MA ICR - Mini-project

Jonas Flückiger

December 20, 2025

1 Introduction

In the context of the class "Industrial Cryptography", students are tasked with developing a small cryptographic application. For the class of 2025, the goal of the project was to develop a "time-capsule" application, a service where users could send each other encrypted files, that would unlock after a chosen date and time. This report describes the submission of Jonas Flückiger, "timelock", available at https://github.com/joflucki/timelock. The project follows a client-server architecture, and is written in Rust. The cryptography library "libsodium" was used for cryptographic primitives, and integrated in the Rust code using FFI bindings available in the libsodium-sys-stable crate. The installation is described in the repo's README file, and the documentation of the CLI tool can be accessed using the `--help` parameter. Cryptographic functions can be found in the respective "crypto" module of each crate. All communications are carried out over TLS, using the native TLS implementation of each platform, thanks to the native-tls crate.

2 Notation

2.1 Keys

- P_i: **P**rivate key of user i
- U_i: P**u**blic key of user i
- S_i: **S**ecret key
- S_{ab}: **S**hared key of users a and b

2.2 Identifiers

- s: **S**erver
- c: **C**lient, sender
- r: Client, **r**ecipient
- m: **M**essage
- I_s: ID of user/message s

2.3 Functions

- $H_a(v)$: Compute hash of value v with salt a
- $E_S^k(m)$: Encrypt message m using symmetric key S and nonce k
- $D_S^k(m)$: Decrypt message m using symmetric key S and nonce k
- $HKDF(S)$: Derive key from master key S
- $DH(P_c, U_r)$: Establish shared key (Diffie-Hellman) using private key P_c and public key U_r.
- $MAC_S(M)$: Compute MAC tag of message M using key S

2.4 Misc

- τ: MAC tag
- t: Timestamp
- k: Nonce
- a: Salt

3 Key structure

The keys are built using the following structure:

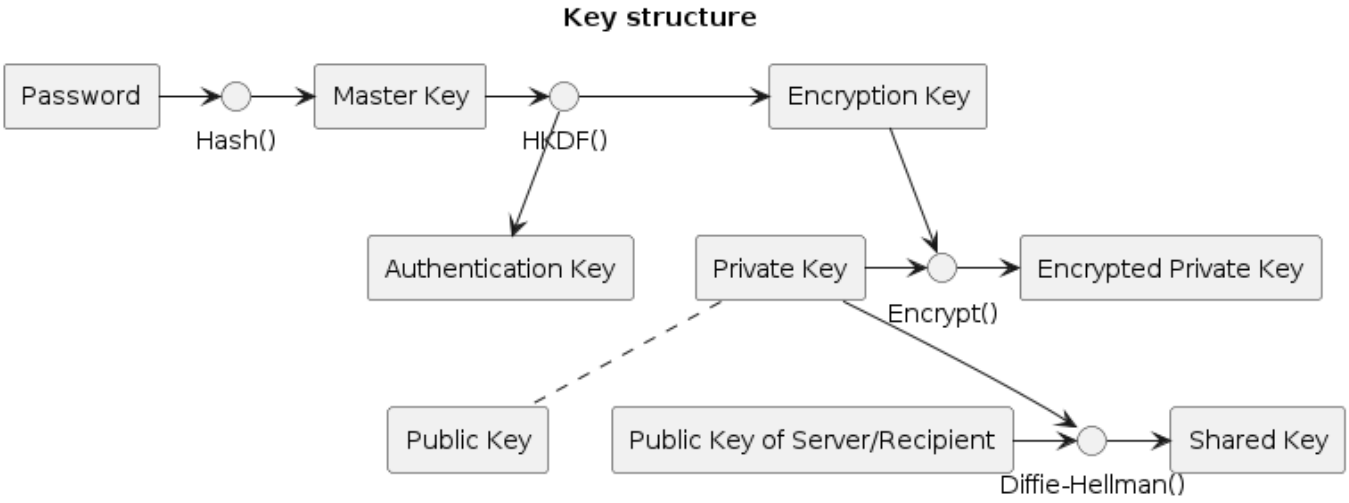

Figure 1: "Key structure"

4 Protocol

All exchanged messages are transmitted over TLS, which ensures confidentiality, integrity, and server authentication.

4.1 Identifying

Identification is the process of creating a user account. In that scenario, the user uses its password to derive a master, as well a set of derived sub-keys. Then they initialize a cryptographic keypair locally, and sends their username, the public key of that key pair, their "authentication" sub key, the private key of that key pair (encrypted), and the nonce and salt used during the process. And this data is stored by the server for future authentication.

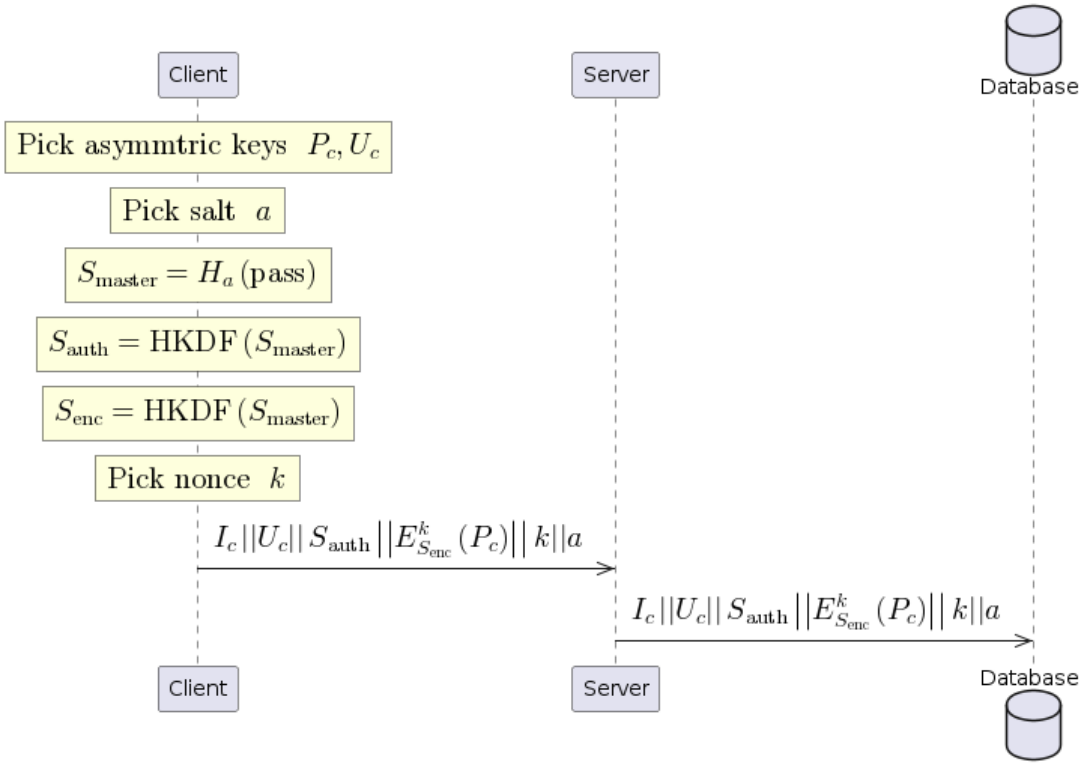

Figure 2: "Identifying"

4.2 Retrieving keys

The process of key retrieval is performed when a user switches devices and has to retrieve their credentials. In that process, the user re-computes their key suite, based on their password and salt. Then, using their "authentication" sub-key, the user can authenticate and retrieve their remaining credentials.

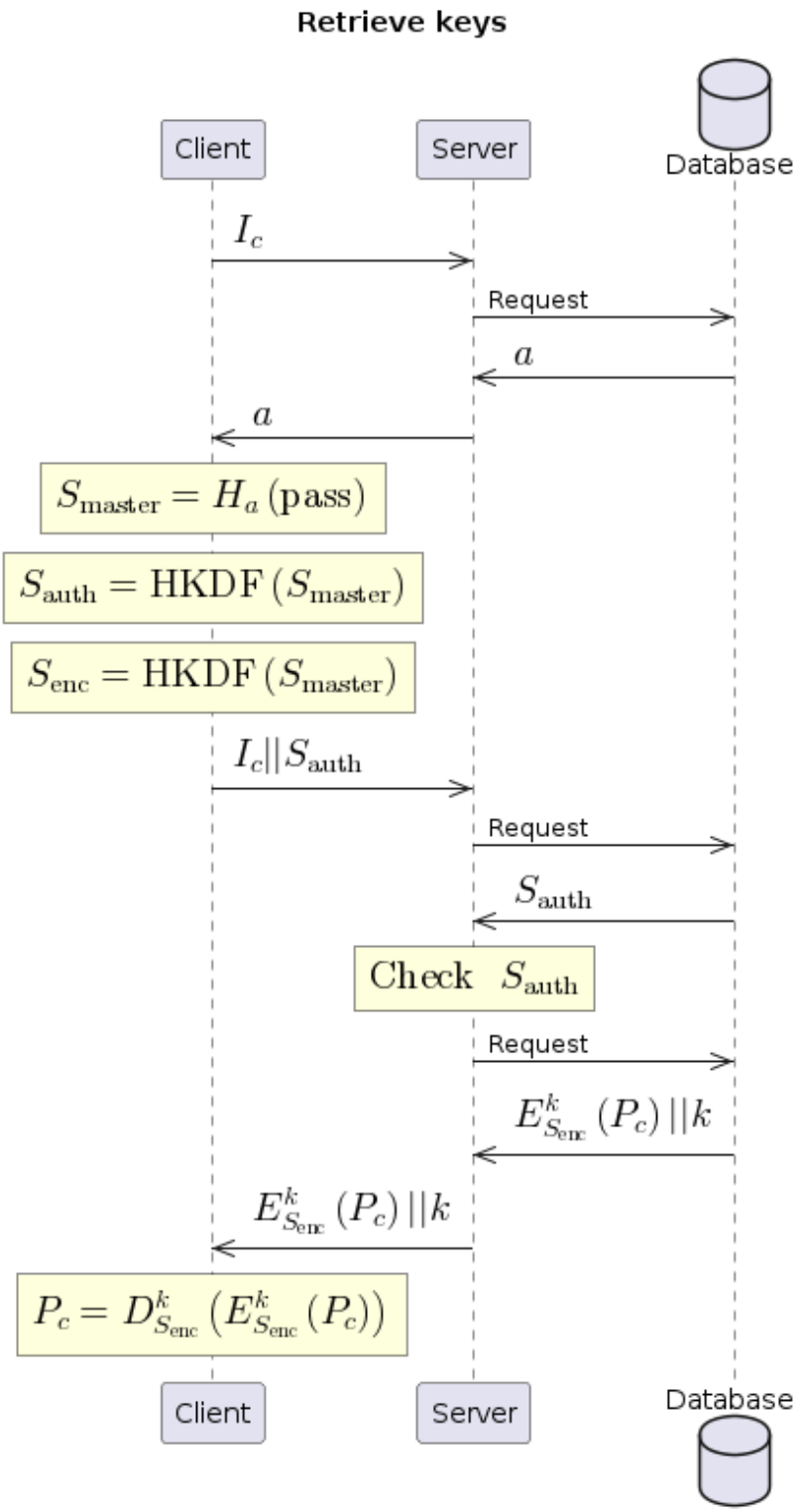

Figure 3: "Retrieving keys"

4.3 Resetting password

Password reset is used when a user whishes to change their password. In order to do so, the (already logged in) computes a new key suite using their new password and a new salt, then encrypts their private key using the newly derived "encryption" sub key, and sends everything to the server.

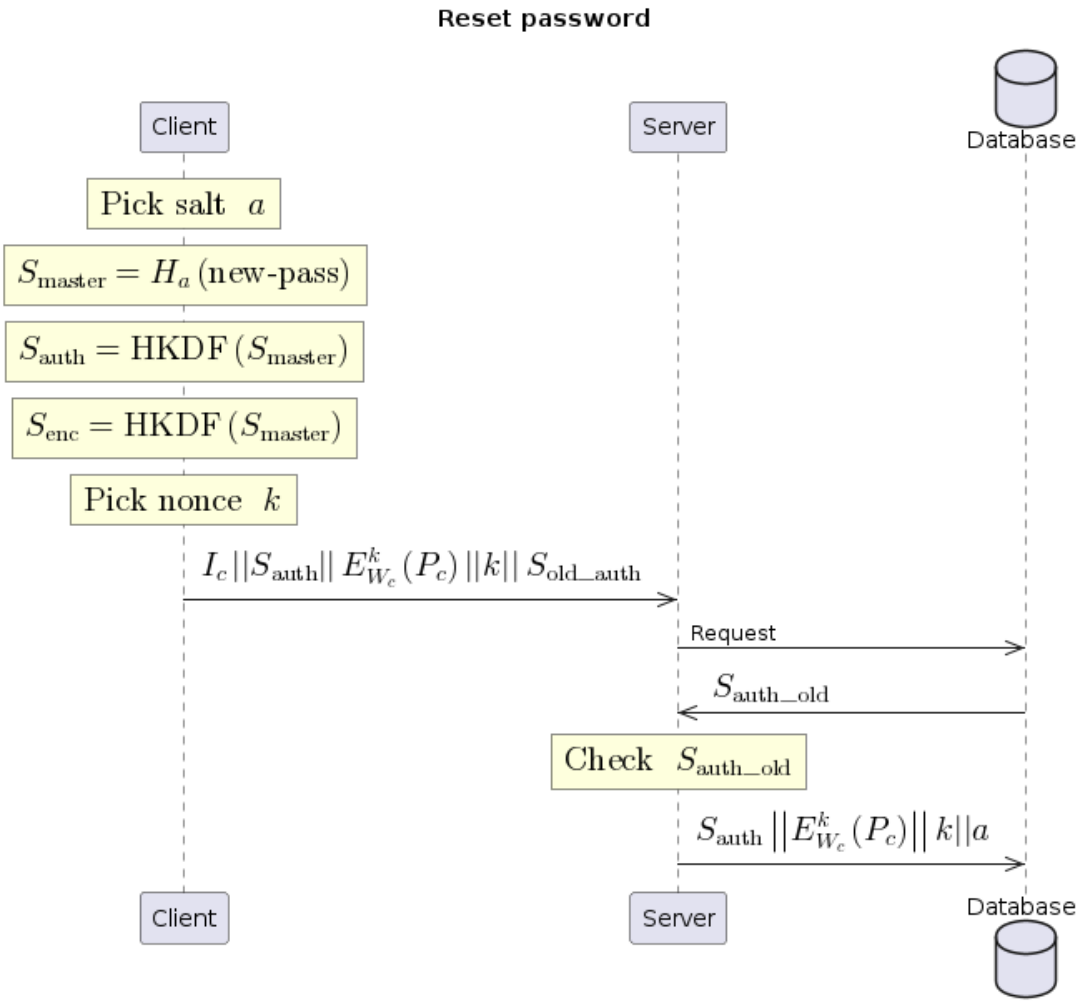

Figure 4: "Resetting the password"

4.4 Sending messages

To send a message to a recipient, the user first fetches the public key of said recipient. Then, the message is encrypted using a randomly generated secret key S_m. Then, this secret key is encrypted using the shared sender-recipient symmetric key S_{cr}. A MAC tag is generated for both the one-time key and the data itself, to allow for separate integrity/authentication check. Those two elements, along with some metadata, are all bundled together and sent to the server.

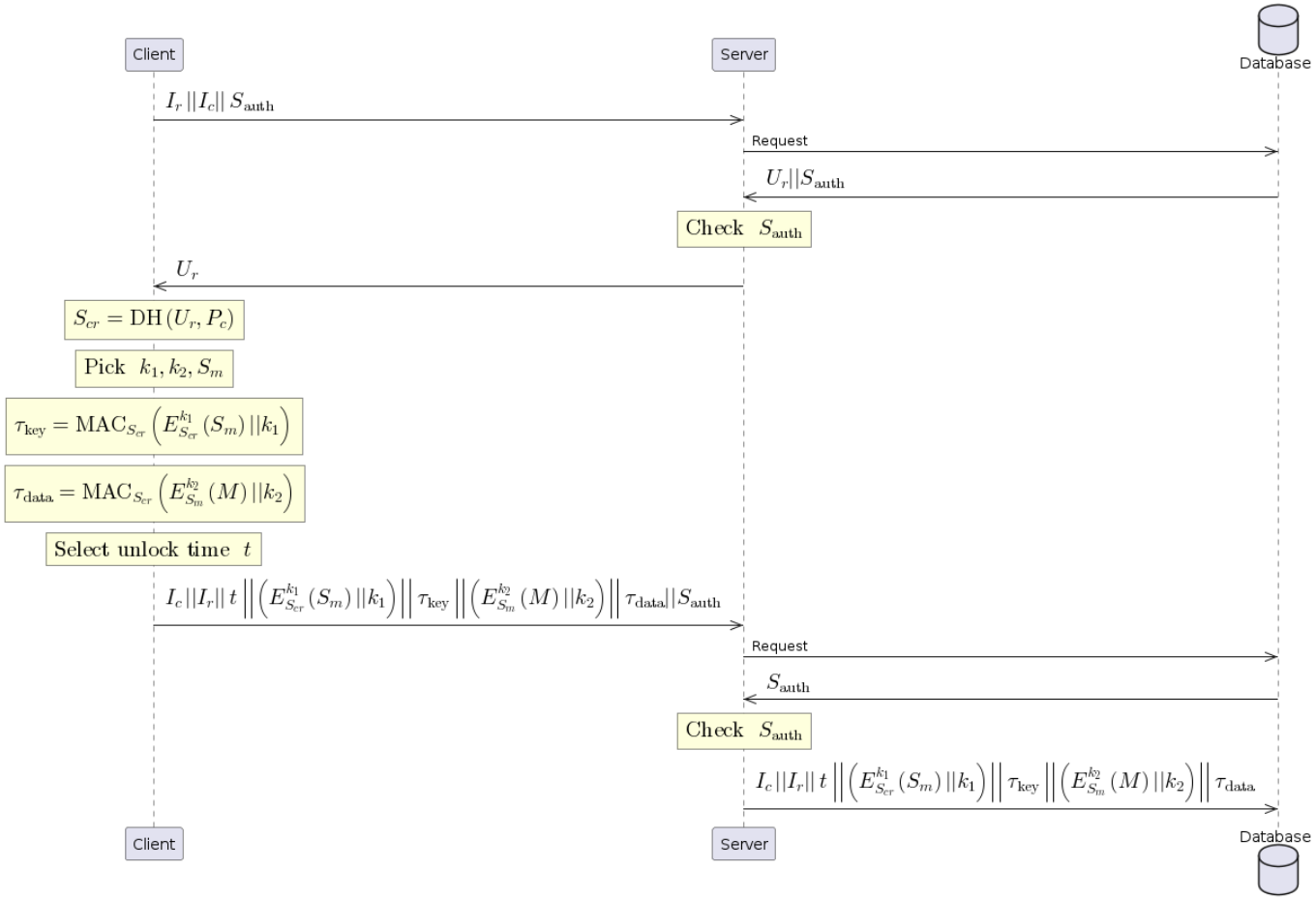

Figure 5: "Sending a message"

4.5 Listing messages

In order to download and unlock messages, the user has to be able to list messages sent to them. The user sends a simple request and includes their authentication key. If its verified, the server responds with a preview of the message the user has received, with a message ID.

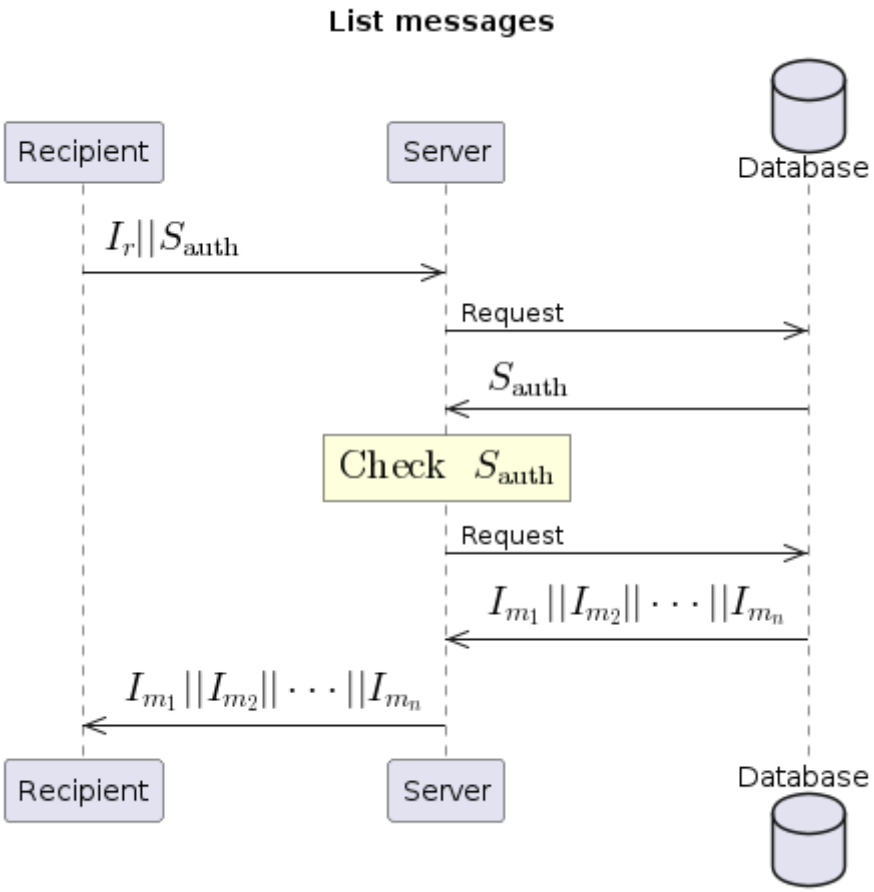

Figure 6: "Listing all messages"

4.6 Downloading messages

Messages are downloaded while still being encrypted by the one-time key generated by the sender. In order to download a message, the user sends a request with their authentication key appended, and if the key is verified, the server responds with sender public key, encrypted message data, nonce, and MAC. The user can they compute the sender-recipient shared key and verify that the message is authentic and has not been tampered with.

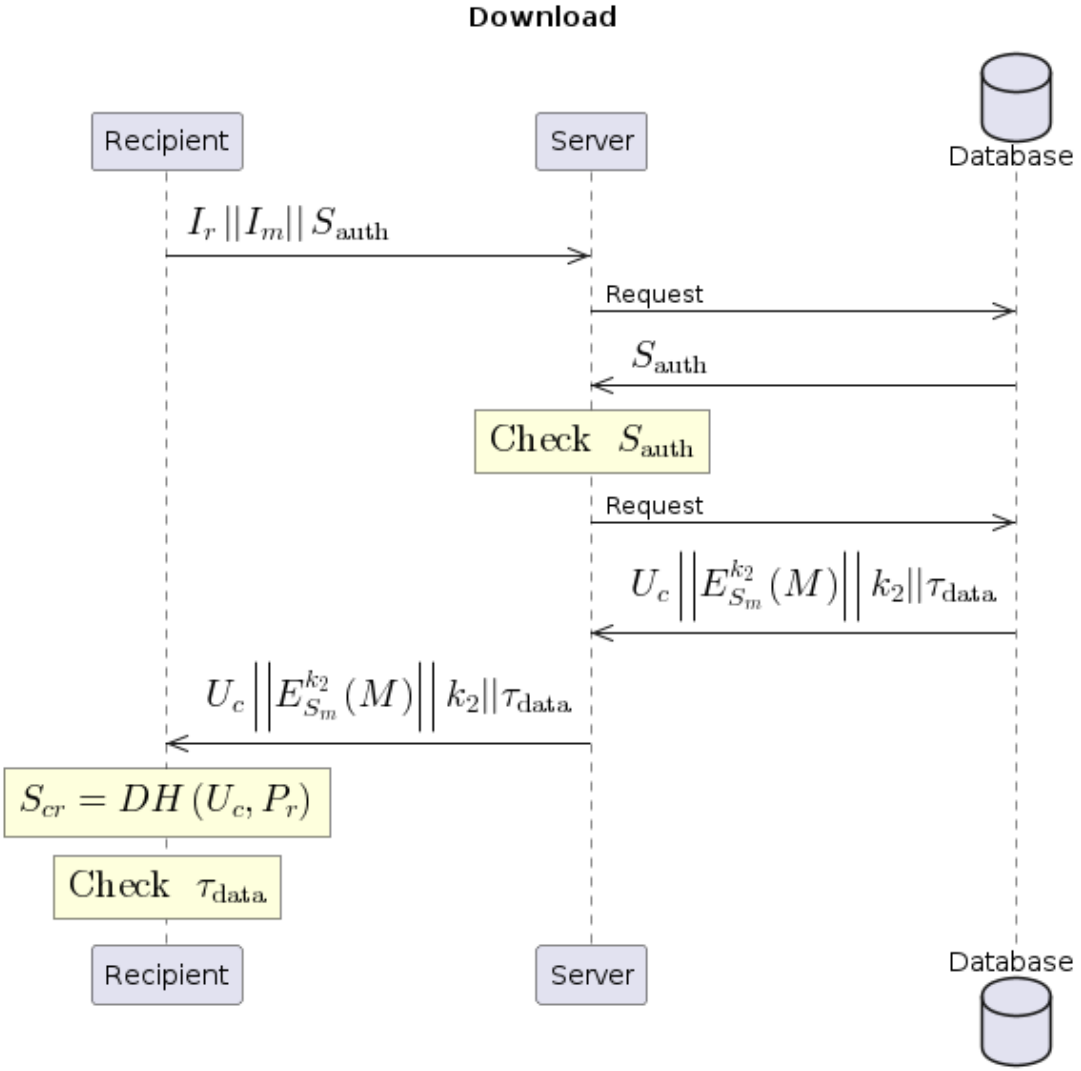

Figure 7: "Downloading a message"

4.7 Unlocking messages

In order to unlock a message, the user makes a similar request to the server as when downloading the encrypted data, but instead receives the sender public key, encrypted one-time key, nonce, and MAC. The user can then decrypt the one-time key using the sender-recipient shared key, and decrypt the message data using the one-time key.

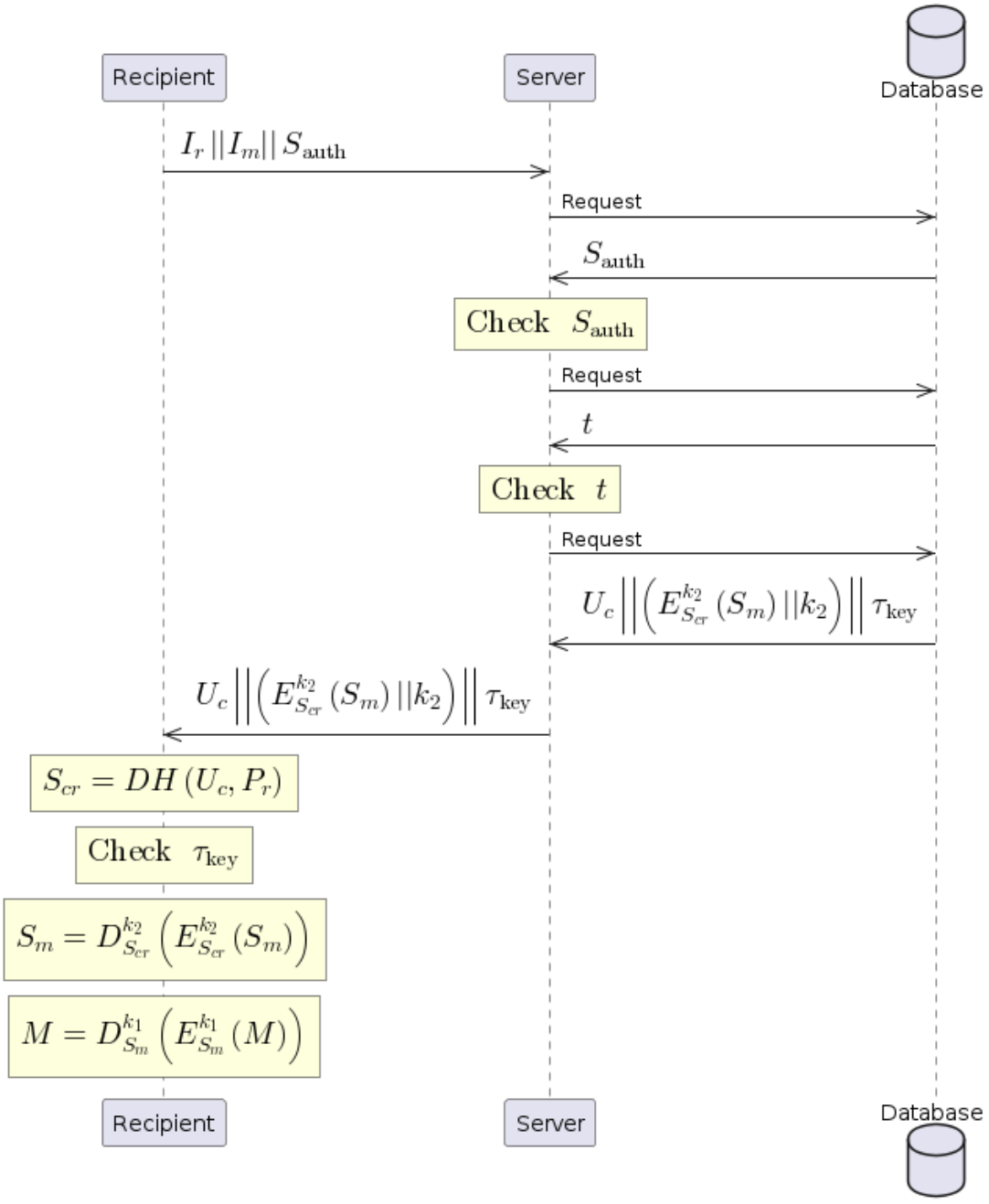

Figure 8: "Unlocking a message"

5 Cryptographic parameters

- Hash function: Argon2id
- Salt: 128 bits
- Symmetric encryption: XChaCha20
- Authentication: HMAC-512-256
- Asymmetric keys: 256 bits
- Symmetric keys: 256 bits
- Diffie-Hellman: ECDH X25519
- KDF: HKDF-SHA256
- Nonce: 192 bits

The encryption function XChaCha20 was used for multiple reasons. First, it allows to encrypt large files ($2^{64} - 1$ bytes theoretically), and the 192 bit nonces allow for random nonces to be used, which simplifies the application and is already implemented in libsodium.

This system would result in an overall security of 128 bits, bound by the weakest link in the chain, the X25519 Diffie-Hellman exchange, which offers 128 bits of security for 256 bit keys.