Summary

This thesis presents the design, implementation, and evaluation of DenIM on SAM, a secure instant messaging system that addresses metadata privacy. While many popular messaging services implement end-to-end encryption using protocols such as the Signal Protocol, communication on their platforms expose metadata. With enough metadata, an adversary can construct social graphs and learn about users relationships and behaviours.

To counter this, the authors implement the DenIM (Deniable Instant Messaging) protocol on top of the Signal Protocol. DenIM depends on a hybrid messaging service where users can send both regular and deniable messages. The protocol provides transport layer privacy by allowing users to piggyback deniable messages onto regular messages. Thereby, the traffic of the deniable messages are covered by the regular messages. This hybrid approach ensures that regular messages maintain low latency, while deniable messages remain hidden in plain sight.

The authors developed the SAM instant messaging from scratch, with modularity and scalability in mind. It uses the Signal Protocol to end-to-end encrypt messages. With the development of DenIM on SAM the, DenIM functionality is integrated through a proxy component that wraps and unwraps deniable payloads transparently, enabling the server to remain unaware of the presence of the DenIM protocol. A Key Distribution Center is introduced to handle prekey generation deterministically based on client-provided seeds, ensuring that key exchange itself does not leak information about the deniable communication.

The system is evaluated in a controlled test environment using Docker and Aalborg Universitys Strato infrastructure. Automated clients simulate realistic behaviour, and the resulting traffic is captured and analysed. The authors demonstrate that deniable messages cannot be distinguished from regular ones in terms of size. A modified version of a normalized statistical disclosure attack tool fails to detect deniable communication, confirming the systems.

The thesis concludes that DenIM on SAM almost satisfies the full DenIM specification and delivers strong metadata privacy. It provides a practical foundation for future privacy-preserving messaging systems and contributes meaningfully to research in secure communications.

DenIM on SAM

Distributed Systems

Alexander S. Steffensen, Magnus J. H. Christensen,
Simon D. Laursen

Software, cs-25-ds-10-09, June 5, 2025

Specialization Project

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

AALBORG UNIVERSITY
STUDENT REPORT

Title:
DenIM on SAM

Theme:
Distributed Systems

Project Period:
Spring Semester 2025

Project Group:
cs-25-ds-10-09

Participant(s):
Simon Deleuran Laursen
Magnus Jørgensen Harder Christensen
Alexander Skytt Steffensen

Supervisor(s):
René Rydhof Hansen
Danny Bøgsted Poulsen

Code Repository:
https://github.com/SAM-Research

Page Numbers: 56

Date of Completion:
June 5, 2025

Abstract:

This thesis presents DenIM on SAM, a secure instant messaging service designed to enhance metadata privacy by implementing the DenIM protocol on top of the Signal Protocol.
DenIM provides transport privacy to certain *deniable* messages by piggybacking them on regular messages, while Signal Protocol provides secure *End-to-End Encryption*.
DenIM on SAM is an attempt at creating the first production ready *Instant Messaging* service that implements DenIM. It is implemented as a Proxy for our *Instant Messaging* Service, SAM. Both services have been created from the ground up to support *End-to-End Encryption* and with scaleability and configurability in mind. We evaluate the system through rigorous testing, including traffic analysis and benchmarking, demonstrating that DenIM on SAM maintains metadata privacy under a strong threat model without compromising usability or scalability.

Acronyms

AWS *Amazon Web Services* 3

E2EE *End-to-End Encryption* 1, 1

HTTP *HyperText Transfer Protocol* . . . . . 2, 15, 25 ff.

IM *Instant Messaging* . . . . . 1, 1 ff., 6, 8 f., 15, 43, 45

KDC *Key Distribution Center* . . . . . 5, 22, 40 f., 46

PQXDH *Post Quantum Extended Diffie-Hellman* 41

X3DH *Extended Triple Diffie-Hellman* 5, 41

Contents

1 | Metadata Privacy

IM services are very popular and continue to see strong growth. With 8.3 billion accounts worldwide in 2021, this number is expected to increase to 9.5 billion in 2025 [1]. Today, many of the popular IM services like WhatsApp [2] and Facebook Messenger [3] use the Signal Protocol to provide E2EE for their users [4]. With E2EE the content of the users message is securely hidden from an adversary but, even though the message content is safe, current IM services still leek metadata when users are messaging.

Metadata in an IM conversation is data related to the conversation, including location and identity of the users in the conversation [5]. This information may seem harmless, but with enough metadata it is possible to create social graphs and identify relationships and behaviours of people [6]. Also, consider the example where the metadata reveals that a government official has had an E2EE conversation with a specific journalist that has just published an article revealing government secrets. It may not even matter what the contents of the message are, the fact that the communication occurred is damning enough.

Even though there have been many proposals on protocols that claim to offer metadata privacy [7, 8, 9, 10], popular IM services do not offer metadata privacy. Signal has made an attempt to hide the sender of a message [11]. This solution is known as Sealed Sender. However, as shown in [12] the sender of messages can still be identified when multiple messages are send back and fourth. This project will look into different transport privacy protocols in order to find the most suitable for an IM service. Furthermore, the project will contribute with an implementation of a transport privacy protocol on top of an already known E2EE protocol.

2 | Transport Privacy

Many protocols that promise transport privacy exist, each with its own set of benefits and trade-offs. Some transport privacy protocols are round-based, which means that communication happens in rounds and all clients must participate in a given round [7, 9]. This approach leads to high network and energy costs over time [13]. It becomes expensive to run on mobile devices where resources are limited. Other transport privacy protocols are delay-based which means that the messages are intentionally delayed at the server, making it difficult to correlate the message that some user sent to a message that someone received from the server [8, 10]. The problem with these transport privacy protocols is that they are not fit for *Instant Messaging* (IM). You could never achieve the same quality of service that users have come to expect from an IM service. Therefore, users would only want to use the protocol when the message they are sending requires transport privacy. This means that communications using these protocols are vulnerable to censorship. As there is no reason to use these protocols if you have nothing to hide, people who have something to hide are the only ones affected by the censorship. Previously, cover-protocols have been suggested to mask the traffic by imitating other, non-suspicious protocols such as HTTP, but as explained in [14], cover-protocols are not sufficient.

The DenIM protocol is a kind of delay-based protocol that uses a hybrid model where most messages are regular instant messages that are delivered instantly, and some messages are *deniable* [15]. Deniable messages are not sent instantly. Instead, they piggyback on regular messages so that an adversary cannot prove that a deniable message was sent or received.

2.1 Combining Transport Privacy and End-to-End Encryption

DenIM piggybacking approach means that the protocol can be layered on top of an existing IM protocol. The DenIM paper argues that such an approach would be more resilient to censorship because it could be implemented in a mainstream messaging

service with millions of users [15]. As explained in chapter 1, many IM services use the Signal Protocol, so it may be a good choice for a DenIM IM service.

Implementing DenIM on top of the Signal protocol requires an IM service using the Signal Protocol that can be modified to support DenIM. It would be easier to implement on top of an already existing implementation instead of implementing one from scratch. Signal has their own IM service by the same name, also using the Signal Protocol. Its implementation is also open source, making it a good candidate. However, in the pre-specialization, prior to this project, an investigation of their implementation revealed that the Signal-Server implementation[1] relies on paid services like *Amazon Web Services* (AWS), Firebase and many more [16]. Signal is open source for the sake of transparency and to allow anyone to audit the code [17], not for the sake of allowing others to easily self-host Signal. Not being able to self host Signal without extensive costs is a deal breaker as this project does not have that amount of resources.

Also, these proprietary technologies that Signal relies on are American, and with the current state of the world, it is popular to opt for a solution not reliant on American companies. Having an open source alternative that is not dependant on the will of Jeff Bezos may be desirable.

As no other candidates were found, a IM service using the Signal Protocol will have to be implemented so that DenIM can be implemented on top of it. Before creating the implementation, we have to dive deeper into the DenIM protocol, to know what it requires to implement the protocol.

[1] The Signal-Server source code: https://github.com/signalapp/Signal-Server

3 | DenIM

The DenIM protocol, as described by [15], uses a hybrid approach to transport privacy where most messages are regular messages that are sent without transport privacy, and some messages are *deniable*. Deniable messages are not instantly delivered. Each time a regular message is sent, chunks of deniable messages can be attached to it. The chunks travel to the server together with the regular message, so that the regular message creates a cover story that explains why the server was contacted. The server will split the regular and deniable parts of the message and forward the regular message to its recipient immediately. The deniable message chunks are saved until the full message can be reconstructed and the server can read who the message is for. Then, the message can be split up into chunks again and attached to any regular messages that are outgoing to its recipient.

Figure 3.1 describes the anatomy of a message in DenIM. As described in [15], the regular part of the message will have some size l and the deniable payload will have size $l \cdot q$ where q can be tuned to accommodate higher or lower throughput of deniable messages.

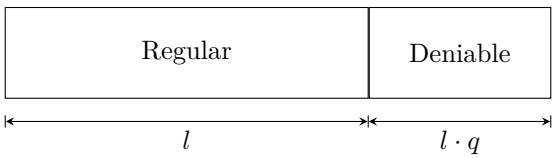


Figure 3.1: A DenIM message has a regular and deniable part. The deniable part size is a product of the length of the regular part and q, a parameter set by the server.

The deniable part of the message may contain any number of chunks, including zero. The only constraint is that the deniable part must have size $l \cdot q$. If there are no more chunks that can be appended, the rest of the space is filled with random bytes [15].

3.1 Key Distribution Center

Since deniable messages must also be encrypted, DenIM has to facilitate the encryption protocol that is implemented under it. In our case, that would be the Signal Protocol.

The only thing that you need to know about the Signal Protocol in relation to DenIM is that it uses the *Extended Triple Diffie-Hellman* (X3DH) key agreement protocol to establish an encrypted session between two users[1]. X3DH requires the clients to upload prekeys that other clients can later retrieve from the server to initialize a session even if the recipient is offline. When clients fetch prekeys, they get a prekey bundle from the server containing all the information that is required to start a session.

The DenIM server must therefore have a *Key Distribution Center* (KDC) that is responsible for distributing keys to clients that request them. This KDC is different from how Signal handles prekeys. According to the X3DH protocol, clients can still participate in the protocol if they run out of one-time prekeys [18]. However, if an adversary sends a key request and the server responds with a prekey bundle without a one-time prekey, it can give information to the adversary about how many deniable conversations a client has started. Therefore, the DenIM specification states that the KDC should be able to generate one-time prekeys on behalf of the user such that it is impossible for a client to run out of one-time prekeys [15].

Since the server is generating the keys, and because DenIM does not want the client to have to request its own keys from the server, the key generation is done using a seeded, cryptographically secure random number generator. Both client and server can then generate the same keys. The client uploads two seeds during registration, one for generating the actual key, and one for the key ID as it has to be random. If the key ID's were simply incremental (as they are in Signal[2]), an adversary could simply subtract one from the current ID to learn how many deniable sessions a user has.

According to the specification, DenIM clients should still be able to upload one-time prekeys, which means there has to be some measure to avoid the client and the server generating two different keys for the same key ID. DenIM on Signal handles this issue by enforcing that every n-th key ID is reserved for the server as depicted on Figure 3.2.

In practice, this is achieved by keeping track of the state of the pseudorandom number generator. Each time the generator is used, the state is incremented by one. When the state reaches some multiple of n, it means that key corresponding to this ID is reserved for the server. The client generates the key using the same method as the server so that it can decrypt any incoming message that claims to be using the prekeys

[1] X3DH has now been deprecated in favour of a post-quantum secure alternative. See subsection 9.1.3.

[2] Signal-Desktop, AccountManager.ts L408-L410

that corresponds to the ID. If the state value is *not* a multiple of n the client can generate the key in any way it wants and upload it to the server using a deniable action. This way, the client can generate however many prekey they want and the server can generate however many it needs.

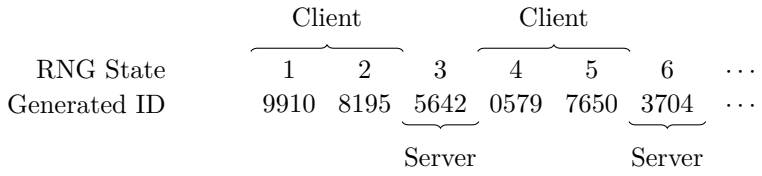


Figure 3.2: Example of Key ID Segmentation where $n = 3$.

In DenIM, clients need some deniable way to retrieve the prekey bundle of other users so that they can initiate a conversation. If an adversary could see that a client made a key request but did not send a regular message immediately after getting a response, they can assume that the client is using DenIM. Therefore, key requests must be handled deniably as well.

3.2 Threat Model

As a part of their design, DenIM defines a threat model [15]. They consider a global active adversary. The adversary can participate in DenIM, which gives them the opportunity to communicate with other users and send requests to the server. Additionally, the adversary can observe, insert, and modify the traffic sent between two parties. The adversary is presumed to be incapable of compromising the internal state of the server and of clients that are not controlled by the adversary. The adversary has three goals. The first is to read or modify deniable payload sent between two honest parties. The second is to learn whether a user is sending deniable messages. The third is to learn who is communicating deniably with whom [15].

As this project will implement DenIM, the same adversary will be considered in this project. The threat model will be discussed later in section 9.6.

3.3 DenIM on Signal

The authors of [15] have created a reference implementation of DenIM using the Signal Protocol called DenIM on Signal[3]. The purpose of this implementation is to perform experiments on the protocol and not to be an actual production-ready *Instant Messaging* (IM) service with DenIM. As such, it is not complete and has issues

[3]DenIM on Signal can be found here: https://github.com/Niteo/denim-on-signal

that would need to be addressed in a proper implementation. The most important omissions of DenIM on Signal in terms of being production ready are as follows:

First of all, DenIM on Signal does not adhere entirely to the specification. It does not support block requests[4] and the deniable payload does not always have $l \cdot q$ size, as the implementation has a bound for how small it can be[5].

Secondly, DenIM chunks in DenIM on Signal do not contain any identifiers to help the recipient to reconstruct the messages if chunks arrive out of order, which means an adversary could block the service simply by delaying packets[6].

Finally, DenIM on Signal is a simple TypeScript project that is clearly not designed to be extensible. It is designed by the authors to run some experiments and as such, there is no support for persistence.

Still, DenIM on Signal serves as a useful reference when designing a DenIM implementation. The implementation gave a closer and more detailed look at how some of the DenIM properties can be implemented. This helped us identifying potential shortcomings, which made it easier to implement a robust solution from the beginning.

[4]Found in DenimClient.cs L576-L582
[5]Found in Util.ts L142-L149
[6]Found in message.proto L68-L71

4 | Problem Formulation

With the description of DenIM and an experimental implementation as reference, an implementation of DenIM on an IM service that uses the Signal Protocol is feasible in this project.

As there is no version of DenIM with the full specification implemented that is suitable for real-world usage, the following problem arises

> *How can we implement an instant messaging service and client library that uses the DenIM protocol on top of the Signal Protocol and is ready for real-world use?*

It should be said that this project focuses on building a library that has all the main functionality using DenIM on top of the Signal Protocol. The library can be used by others to develop a application for a specific medium whether that is for computer or smartphones. To validate that the DenIM protocol has been properly implemented, it needs to be tested against a traffic analysis attack. Additionally, an evaluation should be conducted where the traffic is observed to make sure that messages with deniable chunks piggybacked are indistinguishable from messages with dummy padding or not.

5 System Requirements

The goal is to implement the DenIM protocol specification in a deployable and scalable system. To do this, we must first define the requirements that have to be satisfied in order to claim that we have in fact implemented DenIM. Also, we must define what we mean by a deployable and scalable system.

The system requirements are be split into the requirements for our Signal Protocol compliant instant messaging service implementation, and the DenIM implementation that is going to be implemented on top.

5.1 Instant Messenger Requirements

In order to be production ready, the system should have persistent storage such that the system can be shut down and brought back up without data loss. It also means that the server itself is stateless, meaning it can be scaled horizontally if it is required due to the number of clients interacting with it.

In practice, this requirement means that all data held by the server should be placed in an external database that can be scaled independently of the server.

Since the plan is to implement DenIM on top of the *Instant Messaging* (IM) service, the infrastructure that we create for the IM service should be designed with reusability in mind. Many of the constructs that are required for regular IM will also be needed for DenIM, so reducing coupling between components should be a priority.

The following table sums up the requirements for the system:

ID	Requirement
R-1	Server must use persistent storage to avoid data loss when the server shuts down.
R-2	Client should use persistent storage to avoid data loss when the client shuts down.
R-3	Multiple server instances should be able to exist simultaneously for horizontal scaling.
R-4	Clients should be able to communicate securely using the Signal Protocol.
R-5	Server components should be reusable and interchangeable so that they can be reused for DenIM.
R-6	The system should be able to recover from errors.

Table 5.1: Requirements for the *Instant Messaging* (IM) service.

5.2 DenIM

The DenIM paper explicitly states several requirements that any implementation must follow. These are covered in subsection 5.2.1 Aside from those, there are requirements that must be satisfied in order to create a viable implementation strategy for a DenIM compliant instant messenger. These are covered in subsection 5.2.2.

5.2.1 DenIM Protocol Requirements

This section aims to formalize the requirements of the DenIM protocol for the purpose of clarity. A summary of the requirements can be found in Table 5.2

DenIM provides transport layer deniability through a hybrid approach where some messages are regular and some messages are deniable [15]. The deniable messages are transported between the clients and server by piggybacking on regular messages. Each regular message that is sent has some amount of space $(l \cdot q)$ for deniable content that can be piggybacked on it. This is covered by DenIM Requirement D-1. The need for higher through-put of deniable messages may necessitate that the server regulates the value of q to achieve the best quality of service. This is covered by DenIM Requirement D-2. If a high volume of deniable messages are being sent, the server should increase the value of q to ensure that clients are able to deliver their deniable messages without too much delay.

Because the amount of content in a deniable payload is limited, deniable messages are split into chunks. Both server and clients uses a buffer to reassemble the chunks as they arrive. This is requirement DenIM Requirement D-3. Additionally when the client and server are preparing the deniable payload for a message, they have to use a buffer that takes from the oldest deniable message. This is requirement DenIM Requirement D-4. Since DenIM payloads sometimes contain garbage and other times contain actual chunks that need to be processed, DenIM implementations need to

be careful about creating timing channels. One mitigation that the server should implement is to always forward the regular message *before* processing its deniable payload. This is DenIM Requirement D-5. Doing this still leaves a potential timing channel when the server attaches deniable payload to an outgoing message. This is discussed in subsection 9.1.4.

Before any message can be sent using DenIM, the sender of the message must fetch a prekey bundle of the recipient to start an encryption session in accordance with the Signal Protocol. According to the Signal Protocol, clients upload new one-time prekeys when they register and also refill the server's key store periodically to avoid running out of one-time prekeys [19]. In DenIM, this is not sufficient. If a client runs out of one-time prekeys, it leaks information to the adversary about how many sessions the client has. Therefore, the server must be able to generate new one-time prekeys for the clients (DenIM Requirement D-6).

Since the client also needs to know the keys that the server has generated on its behalf, the key generation is performed using a cryptographically secure seeded pseudorandom generator (DenIM Requirement D-9). The key needs to be randomly generated but, as mentioned in section 3.1, the identifier of the key must also be random (DenIM Requirement D-10). The client therefore uploads two seeds during registration that both the server and the client can then use to generate the same keys and key identifiers for each iteration (DenIM Requirement D-7). To ensure that the server and the clients agree on how many keys have been generated, the server includes the iteration count of the random generator in each message that it sends to the clients (DenIM Requirement D-8).

as described in chapter 3, the primary feature of DenIM is the fact that deniable messages are piggybacked on regular messages. "Messages", in this context, refers to all communication between client and server, not just the messages that are addressed to other users (DenIM Requirement D-11).

The DenIM paper outlines three actions that a client has to be able to perform deniably [15]. They are as follows:

- Clients have to be able to silently block other users from contacting them. This is to avoid an adversary blocking the receive queue of a client by spamming them (DenIM Requirement D-12.
- Clients also have to be able to exchange keys with the server. The client must be able to request keys of other users (DenIM Requirement D-13) and upload its own keys (DenIM Requirement D-14).
- Finally, clients must obviously be able to send messages to each other deniably (DenIM Requirement D-15).

ID	Requirement
D-1	The deniable part of a message should be exactly $l \cdot q$ bytes.
D-2	q is a system-wide global configuration value dictated by the server.
D-3	Clients and server must have buffers to reassemble deniable messages as they arrive.
D-4	Clients and server must have buffers used to send the oldest deniable messages first.
D-5	Server should forward regular message before processing DenIM chunks.
D-6	Server must have a KDC that can generate one-time prekeys on behalf of the user.
D-7	The KDC must generate keys using a seed provided by the client so that the client can also generate the same key.
D-8	The server must inform the client how many times the key generator has been used so that the client can synchronize.
D-9	Keys must be generated with a cryptographically secure random generator.
D-10	The identifier for each key must be random.
D-11	Client and server must be able to communicate deniably by piggybacking messages.
D-12	Clients must be able to block other users deniably.
D-13	Clients must be able to request prekeys deniably.
D-14	Clients must be able to upload prekeys deniably.
D-15	Clients must be able to send messages to other users deniably.

Table 5.2: Explicit requirements dictated by the DenIM specification.

5.2.2 DenIM Implementation

ID	Requirement
R-7	The DenIM specification should be implemented as described by [15].
R-8	Server should have a strategy for deleting old key IDs when they are no longer able to cause a collision.

Table 5.3: Requirements specific to our DenIM implementation.

6 | SAM Instant Messenger

SAM[1] is a complete overhaul of the instant messaging service that we developed during the previous semester[2]. There were enough problems with the previous implementation that we concluded that a complete rewrite was in order. The most important issues that we identified included the error handling. If something went wrong at runtime, it was difficult to figure out what and how to handle it properly. Another issue was that the project had a rigid structure with high coupling that did not allow for us to tinker with the configuration.

SAM consists of a client and a server. The following sections cover how they are structured and how they communicate.

6.1 SAM Server

SAM Server is architected around a request router that routes incoming requests from clients to a corresponding endpoint handler function. The endpoint handlers perform the desired operations and return a result to the request router which forwards the result to the client. The data that is required to perform the operations is held by managers. For instance, the key manager is responsible for managing all of the prekeys and supports actions such as adding and deleting a prekey.

A manager in SAM is a type that implements one of the four manager interfaces: `AccountManager`, `DeviceManager`, `KeyManager` and `MessageManager`, and their area of responsibility should be obvious from their names. `AccountManager` manages accounts, `DeviceManager` manages information about user devices, `KeyManager` manages prekeys and `MessageManager` stores messages until the recipient is ready to receive them.

Managers are responsible for storing data, which means that any type that implements a manager interface must have some strategy for doing that. When deploying a SAM Server instance in a production environment, this storage strategy should be using

[1] SAM is the initials of the authors of the report
[2] Available here: https://github.com/Diesel-Jeans/signal

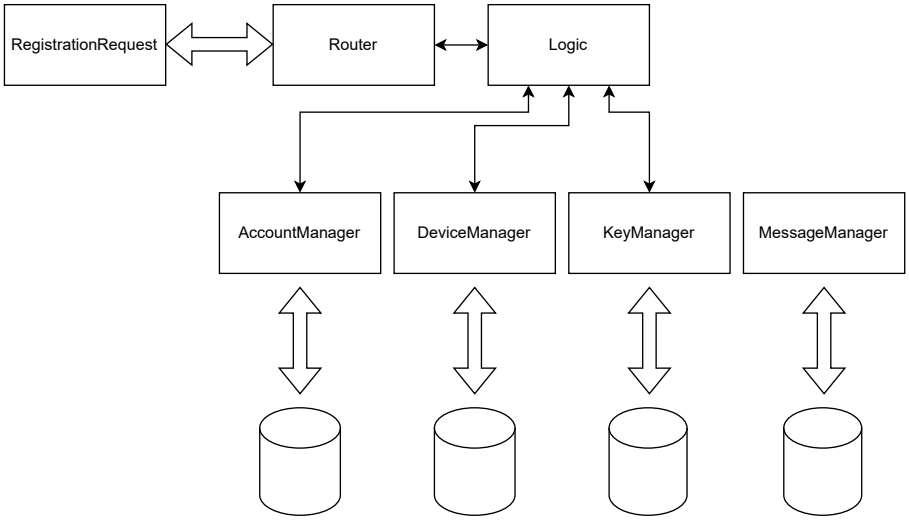


Figure 6.1: SAM Server: Parts related to handling a registration

some form of persistent storage such as an SQL database.

To illustrate how the architecture works, consider a registration request as in Figure 6.1: A user sends a request containing information about the account that they would like to register (such as their desired username), but also information about their primary device (such as a key bundle) since the Signal Protocol requires users to include this during the registration phase. Assuming that the registration request is valid, the endpoint handler function for this request invokes a series of functions that will eventually have the account manager store the account, the device manager will have stored the user's primary device and the key manager will have processed the keys in the key bundle and stored them as well.

6.2 SAM Client

The SAM Client contains all methods that you would need to communicate with other clients through the server. It also contains a collection of data stores which are responsible for storing all data related to a client. Some store types are required by `libsignal`. These are for storing keys, prekeys and sessions. Some stores are defined by us. These are for storing contacts, account information and messages and are configured to use SQLite for persistent storage.

Sending a message works as illustrated by Figure 6.2: The client calls a function in the logic module that constructs an encrypted message during step 1. Since the recipient may have multiple devices, each with their own keys, the message must be encrypted once per recipient device as per the Signal Protocol. Therefore, in step

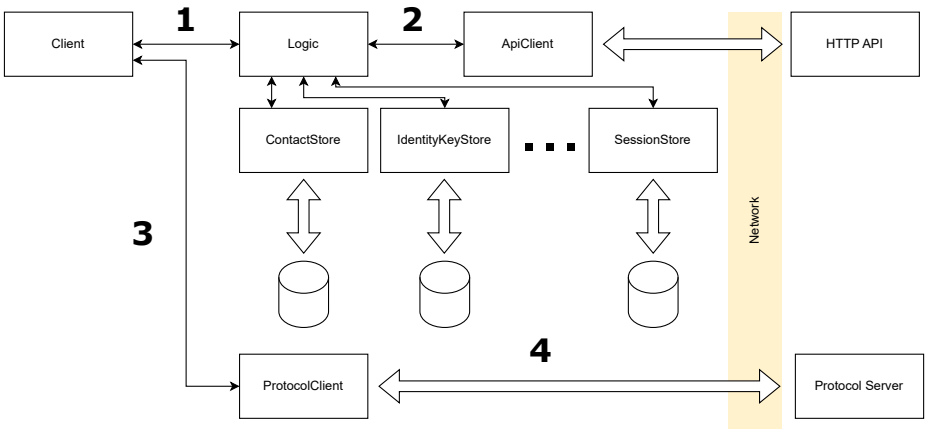


Figure 6.2: Client Sending a Message

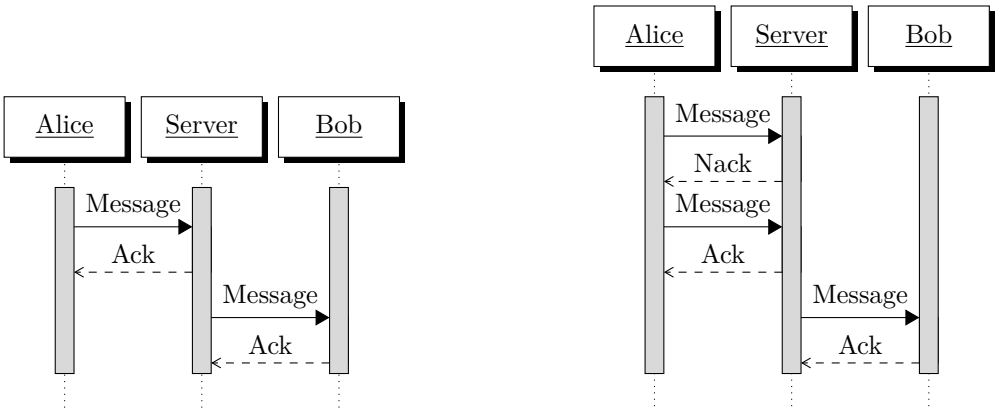
2, the API client is invoked to fetch prekeys for each recipient device if the client does not already have an established session with that device. Once the prekeys have been fetched, the message can be encrypted for each recipient device and all of the messages are put into an envelope and sent to the server in step 3. If the envelope was prepared correctly, the server replies with an acknowledgement. Otherwise, the server will respond with the reason that the envelope was rejected in step 4.

Imagine the case where Alice and Bob have a conversation and Bob decides that he would like to also have SAM on his other computer. He links the new device to his account, but Alice does not know about this new device yet. Next time they communicate, she sends a message addressed to Bob's initial device only. Once the envelope reaches the server, the server will discover that it does not contain a message for Bob's second device and reject it. Upon rejection, Alice's client will call a handler function in the logic module that attempts to fix the issue by fetching the prekeys for Bob's new device, encrypt the message for the new device, and re-deliver the message to the server.

6.3 Client - Server Communication

SAM clients have two ways that they communicate with the server. The method used depends on the scenario. Requests that a client might need to make to the server are performed using HTTP. This includes performing key requests and account registration. For *Instant Messaging* (IM), SAM uses a simple communication scheme created with Protobuf and WebSocket. Using WebSocket ensures that messages can be delivered to the recipient instantly if they are online.

The communication protocol that SAM uses through WebSocket is called *SAM Protocol* and consists of a `ClientMessage` type that the clients can send to the server



(a) SAM Protocol: Successful message

(b) SAM Protocol: Rejected message

Figure 6.3: SAM Protocol Examples

and a `ServerMessage` type that the server can send to clients.

A `ClientMessage` can either contain an encrypted message that the user wants the server to deliver to some recipient or it can contain an acknowledgement that the client has received a message that the server has previously sent.

A `ServerMessage` can either contain an encrypted message that was addressed to the recipient or some status message such as an acknowledgement or a rejection of the client's request. If the client receives a rejection, the server will include the reason why the request failed so that the client can attempt to fix the problem before retrying the request as seen on Figure 6.3b.

7 DenIM on SAM

As DenIM on SAM will use the DenIM protocol, its design will be different from SAM. However, DenIM on SAM should reuse as much of SAM as possible. This chapter explains how we constructed a DenIM extension on top of SAM and how we reused as much as possible from SAM to do it most efficiently.

7.1 DenIM Proxy

Because DenIM on SAM is an extension of SAM, we decided to implement DenIM as a WebSocket proxy for SAM message communication. This creates a distributed system where the DenIM Proxy communicates with the SAM Server over the SAM Protocol and DenIM clients using the DenIM Protocol. This is illustrated in Figure 7.1.

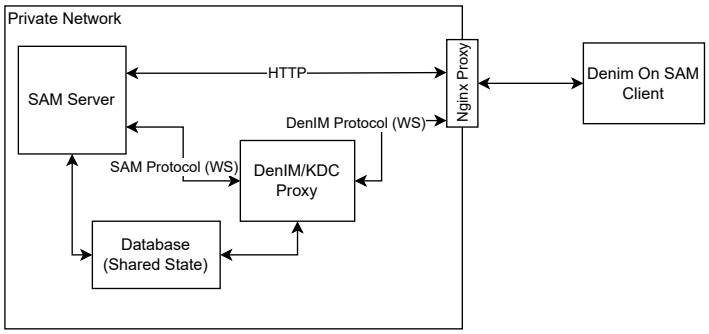


Figure 7.1: Infrastructure of the server on DenIM on SAM.

When the client sends a regular message, it is wrapped in a DenIM Protocol message as the regular payload. Afterwards, the deniable payload is added before sending the message. The client sends the WebSocket message to the DenIM Proxy. Once the DenIM Proxy receives the message, it will immediately forward the regular payload (the SAM Protocol message) to the SAM Server to reduce the chance for timing leaks. The message is forwarded over an authorized WebSocket connection created on behalf of the client. The SAM Server does not know that it communicates with

a DenIM Proxy. When the SAM Server sends a WebSocket message out it is send through the DenIM Proxy which wraps the message in a DenIM Protocol message and adds a deniable payload to the message. The SAM Server and DenIM Proxy shares a database running on another server, which allows the DenIM Proxy to retrieve account information of clients without the need to ask the SAM Server. This design also has the benefit of being stateless for both the SAM Server and DenIM Proxy, allowing for horizontal scaling.

7.2 Reconstructing Deniable Messages

The DenIM paper does not specify how deniable messages are reconstructed when the deniable payload is received. Therefore we have to design our own way of reconstructing deniable messages from DenIM chunks.

As DenIM chunks can be received out of order it is important to be able to tell which message each DenIM chunk belongs to and also its position in the sequence of chunks for that message. To do this, each deniable message is given an ID that is unique for the receiver such that they can differentiate between the messages. Furthermore each chuck is also given a sequence number, indicating its position so each message can be ordered correctly. The receiver should additionally be able to know whether a chunk is the last in the sequence. This is achieved by each chunk having a flag indicating whether it is the last chunk of a message or not. An example of two deniable messages parted into chunks is seen in Figure 7.2. It is possible to distinguish each chunk and know which message it belongs to.

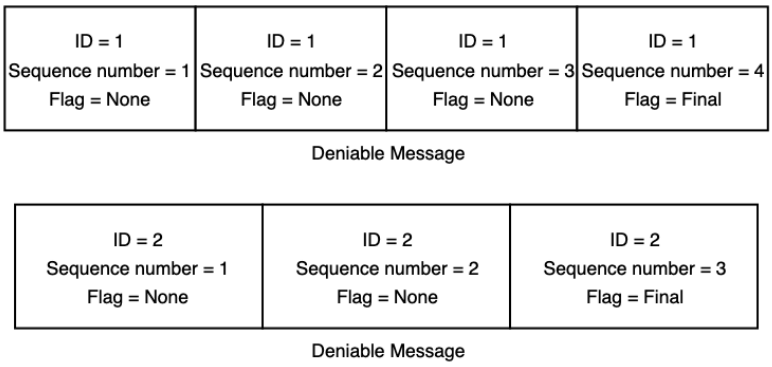


Figure 7.2: Two Deniable Messages parted into chunks, each chuck is distinguishable from another.

When the proxy or the client receives a deniable payload they will use a `ReceivingBuffer` to recreate the deniable message. Listing 7.1 shows a possible implementation of a `ReceivingBuffer`. It consists of a `HashMap` that maps a `MessageId` to a `ChunkBuffer`. The `ChunkBuffer` holds the current chunks received in `ChunkBuffer.chunks` and the

`SequenceNumber`s of the chunks it knows that is missing in `ChunkBuffer.waiting_for`.

```
1 struct ReceivingBuffer {
2     chunk_buffers: HashMap<MessageId, ChunkBuffer>,
3 }
4
5 struct ChunkBuffer {
6     chunks: HashMap<SequenceNumber, Vec<u8>>,
7     waiting_for: HashSet<SequenceNumber>,
8 }
```

Listing 7.1: `ReceivingBuffer` struct used to receive and reconstruct a `DeniableMessage` from chunks for a user.

When a `DeniablePayload` is received, a function that processes the chunks and decodes the messages is called. The pseudocode for this function is shown in Algorithm 1. It ensures that all chunks of a message are received before attempting to reconstruct it and that all chunks are ordered correctly.

The algorithm keeps track of the chunks that it must receive before it can reconstruct a message. When a chunk is received that is not a final chunk, the algorithm adds the next chunk ID to the list of awaited IDs. This is handled in the first highlighted section (from line 9 to 11). The second highlighted section (from line 12 to 20) handles the case where a chunk is received out of order. In this case, all IDs before the ID of the current chunk must be added to the list of awaited IDs if they have not already been received. Finally, if all the chunks of a message have been received, the message bytes are decoded.

Algorithm 1 Process and Decode Chunks

Input: `ReceivingBuffer` and a list of `DenIMChunk`s
Output: list of `DeniableMessage`s

```
1:  Messages ← empty list
2:  for all Chunk in Chunks do
3:    if Chunk.Flag = DummyPadding then
4:      continue
5:    end if
6:    ChunkBuffer ← ReceivingBuffer.Buffers[Chunk.MessageId] or create new
7:    seq ← Chunk.SequenceNumber
8:    next ← seq + 1
9:    if Chunk.Flag ≠Final and next ∉ ChunkBuffer.WaitingFor then
10:     ChunkBuffer.WaitingFor.insert(next)
11:   end if
12:   if seq ∉ ChunkBuffer.WaitingFor then
13:     for id = 0 to seq − 1 do
14:       if id ∉ ChunkBuffer.WaitingFor then
15:         ChunkBuffer.WaitingFor.insert(id)
16:       end if
17:     end for
18:   else
19:     ChunkBuffer.WaitingFor.Remove(seq)
20:   end if
21:   ChunkBuffer.Chunks[seq] ← Chunk.Content
22:   if ChunkBuffer.WaitingFor is not empty then
23:     continue
24:   end if
25:   DeniableMessage ← Decode(ChunkBuffer.Chunks)
26:   Messages.Add(DeniableMessage)
27: end for
28: return Messages
```

Both the client and server receives deniable payloads, and therefore both use the functionality explained in this section. When a client sends a message to the server, it uses a unique message ID to ensure that the server can assemble the message correctly. Then, when the server forwards the message to the recipient, the server assigns a new message ID to ensure that the message ID does not collide with the ID of another message that the recipient has previously received.

7.3 Message Structure

A message in DenIM on SAM follows the structure seen in Figure 7.3. A `DenIM Envelope` is used as a wrapper around `DenIM Message`, as `DenIM Envelope` also can be a `QStatus` message just containing the q value. `QStatus` message is only used when a client connects to the proxy and needs to know the q value. `DenIM Message` consists of a `Regular Payload`, `Deniable Payload` and q. The `Regular Payload` is the regular message, and the `Deniable Payload` are the piggybacked deniable messages.

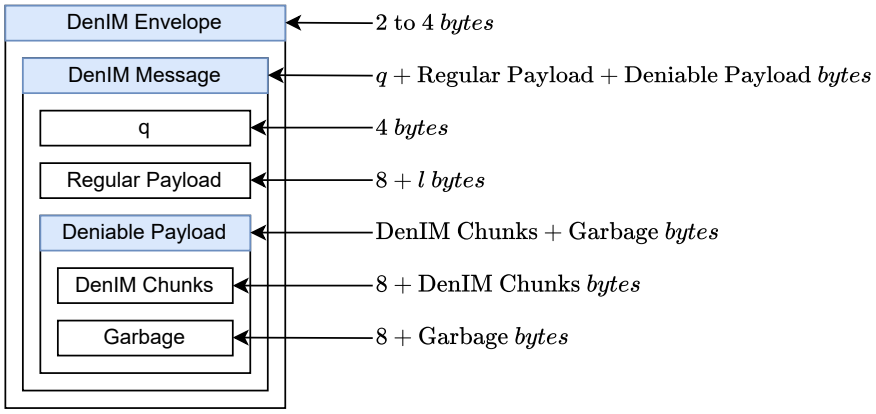


Figure 7.3: Structure of a message in DenIM on SAM.

The `Deniable Payload` has two parts, `DenIM Chunks` and `Garbage`. This is to satisfy DenIM's requirement that the deniable payload should always have a size of $l \cdot q$. During preparation of the deniable payload, DenIM on SAM will have to check if there are enough bytes available to include a DenIM chunk. As explained earlier in section 7.2 a chunk has overhead and as a result, a scenario can happen where there are some bytes available but not enough for a DenIM chunk. When this happens the deniable payload is padded with random bytes to reach a size of $l \cdot q$.

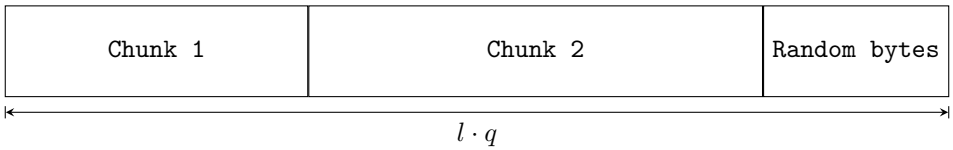


Figure 7.4: A Deniable payload with two chunks. As there are not enough bytes available to fit another chunk it is padded with random bytes up to size $l \cdot q$.

Furthermore, due to the DenIM requirement, DenIM on SAM cannot use Proto-

buf for encoding `DenIM Message`s because Protobuf uses variable-width integers [20]. Variable-width integers allows one to encode an unsigned 32 bit integer using between one and five bytes depending on its value. Because its size is value dependent, it will be harder to predict encoded size of the deniable payload. Instead, DenIM on SAM should use `bincode`[1] with a fixed-length integer encoding. This means the encoded size of a unsigned 32 bit integer would always have a size of four bytes. `Bincode` is used to encode `DenIM Message` and creates a constant overhead of 28 bytes. These bytes comes from the q type size and the vector length value of `Regular Payload`, `Denim Chunks` and `Garbage` as shown in Figure 7.3. Additionally the size of a encoded `DenIM Envelope` can range from two to four bytes because it uses Protobuf. This means the total size of a message sent over DenIM on SAM is

$$l + l \cdot q + o$$

where $30 \leq o \leq 32$ bytes. The overhead value o will not leak any information about the deniable payloads, as the variable size of o is also publicly known.

7.4 Starting a Deniable Conversation

In SAM, when a user wants to start a new conversation with another user, they would first have to fetch the prekey bundle of that other user before sending the first message. The procedure is similar when starting a new deniable conversation in DenIM on SAM, but because everything is sent as deniable messages, and DenIM uses a KDC, the procedure has a little twist.

Consider the example where Alice wants to send a message to Bob as depicted on Figure 7.5. It starts with Alice who wants to start a conversation with Bob. She creates her initial message to Bob, and her client notices that Bob is a new contact, so instead of enqueueing the message, it will store it for encryption once Bob's prekey bundle is received.

Alice's client enqueues a key request for Bob's prekey bundle. Once the key request reaches the proxy, the KDC will try to create a prekey bundle using Bob's keys. If the proxy has not received Bob's seeds for generating one-time prekeys, the KDC cannot generate a prekey bundle. When this happens, the proxy will store Alice's key request on the proxy, and wait for Bob to upload his seeds. Once Bob has uploaded his seeds, the KDC will generate a prekey bundle for Alice and the proxy will send it to the receivers.

Once Alice receives Bob's prekey bundle, she will find the message she has stored, encrypt it, and send it to Bob. When Bob receives Alice's initial message, he will use the seeds he uploaded to generate the keys Alice used when she encrypted the initial

[1] bincode https://github.com/bincode-org/bincode

message.

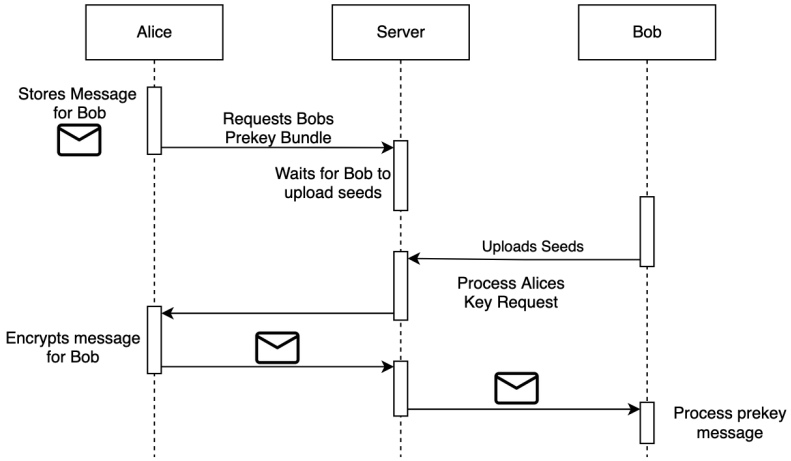


Figure 7.5: A sequence chart showing how a conversation can be started on DenIM on SAM. The envelope is the initial message.

7.5 Key Distribution Center

According to the DenIM specification, clients should be able to upload one-time prekeys to the server to avoid the server having to generate them. In our implementation, the client cannot do this. This is to prevent the client's sending buffer being filled with key upload requests instead of deniable messages. We discuss the issue further in subsection 9.1.2.

8 Evaluation

To verify our implementations of both SAM and DenIM on SAM, we have implemented both services with rigorous unit testing, integration testing, end-to-end testing and system testing.

8.1 Testing Strategy

From the start of development, we wanted to have an extensive test suite, so we could catch bugs early. Whenever we implemented some functionality for our libraries, we would write unit tests for it and later, when our libraries had become more mature, we would write integration testing for the different components in the libraries. When writing our integration tests, we would sometimes realize that changes to the functionality were required. Making these changes sometimes had unintended side effects that might have gone unnoticed if our suite of unit test had not caught them. When our libraries, were done and all unit and integration tests were passing, we began to implement end-to-end tests. Here again we would need to make changes to the code base, and our test suite would catch errors and unintended side effects, which would have been hard to detect without the test suite. This approach lead us to be confident in our development process and not being afraid to make changes to the code base. If our test suite was passing, we were confident that our solutions were working correctly. After this we would go on to do full system testing of our implementations.

8.2 System Testing Environment

For the system test of DenIM on SAM we used Docker Compose to run all the services required to make it work. We used a VPS from Aalborg University's Strato cluster running Ubuntu 24.04.2 LTS on a Intel Xeon Skylake processor with 16 cores and 64 Gigabytes of RAM.

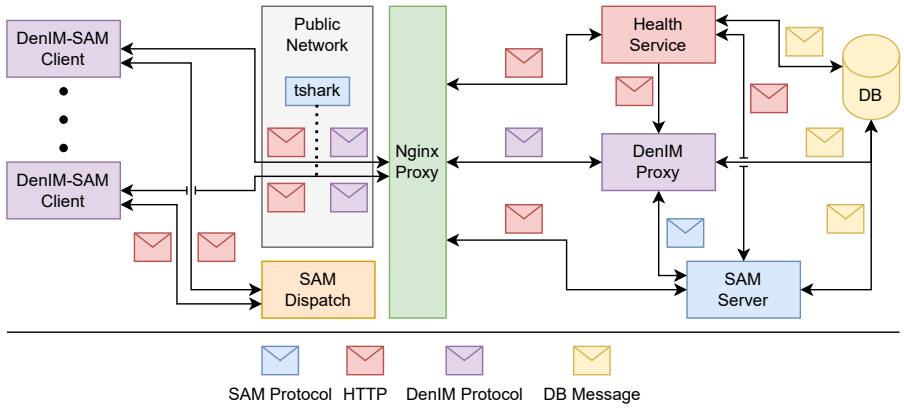


Figure 8.1: DenIM on SAM System Testing Environment

8.2.1 Message Types

Depicted as envelopes in Figure 8.1 are the different protocols used in the system testing environment.

The blue SAM Protocol envelopes are WebSocket messages containing SAM Protocol messages. These messages are only sent between the SAM Server and DenIM Proxy. The red envelopes are HTTP messages that are sent between the following instances.

- DenIM on SAM Clients to and from the SAM Dispatch.
- DenIM on SAM Clients through the Nginx Proxy to and from the SAM Server and the Health Service.
- Health Service to and from the DenIM Proxy and SAM Server.

The purple DenIM Protocol envelopes are WebSocket messages with the DenIM Protocol Protobuf messages. These are sent to and from DenIM on SAM Clients through the Nginx Proxy. The yellow Database messages are SQL queries and SQL results, which are received and sent from the following instances.

- DenIM Proxy and the Database
- SAM Server and the Database
- Health Service and the Database, where it only checks if certain tables exist.

8.2.2 SAM Dispatcher

To make the testing environment versatile and configurable we developed a SAM Dispatcher seen in Figure 8.1. The SAM Dispatcher is responsible for giving clients in the testing environment an identity, synchronizing the clients and saving a report of all the clients activities.

The SAM Dispatcher works by reading a configuration file containing client behaviour settings and the amount of clients it should expect to be contacted by. It will then generate client identities that include client behaviour that is configurable on their sending rate, reply rate, their reply probability, and when to discard received messages.

Reply rate is how many ticks pass between replying to messages whereas reply probability is the probability that a reply will be sent when the current tick is a reply tick.

The client identity also includes a weighted friend list such that each client has a number of friends and a few friends are weighted higher than others. These are the *best friends*. Best friends are mutual meaning that if Alice's best friend is Bob, then Bob's best friend is Alice. The friend lists also represents clusters where two clusters never talk to each other using regular messages, only deniable messages.

Clients will ask the dispatcher for an identity, and when the clients are ready they will tell the dispatcher that they are waiting for synchronization. When all clients have been given an identity and told the dispatcher that they are ready, the dispatcher will give them a ready signal and the simulation will begin.

When the clients are done with their experiments they will upload a report to the dispatcher about with whom they have communicated with and all the messages along with timestamps. When the dispatcher has received reports from all the clients it will save them as one big report. All communication between clients and the dispatcher happens over HTTP.

8.2.3 DenIM on SAM Test Clients

Depicted on in the Figure 8.1, the DenIM on SAM Test Clients perform actions dynamically depending on the specific identity they receive from the SAM Dispatcher.

We modelled and implemented these clients as simple test client from using the DenIM on SAM client API and the SAM client API. It uses both APIs to be able to perform experiments with only SAM as well. When a client has received an identity from the SAM Dispatcher, it will register itself with the DenIM on SAM Server, and signal to the SAM Dispatcher that it is ready to begin the experiment. When the SAM Dispatcher signals to the test client that it can begin, it will follow the state machine depicted below.

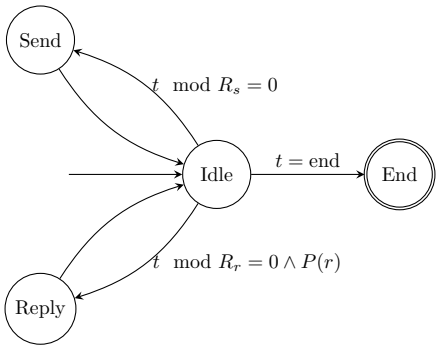


Figure 8.2: DenIM on SAM Test Client State Machine

Every client has a clock that is configured by the SAM Dispatcher, so an experiment can be run at configurable speeds. Each time the clock ticks the value t is incremented by one.

The client also has two action rates, where R_s represents the rate at which the client will send messages to a user and R_r represents the rate at which a client will reply to a received message. Each client also has a probability of which to respond to a message, so it does not respond to all messages, this is represented by $P(r)$.

A client has four states. It will begin in the *Idle* state and each time the clock ticks it will decide if it should reply to a received message and if it should send a new message by going to either the *Reply* or *Send* state respectively. A client can go to both states in one tick, and come back to the idle state. When the clients clock reaches the end tick it will stop experiment and send their reports to the SAM Dispatcher.

8.2.4 Public Network

All test clients communicate with the DenIM on SAM server through a public network interface. On this network we run a TShark instance capturing all packets to and from the DenIM on SAM server. This enables us and potential users of our system testing environment to perform traffic analysis on the captured traffic.

8.2.5 Private Network

To hide all internal traffic from outsiders and clients, we configured a private network where the distributed DenIM on SAM infrastructure resides. Seen in Figure 8.1, the DenIM on SAM infrastructure is behind an Nginx Proxy. The Nginx Proxy only allows clients to establish WebSocket connections to the DenIM Proxy. HTTP requests about SAM registrations and keys are routed to the SAM Server. The Nginx Proxy in this environment also exposed a health endpoint which is routed to the Health service. This has been done so the clients can synchronize with the DenIM

on SAM Server on the initial docker compose startup phase. The Health Service is a simple service that just asks the other SAM services if they are up.

8.2.6 Configuration

The system testing environment can be configured to a wide range of scenarios, it supports both TLS and insecure communication between clients and the DenIM on SAM infrastructure DenIM can also be disabled entirely, so that the client only can use SAM. The internal communication of services can also be configured to use mutual TLS, enabling services authenticating each other instead of only client services authenticating server services.

Clients can also be configured for many different scenarios. All services and clients need to be configured with their own TLS certificates, configuration files and docker configurations. Managing all of this can be very difficult, so instead of having to configure everything from many different configurations, we have made a setup generator that will configure everything from one configuration.

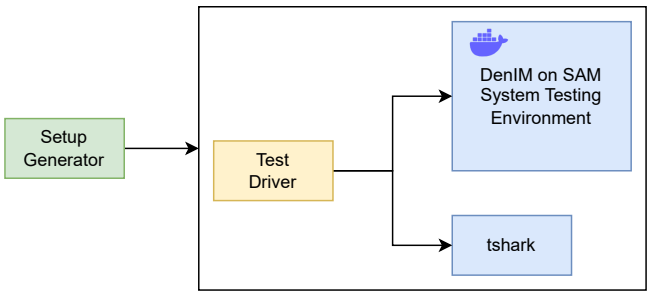


Figure 8.3: DenIM on SAM System Testing Environment Generator

Depicted in Figure 8.3, the Setup Generator takes in a configuration file describing all settings for both the internal DenIM on SAM infrastructure and the client behaviour. Subsequently, it generates a project directory that contains a test driver and a docker compose configuration file. Running the testing environment only requires one to launch the Test Driver, which will start all services and the TShark instance. While the experiment is running the Test Driver monitors the status of the experiment. Once the experiment has finished the Test Driver will stop the TShark instance from capturing and shutdown the testing environment. When a test run is done the system testing environment and TShark instance have generated a report JSON file and a traffic pcap file, that can be used for further analysis.

8.3 Anonymity Assurance

We want to assure that our implementation of the DenIM on SAM client API and the DenIM on SAM server infrastructure is correct and correctly uses DenIM to piggyback messages between a client in one friend group and client in another friend group, without being able to observe it in the traffic.

Our scenario is inspired by example used to explain DenIM in [15] and contains four clients: Alice, Bob, Charlie and Dorothy. Alice and Bob communicates over regular messages, and Charlie and Dorothy do the same. Alice and Dorothy also communicate with each other using deniable messages.

We will do this by creating an instrumented program and make the clients perform the scenario while tracing their traffic. Each event in the scenario happens in a noticeable time delay (5 seconds) apart from each other to make it easier to analyse the traffic trace.

The clients will perform these actions in the given order:

ID	Action
AA-1	Alice sends a message to Bob with a deniable key request for Dorothy's keys.
AA-2	Bob sends a message to Alice where the proxy has put a deniable key response with Dorothy's keys in it.
AA-3	Alice sends a message to Bob containing a deniable message to Dorothy.
AA-4	Charlie sends a message to Dorothy where the proxy has put Alice's deniable message to Dorothy in it.
AA-5	Dorothy reads the deniable message.
AA-6	Dorothy sends a message to Charlie containing a deniable message to Alice.
AA-7	Bob sends a message to Alice where the proxy has put Dorothys deniable message to Alice in it.
AA-8	Alice reads the deniable message.

Table 8.1: Actions the clients will perform in the scenario with Alice, Bob, Charlie and Dorothy.

We ran the instrumented program with the four clients and the DenIM on SAM server infrastructure performing the actions in Table 8.1. The clients would all communicate with a different message size, for example Alice would always send 400 bytes. This will help us identify if our DenIM implementation works as expected. The message sizes can be seen in section A.1.

From this we obtained a log from the server and clients describing what they have done and the traffic trace encrypted with TLS. We also ran it without TLS encryption to be able to see the sizes of each WebSocket packet to confirm that they do not change

size if there are DenIM chunks in them. We will first confirm that our deniable padding of $l \cdot q$ is working and later explain the traffic traces and how you cannot see that two clients are communicating over DenIM. Secondly, we will confirm that, when observing line charts, we cannot see that Alice and Dorothy communicating with each other.

When running the instrumented test program we observed that the size of Alice's message to Bob is always 1195 bytes, no matter if it contains DenIM chunks or not. The value of q is 1 and we found that the size of the encrypted message (l) is 582 bytes. Since the DenIM portion has 31 bytes of overhead, the actual calculation is $l + l \cdot q + o$ where o is the extra overhead of the structure. The calculation comes out to:

$$582 \text{ bytes} + 582 \text{ bytes} \cdot 1 + 31 \text{ bytes} = 1195 \text{ bytes}$$

The logs from the test run can be found in section A.1.

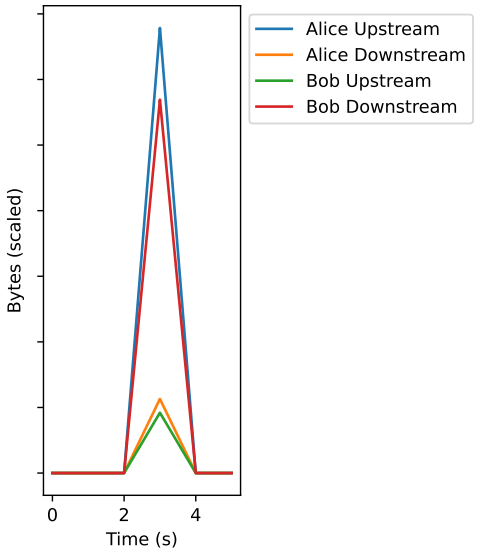


Figure 8.4: Alice sending a regular message to Bob.

In Figure 8.4 we can see Alice sending a message to Bob. Each spike either represents upstream from a client to the server or the downstream from the server to a client. They are scaled a bit so that they are not directly on top of each other. The biggest spike (blue) is Alice sending a message to Bob. The second biggest spike (red) is the server sending Alice's message to Bob. The smaller spike (orange) is the server telling Alice that it has received her message. The smallest spike (green) is Bob telling the server that he has received its message. This traffic happens almost instantaneously, which is common for all regular traffic.

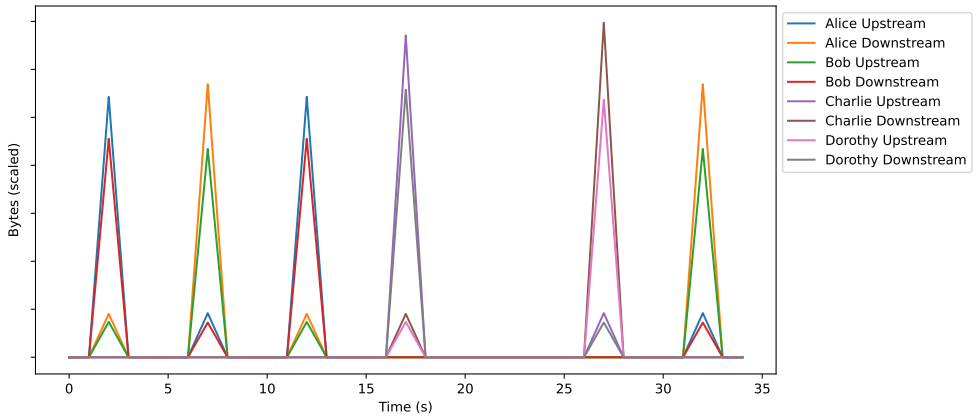


Figure 8.5: Traffic trace from the senario explained in Table 8.1.

Figure 8.5 contains the traffic trace from performing the actions described in Table 8.1. It shows the upstream and downstream of all clients in the instrumented test. After timestep 20, Anonymity Assurance Step AA-5 (Dorothy reads the deniable message.) is performed, and does not show up in the network traffic as its happens inside the client. The final two actions appear to be Charlie and Dorothy talking and Alice and Bob talking, but a deniable message was in fact transferred to Alice.

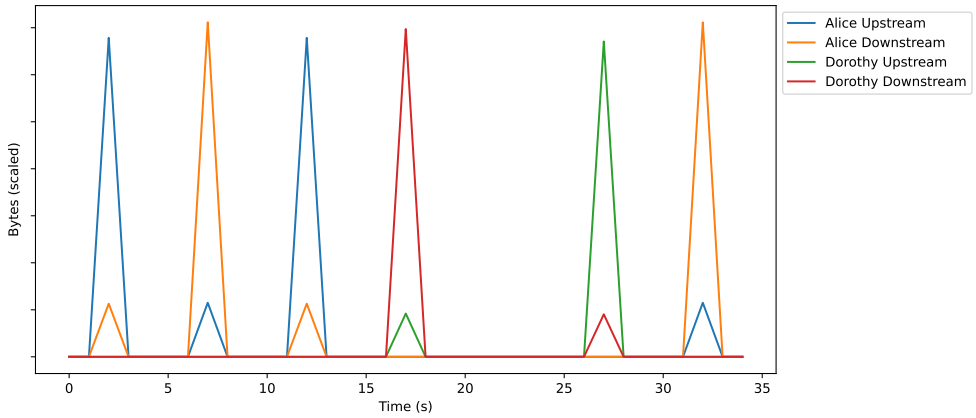


Figure 8.6: Alice and Dorothy deniable communication.

If we look at the traffic from Alice and Dorothy in isolation, we can see that there are only two spikes at a time and that their streams do not overlap. Whenever Alice sends a message, Dorothy is not receiving any message and whenever Dorothy is sending a message Alice is not receiving any.

8.4 Traffic Analysis

Our results from section 8.3 are promising, but to further prove that we have implemented DenIM correctly we will use our system testing environment to generate traffic traces of clients communicating with each other through both regular messages and deniable messages.

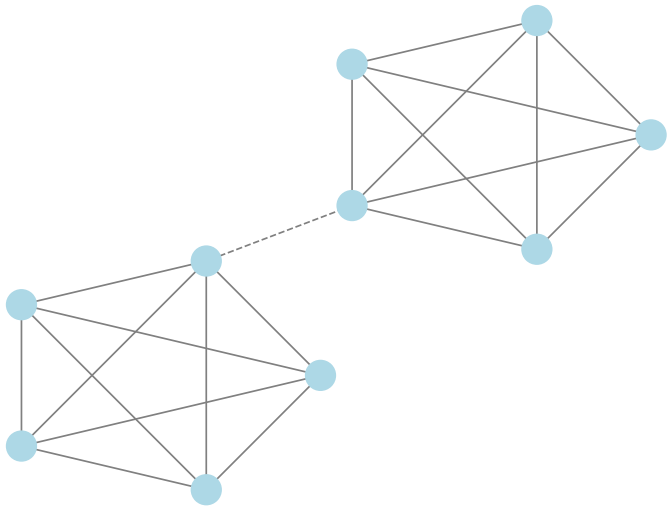


Figure 8.7: Sample System Testing Clusters, where the dashed line represents a denim friendship between two clients

In Figure 8.7 a representation of the client and their friendships can be seen. A solid line represents a friendship between two clients that do not send any deniable messages to each other, and the dashed line represents a friendship between clients that communicate with deniable messages. Clients communicating with deniable messages are not in the same cluster, and the only link between two clusters is a single deniable friendship. We want to see if our implementation of DenIM on SAM successfully hides this connection between clusters.

To verify that our implementation successfully hides the connection we will be using Traffic Analysis Tools[1] developed by a fellow Master's thesis group. The attack used is a form of statistical disclosure attack [21]. The attack tries to match a message that is incoming to the server to an outgoing message such that the destination of the

[1]The tool can be found at https://github.com/cs-25-ds-10-08/Traffic-Analysis-Tools

incoming message can be determined. It works by considering all outgoing messages that leave the server between 0 and ϵ timesteps from the incoming message that you are investigating.

When testing DenIM on SAM with the tool, we used an ϵ value that was recommended in their thesis.

We configured the system testing environment for DenIM on SAM with 60 clients in 20 groups of three clients. Clients in each cluster will only communicate with other clients in the same cluster except one, which will communicate with one from another cluster.

For the sake of comparison, the same setup was used for SAM, but obviously without an option to communicate deniably.

Each test ran for 100 seconds. The total amount of messages sent on DenIM on SAM was 210,000 and the total amount of messages sent on SAM were 251,000 messages. Clients would send a message to a random friend every 200 to 600 milliseconds. The Reply Probability of each client was random between 70% and 95% and the Reply Rate was between 2 and 5. Each client also had a threshold after which they would consider a received message too old to reply. This value was also set randomly between 2 to 5 seconds.

A possible explanation for why the SAM test yielded more messages is that DenIM on SAM deniable messages needs to be piggybacked to reach the receiver, where messages on SAM are instant.

Type	Target	Secret Friend	Guessed Client	Confidence
SAM	192.168.0.16	192.168.0.19	192.168.0.19	100%
SAM	192.168.0.19	192.168.0.16	192.168.0.16	100%
DenIM	172.29.0.34	172.29.0.37	172.29.0.36	100%
DenIM	172.29.0.37	172.29.0.34	172.29.0.19	100%

Table 8.2: Results from traffic analysis using a tool developed by a fellow master thesis group.

We ran the tool on two clients in each run. When testing with SAM, the tool could classify the secret friend with a confidence of 100% in both cases, which is understandable, considering that they communicate with regular messages. However when testing with DenIM on SAM, the tool could not classify any of the secret friends correctly. In the test where the target was 172.29.0.34, the tool guessed that the secret friend was one inside its own cluster with a confidence of 100%. In the last test, something interesting happened. The tool classified the secret friend as someone, the target had not communicated with, also with a high confidence of 100%. The results

from the tests on DenIM on SAM make sense when we also take the results from section 8.3 and Figure 8.6 into account.

8.5 Benchmarking

To ensure that our infrastructure is robust, and can handle many clients we performed benchmarks. In the benchmark we registered and deleted 10.000 clients, to see how many resources our SAM Server would use, as registration is an expensive task to do. We also benchmarked our system testing environment where 500 clients communicated with each from one server instance to another in Strato. The clients would send messages and reply to incoming messages as fast as they possibly could. Our registration benchmark yielded some concerning results in terms of memory usage.

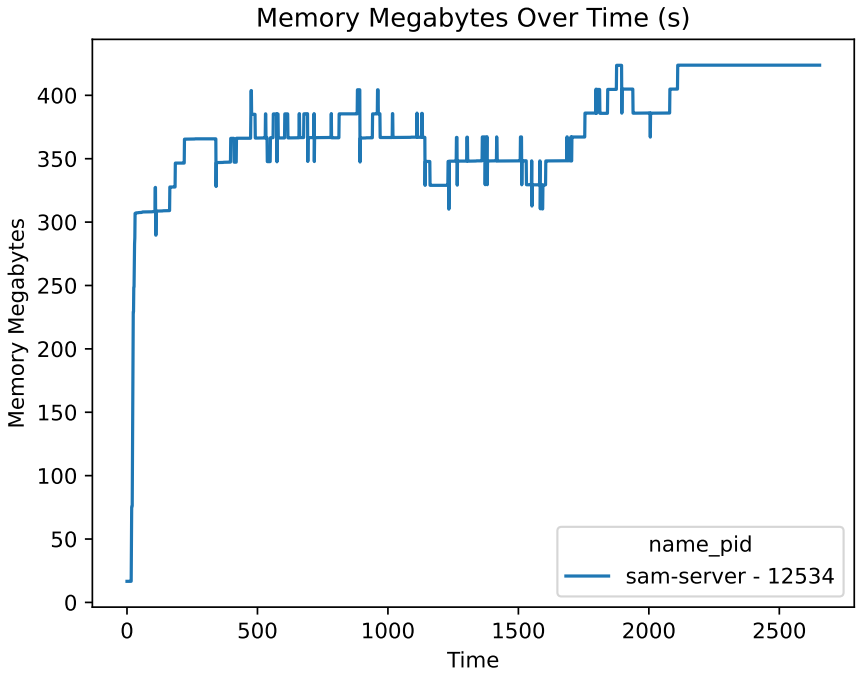


Figure 8.8: Memory usage when registering and deregistering 10,000 clients using default memory allocator.

Figure 8.8 shows the memory usage of the SAM Server in the registration benchmark. Its usage would rise to 423 megabytes of RAM while performing 10,000 registrations and account deletions over 41 minutes. The slow speed was due to running the entire system, including performing all 10,000 registrations, on a single machine. The interesting part is that the memory usage did not fall when all clients were done performing the benchmark. We suspect that the SAM Server has a memory leak

somewhere.

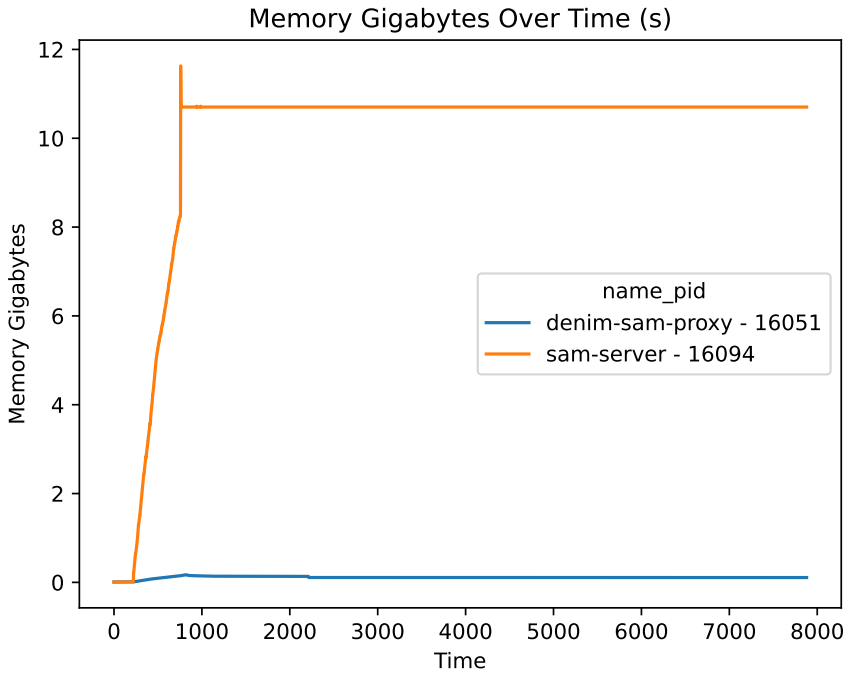


Figure 8.9: Memory usage when 500 clients communicating using default memory allocator

The next benchmark was making 500 clients communicate over two servers. The benchmark, shown in Figure 8.9, confirmed our suspicion of a memory leak. During the benchmark, the SAM server peaked at 12 GB RAM usage in the start of the test and kept a consistently high RAM usage through the entire benchmark. The DenIM Proxy however performed quite well while forwarding messages for 500 clients communicating at a rate of 1 message per millisecond, with a peak of 173 megabytes of RAM usage.

With the memory leak confirmed, we ran the registration benchmark again. But this time using the `tikv-jemallocator`[2] memory allocator, which reduces memory fragmentation and has great scalability in high concurrency use cases[22].

[2]tikv-jemallocator can be found at https://github.com/tikv/jemallocator

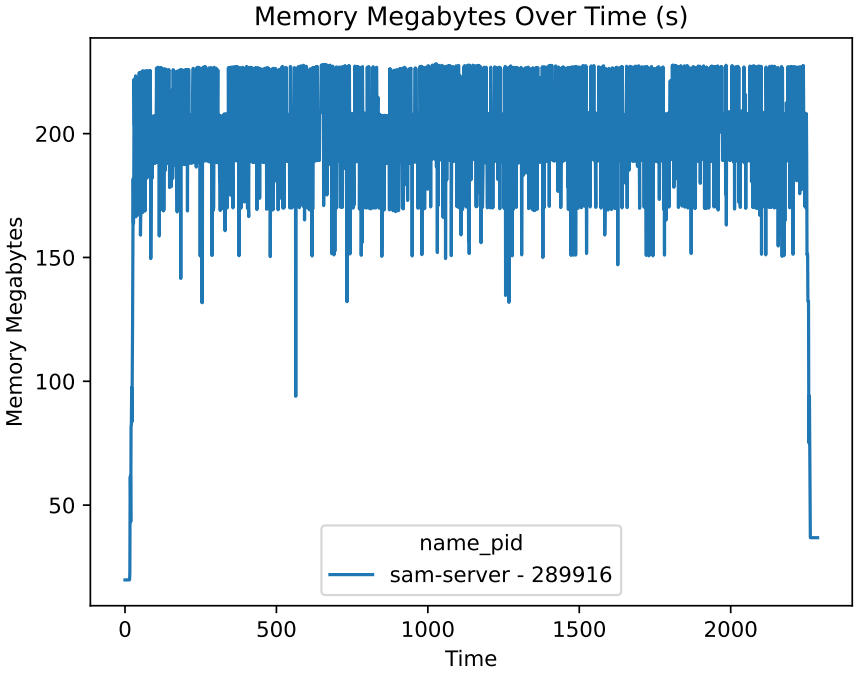


Figure 8.10: Memory usage when registering and deregistering 10.000 clients using tikv-jemallocator memory allocator

As seen in Figure 8.10 this benchmark showed better results. It would reach a maximum of 228 megabytes, and would climb down again after the clients were done performing the tests. But as can be seen in Figure 8.10 it would not go down to its start memory usage again. When we tried to kill the client program and start a new one repeatedly, we found that that the SAM Servers memory usage would climb after each iteration, starting at a higher baseline each run.

8.6 System Requirements

We have almost implemented most of our requirements to the fullest with some of the requirements having minor things missing, in order to be fully implemented.

8.6.1 SAM Instant Messenger Requirements

For SAM, we have implemented persistent storage fully for the SAM client API completing Requirement R-2. The SAM Server has not been implemented with persistent storage for messages, which means that Requirement R-1 is not complete. This also leads to SAM not being scalable, failing Requirement R-3. Our infrastructure is compliant with the Signal Protocol, where our clients use the libsignal library for Rust to

utilize the Signal Protocol, this completing Requirement R-4. All relevant managers from SAM is reused on DenIM on SAM, which completes Requirement R-5. During system testing the SAM Server did not output any errors to the logs, even with 60 clients communicating, which leads us to believe we have successfully implemented Requirement R-6. Table 8.3 shows which of the SAM requirements were satisfied.

Requirement	Satisfied
Requirement R-1	✗
Requirement R-2	✓
Requirement R-3	✗
Requirement R-4	✓
Requirement R-5	✓
Requirement R-6	✓

Table 8.3: Satisfied SAM Instant Messenger Requirements

8.6.2 DenIM on SAM Requirements

Our DenIM on SAM common module contains the `SendingBuffer` which has been tested[3] to ensure that the messages are always $l + l \cdot q + o$ and along with our manual analysis of traffic traces in section 8.3, we are confident that we have satisfied DenIM Requirement D-1.

Our DenIM on SAM client API will not begin to send messages to the server if it has not received a status message from the DenIM on SAM Proxy containing the current q. We have validated this using tests[4]. When clients connects to the DenIM on SAM Proxy, the proxy will immediately send a message with the current q, from this we can confidently say that we have satisfied DenIM Requirement D-2.

We have implemented both a `ReceivingBuffer` and `SendingBuffer` in our DenIM on SAM common module. The `SendingBuffer` and `ReceivingBuffer` have been tested together to assert that the `ReceivingBuffer` can reassemble deniable messages from payloads created by the `SendingBuffer`, even if they are received out of order[5]. With this the DenIM Requirement D-3 and DenIM Requirement D-4 requirements are satisfied.

Our processes for forwarding messages between clients and the SAM Server in our DenIM on SAM Proxy, will forward regular messages immediately before processing any deniable payloads[6], which satisfies DenIM Requirement D-5.

We have implemented an `InMemoryDenimEcPreKeyManager`[7] on the DenIM on SAM Proxy. It takes a cryptographically secure RNG in, that will be used to generate

[3] Test is found in send.rs L259-L317
[4] Found in denim_client.rs L254
[5] Tests are found in buffers.rs L29-L98
[6] Found in proxy.rs L191-L193
[7] Found in keys.rs L117

keys for users with provided seeds, this satisfies DenIM Requirement D-6, DenIM Requirement D-7, DenIM Requirement D-8 and DenIM Requirement D-9.

The `next_key_id`[8] method on the `InMemoryDenimEcPreKeyManager` uses rejection sampling while generating random key IDs, which satisfies DenIM Requirement D-10.

Our analysis in section 8.3 and section 8.4 makes us confident in the satisfactory of DenIM Requirement D-11. Our end-to-end tests cover cases where clients blocks other clients deniably[9], which verifies that DenIM Requirement D-12 has been satisfied.

We have tested ongoing deniable communication in our end-to-end tests, which requires users to successfully exchange keys using DenIM and also send many deniable messages, verifying that DenIM Requirement D-13 and, DenIM Requirement D-15 have been satisfied.

We have not implemented clients being able to upload their own prekeys for DenIM, which means DenIM Requirement D-14 has not been satisfied.

Table 8.3 shows which of the DenIM Protocol Requirements have been satisfied.

Requirement	Satisfied
DenIM Requirement D-1	✓
DenIM Requirement D-2	✓
DenIM Requirement D-3	✓
DenIM Requirement D-4	✓
DenIM Requirement D-5	✓
DenIM Requirement D-6	✓
DenIM Requirement D-7	✓
DenIM Requirement D-8	✓
DenIM Requirement D-9	✓
DenIM Requirement D-10	✓
DenIM Requirement D-11	✓
DenIM Requirement D-12	✓
DenIM Requirement D-13	✓
DenIM Requirement D-14	✗
DenIM Requirement D-15	✓

Table 8.4: Satisfied DenIM Protocol Requirements

We have not implemented DenIM as described by [15] as clients are missing the ability to upload keys to DenIM on SAM, and therefore we have not satisfied Requirement R-7. We have implemented a strategy for marking key IDs as unused when clients have sent a prekey message and the DenIM on SAM Proxy receives it, this can be

[8] Found in keys.rs L198
[9] Test can be found in message.rs L187

seen in `handle_user_message`[10], this satisfies Requirement R-8. Table 8.3 shows the satisfactory of DenIM Implementation Requirements. The requirements not fulfilled will be discussed in chapter 9.

Requirement	Satisfied
Requirement R-7	✗
Requirement R-8	✓

Table 8.5: Satisfied DenIM Implementation Requirements

[10]Found in denim_routes L211

9 Discussion

9.1 System Requirements

This section will discuss the system requirements for SAM and DenIM on SAM that were not fulfilled. The first requirement not fulfilled, is for the server of SAM and DenIM on SAM to have persistent storage and be able to scale in order to serve a lot of users.

9.1.1 Persistent Storage

With the current state of SAM Server, the only thing needed for persistent storage is a persistant `MessageManager`. A solution could be to implement message storage like Signal does it. Signal first stores messages in a Redis cache for up to 10 minutes before it is put into a database for long-term storage[1]. This could be a good way of storing the regular messages. For the deniable messages it is important to choose the fastest storage option to help mitigate potential timing leaks. Using a slow storage option could increase the timing difference between sending a message with dummy padding or with deniable content.

A way to solve this problem could be to use a Redis cache for everything handled by the DenIM Proxy. The deniable buffers of all currently connected users would be loaded into the cache. Then, when the user disconnects, the proxy would put the rest of the content from the deniable buffer that has not yet been delivered into persistent storage. This could be a database. Once SAM and DenIM on SAM has persistent storage, it is also possible scale the server horizontally.

9.1.2 Clients Uploading Their Own Keys

A requirement of DenIM that was not satisfied on DenIM on SAM is giving the client the option to upload their own generated one-time prekeys for deniable sessions. This was not implemented because we determined that the trade-off was not worth it in practice. DenIM assumes that the KDC is not compromised as stated in section 3.2.

[1] line 148-149 in MessagePersister.java

Also, if the client has to upload keys, then the key refill request will take up space in the deniable buffer so that any message the client wants to send afterwards will be delayed even further.

In order to implement this requirement, the DenIM Protocol would have to be extended with a key refill message. This message type would include one or more one-time prekeys. For the server to facilitate this feature, it would need functionality to handle the key refill messages sent by the client. Once a key refill message is received, it would store them in a persistent storage solution. In section 3.1 we described how DenIM on Signal handles generating key IDs without collision. We could also use this same approach. Upon starting new deniable sessions, the server would always prefer using a one-time prekey uploaded by the client instead of a one-time prekey generated by the Key Distribution Center.

9.1.3 Key Agreement Protocol

The DenIM paper describes the X3DH key agreement protocol that Signal uses, and this is the protocol that DenIM modifies slightly by having the KDC generate prekeys on behalf of the user [15].

X3DH is the old key agreement protocol that Signal has since moved on from. Today, Signal uses the *Post Quantum Extended Diffie-Hellman* (PQXDH) protocol which provides protection against "harvest now, decrypt later"-attacks where a future adversary with access to a quantum computer is able to break the encryption on messages collected today [19]. PQXDH extends X3DH with a post quantum prekey which is cryptographically signed. The KDC cannot generate this type of prekey on behalf of the user because this would require the server to have the client's private key for generating the signature and therefore, deniable messages cannot have post quantum security.

If the content of the message itself is so sensitive that the possibility that an attacker with access to a compromised server might one day use a quantum computer to break the encryption of a session that you start today, then DenIM on Signal may not be sufficiently secure. Whether it is more likely for the scenario above or simply for the sender or the recipient to be compromised will determine if DenIM is secure enough.

Regular messages can of course still use PQXDH.

9.1.4 Timing Attacks

Since regular messages are supposed to be delivered instantly but also have a deniable payload attached to it before it can be delivered, DenIM is potentially vulnerable to timing attacks. It may be the case that the time it takes for the server to attach deniable payload is different from the time it takes to attach dummy padding. In

that case, an adversary might be able to deduce that a deniable message was received. Some form of mitigation will be required to ensure that our implementation is not vulnerable to timing attacks.

9.1.5 Reconstructing Deniable Messages

In section 7.2 we described a solution where deniable messages can be reconstructed disregarding what order the DenIM chunks arrives in. However if there are chunks that never arrive, the receiver will hold the rest of the chunks indefinitely. To resolve this problem, each DenIM chunk should have a timestamp so that old messages that were never reconstructed can eventually be evicted.

9.2 Benchmark

The benchmarks of DenIM on SAM showed some subpar results. We discovered that the SAM Server had a big memory leak. However the DenIM Proxy performed well under the benchmarks with a peak of 173 megabytes of RAM usage. We have not located the memory leak in SAM Server.

We could have mitigated this by also making benchmarks for each of the components in the SAM and our DenIM implementations. Then, it would be easier to find the leak, and we might have caught it earlier. Also, it would be useful during development to know that a change in the code would not lead to a performance regression.

We should have used the same strategy for our benchmarks as we did for our testing as described in section 8.1. We tried to identify the source of the memory leak, by profiling our entire SAM Server with heaptrack[2], Valgrind[3] using massif and tokio-console[4], but we did not have enough time to find the issue. If we had done it earlier in the development process we could profile each component separately in isolation, making it easier. For future work on this project, benchmarking should be a high priority to find the memory leak.

9.3 Road To Production

There are still quite some way to have a production ready implementation of SAM and DenIM on SAM, aside from that DenIM on SAM is not scalable yet as described in subsection 9.1.1.

Our system testing environment described in chapter 8 wrote logs that contained warnings and errors. We sometimes encountered bad MACs from libsignal's MAC verification process. These warnings lead to decryption errors as the MAC verification

[2]Heaptrack https://github.com/KDE/heaptrack
[3]Valgrind https://valgrind.org/
[4]tokio-console https://github.com/tokio-rs/console

failed. This is a vital error where the cause would have to be looked further into. However, this was not a common sight as we only got 177 of them in total from both of our tests in section 8.4 where there were sent a total of 462,000 messages. This means it only occurs 0.00038% of the time, which is very low, but it still needs to be addressed. The tests for DenIM on SAM in section 8.4 yielded 13 errors where a client would send a message using the same prekey ID when it was already pending deletion. Additionally, we got 2 errors where a client would simply completely fail to decrypt a message. We also got 3 errors where a client failed to process a message because our `PreKeyStore` failed to save a prekey.

The server infrastructure is also missing some vital parts before it can be safely deployed.

- Rate limiting: We have not implemented any rate limiting of any kind on either the SAM Server and DenIM Proxy.
- WebSocket timeouts: Clients WebSocket connects to the server does not timeout, which is needed so the server does not use resources to keep a WebSocket connection alive when a client is not active.
- User reporting feature: Our implementation lacks a report and blocking feature of regular messages, which means users can be subject to spam.
- Scalability: Missing implementations of persistent managers yields the infrastructure unsuitable for real world usage.

The infrastructure is also missing popular features from other *Instant Messaging* (IM) services such as group chats and file sharing.

9.4 Architecture

As explained in section 7.1, the SAM Server does not know that it is communicating with a DenIM Proxy rather than a normal SAM client. This shows that it could be possible to implement a DenIM Proxy on more established services such as Signal. The DenIM Proxy implementation relies on some SAM Managers, such as the `Key`, `Device` and `AccountManager`s. This was done to validate that clients were actually trying to send *deniable* messages to a valid receiver and to reuse signed prekeys and identity keys from SAM. This could be difficult to do if it was to be done on the official Signal Server as it requires access to Signals databases. Another method to be completely independent from Signal, is to have a double identity, meaning that users would have to upload a separate identity key and signed prekeys or the same to the DenIM Proxy. Using this approach would also mean that the DenIM Proxy would have no way of validating receivers. Unless users would register through the DenIM Proxy, so it could obtain account IDs and device IDs, or users uploading them to the

DenIM Proxy whenever they make an update to their account. These requests should also be carried out through the DenIM Protocol, to keep them deniable.

9.5 Updating Seed in DenIM

In DenIM the user only uploads their seed once during registration. But if somehow their seed has been compromised, it should be possible for a user to update their seed. Updating a users seed would allow the proxy to generate one time prekeys using the new seed, and thereby new conversations. But scenarios can happen, where an old seed would have to be used in order to preserve anonymity.

Consider the example where Alice and Bob, do not know each other, and Alice wants to update her seed. If Bob starts a conversation with Alice and gets a prekey bundle based on the past seeds, the conversation would have to be started based on the past seed. Even if Bob's prekey message reaches the proxy after it has received Alice's new seeds, the proxy cannot force Bob to create a new prekey message using a prekey bundle based on the new seeds, as this would reveal that Alice has performed a seed update. The proxy would have to check what seed each prekey message was created on, and send that information to Alice so she knows which seed she has to use to generate the one time prekey. Prekeys could have a time-to-live such that once you have requested a prekey, you must use it before a specified time. This means that both the client and proxy have to keep the old seeds until all prekeys based on a specific seed have been used in a prekey message or until a the prekey has timed out, as new conversations could have been started using them.

9.6 Threat Model

The threat model DenIM uses was presented earlier in section 3.2. Here DenIM assumes that the adversary cannot compromise the internal state of the server. The need for this assumption gives an indication to how much responsibility is given to the server.

The way that DenIM works, if the server is compromised, an adversary would be able to see who is communicating with whom, completely defeating the purpose of DenIM. Also, the adversary would have access to a third of the key material that is used for generating the shared secret key that the encryption session is based on. We cannot rule out the possibility of an attack where an adversary exploits the reduced entropy, but this is the same as if a Signal Server was compromised. The only difference is that the adversary would have the seed for generating new one-time prekeys.

10 | Conclusion

As no popular *Instant Messaging* (IM) service fully supports metadata privacy, this Master's thesis set out to implement a transport privacy protocol to hide metadata from messages. After looking at different types of transport privacy protocols, the choice landed on the DenIM protocol which is a kind of delay protocol. It is a hybrid protocol, meaning it supports regular and *deniable* messages. It works by piggybacking the deniable messages to the regular messages upon transmission, thereby hiding the deniable messages in the traffic. As the Signal Protocol provides end-to-end encryption and is widely used on popular IM services, this project decided to implement DenIM on top of the Signal Protocol. Without having access to an IM service that is using the Signal Protocol to implement DenIM onto, such IM service had to be built from scratch in addition to a DenIM implementation. These discoveries lead us to the following problem:

> *How can we implement an instant messaging service and client library that uses the DenIM protocol on top of the Signal Protocol and is ready for real-world use?*

With the implementation of the instant messenger using the Signal Protocol, SAM, the next objective was to implement DenIM on top, creating DenIM on SAM. With the implementation of DenIM, most of the components from SAM was reused by using a proxy that handled all *deniable* messages.

An anonymity assurance analysis was conducted where the traffic was observed with and without deniable communication. When observing the traffic there was no difference between a regular message with deniable chunks and one with dummy padding, and we could not correlate deniable communication. Additionally a type of statistical disclosure attack was conducted on DenIM on SAM. Results showed that the tool could not correlate deniable communication, but it could correlate communication between clients that were not using DenIM. We also evaluated the DenIM implementation on a list of system requirements set before implementation. Here we fulfilled the requirement that the deniable payload of all messages always reached $l \cdot q$. Furthermore DenIM on SAM does not have the functionality for the client to upload

their own deniable keys to the KDC. However, the functionality that DenIM provide works without this requirement fulfilled.

We can conclude that we have a DenIM implementation that does not fully live up to the specification. However, a client can send deniable messages that are not identifiable over the traffic. Furthermore, DenIM on SAM is resilient against a form of a statistical disclosure attack. The implementation is not ready for production, because of missing persistent storage solutions, rate limiting, client timeouts and memory leaks in the server. Thereby this project achieves its primary goal, enabling deniable messaging without revealing metadata.

11 | Future Work

As there is room for improvement on DenIM on SAM, this chapter will describe how DenIM on SAM can be improved.

Many of the sections from chapter 9 require some future work. The tasks that are required for a fully production ready DenIM on SAM Instant Messenger service, are listed below.

- **Benchmarks and stress tests**: The infrastructure should undergo more rigorous benchmarks and stress testing, before it can be deemed ready for production, as there were some errors and warnings while we did system testing as mentioned in section 9.3 and section 9.2
- **Bug Fixes**: In section 9.3, we described that the DenIM on SAM infrastructure experienced errors and warnings, where some lead to decryption errors. The source of the errors and warnings would have to identified and fixed.
- **Timing attack mitigation strategy for dummy padding**: as described in subsection 9.1.4 there needs to be some sort of strategy so that an adversary cannot tell when the DenIM Proxy is creating denim chunks or dummy chunks to piggy back on regular messages using timing attacks.
- **DenIM chunk lifetime limits**: As described in section 7.2, clients and the DenIM Proxy should have a strategy for when to delete DenIM chunks if the remaining chunks needed for the message never arrive. This would mitigate unnecessary storage of chunks that are essentially orphans.
- **Persistent Message Manager**: Mentioned in subsection 9.1.1 the message manager should be persistent as with the other managers, but also have a short time storage for online clients, for faster query times.
- **Persistent Chunking Buffers**: Also mentioned in section 7.2, the buffers holding chunks should also have a persistent storage and fast storage for online users to mitigate timing attacks as mentioned in subsection 9.1.1.
- **Seed Updates**: As described in section 9.5, users need to update their seed if

their seed has been compromised. This goes beyond DenIM's specification.

- **Key Uploads**: Users need to be able to upload their own keys to DenIM to support the specification, and DenIM Requirement D-14.
- **Rate limiting**: To be production ready the infrastructure would also need to have some kind of rate limiting as described in section 9.3.
- **WebSocket Timeouts**: Inactive clients does not timeout of their WebSocket connection, this is is also needed to be implemented to satisfy section 9.3.
- **User reporting**: Users cannot report or block regular users, which can make them subject to spam as mentioned in section 9.3.

Bibliography

[1] Radicati Group. *Instant Messaging Statistics Report, 2021-2025: Executive Summary*. Accessed: 2025-05-28. 2020. URL: https://www.radicati.com/wp/wp-content/uploads/2020/12/Instant-Messaging-Statistics-Report-2021-2025-Executive-Summary.pdf.

[2] WhatsApp. *WhatsApp Encryption Overview*. Tech. rep. WhatsApp, 2024. URL: https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf.

[3] Meta. *Messenger End-to-End Encryption Overview*. Tech. rep. Version 1. Meta Platforms, Inc., Dec. 2023. URL: https://engineering.fb.com/wp-content/uploads/2023/12/MessengerEnd-to-EndEncryptionOverview_12-6-2023.pdf.

[4] 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.

[5] Marlon Cordeiro Domenech, André Ricardo Abed Grégio, and Luis Carlos Erpen de Bona. "On Metadata Privacy in Instant Messaging". In: *2022 IEEE Symposium on Computers and Communications (ISCC)*. 2022, pp. 1–7. DOI: 10.1109/ISCC55528.2022.9912901.

[6] Alena Birrer, Danya He, and Natascha Just. "The state is watching youA cross-national comparison of data retention in Europe". In: *Telecommunications Policy* 47.4 (2023), p. 102542. ISSN: 0308-5961. DOI: https://doi.org/10.1016/j.telpol.2023.102542. URL: https://www.sciencedirect.com/science/article/pii/S0308596123000538.

[7] Jelle van den Hooff et al. "Vuvuzela: scalable private messaging resistant to traffic analysis". In: *Proceedings of the 25th Symposium on Operating Systems Principles, SOSP 2015, Monterey, CA, USA, October 4-7, 2015*. Ed. by Ethan L. Miller and Steven Hand. ACM, 2015, pp. 137–152. DOI: 10.1145/2815400.2815417. URL: https://doi.org/10.1145/2815400.2815417.

[8] Ania M. Piotrowska et al. "The Loopix Anonymity System". In: *CoRR* abs/1703.00536 (2017). arXiv: 1703.00536. URL: http://arxiv.org/abs/1703.00536.

[9] David Lazar, Yossi Gilad, and Nickolai Zeldovich. "Karaoke: Distributed Private Messaging Immune to Passive Traffic Analysis". In: *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*. Ed. by Andrea C. Arpaci-Dusseau and Geoff Voelker. USENIX Association, 2018, pp. 711–725. URL: https://www.usenix.org/conference/osdi18/presentation/lazar.

[10] 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/.

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

[12] 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*. [Accessed 2025-05-30]. The Internet Society, 2021. URL: https://www.ndss-symposium.org/ndss-paper/improving-signals-sealed-sender/.

[13] Yossi Gilad. "Metadata-private communication for the 99%". In: *Commun. ACM* 62.9 (2019), pp. 86–93. DOI: 10.1145/3338537. URL: https://doi.org/10.1145/3338537.

[14] Amir Houmansadr, Chad Brubaker, and Vitaly Shmatikov. "The Parrot Is Dead: Observing Unobservable Network Communications". In: May 2013, pp. 65–79. ISBN: 978-1-4673-6166-8. DOI: 10.1109/SP.2013.14.

[15] 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.

[16] Andreas Knudsen Alstrup et al. *Signal Implementation for Testing Metadata Privacy Protocols*. ID: alma9921964544205762. 2025.

[17] Meredith Whittaker and Joshua Lund. *Privacy is Priceless, but Signal is Expensive*. URL: https://signal.org/blog/signal-is-expensive/ (visited on 06/04/2024).

[18] Moxie Marlinspike and Trevor Perrin. "The X3DH Key Agreement Protocol". In: *Applied Sciences* (2016). URL: https://signal.org/docs/specifications/x3dh/x3dh.pdf.

[19] Signal. *The PQXDH Agreement Protocol*. https://signal.org/docs/specifications/pqxdh/. [Accessed: 2025-19-5].

[20] Google. *Protobuf Encoding*. Accessed: 06-05-2025. URL: https://protobuf.dev/programming-guides/encoding/https://protobuf.dev/programming-guides/encoding/.

[21] Mikkel Boje Larsen and Mads Møller Kristensen. *Traffic Analysis Tool and The Deniable Disclosure Attack*. 2025.

[22] Salvo Project. *Rust Memory Allocator Alternatives - Salvo*.

A | Appendix

A.1 Anonymity Assurance

The following appendices shows what clients and proxy done in the Anonymity Assurance program. the following symbols will be seen in the listings:

- `A -(X)-> B`: A sent X bytes to B
- `A <-(X)- B`: A recieved X bytes from B through the server.
- `A <-(KEY)- B`: A received a deniable key from B
- `A <-(X)- B DENIM`: A received X deniable bytes from B,

```
11 |SCENE| Alice -(400)-> Bob
11 |SCENE| REGULAR MESSAGE SIZE: 582
11 |SCENE| DENIM MESSAGE SIZE: 1192
11 |SCENE| MESSAGE SIZE: 1195

11 |PROXY| Received Message From Alice
11 |PROXY| Received Key Request
11 |PROXY| Received Message From Bob

11 |SCENE| Bob <-(400)- Alice

11 |PROXY| Enqueued Key Response
```

Listing A.1: Logs from server and clients of Alice performing Anonymity Assurance Step AA-1

```
16 |SCENE| Bob -(450)-> Alice
16 |SCENE| REGULAR MESSAGE SIZE: 582
16 |SCENE| DENIM MESSAGE SIZE: 1192
16 |SCENE| MESSAGE SIZE: 1195

16 |PROXY| Received Message From Bob
16 |PROXY| Received Message From Alice

16 |SCENE| Alice <-(KEY)- Dorothy
16 |SCENE| Alice <-(450)- Bob
```

Listing A.2: Logs from server and clients of the server performing Anonymity Assurance Step AA-2

```
21 |SCENE| Alice -(400)-> Bob
21 |SCENE| REGULAR MESSAGE SIZE: 582
21 |SCENE| DENIM MESSAGE SIZE: 1192
21 |SCENE| MESSAGE SIZE: 1195

21 |PROXY| Received Message From Alice
21 |PROXY| Received User Message Request
21 |PROXY| Enqueued Deniable Message
21 |PROXY| Received Message From Bob

21 |SCENE| Bob <-(400)- Alice
```

Listing A.3: Logs from server and clients of Alice performing Anonymity Assurance Step AA-3

```
26 |SCENE| Charlie -(500)-> Dorothy
26 |SCENE| REGULAR MESSAGE SIZE: 742
26 |SCENE| DENIM MESSAGE SIZE: 1512
26 |SCENE| MESSAGE SIZE: 1515

26 |PROXY| Received Message From Charlie
26 |PROXY| Received Message From Dorothy

26 |SCENE| Dorothy <-(500)- Charlie
```

Listing A.4: Logs from server and clients of Charlie performing Anonymity Assurance Step AA-4

```
31 |SCENE| Dorothy <-(200)- Alice DENIM
```

Listing A.5: Logs from server and clients of Dorothy performing Anonymity Assurance Step AA-5

```
36 |SCENE| Dorothy -(550)-> Charlie
36 |SCENE| REGULAR MESSAGE SIZE: 742
36 |SCENE| DENIM MESSAGE SIZE: 1512
36 |SCENE| MESSAGE SIZE: 1515

36 |PROXY| Received Message From Dorothy
36 |PROXY| Received User Message Request
36 |PROXY| Enqueued Deniable Message
36 |PROXY| Received Message From Charlie

36 |SCENE| Charlie <-(550)- Dorothy
```

Listing A.6: Logs from server and clients of Dorothy performing Anonymity Assurance Step AA-6

```
41 |SCENE| Bob -(450)-> Alice
41 |SCENE| REGULAR MESSAGE SIZE: 582
41 |SCENE| DENIM MESSAGE SIZE: 1192
41 |SCENE| MESSAGE SIZE: 1195

41 |PROXY| Received Message From Bob
41 |PROXY| Received Message From Alice

41 |SCENE| Alice <-(450)- Bob
```

Listing A.7: Logs from server and clients of Bob performing Anonymity Assurance Step AA-7

```
46 |SCENE| Alice <-(200)- Dorothy DENIM
```

Listing A.8: Logs from server and clients of Alice performing Anonymity Assurance Step AA-8

```
0 |SCENE| Alice -(400)-> Bob
0 |SCENE| REGULAR MESSAGE SIZE: 582
0 |SCENE| DENIM MESSAGE SIZE: 1192
0 |SCENE| MESSAGE SIZE: 1195

0 |PROXY| Received Message From Alice
0 |PROXY| Received Message From Bob

0 |SCENE| Bob <-(400)- Alice
```

Listing A.9: Logs from Alice sending a message to bob containing no deniable messages

```
Frame 969: 1269 bytes on wire (10152 bits), 1269 bytes
   ↪ captured (10152 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00),
   ↪ Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst:
   ↪ 127.0.0.1
Transmission Control Protocol, Src Port: 43144, Dst Port:
   ↪ 9443, Seq: 6151, Ack: 2862, Len: 1203
WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0010 = Opcode: Binary (2)
    1... .... = Mask: True
    .111 1110 = Payload length: 126 Extended Payload
   ↪ Length (16 bits)
    Extended Payload length (16 bits): 1195
    Masking-Key: 42a0f98c
    Masked payload
    Payload
Data (1195 bytes)
    Data []: 0aa809....
    [Length: 1195]
```

Listing A.10: Alice's WebSocket packet without TLS containing a ciphertext for Bob and a deniable ciphertext for Dorothy

```
Frame 1105: 1269 bytes on wire (10152 bits), 1269 bytes
   ↪ captured (10152 bits) on interface lo, id 0
Ethernet II, Src: 00:00:00_00:00:00 (00:00:00:00:00:00),
   ↪ Dst: 00:00:00_00:00:00 (00:00:00:00:00:00)
Internet Protocol Version 4, Src: 127.0.0.1, Dst:
   ↪ 127.0.0.1
Transmission Control Protocol, Src Port: 43144, Dst Port:
   ↪ 9443, Seq: 7430, Ack: 4203, Len: 1203
WebSocket
    1... .... = Fin: True
    .000 .... = Reserved: 0x0
    .... 0010 = Opcode: Binary (2)
    1... .... = Mask: True
    .111 1110 = Payload length: 126 Extended Payload
   ↪ Length (16 bits)
    Extended Payload length (16 bits): 1195
    Masking-Key: 09d1e07d
    Masked payload
    Payload
Data (1195 bytes)
    Data []: 0aa809....
    [Length: 1195]
```

Listing A.11: Alice's WebSocket packet without TLS containing a ciphertext for Bob and DenIM garbage