UNIVERSITY OF PISA

MASTER'S DEGREE IN COMPUTER ENGINEERING

Foundations of Cybersecurity

Bulletin-Board-System

Professor:

Gianluca Dini

Student:

Giovanni Ligato

ACADEMIC YEAR 2023/2024

Index

1. Introduction

The system under review is a Bulletin Board System (*BBS*), designed as a distributed service where users can interact by reading and posting *messages*. The BBS provides users with the following core functionalities:

- *List*: Displays the latest messages available on the BBS.
- *Get*: Downloads a specific message from the BBS based on the provided message identifier.
- *Add*: Allows users to add a new message to the BBS.

Before using these operations, users must successfully *log* into the system. A *secure connection* ensures that all operations are executed safely. Additionally, new users must *register* before they can log in and access the system. Once *logged out*, users must log in again to perform further actions.

The system is designed with a strong focus on *security*, requiring user authentication for all operations and safeguarding communication through encryption. The BBS strives to deliver a reliable and user-friendly platform for securely sharing and organizing information.

2. Design

This section outlines the key entities involved in the BBS, along with the assumptions and requirements that support its functionality. It concludes with a detailed explanation of how a secure connection is established between the client and server, with a focus on the custom protocol developed for this purpose.

2.1. System Specification

In this section, the main components of the BBS system are defined, including the server, users, and messages. The subsections below describe the responsibilities and attributes of these entities, setting the foundation for how they interact within the BBS framework.

2.1.1. Server

The BBS server is a centralized entity responsible for securely managing client requests in a multi-threaded environment. It operates at a fixed (IP, port) combination. The server possesses a public-private key pair, with its public key ($pubK_S$) shared and known to all clients. This assumption (i.e. the fact that the server's public key is universally known) simplifies the protocol for establishing a secure connection between the client and server, as it eliminates the need for involving a Certification Authority (CA).

2.1.2. User

Users interact with the BBS through *clients*. Each user is uniquely identified by a nickname, set during registration, along with a password. Additionally, users provide an email address, which the system uses as a secure authentication channel during the registration phase.

2.1.3. Message

A message in the BBS is represented as a tuple consisting of an identifier, title, author, and body. The identifier is unique to each message, while the author field records the nickname of the user who created the message.

2.2. Requirements

The system must meet the following key security requirements:

1. *Confidentiality*: Protect sensitive information from unauthorized access.
2. *Integrity*: Ensure data remains unaltered during transmission.
3. *Replay Protection*: Prevent attackers from reusing previous messages to gain unauthorized access.
4. *Non-Malleability*: Ensure that any intercepted and modified encrypted messages cannot affect the system in a valid way.
5. *Perfect Forward Secrecy*: Guarantee that even if a user's long-term secret key is compromised, past session data remains secure.

Additionally, passwords must never be stored or transmitted in the clear. The system is implemented in C++ with the *OpenSSL* library to minimize code vulnerabilities.

2.3. Secure Connection

When a client *starts*, a *TCP connection* is established between the client and the BBS server, as shown in Figure 1. However, this type of connection, on its own, is insecure and susceptible to several vulnerabilities.

- *Eavesdropping*: Unauthorized parties can intercept the exchanged messages, compromising confidentiality.
- *Message Tampering*: The integrity of the communication is at risk, as messages can be altered.
- *Replay Attacks*: Previously transmitted messages can be reused to bypass security measures.
- *Message Modification*: Even encrypted messages can be manipulated, compromising non-malleability.

To address these risks, the system establishes a *secure connection* before any client-initiated operation using a custom security protocol. This protocol generates a shared secret key between the client and server, which is then used as a symmetric key for encrypting and decrypting subsequent communications. Specifically, if the operation is a successful login, the shared secret key remains active until logout, maintaining a secure *session* between client and server. For other operations or if login fails, a new secure connection is established again every time. Figure 2 illustrates the protocol in action.

In the first message ($M1$), the client sends two components to the server:

1. The first component is a client-generated authentication key ($authK$), encrypted using the server's public key to ensure that only the server can decrypt it.
2. The second component contains the client's public key for the Diffie-Hellman key exchange and a nonce, which helps link $M1$ to $M2$ and prevents replay attacks.

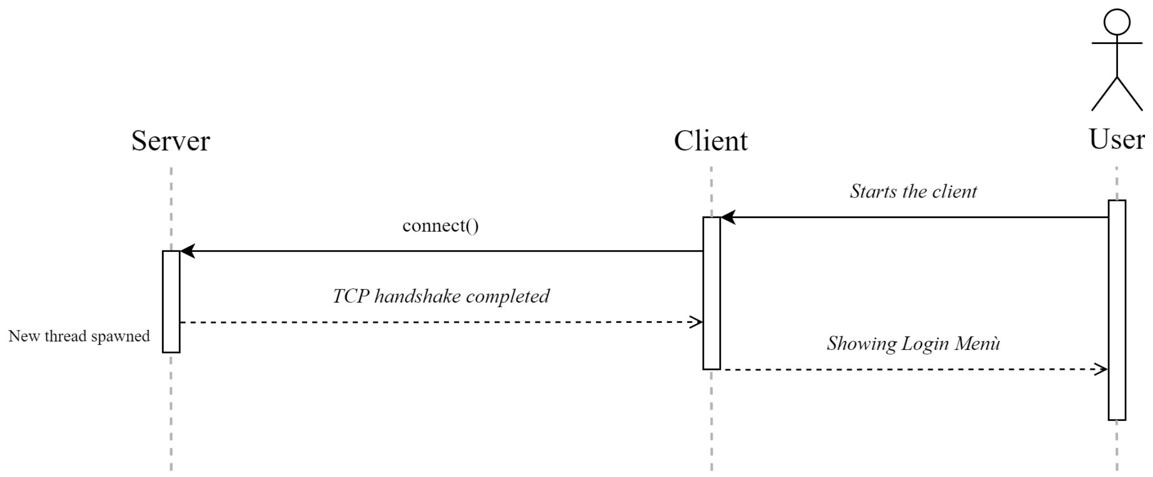


Figure 1: Initial unsecured TCP connection between the client and server.

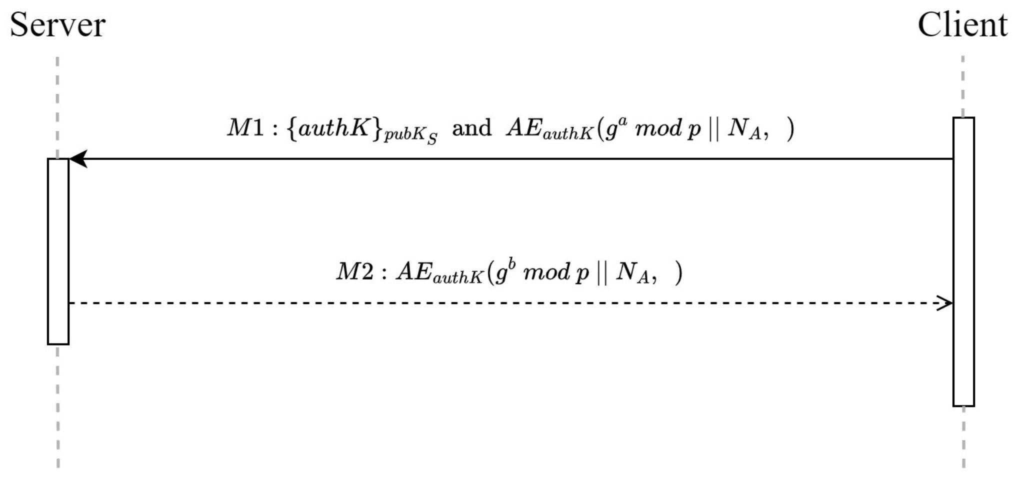

Server
Client
$M1 : \{authK\}_{pubK_S}$ and $AE_{authK}(g^a \, mod \, p \,||\, N_A, \;)$
$M2 : AE_{authK}(g^b \, mod \, p \,||\, N_A, \;)$

Figure 2: Secure connection establishment between client and server using a custom protocol.

Before delving into the details of the second component, it's important to understand the concept of *Authenticated Encryption (AE)*. AE serves two purposes: it provides both encryption and authentication for the exchanged data. Specifically, *Additional Authenticated Data (AAD)* is authenticated but not encrypted, while the plaintext is both encrypted and authenticated. The *tag* produced by the AE process ensures the integrity of both the ciphertext and the AAD, acting as a verification mechanism. The AE process can be summarized as follows:

$$AE_K(AAD, Plaintext) = (Ciphertext, Tag)$$

where:

- AE_K: Authenticated encryption using key K.
- AAD: Additional authenticated data that is not encrypted.
- *Plaintext*: The data that is encrypted.
- *Ciphertext*: The encrypted form of the plaintext.
- *Tag*: The tag used to verify the authenticity of both the ciphertext and the AAD.

In the protocol under consideration, only the AAD is specified, as no plaintext is present. Specifically, the AAD of M1 consists of the client's Diffie-Hellman public key and a client-generated nonce, neither of which require confidentiality. This is because the focus is on tampering protection, and recovering the private key a from the public key $g^a \mod p$ would require solving the discrete logarithm problem, which is computationally infeasible. These values are authenticated using $authK$, the key shared between the client and server.

The Diffie-Hellman parameters (g and p) are publicly known and standardized, meaning both the client and server use the same values. Upon receiving M1, the server prepares M2, which also contains only AAD for the same reasons. In M2, the server includes its own Diffie-Hellman public key and the client-provided nonce. The server does not generate a new nonce because its unique public key ensures that the same nonce (N_A) will typically correspond to different shared secrets. This nonce is only used to protect the client side against replay attacks from a man-in-the-middle.

Upon receiving M2, the client verifies that N_A matches the previously sent value and confirms the message's authenticity, as only the client and the server have access to $authK$. The client then computes the tag to confirm the integrity of M2, ensuring no tampering has occurred.

After exchanging M1 and M2, both the client and server can compute the shared secret, $g^{ab} \mod p$. Before using this shared secret as a symmetric key, it is hashed to enhance security. This hashed value becomes the key for symmetric encryption and decryption.

Evaluating the protocol against security requirements:

1. *Confidentiality* is ensured in subsequent packet exchanges through symmetric encryption using the shared symmetric key, known only to the client and server.
2. *Integrity* is guaranteed by the AE process, which uses a tag to verify the data.
3. *Replay Protection* is achieved in this phase through nonces, and in future phases by additional safeguards.
4. *Non-Malleability* is enforced by the AE process, which prevents tampering with the ciphertext or AAD.

5. *Perfect Forward Secrecy* is guaranteed because the shared secret is unique to each session and cannot be recovered even if the server's long-term private key is compromised. The use of ephemeral Diffie-Hellman key exchange ensures this, as the server's public key is only used for authentication, not for encrypting the shared secret.

3. Implementation

This section outlines the implementation of the BBS system, detailing its architecture, modular components, and the flow of client-server operations. Additionally, it describes the packet structure used for secure communication between the client and server, emphasizing the secure exchange of messages.

3.1. System Architecture

The system architecture is organized into distinct modules, each grouped into folders based on specific responsibilities. This modular structure promotes maintainability and scalability. The following sections outline the purpose and role of each module within the overall system.

3.1.1. Server

The Server module contains the main server-side logic for the BBS, which is responsible for handling client requests and managing communications in a secure, multi-threaded environment. It also includes the server Storage, which persistently stores the server's cryptographic keys, user accounts, and messages posted on the bulletin board. All data stored by the server is securely managed to maintain consistency across multiple executions.

3.1.2. Client

The Client module contains the code that allows users to interact with the BBS. It also includes Client Storage, where the server's public key is stored. This module includes a simulated email folder used for authentication during registration. For example if a user's email is *user123@gmail.com*, the server writes a challenge to a file at `Client\Storage\Emails\user123@gmail.com.txt`.

3.1.3. FileSystem

The FileSystem module manages the persistence of user data and bulletin board messages across sessions. It securely stores user accounts, including nicknames and hashed passwords, and manages the messages posted on the bulletin board. This ensures that messages, along with their metadata like identifier, author, title, and body, are consistently available for retrieval by clients.

3.1.4. Packets

The Packets module handles the low-level communication details between the client and server. It defines custom packet structures that are used to securely transmit data during operations like key exchange and general communication. Packets are designed with cryptographic elements such as initialization vectors (IV), authentication tags (TAG), and encrypted message content.

As previously discussed, the initial packet for secure connection establishment ($M1$) is divided into two parts. By design, the first part, which securely transmits the authentication key ($authK$) encrypted with the server's public key ($pubK_S$), is sent separately. Since this is simply a 256-byte quantity, it is not defined as a distinct packet in the module but is transmitted as is.

$\{AUTH\_KEY\}_{pubK_S}$
256

The first packet structure formally defined in the packets module is the *start packet*, which is used for both the second part of $M1$ and for $M2$. The structure of this packet is as follows. The AAD field, totaling 670 bytes, is split into two sections: the first 654 bytes hold the peer's Diffie-Hellman public key, while the remaining 16 bytes store the client's nonce.

IV	AAD	TAG
12	670	16

Once the secure connection is established, all subsequent communication between the client and server follows a standardized format known as the *general packet*.

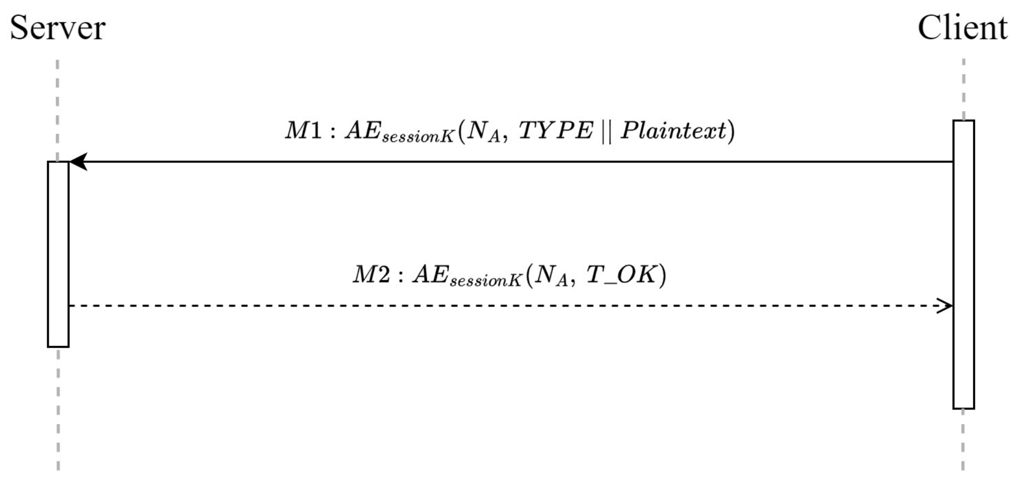
IV	AAD_SIZE	AAD	CIPH_SIZE	CIPH	TAG
12	4	AAD_SIZE	4	CIPH_SIZE	16

Here the encrypted content ($CIPH$) is produced using *AES-128 GCM* encryption, which secures the concatenation of the *operation type* and any associated *plaintext*, with the plaintext possibly being empty. The operation type, a single byte, facilitates efficient processing and clear differentiation between the various client-server operations.

3.1.5. Utility

The Utility module provides shared functions and helper methods used by both the client and server. These include cryptographic operations such as secure random number generation, *SHA-256* hashing, and public-key encryption and decryption. The module also implements the *Diffie-Hellman* key exchange with 1024-bit parameters, ensuring secure key agreement between client and server. Additionally, *AES-128 GCM* encryption and decryption are handled in this module. This algorithm is used for authenticated encryption of messages exchanged between the client and server, offering the benefit of no ciphertext expansion relative to the plaintext.

Robust *error handling* and *input validation* are integrated into all modules to ensure system security and prevent insecure behavior in the event of unexpected situations. For instance, if the server detects any anomalies or irregularities, it promptly closes the connection as a safeguard to preserve system security.



Server
Client
$M1 : AE_{sessionK}(N_A,\ TYPE \,||\, Plaintext)$
$M2 : AE_{sessionK}(N_A,\ T\_OK)$

Figure 3: General packet exchange process between client and server.

3.2. Client-Server Interaction

To understand how the BBS handles various operations, it is essential to examine the sequence and protocols involved in client-server *interactions.* This section outlines the general process for packet exchanges and provides specific examples, including user registration and adding a message to the BBS. Each of these examples demonstrates the system's adherence to security protocols and illustrates how operations are securely managed.

3.2.1. General Packet Exchange

The general packet exchange process is depicted in Figure 3. In this exchange, the $TYPE$ field identifies the operation being executed by the client and responded to by the server. Additional data requiring both confidentiality and integrity, referred to as *Plaintext*, is appended after the $TYPE$. Typically, the server responds with a T_OK message ($M2$), though there are special cases. For instance, adding a message requires a more specific protocol to handle potential replay attacks. Details of this specialized protocol are discussed in section *3.2.3. A Particular Operation: Addition of a Message.*

3.2.2. User Registration, an Example

Figure 4 illustrates how a client initiates a registration request to the server. While this example focuses on registration, the key aspects of the process are applicable to other operations as well.

Before any operation, such as registration, the simple connection shown in Figure 1 has already been established at the start of the client, as previously mentioned. Once the user presses the button to initiate the operation (1 in this case), and before exchanging any sensitive information, the secure connection described in section *2.3. Secure Connection* is established. Although Figure 4 does not show all details of the secure connection, it references messages $M1$ and $M2$ from Figure 2.

Once the user inputs their email, nickname, and password, the client sends a registration packet to the server. The server, in turn, sends a challenge to the specified email address to verify the user's identity through an authenticated secure channel. The user then retrieves the challenge from their email and submits it to the server to confirm their identity. If the challenge is correct, the server saves the user's information to the file system and sends a confirmation packet to the client, signaling that the registration was successful.

3.2.3. A Particular Operation: Addition of a Message

When adding a message, a specialized protocol is employed to mitigate potential replay attacks from the *client side*. Unlike operations such as *login* and *registration*, which are protected against replay attacks by using newly established session keys for each interaction, adding a message could result in *multiple instances* of the same message being posted to the BBS if replayed. This issue arises because the same session key is used throughout the session, which is established after a successful login. Operations such as *list* and *get* do not have significant side effects if replayed, so they are not considered in this context. However, in all operations, replay attacks from the *server side* are mitigated by using the nonce (N_A) included in the initial message. The client verifies that the nonce in the server's response matches the one it originally sent, following the standard protocol already presented. To prevent unintended duplications of messages, a more secure protocol is employed for adding messages, as depicted in Figure 5. This protocol is designed to ensure that replay attacks do not result in the unintended repetition of messages.

In this protocol, $sessionK$ used by AE refers to the session key established after the secure connection protocol. The $M1$ message from the client indicates the *intent* to add a message to the BBS but does not include the message itself. The server responds with an $M2$ message, which includes its own nonce (N_B) placed in the AAD field alongside the client's nonce. The client then sends the actual message in $M3$, which includes the server's nonce in the AAD. If no nonce mismatches occur and the message is successfully added to the BBS, the server informs the client of the success with the $M4$ (T_OK) message.

4. Conclusion

The Bulletin Board System developed provides a secure and reliable platform for users to communicate, adhering to best practices in modern cryptography and system architecture. The multi-threaded server design allows for concurrent client interactions while maintaining a high level of security, ensuring the integrity, confidentiality, and authenticity of all exchanges.

By leveraging cryptographic protocols such as Diffie-Hellman key exchange, authenticated encryption, and structured communication packets, the system guarantees the privacy and integrity of communications while remaining adaptable to future requirements. Its robust, modular design ensures scalability and maintainability, allowing for easy enhancements in both functionality and security as needed.

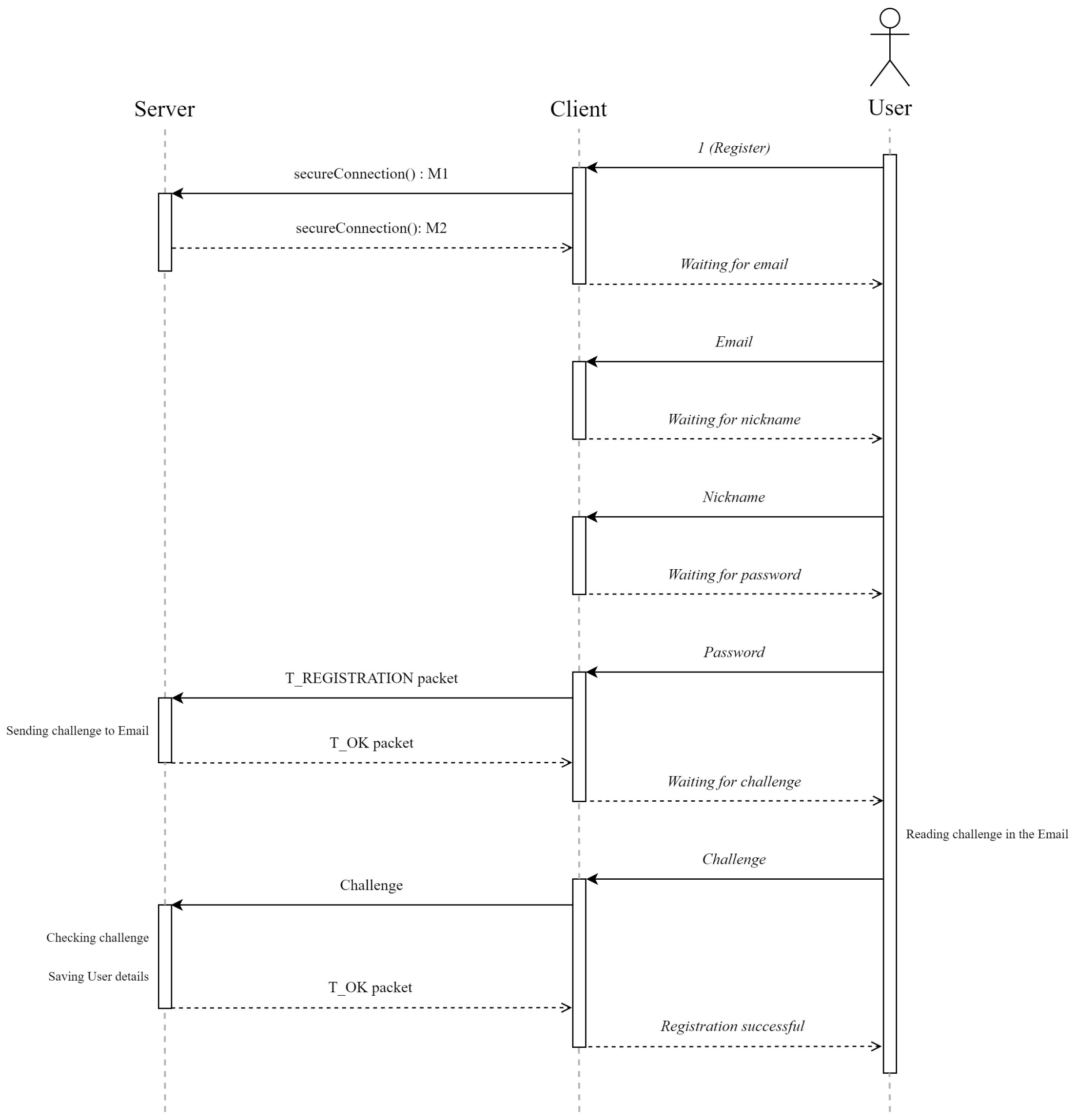


Figure 4: User Registration sequence diagram.

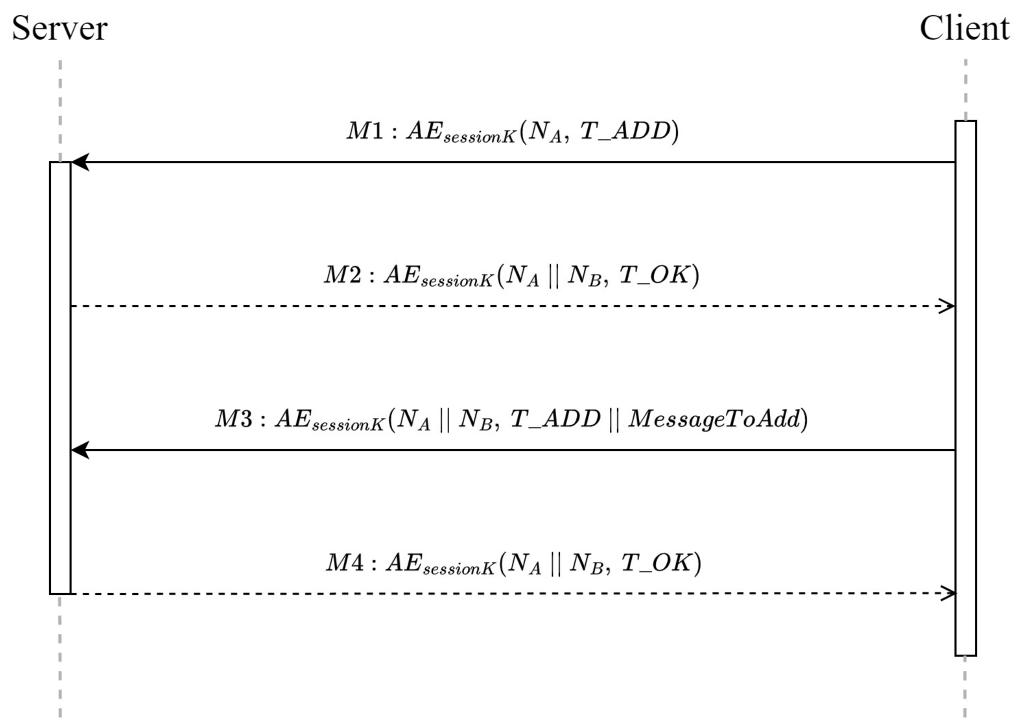

Server
Client
$M1 : AE_{sessionK}(N_A,\ T\_ADD)$
$M2 : AE_{sessionK}(N_A \,||\, N_B,\ T\_OK)$
$M3 : AE_{sessionK}(N_A \,||\, N_B,\ T\_ADD \,||\, MessageToAdd)$
$M4 : AE_{sessionK}(N_A \,||\, N_B,\ T\_OK)$

Figure 5: Protocol for securely adding a message to the BBS.