CSCI4230 Project Documentation

Aidan McHugh, Kai Orita, and Bradley Presier

April 2023

GitHub

Note: If links do not work, try opening this file from its directory using Edge or Chrome.

Contents

1 Overview

This project implements a bank ATM system, allowing users to deposit, withdraw, and view their balance through an ATM client. Due to the lack of a physical machine, no real cash can be exchanged, but we assume for our purposes that this occurs if the ATM client is not an impostor.

The implementation operates on three levels with different protocols, which are

1. TCP - enabling a connection between client and server
2. TLS/SSL - authenticating and encrypting communication between client and server
3. Application - authenticating the user and providing ATM functionality

Note that we consider the server, client, and user as three parties who must each be authenticated at some level. Also note that the TLS/SSL and Application levels each have their own mode of error handling, which will be discussed in their respective sections.

2 Usage

2.1 Setup

This project requires Python 3, and has not been tested with versions earlier than Python 3.10.0. We compile and build local Python packages. This should not be an issue in most Python 3 installations.

To install dependencies,

1. (optional) Create a virtual environment of your choice.
2. In this directory, run `pip install -r requirements.txt`

2.2 Run Application

1. Run `python server.py` to start the server.
2. Run `python client.py` to start a client instance.

2.3 Provided Accounts

We have provided several accounts in the database that team blackhat might use or attack:

Mallory Malificent

This is you!

Card Number	`0000000000000000`
CVC	`666`
Expiration Date	`04/2025`
PIN	`6969`

Charlie Collaborator

If you steal from Charlie you're a bad friend. But maybe they'll let you intercept their messages for science.

Card Number	`0000000000000505`
CVC	`111`
Expiration Date	`05/2025`
PIN	`1111`

Alice Allison

Alice keeps her bank information very secret.

Card Number	(random)
CVC	(random)
Expiration Date	`05/2025`
PIN	(random)

Bobby McBobface

Bobby is less good at keeping secrets than Alice.

Card Number	`0505050505050505`
CVC	`123`
Expiration Date	`06/2023`
PIN	(random)

Victor Evilson

Maybe you feel bad about stealing from Bobby. Victor is very evil so the only issue with stealing from him is that he might come find you.

Card Number	`4111111111111111`
CVC	(random)
Expiration Date	`09/2026`
PIN	(random)

Billy Bazillionaire

Billy has a lot of money. He probably wouldn't miss it if some disappeared, right?

Card Number	(random)
CVC	(random)
Expiration Date	`12/2100`
PIN	(random)

2.4 Structure Overview

This project is implemented using Python 3 and C++. Specifically, Python is primarily used while an included Python package is implemented using C++. Throughout this project, we have tried to be thorough in In the following, we will describe the different components:

2.4.1 Client/Server

The client (`client.py`) and server (`server.py` and `database.py`) exist in the main directory of this project. These are the main entrypoints for this project, and neither party should have access to the internal memory of the other process.

2.4.2 Shared Libraries

These libraries are contained in `./shared/` and are used by both the server and client (but do not share data between the two). The libraries in this directory are implemented in Python.

2.4.3 C++ library

This Python module is implemented using C++ in `./lib/` and will be built by `pip` when installing dependencies.

2.5 Order of Operations

The operation of this system can be categorized into three primary phases, in addition to the establishment of the TCP connection.

1. (TCP Layer) TCP connection established
   Client: Connection via `socket`
   Server: Connection via `socketserver` using `server.Handler`

2. (TLS Layer) TLS/SSL handshake establishes `shared.rpi_ssl.Session`
   Client: Buffer control given to `shared.handshake_handler.client_handle_handshake`
   Server: Buffer control given to `shared.handshake_handler.server_handle_handshake`

3. (App Layer) Authenticate user to associate `database.Account` with session
   Client: Get `shared.card.Card` information from user
   Server: Verify information in `database`

4. (App Layer) Accept routine ATM commands
   Client: Get command details and send request to server
   Server: Respond to command as appropriate

3 TLS/SSL Implementation

At this level, we attempt to implement a simplified version of TLS 1.3. In this, we attempt to implement cryptographic algorithms in a manner that is usage-independent and reusable.

3.1 Symmetric Encryption

We implement only AES-256 with Cipher Block Chaining. This is contained in the C++ library. This differs from TLS 1.3 in that CBC-mode is not permitted by TLS 1.3, but is otherwise compliant.

3.2 Hash Functions

We implement SHA-1, SHA-256, and SHA-384 in `shared.rpi_hash`. However, in keeping with TLS 1.3 and best practices, SHA-1 is not considered a valid option for any purpose.

3.3 Message Authentication Codes (MACs)

We implement HMAC in `shared.rpi_hash`.

3.4 Digital Signatures

We implement only RSASSA-PSS (Signature Scheme with Appendix; Probabilistic Signature Scheme) in `shared.rsassa_pss` as specified in RFC 8017. This is a preferred digital signature scheme by TLS 1.3 and produces a non-deterministic signature that can be used to verify the original message using the RSA public key. Our implementation can apply any hash function, but we use SHA-256 and SHA-384 in keeping with TLS 1.3 and best practices. Both `rsa_pss_rsae` and `rsa_pss_pss` modes as defined by TLS 1.3 use this algorithm, **TODO: DO WE SUPPORT BOTH OR JUST ONE?**

3.5 TLS Records

Each message in a TLS/SSL session is contained within a TLS record. Functions to encode and decode these, along with a `Session` class containing negotiated TLS/SSL session information are implemented in `shared.rpi_ssl`. A `Session` will only exist after the handshake is completed, and thus the encrypted Alert and Application record implementations are included within it. The `Session` can automatically manage encryption, decryption, and verification/creation of MAC. Essentially, it provides an interface for the Application-level.

3.6 TLS/SSL Error Handling

Should a TLS/SSL-related error occur on this level, an `SSLError` will the thrown, which should be reported to the other party via a corresponding TLS Alert record. As specified by TLS 1.3, error alerts (all alerts except `close_notify` and `user_canceled`) must be considered fatal.

4 Application-Level Protocol

This is also partially described in `shared.protocol`. All messages at the application level should be in the body of an SSL/TLS application record. This means that application level messages are encrypted and authenticated between the client and server (but not necessarily the user). Note messages sent follow the format of a client request, followed by a server response. The server should never spontaneously send a message.

4.1 Application Message Format

Each application-level message starts with a single header byte describing what application message type it contains. These types are defined by the enum `shared.protocol.MsgType`. The rest of the message contains further details:

4.1.1 `ACCOUNT_AUTH` message type

This message type is used when authenticating the user. Prior to user authentication being completed, only `ACCOUNT_AUTH` and possibly `ERROR` messages should be sent. If the server responds successful, user authentication is now completed and the session is permanently associated with the provided account.

Request Format

Start	Length	Content	
0x00	0x01	`0x00`	(`ACCOUNT_AUTH` header byte)
0x01	0x0c	Card Data	(formatted with `Card::to_bytes()`)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x00`	(`ACCOUNT_AUTH` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

or, if attempts have been exceeded, the response is a fatal `ATTEMPTS_EXCEEDED` error message↓ (this can be considered unsuccessful).

4.1.2 `BALANCE` message type

This message type should only be used after the user is authenticated. It requests the current account balance of the user from the server.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x01`	(`BALANCE` header byte)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x01`	(`BALANCE` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(account balance in cents)

4.1.3 `DEPOSIT` message type

This message type should only be used after the user is authenticated. It requests that an amount be added to the user's balance. With a valid ATM client, the corresponding amount of cash will have been inserted. While `DEPOSIT` returns its success status in a similar manner to `WITHDRAW`, it will rarely if ever be unsuccessful.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x02`	(`DEPOSIT` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(deposit amount in cents)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x02`	(`DEPOSIT` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

4.1.4 `WITHDRAW` message type

This message type should only be used after the user is authenticated. It requests that an amount be deducted from the user's balance. With a valid ATM client, the corresponding amount of cash will be provided if successful.

Request Format

Start	Length	Content	
`0x00`	`0x01`	`0x03`	(`WITHDRAW` header byte)
`0x01`	`0x08`	big-endian unsigned integer	(withdraw amount in cents)

Response Format

Start	Length	Content	
`0x00`	`0x01`	`0x03`	(`WITHDRAW` header byte)
`0x01`	`0x01`	`0x00` or `0x01`	(unsuccessful or successful, respectively)

4.1.5 `ERROR` message type

This message type represents a serious error with a request at the application level. It should only be sent by the server as a response to a client request. Error codes are defined by the enum `AppError` in `shared.protocol`. See the next section↓ for error type details.

Response Format

Start	Length	Content	
0x00	0x01	`0xFF`	(`ERROR`)
0x01	0x01	error code	(from `AppError` header byte)

4.2 Application Error Codes

Application error codes may be sent by the server in an `ERROR`-type application message in response to client requests. They should not occur or be sent in any other situation. If a valid response exists for the request's own message type (for example insufficient funds for `WITHDRAW`), then that will be used instead.

4.2.1 `INVALID_STAGE` (`0x00`)

This error code may be sent when a application message is sent at an improper time. For example, if `ACCOUNT_AUTH` messages are sent after the user is authenticated, or if any `BALANCE`, `DEPOSIT`, or `WITHDRAW` messages are sent before a user is authenticated. Additionally, this error code may be sent if a nonexistant message type is received.

4.2.2 `BAD_MESSAGE` (0x01)

This error code may be sent when the content following a valid error code appears to be improperly formatted. Note that as an application-level error, this will not occur if SSL/TLS issues arise.

4.2.3 `ATTEMPTS_EXCEEDED` (0x02)

This error will occur if the session has exceeded its permitted failed user authentication attempts or if the specified card number has recieved too many recent failed login attempts. As such, it can only occur in response to an `ACCOUNT_AUTH` request. This error is always fatal. For more details, see below↓

4.3 Credit/Debit Cards

Implementation for card formats and related details is in `shared.card`. The `Card` class represents the fixed-length fields of a standard credit or debit card:

Card Number	A 16-character numerical string which passes the Luhn test.
CVC	An integer in the range `[000,999]`.
Month	An integer in the range `[1,12]`.
Year	An integer in the range `[2000,3023]`.
PIN	An integer in the range `[0000,9999]`.

These can be serialized to and from 12 bytes for transmission.
Additionally, a `Card::generateRandom()` method is provided which will generate a random valid card, optionally with some set fields.

4.4 Database

Implementation for the `database` relies heavily on the above `shared.card.Card` implementation. Primarily, the database is made up of `database.Account` objects, which store information associated with the account, including the card, balance, and accountholder's name. Note that the name is not used for verification.
The database implemented here is intended to represent an abstraction of a real database, but does not fully implement the features of a typical database. For example, restarting the server will cause a database reset, as all data is stored exclusively in memory.

The database exposes the `get_account()` function to the server (and only the server), and the server is permitted to modify a returned `Account`'s balance. The client and user may not access the database except through the server.

4.4.1 Login Attempt Limit

The database implements a limit on recent attempts to login with any card number. If 5 or more failed attempts have occurred in the past 30 minutes (including any attempts that failed because of this), the database will throw an `AttemptsExceededError` that should be caught by the caller and transmitted to the client.
Note that to prevent this being used as a method to find other users' card numbers, login attempts for unused card numbers will have similar behavior in this regard as login attempts providing invalid verification details. So just as five attempts with a correct card number but incorrect PIN will cause further attempts with this card number to error, so too will five attempts with an incorrect card number. This error is fatal.

4.4.2 Generating Accounts

A number of accounts are generated when the database is started. Many of these randomize their card fields. See above↑ for the known account details.

5 Provided Server Instance

For the convenience of the testing and accessibility, a Ubuntu Server Node running 'Server.py is provided.

URL	**bank.b-rad.solutions**
Port	**8000**

The servers Firewall Will block all traffic accept TCP traffic on application port.

5.1 Configuring Server Address

In 'client.py' change the first non-import line 'conn = ... ' to the desired URL or IP address as the first arg. Ensure that 'shared/port.py' is correct

```
conn = socket.create_connection(("localhost", PORT))
```

6 File Structure Overview

- lib
- my_secrets
- shared
- .gitignore
- README.md
- client.py
- database.py
- documentation.pdf
- documentation.tex
- generate_keys.py
- requirements.txt
- server.py

6.1 Root Directory

6.1.1 client.py

Main Run time file for the program on client side.

1. **fetch_record()** CLI interface for the client to input card details
2. **input_card_legacy()** CLI interface for the client to input card details
3. **input_card()** CLI interface for the client to input card details
4. **select_mode_legacy()** CLI interface for the client to select mode of operation
5. **select_mode()** CLI interface for the client to select mode of operation
6. **request_balance()** This will retrieve the user's account balance from the bank's servers database and return it to the user.
7. **request_deposit()** This will send a request to the servers database for a deposit
8. **request_withdraw()** This will send a request to the servers database for a withdraw

   or invalid account information. It may also update the transaction history for the user's account to reflect the withdrawal.

6.1.2 database.py

Family of functions to interact with the data of the bank on the server side.

1. **balance()** This will retrieve the user's account balance and return it to the user.
2. **deposit()** This will add the deposit amount to the user's account balance and update the account balance accordingly. The function would then return a message indicating the successful deposit and the updated account balance, or an error message if the deposit could not be completed.
3. **withdraw()** This will check if the user has sufficient funds in their account to cover the withdrawal amount. If the user has enough funds, the function would subtract the withdrawal amount from the user's account balance and update the account balance accordingly. The function would then return a message indicating the successful withdrawal and the updated account balance, or an error message if the withdrawal could not be completed due to insufficient funds.

4. **_add_account()** the function is called with the account number, CCV number, and Exp. Date. If the operation is successful, the function returns a success flag and a message indicating that the account was added to the database. If there is an error, the function returns a failure flag and an appropriate error message

5. **get_account()** Retrieves info on a user account.

6.1.3 documentation.pdf

This PDF!

6.1.4 generate_keys.py

SSH (Secure Shell) key generation involves creating a public and private key pair that can be used for authentication and encryption in secure communication between two devices.

The key generation process usually involves using a program such as ssh-keygen to create a unique key pair. The public key can be shared with anyone, while the private key is kept secret and stored on the device that will be used to establish the secure connection.

The key pair is created using a mathematical algorithm, typically either RSA or DSA, which generates a unique set of numbers that are used to encrypt and decrypt messages.

6.1.5 requirements.txt

The requirements.txt file in Python is a text file that lists all the Python packages and their versions that are required to run a Python project or application. The file contains a list of dependencies that need to be installed before running the application or project.

The format of a requirements.txt file is simple: each line contains the name of a package and its version number, separated by an == sign. To install the dependencies listed in a requirements.txt file, one can use the command pip install -r requirements.txt

6.1.6 server.py

Main Run time file for the program the Server side.

1. **fetch_record()**
2. **handle_handshake()**
3. **handle_account_auth()**
4. **handle_routine()**
5. **setup()**

6. **message_handler()**
7. **handle()**
8. **finish()**

6.2 lib Directory

PyBind is a C++ library that provides a simple and lightweight way to expose C++ functions and classes to Python. It allows you to write Python modules in C++ by generating Python bindings from C++ code. PyBind is designed to be easy to use and offers a range of features to make it flexible and efficient.

With PyBind, you can easily expose C++ classes, functions, and objects to Python, and you can call Python code from C++ as well. PyBind offers a number of features that make it easy to work with both languages, such as automatic type conversion and the ability to use Python classes and objects in C++ code.

6.2.1 aes.h

AES, or Advanced Encryption Standard, is a widely used symmetric encryption algorithm. It was established by the National Institute of Standards and Technology (NIST) in 2001 as a replacement for the older Data Encryption Standard (DES).

AES uses a block cipher algorithm, which means that it divides the data to be encrypted into fixed-length blocks and applies the encryption algorithm to each block individually. The standard block size for AES is 128 bits, but it can also be used with block sizes of 192 and 256 bits.

AES is a symmetric encryption algorithm, which means that the same key is used for both encryption and decryption. The key length can be 128, 192, or 256 bits, and the longer the key, the stronger the encryption.

The AES encryption algorithm consists of several rounds of substitution and permutation operations. Each round involves a substitution step, where each byte of the block is replaced with a different byte from a substitution table, and a permutation step, where the bytes are rearranged according to a fixed pattern.

Overall, AES is considered to be a highly secure encryption algorithm and is widely used in various applications, including electronic banking, digital signatures, and secure communications.

6.2.2 aes_constants.h

The AES constants refer to the pre-computed values used in the AES algorithm. These values are generated from the irreducible polynomial $x\hat{8} + x\hat{4} + x\hat{3} + x + 1$, which is used in the finite field arithmetic operations of AES.

The AES constants consist of four lookup tables, each with 256 32-bit words:

The S-box: This is a substitution table used in the byte substitution step of the AES algorithm. It maps each 8-bit input to a corresponding 8-bit output.

The inverse S-box: This is the inverse of the S-box and is used in the decryption process of the AES algorithm.

The Rcon table: This is a table of round constants used in the key schedule of the AES algorithm. There are 10, 12, or 14 round constants depending on the key size used.

The Galois table: This is a lookup table used in the mixColumns step of the AES algorithm. It is used to multiply columns of bytes in the finite field arithmetic.

These AES constants are precomputed and stored in memory to speed up the encryption and decryption processes. They are an essential part of the AES algorithm and are used in all implementations of the algorithm.

6.2.3 aes_test.cpp

THis File shows AES works out side of the banking application

6.2.4 cpplib.cpp

Creates Links between Cpp and Python

6.2.5 setup.py

Creates Links between Cpp and Python

6.3 my_secrets Directory

A directory called m̈y_secretsẗhat stores a public key for a client and a server is a folder on a computer system that contains two files, one for the client's public key and the other for the server's public key. The directory may be protected by appropriate access controls, to ensure that only authorized users or processes can access the files within it. The files containing the public keys are used to establish secure communication between the client and server, through methods such as SSL/TLS encryption, SSH connections, or other secure protocols. The directory must be located in a specific location on the file system.

6.4 shared Directory

A directory called "shared" that stores helper functions for both a client and a server is a folder on a computer system that contains files with code that can be used by both the client and the server. These files might include common utility functions, configuration files, or other shared resources that are used by both the client and server applications. The directory may be organized in a specific way, such as by function or module, to make it easy for developers to find the resources they need. The shared directory must be located in a specific location on the file system, such as in the root directory of the application, or in a dedicated directory for shared resources. The directory may also be subject to appropriate access controls, to ensure that only authorized users or processes can access the files within it. The purpose of the shared directory is to promote code reuse and to make it easier to maintain both the client and server applications by avoiding duplicating code that is common to both.

6.4.1 card.py:

This Python file contains a class or function that generates and handles digital Creit cards.

1. **valid_card_num()** Check to make sure a user entered card number meets all the requirements

    (a) card is decimal or number

    (b) card lengh is 16

    (c) $0/ \leq CVV \leq 1000$

    (d) $1 \leq Month \leq 12$

    (e) $2023 \leq year$

6.4.2 card_test.py:

This is a Python test file that tests the functionality and features of the name-card.py file. It includes various test cases that ensure the name card is generated and handled correctly.

6.4.3 handshake_handler.py:

This Python file contains a class or function that handles handshakes between two entities or systems. Handshakes are a series of communications that establish a connection or exchange information.

1. **gen_hash_input()**
2. **from_shared_secret()**
3. **server_handle_handshake()**
4. **client_handle_handshake()**

6.4.4 keygen.py:

This Python file contains a function or class that generates cryptographic keys used to encrypt and decrypt data, which is essential for secure communication and data storage. Diffie-Hellman is a cryptographic algorithm used to establish a shared secret between two parties over an insecure communication channel. It was invented by Whitfield Diffie and Martin Hellman in 1976 and is widely used today in various protocols such as TLS, SSH, and VPNs. The main idea behind the Diffie-Hellman algorithm is to use the properties of modular arithmetic to enable two parties to agree on a shared secret without actually exchanging it directly over the insecure channel. The algorithm involves the following steps:

1. First, the two parties, Alice and Bob, agree on a large prime number and a primitive root modulo that prime.

2. These values are publicly known and can be shared over the insecure channel.

3. Alice chooses a secret number, a, and computes A = g^a mod p, where g is the primitive root and p is the prime number.

4. She sends A to Bob over the insecure channel.

5. Bob chooses a secret number, b, and computes B = $g\hat{b}$ mod p. He sends B to Alice over the insecure channel.

6. Alice computes the shared secret as S = $B\hat{a}$ mod p.

7. Bob computes the shared secret as S = $A\hat{b}$ mod p.

8. The shared secret, S, can now be used as a symmetric key for encryption and decryption of messages between Alice and Bob.

One of the main advantages of the Diffie-Hellman algorithm is that it provides perfect forward secrecy, which means that even if an attacker were to obtain the private keys of Alice or Bob at some later time, they would not be able to decrypt past communications because the shared secret was never transmitted over the insecure channel

6.4.5 paillier.py:

This Python file contains a class or function that implements the Paillier cryptosystem, a public-key cryptosystem used for secure data encryption. Paillier is a public-key cryptosystem invented by Pascal Paillier in 1999. It is based on the computational difficulty of the decisional composite residuosity assumption (DCRA) problem, which is closely related to the RSA problem. The Paillier cryptosystem is considered a partially homomorphic encryption scheme, which means that it supports both additive and multiplicative homomorphic operations on ciphertexts. The Paillier cryptosystem uses two large prime numbers to generate a public key and a private key. The encryption process involves raising the plaintext to a randomly generated power modulo the product of two prime numbers, and then multiplying the result by another randomly generated value raised to the product of the prime numbers. The decryption process involves raising the ciphertext to a power modulo the product of the prime numbers and then using a modular inverse to recover the original plaintext. One of the unique features of the Paillier cryptosystem is its homomorphic properties, which allow for computations to be performed on ciphertexts without decrypting them first. Specifically, the system supports both additive and multiplicative homomorphic operations, meaning that ciphertexts can be added together or multiplied together and the resulting ciphertext will decrypt to the sum or product of the original plaintexts.

1. **genkeys()**The Paillier cryptosystem uses a public key and a private key, which are generated as follows:

   Public Key Generation:

   (a) Choose two large prime numbers p and q, such that p and q are of equal length in bits and are kept secret. Compute the product of the two primes, n = p*q, which is used as the modulus for both the public and private keys.
   (b) Compute the Carmichael's totient function of n, $\lambda(n) = lcm(p-1, q-1)$, which is used to compute the private key.
   (c) Choose a random integer g that is relatively prime to n$\hat{2}$. This is the generator of the cyclic group of integers modulo n.
   (d) Compute the public key (n, g), which is made public.

   Private Key Generation:

   (a) Choose a random integer L, such that L is relatively prime to $\lambda(n)$.
   (b) Compute the private key $\mu = L^{(-1)} \text{mod} \lambda(n)$, where $L^{(-1)}$ is the modular multiplicative inverse of L modulo $\lambda(n)$.

   Once the public and private keys are generated, they can be used for encryption and decryption operations. It is important to note that the security of the Paillier cryptosystem relies on the fact that it is difficult to compute the discrete logarithm of g modulo n$\hat{2}$, and the decisional composite residuosity assumption is true for n.

2. **encrypt()**
3. **decrypt()**
4. **summation()** Adds all the chpiertext in list together

6.4.6 paillierTest.py:

This is a Python test file that tests the functionality and features of the paillier.py file. It includes various test cases that ensure the Paillier cryptosystem is implemented correctly.

6.4.7 port.py:

This Python file contains a class that manages network ports, which are used to establish connections between two entities or systems over a network.

6.4.8 protocol.py:

This Python file contains a class or function that implements a communication protocol, a set of rules and standards that govern how data is exchanged between two entities or systems.

6.4.9 rpi_hash.py:

This Python file contains a class or function that implements a hashing algorithm, used to convert data into a fixed-length string of characters.

1. **SHA1():** SHA-1 (Secure Hash Algorithm 1) is a cryptographic hash function that was developed by the United States National Security Agency (NSA) and published as a federal standard in 1995. SHA-1 generates a fixed-sized output of 160 bits, which is typically represented as a 40-digit hexadecimal number. SHA-1 takes an input message of any length and produces a fixed-length output that is unique to that input. This makes SHA-1 useful for verifying the integrity of data, as even a small change in the input message will result in a completely different hash value. However, SHA-1 has been found to be vulnerable to collision attacks, where two different input messages can produce the same hash value. As a result, it is no longer considered to be a secure cryptographic hash function and has been deprecated in favor of newer, stronger algorithms such as SHA-256 and SHA-3. Despite its weaknesses, SHA-1 is still used in some legacy systems and protocols. However, it is recommended that any new systems and applications use stronger and more secure hash functions.

2. **SHA256():** SHA-256 (Secure Hash Algorithm 256) is a cryptographic hash function that is widely used for data integrity and authentication purposes. It is part of the SHA-2 family of hash functions, which were designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. Like other cryptographic hash functions, SHA-256 takes an input message of any length and produces a fixed-length output of 256 bits. The output is typically represented as a 64-digit hexadecimal number. SHA-256 is a stronger hash function than its predecessor, SHA-1, and is less vulnerable to collision attacks. It is also faster and more efficient than some other hash functions, such as SHA-512 and Whirlpool. SHA-256 is widely used in many applications, including digital signatures, password storage, and blockchain technology. It is also used as a component in other cryptographic protocols, such as Transport Layer Security (TLS) and Secure Sockets Layer (SSL). Overall, SHA-256 is considered to be a secure and reliable cryptographic hash function for a wide range of applications.

3. **SHA384():** SHA-384 is a cryptographic hash function that belongs to the SHA-2 family of hash functions, which were designed by the National Security Agency (NSA) and published by the National Institute of Standards and Technology (NIST) in 2001. SHA-384 is similar to SHA-256, but produces a longer output of 384 bits. It uses the same basic algorithm as SHA-256, but with more rounds of processing and a larger initial hash value. Like other hash functions, SHA-384 takes an input message of any length and produces a fixed-length output. The output is typically represented as a 96-digit hexadecimal number. SHA-384 provides a higher level of security than SHA-256, as it has a larger output size and uses more rounds of processing. It is particularly useful in applications where stronger security is required, such as in digital signatures and key derivation. However, SHA-384 is also slower and less efficient than SHA-256, due to its longer output size and more complex processing. As a result, it may not be suitable for applications where speed and efficiency are important. Overall, SHA-384 is a secure and reliable cryptographic hash function that can provide a higher level of security than SHA-256 in certain applications. However, it should be used carefully, taking into consideration its slower speed and larger output size.

4. **HMAC():** HMAC (short for ¨keyed-hash message authentication code¨) is a type of message authentication code (MAC) that involves a cryptographic hash function and a secret key. HMAC is designed to verify the authenticity and integrity of a message, ensuring that it has not been altered or tampered with during transmission. HMAC works by taking a message and a secret key as input and applying a cryptographic hash function (such as SHA-256 or SHA-512) to them. The resulting hash value is then combined with the secret key again and hashed again to produce the final HMAC value. HMAC is widely used in various security protocols and applications, including secure email, secure remote access, and secure web browsing. It provides a simple and efficient way to verify the authenticity and integrity of messages without requiring a secure channel for communication.

6.4.10 rpi_ssl.py:

This Python file contains a class or function that implements the SSL (Secure Sockets Layer) protocol used to establish secure connections between two entities or systems over a network.

6.4.11 rpi_ssl_test.py:

This is a Python test file that tests the functionality and features of the rpi_ssl.py file. It includes various test cases that ensure the SSL protocol is implemented correctly.

6.4.12 rsassa_pss.py:

This Python file contains a class or function that implements the RSASSA-PSS digital signature algorithm, used to create and verify digital signatures.

1. **MGF1** MGF1 (Message Digest with a Fixed-length output) is a hash function that produces a fixed-length hash value. It was originally proposed in 1999 by Ronald Rivest as an alternative to the SHA-1 hash function, which was later found to be vulnerable to collision attacks.

   The MGF1 hash function is based on a pseudorandom function (PRF) that uses a secret key and a seed value to produce a random output. The PRF is then used to produce a hash value by iteratively applying the function to the input data.

   The MGF1 algorithm has several advantages over other hash functions, including its simplicity, speed, and resistance to collision attacks. However, it is not as widely used as some other hash functions, such as SHA-2 and SHA-3.

   MGF1 is often used in conjunction with other cryptographic algorithms, such as RSAOAEP (Optimal Asymmetric Encryption Padding), which uses MGF1 to generate a random mask for data encryption. It is also used in the PKCS#1 standard for RSA cryptography and in some digital signature schemes.

6.4.13 rsassa_pss_test.py:

This is a Python test file that tests the functionality and features of the rsassa_pss.py file. It includes various test cases that ensure the RSASSA-PSS algorithm is implemented correctly.

6.4.14 tls_handshake.py:

This Python file contains a class or function that implements the TLS (Transport Layer Security) protocol handshake process, used to establish secure connections between two entities or systems over a network.

6.4.15 tls_handshake_test.py:

This is a Python test file that tests the functionality and features of the tls_handshake.py file. It includes various test cases that ensure the TLS handshake process is implemented correctly.

6.4.16 tls_handshake_test2.py:

This is another Python test file that tests the functionality and features of the tls_handshake.py file, with additional test cases not included in tls_handshake_test.py.

7 The Transport Layer Security (TLS) Protocol Version 1.3

This project is adapted from TLS 1.3 spec, RFC 8446. This section will overview the protocol and any changes to implmentation makes to it.

7.1 Protocol Overview

The cryptographic parameters used by the secure channel are produced by the TLS handshake protocol. This sub-protocol of TLS is used by the client and server when first communicating with each other. The handshake protocol allows peers to negotiate a protocol version, select cryptographic algorithms, authenticate each other, and establish shared secret keying material. Once the handshake is complete, the peers use the established keys to protect the application-layer traffic.

A failure of the handshake or other protocol error triggers the termination of the connection

7.1.1 Message Flow for Full TLS Handshake

The handshake can be thought of as having three phases (indicated in the diagram below)

```
       Client                                           Server

Key  ^ ClientHello
Exch | + key_share*
     | + signature_algorithms*
     | + psk_key_exchange_modes*
     v + pre_shared_key*       -------->
                                                  ServerHello  ^ Key
                                                 + key_share*  | Exch
                                            + pre_shared_key*  v
                                        {EncryptedExtensions}  ^  Server
                                        {CertificateRequest*}  v  Params
                                               {Certificate*}  ^
                                         {CertificateVerify*}  | Auth
                                                   {Finished}  v
                               <--------  [Application Data*]
     ^ {Certificate*}
Auth | {CertificateVerify*}
     v {Finished}              -------->
       [Application Data]      <------->  [Application Data]

              +  Indicates noteworthy extensions sent in the
                 previously noted message.

              *  Indicates optional or situation-dependent
                 messages/extensions that are not always sent.

              {} Indicates messages protected using keys
                 derived from a [sender]_handshake_traffic_secret.

              [] Indicates messages protected using keys
                 derived from [sender]_application_traffic_secret_N.
```

1. Key Exchange: Establish shared keying material and select the cryptographic parameters. Everything after this phase is encrypted.
2. Server Parameters: Establish other handshake parameters (whether the client is authenticated, application-layer protocol support, etc.).
3. Authentication: Authenticate the server (and, optionally, the client) and provide key confirmation and handshake integrity.

The server then sends two messages to establish the Server Parameters:

1. **EncryptedExtensions:** responses to ClientHello extensions that are not required to determine the cryptographic parameters, other than those that are specific to individual certificates.
2. **CertificateRequest:** if certificate-based client authentication is desired, the desired parameters for that certificate. These Params will be deatailed in a later section

Finally, the client and server exchange Authentication messages. TLS uses the same set of messages every time that certificate-based authentication is needed. Specifically:

1. **Certificate:** The certificate of the endpoint and any per-certificate extensions. This message is omitted by the server if not authenticating with a certificate and by the client if the server did not send CertificateRequest (thus indicating that the client should not authenticate with a certificate).

2. **CertificateVerify:** A signature over the entire handshake using the private key corresponding to the public key in the Certificate message. This message is omitted if the endpoint is not authenticating via a certificate.

3. **Finished:** A MAC (Message Authentication Code) over the entire handshake. This message provides key confirmation, binds the endpoint's identity to the exchanged keys, and in PSK mode also authenticates the handshake.

Upon receiving the server's messages, the client responds with its Authentication messages, namely Certificate and CertificateVerify (if requested), and Finished.

At this point, the handshake is complete, and the client and server derive the keying material required by the record layer to exchange application-layer data protected through authenticated encryption. Application Data MUST NOT be sent prior to sending the Finished message. Note that while the server may send Application Data prior to receiving the client's Authentication messages, any data sent at that point is, of course, being sent to an unauthenticated peer.

7.2 Handshake Structures

Messages are stored as a struct that is flattened and reassembled on the other side of the connection. Structs are represented as Python Classes.

7.2.1 struct ClientHello

When a client first connects to a server, it is REQUIRED to send the ClientHello as its first TLS message. The client will also send a ClientHello when the server has responded to its ClientHello with a HelloRetryRequest.

Because TLS 1.3 forbids renegotiation, if a server has negotiated TLS 1.3 and receives a ClientHello at any other time, it MUST terminate the connection with an unexpected_message alert.

If a server established a TLS connection with a previous version of TLS and receives a TLS 1.3 ClientHello in a renegotiation, it MUST retain the previous protocol version. In particular, it MUST NOT negotiate TLS 1.3.

Structure of this message:

```
Structure of this message:

   uint16 ProtocolVersion;
   opaque Random[32];

   uint8 CipherSuite[2];    /* Cryptographic suite selector */

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id<0..32>;
       CipherSuite cipher_suites<2..2^16-2>;
       opaque legacy_compression_methods<1..2^8-1>;
       Extension extensions<8..2^16-1>;
   } ClientHello;
```

After sending the ClientHello message, the client waits for a ServerHello or HelloRetryRequest message. If early data is in use, the client may transmit early Application Data while waiting for the next handshake message.

7.2.2 struct ServerHello

The server will send this message in response to a ClientHello message to proceed with the handshake if it is able to negotiate an acceptable set of handshake parameters based on the ClientHello.

```
Structure of this message:

   struct {
       ProtocolVersion legacy_version = 0x0303;    /* TLS v1.2 */
       Random random;
       opaque legacy_session_id_echo<0..32>;
       CipherSuite cipher_suite;
       uint8 legacy_compression_method = 0;
       Extension extensions<6..2^16-1>;
   } ServerHello;
```

7.2.3 struct Hello Retry Request

The server will send this message in response to a ClientHello message if it is able to find an acceptable set of parameters but the ClientHello does not contain sufficient information to proceed with the handshake. The HelloRetryRequest has the same format as a ServerHello message, so it is hard coded in the implementation.

7.2.4 struct Extensions

A number of TLS messages contain tag-length-value encoded extensions structures.

```
struct {
    ExtensionType extension_type;
    opaque extension_data<0..2^16-1>;
} Extension;

enum {
    server_name(0),
    max_fragment_length(1),
    status_request(5),
    supported_groups(10),
    signature_algorithms(13),
    use_srtp(14),
    heartbeat(15),
    application_layer_protocol_negotiation(16),
    signed_certificate_timestamp(18),
    client_certificate_type(19),
    server_certificate_type(20),
    padding(21),
    pre_shared_key(41),
    early_data(42),
    supported_versions(43),
    cookie(44),
    psk_key_exchange_modes(45),
    certificate_authorities(47),
    oid_filters(48),
    post_handshake_auth(49),
    signature_algorithms_cert(50),
    key_share(51),
    (65535)
} ExtensionType;
```

Here:

1. "extension_type" identifies the particular extension type.
2. "extension_data" contains information specific to the particular extension type.

7.2.5 struct Finshed

The Finished message is the final message in the Authentication Block. It is essential for providing authentication of the handshake and of the computed keys.

Recipients of Finished messages MUST verify that the contents are correct and if incorrect MUST terminate the connection with a decrypt_error alert.

Once a side has sent its Finished message and has received and validated the Finished message from its peer, it may begin to send and receive Application Data over the connection.

```
finished_key =
    HKDF-Expand-Label(BaseKey, "finished", "", Hash.length)

Structure of this message:

   struct {
       opaque verify_data[Hash.length];
   } Finished;

The verify_data value is computed as follows:

   verify_data =
       HMAC(finished_key,
            Transcript-Hash(Handshake Context,
                            Certificate*, CertificateVerify*))

   * Only included if present.
```

7.3 Transcript Hash

Many of the cryptographic computations in TLS make use of a transcript hash. This value is computed by hashing the concatenation of each included handshake message, including the handshake message header carrying the handshake message type and length fields, but not including record layer headers. I.e., This is done with SHA256//

Transcript-Hash(M1, M2, ... Mn) = Hash(M1 || M2 || ... || Mn)

7.4 Signature Algorithms: RSASSA-PSS

RSASSA-PSS (RSA Signature Scheme with Appendix - Probabilistic Signature Scheme) is a digital signature scheme that combines the RSA algorithm with probabilistic methods for added security. It was introduced in 2002 as an enhancement to the older RSA-PKCS#1 v1.5 signature scheme.

In RSASSA-PSS, a hash function is first applied to the message to be signed to produce a fixed-length digest. The digest is then padded with random bits to a predetermined length, and the resulting padded digest is used as the message input to the RSA signing algorithm. The RSA algorithm then produces a signature that is appended to the message. The probabilistic padding ensures that the same message will produce different signatures each time it is signed, making the scheme resistant to certain types of attacks.

The scheme also includes a salt value that is used to further randomize the padding, making it even more difficult for attackers to forge a signature. The salt value is included in the signature and can be verified by the recipient to ensure that the signature is authentic.

RSASSA-PSS is considered to be more secure than the older RSA-PKCS#1 v1.5 signature scheme, as it provides better protection against certain types of attacks such as chosen-message attacks and key-recovery attacks. It is widely used in modern cryptography applications, including SSL/TLS, SSH, and digital certificates.

7.4.1 Signature Generation Operation

```
RSASSA-PSS-SIGN (K, M)

Input:

   K        signer's RSA private key
   M        message to be signed, an octet string

Output:

   S        signature, an octet string of length k, where k is the
            length in octets of the RSA modulus n

Errors:  "message too long;" "encoding error"
```

7.4.2 Signature Verification Operation

```
RSASSA-PSS-VERIFY ((n, e), M, S)

Input:

   (n, e)  signer's RSA public key
   M       message whose signature is to be verified, an octet string
   S       signature to be verified, an octet string of length k,
           where k is the length in octets of the RSA modulus n

Output:  "valid signature" or "invalid signature"
```

7.5 Certificates

7.5.1 Receiving a Certificate Message

If the server supplies an empty Certificate message, the client MUST abort the handshake with a decode_error alert.

All endpoints are transitioned to SHA-256 to maintain interoperability with implementations currently in the process of phasing out SHA-1 support.

7.5.2 struct CertificateRequest

A server which is authenticating with a certificate will request a certificate from the client. This message, if sent, MUST follow EncryptedExtensions.

```
struct {
    opaque certificate_request_context<0..2^8-1>;
    Extension extensions<2..2^16-1>;
} CertificateRequest;
```

7.5.3 struct CertificateVerify

This message is used to provide explicit proof that an endpoint possesses the private key corresponding to its certificate. The CertificateVerify message also provides integrity for the handshake up to this point. Servers MUST send this message when authenticating via a certificate. Clients MUST send this message whenever authenticating via a certificate. When sent, this message MUST appear immediately after the Certificate message and immediately prior to the Finished message.

```
Structure of this message:

   struct {
       SignatureScheme algorithm;
       opaque signature<0..2^16-1>;
   } CertificateVerify;
```

The algorithm field specifies the signature algorithm used. The signature is a digital signature using that algorithm.