CAA 24-25 : Mini-project

Nathan Rayburn

January 01, 2025

Abstract

This project focuses on designing and implementing a secure application that enables users to send messages that can only be decrypted after a specified date in the future. The system supports user authentication through simple username and password credentials, with functionalities allowing users to log in from any device, change their password, and send confidential messages to other users. The confidentiality of messages is ensured such that only the intended recipient can access them after the specified unlock date.

To achieve this, the system assumes an honest-but-curious server model that adheres to the protocol but may attempt to compromise the confidentiality of messages. Users can download encrypted messages before their unlock date, with a final interaction required to decrypt them once the date is reached.

The application includes a typical interface for core functionalities: sending messages, reading received messages (with clear visibility of unlock dates and senders), and changing passwords. Robust cryptographic mechanisms are integrated to ensure non-repudiation by senders and protect against unauthorized access, even from the server. This project highlights secure communication, scalability, and user-centric design while addressing practical challenges in delayed message decryption

1 Architecture

The program's architecture follows a client-server model. In our case, the client interacts with the server by calling functions through an API, which would typically use a TLS connection to ensure secure communication. The primary focus of this project is on the cryptographic aspects due to time constraints.

Whenever the client invokes functions located on the server, we will assume the presence of a TLS-secured connection between the client and the server. However, since no API has been implemented, user sessions and access rights are not managed in this setup. Instead, we will assume the existence of an API responsible for handling access control and managing function calls.

1.1 Client-Server secured communications with TLS

As mentioned earlier, we assume a secure communication channel between the client and server, established using Transport Layer Security (TLS). For this project, we will specifically consider the use of TLS 1.3, the latest version of the protocol, which offers improved security and performance over its predecessors.

TLS 1.3 eliminates many legacy features and vulnerabilities present in older versions, such as weak cipher suites and unnecessary handshake steps, ensuring a more robust security framework. It provides confidentiality, integrity, and authentication for the data exchanged between the client and server.

1.2 Server Model

The server in this project is assumed to adhere to the communication protocol in its entirety, ensuring correct interactions with the client. However, we also consider the server to be curious-but-honest. This means that while the server correctly follows the protocol and does not deviate from expected behavior, it may attempt to learn or infer sensitive information from the data it processes. Therefore we have implemented a cryptographically secured message

2 Authentification

2.1 Register Client

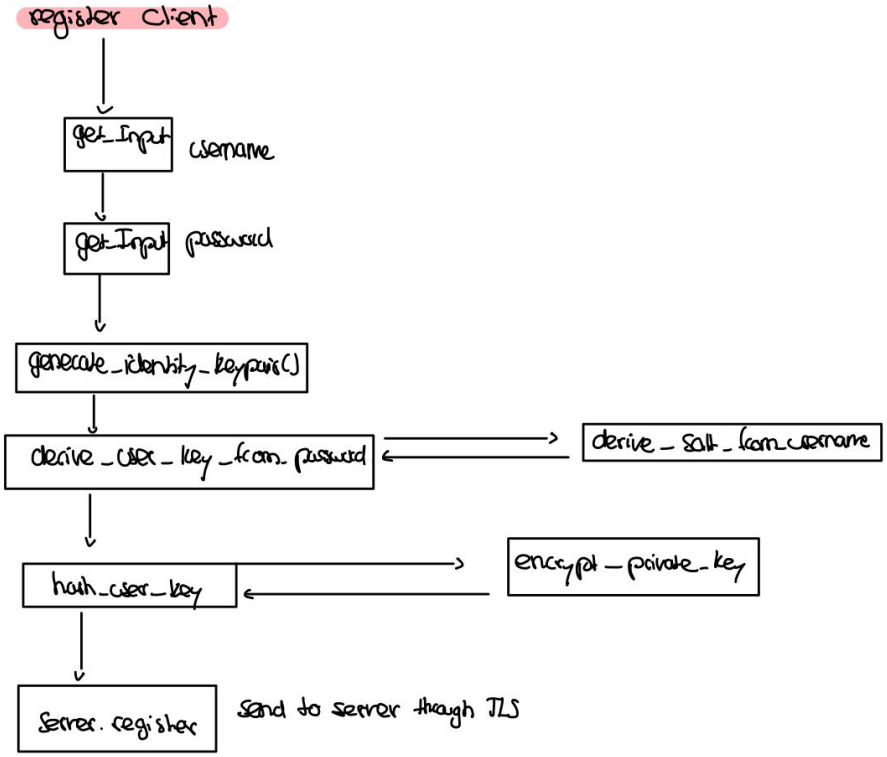

Figure 1: Register

The registration process must be carefully designed to protect clients from a curious-but-honest server. We face two main challenges: ensuring users can log in from any device and protecting sensitive data against potential server misuse.

To enable secure message exchange, we require private and public key pairs. These keys must be stored on the server securely, but the private key should remain unreadable by the server.

Our solution is to derive a key from the user's password to encrypt the private key before sending it to the server. This way, users can log in from any device and use their password to decrypt the private key.

The challenge, however, is that the password itself cannot be sent to the server, as the server could misuse it to generate the derived key. To address this, we use an HKDF to derive a "password" from the user's key, adding an

extra layer of security.

To mitigate the risk of rainbow table attacks in case the server's database is leaked, we also incorporate a salt in the key derivation process with Argon2. The salt is generated dynamically using an HKDF, which is then input into Argon2 to produce a securely derived key.

The workflow for the registration is quite simple, we fill up a form for the user's credentials, and then we do the following for the hashing :

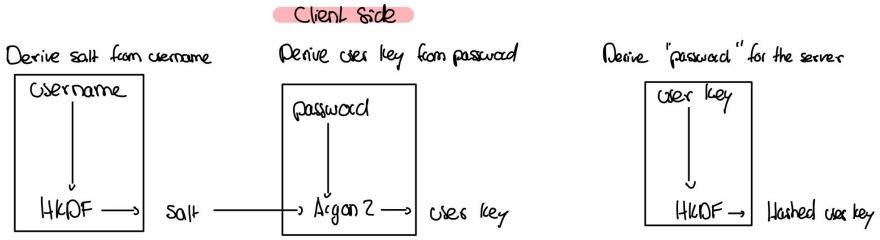

Figure 2: Registration hashing workflow

2.2 Login

The login process builds upon the registration mechanism. During login, the client sends only the username and the derived "password" to the server. This "password" is a hash of the user's key, as illustrated in the schematic above.

Upon receiving the "password," the server hashes it using SHA3 and compares the result with the stored hash. If the hashes match, the login is considered valid.

Once authenticated, the user can retrieve their encrypted private and public keys from the server. The encrypted private key can be decrypted locally, as only the user can derive the decryption key from their password.

```
1
2  def login_client():
3      """
4      \brief Log in a client by verifying username and password, and
       retrieving user data from the server.
5      \return A tuple containing the user object and the user key.
6      """
7      username = get_input("Enter your username")
8      password = get_input("Enter your password")
9
10     user_key = crypto.derive_user_key_from_password(username,
       password)
11     hashed_user_key = crypto.hash_user_key(user_key)
12
13     user = server.login(username, hashed_user_key) # Sever
       authentification through TLS
14     return user, user_key
```

Listing 1: Login

2.3 Change Password

To change the password, the user must be logged in and must also know the old password. This is necessary to decrypt the private key, after which the key is re-encrypted using the new password-derived key.

```
1
2  def modify_password(user: User):
3      """
4      \brief Modify the password of a user.
5      \param user The user whose password is to be modified.
6      """
7      old_password = get_input("Enter your old password")
8      user_key = crypto.derive_user_key_from_password(user.username,
       old_password)
9      hashed_password = crypto.hash_user_key(user_key)
10
11     private_key = crypto.decrypt_private_key(user_key, User.
       getEncryptedPrivateKey(user), User.getNonce(user))
12
13     new_password = get_input("Enter your new password")
14     new_user_key = crypto.derive_user_key_from_password(user.
       username, new_password)
```

```
15      new_hashed_user_key = crypto.hash_user_key(new_user_key)
16      encrypted_private_key, nonce = crypto.encrypt_private_key(
        new_user_key, private_key)
17
18      server.modify_password(user.username, hashed_password,
        encrypted_private_key, nonce, new_hashed_user_key) # Assuming
        we already have a user session with the server and connected
        with TLS
19      print("Password updated successfully!")
```

Listing 2: Change Password

3 Messages

3.1 Send/Receive a message - Scheme

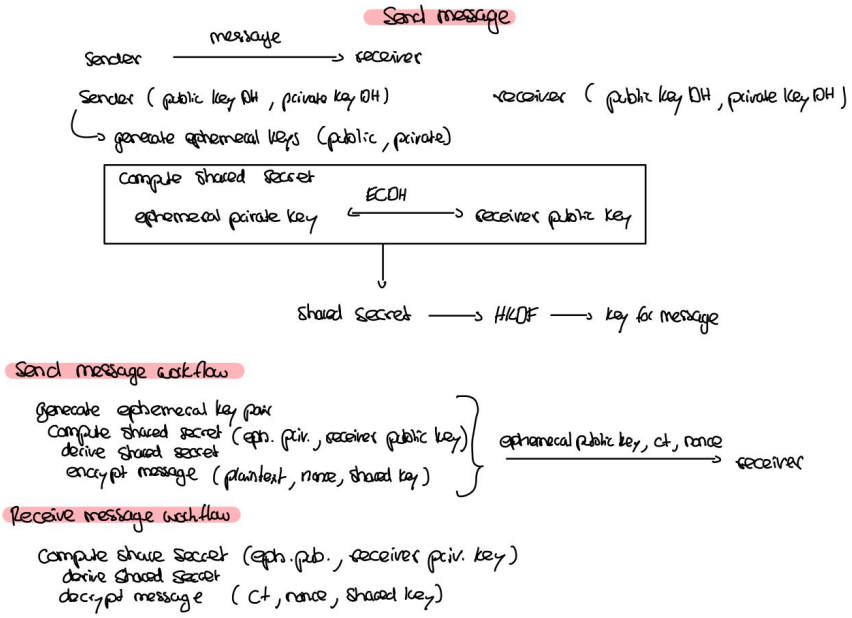

Figure 3: Message Model

3.2 Sending Secure Messages with Timestamp Unlock

The current implementation for securely sending messages to be unlocked at a specific timestamp is based on **asymmetric cryptography**.

When users create an account, a default pair of cryptographic keys is generated for them. These keys are primarily used for performing **Elliptic Curve Diffie-Hellman (ECDH)** key exchanges with other clients. This ensures that secure and private communication channels can be established between users.

3.2.1 Workflow for Sending a Message

When a user sends a message, the following steps are followed:

1. **Ephemeral Key Generation**: A new pair of **ephemeral keys** is generated for each message. These ephemeral keys ensure **forward secrecy**, meaning even if the long-term private key is compromised, previously encrypted messages remain secure.

2. **ECDH Key Exchange**: Using the ephemeral private key of the sender and the known public key of the recipient, an **ECDH key exchange** is performed. This process generates a shared secret, which forms the basis for secure encryption.

3. **Key Derivation**: The shared secret generated from ECDH is not directly used for encryption. Instead, it is passed through a **key derivation function (KDF)** to produce a symmetric key compatible with the **ChaCha20-Poly1305** encryption algorithm.

4. **Message Encryption**: The message is encrypted using the derived symmetric key and a **cryptographically secure nonce**. ChaCha20-Poly1305 is chosen for its speed, security, and authenticated encryption capabilities, ensuring both confidentiality and integrity of the message.

5. **Signing the Message**: To verify the authenticity of the message and prevent tampering, the sender signs the following components using their long-term private key:

   - The encrypted message content
   - The timestamp
   - The nonce

   This ensures that the message can be verified as originating from the sender and has not been altered.

6. **Storing the Message on the Server**: The sender transmits the encrypted message and its metadata to the server, which stores it in a structured format. The server does not have the ability to decrypt the message due to the asymmetric encryption design and the use of ephemeral keys.

3.2.2 Stored Message Format

The server stores the message as a JSON object. Below is an example of the structure:

```
{
    "sender": "test",
    "receiver": "test2",
    "id": 1,
    "senderEphemeralPublicKey": "-----BEGIN PUBLIC KEY-----\
    nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEUAkca3yDAYIZeQT2wfVG29yokN
    /g\nKhA2kCiQ4WyUvADjrV2g231KnW8IkFvK/PNVXKxwaG+aMIsg8R0fcrInFg
    ==\n-----END PUBLIC KEY-----\n",
    "content": "
    i7sdWEowoCUqPTKMMK9INKUzVZaUiNisCmkNrvAF7uVgoEqrpWIJ1/v+jw==",
    "nonce": "P8nU9XzzbmZMNb83",
    "signature": "MEUCIQD4PAi0xqngjybY9FGowp5tEwI584J/
    nPpcDvPuisf7hwIgB02lo+de9wkW50XVAADd3bHM95zt4BBEElyW9zHfKFE=",
    "timeBeforeUnlock": "2025-01-05T20:20:20"
}
```

Listing 3: Stored Message JSON

3.2.3 Explanation of the Fields

- **sender**: The username of the message sender.
- **receiver**: The username of the intended recipient.
- **id**: A unique identifier for the message.
- **senderEphemeralPublicKey**: The ephemeral public key of the sender, which the receiver will use during the decryption process.
- **content**: The encrypted message content (ciphertext).
- **nonce**: A unique value used during encryption to ensure that ciphertexts are not repeated.
- **signature**: A digital signature to authenticate the sender and verify the integrity of the message.
- **timeBeforeUnlock**: The timestamp indicating when the message becomes readable.

3.2.4 Security Benefits

- **Forward Secrecy**: The use of ephemeral keys ensures that compromising a private key in the future does not reveal past messages.
- **Backward Secrecy**: Even if the sender's long-term private key is compromised after the message has been sent, the message remains secure because each message uses a unique ephemeral key pair for encryption.

- **Authenticity**: Digital signatures confirm the identity of the sender and protect against tampering.
- **Confidentiality**: Even an honest-but-curious server cannot decrypt the message due to the separation of keys and the secure use of ChaCha20-Poly1305.

This implementation leverages the strengths of asymmetric and symmetric cryptography to provide a secure, efficient, and scalable solution for time-delayed message delivery.

3.3 Receiving Secure Messages with Timestamp Unlock

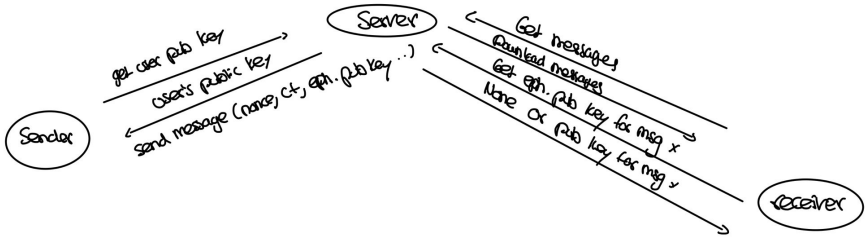

Figure 4: Message Sequence — Client-Server

The message receiving workflow follows a process similar to sending a message. First, the receiver verifies the digital signature of the incoming message. This step ensures that the message was indeed sent by the correct sender and that its content has not been tampered with during transmission.

If the message has a valid signature, the decryption process begins. The receiver uses their own private key, which was securely delivered during the login process, to decrypt the private key. Once the private key is decrypted, the receiver performs the standard **Elliptic Curve Diffie-Hellman (ECDH)** operation with the sender's ephemeral public key and the receiver's private key to compute the shared secret.

Next, the shared secret is passed through a **key derivation function (HKDF)** to produce the symmetric encryption key that will be used for decrypting the message content. The derived key is then used to decrypt the message, which was originally encrypted using the **ChaCha20-Poly1305** algorithm.

The server adheres to the protocol by only sending the ephemeral public key of the message when the timestamp for unlocking the message has passed. This feature adds an extra layer of security, as it prevents premature access to the message. Furthermore, since the ephemeral public key is publicly shared, there is no need to protect it, simplifying key management. It serves its purpose as

the method to decrypt the message, but it can only be used after the designated timestamp.

3.3.1 Security Considerations

- **Message Authentication**: By verifying the digital signature, the receiver can confirm the authenticity of the sender and ensure that the message has not been altered during transit.
- **Confidentiality**: The use of ECDH and a key derivation function ensures that only the intended receiver can decrypt the message, even if the server is honest-but-curious and stores the message.
- **Controlled Access**: The server only transmits the ephemeral public key once the timestamp for unlocking has passed, ensuring that the message remains inaccessible to the receiver before its designated time.
- **Simplified Key Management**: Since the ephemeral public key is transmitted in plain text, there is no need to secure it, reducing complexity in key management. The key can only be used for decrypting the message after the timestamp, providing an additional security feature.

This receiving workflow ensures that the message remains secure, authenticated, and accessible only at the correct time, while maintaining a simple and efficient process for managing encryption keys.

3.4 Download messages

To test the simple proof of concept of our cryptography in this project, I simply created a function to download all messages for the user, it downloads the locked messages and unlocked messages.

```
def get_my_messages(user: User):
    """
    \brief Retrieve and download messages for a user.
    \param user The user whose messages are to be retrieved.
    """
    unlocked_messages, locked_messages = server.get_user_messages(
    user.username, user.hashedPassword)
    download_messages(user, unlocked_messages, locked_messages)
```

Listing 4: Download messages


```
[
    {
        "id": 1,
        "sender": "test",
        "receiver": "test2",
        "content": "yBYudsmU+wgvc1qSiNadhzF5ljW4",
        "nonce": "HEmxNXYFFqPsAdrY",
        "signature": "MEYCIQDLUhYlgn/
    ZlXcHDowyfUT5PRfZNg1CakfjtHOfbmVPnAIhALN/WOcWkrY2SXBALMw+
    DJcBWqJlMODN5qH6CUxIZzGI",
        "senderEphemeralPublicKey": "-----BEGIN PUBLIC KEY-----\
    nMFkwEwYHKoZIzj0CAQYIKoZIzj0DAQcDQgAEvK1UrPuMcYLx9fDosj/Glhtt/
    Zv+\nIRfLocUf7xp523ASWWg3rh32Lcqa9FN7D3F6HNLArT5EAEFsgULV0AIHYg
    ==\n-----END PUBLIC KEY-----\n",
        "timeBeforeUnlock": "2024-01-01T20:20:20",
        "is_decrypted": true,
        "decrypted_content": "test2",
        "timestamp": "2025-01-05T23:57:56.869546"
    },
    {
        "id": 2,
        "sender": "test",
        "receiver": "test2",
        "content": "6zKl9chyHuWSJK97sIfczM4AqB99emmuV+
    zwJ3pPZZfVviyIfinp",
        "nonce": "Q9nx7QwP/8WyNakr",
        "signature": "MEUCIQCPQ8RJ/+2cUc/D+PXinRAjGjNqbNrsxYkxpr+d6
    +nq0AIgDkIrUh3tjgWLIb5qhv6rM8/LMpLM2PYujfC6eBWApfY=",
        "senderEphemeralPublicKey": null,
        "timeBeforeUnlock": "2025-02-01T20:20:20",
        "is_decrypted": false,
        "decrypted_content": null,
        "timestamp": "2025-01-06T00:33:06.511444"
    }
]
```


Listing 5: Downloaded client messages (Unlocked and Locked)

3.5 Download a new message

Created a function so we don't need to download the same message twice, in case messages are very very heavy... This downloads the missing messages that aren't locally saved.

```
def download_new_messages(user: User):
    """
    \brief Download new messages for a user.
    \param user The user whose new messages are to be downloaded.
    """
    message_ids = db_local_message.getAllMessageIDs()
    unlocked_messages, locked_messages = server.get_new_messages(
    user.username, user.hashedPassword, message_ids)
    download_messages(user, unlocked_messages, locked_messages)
```

Listing 6: Download new messages

3.6 Unlock available downloaded messages

I created a function to unlock previously locked messages that have surpassed their timestamp. The client sends the message IDs to the server, requesting the ephemeral public keys to perform the Diffie-Hellman (DH) exchange and decrypt the messages. Assuming the user session is active via TLS, the server will return the ephemeral public keys if the timestamp has expired. The user can then decrypt the messages by performing an ECDH exchange, computing the shared secret, and using it to decrypt the received message.

```
def unlock_available_messages(user: User):
    """
    \brief Unlock available messages for a user.
    \param user The user whose messages are to be unlocked.
    """
    message_ids = db_local_message.getUndecryptedUnlockedMessageIDs
    ()
    ephemeral_keys = server.get_message_ephemeral_public_keys(user.
    username, user.hashedPassword, message_ids)
    if not ephemeral_keys:
        print("No messages are ready to be unlocked.")
        return

    for message_id, ephemeral_key in ephemeral_keys.items():
        if ephemeral_key:
            local_message = db_local_message.get_message_by_id(
    message_id)
            local_message.senderEphemeralPublicKey = ephemeral_key
            message = tools.convert_local_to_message(local_message)
            receive_message_from_user(user, message)
```

Listing 7: Unlock messages

4 Key Management

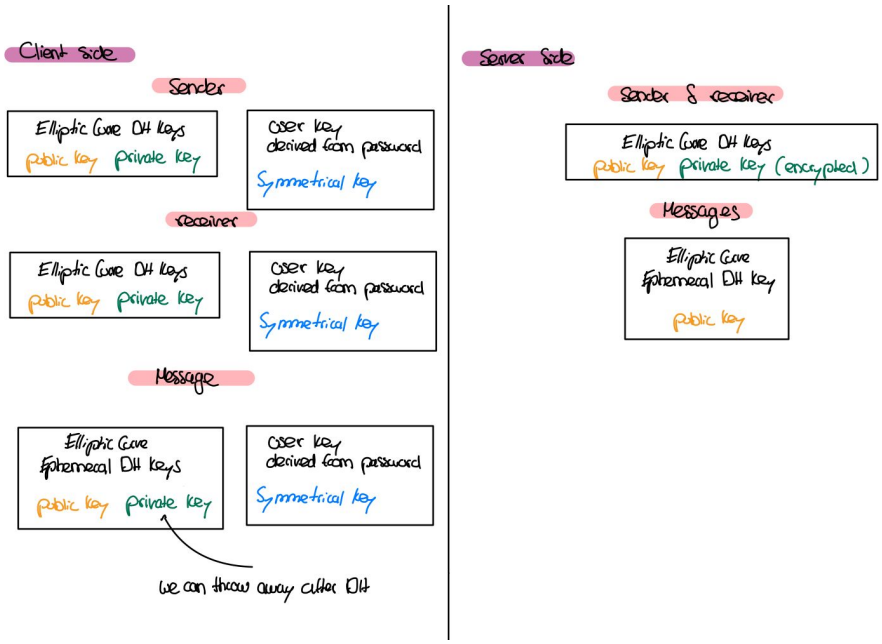

Figure 5: Key Management

In our application, key management is crucial to ensuring both the security and functionality of the system. We utilize a set of Elliptic Curve Diffie-Hellman (ECDH) keys for each registered client, along with a pair of ephemeral keys for each message.

When a client registers, they generate a pair of ECDH keys: a private key and a corresponding public key. The private key is then encrypted using a symmetric key derived from the user's password, ensuring that the private key remains protected at all times. The public key and the encrypted private key are sent to the server, where they are securely stored. This allows the client to log in from any device, as the server holds the necessary key material to decrypt the private key and authenticate the user.

For each message the client wishes to send, a new pair of ephemeral keys is generated. Only the public key of the ephemeral key pair is transmitted to the server, while the private key is discarded immediately after it is used. The ephemeral private key is employed to perform the Diffie-Hellman key exchange with the recipient's public key, allowing the client to compute the shared secret. This shared secret is subsequently used for encrypting the message content with symmetric encryption algorithms, such as ChaCha20-Poly1305.

The use of ephemeral keys ensures that no sensitive information is retained longer than necessary, and that each message has a unique set of keys for en-

cryption. This approach also enhances forward and backward secrecy, as each session and message is encrypted with a fresh key pair that is discarded after use.

In summary, key management in our application relies on a combination of long-term ECDH keys for user authentication and short-term ephemeral keys for secure message exchange (even against the server it-self).

5 Cryptography choices

5.1 Cryptography - Hash Functions

To respect the current conditions of the project, we are using three different hash functions to process user registration and login.

5.1.1 HKDF

HKDF (HMAC-based Key Derivation Function) plays a central role in our system for deriving cryptographic keys and salts in a secure and flexible manner. It is a standardized key derivation function that builds upon HMAC (Hash-based Message Authentication Code) and is specifically designed to take a shared secret or input key material (IKM) and derive multiple, cryptographically strong output keys.

In our implementation, HKDF is used for several critical purposes:

- **Key Derivation:** HKDF derives encryption keys from shared secrets generated during the ECDH key exchange. This ensures that the output keys have the desired cryptographic strength and can be safely used for symmetric encryption algorithms, such as ChaCha20-Poly1305.
- **Salt Generation:** HKDF derives salts from user-specific information, such as usernames. This allows us to securely generate a unique salt for each user without storing the username on the server. By re-deriving the salt dynamically at runtime, we further protect sensitive user data.
- **Password Hashing:** HKDF is used to derive keys for hashing user passwords. This enhances security by ensuring that the derived keys are application-specific and contextually tied to their intended use.

Why HKDF is a Good Fit for Our Case
HKDF is well-suited to our application for the following reasons:

- **Security:** HKDF ensures that the derived keys are independent of each other, even when derived from the same input key material. This property, known as key separation, is critical in a system with multiple cryptographic operations.
- **Flexibility:** HKDF can derive keys of arbitrary lengths and can be customized for different contexts using its optional `info` parameter. This enables us to generate keys tailored to specific tasks, such as encryption, salt generation, or password hashing. In our case, everything is outputted to 32 bytes, which is complex to brute force.
- **Simplicity:** HKDF has a straightforward design, making it easy to implement and integrate with existing cryptographic libraries. Its reliance on HMAC ensures robust cryptographic guarantees.

The library we use (e.g., Python's `cryptography` library) implements HKDF in compliance with the standard, ensuring that the derivation process is secure and efficient.

In our system, HKDF strengthens the security of our key management and encryption mechanisms while simplifying the handling of cryptographic operations. By using HKDF, we ensure that the derived keys are robust against attacks, and the system remains scalable and maintainable.

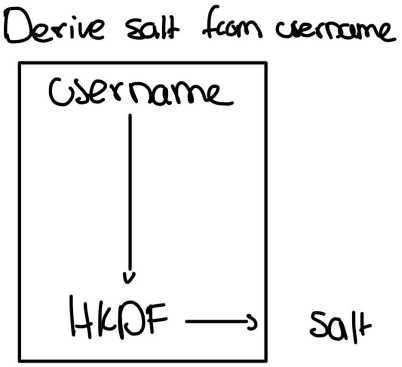

Figure 6: Deriving Salt from Username with HKDF

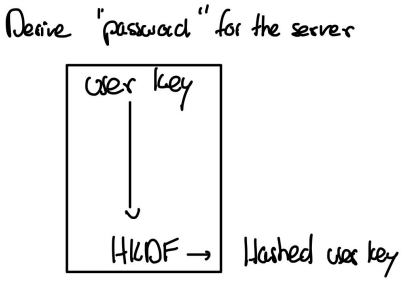

Figure 7: Deriving User Key with HKDF

```
def hash_user_key(userkey):
    """
    \brief Hashes a user key using HKDF.
    \param userkey The user key to hash.
    \return The hashed user key.
    """
```

```
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b"server-pwd-hash",
        backend=default_backend()
    )
    return hkdf.derive(userkey)

def derive_salt_from_username(username):
    """
    \brief Derives a salt from a username using HKDF.
    \param username The username to derive the salt.
    \return The derived salt.
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b"password-salt",
        backend=default_backend()
    )
    return hkdf.derive(username.encode())

def derive_encryption_key(shared_secret):
    """
    \brief Derives an encryption key from a shared secret using
    HKDF.
    \param shared_secret The shared secret.
    \return The derived encryption key.
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
        info=b"dh-ratchet",
        backend=default_backend()
    )
    encryption_key = hkdf.derive(shared_secret)
    return encryption_key
```

Listing 8: Key Derivation with HKDF

5.1.2 Argon2

Argon2 is a key component in our system for securely hashing user passwords. It is a modern, memory-hard password hashing function designed to protect against brute-force attacks and computationally intensive hardware, such as GPUs or FPGAs. Argon2 was the winner of the Password Hashing Competition (PHC) in 2015 and is recognized for its robustness and efficiency.

In our implementation, Argon2 is used to derive a key to encrypt the asymmetrical private key by the user. In this schematic, the derived key is "user key".

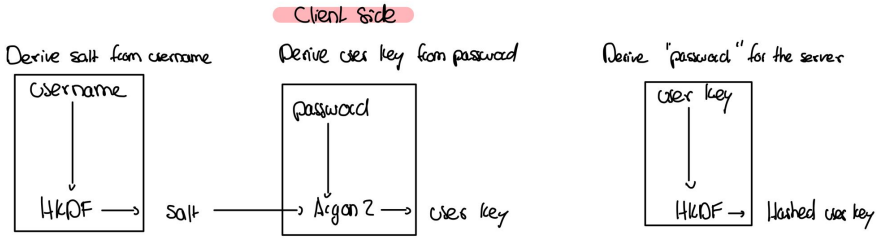

Figure 8: Derive key from password

Why Argon2 is a Good Fit for Our Case

Argon2 was chosen for its balance between security, performance and eaasy to use. Below are the reasons it is well-suited for our system:

- **Memory Hardness:** Argon2 is designed to require a significant amount of memory, making it costly for attackers to parallelize brute-force attacks using modern hardware.
- **Configurable Parameters:** Argon2 allows customization of memory usage, iterations, and parallelism, enabling us to tailor the function to our security needs and system resources.
- **Resilience to Side-Channel Attacks:** Argon2 provides protection against side-channel attacks by ensuring that its operations are difficult to predict or manipulate.
- **Ease of Integration:** Argon2 is widely supported by modern cryptographic libraries, simplifying its integration into our system.

Configuration Parameters

To balance security and performance, we have configured Argon2 with the following parameters:

- **Memory Cost:** 64 MB, which ensures sufficient memory usage to deter hardware-based brute-force attacks.

- **Iterations:** 5 iterations, which control the computational workload and hashing time.
- **Parallelism:** 4 threads, optimizing performance on multi-core processors.
- **Salt Length:** 16 bytes, ensuring unique salts for each password to prevent rainbow table attacks.
- **Hash Length:** 32 bytes, providing a fixed-length output that is sufficiently large for security purposes.

Implementation

The Argon2 implementation in our system utilizes the Python `argon2-cffi` library, which adheres to the Argon2 specification. This library ensures robust security and reliable performance.

```
def derive_user_key_from_password(username, password):
    """
    \brief Derives a user key from a password using Argon2id.
    \param username The username to derive the salt.
    \param password The password to derive the key.
    \return The derived user key.
    """
    salt = derive_salt_from_username(username)
    user_key = hash_secret_raw(
        secret=password.encode(),
        salt=salt,
        time_cost=5,
        memory_cost=2**16,
        parallelism=1,
        hash_len=32,
        type=Type.ID
    )
    return user_key
```

Listing 9: Hashing and Verifying Passwords with Argon2

Security Considerations

- **Resistance to Pre-Computed Attacks:** By combining a unique salt and a memory-hard hashing function, Argon2 ensures that pre-computed attacks, such as rainbow tables, are infeasible.
- **Parameter Tuning:** The configurable parameters allow us to adapt Argon2 to evolving hardware capabilities and threat models.

5.1.3 SHA3

SHA3 (Secure Hash Algorithm 3) has been integrated into our system to provide an efficient and robust mechanism for hashing. While Argon2 is used for securely deriving and hashing passwords during user registration and login, SHA3 serves as an additional layer of hashing for the password received by the server.

By leveraging SHA3, we ensure efficient processing without reusing Argon2 unnecessarily, as Argon2 has already handled the complex hashing and key derivation tasks (the user uses Argon to derive a key from a password, then we derive a "password" from that key to send to the server).

Why SHA3 is a Good Fit for Our Case

SHA3 was chosen for this specific use case due to the following reasons:

- **Performance:** SHA3 provides high-speed hashing, making it ideal for the server-side hashing of the pre-hashed password sent by the user.
- **Security:** SHA3-512 offers strong cryptographic guarantees, including resistance to collision and pre-image attacks, ensuring the integrity of the hashed password.
- **Standardized Algorithm:** SHA3 is part of the NIST-approved cryptographic standards, ensuring it meets modern security requirements and is supported by widely used cryptographic libraries.
- **Simplicity:** SHA3 is easy to implement and integrates seamlessly into our existing cryptographic architecture.

Implementation

The following code snippet demonstrates how SHA3-512 is used to hash the password received by the server:

```
import hashlib

def hash_password(password: bytes) -> bytes:
    """
    \brief Hashes a password using SHA3-512.
    \param password The password to hash.
    \return The hashed password.
    """
    sha3_hasher = hashlib.sha3_512()
    sha3_hasher.update(password)
    return sha3_hasher.digest()
```

Listing 10: Server Password Hashing with SHA3-512

Security Considerations

- **Avoiding Redundant Hashing:** By combining Argon2 for initial hashing and SHA3-512 for server-side processing, we ensure optimal performance without unnecessary duplication of cryptographic steps.
- **Cryptographic Strength:** SHA3-512 provides strong resistance to cryptographic attacks, ensuring the integrity of the server-side password hash.

By integrating SHA3-512 into our system, we achieve a balance between performance and security, ensuring that the server can efficiently handle user authentication while maintaining robust protection against cryptographic threats.

5.2 Cryptography - Asymmetrical

Asymmetrical cryptography is the core security mechanism of the application. It ensures that messages are sent securely without requiring prior interaction between users or reliance on the server for confidentiality. This is achieved using Elliptic Curve Diffie-Hellman (ECDH) for key exchange and a combination of ephemeral and static key pairs for enhanced security and forward secrecy.

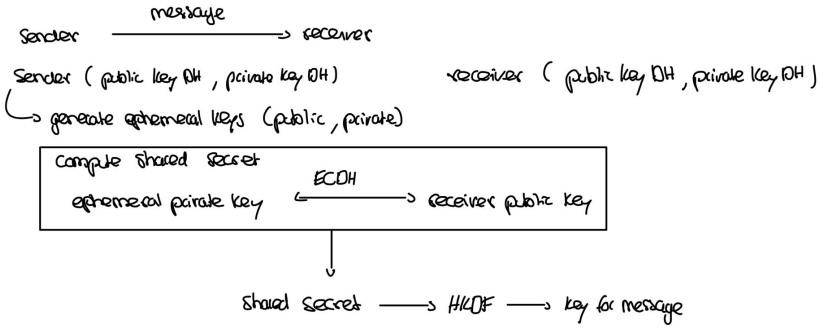

Figure 9: Core usage of asymmetrical cryptography

5.2.1 Elliptic Curve Cryptography (ECC) Configuration

Our application uses Elliptic Curve Cryptography (ECC) for key generation, exchange, and signing. The following configuration and standards are employed:

- **Curve:** SECP256R1 (also known as P-256), which is a widely supported and NIST-recommended elliptic curve.
- **Key Size:** 256 bits, providing a high level of security while ensuring computational efficiency.
- **Key Exchange Algorithm:** Elliptic Curve Diffie-Hellman (ECDH) for secure shared secret computation between parties.
- **Digital Signatures:** Elliptic Curve Digital Signature Algorithm (ECDSA) is used for signing keys and verifying their authenticity.
- **Hashing Algorithm:** SHA-256 is employed for HKDF and cryptographic operations, ensuring robust hash security.
- **Symmetric Encryption:** ChaCha20-Poly1305 is used for message encryption, providing authenticated encryption with associated data (AEAD) and resistance against nonce reuse.

5.2.2 Why ECC and ECDH Are Used

ECC provides equivalent security to traditional methods (such as RSA) with smaller key sizes, making it efficient in terms of storage and computational overhead. By using ephemeral keys for each message and static keys for user identities, our system achieves:

- **Forward Secrecy:** Ensures that past communications remain secure even if long-term private keys are compromised.
- **Low Computational Overhead:** ECC operations are computationally efficient, making the system suitable for resource-constrained environments.
- **Compact Keys:** Smaller keys result in reduced bandwidth and storage requirements, which is ideal for modern applications.

5.2.3 Key Management and Operations

Key Pair Generation Key pairs are generated for both static identity keys and ephemeral session keys. The static keys are used for long-term authentication, while ephemeral keys ensure that each message exchange remains independent and secure.

```
def generate_identity_keypair():
    private_key = ec.generate_private_key(ec.SECP256R1(),
    default_backend())
    public_key = private_key.public_key()
    return private_key, public_key

def generate_ephemeral_keypair():
    private_key = ec.generate_private_key(ec.SECP256R1(),
    default_backend())
    public_key = private_key.public_key()
    return private_key, public_key
```

Listing 11: Key Pair Generation

Shared Secret Computation and Key Derivation The shared secret between the sender and receiver is computed using ECDH. To derive encryption keys from the shared secret, we use HKDF. This ensures that the derived keys are unique and context-specific, preventing key reuse vulnerabilities.

```
def compute_shared_secret(private_key, peer_public_key):
    shared_secret = private_key.exchange(ec.ECDH(), peer_public_key
    )
    return shared_secret

def derive_encryption_key(shared_secret):
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None,
```

```
        info=b"dh-ratchet",
        backend=default_backend()
    )
    encryption_key = hkdf.derive(shared_secret)
    return encryption_key
```

Listing 12: Shared Secret Computation and Encryption Key Derivation

5.2.4 Security Guarantees

By combining ECC, HKDF, and AEAD encryption, our system ensures:

- **Confidentiality:** Messages are encrypted end-to-end.
- **Integrity:** Authenticated encryption prevents tampering.
- **Forward Secrecy:** Ephemeral keys protect past communications.
- **Compact and Efficient Design:** ECC ensures lightweight cryptographic operations.

5.2.5 Digital Signatures: Elliptic Curve Digital Signature Algorithm (ECDSA)

To ensure the authenticity and integrity of messages and keys, the application utilizes the Elliptic Curve Digital Signature Algorithm (ECDSA). Digital signatures provide a mechanism for verifying that messages or public keys originate from the intended sender and have not been tampered with during transmission.

Why ECDSA is Used

ECDSA was chosen for its alignment with the security goals of the application and its efficiency:

- **Security:** ECDSA provides strong guarantees of authenticity and non-repudiation, ensuring that only the holder of a private key can generate a valid signature.
- **Efficiency:** Due to the use of ECC, ECDSA achieves equivalent security to traditional algorithms like RSA but with smaller key sizes and faster computations.
- **Compact Signatures:** The signatures generated by ECDSA are much smaller compared to RSA, which reduces bandwidth and storage overhead.
- **Standardization:** ECDSA is widely supported and standardized, making it interoperable with other systems and libraries.

Configuration and Usage

In the application, ECDSA is implemented using the following configuration:

- **Curve:** SECP256R1 (P-256), providing a good balance between security and performance.

- **Hash Function:** SHA3-512, a secure and efficient hash function used to pre-hash messages before signing or verifying.
- **Signing Keys:** The sender's private key is used to generate the signature, and the corresponding public key is used by the receiver to verify it.

Implementation

The following code demonstrates the signing and verification process using ECDSA with SHA3-512:

```
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.primitives import hashes

def sign_message(private_key, message):
    """
    \brief Sign the message with the sender's private key.
    \param private_key The private key used to sign the message.
    \param message The message to be signed.
    \return The generated signature.
    """
    signature = private_key.sign(
        message,
        ec.ECDSA(hashes.SHA3_512())
    )
    return signature

def verify_signature(public_key, message, signature):
    """
    \brief Verify the message signature with the sender's public
    key.
    \param public_key The public key used to verify the signature.
    \param message The message whose signature is to be verified.
    \param signature The signature to be verified.
    \return True if the signature is valid, False otherwise.
    """
    try:
        public_key.verify(
            signature,
            message,
            ec.ECDSA(hashes.SHA3_512())
        )
        return True
    except Exception:
        return False
```

Listing 13: Signing and Verifying Messages

Workflow in the System

- The sender generates a digital signature for the message using their private key.
- The signature is appended to the message and sent to the receiver.
- The receiver uses the sender's public key to verify the signature. If the signature is valid, the message integrity and authenticity are confirmed.

Security Considerations

- **Non-repudiation:** The use of the sender's private key ensures that only the sender can generate a valid signature, preventing denial of authorship.
- **Integrity:** Any modification of the message invalidates the signature, ensuring that the message remains untampered.
- **Replay Protection:** Including a timestamp or unique identifier in the message can prevent replay attacks.

By implementing ECDSA for digital signatures, the application provides robust guarantees of authenticity, integrity, and non-repudiation, ensuring secure communication and key management in the system.

5.3 Cryptography - Symmetrical

Symmetrical cryptography is used in our application to ensure the confidentiality and integrity of messages exchanged between users. We rely on ChaCha20-Poly1305, an authenticated encryption algorithm, to achieve this. ChaCha20-Poly1305 is chosen due to its performance for any device, security, and resistance to side-channel attacks, making it particularly suitable for modern and diverse systems.

5.3.1 ChaCha20-Poly1305 Encryption

ChaCha20 is a high-speed stream cipher that forms the basis of ChaCha20-Poly1305. Poly1305, a one-time authenticator, is combined with ChaCha20 to provide authenticated encryption with associated data (AEAD). This ensures that the ciphertext is both encrypted and authenticated, offering protection against tampering and unauthorized access.

- **Nonce:** A unique 12-byte random value is generated for every encryption operation to ensure uniqueness and prevent nonce reuse vulnerabilities.
- **Encryption Key:** A 32-byte key is derived using HKDF, ensuring strong cryptographic properties.
- **Authentication:** Poly1305 generates a tag for verifying the integrity of the ciphertext and associated data.

5.3.2 Encryption Workflow

During encryption, ChaCha20-Poly1305 takes a plaintext message, a unique nonce, and an encryption key to produce a ciphertext and an authentication tag. The tag ensures that any modifications to the ciphertext or associated data are detected during decryption.

```
def encrypt_message(encryption_key, plaintext):
    """
    \brief Encrypts a message using ChaCha20-Poly1305.
    \param encryption_key The encryption key.
    \param plaintext The plaintext message.
    \return The nonce and ciphertext.
    """
    nonce = os.urandom(12)
    cipher = ChaCha20Poly1305(encryption_key)
    ciphertext = cipher.encrypt(nonce, plaintext, None)
    return nonce, ciphertext
```

Listing 14: Encrypting a Message with ChaCha20-Poly1305

5.3.3 Decryption Workflow

Decryption requires the same encryption key, the nonce used during encryption, and the ciphertext. ChaCha20-Poly1305 ensures that decryption only succeeds if the authentication tag matches, guaranteeing message integrity.

```python
def decrypt_message(encryption_key, nonce, ciphertext):
    """
    \brief Decrypts a message using ChaCha20-Poly1305.
    \param encryption_key The encryption key.
    \param nonce The nonce used for encryption.
    \param ciphertext The encrypted message.
    \return The decrypted plaintext message.
    """
    cipher = ChaCha20Poly1305(encryption_key)
    plaintext = cipher.decrypt(nonce, ciphertext, None)
    return plaintext
```

Listing 15: Decrypting a Message with ChaCha20-Poly1305

5.3.4 Security Properties

ChaCha20-Poly1305 offers the following guarantees:

- **Confidentiality:** Ensures that the message remains unreadable without the encryption key.
- **Integrity:** Detects any tampering of the ciphertext or associated data.
- **Nonce Uniqueness:** Requires a unique nonce for every encryption operation to prevent attacks.

5.3.5 Why ChaCha20-Poly1305?

- **Efficiency:** ChaCha20 is faster than AES on platforms without dedicated hardware acceleration.
- **Security:** It is resistant to timing attacks and other side-channel vulnerabilities.
- **Flexibility:** Suitable for a wide range of devices, from servers to embedded systems.