**SSL Key Aspects:** Authentication (via certificates), Privacy (via encryption), Integrity (via hash functions)

# **Authentication** - Asymmetric Key Ciphers

The possible asymmetric key ciphers the ATM and Bank can use include: RSA, ElGamal. We use them for digitally signing private symmetric keys and MAC keys.

\*mention security against ciphertext only model and ind-cpa model\*

## Rivest-Shamir-Adleman

On top of the *textbook* Rivest-Shamir-Adleman (RSA) cryptosystem, we augmented it by introducing random padding. Specifically, we implemented optimal asymmetric encryption padding (OAEP), as detailed in version 2.2 of Public-Key Cryptography Standards (PKCS#1 v2.2). This scheme allows us to

1) encode plaintext messages prior to encrypting them with RSA

2) decode ciphertext messages after decrypting them with RSA

To encode messages, we first combine them with some hash output, the number 1 represented as a byte, and a padding string. All these things get concatenated into a new string (DB).

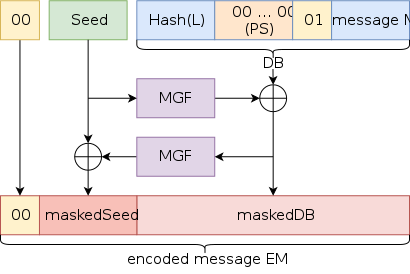
The hash output is, more precisely, the output of a chosen hash function (Hash) after passing it a string called label (L). The label is optional, so if one is not provided it will default to an empty string. In our implementation, we use this default.

This padding string (PS) is a string of null-bytes, with an exact length equal to k – mLen – (2\*hLen) – 2. Here, k equals the byte length of the modulus being used in the accompanying RSA algorithm, mLen equals the byte length of the message to encrypt, and hLen equals the byte length of any given output from Hash.

Next, we need to define a mask generation function. A mask generation function (MGF) is essentially a hash function, but it produces output of varying lengths. One generates a random value as the seed for the MGF, ensuring it has the same length as hLen. In our implementation, we utilize MGF1 which is a mask generation function defined in PKCS#1.

Next, DB is XOR’d with the output of the MGF. Note we are allowed to XOR these two because just like DB, the output also has k – hLen – 1 bytes. This acts as our way of “masking” DB. This result is now our new temporary seed for MGF. We use it to generate a string of length hLen. Now we can also mask the original random value seed by XOR’ing it with this newly generated string.

Finally, the encoding of the original message is the combination of the seed mask, DB mask, and an extra null-byte.



To decode messages, we basically reverse all the steps of the encoding algorithm. We want to ‘undo’ the masking that occurred by XOR’ing an already masked string and the mask output that was used to do the encoding. For example, the masking of DB = MGF(seed, k – hLen - 1) ^ DB. Thus, the demasking of this masking…

= masking ^ MGF(seed, k – hLen - 1)

= (MGF(seed, k – hLen - 1) ^ DB) ^ MGF(seed, k – hLen - 1) by substitution

= (MGF(seed, k – hLen - 1) ^ MGF(seed, k – hLen - 1)) ^ DB by associativity

= 0 ^ DB = DB. by identity

First, we compute the hash of the given label, which is an empty string. Then from the encoded message, extract the first byte as Y, the next hLen bytes as the masked seed, and the remaining k – hLen – 1 bytes as the masked DB. Compute MGF with masked DB as the input and a desired length of hLen. This is the mask that was used to create the masked seed value. Hence, we can determine the original seed by XOR’ing the masked seed and this newly found mask. Similarly, compute the DB mask by passing this original seed and a desired length of k – hLen - 1 into the MGF. Then we can also determine the original DB by XOR’ing the masked DB and this newly found mask.

Now that we have the original DB, we can extract the original message from the last hLen bytes.

The OAEP scheme uses random oracles (RO) in the form of the mask generating and hash functions. We inject randomness via these random oracles. Such randomness turns the subsequent RSA encryption probabilistic. Hence, the implementation is **semantically secure** against chosen plaintext attacks (**IND-CPA**).

By encoding a message before feeding it into RSA’s trapdoor permutation function, the overall encryption process becomes **plaintext aware**. To produce any ciphertext, one must know its associated plaintext.

Because our implementation is both semantically secure and plaintext aware, it is also secure against chosen ciphertext attacks (**IND-CCA1**). An attacker with access to encryption and decryption oracles cannot gain a meaningful advantage in deciding which plaintext corresponds to a selected (by a challenger) ciphertext. Note that in this context, the attacker loses their decryption oracle once the challenge is issued.

Using OAEP with RSA is also secure against adaptive chosen ciphertext attacks (**IND-CCA2**). This is essentially the same as IND-CCA1 except now the attacker can *still* use their decryption oracle even after the challenge ciphertext was declared, as long as they do not submit the challenge ciphertext itself.

# **Privacy** - Symmetric Key Ciphers

The possible symmetric key ciphers the ATM and Bank can use include: AES. We use them for encrypting and decrypting messages sent in a secured channel.

# Homomorphic Ciphers

The possible homomorphic ciphers the ATM and Bank can use include: .

# ATM

This code defines a class called ATM. This class represents an Automated Teller Machine and allows a user to connect to a bank server via a socket and perform transactions on their bank account.

The code imports various libraries such as json, hash, socket, ast, secrets, and some classes from other files such as rsa, elgamal, and aes.

The constructor \_\_init\_\_ initializes various instance variables such as aeskey, mackey, p, prefs, counter, id\_num, and a socket object s. The s.connect(('127.0.0.1', 5432)) statement establishes a connection to a server with IP address 127.0.0.1 and port number 5432.

The countercheck function checks if a message is tampered with or if the counter is less than or equal to the current counter.

The post\_handshake function handles the exchange of messages between the client and server to establish a secure connection. It also authenticates the user by asking for their username and password, encrypting and hashing them, and sending the result to the server.

The ATM class provides a command-line interface to the user, where they can perform various transactions such as deposit, withdraw, and check balance.

The code uses various encryption and hashing techniques such as AES encryption, HMAC, and SHA1 hashing to ensure the security of the communication between the client and server.

# Bank

This code is an implementation of a banking server that listens for requests from ATM clients. The Bank class is the main component of the server, and it contains methods for handling client requests such as withdrawal, deposit, and balance checks.

Summary of the code:

The Bank class initializes by reading two JSON files that contain user data: usertohashpass.txt and usertomoney.txt. It also sets up a list of available public key encryption methods (RSA and ElGamal), initializes some cryptographic variables, and creates a TCP socket to listen for incoming connections from ATM clients.

The countercheck method checks if the message received from the client has a counter value greater than the server's counter value. This is used to prevent replay attacks, where an attacker captures and resends a previously sent message.

The withdraw, deposit, and check methods handle client requests for withdrawing money, depositing money, and checking account balances respectively. Each method sends a response back to the client after encrypting the response and appending a HMAC (hash-based message authentication code) to ensure message integrity.

The post-handshake method is called after a client connects to the server and completes a handshake. This method exchanges a counter value with the client to ensure that both sides are synchronized. It also sets a flag to indicate that the client is now logged in.

The main loop of the Bank class listens for incoming commands from the client and dispatches them to the appropriate method. Each command is decrypted, verified for message integrity using the HMAC, and checked for replay attacks using the counter value. If any of these checks fail, an exception is raised.

# Conclusion

The code consists of two parts, one for simulating an ATM and one for managing bank accounts. The ATM simulation allows the user to withdraw and deposit money, and also displays the account balance. It has built-in error handling for invalid inputs and insufficient funds. The bank account management code allows the user to create new accounts, deposit and withdraw money, and display the account information. It also has error handling for invalid inputs and negative balances.

Overall, the code provides basic functionality for a banking system, but it lacks more advanced features such as transaction history, interest rates, and user authentication. Additionally, the code could benefit from better organization and separation of concerns, such as creating separate classes for the ATM and the bank account management.

# **Integrity**

## Hash Function

The code implements the SHA-1 hashing algorithm, which is a cryptographic hash function that produces a 160-bit (20-byte) hash value. The SHA-1 algorithm operates on messages of up to 2^64 bits and processes the message in 512-bit blocks.

## Message Authentication Code

The code also implements the HMAC algorithm, which is a mechanism for message authentication using a cryptographic hash function in combination with a secret key. The code uses four constants, called K values, as inputs to the SHA-1 algorithm.

These K values are used in the calculation of the intermediate hash values and are specific to the SHA-1 algorithm. The code includes several helper functions, including a padding function that pads the message to be hashed to ensure it is a multiple of 512 bits, a function for circular left shift rotation, and a function that applies the SHA-1 algorithm to the padded message to generate the hash value.

Finally, the code includes a function for generating a secret key for use with the HMAC algorithm.