Sharif University of Technology
Faculty of Mathematical Sciences

Introduction to Cryptography — Winter 2021

Secure Messaging Project

Instructor: Prof. Khazaei — Deadline: July 10, 2021

Important Notes

Please pay attention to the following points first:

- Use *JAVA* version 8 for implementing this assignment.
- Submit your final answer as a *ZIP* file only in the *Courseware* system under the programming project section. Submitting answers via email or messaging apps is not acceptable.
- Sharing your project-related code with anyone other than the instructor and teaching assistants (even for debugging) is not allowed. You are also not allowed to use code from the internet or other people's code.
- All answers submitted for this project in the system, from the moment of submission, transfer intellectual property and publication rights to this course, and you do not have the right to publish your answers anywhere (even to your friends) without the direct permission of the course instructor.
- For questions regarding the project, contact amir77ni@gmail.com. The notes provided for emailing in the main course description file also apply here.

Good luck.

Introduction

In this course, as a programming project, you are tasked with implementing some of the algorithms and systems you have learned in the course in the form of a messaging application. The graphics and networking sections of this messaging application have already been designed and provided to you as initial code. The main parts you are expected to implement in the project are as follows:

1. Public Key Encryption System *RSA*
2. Symmetric Encryption System *AES256*
3. Key Exchange Protocol *Diffie–Hellman*
4. Pseudo-Random Number Generator based on *RC4*
5. Hash Function *SHA256* (256-bit version of *SHA-2*)

For implementing some of the aforementioned items (such as the *RSA* system), you will also need to implement functions for working with large numbers and checking their primality. Implement the primality test based on the *Miller-Rabin* test. For factoring numbers into two prime factors, implement the *Pollard's Rho* algorithm.

The features and capabilities of this messaging application are as follows:

- This messaging application has only one server.
- Each user has a unique username and password.
- Each individual can join the messaging application by selecting a username and password.
- Each user can be connected to the messaging application through only one device at a time.
- Users can add any other user to their contact list by knowing their username.
- Each user can send messages to any user in their contact list.
- To send a message to another user, the messages are first sent to the server, and the server forwards them to the recipient.
- Communication between the user and the server is end-to-end encrypted.
- User information is securely stored on the server.

Part of the project, which includes some of the cryptography basics required for the project, must be completed by a specific deadline as an interim submission. The list of required items is provided below, and their features must match the requirements set forth for the remainder of the project.

- Symmetric encryption system *AES256* with *CBC* mode, including encryption, decryption, and key generation functions.
- Functions needed for working with large numbers (including subtraction, multiplication, modular exponentiation, modular multiplicative inverse, and GCD).
- *Miller-Rabin* primality test for large numbers.
- *Pollard's Rho* algorithm for prime factorization.

Explanation

Server Connection Protocol

In this protocol, two 128-bit random numbers, one chosen by the server and the other by the user, are used to distinguish the connection between the server and different users. These two numbers are referred to as *nonce* and *serverNonce*, and will be included in all messages except the first message, where *serverNonce* has not yet been selected, in an unencrypted form.

In this protocol, the communications are as follows:

1. A user sends a connection request along with a 128-bit random number to the server.

$$\mathcal{M} = \{\text{nonce} : \text{int128}\}$$

2. The server sends a message containing the user's selected number, another 128-bit random number chosen by the server, the server's public key fingerprint, and a challenge number to the user. The challenge number is the product of two 30 or 31-bit prime numbers, which is used to counter denial-of-service attacks. The user must factor this number into its prime factors to prove the legitimacy of their request. The challenge number must be generated randomly and be different for each message. Additionally, the server's public key fingerprint is sent for verification against the pre-existing key in the user's program.

$$\mathcal{M} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{publicKeyFingerprint} : \text{string}, \text{pq} : \text{string}\}$$

3. The user factors the challenge number into its two prime factors and sends them along with the random numbers from the previous step and encrypted content to the server. The encrypted content includes the username and password and a new 256-bit random number along with the random numbers from the previous step, encrypted using the server's public key, which the user received the fingerprint of in the previous step. Encryption is done using *RSA*.

$$\text{data} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{newNonce} : \text{int256}, \text{username} : \text{string}, \text{password} : \text{string}\}$$

$$\text{encryptedData} = \text{RSA}((\text{SHA256}(\text{data}) + \text{data} + \text{padding}), \text{serverPublicKey})$$

$$\mathcal{M} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{p} : \text{string}, \text{q} : \text{string}, \text{encryptedData}\}$$

4. The server computes the temporary *AES* key and IV using the exchanged random numbers. It randomly selects a number a and a generator g and sends a message containing the value of g and g^a of the *Diffie–Hellman* protocol, encrypted with the computed key, to the user.

$$\text{tmpAESKey} = \text{SHA256}(\text{newNonce} + \text{serverNonce})[0..15]$$

$$+\text{SHA256}(\text{serverNonce} + \text{newNonce})[16..31]$$

$$\text{tmpAESIV} = \text{SHA256}(\text{serverNonce} + \text{newNonce})[0..15]$$

$$\oplus\text{SHA256}(\text{newNonce} + \text{newNonce})[0..15]$$

$$\text{answer} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{g} : \text{string}, \text{gA} : \text{string}\}$$

$$\text{answerHashed} = \text{SHA256}(\text{answer}) + \text{answer} + \text{padding}$$

$$\text{encryptedAnswer} = \text{AES256CBC}(\text{answerHashed}, \text{tmpAESKey}, \text{tmpAESIV})$$

$$\mathcal{M} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{encryptedAnswer} : \text{string}\}$$

5. The user also computes the temporary *AES* key and IV. They select a random number b and send g^b of the *Diffie–Hellman* protocol, encrypted with the computed key, to the server.

$$\text{answer} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{gB} : \text{string}\}$$

$$\text{answerHashed} = \text{SHA256}(\text{answer}) + \text{answer} + \text{padding}$$

$$\text{encryptedAnswer} = \text{AES256CBC}(\text{answerHashed}, \text{tmpAESKey}, \text{tmpAESIV})$$

$$\mathcal{M} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{encryptedAnswer} : \text{string}\}$$

6. Finally, the server sends a message to confirm the completion of the protocol.

$$\text{responseHash} = \text{SHA256}(\text{newNonce} + \text{SHA256}(\text{gAB})[0..7])$$

$$\mathcal{M} = \{\text{nonce} : \text{int128}, \text{serverNonce} : \text{int128}, \text{responseHash} : \text{string}\}$$

The symbol *X[a..b]* means the a-th to b-th bytes of variable X, starting from zero, and the symbol $a + b$ means the concatenation of two strings a and b, and the symbol $a \oplus b$ means the bitwise *XOR* operation between two strings a and b.
From now on, the exchanged key will be referred to as *authKey*.
Finally, both parties calculate the following two variables and use them in message transmissions.

$$\text{salt} = \text{newNonce}[0..7] \oplus \text{serverNonce}[0..7]$$

$$\text{authKeyId} = \text{SHA256}(\text{authKey})[0..7]$$

Message Sending Protocol

When you want to transmit a message with the content *messageData* from the server to the user or vice versa, follow this protocol. First, the variables and functions used in this protocol are introduced, and then the rest of the calculations to obtain the transmitted package $\mathcal{M}$ are outlined. A schematic of this protocol is also provided in Figure 1.

- The function *unixtime* outputs the current time in the Unix timestamp format. The unit of this format is seconds, and its value is equal to the number of seconds that have passed since the start of 1970. You can use the function *System.nanoTime* to compute it.
- The variable *messageId* is used to uniquely identify a message between two users, and its value is 2^{32} times the value of *unixtime* plus a random number. This random number can be between 2 and 2^{30}, ensuring that the final calculated value of *messageId* is greater than the *messageId* of previously sent messages and is even for messages sent by the user and odd for messages sent by the server.

$$\text{messageId} = \text{unixtime}() \times 2^{32} + \text{randomNumber}$$

- The variable *sequenceId* equals the sent message's number. This number is initialized to 0 for the user and 1 for the server at the beginning of the key exchange with the server and is increased by 2 units for the sender after each sent message. Therefore, this number is always odd for messages sent by the server and even for the user.
- The variable *messageDataLength* is the length of *messageData* in bytes.
- The value x is equal to 0 for messages sent by the user and 8 for messages sent by the server.

1. Calculations for the message's structure and key:

$$\text{data} = \{\text{salt} : \text{int64}, \text{messageId} : \text{int64}, \text{sequenceId} : \text{int64},$$

$$\text{messageDataLength} : \text{int32}, \text{messageData} : \text{bytes}, \text{padding} : \text{bytes}\}$$

$$\text{msgKey} = \text{SHA256}(\text{authKey}[88 + \text{x}..119 + \text{x}] + \text{data})[8..23]$$

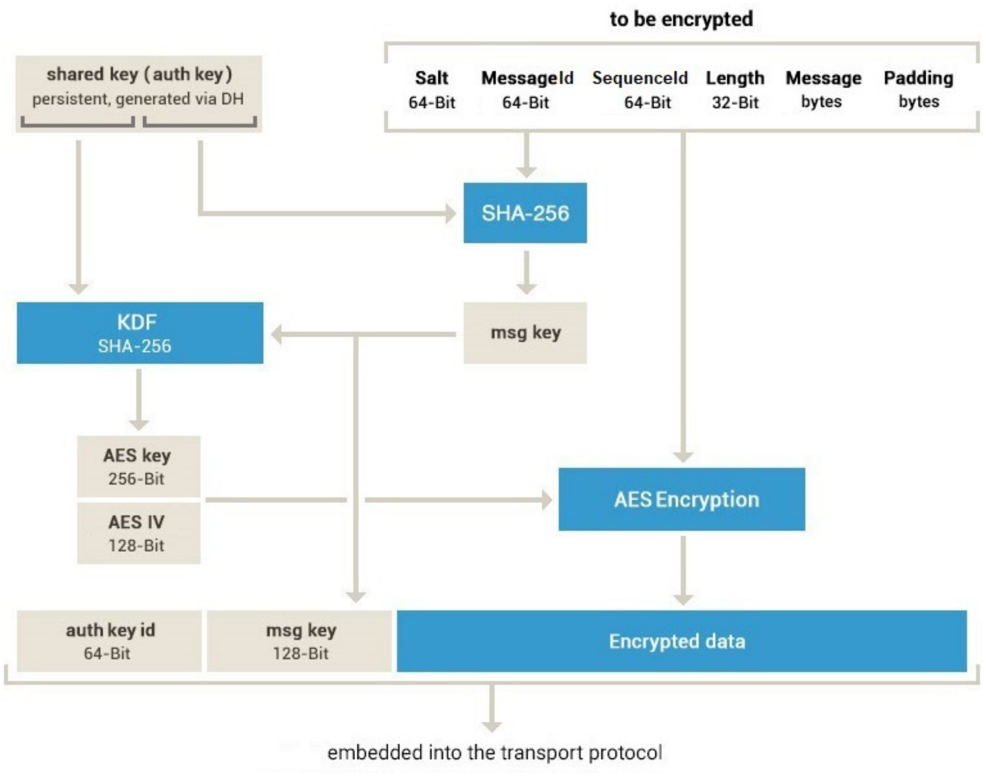

Figure 1: Message Sending Protocol

2. Calculations for generating the values needed to encrypt the message:

$$\text{sha256A} = \text{SHA256}(\text{msgKey} + \text{authKey}[\text{x..x} + 35])$$

$$\text{sha256B} = \text{SHA256}(\text{authKey}[40 + \text{x..}75 + \text{x}] + \text{msgKey})$$

$$\text{AESKey} = \text{sha256A}[0..7] + \text{sha256B}[8..23] + \text{sha256A}[24..31]$$

$$\text{AESIV} = (\text{sha256B}[0..7] + \text{sha256B}[24..31]) \oplus \text{sha256A}[8..23]$$

3. Calculations for generating the final sent message:

$$\text{encryptedData} = \text{AES256CBC}(\text{data}, \text{AESKey}, \text{AESIV})$$

$$\mathcal{M} = \{\text{authKeyId} : \text{int64}, \text{msgKey} : \text{int128}, \text{encryptedData} : \text{bytes}\}$$

Secure Storage

The server needs to verify the correctness of the user's password during key exchange, which requires storing some information about the user's password. For enhanced user security, passwords are not stored in plaintext but as a hash, and the server checks the received password against the stored hash by hashing it. This method ensures that even if the information related to user passwords stored on the server is compromised, users remain secure. It is obvious that the method used for hashing is crucial for security. You will use the *SHA256* hashing algorithm with some modifications in this implementation. To counter the rainbow table attack, which relies on precomputed hash-to-plaintext mappings, use a *salt*.

If you intend to store a password contained in the variable x, first generate a 128-bit random variable y, concatenate it with x, and then repeatedly hash the result 1024 times. When storing the resulting hash, also store the y variable so that you can verify the user's password against the stored hash in the future. Note that the number of hash repetitions is not necessarily fixed and is one of the system's security parameters; 1024 is just a suggested value.

The server should also not keep the contact list of each user in its memory (RAM) and must store it as a file in external storage. This storage should not be in plaintext, but unlike the password, it should not be one-way and must allow content retrieval. The choice of encryption system and file storage format is up to you, but all mentioned security considerations must be followed.

Details

- If a user is active and an input request with their username is received, the request is ignored. An active user is one who has not sent a logout message to the server after their last login and has sent a message to the server within the last 20 minutes. If a user has not sent a message to the server for 20 minutes, their identity key is invalidated, and they need to log in again to send messages.
- The server does not keep a history of messages. If a user is inactive, messages sent to them are ignored.
- In all parts where *padding* has been added to a variable, its size is between 1 and 16 bytes so that the final size of the resulting variable is a multiple of 16. Use the padding method provided in *PKCS#7*.
- In the third step of establishing a connection, after receiving the message from the server, if a user with that username already exists, the server must check the correctness of their password. If the username does not exist, create the user with the received password and proceed with the protocol.
- At each step of the protocols, after receiving a message from the other party, the recipient must verify all necessary aspects depending on the received message (including password correctness or challenge number factorization in the third step of the connection protocol). If the verification fails, they must send the following message and terminate the connection, invalidating any identity keys used in the connection.

$$\mathcal{M} = \{data = \text{'error'}\}$$

- For convenience, you can prepend a 4-byte string to the beginning of all messages ($\mathcal{M}$) to indicate the type of message. (The type of message can be connection initiation, login request, logout request, etc.)

Implementation

The client program is located in the *Client* folder, and the server program is in the *Server* folder. The client program includes graphics and networking code, and the server program includes command-line execution and networking code.

In each of the two folders, there is a file named *Core.java*, which is the interface between the initial code and the part you will implement. You are allowed to add other files to the folders but cannot modify the initial code provided, except for the *Core.java* files. You can also add other functions to this file but are not allowed to change the names, inputs, or output formats of the pre-existing functions in this file. Functions in this file that return *boolean* should return a value indicating whether the command was successful. More detailed information about these functions will be provided to you later.

Client Side

To send a message to the server, use the *ChatClient.connection.sendData* function, which takes an array of bytes as input, and its input is directly passed to the server's *receiveData* function.

To display messages to the user, there is a function named *receiveMessage* in the *ChatClient* class that must be called with the correct *Message* type input for each message received while the user is active.

On the client side *Core.java* file, there are seven functions that you need to complete correctly. The operation of these functions must comply with the stated protocols. The timing of each of them being called is as follows:

- *receiveData*: This function is called with the received content as input whenever a message is received from the server.
- *loginOrRegister*: This function is called with the username and password values when a user attempts to log in or register in the graphical interface.
- *logOut*: This function is called when the user attempts to log out from the application in the graphical interface, before exiting, to allow you to perform the necessary operations, such as invalidating the current identity key.
- *getMessages*: This function is called with the recipient's username as input when the user opens the message sending page for a contact and should return the last 100 messages exchanged between the current user and the contact in order of *messageId*. (If fewer than 100 messages were exchanged, the array will only contain all the existing messages.)
- *getContacts*: This function is called when the user opens the contact list page and should return the current user's contact list as an array of usernames. (The array may be empty.)
- *addContact*: This function is called with the username of the user the current user wants to add as a contact in the graphical interface.
- *sendMessage*: This function is called with the message text and the recipient's username as input when the user attempts to send a message to another user in the graphical interface.

Server Side

To send messages to users, use the *ChatServer.connection.sendData* function, which takes an array of bytes and the user's unique number as input. The input is directly passed to that user's *receiveData* function. The user's unique number is connection-dependent and may change for a user in different connections.

On the server side *Core.java* file, there are two functions that you need to complete correctly. Their operation must be as follows:

- *receiveData*: This function is called with the message sent and the user's unique number as input when data is received from the user. Distinguishing the user who sent the message should only be done using the information contained in the message, as described in the protocols. This function should perform the necessary operations according to the protocol.

- *generateRSAKeyPair*: This function generates an *RSA* key pair and returns it as an array of *String* where the first element is the private key, and the second element is the public key.

Notes

- You are only allowed to use the *java.io* library in your code for writing to files and are not allowed to use any other *JAVA* libraries, including all classes in *java.math, java.util, javax.crypto, java.security*, etc. If you use them in any part of the program, points for that part will be deducted.
- In your implementation, there should be functions for executing the *RSAKeyGen*, *RSAEncrypt*, *RSADecrypt*, *AESEncrypt*, *AESDecrypt*, *RC4*, *SHA256*, *Miller-Rabin*, and *Pollard's Rho* algorithms that are specified and independently usable. The input and output of these functions should match the structures provided in the course textbook, and the security parameters for all implementations should be easily adjustable. The functions' inputs should be in text string format. The *AES* functions, like their implementation in the protocol, should be 256-bit with *CBC* mode.
- In the final program, it must be able to function correctly as a messaging application, and all its features should be operational. The program should match the project document in the specified areas, but for unspecified areas (such as file storage format), you have the freedom to choose.
- The speed of the large number handling functions you implement greatly impacts the time taken to establish a connection with the server, so you must use the fastest available algorithms for implementing these functions. For example, you can use divide-and-conquer-based algorithms (like *Karatsuba* and *Burnikel-Ziegler*) for multiplication, exponentiation, and division, and use the extended Euclidean algorithm for computing the multiplicative inverse.
- At the end of the project implementation, to test its execution, generate an *RSA* key pair with parameter $n = 512$ for use in the connection protocol, and include it in the client and server code for use. This key should be easily replaceable with the key generated by *generateRSAKeyPair*.
- At the start of execution, initialize the random number generator *RC4* that you implemented with an initial *seed* and use it whenever a random number is needed in the program.
- In client-side functions that require a response from the server to calculate the output, you can use the *wait* and *notify* functions to implement the wait for a response.
- The project has an interim submission and a final submission, both of which are online, and their scheduling will be announced later.