Rabbit Cipher
McEliece Cryptosystem
ECDSA Signature

Liad Vaksman
Elroye Cahana
Or Steiner
Lior Wunsch

The Challenge

1. Alice wants to make sure that nobody will see her request except the server.
2. The server wants to make sure that Alice is the one who sent the request.

The Solution

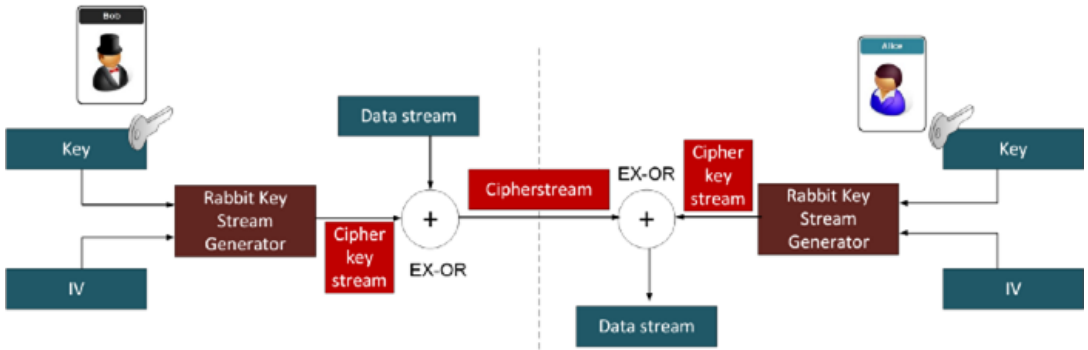
1. Encrypt the Read/Write request - using Rabbit Cipher.
2. Encrypt the cipher key - using McEliece Cryptosystem.
3. Sign the request via Digital Signature - ECDSA Version.

Rabbit Cipher - Background

- Designed by: Martin Boesgaard, Mette Vesterager, Thomas Pedersen, Jesper Christiansen and Ove Scavenius.
- Submitted to the eSTREAM project of the EU CRYPT network.
- Optimized for embedded systems such as RFID and Sensor Networks.

Rabbit Cipher - Specification

- Rabbit is a symmetric stream cipher.
- Supports a key size of 128 bits.
- Uses an initialization vector of 64 bits (optional).
- Uses a state machine for key generation.



Rabbit Cipher - Notations

$A^{[g..h]}$ = bits h through g of variable A

K = the input key

$A_{j,i}$ = variable A of j^{th} subsystem at iteration i

K_j = j^{th} subkey of K

IV = initialization vector

$x_{j,i}$ = state variable

$c_{j,i}$ = counter variable

$\phi_{j,i}$ = counter carry bit

$\diamondsuit$ = concatenation of two bit sequences

s_i = 128-bit cipher keystream block at iteration i

Rabbit Cipher - Key Generation

1. **Key Setup Scheme**

The key, $K^{[127..0]}$, is divided into eight subkeys : $k_0 = K^{[15..0]}, \ldots, k_7 = K^{[127..112]}$.

The state and counter variables are initialized from the subkeys as follows :

$$x_{j,0} = \begin{cases} k_{(j+1 \bmod 8)} \diamond k_j & \text{for } j \text{ even} \\ k_{(j+5 \bmod 8)} \diamond k_{(j+4 \bmod 8)} & \text{for } j \text{ odd} \end{cases}$$

$$c_{j,0} = \begin{cases} k_{(j+4 \bmod 8)} \diamond k_{(j+5 \bmod 8)} & \text{for } j \text{ even} \\ k_j \diamond k_{(j+1 \bmod 8)} & \text{for } j \text{ odd.} \end{cases}$$

Rabbit Cipher - Key Generation

2. Counter System

The counter carry bit, $\phi_{j,i+1}$ is given by :

$$\phi_{j,i+1} = \begin{cases} 1 & \text{if } c_{0,i} + a_0 + \phi_{7,i} \geq 2^{32} \wedge j = 0 \\ 1 & \text{if } c_{j,i} + a_j + \phi_{j-1,i+1} \geq 2^{32} \wedge j > 0 \\ 0 & \text{otherwise.} \end{cases}$$

The a_j constants are defined as:

$$\begin{aligned} a_0 &= \text{0x4D34D34D} & a_1 &= \text{0xD34D34D3} \\ a_2 &= \text{0x34D34D34} & a_3 &= \text{0x4D34D34D} \\ a_4 &= \text{0xD34D34D3} & a_5 &= \text{0x34D34D34} \\ a_6 &= \text{0x4D34D34D} & a_7 &= \text{0xD34D34D3.} \end{aligned}$$

Rabbit Cipher - Key Generation

2. Counter System

The dynamics of the counters is defined as follows :

$$
\begin{aligned}
c_{0,i+1} &= c_{0,i} + a_0 + \phi_{7,i} \mod 2^{32} \\
c_{1,i+1} &= c_{1,i} + a_1 + \phi_{0,i+1} \mod 2^{32} \\
c_{2,i+1} &= c_{2,i} + a_2 + \phi_{1,i+1} \mod 2^{32} \\
c_{3,i+1} &= c_{3,i} + a_3 + \phi_{2,i+1} \mod 2^{32} \\
c_{4,i+1} &= c_{4,i} + a_4 + \phi_{3,i+1} \mod 2^{32} \\
c_{5,i+1} &= c_{5,i} + a_5 + \phi_{4,i+1} \mod 2^{32} \\
c_{6,i+1} &= c_{6,i} + a_6 + \phi_{5,i+1} \mod 2^{32} \\
c_{7,i+1} &= c_{7,i} + a_7 + \phi_{6,i+1} \mod 2^{32}
\end{aligned}
$$

Rabbit Cipher - Key Generation

3. Next-state Function

The core of the Rabbit algorithm is the iteration of the system defined by :

$$
\begin{aligned}
x_{0,i+1} &= g_{0,i} + (g_{7,i} \lll 16) + (g_{6,i} \lll 16) \\
x_{1,i+1} &= g_{1,i} + (g_{0,i} \lll 8) + g_{7,i} \\
x_{2,i+1} &= g_{2,i} + (g_{1,i} \lll 16) + (g_{0,i} \lll 16) \\
x_{3,i+1} &= g_{3,i} + (g_{2,i} \lll 8) + g_{1,i} \\
x_{4,i+1} &= g_{4,i} + (g_{3,i} \lll 16) + (g_{2,i} \lll 16) \\
x_{5,i+1} &= g_{5,i} + (g_{4,i} \lll 8) + g_{3,i} \\
x_{6,i+1} &= g_{6,i} + (g_{5,i} \lll 16) + (g_{4,i} \lll 16) \\
x_{7,i+1} &= g_{7,i} + (g_{6,i} \lll 8) + g_{5,i}
\end{aligned}
$$

$$
g_{j,i} = \left((x_{j,i} + c_{j,i+1})^2 \oplus ((x_{j,i} + c_{j,i+1})^2 \gg 32)\right) \bmod 2^{32}
$$

* all additions are modulu 2^{32}

Rabbit Cipher - Key Generation

4. Extraction Scheme

After each iteration the output is extracted as follows :

$$
\begin{aligned}
s_i^{[15..0]} &= x_{0,i}^{[15..0]} \oplus x_{5,i}^{[31..16]} & \qquad s_i^{[31..16]} &= x_{0,i}^{[31..16]} \oplus x_{3,i}^{[15..0]} \\
s_i^{[47..32]} &= x_{2,i}^{[15..0]} \oplus x_{7,i}^{[31..16]} & \qquad s_i^{[63..48]} &= x_{2,i}^{[31..16]} \oplus x_{5,i}^{[15..0]} \\
s_i^{[79..64]} &= x_{4,i}^{[15..0]} \oplus x_{1,i}^{[31..16]} & \qquad s_i^{[95..80]} &= x_{4,i}^{[31..16]} \oplus x_{7,i}^{[15..0]} \\
s_i^{[111..96]} &= x_{6,i}^{[15..0]} \oplus x_{3,i}^{[31..16]} & \qquad s_i^{[127..112]} &= x_{6,i}^{[31..16]} \oplus x_{1,i}^{[15..0]}
\end{aligned}
$$

Rabbit Cipher - Key Generation

5. Key Setup Scheme

The system is iterated 4 times, according to the next-state function, to diminish correlations between bits in the key and bits in the internal state variables.

Finally, the counter variables are re-initialized, to prevent recovery of the key by inversion of the counter system, according to :

$$c_{j,4} = c_{j,4} \oplus x_{(j+4 \bmod 8),4}$$

Rabbit Cipher - Key Generation

6. IV Setup Scheme

Let the internal state after the key setup scheme be denoted the master state and let a copy of this master state be modified according to the IV scheme.

The counters are modified as :

$$
\begin{aligned}
c_{0,4} &= c_{0,4} \oplus IV^{[31..0]} & \qquad c_{1,4} &= c_{1,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\
c_{2,4} &= c_{2,4} \oplus IV^{[63..32]} & \qquad c_{3,4} &= c_{3,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]}) \\
c_{4,4} &= c_{4,4} \oplus IV^{[31..0]} & \qquad c_{5,4} &= c_{5,4} \oplus (IV^{[63..48]} \diamond IV^{[31..16]}) \\
c_{6,4} &= c_{6,4} \oplus IV^{[63..32]} & \qquad c_{7,4} &= c_{7,4} \oplus (IV^{[47..32]} \diamond IV^{[15..0]}).
\end{aligned}
$$

Rabbit Cipher - Encryption | Decryption

The extracted bits are XOR'ed with the plaintext / ciphertext to encrypt / decrypt.

Encryption: $c_i = p_i \oplus s_i$

Decryption: $p_i = c_i \oplus s_i$

where :

c_i = the i^{th} 128-bit ciphertext block.

p_i = the i^{th} 128-bit plaintext block.

Rabbit Cipher - Security

Brute force:

- When not using IV : the key is 128 bit long, therefore it would take 2^{128} tries to find the key.
- When using IV : the first 4 blocks can be found in 2^{128} as mentioned above, but the next iterations will be dependent on the IV value which would take an additional 2^{64} tries to find.

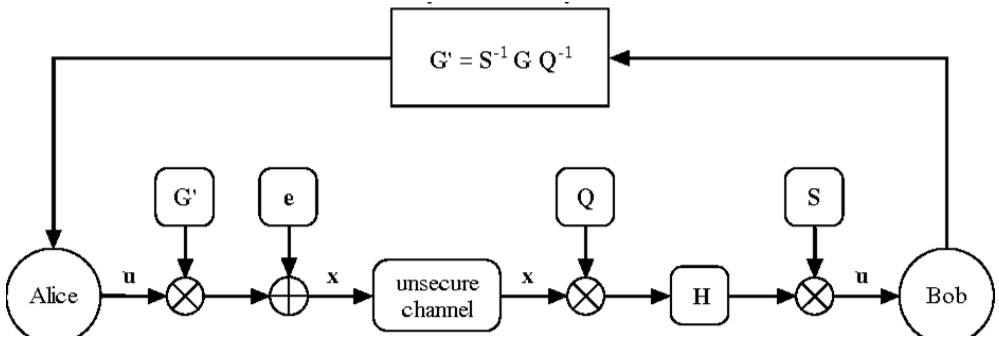
McEliece Cryptosystem - Background

- Designed by: Robert McEliece.
- A candidate for "post-quantum cryptography".
- The first to use randomization in the encryption process.

Robert McEliece

McEliece Cryptosystem - Specification

- McEliece is an asymmetric encryption algorithm.
- Suggested security parameter sizes are $n = 1024, k = 524, t = 50$, resulting in a public key size of 262,000 bits.
- Uses an error-correcting code and a linear code for key generation.



McEliece Cryptosystem - Notations

n, k, t = a set of common security parameters shared between all users

C = a public knowledge (n, k) linear code capable of efficiently correcting t errors

A = an efficient decoding algorithm

G = a random (n, k) binary generator matrix for C

S = a random (k, k) binary non-singular invertible matrix

P = a random (n, n) binary permutation matrix.

c = ciphertext

x = a message the size of k bits

e = a random binary error vector the size of n bits

McEliece Cryptosystem - Asymmetric

- Everyone has a Public Key and a Secret Key.
- Alice encrypts her message to Bob using his Public Key.
- Only Bob’s secret key can decode the message.

McEliece Cryptosystem - Key Generation

1. Bob chooses his secret key (S, G, P) :

   $G \in {F_2}^{n\ x\ k}$, a generator matrix for an appropriate linear code C.

   $S \in {F_2}^{k\ x\ k}$, a random invertible matrix.

   $P \in {F_2}^{n\ x\ n}$, a random permutation matrix.

2. Bob calculates his public key $\left(\hat{G} = S\ x\ G\ x\ P,\ t\right)$.

McEliece Cryptosystem - Encryption

Suppose Alice wishes to send Bob a message x :

1. Alice chooses a random error vector $e \in {F_2}^n$ of weight t.
2. Alice uses Bob's public key $(\hat{G},\ t)$ to calculate the ciphertext.

   She calculates $c = \hat{G} * x + e$ and sends to Bob.

McEliece Cryptosystem - Decryption

Upon receipt of c, Bob performs the following steps to decrypt the message :

1. Bob computes $P^{-1} * c = P^{-1}(\hat{G}x + e) = G * S * x + P^{-1} * e = G(Sx) + e'$.
2. Bob uses the decoding algorithm A to get $S * x$.
3. Bob computes $S^{-1}(Sx) = x$.

McEliece Cryptosystem - Security

An attack consists of an adversary, who knows the public key $(\hat{G},\ t)$ but not the private key, deducing the plaintext from some intercepted ciphertext y ∈ ${F_2}^n$. Such attempts should be infeasible.

Brute force:

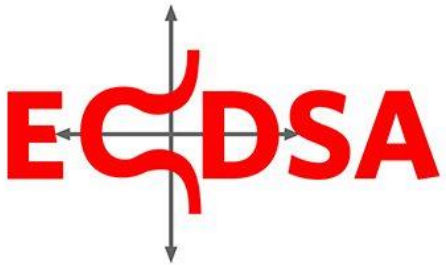
- The attacker knows $\hat{G}$ which is the generator matrix of an (n, k) code $\hat{C}$ which is combinatorially able to correct t errors.
- The attacker may ignore the fact that $\hat{C}$ is the obfuscation of a structured code chosen from a specific family, and instead just use an algorithm for decoding with any linear code.
- Decoding a general linear code, however, is known to be NP-hard, however, and all of the above-mentioned methods have exponential running time.

ECDSA - Background

- Digital Signature is used to simulate the security properties of a handwritten signature.
- It can be used in all electronic communications as we append it to the document and it ensures the document hasn't changed during transmission.
- The Digital signature provides us Authentication, Data integrity, and Non-Repudiation.

ECDSA - Specification

- Digital signature is an asymmetric cryptography.
- Uses elliptic curve cryptography.
- Signature size is approximately 4t.



ECDSA - General Flow

Sign Message:

Parameters:

- PubKey = Public Key
- PrivKey = Private Key
- M = Message

Steps:

1. Digest = Hash (M)
2. Sig = Encrypt (Digest, PrivKey)
3. Send bob M
4. Send bob Sig

Verification Signature:

Parameters:

- s_PubKey = Sender Public Key
- M = Message
- Sig = Message Signature

Steps:

1. Digest = Hash (M)
2. Digest' = Decryption (Sig, s_PubKey)
3. If Digest == Digest' , Signature is valid

ECDSA – Parameters

$CURVE$ = the elliptic curve equation

$y^2 = x^3 + ax + b$

n = prime number $(n > 3)$

G = elliptic curve base point

Hash() = Hash function

Q_A = the public key, point on the curve

d_A = the randomly selected private key

m = the message Alice desired to send

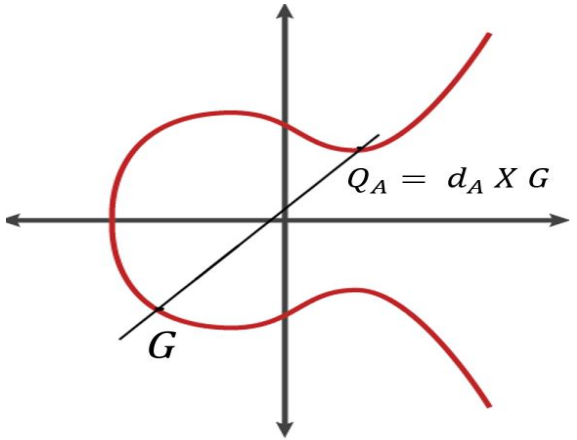
ECDSA - Key Generation

We create a key pair of public and private keys.

- Private key is an integer d_A , it's a randomly selected in range of $[1, n-1]$.
- Public key is a curve point $Q_A = d_A \times G$.

* *Multiplication of curve point by scalar is defined by addition*

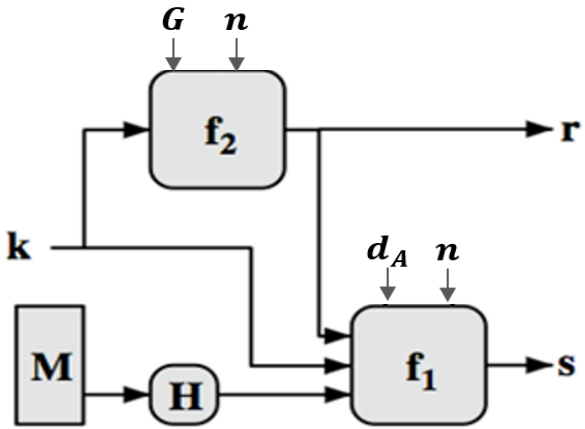
$$Q_A = d_A \times G = \underbrace{G + G + G \ldots + G}_{d_A}$$



ECDSA - Signature

For signing the message m we'll do the next steps :

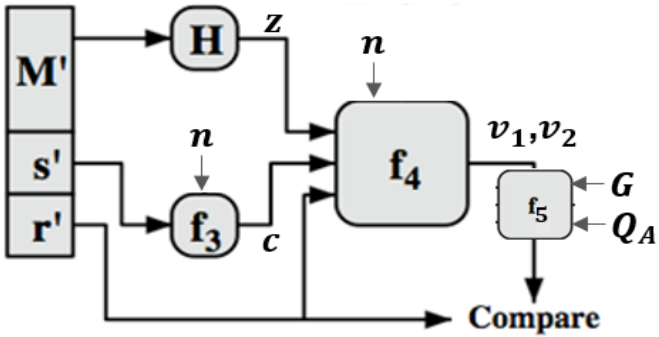
1. Create $e = Hash(m)$.
2. Define z as leftmost bits of e.
3. Generate random number $k\ in\ [1, n-1]$ for each message.
4. Calculate curve point $(x_1, y_1) = k\ x\ G$.
5. Calculate $r = x_1\ (mod\ n)$.
6. Calculate $s = k^{-1}(z + r * d_A)\ (mod\ n)$.
7. If r or s is equal to 0 go back to step 3.
8. Sign the message with the Signature (r, s).



ECDSA - Verification

For the verification of the signature, we'll do the next steps :

1. Create $e = Hash(m)$.
2. Define z as leftmost bits of e.
3. Calculate $c = s^{-1}\ (mod\ n)$.
4. Calculate $v_1 = z * c\ (mod\ n)$.
5. Calculate $v_2 = r * c\ (mod\ n)$.
6. Calculate the curve point $(x_1, y_1) = v_1\ x\ G + v_2\ x\ Q_A$.
7. If $r == x_1\ (mod\ n)$ the signature is valid.



Alice McEliece public key

Alice McEliece private key

Flow

Server McEliece public key

Server McEliece private key

1. Alice and Server generate McEliece key pairs each.

Request

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

1. Alice and Server generate McEliece key pairs each.
2. **Alice and Server generate ECDSA key pairs each.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Request

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. **Alice and Server share their public keys.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. **Alice generates a Rabbit key.**

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Rabbit key

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Rabbit key

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. **Alice encrypts the request using Rabbit Cipher.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Rabbit Key

Request

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. **Alice encrypts the Rabbit key using Server's public McEliece key.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Flow

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Rabbit Key

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. **Alice signs the cipher using her private ECDSA key.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. Alice signs the cipher using her private ECDSA key.
8. **Alice sends Server the cipher, the encrypted key and the signature.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. Alice signs the cipher using her private ECDSA key.
8. Alice sends Server the cipher, the encrypted key and the signature.
9. **Server verifies Alice's signature on the cipher using Alice's public ECDSA key.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key

Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

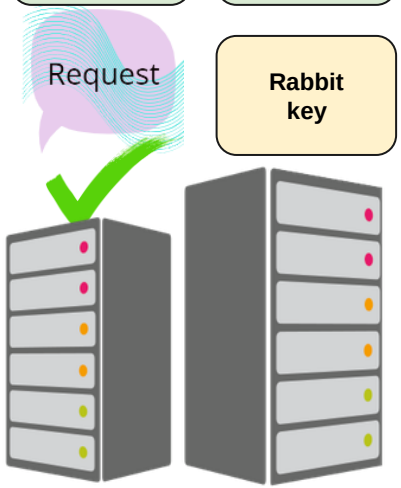
1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. Alice signs the request using her private ECDSA key.
8. Alice sends Server the cipher, the encrypted key and the signature.
9. Server verifies Alice's signature on the cipher using Alice's public ECDSA key.
10. **Server decrypts the Rabbit key using his private McEliece key.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key



Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

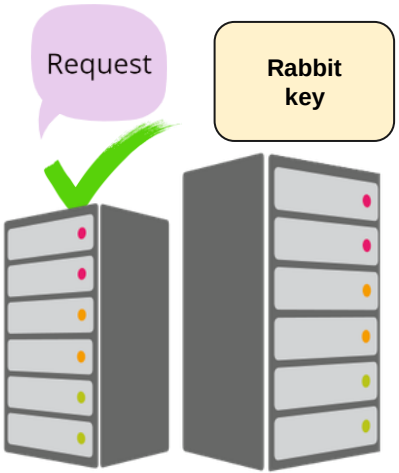
1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. Alice signs the request using her private ECDSA key.
8. Alice sends Server the cipher, the encrypted key and the signature.
9. Server verifies Alice's signature on the cipher using Alice's public ECDSA key.
10. Server decrypts the Rabbit key using his private McEliece key.
11. **Server decrypts the request using the Rabbit key.**

Server McEliece public key

Server McEliece private key

Server ECDSA public key

Server ECDSA private key



Alice McEliece public key

Alice McEliece private key

Alice ECDSA public key

Alice ECDSA private key

Flow

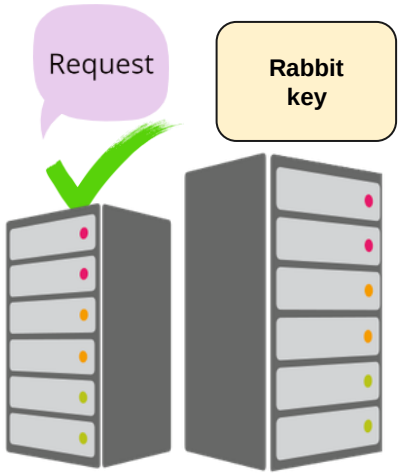
1. Alice and Server generate McEliece key pairs each.
2. Alice and Server generate ECDSA key pairs each.
3. Alice and Server share their public keys.
4. Alice generates a Rabbit key.
5. Alice encrypts the request using Rabbit Cipher.
6. Alice encrypts the Rabbit key using Server's public McEliece key.
7. Alice signs the request using her private ECDSA key.
8. Alice sends Server the cipher, the encrypted key and the signature.
9. Server verifies Alice's signature on the cipher using Alice's public ECDSA key.
10. Server decrypts the Rabbit key using his private McEliece key.
11. Server decrypts the request using the Rabbit key.
12. **Server processes Alice's request.**

Server McEliece public key

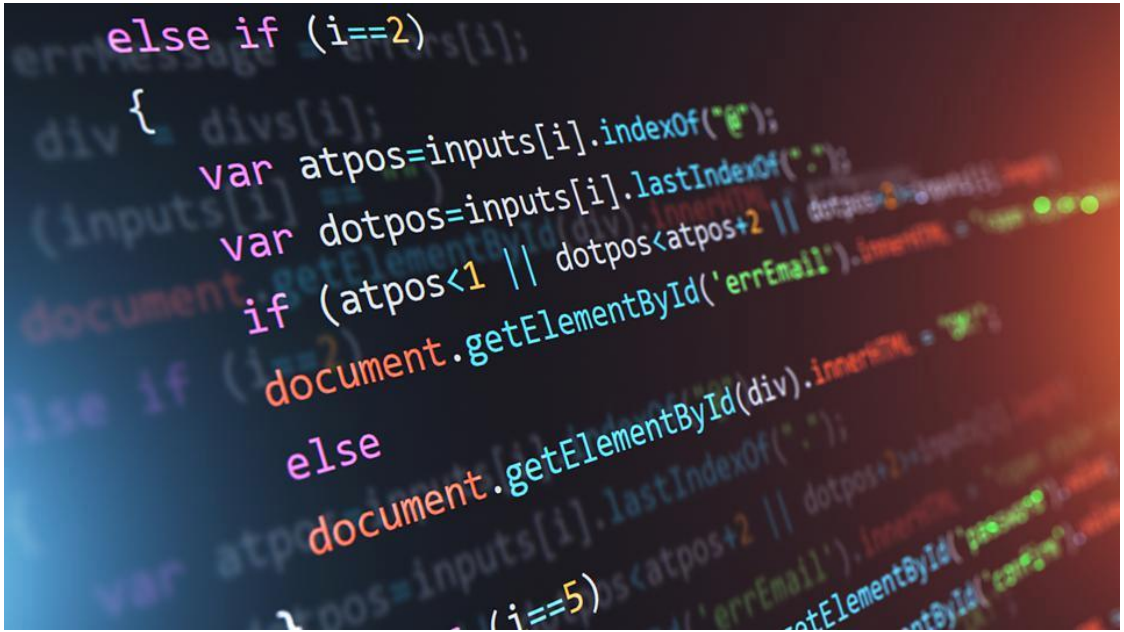
Server McEliece private key

Server ECDSA public key

Server ECDSA private key



Implementation



References

1. Bill Buchanan OBE. Light-weight Crypto: Rabbit.
2. Martin Boesgaard, Mette Vesterager, Thomas Christensen, Erik Zenner. The Stream Cipher Rabbit.
3. Mary Wootters. Lecture 3, Video 3: The McEliece Cryptosystem.
4. Wikipedia. McEliece Cryptosystem.
5. Bill Buchanan OBE. ECDSA, The Nonce and The Private Key.
6. Wikipedia. Elliptic Curve Digital Signature Algorithm.