Hanoi University of Science and Technology

School of Information and Comunication Technology

Applied Cryptography

Project 2

Supervisor: PhD. Tran Vinh Duc
Class: 157216
Group: 7
Member: Dao Minh Quang 20225552
Vu Duc Thang 20225553
Le Anh Tuan 20225559

Hanoi, 3/2025

1. API Implementation

1.1. __init__(self, cert_authority_public_key, gov_public_key)

- Purpose: Initializes the MessengerClient object with the certificate authority's public key and the government's public key.
- Key steps:
    - Stores the CA public key and the government's public key
    - Initializes empty dictionaries to maintain:
        - Conns: Conversation state for each peer
        - Certs: Ceritficates of other users
    - Sets placeholders for the client's own ceritificates, private key, and username.
- Security features:
    - The CA public key is used to verify the integrity of incoming certificates.
    - State isolation per conversation helps ensure that compromising one session does not affect others.

1.2. generate_certificate(username: str) -> dict

- Purpose: Generates and returns a certificate for the client that will be registered with the certificate authority.
- Key steps:
    - Generates an ElGamal key pair using the provided cryptographic primitive.
    - Constructs a certificate dictionary containing:
        - username: The client's username
        - public_key: The generated public key
    - Stores the certificate and the corresponding private key locally.
- Security features:
    - Using ElGamal ensures that key exchanges are secure.
    - The certificate contains minimal necessary information (username and public key) to reduce exposure.

1.3. receive_certificate(certificate: dict, signature: bytes) -> None

- Purpose: Receives and verifies another user's certificate.
- Key steps:
    - Converts the certificate into a canonical string representation.
    - Uses the CA's public key and ECDSA verification to check the integrity and authenticity of the certificate.
    - Raises a ValueError("Tampering detected!") if verification fails.
    - Stores the valid certificate in an internal dictionary, keyed by the sender's username.
- Security features:

- Prevents adversaries from injecting or swapping certificates by relying on trusted signature verification.
- Ensures that only valid and untampered certificates are accepted.

1.4. send_message(name: str, plaintext: str) -> tuple[dict, tuple[bytes, bytes]]

- Purpose: Encrypts and sends a message to a specified recipient.
- Key steps:
  - Conversation Initialization:
    - If no conversation exists with the recipient, performs a Diffie–Hellman exchange using the sender's private key and the recipient's public key to derive two chains via HKDF.
  - Ratchet Update:
    - Uses a ratchet mechanism to update the sending chain and derive a fresh message key for each message.
    - Maintains a message counter (msg_num) to track key progression.
  - Government Encryption:
    - Generates an ephemeral key pair and performs a DH exchange with the government's public key.
    - Derives an AES key using HMAC and encrypts the newly derived message key.
    - Embeds the government encryption outputs (ephemeral public key, IV, and ciphertext) in the message header.
  - Message Encryption:
    - Generates a random IV (the "receiver_iv") for encrypting the plaintext using AES-GCM.
    - Uses the message key to encrypt the plaintext with the header (which includes sender, recipient, message number, and government fields) as authenticated data.
  - State Update:
    - Updates the send chain and increments the message counter.
- Security features:
  - Provides forward secrecy by updating keys on every message.
  - Incorporates government decryption information without compromising user confidentiality.
  - Uses authenticated encryption (AES-GCM) to guarantee integrity and prevent tampering.

1.5. receive_message(name: str, message: tuple[dict, tuple[bytes, bytes]]) -> str

- Purpose: Decrypts an incoming message from a specified sender.

- Key steps:
    - Header Verification:
        - Checks that the message header’s "recipient" field matches the client’s username.
        - Validates that the header includes the sender information and a message number.
    - Conversation State Initialization:
        - If no state exists for the sender, initializes the conversation using a reversed role in the Diffie–Hellman exchange to derive the receive and send chains.
    - Handling Out-of-Order and Replay Protection:
        - Maintains a receive counter and uses a “skipped” keys dictionary to store keys if messages arrive out-of-order.
        - Checks the message number against a set of processed numbers to detect and reject replayed messages.
    - Message Decryption:
        - Uses the appropriate message key (either freshly derived or retrieved from skipped keys) to decrypt the ciphertext with AES-GCM, using the header as authenticated data.
    - State Update:
        - Updates the receive chain and increments the receive counter.
        - Marks the message number as processed.
- Security features:
    - Ensures that messages not intended for the client are rejected.
    - Provides replay attack protection by tracking message numbers.
    - Supports break-in recovery and forward secrecy through continuous key updates.
    - Handles out-of-order messages by storing skipped keys, ensuring no valid message is lost due to shuffling.

2. Short-answer Questions

2.1. Could the protocol be modified to have them increment the DH ratchets once every ten messages without compromising confidentiality against as eavesdropper?

Yes, in theory the encryption scheme (AES-GCM) remains semantically secure even if one key encrypts multiple messages—as long as a unique IV (nonce) is used for each encryption. In such a modification, the same DH key would be used for ten messages. However, while semantic security might hold, this approach enlarges the window in which a key is used. That means if a key were compromised, more messages (up to ten) would be exposed. In contrast, the per-message ratchet greatly limits potential exposure and also underpins forward secrecy and break-in recovery. Therefore, while you could update every ten messages without losing semantic security, it would weaken the overall forward secrecy guarantees.

2.2. What if they never update their DH keys at all? What are the security consequences regarding Forward Secrecy and Break-in Recovery.

If the DH keys are never updated, the protocol loses two critical properties:

- Forward Secrecy: Without key updates, the compromise of a long-term DH key or even a current session key would allow an attacker to decrypt all previous messages, since the same key (or a fixed derivation from it) is used for every message.
- Break-in Recovery: In the absence of periodic DH ratchet updates, once an adversary gains access to the keys, they could continuously decrypt future messages. The protocol would not "recover" security even after new messages are sent, because the same static keys would be used indefinitely.

2.3. Given the conversation. What is the length of the longest sending chain used by Alice? By Bob? Please explain

In a Double Ratchet setup each time a party sends a message, it updates its sending key:

- Alice: Alice sends three messages ("Hey Bob…", "I need…", and "Great, thanks…"). Therefore, her sending chain is incremented three times; its longest length is **3**.
- Bob: Bob sends two messages ("Sure, it's 1234 !" and "Did it work?"). His sending chain, therefore, reaches a length of **2**.

2.4. Unfortunately, in the situation above, Mallory has been monitoring their communications and finally managed to compromise Alice's phone and steal all her keys just before she sent her third message. Mallory will be unable to determine the locker combination. State and describe the relevant security property and justify why double ratchet provides this property.

The relevant property is Forward Secrecy:

- Forward Secrecy: Even though Mallory obtains all of Alice's keys at the time of compromise, the Double Ratchet protocol ensures that previous message keys have been securely discarded. As a result, past communications – including the message containing the locker combination – remain confidential.
- Break-in Recovery: Once a new message is sent after the compromise the session's security "recovers" and future messages are protected even though earlier keys were compromised.

2.5. The method of government surveillance is deeply flawed. Why might it not be as effective as intended? What are the major risks involved with this method?

There are several reasons and risks:

- Limited Practicality and Circumvention: While the government is provided with an encrypted version of the session key, the end-to-end encryption is still maintained. Adversaries (or even users) might design countermeasures, such as protocol modifications or obfuscation techniques, that make it difficult for the government to reliably decrypt messages
- Key Management Risks: Storing and managing the government's private key becomes a critical point of failure. If this key is compromised, a large amount of sensitive communication could be exposed.
- Privacy and Abuse Concerns: Mandating government decryption can be a slippery slope toward mass surveillance, which not only compromises user privacy but also risks abuse by authorities.
- Operational Risks: Implementing an extra encryption layer increases protocol complexity, which might inadvertently introduce vulnerabilities or performance issues, reducing overall system security.

2.6. Compare ECDSA and RSA-based signature techniques:

a. Which keys take longer to generate?
RSA key generation (especially for 4096-bit keys) is significantly slower than ECDSA key generation due to the computationally intensive operations required for large prime generation and modular arithmetic.

b. Which signature takes longer to generate?
   RSA signing tends to be slower than ECDSA signing. In our experiments (as shown by the timing in question_6_code.py), RSA signing (using PSS padding) generally takes more time because of the large key size and the required modular exponentiations, whereas ECDSA signing is relatively fast.
c. Which signature is longer in length?
   RSA signatures are substantially longer (often around 512 bytes for a 4096-bit key) compared to ECDSA signatures, which are typically around 70–100 bytes. This makes ECDSA more bandwidth-efficient.
d. Which signature takes longer to verify?
   Verification times can vary, but generally RSA verification can be slower than ECDSA verification. Although RSA verification sometimes benefits from optimizations (like using a small public exponent), the overall cost of handling large keys and the signature padding scheme typically makes ECDSA verification faster.