[CS304] Introduction to Cryptography and Network Security

Course Instructor: Dr. Dibyendu Roy — Winter 2022-2023
Scribed by: Chitranshi Srivastava (202051055) — Lecture 23 and 24 (Week 13)

1 Secured Sockets Layer

Most of the cryptographic protocols are based on three cryptographic primitives:

- Symmetric Key Encryption Algorithms. These are preferred as computations in Public Key Cryptography are usually harder than symmetric key cryptography.
- Key Exchange Protocol so as to get the common shared key at both ends. A signature mechanism to verify the authenticity of the source and the public keys to avoid attacks such as Man in the Middle Attack.
- Message Authentication Code for authenticating the encrypted data. It will give the assurance that you have received the message from the correct party and the message has not been altered.

SSL protocol is implemented everywhere in HTTPS. It provides a secure communication between two parties where you're going to perform a key-exchange, encryption using symmetric key, a signature or authenticating the public keys, perform a Message Authentication Code to authenticate the message.

It starts with a connection. A **connection** is a transport that provides a suitable type of service. The connections are transient. Every connection is associated with one session. A **session** is when you hit a HTTPS website, you establish a session with their server. The session will going to have a time period and during this period you will be exchanging the data. All this data exchange will be completely encrypted. Sessions are created by the Handshake Protocol. Sessions define a set of cryptographic security parameters, which can be shared among multiple connections. Sessions are used to avoid the expensive negotiation of new security parameters for each connection.

A session state is defined by the following parameters:

1. **Session identifier:** An arbitrary byte sequence chosen by the server to identify an active or resumable session state.
2. **Peer certificate:** An X509.v3 certificate of the peer. This element of the state may be null. It is usually a signed public key. If we say that you are buying a SSL certificate that means you are basically getting your public key signed by a certified authority (using their secret key so that it is verifiable by everyone that the authority has authorised it). This certificate is used by the receiver to verify your public key. One such certified authority is X509.v3.
3. **Compression method:** The algorithm used to compress data prior to encryption. Corresponding to this algorithm, there is a decompression algorithm.

4. **Cipher spec:** It specifies the bulk data encryption algorithm (such as null, AES, etc.), a hash algorithm (such as MD5 or SHA-1) used for MAC calculation, a mechanism for key exchange (Diffie-Hellman, ECDH). It also defines cryptographic attributes such as the hash size.

5. **Master secret:** 48-byte secret shared between the client and server. It is used to generate certain keys such as key for encrypting the data and a key for generating MAC.

6. **Is resumable:** A flag indicating whether the session can be used to initiate new connections.

A connection state is defined by the following parameters:

1. **Server and client random:** Byte sequences that are chosen by the server and client for each connection. These are generated individually at the client and the server side. These numbers are involved in certain computation to prevent certain attacks.

2. **Server write MAC secret:** The secret key used in MAC operations on data sent by the server. It is same at the client and the server side.

3. **Client write MAC secret:** The symmetric key used in MAC operations on data sent by the client.

4. **Server write key:** The symmetric encryption key for data encrypted by the server and decrypted by the client.

5. **Client write key:** The symmetric encryption key for data encrypted by the client and decrypted by the server.

6. **Initialization vectors:** When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key. This field is first initialized by the SSL Handshake Protocol. Thereafter, the final ciphertext block from each record is preserved for use as the IV with the following record.

7. **Sequence numbers:** Each party maintains separate sequence numbers for transmitted and received messages for each connection. When a party sends or receives a "change cipher spec message", the appropriate sequence number is set to zero. Sequence numbers may not exceed $2^{64} - 1$.

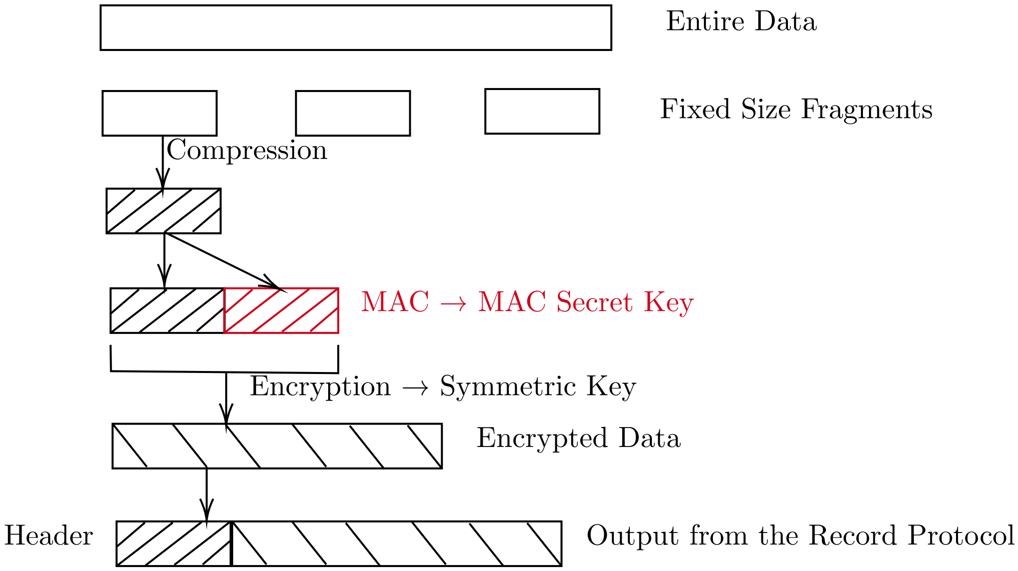
1.1 SSL Record Protocol

There are certain protocols involved in the entire SSL. The first protocol involved is the SSL Record Protocol. This protocol provides two services for SSL connections:

1. **Confidentiality:** The Handshake Protocol defines a shared secret key that is used for conventional encryption of SSL payloads.

2. **Message Integrity:** The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC).

Given a data, let's say you have a packet, first break the packet into multiple blocks of fixed size. The length of the block depends on the protocol version. On each block, perform the compression algorithm. It will compress the fragmented data to a fixed length size. On this compressed data, the MAC will be generated using the MAC Secret Key. This MAC will be concatenated with the compresses data. Now this data, that is, compressed data concatenated with MAC will be encrypted using an encryption algorithm. After encryption, a header will be added before the encrypted data. The header contain contains public information that we do not want to encrypt. This will be the complete packet that will be transferred either from client to server or vice-versa. A flowchart for the above described process is given below.



Now, suppose we have received this data from the server. The length of the header part is known, hence, it is discarded from the packet and any necessary information required is taken from it. Now, we will first perform the decryption and we will get the compressed data concatenated with the MAC. The length of MAC is known and hence that part will be taken away. Therefore, we have the compressed data from which we will generate a the MAC. If the generated MAC matches with the received MAC, then the data is authenticated. If it is matching, we will perform the decompression operation to get the original data contained in the packet.

The two keys required are MAC Secret Key and the key used for encryption. These two keys will be same at client and server side. The MAC that is generated is computed as:
MAC: hash(MAC_write_secret || pad2 || hash(MAC_write_secret || pad1 || seq_num || SSLCompressed.type || SSLCompressed.length || SSLCompressed.fragment))

The list of encryption algorithms supported by SSL is given below.

AES, IDEA, RC2-40, DES-40, DES, 3DES, Fortezza, RC4-40, RC4-128

There will be a negotiation between the client and the server and they will agree on one encryption algorithm.

Header is basically some public information that is not required to be encrypted. The header contain the following information.

- Content Type (8 bits): The type of data
- Major Version (8 bits): Suppose you are using version 3 of SSL, then major version will be 3.
- Minor Version (8 bits): Suppose you are using version 3 of SSL, then minor version will be 0.

- Compressed Length (16 bits): length of the compressed data.

1.2 Change Cipher Spec Protocol

Another protocol used in SSL is Change Cipher Spec Protocol. It basically update the cipher suite to be used on this connection. Suppose you are using the RC4 for encrypting the data. After certain time, the client and server want to change the encryption mechanism and want to use AES. This protocol is used in such scenarios.

1.3 Alert Protocol

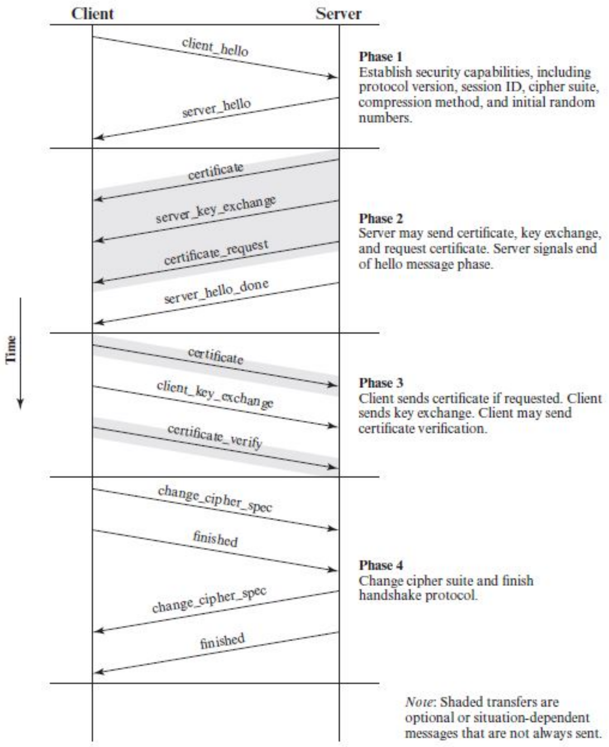
Its main function is to give some kind of alert if something is wrong. A list of few(not all) alerts is given below alongwith the description when these alerts are given by the Alert Protocol.

- **unexpected_message:** An inappropriate message was received.
- **bad_record_mac:** An incorrect MAC was received.
- **decompression_failure:** The decompression function received improper input (e.g., unable to decompress or decompress to greater than maximum allowable length).
- **handshake_failure:** Sender was unable to negotiate an acceptable set of security parameters given the options available.
- **illegal_parameter:** A field in a handshake message was out of range or inconsistent with other fields.
- **close_notify:** Notifies the recipient that the sender will not send any more messages on this connection. Each party is required to send a close_notify alert before closing the write side of a connection.
- **bad_certificate:** A received certificate was corrupt (e.g., contained a signature that did not verify).
- **unsupported_certificate:** The type of the received certificate is not supported.
- **certificate_revoked:** A certificate has been revoked by its signer.
- **certificate_expired:** A certificate has expired.
- **certificate_unknown:** Some other unspecified issue arose in processing the certificate, rendering it unacceptable.

1.4 Handshake Protocol

It is the most important protocol of the SSL. It performs the handshaking between the two parties and generate a common secret key. Using this secret key, the encryption and MAC keys are generated which are used for encryption and MAC generation.

Let us understand the working of the protocol. The initiation of communication between client and server will be done by the client. The first message from the client side is named as a hello message (client_hello). Some data will be passed from client to server in this hello message. In reply, the server will send its hello message (server_hello). It contains some data which will be used for further communication. Then the server, will send the certificate (signed public key) to



Phase 1
Establish security capabilities, including protocol version, session ID, cipher suite, compression method, and initial random numbers.

Phase 2
Server may send certificate, key exchange, and request certificate. Server signals end of hello message phase.

Phase 3
Client sends certificate if requested. Client sends key exchange. Client may send certificate verification.

Phase 4
Change cipher suite and finish handshake protocol.

Note: Shaded transfers are optional or situation-dependent messages that are not always sent.

Figure 1: Action of Handshake Protocol

the client and will begin the key exchange. It will send some data which will be required for key exchange(server_key_exchange). Server might ask the client for the certificate for client's browser (certificate_request). Finally, the server will send the hello done message. Using this, it can be identified that the server's hello is done here.

In reply, the client will send the certificate if requested. Then client will send the key-exchange data (client_key_exchange). If the client and server are agreeing on Diffie-Hellman key exchange, then they will have to share their Diffie-Hellman Public Keys with each other, otherwise key cannot be established. The client will verify the certificate given by the server and send a reply (certificate_verify). The will send a cipher spec message (change_cipher_spec) indicating what are the ciphers client supports. Afterwards, the client will send a finish message (finished). The server will reply with a cipher spec message (change_cipher_spec) indicating if the server can support the requested cipher or not or some algorithm for further communication. If it is supported, then the server will send a finish message (finished).

Now, let us see what are the information which will be a part of the different messages from the client side and the server side.

1. **client_hello_message:** It contains the following data:
    - Version: the version of SSL protocol that client supports
    - Random: A client-generated random structure consisting of a 32-bit timestamp and 28 bytes generated by a secure random number generator.
    - **Session ID:** A variable-length session identifier. A nonzero value indicates that the

client wishes to update the parameters of an existing connection or to create a new connection on this session. A zero value indicates that the client wishes to establish a new connection on a new session.

- **Cipher Suite:** This is a list that contains the combinations of cryptographic algorithms supported by the client, in decreasing order of preference. Each element of the list (each cipher suite) defines both a key exchange algorithm and a CipherSpec; these are discussed subsequently.
- **Compression Method:** This is a list of the compression methods the client supports.

After sending the client_hello_message, the client waits for the server_hello_message, which contains the same parameters as the client_hello_message. Let us now go through the contents of the Cipher Suite parameter in the hello messages.

The first element of the Ciphersuite parameter is the key exchange method. The following key exchange methods are supported:

- **RSA:** The secret key is encrypted with the receiver's RSA public key. A publickey certificate for the receiver's key must be made available.
- **Fixed Diffie-Hellman:** This is a Diffie–Hellman key exchange in which the server's certificate contains the Diffie–Hellman public parameters signed by the certificate authority (CA). That is, the public-key certificate contains the Diffie–Hellman public-key parameters. The client provides its Diffie–Hellman public-key parameters either in a certificate, if client authentication is required, or in a key exchange message. This method results in a fixed secret key between two peers based on the Diffie–Hellman calculation using the fixed public keys.
- **Ephemeral Diffie-Hellman:** This technique is used to create ephemeral (temporary, one-time) secret keys. In this case, the Diffie–Hellman public keys are exchanged and signed using the sender's private RSA or DSS key. The receiver can use the corresponding public key to verify the signature. Certificates are used to authenticate the public keys.
- **Anonymous Diffie-Hellman:** The base Diffie–Hellman algorithm is used with no authentication. That is, each side sends its public Diffie–Hellman parameters to the other with no authentication. This approach is vulnerable to man-in-the-middle attacks.

Following the definition of a key exchange method is the CipherSpec, which includes the following fields:

- **CipherAlgorithm:** Any of the algorithms mentioned earlier: RC4, RC2, DES, 3DES, DES40, or IDEA
- **MACAlgorithm:** MD5 or SHA-1
- **CipherType:** Stream or Block
- **IsExportable:** True or False
- **HashSize:** 0, 16 (for MD5), or 20 (for SHA-1) bytes
- **Key Material:** A sequence of bytes that contain data used in generating the write keys
- **IV Size:** The size of the Initialization Value for Cipher Block Chaining (CBC) encryption

2. **server_key_exchange message:** It is not required in two instances: (1) The server has sent a certificate with fixed Diffie–Hellman parameters; or (2) RSA key exchange is to be used. The server_key_exchange message is needed for the following:

   - **Anonymous Diffie–Hellman:** The message content consists of the two global Diffie–Hellman values (a prime number and a primitive root of that number) plus the server's public Diffie–Hellman key.
   - **Ephemeral Diffie–Hellman:** The message content includes the three Diffie–Hellman parameters provided for anonymous Diffie–Hellman plus a signature of those parameters.
   - **RSA key exchange (in which the server is using RSA but has a signature-only RSA key):** Accordingly, the client cannot simply send a secret key encrypted with the server's public key. Instead, the server must create a temporary RSA public/private key pair and use the server_key_exchange message to send the public key. The message content includes the two parameters of the temporary RSA public key plus a signature of those parameters.

   As usual, a signature is created by taking the hash of a message and encrypting it with the sender's private key. In this case, the hash is defined as:

   $$\text{hash(ClientHello.random} \mid\mid \text{ServerHello.random} \mid\mid \text{ServerParams)}$$

   So the hash covers not only the Diffie–Hellman or RSA parameters but also the two nonces from the initial hello messages.

3. **client_key_exchange message:** The content of the message depends on the type of key exchange, as follows:

   - **RSA:** The client generates a 48-byte pre-master secret and encrypts with the public key from the server's certificate or temporary RSA key from a server_key_exchange message. Its use to compute a master secret is explained later.
   - **Ephemeral or Anonymous Diffie–Hellman:** The client's public Diffie–Hellman parameters are sent.
   - **Fixed Diffie–Hellman:** The client's public Diffie–Hellman parameters were sent in a certificate message, so the content of this message is null.

4. **certificate_verify message:** Client may send a certificate_verify message where the client is going to generate two hash values.

   - CertificateVerify.signature.md5_hash = MD5(master_secret || pad_2 || MD5(handshake_messages || master_secret || pad_1));
   - CertificateVerify.signature.sha_hash = SHA(master_secret || pad_2 || SHA(handshake_messages || master_secret || pad_1));

   The master_secret is generated using the pre-master secret key. On this hash value the client will produce a signature. This entire hash will be signed using the private key of the client and will be sent to the server, if the client wants the certificate_verify message.

5. **change_cipher_spec message:** The client sends a change_cipher_spec message and copies the pending CipherSpec into the current CipherSpec. The client then immediately sends the finished message. The contents of the finished message are concatenation of two hash values. These hash values are given below:

  - MD5(master_secret || pad_2 || MD5(handshake_messages || Sender || master_secret || pad_1));
  - SHA(master_secret || pad_2 || SHA(handshake_messages || Sender || master_secret || pad_1));

6. In response to the above mentioned two messages, the server sends its own change_cippher_spec message and transfers the pending to the current CipherSpec, and sends its finished message.

Let us now see how the master secret key is generated using the pre-master key. The master secret key is generated as:
master_secret = MD5(pre_master_secret || SHA(A || pre_master_secret || ClientHello.random || ServerHello.random)) || MD5(pre_master_secret || SHA(BB || pre_master_secret || ClientHello.random || ServerHello.random)) || MD5(pre_master_secret || SHA(CCC || pre_master_secret || ClientHello.random || ServerHello.random))
A, BB and CCC are pre-defined constants. Now, both the parties have master secret key. The involvement of random numbers in master secret key is explained below.

Suppose, they are having only fixed Diffie-Hellman (no ephemeral Diffie-Hellman). If we have the Diffie-Hellman public keys and we perform Diffie Hellman Key Exchange for multiple times, you will lead to have the same secret key everytime. Now, let us say one session has been corrupted and some data has been leaked (not the secret key). But you can see that the master secret will be used for encryption and the MAC generation. If one session is corrupted, then the second session cannot be corrupted because we are involving a random number as an input to a hash function which are different for every session. Hence, the master secret key is different for every session. These values serve as nonces and are used during key exchange to prevent replay attacks.

Using the master secret key, we will be generating a block of keys. This key generation will be continue till you have sufficient amount of keys.
key_block = MD5(master_secret || SHA(A || master_secret || ServerHello.random || ClientHello.random)) || MD5(master_secret || SHA(BB || master_secret || ServerHello.random || ClientHello.random)) || MD5(master_secret || SHA(CCC || master_secret || ServerHello.random || ClientHello.random)) || ...

This will be the keyblock where few bits are for encryption, few bits for MAC and few bits for the IV. This entire concatenation will continue till you reach the required number of bits.

2 An Overview on Signal Protocol

When we use message applications like Whatsapp, the communication channel is public but we want our messages to be only read by the specific receiver. So, we need End-to-End Encryption. For this, we use a server which sends the message but has no idea about the plaintext. The server is not able to decrypt the message. In this setup, shared key is involved and it is shared between the two parties only.
We need to recall the following to understand Signal protocol:

- Elliptic Curve Cryptography
- Diffie Hellman Key Exchange
- AES-256 algorithm

- CBC mode of operation of AES

2.1 Authenticated Encryption with Associated Data (AD)

- Associated Data(AD) is authenticated but not encrypted
- Schemes are nonce-based(and deterministic)

Sender : C = Enc(K, N, AD, M)

K = key
N = Nonce(random number to be only used once)
AD = Associated Data
M = Message

$$\text{Sender} \xrightarrow{\text{N, AD, C}} \text{Receiver}$$

Receiver : M = Dec(K, N, AD, C)

We are sending AD along with the message over the server but AD is not encrypted. If the AD received by the receiver is correct, only then my decryption is valid. So, we can say that AD is authenticating the source of the message.

2.2 Details about Signal Protocol

- Generation of shared secret key
- Ensuring End-to-End Encryption
- Secure Transfer of data from one user to another via Server
- Ensuring Forward Secrecy(For example: Even if the secrecy of 10th message is compromised, the previous communication will still be secure). In signal protocol, suppose secrecy of mth message is compromised, still 1 to m-1 messages are secure. Also, messages from m+1 are also secure.

This protocol is used in several apps like whatsapp, signal, facebook messenger, skype.
It is an open source protocol: https://signal.org/

2.3 Main Cryptographic modules of Signal Protocol

- X3DH(Extended Triple Diffie-Hellman)
- ECDSA
- Double Ratchet

X3DH is used to generate the shared keys. ECDSA is used to sign the public keys so that no one else can produce valid public key on behalf of me. Finally Double Ratchet performs the encryption and different keys are generated for every message.
Note: Curve X25519 or X448 is used in ECDH and ECDSA.

2.3.1 Registration

Alice registers himself to the Server. He has to provide the id to register. Like, for whatsapp, phone number is the id and OTP is used to verify it.

$$\textbf{Alice} \xrightarrow{\text{Registration}} \textbf{Server}$$

During this process, a packet of keys are generated. Let us say that Alice generates the $Packet_A$ which consists of the following:

- **Identity key :** $IK_A \rightarrow\ (n_{IA}, n_{IA}P)$
  Updated once to the server - Long term

- **Signed prekey :** $SPK_A \rightarrow\ (n_{SPA}, n_{SPA}P)$
  Updated weekly or once in a month

- **Prekey Signature : Sig(**$IK_A,\ Encode(SPK_A)$**)**
  Updated weekly or once in a month
  Note: $n_{SPA}P$ is signed using secret key component of IK_A

- **A set of one-time prekeys :** $OPK^1{}_A,\ OPK^2{}_A,\ OPK^3{}_A$
  (Optional) Uploaded when the server requests for it

This packet is then uploaded to the server.
Alice then sends $Packet_A$ to the Server.

$$\textbf{Alice} \xrightarrow{Packet_A} \textbf{Server}$$

Similarly, Bob will do the same thing.

$$\textbf{Bob} \xrightarrow{\text{Registration}} \textbf{Server}$$

Bob generates the $Packet_B$ which consists of the following:

- **Identity key :** $IK_B \rightarrow\ (n_{IB}, n_{IB}P)$
  Updated once to the server - Long term

- **Signed prekey :** $SPK_B \rightarrow\ (n_{SPB}, n_{SPB}P)$
  Updated weekly or once in a month

- **Prekey Signature : Sig(**$IK_B,\ Encode(SPK_B)$**)**
  Updated weekly or once in a month
  Note: $n_{SPB}P$ is signed using secret key component of IK_B

- **A set of one-time prekeys :** $OPK^1{}_B,\ OPK^2{}_B,\ OPK^3{}_B$
  (Optional) Uploaded when the server requests for it

This packet is then uploaded to the server.

$$\textbf{Bob} \xrightarrow{Packet_B} \textbf{Server}$$

Server stores both the packets.

Alice	**Server**	**Bob**
PacketA	PacketA, PacketB	PacketB

$$\textbf{Alice} \xleftarrow{\text{PacketB}} \textbf{Server}$$

$$\textbf{Alice} \overset{\text{Verify?}}{=} \text{Signature of PacketB}$$

Alice	**Server**	**Bob**
PacketA, PacketB	PacketA, PacketB	PacketB

Alice verifies the signature using the public key of Bob($n_{IB}P$. Now Alice has all public components of Bob's packet(IK_B, SPK_B) and her packet(IK_A, SPK_A). Now she performs the Diffie-Hellman key exchange.

1. Alice generates an ephemeral key $EK_A = n_{EA},\ n_{EA}P$
2. $DH_1 = \text{DH}(IK_A,\ SPK_B)$
   DH is performed using $n_{IA},\ n_{SPB}P$
3. $DH_2 = \text{DH}(EK_A,\ IK_B)$
   DH is performed using $n_{EA},\ n_{IB}P$
4. $DH_3 = \text{DH}(EK_A,\ SPK_B)$
   DH is performed using $n_{EA},\ n_{SPB}P$
5. If key bundle contains one-time prekey,
   - $DH_4 = \text{DH}(EK_A,\ OPK_B)$
   - $SK_A = \text{KDF}(DH_1 \mid\mid DH_2 \mid\mid DH_3)$
6. Else $SK_A = \text{KDF}(DH_1 \mid\mid DH_2 \mid\mid DH_3)$

Here, KDF = SHA-256. Now we are done with step 1, that is, X3DH. Let us move forward.

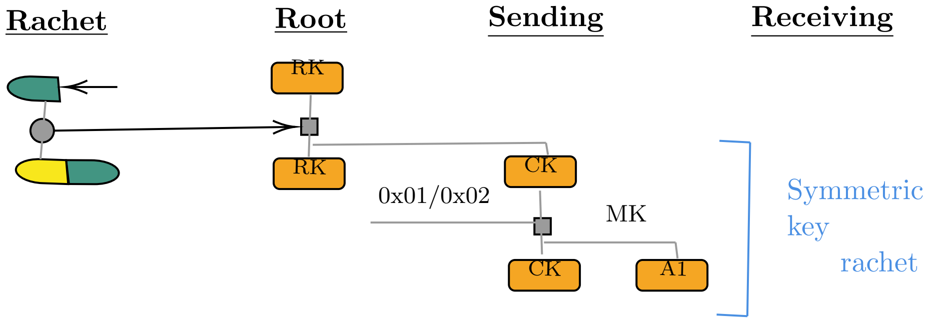
1. Alice computes associated data
   AD = Encode(IK_A) || Encode(IK_B)
   (Only public parts are used here)
2. Alice encrypts an initial message using AD, SK_A and generates the corresponding initial ciphertext.

$$\text{Alice} \xrightarrow{\text{initial ciphertext, } IK_A,\ EK_A, \text{ Bob's Prekeys}} \text{Server}$$

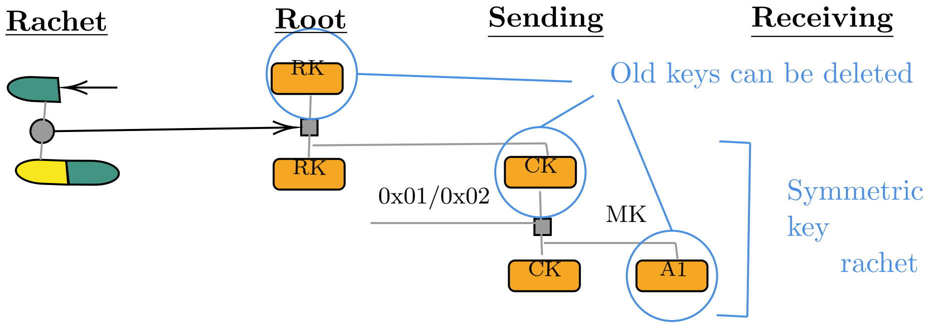
$$\text{Server} \xrightarrow{\text{initial ciphertext, } IK_A,\ EK_A, \text{ Bob's Prekeys}} \text{Bob}$$

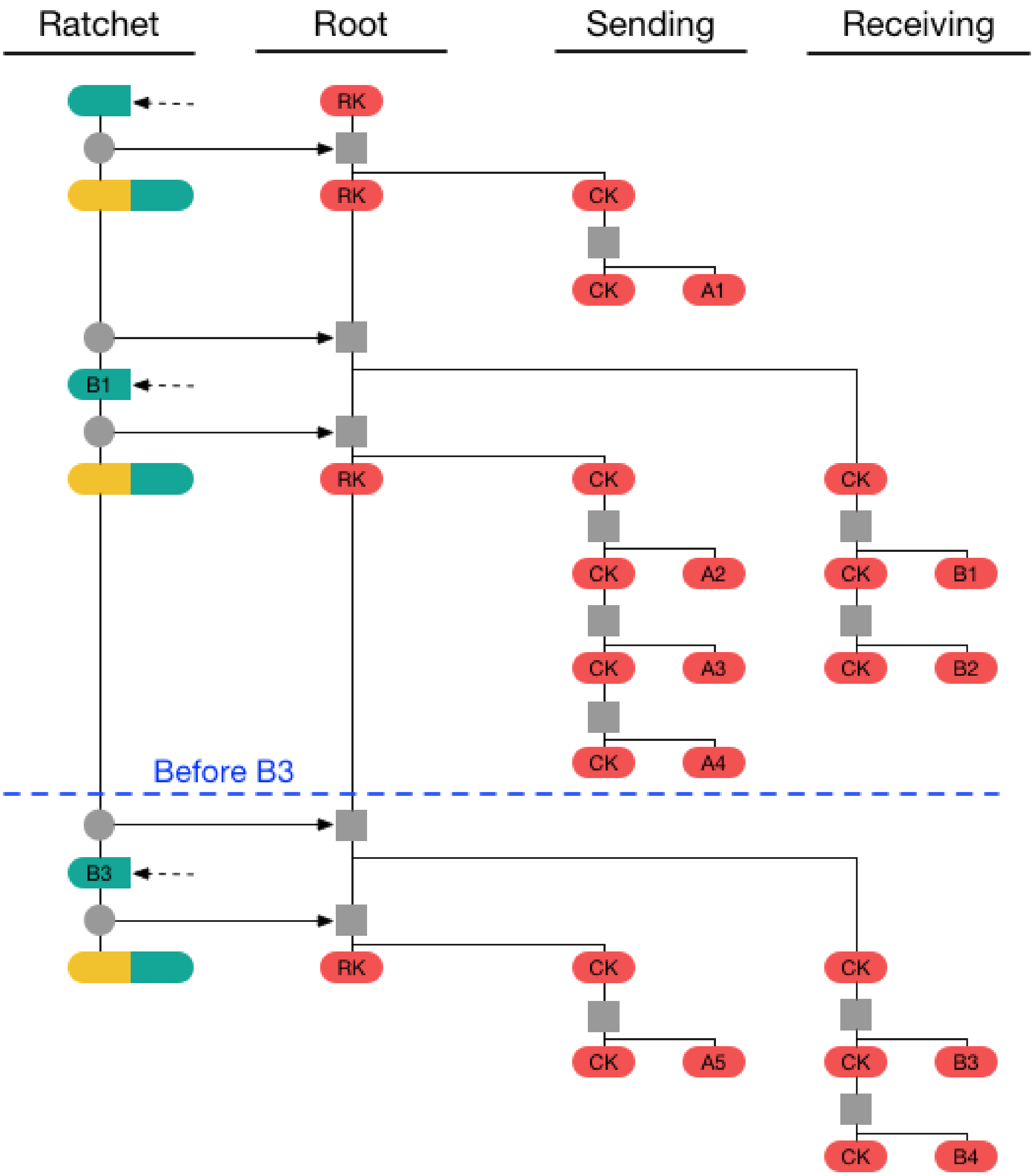
Now, let us see what happens on Bob's side.

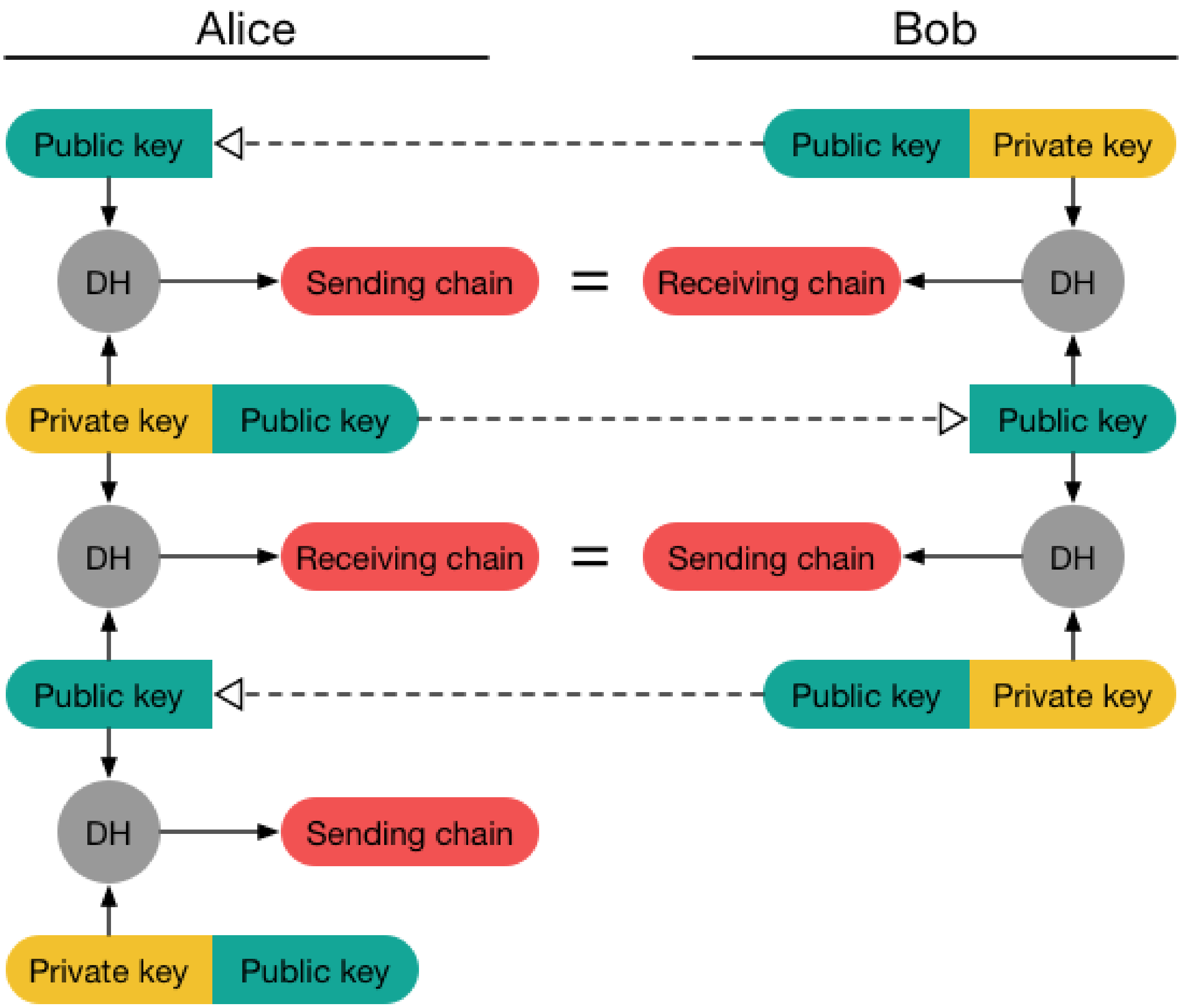
1. Bob recovers IK_A, EK_A first.
2. Bob performs DH(same as Alice) and KDF by using his secret keys.
   $SK_B = \text{KDF}(DH_1 \mid\mid DH_2 \mid\mid DH_3)$
3. Due to properties of DH and KDF, $SK_B = SK_A$.
4. Constructs AD = Encode(IK_A) || Encode(IK_B) (public parts)
5. Decrypts the initial ciphertext using SK_B), AD and recovers the original message.



Where RK = root key, CK = chain key. When input is 0x01, new chain key is created. When input is 0x02, A1 is created which is used to encrypt second message. Let us see the chain process.




Ratchet
Root
Sending
Receiving
RK
RK
CK
CK
A1
B1
RK
CK
CK
CK
A2
CK
B1
CK
A3
CK
B2
CK
A4
Before B3
B3
RK
CK
CK
CK
A5
CK
B3
CK
B4


Alice
Bob
Public key
Public key
Private key
DH
Sending chain
=
Receiving chain
DH
Private key
Public key
Public key
DH
Receiving chain
=
Sending chain
DH
Public key
Public key
Private key
DH
Sending chain
Private key
Public key

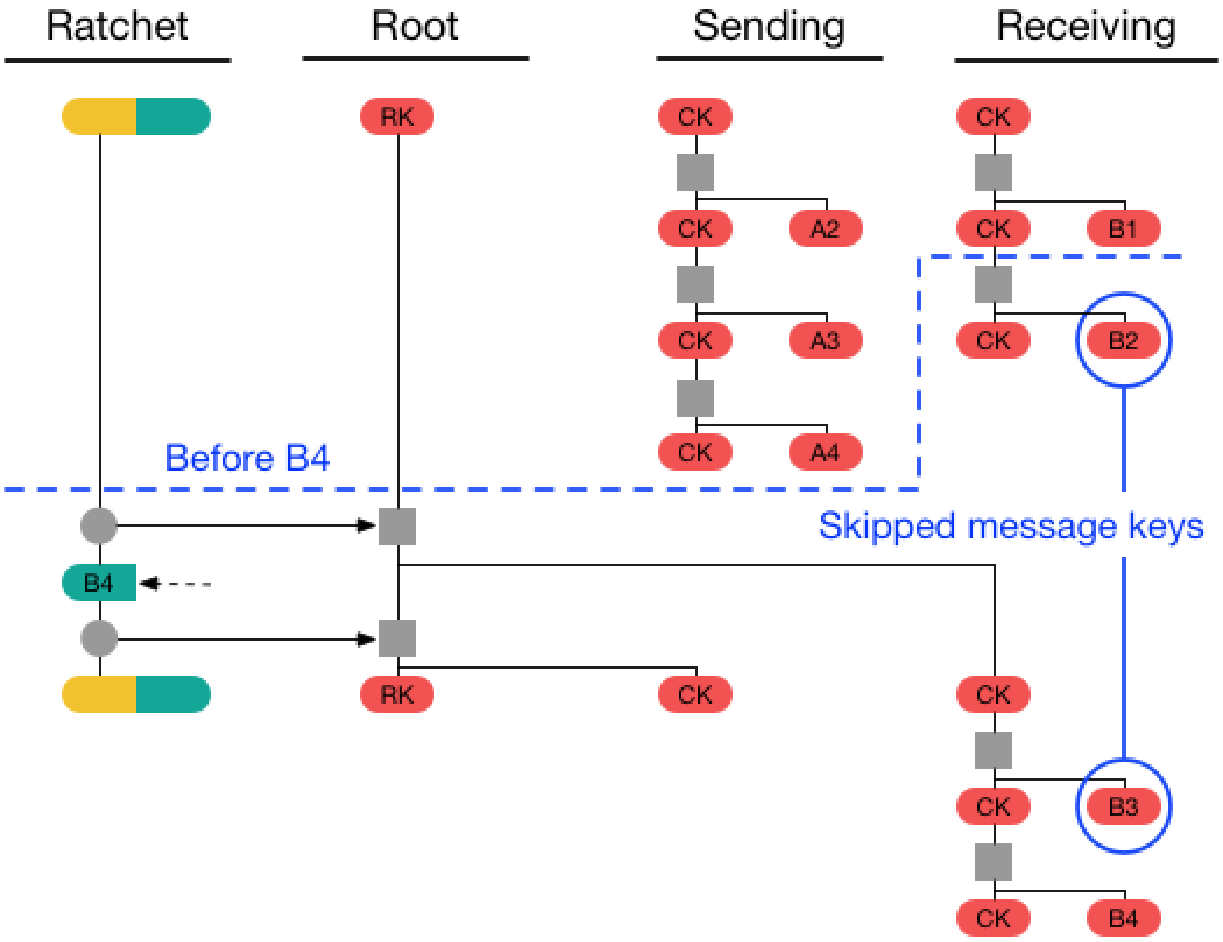
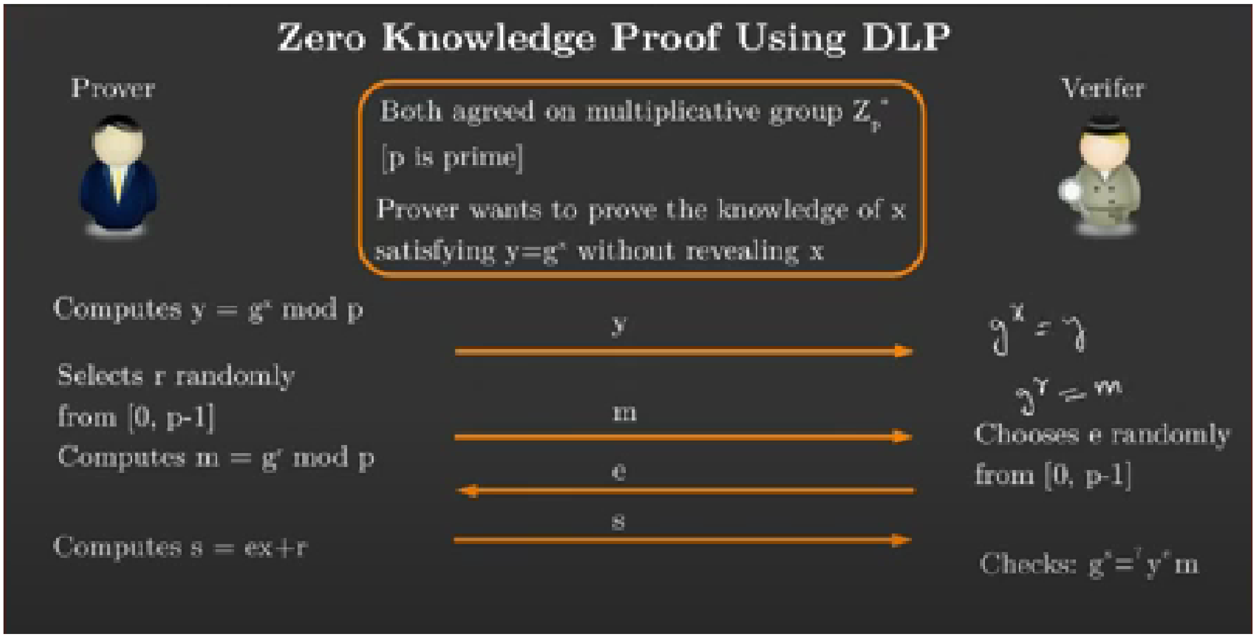


Figure 2: In this figure, we can see the change in sequence of messages because of internet speed.

3 Zero Knowledge Proof using DLP

Suppose I have certain document and I want to prove that I am the owner of it without sharing the document.



Zero Knowledge Proof Using DLP

Prover

Verifer

Both agreed on multiplicative group Z_p^* [p is prime]

Prover wants to prove the knowledge of x satisfying $y=g^x$ without revealing x

Computes $y = g^x \bmod p$

y

$g^x = y$

Selects r randomly from [0, p-1]

Computes $m = g^r \bmod p$

m

$g^r = m$

e

Chooses e randomly from [0, p-1]

Computes $s = ex+r$

s

Checks: $g^s \overset{?}{=} y^e m$

$$g^s = g^{ex+r} = g^{ex} \cdot g^r = y^e m$$

4 Recall

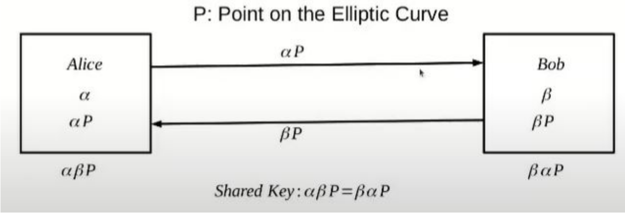


Figure 3: Classical Diffie-Hellman on Elliptic Curve

The structure for the AES-128 encryption algorithm is given below. Although, we will be using the AES-256 but the basic structure remains the same. The AES-256 has only three differences when compared to AES-128. These are mentioned below:

1. AES-256 takes a key of length of 256-bits, whereas AES-128 takes 128-bit key as input.

2. The Number of Rounds in AES-256 are 14 which are 10 for AES-128.

3. The Key Scheduling Algorithm for AES-256 differs slightly from Key Scheduling Algorithm of AES-128 as more round keys need to be generated in AES-256.

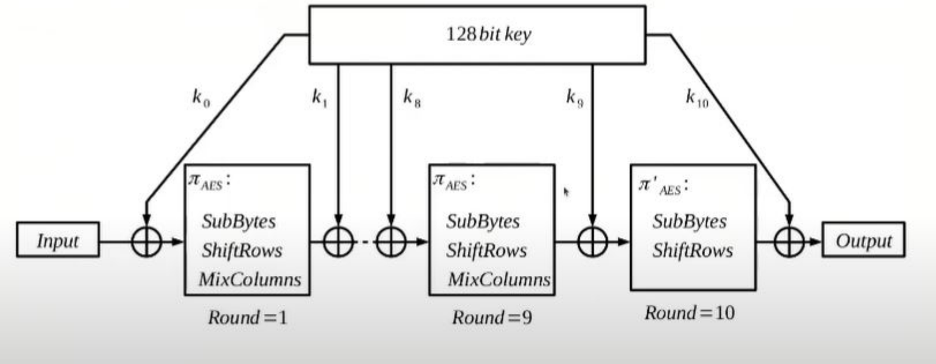
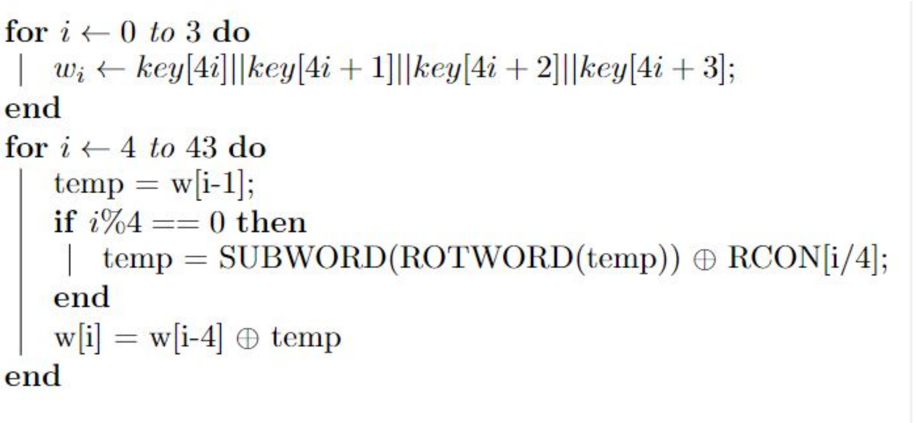


Figure 4: Structure of AES-128

The Key Scheduling Algorithm for both AES-128 and AES-256 are given below.



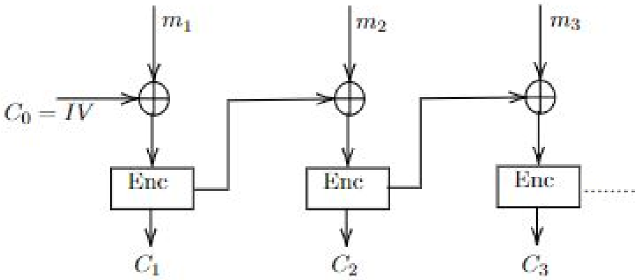
```
for $i \leftarrow 0$ $to$ 3 do
    $w_i \leftarrow key[4i]||key[4i+1]||key[4i+2]||key[4i+3]$;
end
for $i \leftarrow 4$ $to$ 43 do
    temp = w[i-1];
    if $i\%4 == 0$ then
        temp = SUBWORD(ROTWORD(temp)) $\oplus$ RCON[i/4];
    end
    w[i] = w[i-4] $\oplus$ temp
end
```

Figure 5: KSA of AES-128

```
for $i \leftarrow 0$ to 7 do
    $w_i \leftarrow key[4i]||key[4i+1]||key[4i+2]||key[4i+3]$;
end
for $i \leftarrow 8$ to 59 do
    temp = w[i-1];
    if $i\%8 == 0$ then
        temp = SUBWORD(ROTWORD(temp)) $\oplus$ RCON[i/4];
    end
    else if $i\%8 == 4$ then
        temp = SUBWORD(temp);
    end
    w[i] = w[i-4] $\oplus$ temp
end
```

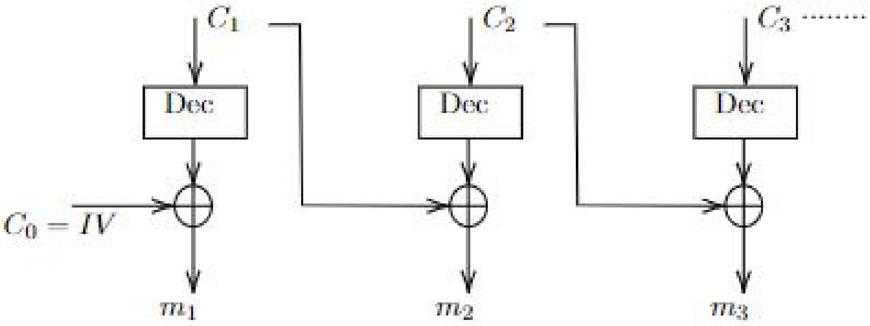
Figure 6: KSA of AES-256

$$C_0 = IV$$
$$C_i = Enc(C_{i-1} \oplus m_i, K) \ \forall \ i \in \{1, 2, 3...., t\}$$
$$C = C_0||C_1||.....||C_t$$



Encryption in CBC Mode

$$m_i = Dec(C_i, K) \oplus C_{i-1} \ \forall \ i \in \{1, 2..., t\}$$
$$\text{where } C_0 = IV$$
$$M = m_1||m_2||.....||m_t$$



Decryption in CBC Mode