COMP 537: Cryptography **Fall 2024**

Programming Assignment 2: Private Messaging

In this assignment, you are tasked with implementing a secure and efficient end-to-end encrypted chat client using the Double Ratchet algorithm, a popular protocol that powers real-world chat systems such as Signal and WhatsApp. As an additional challenge, you will include a "message reporting" feature that encrypts a message that a user reports under a public key provided to you by the platform, so that a moderator who works for the platform can review abusive messages. In your implementation, you will make use of various cryptographic primitives we have discussed in class – notably key exchange, public key encryption, digital signatures, and authenticated encryption. Because it is ill-advised to implement your own primitives in cryptography, you should use an established library. In this case, we will use the Python `cryptography` library. You will be provided with starter code that contains a basic template, which you will be able to fill in to satisfy the functionality and security properties described below.

Acknowledgments. This assignment is adapted from a similar assignment from Stanford's CS255 course by Prof. Dan Boneh.

1 End-to-End Encrypted Chat Client

1.1 Implementation Details

Your chat client will use the Double Ratchet algorithm to provide end-to-end encrypted communications with other clients. To evaluate your messaging client, we will check that two or more instances of your implementation can communicate with each other properly.

We feel that it is best to understand the Double Ratchet algorithm straight from the source, so we ask that you read Sections 1, 2, and 3 of Signal's published specification here: `https://signal.org/docs/specifications/doubleratchet/`. Your implementation must correctly use the Double Ratchet algorithm as described in Section 3 of the specification, with the following changes and clarifications. Feel free to use the code samples provided in the documentation as part of your solution.

- You will use HKDF with SHA256 to ratchet the Diffie-Hellman keys. Proper usage of HKDF is explained in Section 5.2 of the specification.
- You will use HMAC with SHA256 to implement the symmetric key ratchet.
- You will use AES-GCM as the symmetric authenticated encryption algorithm.
- You will use curve P-256 as the elliptic curve for all public key operations.
- disregard the AD byte sequence input for the `ratchetEncrypt` and `ratchetDecrypt` functions in the Signal specification. Message headers should still be authenticated.
- You do not need to handle and recover from dropped or out-of-order messages. You do not need to worry about Section 2.6 of the specification. You can assume that a message being sent and received together constitute an atomic operation, meaning if a message is sent, the very next action that

occurs is that the same message will be received by the other party. It will never be the case that two messages are sent before the first one is received. If you detect that a message has been blocked, dropped, or received out of order, return None.

- The memory cost of key storage for your algorithm should always be O(1) and independent of the number of messages sent. In order to satisfy this, your implementation will discard old keys whenever a ratchet occurs.
- every client must create an initial DH key pair. They keys will be used to derive initial root keys for new communication sessions and as the initial public key ratchet. Note that this means the root key and the first DH key exchange output will be the same in your implementation.
- Public keys will be distributed through simple certificates. Each client generates its own certificate upon initialization which contains its public key. The messaging server will receive certificates generated by clients and sign them (with ECDSA signatures). The platform's signature on each certificate serves to endorse the authenticity of the certificate owner's identity and to prevent any tampering with the public key by an adversary. The signed certificates are then distributed to the clients as needed so that each client can access the public keys of other clients using the system.
- Users can report messages that they feel abuse the platform. To report a message, the client will encrypt the name of the sender and the message under the messaging server's public key and send it to the server. A moderator who works for the messaging service will then be able to decrypt the message and check whether or not it violates platform policies. Reported messages will be encrypted with the CCA-secure variant of El-Gamal encryption. Note that El-Gamal encryption is not provided by the `cryptography` library, so you will have to use the algorithms that the library does support to implement it.

1.2 Threat Model

The goal of the double ratchet algorithm is to provide *forward secrecy*: compromise of long term keys or current session keys must not compromise past communications. Specifically, consider a person-in-the-middle attacker Eve who sits between Alice and Bob. Eve sees every encrypted message passed between Alice and Bob and writes all of them to persistent storage. Then at some point, Alice's device is compromised and Eve learns Alice's current secret keys. Assume that Alice has deleted her keys for old messages, as encouraged by the Signal specification. Your implementation must ensure that under this scenario, Eve cannot decrypt any of the past messages in her persistent storage despite having full access to Alice's current keys.

After Alice's keys are compromised, the adversary can now launch an active person-in-the-middle attack where she can impersonate Alice or eavesdrop on all future messages. However, if Alice manages to send a single message to Bob without the attacker being able to intercept it, your implementation must ensure that the attacker loses all ability to decrypt communications once again. This property is called *break-in recovery*.

Implementing the double ratchet algorithm as described in the Signal documentation is sufficient to ensure both of these properties.

2 API Description

Here are the descriptions of the methods you will need to implement.

2.1 `Client.generateCertificate()`

This method should initialize the messenger client for communication with other clients by generating the necessary Diffie Hellman key pair for key exchanges. The public key should be placed into a certificate to send to other clients. You are free to design your own certificate object, so long as it includes the name of the user and the corresponding public key.

2.2 `Client.receiveCertificate(certificate, signature)`

This method takes a certificate from another client and stores it in the messenger's internal state, so that the client can now send and receive messages from the owner of that certificate. The second argument is the messaging server's signature on the certificate. You must verify the validity of the signature (using the server's public key, provided to you in the client's constructor) to ensure that the certificate has not been modified by an adversary. If you detect tampering, throw an exception.

2.3 `Client.sendMessage(name, message)`

Send an encrypted message to the user specified by name. You can assume that communicating users have already received each others' certificates. If you have not previously communicated, set up the session by generating the necessary double ratchet keys according to the Signal spec. The process of sending messages will follow the Signal specification as well.

2.4 `Client.receiveMessage(name, header, ciphertext)`

Receive an encrypted message from the user specified by name. You can assume that communicating users have already received each others' certificates. If you have not previously communicated, set up the session by generating the necessary double ratchet keys according to the Signal spec. The process of receiving messages will follow the Signal specification as well. If tampering is detected in any way, return `None`. Otherwise, return the message that was received.

2.5 `Client.report(name, message)`

This method creates an abuse report with the provided name and message. The structure of the report plaintext is up to you as long as it clearly allows a human moderator to see the name of the sender and the message contents. The report plaintext is encrypted under the encryption public key of the messaging server, provided in the client's constructor. You will use a CCA-secure El-Gamal encryption scheme to encrypt the report. To enable testing, this function has two outputs, the report plaintext and the report ciphertext.

2.6 `Server.signCertificate(certificate)`

This method signs a provided certificate with the server's signing key, which is provided to you in the server's constructor. You will sign the message with an ECDSA signature using SHA256 as the hash.

2.7 `Server.decryptReport(ct)`

This method decrypts and returns an abuse report using the server's private decryption key, which is provided to you in the server's constructor.

3 Additional Instructions

The setup for this assignment is similar to that of Programming Assignment 1. In particular, you will find starter code in `messenger.py` that you will modify and which should pass the tests in the testing script `main.py`. As before, we will be using Python 3 for this assignment. You are encouraged to design additional test cases to evaluate the correctness and security of your implementation.

You cannot change the signatures of the methods we provide, as your implementation must work with our provided `main.py` script. That said, you are welcome (and encouraged) to add additional helper methods in your implementation.

Your implementation can only make use of standard Python modules and the `cryptography` library. You will be using the library's "hazardous materials" layer. For serialization and deserialization of basic Python data structures (including dictionaries, lists, strings, byte-arrays, etc.), you can use the `pickle` library.

The number of lines of code you need to write should be modest. As a point of reference, our reference solution file is 200 lines of Python code, and the diff with the starter code is even smaller:

```
$diff -y --suppress-common-lines base/messenger.py solution/messenger.py  | wc -l
  169
```

4 Short-Answer Questions

In addition to your implementation, please include *short* responses (e.g., 1-5 sentences) to the following questions regarding your design and implementation. You do not need to give formal proofs, but you should be precise and include important details in your responses.

1. In our implementation, Alice and Bob increment their Diffie-Hellman ratchets every time they exchange messages. Could the protocol be modified to have them increment the DH ratchets once every ten messages without compromising confidentiality against an eavesdropper (i.e., semantic security)?

2. What if they never update their DH keys at all? Please explain the security consequences of this change with regard to forward secrecy and break-in recovery.

3. The message reporting feature included in our messaging scheme is not adequate for use on an actual messaging platform. What is one shortcoming of our approach?

4. Our messaging system relies on the platform to verify the authenticity of users' public keys, but we are also trying to give users confidentiality from the platform. Using this partial trust we place in it, how could a malicious platform learn the contents of a message one user is sending to another?

5. How do end-to-end encrypted messaging apps help users avoid this problem? Feel free to look this up or explore one of these messaging apps to answer this question.

6. **Optional feedback.** How much time did you spend on this assignment? Did you find it too easy/hard or just right?

7. **Optional feedback.** Please let us know if you have any feedback on the design of this assignment or on the course in general.

Please submit your responses as a PDF file `answers.pdf` with your submission.

5 Submission Instructions

To submit your assignment, upload the following two files to Gradescope:

- `messenger.py`: this file contains your implementation of the messaging code.
- `answers.pdf`: this file contains your answers to the short answer questions.

Do *not* submit any other files with your submission. If submitting with a partner, please have one partner submit on behalf of both of you, and please include your names in a comment at the top of the python file.

Grading: The short answer questions are each worth 4 points for a total of 20 points. There are 20 points of automated tests (contained in `main.py`), and there are additionally 10 points of code review for security, making a total of 50 points.