 describes the novel attacks we perform on the protocol, then analyses and evaluates the data gathered from the attacks.
- Chapter 7 discusses the limitations and future work.
- Chapter 8 concludes the work.

2 Related Work

Several methods of traffic analysis for IM services can be found in existing literature. Bahramali et al. (2020) describe several algorithms which can be utilised to identify users communicating with each other as well as identify administrators of group chats [4]. Martiny et al. (2021) show that the stated privacy of Signal Foundation's sealed sender feature can be compromised by a global passive adversary using a *Statistical Disclosure Attack* (SDA) [7].

The field of deniable communication contains several other proposals. Chakraborti et al. (2023) propose a framework called Wink, which enables deniable communication on compromised devices [10]. They accomplish this by utilising Trusted Execution Environments to execute all encryption schemes. Wink is limited by the requirement of regular communication with the recipient of a deniable message, which breaks anonymity.

User behaviour on instant messaging platforms is an essential part of being able to properly simulate communications, and different models have been presented to most accurately depict real user behaviour. Cui et al. (2018) try to model human online behaviour through the inter-event time between two consecutive visits to instant messaging platforms [11].

3 Preliminaries

Some preliminary information is necessary, as it will be used throughout the rest of the thesis, and therefore, a basic understanding of these principles is vital before getting into the specific implementation details.

3.1 Deniable Instant Messaging (DenIM)

This thesis focuses on analysing the deniable protocol, DenIM, as presented in the paper by Nelson et al. (2024) [8]. DenIM uses a hybrid messaging model, which means it supports both regular instant messaging and deniable messaging. This hybrid model is essential for the functionality of the protocol, as the deniable part is dependent on regular traffic in order to be delivered.

When a user sends a deniable message, it gets put into a buffer waiting to be sent. Then, once the client sends a regular message, part of the deniable message gets chunked up and piggybacked inside the regular message. The size of the deniable part is decided based on a global parameter q, such that its length will be q times the length of the regular message.

Once the regular message reaches the server, it will unpack the deniable chunks and keep them in a local incoming buffer until the entire deniable message has been received and reconstructed. When a deniable message is reconstructed, it is then added to an outgoing deniable buffer for the intended receiver. The server will then chunk up any outgoing deniable message to the receiver and add it to the regular message, which is then finally forwarded to the receiver. The receiving user then unpacks the chunks from the regular messages and reconstructs the deniable message.

If any of the outgoing deniable buffers are empty when a message is being sent either from the client or from the server, the regular message will instead be padded with dummy chunks, such that the size, depending on the q value, always is consistent. The dummy chunks are simply discarded when received.

3.2 Threat Model

As this thesis focuses on the analysis of the deniable instant messaging protocol, DenIM, we use the same threat model described in Nelson et al. (2024) [8]. They use a global active adversary, in which the adversary may participate in the protocol and observe all network traffic between the clients and server. Looking at their specific trust assumption in our analysis allows us to loosen or remove some of the assumptions, and then test the rigidity of their protocol. We keep the following trust assumptions from Nelson et al. (2024) [8]:

- The internal state of honest participants cannot be accessed.
- Recipients of deniable traffic are trusted.
- The servers handling delivery of keys and messages are trusted.
- The *Key Distribution Center* (KDC) is trusted.
- Honest users do not interact with adversaries over the protocol

It is important to note that the trust assumption regarding user behaviour has been removed in this work. This means a user can exhibit different behaviour when sending deniable traffic, such as increasing the volume of messages a user sends to piggyback deniable messages.

4 | Deniable Instant Messaging

In this chapter, the implementation details of Deniable Instant Messaging (DenIM) are explained, and it is further detailed how DenIM is slightly extended to address an issue when chunks are delivered out-of-order.

The DenIM protocol is built upon a prior implementation of Signal in Alstrup et al. (2025), which is a derived implementation from the Signal Foundations implementation that mimics the architectural choices and behaviour of one-to-one communication in the Signal Protocol [9]). As messages are cached using Redis in the signal server, DenIM is implemented by extending this caching infrastructure for fast, scalable and reliable deniable storage [12].

4.1 Key Request

Upon initialising a new session with a contact, a key request is first sent to the server, asking for a key bundle to enable the initialisation of the ratchets. In a regular non-deniable Signal session, this step will be completed shortly before sending the first encrypted message. However, in a deniable conversation, the key request will be done deniably, and therefore, there is no time guarantee for the response, which makes it impossible to encrypt the outgoing message immediately on the client, because that would require the key bundle. Instead, the client will locally store the deniable message awaiting encryption in plaintext, along with an indication that a key request has been sent. On every subsequent deniable message sent to the same contact, the client will then know not to send a new key request and instead just store the plaintext message. When the client then eventually receives the key bundle response from the server, it will start to encrypt all the stored plaintext messages for that user, and ready them to be sent deniably to the recipient.

4.1.1 Contact Discovery

Almost all activity in Signal and DenIM requires the *service ID* of a contact. Normally, the process of getting a *service ID* from a phone number in Signal follows a complicated process to try to keep the user's social graph as private as possible, and not give the Signal server any unnecessary data. In Signal, this is achieved by running code in an Intel SGX secure enclave, which supports remote attestation and hides memory access patterns [13]. Such a solution was deemed to be out of scope, and we instead opted for a simple new endpoint on the server side, which takes the phone number of a user and returns the *service ID*.

4.2 Deniable Buffers

In DenIM, the server is responsible for storing and forwarding deniable messages. To achieve this, an *incoming chunk buffer* and an *outgoing payload buffer* have been implemented to store deniable data before it gets piggybacked on a regular message. The *incoming chunk buffer* will store payload chunks from a sending client until these chunks can make a deniable payload, which is then stored in the *outgoing payload buffer*. When a client is about to receive a regular message, data from the *outgoing payload buffer* is piggybacked as chunks on that regular message as described in section 4.4.

As seen in Figure 4.1, when the server receives a regular message from a client, all the piggybacked chunks in that message get extracted and stored in the *incoming chunk buffer* for that sending client. As the sending client continuously sends regular messages, the *incoming chunk buffer* will grow as piggybacked chunks get stored in the buffer. Once the server receives a final chunk, the *incoming chunk buffer* is flushed and deserialised into a deniable payload. Given the type or the content of the deniable payload, it is decided which client should receive this payload. The payload is then stored in a client's respective *outgoing payload buffer*. From the *outgoing payload buffer*, data will be dequeued and piggybacked to deliver a payload in chunks.

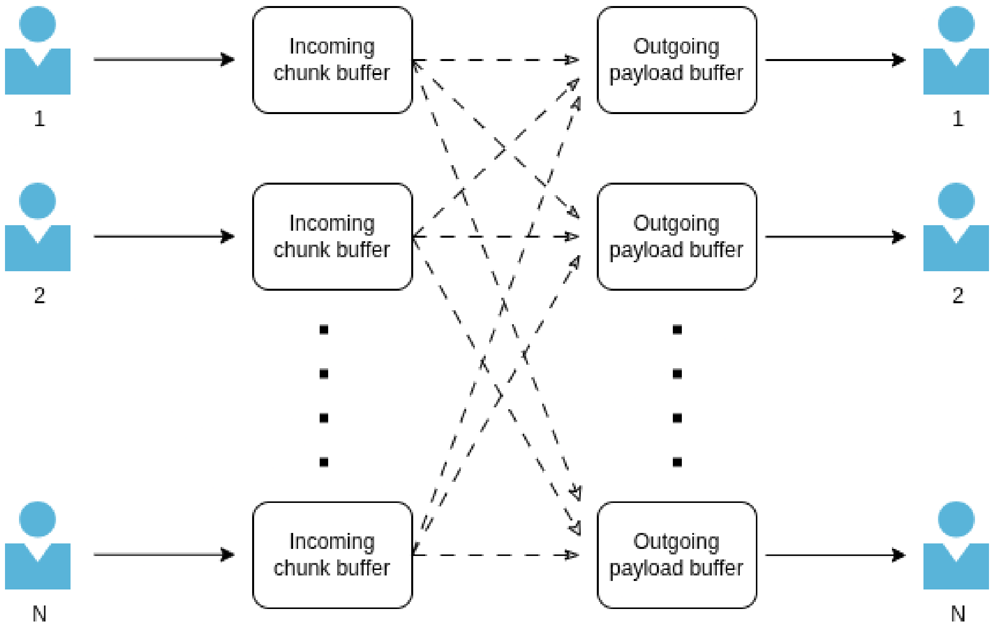

Figure 4.1: Illustration of the relations between clients and the deniable buffers, where incoming chunks get deserialised to payloads to then be piggybacked on regular messages on delivery to a client.

4.3 Chunk Sizing

The original DenIM implementation from Nelson et al. (2024) is done in TypeScript and uses protobuf to create the different message types [8]. DenIM's specifications show that the deniable part of a message is always supposed to be a ratio of the regular payload. Protobuf uses variable-width integers, which makes for clever encoding, but causes problems when the goal is to have specific message lengths, as with DenIM. A lot of work goes into getting around this encoding in the original implementation, such as creating two ballast integers, which can be manually manipulated to create the wanted length.

In our implementation, we instead rely on Rust structs and the serialisation done using the *bincode* crate. This approach simplifies the padding of the messages significantly, as integers have constant length, and the only variables needed for the calculations are the overheads for creating new structs and vectors.

This is perhaps best shown by taking a look at the code for creating chunks on the client side, as seen below in Listing 4.1. It takes the trait `DeniableSendingBuffer` as input, which is implemented directly with the underlying SQLite database to get deniable payloads. It then calculates the total size of free bytes using the regular payload size and the q value. In the actual chunking, two constants are used to account for the overheads of vectors and structs, they are `EMPTY_VEC_SIZE` and `EMPTY_-`

`DENIMCHUNK_SIZE`. The first is to account for the overhead of the vector containing all the DenIM chunks. The second is for the overhead of creating a chunk struct, so it scales linearly along with the amount of chunks being sent.

Once the space has been calculated, any outgoing deniable payload is retrieved from the database (line 11). If no deniable payload is found, it means that it will just create a dummy chunk (line 39). Otherwise, a deniable payload was found, and then it will determine if the entire payload can fit in the available space. If it cannot, then it will create a chunk to fill the available space (line 25), and then put the rest of the payload back in the database (line 31). If the whole payload fits, it will put it into a final chunk and remove it completely from the database (lines 18-23), and then start over by getting the next deniable payload to fill up any remaining space.

Finally, the function will return all the chunks created, along with any unused free space, that will then get added to a ballast field, to ensure that the sizing is consistent and always uses all available space.

```rust
pub async fn create_chunks<T: DeniableSendingBuffer>(&self,
    regular_payload_size: f32, buffer: &mut T,
) -> Result<(Vec<DenimChunk>, usize), String> {
    let mut outgoing_chunks: Vec<DenimChunk> = vec![];
    let total_free_space =
        (regular_payload_size * self.q_value).ceil() as usize;

    let mut free_space = total_free_space - constants::EMPTY_VEC_SIZE;
    while free_space >= constants::EMPTY_DENIMCHUNK_SIZE {
        let chunk_size = free_space - constants::EMPTY_DENIMCHUNK_SIZE;
        let current_outgoing_message =
            buffer.get_outgoing_message().await.unwrap_or_default();

        let new_chunk;
        if !current_outgoing_message.1.is_empty() && chunk_size != 0 {
            //Deniable
            if current_outgoing_message.1.len() <= chunk_size {
                new_chunk = DenimChunk {
                    chunk: current_outgoing_message.1.to_vec(),
                    flags: ChunkType::Final.into(),
                };
                buffer.remove_outgoing_message(current_outgoing_message.0)
                    .await.map_err(|err| format!("{err}"))?;
            } else {
                new_chunk = DenimChunk {
                    chunk: current_outgoing_message.1[..chunk_size].to_vec(),
                    flags: ChunkType::Data(current_outgoing_message.2).into(),
                };
                let remaining_current_outgoing_message =
                    current_outgoing_message.1[chunk_size..].to_vec();
                buffer.set_outgoing_message(
                        Some(current_outgoing_message.0),
                        current_outgoing_message.2 - 1,
                        remaining_current_outgoing_message,
                    ).await.map_err(|err| format!("{err}"))?;
            }
        } else {
            //Dummy
            new_chunk = DenimChunk {
                chunk: vec![0; chunk_size],
                flags: ChunkType::Dummy.into(),
            };
        }
        outgoing_chunks.push(new_chunk);
        free_space =
            total_free_space - serialize(&outgoing_chunks).unwrap().len();
    }
    Ok((outgoing_chunks, free_space))
}
```

Listing 4.1: Client side chunking of deniable payloads when sending a message.

4.4 Chunk Ordering

Every piggybacked chunk has a flags field, containing some metadata about the chunk. In the DenIM implementation found in Nelson et al. (2024), the flags are only used to indicate whether a chunk is a *dummy chunk*, a *deniable payload chunk*, or the *final chunk* of a payload [8]. This means that all deniable chunks are unordered, and therefore vulnerable to deserialisation errors if the chunks are delivered out-of-order.

A simple mitigation was implemented, where every non-final chunk would get a descending number starting from zero and going negative. When the final chunk is then received, all prior chunks are sorted in descending order and finally appended with the *final chunk*. This solution mitigates most simple out-of-order problems, but still has problems if the *final chunk* is out-of-order.

When the *final chunk* is received on the server, it can sort the incoming buffer. The payload is then enqueued in cache, and the server should be able to dequeue a desired amount of payload data from the cache to piggyback on a message as one or more chunks. Listing 4.2 shows the implementation details for when payload data gets dequeued, which later will be used to create chunks.

```
1  pub async fn dequeue_bytes(
2      mut connection: Connection,
3      queue_key: String,
4      queue_metadata_key: String,
5      queue_total_index_key: String,
6      queue_lock_key: String,
7      bytes_amount: usize,
8  ) -> Result<(Vec<u8>, usize, i32)> {
9      // Return early when buffer is empty
10     let first = match get_first(// ..
11 
12     let field_id = get_field_metadata(&first)?;
13     let mut value = Bytes::decode(vec![first.clone()])? // ..
14 
15     // Get some data from first value and remove
16     if bytes_amount < value.len() {
17         let rest = value.split_off(bytes_amount);
18         let (updated, order) = update_value(&mut connection, &queue_key, field_id, rest).await?;
19         if !updated {
20             return Err(anyhow!("Failed to update value."));
21         }
22         return Ok((value.clone(), value.len(), ChunkType::Data(order).into()));
23     // Get whole of first value and remove
24     } else {
25         // Get guid for proper removal
26         let field_guid: Option<String> = cmd("HGET")
27             .arg(format! {"{}:rev", &queue_metadata_key})
28             .arg(&field_id)
29             .query_async(&mut connection)
30             .await?;
31         // Delete and retrieve
32         if let Some(guid) = field_guid.clone() {
33             let removed: Vec<Vec<u8>> = remove(vec![guid], // ..
34             let value = removed.into_iter().flatten().collect::<Vec<u8>>();
35             return Ok((value.clone(), value.len(), ChunkType::Final.into()));
36         }
37         return Err(anyhow!("Failed to take values: field guid not found."));
38     }
39 }
```

Listing 4.2: Function responsible for dequeuing data from the set of payloads stored in cache. Some error-handling logic has been omitted to make the code snippet more concise.

Payloads are enqueued in the cache, but *payload data* should be dequeued from the set of all payloads belonging to the receiving user. The `dequeue_bytes()` function in Listing 4.2 implements this by getting the first payload from the cache and returning early if no payload is found in the cache; this would indicate that a *dummy chunk* should be used (line 10). When a payload is found, the ID is extracted, which is stored alongside the payload data as Base64 and the payload's current chunk order (line 12). A payload entry is therefore a string in the format `{id}:{Base64}:{ChunkType}` where the metadata surrounding the Base64 encoded payload is maintained as the payload data gets dequeued.

The Base64 encoded payload gets decoded to a vector of bytes (line 13), and if the desired amount of data to take from the payload is lower than the total amount of data, then the remaining amount is returned, which is used to update the payload and advance the order (line 16-22).

A payload's data will therefore shrink as data gets taken and updated in the cache. Then, when the amount of data to take is greater than or equal to the payload data, a lookup is made to get the *Global Unique Identifier* (GUID) of the payload (line 26-30). The Signal implementation uses the GUID to store values' expiration time and ID counter in a metadata queue. A removal of an entry requires this GUID to properly remove the value and its metadata. Since an envelope in the Signal implementation contains a GUID (`server_guid`) for insertion and removal in the cache, another way was needed in DenIM to get the GUID of a payload or a chunk, while reusing the logic for envelope insertion and removal [6]. To achieve this, a reverse lookup queue is maintained when inserting and removing an entry in the cache. The reverse lookup queue is a metadata queue that is used to map a payload's ID to its GUID. On line 33 the looked-up GUID is used to remove a payload from cache by using the same `remove` function that removes chunks and envelopes. This results in the return of the removed value, together with its size and marked as *final chunk* since it's the last data taken from a payload (line 35).

When the server sends a regular message to a client, the amount of bytes of piggy-backed data is relative to the size of the regular message and the callee of `dequeue_bytes` can thereby request payload data given the regular message size.

5 Instant Messaging Simulation

In this chapter, the DenIM-sim architecture is presented using our developed IM simulation tool. Using the tool allows the simulation of user behaviour in IM protocols to generate network traffic, and the implementation to facilitate this is detailed.

5.1 Architecture

The Instant Messaging Simulation (IM-sim) is a tool that uses the Docker SDK to manage and control the communication between multiple containerised instant messaging clients [14]. Figure 5.1 shows the architecture of a DenIM simulation (DenIM-sim), where the server and all the clients communicate on their own isolated network for the ease of capturing the network traffic related to the protocol.

The architecture of an instant messaging protocol can vary quite differently, and Figure 5.1 is just one such configuration that can be simulated using the tool. Many different IM architectures can be represented using the tool while supporting all the network configurations that exist for Docker [15]. Furthermore, processes can be started in each client to control the behaviour of each client, and while the simulation is running, all the network traffic will be captured.

5.1.1 Containerisation

Docker is used to containerise any component necessary for simulating communication between clients in an instant messaging network. For DenIM that means a Dockerfile for the server, database, cache and client respectively. Then, IM-sim uses the Docker SDK to build containers using these Dockerfiles and sets up the network configurations, whereafter the simulation can be conducted, in which IM-sim controls how containers need to communicate.

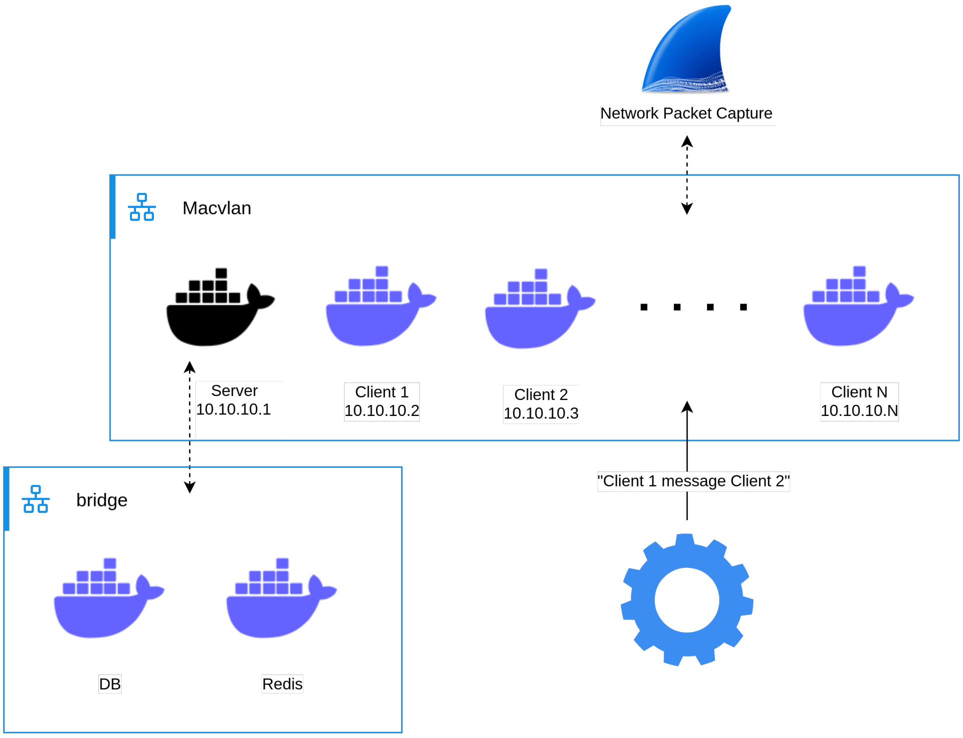

Figure 5.1: An illustration of the DenIM-sim architecture used to capture network traffic to later be used for traffic analysis.

5.1.2 Packet Capture

The goal of IM-sim is to capture instant messaging traffic to later analyse, and it is therefore important to differentiate client traffic in a realistic way. For DenIM-sim the server and clients are all running on a MacVlan network driver to ensure that the clients have a unique IPv4 address and MAC address. This makes the server and clients' network appear to be a physical network interface that is directly connected to the physical network [16]. This allows IM-sim to use Wireshark (Tshark) to capture all the network traffic on that specific network interface to then only capture what is happening between the server and the clients for the duration of the simulation [17].

5.2 Implementation

The implementation choices for scaling clients and facilitating custom user behaviour in IM protocols are explained in this section. Furthermore, to simulate traffic between users on a single host machine, it is needed to faithfully represent users in an IM network, where each user is isolated from each other on the transport layer. The Docker SDK is used for the isolation of users, where orchestrating user communication is the main responsibility of IM-sim. This allows IM-sim to generate network traffic for DenIM to later analyse in chapter 6.

5.2.1 Scaling Clients

Since each client container uses the same Docker image, that image should scale to multiple individual client containers. As a result of implementing the creation of one container, creating multiple containers are implemented by looping N times to create the desired number of clients. Since creating and running multiple containers was making IM-sim quite slow to set up the containers, a worker pool pattern was implemented. This helped speed up this process by running these set-up tasks with a fixed number of concurrent tasks.

5.2.2 Client-server Communication

When all containers are running, each client needs to be able to communicate with one another. This is done using the Docker SDK to make a system call for a given client to instantiate a DenIM client. However, since each system call starts a new process for the client, this will not suffice to send messages in the context of an instantiated client. The Docker SDK does not inherently support sending commands to a running process. But since a new process returns a stream-oriented network connection, IM-sim holds that handle to the process throughout the simulation. This allows IM-sim to write to the process via the standard input stream and read from the process via the standard output stream. The result is a fine-grained control over what a given client should send to another client and the ability to log the output message when it is received by the other client.

5.2.3 IP Assignment

IM-sim supports the existing network configurations that come with Docker, which makes it possible with IM-sim to configure a network with a given subnet and let Docker handle IP assignment. When statically assigning an IP, that IP must not be in the configured IP range, as Docker can otherwise dynamically assign that IP while it's already in use. Since some services, such as the server and a handful of clients, could benefit from having a static IP to later ease the analysis of a specific pattern in network traffic, IM-sim needed to implement IP assignment.

IP assignment in IM-sim is implemented by using the IP range that is specified for a new network and then calculating the available set of IP addresses. Then, when a container is created with an assigned docker network and a specific IP, that IP is removed from the set of available addresses. When a container is created without a specific IP, it just pops an available address from the set. This implementation allows reserving IPs while having IM-sim dynamically assign IPs from the same IP range.

5.2.4 Simulate in Cloud

IM-sim supports simulations with hundreds or more clients, and these simulations can run for multiple hours. Since each container uses some RAM, a simulation can scale proportionally to the amount of RAM available. Table 5.1 shows the approximate memory usage for each container in DenIM-sim and can give an estimate of how many clients can be simulated on a given system.

Container	RAM Usage
Denim-client	≈ 6.5 MB
Denim-server	≈ 50 MB
Redis	≈ 17.3 MB
Database	≈ 502 MB

Table 5.1: The approximate memory usage for each container in DenIM-sim with 500 simulated users.

If a simulation benefits from having multiple thousands of clients, those simulations can be run on a cloud computer with more available memory. Since the Docker Engine can be configured to allow incoming connections, IM-sim therefore supports connecting to a remote host to run all the containers in the cloud. When IM-sim is specified to use a remote host, the context of the host machine will be used to build the Docker images on the remote machine. After this initial process, the containers will start on the remote host, but all the control of the container will be managed by the host matching running IM-sim.

5.3 Simulating Users

Simulation of user behaviour was chosen since those can affect the trust assumptions of IM protocols, and since no real-world IM application implements the DenIM protocol. Furthermore, there are no existing datasets for this unestablished protocol. Simulation also enables data collection of multiple runs with small changes between each run. This is especially useful when finding limits of what is possible to detect with the weakened trust assumptions.

Modelling user behaviour is a key part of creating a simulation of IM clients. Simulated users should exhibit behaviour which is as close as possible to real users. Existing literature utilises *Inter-Message Delay* (IMD) as a feature, which various models attempt to estimate as a probability distribution. IMD is defined as the delay between any two messages being sent from a given user. Bahramali et al. (2020) use an exponential distribution to estimate the IMD using Maximum Likelihood Estimator to find the rate parameter [4]. Figure 5.2 shows an example of IMD of real-life data.

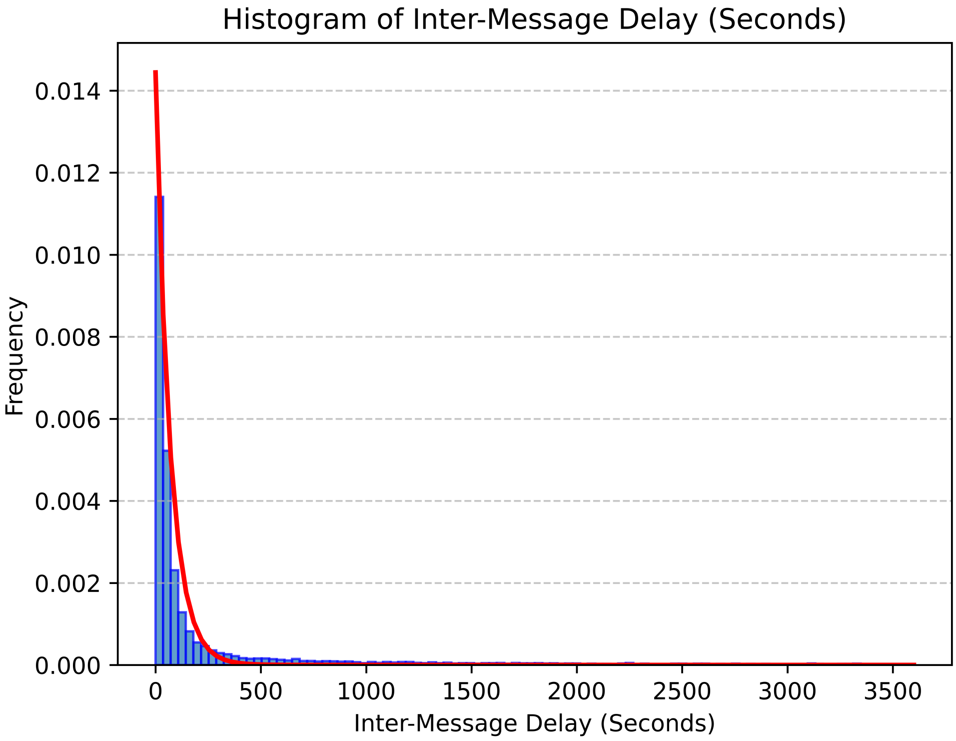

Figure 5.2: Histogram of the IMD of one of the authors' usage of Discord totalling around 30000 messages over 5 years. Group chats and servers are excluded from this dataset. The data was obtained by requesting all the author's data Discord stored.

Cui et al. (2018) classify existing models of user behaviour into three categories: models based on queue theory, models based on human properties such as memory, interest, etc., and models based social interactions [11]. All three categories of models attempt to describe the IMD.

This thesis uses a mix of the current literature to simulate users such that the IMD of any user follows a distribution resembling an exponential distribution, while using a simplified model of human behaviour to create the messages. Each user is assigned individual probabilities of starting new regular and deniable message chains, and replying to existing message chains. In our implementation, the time between messages is a uniform distribution between 0 seconds and an arbitrary upper limit. The time between a user receiving a message and sending a reply follows the same distribution, but with the restriction that the latest time a reply can be sent is the time the next regular message is sent at. This assumption does not reflect the real-world behaviour of users perfectly, but makes the IMD closely follow an exponential distribution using only uniform distributions.

5.3.1 User Behaviour

The simulator is implemented as a dispatch server similar to the approach of Nelson et al. (2024) [8]. This approach separates the implementation of the IM clients and server from the simulator, which handles the behaviour of individual users and determines when a given IM client sends a message.

The individual behaviour of clients can be implemented through a Go interface. This ensures the simulation can easily switch to different types of behaviours, as well as change messaging protocols without changing the code in the simulation itself. The interface requires a behaviour struct to implement functions for calculating time between messages, creating all types of messages supported in the protocol, as well as handling all logic concerning replies and sending bursts of messages. All required functions can be seen in Listing 5.1. For the purposes outlined in this thesis, a single implementation is used for all simulations, which can easily be configured for the relevant protocol experiments.

```go
package Behavior

import (
	Types "deniable-im/im-sim/pkg/simulation/types"
)

type Behavior interface {
	GetNextMessageTime() int
	WillRespond(Types.Msg) bool
	GetResponseTime() int
	IsBursting() bool
	MakeMessages() []Types.Msg
	MakeReply(Types.Msg) Types.Msg
	ParseIncoming(string) (*Types.Msg, error)
}
```

Listing 5.1: The Behaviour interface used for handling message timings, parsing incoming messages as well as creating valid messages for the chosen protocol.

The simulation is made as generalised as possible to allow other implementations of protocols and behaviour to be tested without rewriting the simulation itself. It still requires the clients to be containerised, as the simulator relies on the use of writing to `std-in` for all types of communication. The simulator itself does not need to know any specifics about the message commands required by each client implementation, as that responsibility is delegated to the behaviour structs. The responsibility of the simulation itself can be split into the following main functionalities: Sending messages to an IM client through its `std-in`, sleeping the specified time between messages, passing incoming text from `std-out` to the behaviour struct for parsing, logging send and receive events, as well as sending responses after the specified sleep time. This can be seen in the functionality shown in Listing 5.2.

```
func (su *SimulatedUser) OnReceive(msg Types.Msg) {
	if su == nil {
		return
	}

	//Determine if Alice responds to the message
	if !su.Behavior.WillRespond(msg) {
		return
	}

	res := su.Behavior.MakeReply(msg)
	sleep_time := su.Behavior.GetResponseTime()
	time.Sleep(time.Duration(sleep_time * int(time.Millisecond)))

	su.SendMessage(res)
}
```

Listing 5.2: The function in the simulation which is responsible for sending replies if the behaviour determines it should happen.

Parameter Tuning

A fuzzing library for Go called gofuzz accelerated the choice of parameters [18]. The library takes an arbitrary Go struct and populates all public fields with random data recursively, with the possibility of customising the random data to better suit any specific use case. The only customisation for gofuzz was setting the probability of a value being `nil` to 0, meaning all public fields in the passed structs will have some value. The use of parameter sweeping with gofuzz allowed for a quick and coarse filtering of parameters, which ensured that most time was spent investigating configurations that were likely to produce desirable results.

The results of the parameter sweeps showed that most simulated users with a suitable IMD distribution had a lower probability of sending a message than the probability of replying to a message. Further testing showed these observations to be correct.

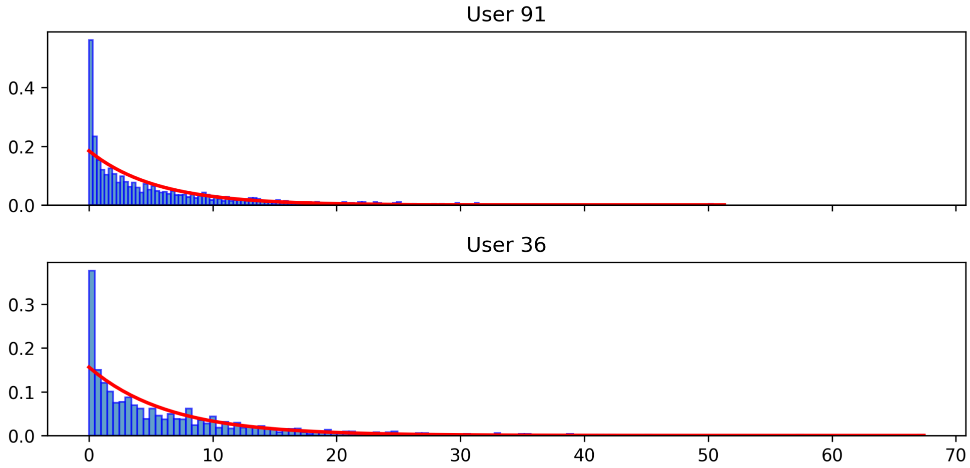

Figure 5.3: Histogram of the IMD of two simulated users.

The selection of suitable parameters took multiple iterations, as the IMD of a given user is dependent on said user's contacts. For example, Bob might have a low probability of responding to each incoming message, but with enough contacts sending messages quickly, Bob will eventually send multiple replies in a short time frame.

Functions were added to generate users easily for the type of behaviour modelled in this thesis. One function takes a struct with all the necessary options to make a customised array of simulated users. A default option exists, which generates random values within ranges we have determined as suitable. The default generation is also used as a fallback in case of encountering `nil` fields when using the options struct. The chosen ranges of the default generation are the ones used to create data for the rest of this thesis.

6 Traffic Analysis

The goal of this chapter is to identify communication between two users, even if they only use deniable communication. This is accomplished by leveraging *Statistical Disclosure Attack*s (SDAs) found in current literature and adapting them to the limitations of the DenIM protocol. This chapter first explores attacks on regular contacts, which is a well-defined problem within the literature of SDAs. Afterwards, attacks are designed to specifically compromise users communicating deniably within the threat model specified in section 3.2.

6.1 Experimental Setup

The simulator and DenIM implementation described in this thesis are used in this chapter to conduct all experiments.

Each experiment has two variations: one where users change behaviour when sending deniable messages, and one where a user never changes behaviour during a simulation. Both variations have been simulated with 10, 100, and 500 users in the IM network. More specific scenarios have also been created to test the anonymity of the DenIM protocol under certain circumstances.

Table 6.1 shows the parameters used when running the simulation.

Parameter	Min	Max
Time Between Send attempts (ms)	0	10000
P(Send message)	0.231	0.431
P(Reply to message)	0.530	0.730
P(Deniable message)	0.1	0.1
Burst length (no. message)	5	5
Burst rate	0.1	0.1

Table 6.1: The configuration of behaviour for all clients when simulating. Some parameters are kept constant, whereas others vary between runs.

The time between attempted send events is randomised between each event, but is at most 10000 ms. This does not guarantee a message being sent after 10000 ms, as that entirely depends on the probability of sending a message. Therefore, only the probabilities of sending and replying to messages vary, as these reflect the uniqueness of users and add necessary variation in traffic to test attacks. The probability of sending a deniable message is kept as a constant, as we attempt to keep it at a ratio of 10 regular messages for each deniable message in line with the findings of Nelson et al. (2024) [8].

Bursting is a consequence of weakening the trust assumptions and is therefore kept as a constant to make the attacks easier to construct and verify. A burst consists of 5 messages sent within a short time interval of each other. The burst time interval is defined as 0.1 times the normal time between send events, with a user always sending messages when bursting.

The simulated traffic has primarily been run on a laptop with a Ryzen 7 PRO 5850U CPU and 32 GB RAM. Due to time constraints, other machines have been utilised to generate some of the data.

6.2 Identifying Regular Contacts

Several approaches to the identification of regular contacts have been implemented. The attacks utilise different methods to identify contacts, but with the shared goal of identifying the most likely contacts of a chosen user. A recurring challenge for all algorithms described in this section is that the attacks rank the likelihood of all other users being a contact, but they do not provide a way to filter the contacts from non-contacts. We outline an approach to find a subset of the contacts which will help prune the list of suspected deniable contacts.

6.2.1 Counting-Based Statistical Disclosure Attack

This attack is based on the work of Martiny et al. (2021) [7]. The article identifies the sender of Signal messages, even when Signal's sealed sender is used to obscure the sender. Signal still sends delivery receipts when a sealed sender message is received, which can be exploited by an adversary. Bob's contacts are identified by monitoring the outgoing network traffic in a time frame just after Bob has received a message. The likely contacts sending Bob a message are more likely to appear in this time frame than in any randomly sampled time frame [7].

Nelson et al. (2024) prevent a one-to-one copy of this attack as their article states that delivery receipts are disabled in the DenIM protocol [8]. The attack is thus repurposed to suit the changed features in network traffic, which can be exploited. The main idea behind the attack remains, as Bob and his contacts will still be communicating, with contacts being more likely to send messages in a time frame just after Bob has sent a message. The pseudocode for the attack can be found in Algorithm 1.

Algorithm 1 Counting-Based Statistical Disclosure Attack

Input: The network capture *input* as table, number of samples *rounds*, window size w, IP address of target user *target_ip*, server IP *server_IP*

Output: a list of users from most likely to least likely contact

```
procedure COUNTING_SDA(input, rounds, target_ip, server_ip)
    suspects ← Zero initialised dictionary
    t_target ← rows in input where Source = target_ip
    input ← rows in input where Source ≠ server_ip
    for 1 to rounds do
        r_target ← random row in t_target
        t_start ← r_target.Time
        t_end ← t_start + w
        target_epoch ← rows in input where Time ∈ (t_start, t_end]
        for all x in target_epoch do
            if x.Source = target_ip then
                continue
            suspects[x] ← suspects[x] + 1

        r_random ← random row in input
        r_start ← r_random.Time
        r_end ← r_start + w
        random_epoch ← rows in input where Time ∈ (r_start, r_end]
        for all x in random_epoch do
            if x.Source = target_ip then
                continue
            suspects[x] ← suspects[x] − 1

    return suspects
```

Martiny et al. (2021) show that the attack works best for associates who have few other contacts than Bob, and when Bob generally appears in few other conversations according to Theorem 1 and Corollary 2 [7].

The attack works for regular traffic in the DenIM implementation. If a chosen contact has a large number of contacts, the number of rounds must be increased to increase the probability that an actual contact is identified as the most likely contact. Similarly, the window size must be adjusted to include any response, but exclude as much other traffic as possible. We found that $rounds = 1000$ and a window size of 1 second returns the majority of regular contacts consistently when using simulated network traffic with the configurations used in this chapter.

6.2.2 Normalised Statistical Disclosure Attack

The Normalised Statistical Disclosure Attack was created by Troncosco et al. (2008) as an improvement to the original SDA while being more performant, but less accurate than the Perfect Matching Disclosure Attack [19].

The attack works by iterating over all packets sent in the network, sorted by time in ascending order. Every time a packet is sent to the server, the source IP is added to the list of senders along with the time window it should exist in the list. When the server sends a packet to a client, all other clients will be assigned probabilities according to how many times they have sent packets to the server during the time window. The probabilities of all senders are made into a probability matrix, which can then be normalised using the Sinkhorn-Knopp algorithm [20]. This attack is adapted from the implementation in Kristensen et al. (2025) but with some changes [21]. The most notable change is the function assigning probabilities to all senders is changed, such that the probabilities go from being $\frac{1}{|senders|}$ to $\frac{1}{|senders|-k}$ where k is the number of times the receiver of a package appears in the senders list. Algorithm 2 shows the pseudocode of the attack with Algorithm 3 and Algorithm 4 as helper functions.

Algorithm 2 Normalised Statistical Disclosure Attack

Input: The captured TLS packets *input* as table, window size w, server IP $server\_IP$.
Output: a probability matrix of the most likely associates of all users.

procedure NORMALISED_SDA($input, w, server\_ip$)
  $senders \leftarrow []$
  $previous\_time \leftarrow input[0].Time$
  $receivers \leftarrow dict(dict())$

  for all $row \in table$ **do**
    $senders \leftarrow$ update_senders($senders, row.Time - prev\_time$)
    if row.Destination == server **then**
      $senders \leftarrow senders \cup (row.Source, w)$
    else
      $receivers \leftarrow$ update_receivers($senders, row.Destination, receivers$)
    $previous\_time \leftarrow row.Time$

  $matrix \leftarrow matrix\_from\_dict(senders)$
  $result \leftarrow Sinkhorn(matrix)$
  return *result*

This attack is consistent in its results as it does not rely on randomly sampling the dataset, but instead iterates over all traffic. The attack correctly identifies one of the regular contacts as the most likely contact. However, not all regular contacts are marked correctly as likely suspects, because it depends on the two users' interactions during the simulation.

Algorithm 3 Update Senders

Input: a list of sender tuples *senders* and time delta since last update *time*.
Output: an updated list of active senders.

procedure UPDATE_SENDERS($senders$, $delta$)
  $updated \leftarrow []$
  for all $sender$, $time \in senders$ **do**
    $remaining \leftarrow time - delta$
    if remaining > 0 **then**
      $updated \leftarrow updated \cup (sender, remaining)$
  return *updated*

Algorithm 4 Update Receivers

Input: a list of sender tuples *senders*, the IP address of the receiver, *receiver*, and the nested dictionary of possible contacts, *contacts*.
Output: an updated list of active senders.

procedure UPDATE_SENDERS($senders$, $receiver$, $contacts$)
  $k \leftarrow 0$
  for all $sender$, $time \in senders$ **do**
    if $sender = receiver$ **then**
      $k \leftarrow k + 1$
  for all $sender$, $time \in senders$ **do**
    if $sender = receiver$ **then**
      continue
    if $sender \notin contacts$ **then**
      $contacts[sender] \leftarrow dict()$
    if $receiver \notin contacts[sender]$ **then**
      $contacts[sender][receiver] \leftarrow 0$
    $contacts[sender][receiver] \leftarrow contacts[sender][receiver] + \frac{1}{|senders| - k}$
  return *updated*

6.2.3 Generating List of Contacts

The *Statistical Disclosure Attack*s (SDAs) in this section may identify different users as the most likely contact. This is not an issue, as each user can have multiple contacts, and each algorithm may rank any of the contacts as the most likely. The contact list of any user is estimated as the intersection of the N most likely contacts from specified attacks. This way, more contacts than the most likely contact can be found from the attacks, with the drawback that an actual contact might be dropped from the final list, if any of the chosen attacks do not rank it among the N most likely. Another risk is getting the empty set if the attacks return disjoint sets of users. This method does not identify the complete list of contacts, but it helps approximate the list while minimising the likelihood of false positives.

6.3 Identifying Deniable Contacts

The attacks used to identify any given user's deniable contacts are based on some additional assumptions. All attacks are based on the following assumptions:

- All users have completely disjoint sets of contacts for regular and deniable messages.
- A user will respond with the same type of message as the received message, i.e. a user will reply deniably to a deniable message.
- A user is allowed to send a burst of messages just after a deniable message is created.

This section will create and evaluate attacks on the DenIM protocol under these assumptions with a varying number of users. Figure 6.1 shows the initial setup used. It contains two separate sets of users, where the only interaction between the sets happens through deniable messaging. This setup is used in all attacks to test the viability of an attack before scaling with more users.

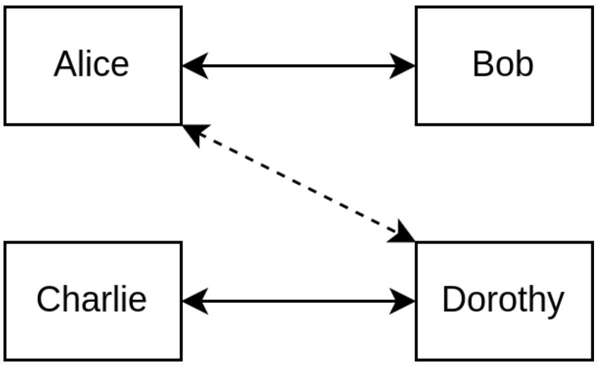

Figure 6.1: Illustration of communication between users. The solid arrows indicate normal traffic, with the dotted line arrow indicating deniable communication happening between Alice and Dorothy.

6.3.1 Identification of Deniable Behaviour

The *Inter-Message Delay* (IMD) is chosen as a feature from which deniable behaviour can be identified [4].

It is assumed that a user will send a burst of messages just after making a deniable message, with the goal of quickly getting the deniable chunks of the message sent to the server. This is a consequence of the fact that this thesis assumes a user cannot observe the state of the outgoing message buffer on the client. In case Alice needs to initialise the session with Dorothy, the burst still helps, as it carries the deniable key request to the server as well as attempting to trigger her regular contacts to send her replies, which will get the deniable key response piggybacked.

The IMD constituting a burst varies, as it depends on the IMD for regular messages. The IMD for regular messages can not be deduced from the network traffic alone, as it includes both types of messages. Some arbitrary percentile of the IMD can be used as a guideline for identifying bursts, with the ability to make some changes if the length of each burst is deemed to be too long.

6.3.2 Recipient Identification

A recipient of a deniable message can be any user receiving TLS packets after a burst has been made. The deniable message is reconstructed on the server and turned into new chunks to fit the length of any regular message sent to the recipient of the deniable message. Deniable communication follows the assumption that a user receiving a deniable message might respond to it. It is assumed that a reply to a deniable message will also be deniable, as it would otherwise disclose that a conversation between the parties is taking place.

We only consider bursting behaviour as the feature from which deniable messages can be identified in the network traffic, but it only showcases that a deniable message has been attempted to be sent. Identifying a client as the recipient of a deniable message is difficult, as a recipient can, in theory, receive an extremely long regular message, which in turn could be used to piggyback the entirety of a small deniable message. This means a simple decrease of probability can be done on non-viable suspected contacts, as a suspected contact who has not received any messages between the target bursting and the suspected contact itself bursting can be eliminated as a potential response.

Implementing SDAs for deniable messages requires a bit more effort, as the time between a sending burst event taking place and a reply being sent can be longer than for regular messages. Thus, it will take more data to ensure the most likely contact is indeed a contact.

The attacks described in subsection 6.2.1 and subsection 6.2.2 are repurposed to be used for deniable messages instead of regular messages. They follow the same base logic as their counterparts used for regular messages and assign probabilities based on identified deniable behaviour instead of regular messages. The largest change is made to Algorithm 2, as receiving deniable messages cannot be detected. Instead, probabilities are assigned when a new burst is started.

6.4 Evaluation of Attacks

The attacks used to compromise deniable contacts are evaluated in simulated runs with 4, 10, 100 and 500 users to determine the capability to handle a scaling user base. They are evaluated in four different variations to determine the effectiveness of each improvement step: the attacks in their simplest form with no pruning, pruning of all regular contacts, pruning of all bursts which have not received a regular message, and both types of pruning. The regular contact list is taken from the ground truth found in the logs generated by the simulator to ensure the attacks are evaluated in ideal conditions. The drawback is the fact that any improvements shown by pruning regular contacts may not be possible in more realistic settings, as it would require attacks on regular messages to perfectly identify all regular contacts. All data shown in this section is for attacks run with the same user as the target.

6.4.1 Whistleblower Contacts News Agency

The DenIM protocol is meant to hide metadata when sending and receiving deniable messages. One such example could be a scenario where a whistleblower wants to publish some information via a news agency.

To simulate such a scenario, 4 entities are required: the whistleblower, the news agency, a regular contact for the whistleblower and one for the news agency. This scenario was set up in IM-sim without bursting, and with bursting to challenge DenIM's trust assumption that deniable behaviour does not affect regular behaviour.

The network packets that were captured during the simulated scenario were then evaluated to identify and prune the whistleblower's regular contacts as described in section 6.2.

The attack conducted in this scenario is designed to identify bursting behaviour between the clients. Table 6.2 shows the result that when the whistleblower participates in the protocol by complying with the trust assumptions in Nelson et al. (2024), the SDA assigns the news agency a low probability of being a deniable contact. When the whistleblower's behaviour is changed when sending deniable messages, the SDA assigns a 92% probability that the whistleblower and the news agency have been exchanging messages.

Configuration	With Bursting	Without Bursting
No pruning	0.0032	0.1528
Regular contact pruning	**0.9205**	0.3808

Table 6.2: Probability of whistleblower sending a deniable message to the news agency, with and without bursting user behaviour.

To investigate whether identification of deniable behaviour with a weakened trust assumption is still valid in a network with many concurrent users participating, further simulations have been run.

6.4.2 Deniable Counting-Based Statistical Disclosure Attack

This attack is based on the SDA described in subsection 6.2.1, but modified to count based on deniable bursts in the sampled epochs instead. The results are evaluated as an average of 10 runs of the attack to minimise the effect of outliers in the sampling. Table 6.3 shows the average placement of the best-ranking deniable contact.

Configuration	10 users	100 users	500 users
No pruning	3.7	38.8	172.8
Regular contacts pruned	2.8	33.4	283.2
Infeasible contacts pruned	4.1	29.1	289.1
Both pruned	3.5	20.1	311.4

Table 6.3: Average of 10 runs with the best ranking deniable contact in the deniable normalised statistical disclosure attack. Lower ranking is better.

The pruning of regular contacts simply moves the deniable contact up a few places on the list of suspects, marking them as slightly more probable. The attacks were re-run for each pruning configuration, which means the difference in averages might be affected by the random sampling, as one would suspect the pruning of regular contacts to simply move all other contacts a few placements up. The attack does not handle the increase to 100 users well, but it still ranks deniable contacts among the $\frac{1}{3}$ most likely contacts. For 500 users, the attack fails to identify deniable contacts or limit the number of possibly deniable contacts in any meaningful way. Thus, this attack should not be used if a lot of users participate in the network concurrently.

6.4.3 Deniable Normalised Statistical Disclosure Attack

This SDA iterates through all packets passed to the attack, which only makes it necessary to run the attack once for each configuration. Table 6.4 shows the highest ranking deniable contact for each configuration.

Configuration	10 users	100 users	500 users
No pruning	2	30	179
Regular contacts pruned	1	31	164
Infeasible contacts pruned	2	28	175
Both pruned	1	29	160

Table 6.4: Placement of the highest ranking deniable contact in the proposed deniable normalised statistical disclosure attack. Lower ranking is better.

The best pruning strategy for 10 users was pruning regular contacts. The normalised probability of the top contact being the correct choice fell marginally when pruning both regular contacts and contacts which were not possible responders, but it kept the correct ranking of the deniable contact. For 100 users, the best pruning strategy was pruning suspects purely on the fact that they could not reply to a specific message. Using both pruning strategies keeps the deniable contact amongst the top $\frac{1}{3}$ of users most likely to be a deniable contact. This is still not ideal, as it would mark 159 users as more likely to be the deniable contact. The practicality of doing further investigation of $\frac{1}{3}$ of all participants in the network makes the attack infeasible at scale.

7 | Discussion

Throughout the project, several interesting discoveries were made, both in terms of the details of our implementations and in relation to our simulations and analysis. In this chapter, a closer look is taken at some of the most interesting things found.

7.1 Message Size and Session Initialisation

In deniable instant messaging, the size of the deniable part of a message is determined as a fraction of the size of the regular payload, denoted with the constant q. However, all Signal messages are not the same length, and two types are mostly used that vary a lot in size. One of them is the regular Signal message, which in our implementation uses around 300 bytes for a simple "hello" message, while the second type, the pre-key Signal message, used for initialising a session after having gotten a key bundle from the server, uses around 2500 bytes. Messages in Signal are padded to blocks of 160 bytes, so if the plaintext content is larger than 160 bytes, it will be padded to 320 bytes and so on. To start a session, one user has to receive a pre-key message and send a response. For regular messaging, this is all done rather quickly, as it just takes one message back and forth to establish the session, and start sending smaller messages from then on.

For deniable messaging, this session initialisation can be quite cumbersome. As the 2500 bytes deniable pre-key messages will most likely have to be piggybacked on the already initialised regular messages of 300 bytes, which means it will take quite a lot of regular messages to deliver the deniable message. This is made worse by the fact that every subsequent message to the same person, before they respond, will also be a pre-key message with a size of 2500 bytes. Also, before any pre-key messages can be sent, a key request to the server has to be made once, where the response also has a size of around 2500 bytes. Below is a sequence diagram showing approximately the amount of regular messages with a size of 300 bytes, it would take to initialise a deniable session with a q-value of 0.6. This is in a perfect scenario, where the fastest way to establish the session is taken.

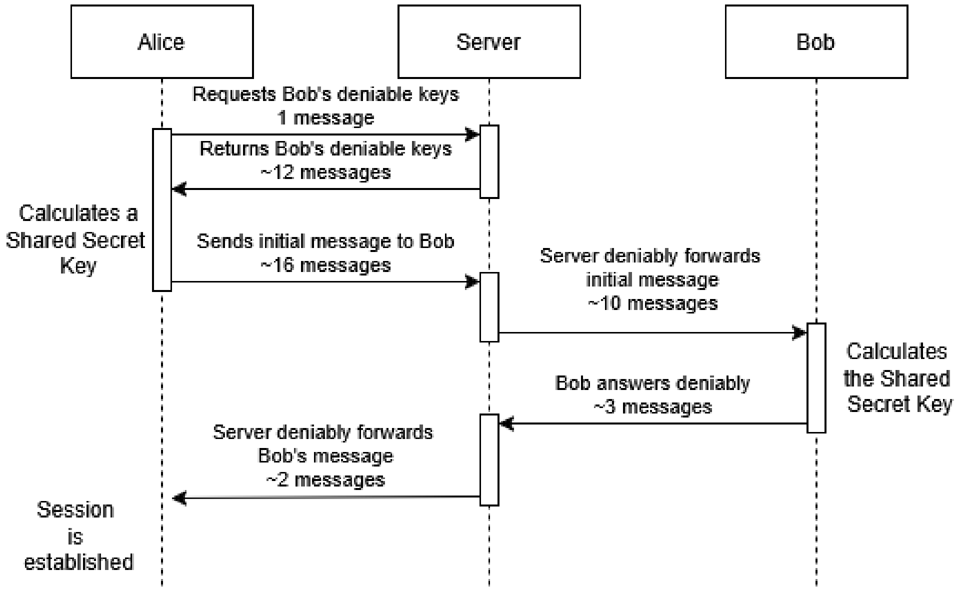

Figure 7.1: Deniable session initialisation over established normal session.

While the message size goes down significantly after initialisation, and therefore the throughput goes up, this is still quite a hefty start. It might lead to user behavioural changes, where they might spam more messages, or just start new regular sessions to have a larger deniable part. The user could even start sending regular messages to self-created users, whom never respond, so that all the messages are pre-key messages with a size of 2500 bytes, just to be able to quickly empty their local buffer of deniable payloads.

Below is the start of another sequence diagram showing the previous process, but where the sending users abuse sending regular pre-key messages to non-responding users, to empty their local buffer fast. Note that this approach will only allow an increase of payload size from the sending clients to the server, and not to the target client, as the sender has no way to control the traffic to the deniable target from the server. And we see that the initial message sending then drops from sixteen to two messages.

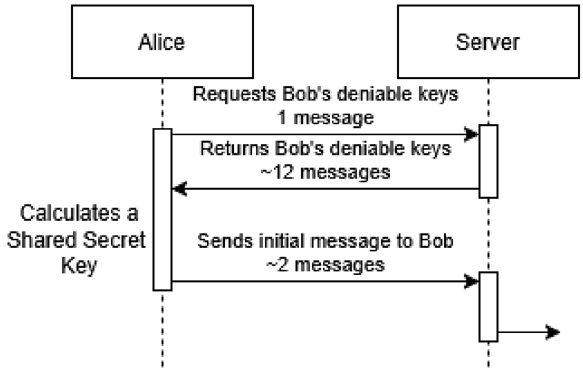

Figure 7.2: Deniable session initialisation over unestablished pre-key messages.

If the abuse of pre-key messages is extended even further to assume both sides of the communication are doing the same for all deniable messaging. Then all parts of the initialisation sequence drop to two or three messages.

7.2 Contact Discovery

In subsection 4.1.1, we described the contact discovery implementation on the sending side of a conversation, however, there are also some issues when it comes to identifying the sender on the receiving side. When sending a message, a phone number or username gets converted into a *service ID*, which the server uses to deliver a message to the correct receiver. On the other side of the interaction, the opposite problem arises of only having the *service ID* of the sender in the message. Our client needs an alias to associate with the sender's *service ID* to be able to save the user as a contact. If the user is not saved locally as a contact, then, when the receiver tries to reply, the receiver will not know how to properly find the already established session, and will instead restart the session initialisation with a new key exchange, leading to an unnecessary extra step.

In a regular Signal implementation, the username and phone number can easily be retrieved from the server by sending a request for profile information. The profile information will be encrypted, but can be decrypted using the sender's profile key, which is contained in every message.

A DenIM implementation, on the other hand, needs to be more careful with requesting data from the server, as getting profile information for a specific user could lead to possible vulnerabilities in the deniability of a conversation. There are two ways to handle this: one is to make deniable profile information requests, which would increase the number of deniable payloads, and potentially slow down the overall communication. Another solution would be to simply include the basic user information that the receiver might normally get from the profile key, directly in the message. This should not affect the deniability of the exchange, as the information would be accessible to the receiver anyway, but it does bypass the profile information request. This request stops unauthorised users from getting profile information, in case they might have gotten hold of the profile key. We decided to go for the latter approach, deeming the trade-off to be more acceptable than increasing the deniable traffic.

7.3 Server Deadlock

When the server receives a message, it is inserted in the cache, and if the receiving client is online, the message is retrieved from the cache and delivered. To accommodate this, the server maintains a `ListenerMap` to manage all currently connected clients. Then, upon receiving a message, a lookup is made to access the receiving WebSocket connection to forward the cached message.

A bug on the server from Alstrup et al. (2025) was discovered, where some clients would stop sending messages in a running simulation. Since the clients were unresponsive, it was not immediately considered that the server was the root cause, and since the behaviour was observed when simulating 100 clients, it could possibly be a problem with the simulation.

To narrow down the issue, 3 clients were set up to communicate with each other and then iteratively shorten the time between messages sent for each simulation. With a very short time between messages, the client became unresponsive, and to exclude the simulation tool as the cause, a manual test was conducted.

As a result, it was discovered that when two clients simultaneously sent a message, the server had a high chance of causing a race condition. Since upon receiving each new message, the server would lock the `ListenerMap`, but since the logic was not wrapped in *tokio::spawn* that creates a new asynchronous task, the execution could not yield back to the Tokio runtime, which in turn is responsible for scheduling pending tasks [22, 23].

This bug was not discovered last semester since it was hard to catch without scaling the number of clients, and since IM-sim is very good at scaling clients, this overlooked implementation detail in the Signal implementation was discovered when simulating a lot of clients, which stress tested the server.

7.4 Scaling Clients

Using containerisation with Docker comes with a lot of benefits when setting up client, server or other services used for an instant messaging protocol, but it becomes a resource-intensive task to run IM-sim when scaling clients. When scaling beyond

500 clients, it started to become infeasible to run a simulation on a host machine with 16 GB of RAM. Since Docker's containerisation is done by managing Linux namespaces, it was uncertain if the scaling issue could be because of Docker overhead or a limitation in Linux namespacing when running many processes.

To investigate further, a script was created to run a simple netcat echo service, in as many Linux namespaces as possible [24]. With a machine with 16 GB of RAM, it ran out of memory at around 7000 Linux namespaces. Running the same service with Docker, using IM-sim to start containers, memory ran out at around 1500 containers. This indicates that Docker has some overhead compared to just using Linux namespaces. However, the ease of setting up services for an instant messaging protocol benefits the use of IM-sim with Docker rather than implementing a Docker alternative that uses Linux namespaces to attempt to gain better performance than Docker Engine.

Another solution for scaling to thousands of clients could be to explore the clustering of clients to be run on multiple host machines as described in section 9.2.

7.5 Simulation data

The number of clients sending messages in the network determines the number of bursts required to definitively determine who communicates with whom. For a small number of participants in the network, the number of deniable messages remains relatively low, as there are fewer other participants generating noise. Any of the two attacks on deniable traffic in this thesis can reasonably determine the most likely deniable contact for a low number of users, especially when using one of the pruning strategies.

The number of participants in the network affect the number of bursts required from each user to identify deniable contacts. This means significantly more data was required to identify contacts when 100 clients participated in the network. Generally speaking, the attacks always benefit from larger amounts of data for each user and fewer participants. However, the drawback comes from the fact that each user can get away with exhibiting a larger amount of deniable behaviour when more users are participating in the network and sending their own messages at the same time. Introducing noise in the network traffic works in the deniable users' favour, which could be a suitable cover for whistleblowers and protesters. Due to time constraints, it has not been possible to identify the lower bounds of bursts needed to identify a target's deniable contacts for a given number of users in the network.

8 Conclusion

This thesis presents an implementation of a generalised Instant Messaging simulation tool with which the DenIM protocol has been simulated. A functional implementation has been made of the deniable instant messaging protocol, which is described in theory in Nelson et al. (2024), where the security guarantees are proven through formal analysis [8].

Using this simulation tool to test the security guarantees, statistical disclosure attacks were performed under the original threat model, and then again under the weakened trust assumption that users' regular behaviour may be influenced by their deniable behaviour.

To facilitate this analysis, the tool was designed to collect data by simulating user traffic based on user behaviour specified in the simulator. Several simulations were run using different scenarios with varying numbers of users to see the effects they had on the attacks performed.

The evaluation of the attacks described in this thesis showcases how other participants in the IM network affect the ability to identify a target user's deniable contacts. At 100 concurrent users sending messages, it became difficult to identify a user's deniable contacts, which means the trust assumption in Nelson et al. (2024) is not vital to prevent our attacks from working as long as enough users are participating.

While the attacks on the DenIM protocol might not have yielded any definite vulnerability results using our methodology, we were still quite successful in creating a helpful simulation tool, which allowed for quick setup and testing of different scenarios, efficient data collection and is full of potential for future improvements to further enhance the ease of use.

9 Future Work

As with any project of this size, other focus points have been left out due to time constraints. In our case, we decided to focus on developing a generalised Instant Messaging Simulation tool and implement the basic functionalities of the DenIM protocol to simulate communication and capture the network traffic. The generalised tool for generating network traffic is at the centre of our project, with section 6.4 using the captured traffic to make analysis on different simulation settings. Improvements in the DenIM implementation and optimising the simulation are therefore a focus for future work.

9.1 Key Distribution Center

One such feature was the *Key Distribution Center* (KDC), while being an essential part of DenIM as shown in the trust assumptions found from Nelson et al. (2024) [8]. The KDC only affects the key exchanges. As this thesis does not try to break the trust assumption of the KDC or try to find vulnerabilities in the key exchange, it was deemed to not be of utmost importance. However, it would be an immediate candidate for future work, both so that the DenIM implementation can be fully finalised, and that perhaps attacks on the DenIM key exchange can be investigated further by simulating deniable key exchanges between users.

9.2 Cluster Simulation

To accommodate the scaling issue described in section 7.4, a solution could be to cluster clients distributed on a set of host machines. Each host machine could, for instance, run 500 clients, where all host machines are on the same virtual private network (VPN) to enable a combined packet capture when clients communicate across host machines. Since IM-sim manages each client's behaviour and controls the messages sent, this responsibility of IM-sim could then be distributed to the host machines, or it could be a responsibility of the individual clients, where each client contacts a dispatch server for initialisation and behaviour.

Bibliography

[1] Radicati Group. *Instant Messaging Statistics Report, 2021-2025: Executive Summary.* Accessed: 2025-2-17. 2020. URL: https://www.radicati.com/wp/wp-content/uploads/2020/12/Instant-Messaging-Statistics-Report-2021-2025-Executive-Summary.pdf.

[2] Katriel Cohn-Gordon et al. "A Formal Security Analysis of the Signal Messaging Protocol". In: *J. Cryptol.* 33.4 (Oct. 2020), pp. 1914–1983. ISSN: 0933-2790. DOI: 10.1007/s00145-020-09360-1. URL: https://doi.org/10.1007/s00145-020-09360-1.

[3] Moxie Marlinspike and Trevor Perrin. "The Double Ratchet Algorithm". In: *Applied Sciences* (2016). URL: https://signal.org/docs/specifications/doubleratchet/.

[4] Alireza Bahramali et al. "Practical Traffic Analysis Attacks on Secure Messaging Applications". In: *27th Annual Network and Distributed System Security Symposium, NDSS 2020, San Diego, California, USA, February 23-26, 2020.* The Internet Society, 2020. URL: https://www.ndss-symposium.org/ndss-paper/practical-traffic-analysis-attacks-on-secure-messaging-applications/.

[5] General Michael Hayden, Dr. David Cole, and Major Garrett. *The Price of Privacy: Re-Evaluating the NSA*. URL: https://youtu.be/kV2HDM86XgI?si=wArJkCPWU1ARGVhD&t=1079.

[6] J. Lund. *Technology Preview: Sealed sender for Signal.* 2018. URL: https://signal.org/blog/sealed-sender/.

[7] Ian Martiny et al. "Improving Signal's Sealed Sender". In: *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021.* The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/improving-signals-sealed-sender/.

[8] Boel Nelson, Elena Pagnin, and Aslan Askarov. "Metadata Privacy Beyond Tunneling for Instant Messaging". In: *9th IEEE European Symposium on Security and Privacy, EuroS&P 2024, Vienna, Austria, July 8-12, 2024.* IEEE,

2024, pp. 697–723. DOI: 10.1109/EUROSP60621.2024.00044. URL: https://doi.org/10.1109/EuroSP60621.2024.00044.

[9] Andreas Knudsen Alstrup et al. *Signal Implementation for Testing Metadata Privacy Protocols*. Tech. rep. Aalborg University, 2025.

[10] Anrin Chakraborti, Darius Suciu, and Radu Sion. "Wink: Deniable Secure Messaging". In: *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 1271–1288. ISBN: 978-1-939133-37-3. URL: https://www.usenix.org/conference/usenixsecurity23/presentation/chakraborti-wink.

[11] Hongyan Cui et al. "Heterogeneous characters modeling of instant message services users' online behavior". In: *PLOS ONE* 13.5 (May 2018), pp. 1–21. DOI: 10.1371/journal.pone.0195518. URL: https://doi.org/10.1371/journal.pone.0195518.

[12] Redis. *Redis - The Real-time Data Platform*. Accessed: 2025-01-06. URL: https://redis.io/.

[13] Moxie Marlinspike. *Technology preview: Private contact discovery for Signal*. Sept. 26, 2017. URL: https://signal.org/blog/private-contact-discovery/ (visited on 05/05/2024).

[14] Docker. *Develop with Docker Engine SDKs*. Accessed: 2025-08-05. URL: https://docs.docker.com/reference/api/engine/sdk/.

[15] Docker. *Networking overview*. Accessed: 2025-08-05. URL: https://docs.docker.com/engine/network/.

[16] Docker. *Macvlan network driver*. Accessed: 2025-08-05. URL: https://docs.docker.com/engine/network/drivers/macvlan/.

[17] *Wireshark · Go Deep*. Wireshark. URL: https://www.wireshark.org/ (visited on 11/28/2024).

[18] Google. *gofuzz*. URL: https://pkg.go.dev/github.com/google/gofuzz#section-readme (visited on 04/24/2025).

[19] Carmela Troncoso et al. "Perfect Matching Disclosure Attacks". In: *Privacy Enhancing Technologies, 8th International Symposium, PETS 2008, Leuven, Belgium, July 23-25, 2008, Proceedings*. Ed. by Nikita Borisov and Ian Goldberg. Vol. 5134. Lecture Notes in Computer Science. Springer, 2008, pp. 2–23. DOI: 10.1007/978-3-540-70630-4_2. URL: https://doi.org/10.1007/978-3-540-70630-4_2.

[20] Richard Sinkhorn and Paul Knopp. "Concerning nonnegative matrices and doubly stochastic matrices". In: *Pacific Journal of Mathematics* 21.2 (1967), pp. 343–348.

[21] Mads Møller Kristensen and Mikkel Boje Larsen. "Traffic Analysis Tool and The Deniable Disclosure Attack". Unreleased manuscript of Kristensen et. al. master's thesis. 2025.

[22] Tokio. *Function spawn*. Accessed: 2025-26-05. URL: https://docs.rs/tokio/latest/tokio/task/fn.spawn.html.

[23] Tokio. *Struct Mutex*. Accessed: 2025-26-05. URL: https://docs.rs/tokio/latest/tokio/sync/struct.Mutex.html.

[24] Netcat. *The GNU Netcat project*. Accessed: 2025-02-06. URL: https://netcat.sourceforge.net/.