# Symmetric Key Cryptography Notes

Data Protection
Master in Cyber Security (UPC) 2025

Jorge L. Villar

Last updated: Sep 1 13:22:06 2025

## Contents


**1 Symmetric Key Encryption** **1**
- 1.1 Stream Ciphers . . . 1
  - 1.1.1 Linear Feedback Shift Registers (LFSR) . . . 2
  - 1.1.2 Practical Stream Ciphers . . . 4
    - 1.1.2.1 RC4 (1987) . . . 4
    - 1.1.2.2 ChaCha (2008) . . . 5
  - 1.1.3 Attack Models . . . 6
- 1.2 Block Ciphers . . . 7
  - 1.2.1 Practical Block Ciphers . . . 7
    - 1.2.1.1 Data Encryption Standard (DES) (1977) . . . 7
    - 1.2.1.2 Advanced Encryption Standard (AES) (2001) . . . 8
  - 1.2.2 Modes of Operation . . . 9
    - 1.2.2.1 Electronic Code Book (ECB) Mode . . . 9
    - 1.2.2.2 Cipher Block Chaining (CBC) Mode . . . 10
    - 1.2.2.3 Cipher Feedback (CFB) Mode . . . 10
    - 1.2.2.4 Output Feedback (OFB) Mode . . . 10
    - 1.2.2.5 Counter (CTR) Mode . . . 11
- 1.3 Self-Synchronizing Stream Ciphers . . . 11
- 1.4 Plaintext Padding . . . 11

**2 Message Authentication** **12**
- 2.1 Message Authentication Codes (MAC) . . . 12
  - 2.1.1 (One-Time) Information Theoretic MAC . . . 13
  - 2.1.2 MACs from Block Ciphers . . . 13
    - 2.1.2.1 CBC-MAC (1989) . . . 13
    - 2.1.2.2 One-Key MAC (2003) . . . 14

**3 Hash Functions** **14**
- 3.1 Random Oracles . . . 15
- 3.2 Practical Constructions . . . 15
  - 3.2.1 Merkle-Damgård Construction . . . 15


(Generated by lineprocx v2.97)

3.2.1.1 MD5 (1991) . . . 16
3.2.1.2 SHA-1 (1995) and SHA-2 (2001) . . . 16
3.2.2 Sponge Construction . . . 17
3.2.2.1 SHA-3 (2015) . . . 17
3.3 Applications . . . 17
3.3.1 HMAC (1996) . . . 17
3.3.2 Merkle Trees (1979) . . . 18
3.3.3 Hash Puzzles (1993) . . . 18

**4 Authenticated Encryption** **19**
4.1 Encrypt-then-MAC Generic Construction . . . 19
4.2 Particular Constructions . . . 19
4.2.1 CCM (2003) . . . 19
4.2.2 OCB (2001) . . . 19

# 1 Symmetric Key Encryption

In a symmetric key encryption scheme a long term secret key $k$ is shared between a sender and a recipient. Two procedures called encryption, $E$, and decryption, $D$, are used to transmit a message or plaintext $m$: The sender computes a ciphertext $c = E(k, m)$ and sends it to the recipient by an insecure channel. Then, the recipient recovers the message computing $m = D(k, c)$, assuming that no transmission error occurred.

The shared key must be reasonably short and it must be possible to use it many times to securely transmit several messages of arbitrary length.

From Shannon's results, it is known that no perfect symmetric key encryption scheme exists, unless the length of the key is at least the sum of the lengths of all transmitted messages. Practical schemes try to imitate Vernam's perfect encryption scheme, by expanding a long term key to a **pseudorandom** sequence, which is used to encrypt the message.

$c = (r, F(k, r) \oplus m)$

An auxiliary input $r$ is needed to provide security when different messages are encrypted with the same key $k$. The value of $r$ must be unique in each encryption. It is normally known as **initialization value** or **nonce**, and it is often generated either at random or using a counter.

## 1.1 Stream Ciphers

`https://en.wikipedia.org/wiki/Stream_cipher` (Stream ciphers at Wikipedia)

Stream ciphers are intended to securely send a stream of information, modelled as a sequence of symbols (e.g., bits, bytes or larger words). A plaintext stream is encrypted symbol by symbol to produce a ciphertext stream that is sequentially sent by an insecure communication channel. Later, on reception, the inverse operation is performed to recover the original message stream. The encryption scheme must be capable to encrypt/decrypt the stream on-line, that is, it does not need to know the entire message/ciphertext to start the encryption/decryption procedure.

Most stream ciphers imitate Vernam's cipher: a key stream is combined (XORed) with the message stream to produce the cipher stream. The key stream is generated from a long term key $k$, both in encryption and in decryption, based on an internal state st and a transition function $F$. They are called **synchronous** if the key stream is generated independently of the message and cipher streams.

$(r_n, st_n) = F(st_{n-1}, k)$
$c_n = m_n \oplus r_n$

The initial state, $st_0$, is necessary for decryption, and does not need to be protected. But it must be different between different encryption operations with the same long term key. Otherwise, two message streams $m$, $m'$ will be encrypted with identical key streams and then $m_n \oplus m'_n = c_n \oplus c'_n$ will be revealed to the attacker.

Isolated transmission errors do not propagate, but loss of synchronization causes a permanent decryption failure.

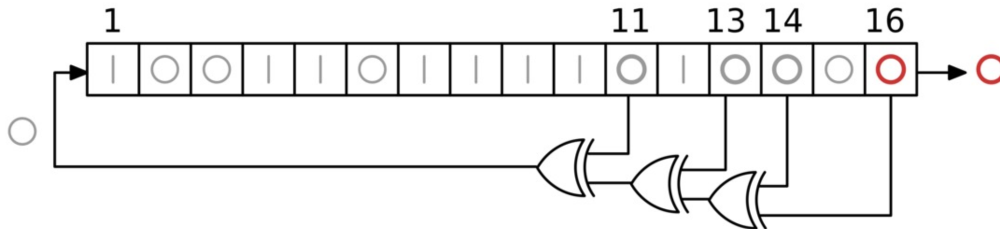
Security is related to the **unpredictability** (a.k.a. pseudorandomness) of the key stream when the long term key is unknown.

### 1.1.1 Linear Feedback Shift Registers (LFSR)

`https://en.wikipedia.org/wiki/Linear-feedback_shift_register` (LFSR at Wikipedia)

A simple (insecure) example of a key stream is a linear feedback shift register (LFSR). A LFSR is a sequence of m cells and a linear feedback function that pushes a new bit into the left cell, and the bits stored in the cells are shifted one position to the right. The output bit of the LFSR is the content of the rightmost cell.

**Example 1**




*Based on an image by KCAuXy4p - Own work, CC0, https://commons.wikimedia.org/w/index.php?curid=60225898*

$X_{1,n+1} = X_{11,n} \oplus X_{13,n} \oplus X_{14,n} \oplus X_{16,n}$
$X_{k,n+1} = X_{k-1,n}$, *for* $k = 2$ *to* $16$

*The output sequence is* $Y_n = X_{16,n}$, *for* $n \geq 0$.

*Then, the output sequence fulfils the equation*

$Y_{n+16} = Y_{n+5} \oplus Y_{n+3} \oplus Y_{n+2} \oplus Y_n$, *for* $n \geq 0$,

*or equivalently*

$Y_{n+16} \oplus Y_{n+5} \oplus Y_{n+3} \oplus Y_{n+2} \oplus Y_n = 0$

*Output sequence example (from initial state* $X_{1..16,0} = 1001101111010000$*):*

```
     state          next bit
1001101111010000 →  0⊕0⊕0⊕0=0
0100110111101000 →  1⊕1⊕0⊕0=0
0010011011110100 →  1⊕0⊕1⊕0=0
0001001101111010 →  1⊕1⊕0⊕0=0
0000100110111101 →  1⊕1⊕1⊕1=0
```

*0000010011011110* → *0⊕1⊕1⊕0=0*
*0000001001101111* → *1⊕1⊕1⊕1=0*
*0000000100110111* → *1⊕0⊕1⊕1=1*
*1000000010011011* → *0⊕1⊕0⊕1=0*
*0100000001001101* → *0⊕1⊕1⊕1=1*
*1010000000100110* → *1⊕0⊕1⊕0=0*
*0101000000010011* → *0⊕0⊕0⊕1=1*
*1010100000001001* → *0⊕1⊕0⊕1=0*
*0101010000000100* → *0⊕0⊕1⊕0=1*
*1010101000000010* → *0⊕0⊕0⊕0=0*
*0101010100000001* → *0⊕0⊕0⊕1=1*
*1010101010000000* → *0⊕0⊕0⊕0=0*
*0101010101000000* → *0⊕0⊕0⊕0=0*
*0010101010100000* → *1⊕0⊕0⊕0=1*
*1001010101010000* → *0⊕0⊕0⊕0=0*
*0100101010101000* → *1⊕1⊕0⊕0=0*
*0010010101010100* → *0⊕0⊕1⊕0=1*
*1001001010101010* → *1⊕1⊕0⊕0=0*
*0100100101010101* → *0⊕0⊕1⊕1=0*
*0010010010101010* → *1⊕1⊕0⊕0=0*

*Produced pseudorandom sequence: 000010111101100100000001010101010010010 00...*

The properties of a LFSR can be stated algebraically, based on its characteristic polynomial. In the example:
$P(z) = 1 + z^{11} + z^{13} + z^{14} + z^{16} \in F_2[z]$.
(We always assume that $deg(P(z))$ equals the number of cells.)

The infinite LFSR output sequence can be seen as a formal series:

$Y(z) = Y_0 + Y_1 z + Y_2 z^2 + Y_3 z^3 + ... \in F_2[[z]]$.

Then, $deg(Y(z)P(z)) < m$.

Indeed, in the example, the coefficient of $z^{n+16}$ in the product is just $Y_{n+16} \oplus Y_{n+5} \oplus Y_{n+3} \oplus Y_{n+2} \oplus Y_n = 0$ for $n \geq 0$.

**Proposition 1** *The output sequence of a LFSR with m cells is ultimately periodic with a period less than* $2^m$.

*Proof.* The output sequence depends deterministically on the initial internal state, and there are $2^m$ many different states. Since the zero state is steady (it produces an infinite sequence of zeros), the maximal nontrivial sequence can involve at most the remaining $2^m - 1$ states.

**Proposition 2** *If* $P(z)$ *is a primitive polynomial of* $F_{2^m}$ *then the output sequence length is maximal for every nonzero initial state of the LFSR.*

*Proof.* The state update function is a linear map and its corresponding matrix is a companion matrix $A$. The characteristic polynomial of $A$ is $P(z)$, and it is also the minimal polynomial of $A$. The statement comes from the fact that $P(z)$ has a common root with the polynomial $z^r - 1$ if and only if there exists a nonzero internal state $X = (X_1, ..., X_m)$ that produces a periodic sequence with a period dividing $r$. Indeed, all roots of a primitive polynomial have order $2^m - 1$, and then the minimum possible period of a nonzero sequence is $2^m - 1$.

The LFSR description can be efficiently obtained from a portion of the output sequence:

**Proposition 3** *Given any section of $2m$ consecutive bits of the output sequence of a LSFR with $m$ cells, the initial internal state and the characteristic polynomial of an equivalent LFSR with at most $m$ cells can be efficiently computed, where 'equivalent' means generating the same output sequence.*

*Proof.* Given any $2m$ consecutive bits of the output sequence $Y_r, ..., Y_{r+2m-1}$, the characteristic polynomial of a LFSR generating the given sequence is the minimal polynomial $P(z)$ such that the product
$P(z)(Y_r + Y_{r+1}z + Y_{r+2}z^2 + ... + Y_{r+2m-1}z^{2m-1})$
has no monomials of degrees $m, m+1, ..., 2m-1$. Clearly, the degree of $P(z)$, $m'$, is at most $m$. Once $P(z)$ is known, one can set the LFSR internal state to $X_{1..m',r} = (Y_{r+m'-1}, Y_{r+m'-2}, ..., Y_r)$ and use the LFSR equations to recover all the output sequence.

The stream cipher is then vulnerable to attacks where the adversary has access to fragments of both the message and cipher streams.

**Definition 1** *The linear complexity of a binary sequence is the minimal $m$ such that there exists a LFSR with $m$ cells and an initial state that generates the given sequence.*

There exist sequences that cannot be generated by any LFSR. The linear complexity of a such sequences is defined to be infinite. In particular, a linear complexity of m implies that the sequence cannot have $m$ consecutive zeros, unless the total number of ones in the sequence is finite. From this fact it is straightforward building sequences of infinite linear complexity.

### 1.1.2 Practical Stream Ciphers

Nonlinear generators are used to improve the stream cipher security.

#### 1.1.2.1 RC4 (1987)

`https://en.wikipedia.org/wiki/RC4` (RC4 at Wikipedia)
Now considered insecure, e.g. WEP was attacked in 2001:
`http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.24.2652` (Weaknesses in the Key Scheduling Algorithm of RC4)

`https://tools.ietf.org/html/rfc6229` (Test Vectors for the Stream Cipher RC4)

The key stream in RC4 is an example of a (byte based) key stream generated with a simple finite state machine, as explained above:

$st_0 = F_0(k)$
$(r_n, st_n) = F(st_{n-1})$

where $F_0$ is a initialization function that expands the key, and $F$ is a transition function that does not depend on the key.

The key stream is computed in blocks of 8 bits, from a long term key of tipically 128 bits. The long term key is initially expanded to an internal state of 2048 bits, representing a map $S$ from 256 elements to 256 elements. Each key stream block is generated by slightly modifying the map $S$ and using the image one of the 256 elements.

**Key scheduling:**

```
for i from 0 to 255
    S[i] = i
endfor
j = 0
```

```
for i from 0 to 255
    j = (j + S[i] + key[i mod keylength]) mod 256
    swap values of S[i] and S[j]
endfor
```

**Key stream generation:**

```
i = 0
j = 0
while GeneratingOutput:
    i = (i + 1) mod 256
    j = (j + S[i]) mod 256
    swap values of S[i] and S[j]
    K = S[(S[i] + S[j]) mod 256]
    output K
endwhile
```

There is no specification of how to add a nonce to the encryption. Thus, the long term key could be used several times. In poor implementations, a nonce is appended to the long term key, and then the initial part of the key stream is predictable once an attacker learns some pairs of plaintext/ciphertext.

#### 1.1.2.2 ChaCha (2008)

`https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant` (ChaCha at Wikipedia)
(widely used nowadays, e.g. Google TLS, OpenSSH, Linux /dev/urandom, ...)

The design of ChaCha key stream notably differs from RC4, in particular, because there is no finite state machine and every key stream block can be computed independently of the others. A nonce $u$ (unique for each encrypted message) and a counter $i$ (unique for every block in the message stream) are used as inputs, as well as the long term key $k$, to compute the key block $k_i$:

$k_i = F(k, u, i)$

Actually, the key stream is computed in blocks of 512 bits, from a long term key of 256 bits. Each block is initialized with a 128-bit constant, the long term key, a 64-bit sequential block counter and a 64-bit nonce. Then, the block contents is shuffled with function chacha_block().

```
define ROTL(a,b) (a << b) | (a >> (32 - b))
define QR(a,b,c,d) (
    a += b, d ^= a, d = ROTL(d,16),
    c += d, b ^= c, b = ROTL(b,12),
    a += b, d ^= a, d = ROTL(d,8),
    c += d, b ^= c, b = ROTL(b,7)
)
define ROUNDS 20

init_block(uint32 b[16], uint32 key[8], uint32 counter[2], uint32 nonce[2])
    // fill the 512-bit block (16 x 32-bit words)
    memcpy(b,"expand 32-byte k",16); // first 4 words
    memcpy(b+4,key,32);              // next 8 words
    memcpy(b+12,counter,8);          // next 2 words
    memcpy(b+14,nonce,8);            // last 2 words
```

```
// The 16 32-bit words in a block form a 4x4 matrix numbered as follows:
//    0   1   2   3
//    4   5   6   7
//    8   9  10  11
//   12  13  14  15

chacha_block(uint32 out[16], uint32 in[16])
    int i;
    uint32 x[16];
    for (i = 0; i < 16; ++i);
        x[i] = in[i];
    // 10 loops x 2 rounds/loop = 20 rounds
    for (i = 0; i < ROUNDS; i += 2)
        // Odd round
        QR(x[ 0],x[ 4],x[ 8],x[12]); // column 0
        QR(x[ 1],x[ 5],x[ 9],x[13]); // column 1
        QR(x[ 2],x[ 6],x[10],x[14]); // column 2
        QR(x[ 3],x[ 7],x[11],x[15]); // column 3
        // Even round
        QR(x[ 0],x[ 5],x[10],x[15]); // diagonal 1 (main diagonal)
        QR(x[ 1],x[ 6],x[11],x[12]); // diagonal 2
        QR(x[ 2],x[ 7],x[ 8],x[13]); // diagonal 3
        QR(x[ 3],x[ 4],x[ 9],x[14]); // diagonal 4

    for (i = 0; i < 16; ++i)
        out[i] = x[i] + in[i];
```

### 1.1.3 Attack Models

The security level achieved by a cryptosystem depends on the type of attacks it can resist.

Defining an attack means giving:

1. The attacker's goal,
2. The attacker's resources.

For instance, in the simplest long term key recovery attack, the attacker can only know the description of the stream cipher and some ciphertexts of unknown plaintexts. The attacker succeeds if it correctly guesses the long term key.

Other attacker goals could be:

- Predicting portions of the key stream,
- Learning some partial information of a plaintext (other than its length),
- Letting the recipient to accept a corrupted message as valid.

In practical scenarios, the attacker can have additional resources, like

- Learning some plaintext/ciphertext pairs, not chosen by itself,
- Learning the ciphertext corresponding to some plaintexts chosen by itself,

- Learning the plaintext corresponding to some ciphertexts chosen by itself,

The attacker can use different information sources, like

- The data observed in a communication channel.
- The timing information in the communication channel.
- The power consumption or the radiation produced by the source or the recipient.

The attacker's behavior can be

- Just eavesdropping the communication channels.
- Modifying the communication by inserting, deleting, replacing or delaying messages.

Limitations on the computational power and attack time are also considered in the attack models.

## 1.2 Block Ciphers

`https://en.wikipedia.org/wiki/Block_cipher` (Block ciphers at Wikipedia)

The length of the plaintexts, ciphertexts and keys are fixed in advance.

Deterministic block ciphers:

$E : K \times M \to C$
$D : K \times C \to M$

Notation:

$E_k(m) = E(k, m)$
$D_k(c) = D(k, c)$

**Correctness:** $\forall k \in K, \forall m \in M, D(k, E(k, m)) = m$

$E_k$ must be an injective map.

A block cipher is a collection of injective maps, $\{E_k\}_{k \in K}$, indexed by the secret key.

Ideal cipher: The set of maps $\{E_k\}_{k \in K}$ is the set of all injective maps from $M$ to $C$.

The ideal cipher is totally impractical: Even in the bijective case, $|K| = |M|!$). But still it is quite useful as an idealized model of a block cipher.

### 1.2.1 Practical Block Ciphers

In practical block ciphers the map $E_k : M \to C$ for a random $k \in K$ must behave like a random injective map.

The typical designs are iterative. In a single iteration (a.k.a. round) part of the key is mixed with the internal state of the encrypting device in a way that all the state bits after the round depend on as many input bits as possible.

- **Confusion:** Every bit in the ciphertext must depend on several bits of the key.
- **Diffusion:** Flipping a single plaintext bit must change half of the ciphertext bits.

Every round uses a different subkey, obtained from the main key in the **key schedule** procedure. The round combines different permutation and substitution operations.

#### 1.2.1.1 Data Encryption Standard (DES) (1977)

`https://en.wikipedia.org/wiki/Data_Encryption_Standard` (DES at Wikipedia)
(now considered insecure, because the short key length)

**Parameters:**
Key length: 56 bits
Round subkey length: 48 bits
Block length (either plaitext or ciphertext): 64 bits
Number of rounds: 16

The iterative procedure is a Feistel Network:

`https://en.wikipedia.org/wiki/Feistel_cipher` (Feistel network at Wikipedia)

$(L_0, R_0) = m$
$(L_{n+1}, R_{n+1}) = (R_n, L_n \oplus F(k_n, R_n))$
$c = (R_{16}, L_{16})$

$k_0, ..., k_{15}$ are the 16 round subkeys produced by in the key schedule procedure.

The same procedure starting with $c$ instead of $m$ and reversing the round subkey sequence recovers $m$.

Description of the function $F()$:

- Expand the half-block with a map $X$, from 32 bits to 48 bits and XOR it with the round subkey.
- Divide into 8 6-bit pieces and replace each of them by a 4-bit word using the corresponding substitution table (a.k.a. S-box) $S_1, ..., S_8$.
- Glue the 8 4-bit words together into a new 32-bit half-block.
- Apply a permutation $P$ to the 32 bits.

Thus, $F(k_n, R_n) = P(S(X(R_n) \oplus k_n))$, where $S(w) = (S_1(w_1), ..., S_8(w_8))$.

A detailed round description would provide the definitions of the maps $X$, $S_1$, ..., $S_8$ and $P$. Also the key schedule procedure uses a permutation network that needs to be described.

`https://en.wikipedia.org/wiki/DES_supplementary_material` (Missing details)

DES is nowadays considered insecure, since the 56-bit key can be guessed by a brute force attack.

**Triple DES** (or 3DES) allows using longer keys, and is still considered secure.

$E_{3\mathrm{DES}}(k_1, k_2, k_3, m) = E_{\mathrm{DES}}(k_3, D_{\mathrm{DES}}(k_2, E_{\mathrm{DES}}(k_1, m)))$

#### 1.2.1.2 Advanced Encryption Standard (AES) (2001)

`https://en.wikipedia.org/wiki/Advanced_Encryption_Standard` (AES at Wikipedia)
(widely used nowadays, e.g. TLS, SSL, disk encryption tools, archive compression tools, Signal (WhatsApp), ...)

**Parameters:**
Key length: 128, 192 and 256 bits
Round subkey length: 128 bits
Block length (either plaitext or ciphertext): 128 bits
Number of rounds: 10, 12 and 14

Description of the typical round:

- The block is arranged as a $4 \times 4$ matrix of bytes.

- In some operations bytes are interpreted as elements in the finite field $F_{2^8}$, implemented by the quotient $F_2[t]/(t^8 + t^4 + t^3 + t + 1)$. Namely, the byte $(x_7, ..., x_1, x_0)$ is mapped to the polynomial $x(t) = x_7t^7 + ... + x_1t + x_0$ and polynomial multiplication is performed modulo the irreducible polynomial $t^8 + t^4 + t^3 + t + 1$.
- **SubBytes:** Each byte $a_{i,j}$ in the block is replaced by another byte $b_{i,j} = S(a_{i,j})$, according to a substitution box $S$. Namely, $S$ is the composition of the inversion map $x \mapsto x^{-1}$ in $F_{2^8}$ (also $0 \mapsto 0$) with a particular bijective $F_2$-affine map.
- **ShiftRows:** The $i$-th row is rotated (cyclically shifted) $i$ positions to the left, thus breaking the column structure of the matrix. That is, $c_{i,j} = b_{i,j-i \bmod 4}$.
- **MixColumns:** A $F_{2^8}$-linear map is applied to the matrix columns. The linear map arises from multiplication in the quotient ring $F_{2^8}[z]/(z^4 + 1)$ by the polynomial $(t + 1)z^3 + z^2 + z + t$. To that end, each column is encoded as a polynomial in $z$ (and $t$) in a way that the byte $c_{i,j}$ is the coefficient of the term $z^i$ and it is interpreted as an element of $F_{2^8}$.
- **AddRoundKey:** The block resulting from the previous steps is XORed with the corresponding round subkey.

An AddRoundKey is also performed before starting the first round, and in the last round the MixColumns step is omitted.

All the steps are invertible, and their inverses are used in AES decryption.

Missing details:

`https://en.wikipedia.org/wiki/Rijndael_key_schedule` (AES key schedule)
`https://en.wikipedia.org/wiki/Rijndael_S-box` (AES S-box description)

### 1.2.2 Modes of Operation

`https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation` (Modes of Operation at Wikipedia)

Block ciphers as they are defined are not very useful in practical applications, because messages are required to have a fixed (short) length, and the encryption function is deterministic. Indeed, an attacker immediately learns whether two ciphertexts produced with the same key contain the same message.

Block ciphers are intended to be used in a specific **mode of operation** that specifies how to deal with messages longer that one message block.

#### 1.2.2.1 Electronic Code Book (ECB) Mode

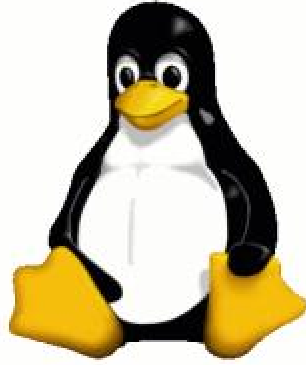
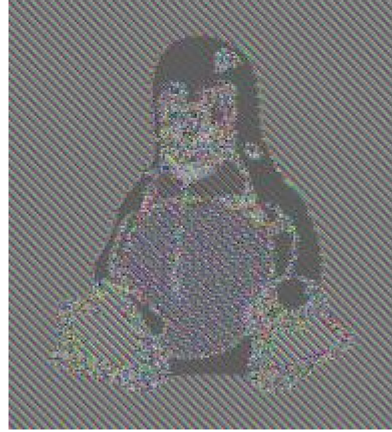
This is the simplest way to encrypt messages of arbitrary length with a block cipher, but it is completely insecure.

The message is divided into blocks, and each block is encrypted separately with the same key:

$m = (m_1, m_2, ..., m_n)$
$c = (c_1, c_2, ...c_n) = (E_k(m_1), E_k(m_2), ..., E_k(m_n))$

The main problem is that the equality of two plaintext blocks in the sequence implied the equality of the corresponding ciphertext blocks. Similarly, if two messages $m$ and $m'$ share a preamble, then the first blocks of their encryptions are also equal.

A classical example is encrypting a digitalized image by splitting it as an array of small squares and encrypting the squares separately. The resulting encrypted image fails to hide the content of the original image.

(Image by Larry Ewing ¡lewing@isc.tamu.edu¿ using GIMP)

#### 1.2.2.2 Cipher Block Chaining (CBC) Mode

This mode of operation uses an initialization value (IV) that allows that two encryptions of the same message with the same key are different. The initialization value must not be reused (so it is a **nonce**).

$m = (m_1, m_2, ..., m_n)$
$c = (c_0, c_1, c_2, ...c_n)$
$c_0 = IV$
$c_i = E_k(c_{i-1} \oplus m_i)$ for $i = 1, ..., n$.

Decryption is performed in a similar way:

$m_i = D_k(c_i) \oplus c_{i-1}$ for $i = 1, ..., n$.

If a ciphertext block $c_i$ has transmission errors, then it would only affect to the decryption of two consecutive message blocks $m_i$ and $m_{i+1}$.

Encryption is not parallelizable, while decryption is.

The blocksize of the block cipher must be large enough to minimize the probability that two ciphertext blocks are equal (a.k.a. collision). Indeed, $c_i = c_j$ implies that $E_k(c_{i-1} \oplus m_i) = E_k(c_{j-1} \oplus m_j)$. Then, $c_{i-1} \oplus m_i = c_{j-1} \oplus m_j$ and also $m_i \oplus m_j = c_{i-1} \oplus c_{j-1}$. Therefore, the attacker learns $m_i \oplus m_j$ since the ciphertext blocks are known to it.

#### 1.2.2.3 Cipher Feedback (CFB) Mode

CFB mode has some similarities with CBC, but it only makes use of the encryption function of the block cipher.

**Encryption:**

$m = (m_1, m_2, ..., m_n)$
$c = (c_0, c_1, c_2, ...c_n)$
$c_0 = IV$
$c_i = E_k(c_{i-1}) \oplus m_i$ for $i = 1, ..., n$.

**Decryption:**

$m_i = E_k(c_{i-1}) \oplus c_i$ for $i = 1, ..., n$.

If two ciphertext blocks collide, $c_i = c_j$, then $c_{i+1} \oplus m_{i+1} = c_{j+1} \oplus m_{j+1}$ and the attacker learns $m_{i+1} \oplus m_{j+1}$.

#### 1.2.2.4 Output Feedback (OFB) Mode

OFB mode works as a synchronous stream cipher. It generates a key stream from the IV and the key and XORs it with the message stream. As in CFB mode, only $E_k()$ function is used.

**Encryption:**

$m = (m_1, m_2, ..., m_n)$
$r = (r_0, r_1, r_2, ...r_n)$
$c = (c_0, c_1, c_2, ...c_n)$
$r_0 = IV$
$r_i = E_k(r_{i-1})$ for $i = 1, ..., n$
$c_0 = r_0$
$c_i = r_i \oplus m_i$ for $i = 1, ..., n$.

**Decryption:**

$r_0 = c_0$
$r_i = E_k(r_{i-1})$ for $i = 1, ..., n$
$m_i = r_i \oplus c_i$ for $i = 1, ..., n$.

Transmission errors do not propagate, if the IV is correctly transmitted. A corrupted IV causes decryption errors in all blocks.

#### 1.2.2.5 Counter (CTR) Mode

In CTR mode, the keystream is generated by encrypting an easy to produce sequence of blocks, that depends on an IV. The usual sequence is defined as $s_i = IV + i$, but XOR or concatenation can safely replace the addition operation.

**Encryption:**

$m = (m_1, m_2, ..., m_n)$
$s = (s_1, s_2, ...s_n)$
$s_i = IV + i$ for $i = 1, ..., n$
$c_0 = IV$
$c_i = E_k(s_i) \oplus m_i$ for $i = 1, ..., n$.

**Decryption:**

$s_i = c_0 + i$ for $i = 1, ..., n$
$m_i = E_k(s_i) \oplus c_i$ for $i = 1, ..., n$.

Encryption and decryption of blocks can be done independently, and therefore, both procedures are fully parallelizable.

As in OFB mode, transmission errors do not propagate, but a wrong IV causes decryption errors in all blocks.

## 1.3 Self-Synchronizing Stream Ciphers

Block ciphers used in OFB or CTR modes are actually synchronous block-oriented stream ciphers, in which the encrypted symbols are not single bits but blocks.

**Self-synchronizing** stream ciphers solve the desynchronization problem. They allow recovering the normal decryption operation after receiving a number of correct cipher stream consecutive symbols aftre the wrong ones. A block cipher operating in CBC or CFB mode can recover from transmission errors after receiving two correct cipher blocks.

## 1.4 Plaintext Padding

In practical applications the plaintext length is not an exact multiple of the block size of a block cipher, and it is necessary to add the missing bits to complete the last block.

Padding must be done in a way that no ambiguity is introduced. Simply appending zeros after the message would introduce some ambiguity unless the last bit in the message is assumed to be one.

A simple padding scheme is appending a 1-bit and zero or more 0-bits. Any message with length an exact multiple of the message block will be padded with an extra block containing 10...0 (also denoted as 10*).

After decryption the last one of the plaintext and the trailing zeros (if any) are discarded.

# 2 Message Authentication

Confidentiality is not the only security property of a communication system that must be guaranteed. An attacker could try to modify the content of a message without being detected, or even to convince the recipient that the corrupted message was created by the sender. **Data integrity** is the property that the recipient of a message can detect (with high probability) that it has been manipulated during transmission.

## 2.1 Message Authentication Codes (MAC)

`https://en.wikipedia.org/wiki/Message_authentication_code` (MACs at Wikipedia)

A commonly used way to add a proof of integrity to a message (either encrypted or not) is appending a **message authentication code** (MAC) to it.

In a MAC scheme a long term secret key $k \in K$ is shared between a sender and a recipient. Two procedures are provided: MAC, used to compute the tag, and Ver, used to verify a given pair message/tag. The sender computes the tag $t = \text{MAC}(k, m) \in T$ for some message $m \in M$ and sends the pair $(m, t)$ to the recipient by an insecure channel. Then, the recipient recovers a pair $(m', t')$, that could be different from $(m, t)$ if the attacker is present, and verifies the integrity of $m'$ by checking whether $\text{Ver}(k, m', t') = 1$.

**Correctness:** $\forall k \in K, \forall m \in M, \text{Ver}(k, m, \text{MAC}(k, m)) = 1$

MAC could be a probabilistic algorithm, meaning that there would be several valid tags for a given message and key. But if MAC is deterministic then $\text{Ver}(k, m', t')$ would simply consist of verifying that $\text{MAC}(k, m') = t'$. In this case, correctness is trivial.

Security of MAC implies at least that, without the key, an attacker is unable to forge a valid pair $(m, t)$. However, as in encryption, different attack scenarios can be defined, depending on the capabilities of the attacker.

In a simple key recovery attack, the attacker can only know the description of the MAC algorithm and some valid pairs $(m, t)$ for messages not generated by itself. The attacker succeeds if it correctly guesses the key.

Other attacker goals could be:

- **Universal forgery:** Forging a valid tag for any possible message,
- **Existential forgery:** Forging a valid tag for a particular message chosen by the attacker, for which no valid tag was previously known.

The attacker resources can be:

- Learning some valid message/tag pairs, for messages **not** chosen by itself,
- Learning some valid message/tag pairs, for messages chosen by itself.

A correct label attached to a message not only gives guarantees that the message has not been modified, but also that the intended sender computed the tag (because he is the only one, other than the recipient, knowing the shared secret key).

Perfect schemes exist if we limit in advance the number of valid message/tag pairs the attacker can get before forging a fresh one.

### 2.1.1 (One-Time) Information Theoretic MAC

$k = (a, b)$ for $a, b \in F_q$
For $m \in F_q$, $\text{MAC}(k, m) = am + b \in F_q$
$\text{Ver}(k, m, t) : am + b = t$?

**Proposition 4** *If the attacker knows at most one valid message/tag pair, say $(m, t)$, even if $m$ has been chosen by itself, then the probability that it outputs a valid forgery $(m', t') \neq (m, t)$ is $1/q$.*

*Proof.* Indeed, valid pairs are points on a (non-vertical) line in the plane. Therefore, guessing another point in the line is essentially guessing its slope $a$.

The MAC is called **information theoretically secure** because the probability $1/q$ is the same as the success probability of a brute-force attack guessing the tag from scratch (i.e., without using at all the description of the scheme).

However, if the attacker knows two different valid pairs $(m_1, t_1)$ and $(m_2, t_2)$, then it can launch a successful key-recovery attack.

Using a random polynomial of degree at most $n$, instead of the degree 1 polynomial used above, the resulting MAC is information theoretically secure if the attacker is allowed to learn at most $n$ valid message/tag pairs.

The main drawback of the scheme is that it requires a key $n$ times as long as the message, and the tag itself has the same length as the message. Practical MACs are far more efficient than this.

### 2.1.2 MACs from Block Ciphers

A block cipher is a natural construction for combining all the bits of a message with a (short) secret key is such a way that the key is not exposed, even if some plaintext/ciphertext pairs are revealed to the attacker. Therefore, some MAC schemes are based on block ciphers operating in chaining modes like CBC.

Since the goal of the MAC is not message recovery but only message integrity, the MAC output can be smaller than the original message (say, only one block of the block cipher).

#### 2.1.2.1 CBC-MAC (1989)

$\text{MAC}(k, m)$ is just the last block of the encryption of m with a block cipher operating in CBC mode and with a zero initial value IV.

$m = (m_1, m_2, ..., m_n)$
$c = (c_0, c_1, c_2, ...c_n)$
$c_0 = 0$
$c_i = E_k(c_{i-1} \oplus m_i)$ for $i = 1, ..., n$,
$\text{MAC}(k, m) = c_n$.

CBC-MAK is proven secure for messages with a fixed length that is an exact multiple of the block length.

**Proposition 5 (informal)** *If there exists an successful attacker $A$ against CBC-MAC for a block cipher $E$, then there exists also a successful attacker $B$ against the block cipher.*

The proof requires a formal description of both attacks models (against CBC-MAC and against $E$).

For CBC-MAC, the attacker $A$ chooses two messages $m_0$ and $m_1$ of the same length and asks for the MAC of one of them. The attacker is successful if it correctly guesses which message has been used to compute the MAC.

For $E$, the attacker $B$ asks for encryption of as many blocks as it wants. Then it either receives the encrypted blocks, or just random elements. The attacker succeeds if it correctly guesses whether it is receiving real encryptions or random elements.

The proof also requires a lower bound in the block size, to make the ciphertext block collision probability small enough.

More details in:

`https://cseweb.ucsd.edu/~mihir/papers/cbc.pdf` (Bellare, Kilian and Rogaway's paper)

**Proposition 6** *CBC-MAC is insecure if applied to messages of arbitrary lengths.*

*Proof.* In the simplest example, consider a message/tag pair, $(m, t)$, given to the attacker, such that the message contains exactly one block. Thus, $t = \mathrm{MAK}(k, m) = E_k(m)$. Now, consider the two-blocks message $m' = (m, m \oplus t)$. We have
$t' = \mathrm{MAK}(k, m') = E_k(E_k(m) \oplus (m \oplus t)) = E_k(t \oplus (m \oplus t)) = E_k(m) = t$.
Therefore, the attacker can forge the new valid pair $(m', t)$.

#### 2.1.2.2 One-Key MAC (2003)

Several modifications of CBC-MAC have been proposed to make it secure for messages of variable length.

In OMAC, the last (possibly incomplete) block in the message is padded if necessary with 10...0, and tweaked with extra keys before performing the last encryption operation in CBC-MAC.

$m = (m_1, m_2, ..., m_n)$
$c = (c_0, c_1, c_2, ...c_n)$
$c_0 = 0$
$c_i = E_k(c_{i-1} \oplus m_i)$ for $i = 1, ..., n-1$,
$c_n = E_k(c_{n-1} \oplus m'_n \oplus k')$
$\mathrm{MAC}(k, m) = c_n$.
where $m'_n$ is the padded block $m_n$, and $k'$ is a subkey obtained from $k$.

Two different values of $k'$ are used depending on whether the last block has been padded or not.

More details in

`https://en.wikipedia.org/wiki/One-key_MAC` (OMAC at Wikipedia)
`https://eprint.iacr.org/2002/180.pdf` (Academic paper at IACR Eprint Archive)

# 3 Hash Functions

A **hash function** is a deterministic and efficiently computable function $H : \{0,1\}^* \to \{0,1\}^m$, that maps binary strings of arbitrary length to binary strings of a fixed length $m$ (or to elements in a finite set), behaving like a random function.

A deterministic function cannot behave exactly as a random function, but at least the following properties are required to any hash function:

**Preimage resistance.** Given a random element $y \in \{0,1\}^m$, it is infeasible to compute $x \in \{0,1\}^*$ such that $H(x) = y$.

**Second preimage resistance.** Given a random element $x \in \{0,1\}^*$, it is infeasible to compute $x' \in \{0,1\}^*$ such that $H(x') = H(x)$ and $x' \neq x$.

**Collision resistance.** It is infeasible to come up with a **collision** pair $x, x' \in \{0,1\}^*$ such that $H(x) = H(x')$ and $x' \neq x$.

In a random function, for any different $x_1, ..., x_n \in \{0,1\}^*$, the images $H(x_1), ..., H(x_n)$ are independent random variables, uniformly distributed on $\{0,1\}^m$. Therefore, one expects to compute $2^m$ hash values to

hit a given $y \in \{0,1\}^m$, and there is no difference between the first and second preimage resistance properties. However, only $2^{m/2}$ hash computations would be needed on average to find a collision (birthday paradox).

Therefore, any cryptographic hash function will use a value of $m$ large enough so that computing $2^{m/2}$ hash values are considered an infeasible task.

## 3.1 Random Oracles

As an heuristic security argument, some cryptosystems built with hash functions are analyzed by replacing the real hash function by a **random oracle**.

A random oracle is nothing more than a random function $H : \{0,1\}^* \to \{0,1\}^m$ defined by the property that for any different $x_1, ..., x_n \in \{0,1\}^*$, the images $H(x_1), ..., H(x_n)$ are independent random variables, uniformly distributed on $\{0,1\}^m$.

All parties involved in the cryptographic protocol have granted access to the random oracle (meaning that for the same query $x$ all of them would obtain the same value $H(x)$).

In security proofs based in the idealized Random Oracle Model (ROM), the random oracle is simulated by means of a table containing the queries and their responses (i.e., storing the known pairs $(x, H(x))$).

Proofs in the ROM are in general not valid in the real world, as meaningful counterexamples are known.

## 3.2 Practical Constructions

Hash functions, like modes of operation of block ciphers, are designed in an iterative way to be able to cope with messages of arbitrary length. To be secure, flipping any message bit must result in flipping about $m/2$ bits of the hash value. Therefore, hash functions use the same tools as block ciphers operating in chaining modes, except for the key schedule part.

### 3.2.1 Merkle-Damgård Construction

This is a generic construction of a hash function $H : \{0,1\}^* \to \{0,1\}^m$ from a compression function $F : \{0,1\}^r \times \{0,1\}^m \to \{0,1\}^m$.

The construction is similar to the CBC mode of operation:

- First, a padding is appended to the message to achieve a length that is an exact multiple of $r$.
- Then, the padded message $m'$ is split into $r$-bit blocks $(m'_1, ..., m'_n)$.
- A fixed initialization value IV is part of the specification of the hash function.
- The $m$-bit long internal state $s$ is updated iteratively:
  $s_0 = IV$,
  $s_i = F(m'_i, s_{i-1})$, for $i = 1, ..., n$.
- The has value is $H(m) = s_n$.

A finalization step $H(m) = G(s_n)$ can be added to the specification of the hash function (e.g., to shorten the output of the hash function).

The padding of the message used in Merkle-Damgård construction must fulfil three conditions:

- $m$ is a prefix of $m'$.
- if two messages $A$ and $B$ have the same length, their padded versions, $A'$ and $B'$, must have the same number of blocks.

- if two messages $A$ and $B$ have different lengths, then the last blocks of their padded versions must be different.

**Proposition 7** *If $F$ is collision resistant then $H$ is also collision resistant.*

*Proof.* Given a collision of $H$, say different messages $A$ and $B$ such that $H(A) = H(B)$, one of the evaluations of $F$ in the computation of $H(A)$ collides with one of the evaluations of $F$ in the computation of $H(B)$. To find it, just start form the last evaluation and go backwards.

The so-called **length padding**, which includes an encoding of the message length, fulfils the previous padding requirements:

$m' = (m, 10...0, d)$, where $d$ is a 64-bit representation of the bit-length of $m$ (only valid for messages shorter than $2^{64}$ bits).

#### 3.2.1.1 MD5 (1991)

MD5 was one of the most widely used Merkle-Damgård based hash functions, until some collisions of the underlying compression function were found.

**Parameters:**
Message block length: $r = 512$ bits
Hash result length: $m = 128$ bits
Padding scheme: length padding, with the actual message bit-length reduced modulo $2^{64}$
Number of rounds: 64

The message block is split into 16 32-bit words and the internal state is divided into 4 32-bit words: $A_0$, $B_0$, $C_0$ and $D_0$.

At round $i$:

- Initialize four words with the previous internal state, $(A, B, C, D) = (A_0, B_0, C_0, D_0)$.
- Add modulo $2^{32}$ to $A$ a nonlinear function of $G_i(B, C, D)$, a message word and a 32-bit constant $K_i$.
- Rotate $A$ some constant $s_i$ bits to the left and $B$ is added to $A$, modulo $2^{32}$.
- Cyclically permute $(A, B, C, D)$ one position to the left.
- Add $(A, B, C, D)$ to $(A_0, B_0, C_0, D_0)$ to obtain the new internal state.

Missing details:

`https://en.wikipedia.org/wiki/MD5` (MD5 at Wikipedia)

#### 3.2.1.2 SHA-1 (1995) and SHA-2 (2001)

SHA-1 uses a similar design but with $m = 160$ (and 5 32-bit words in the internal state). It is nowadays considered insecure, because known attacks can find collisions with feasible resources.

However, SHA1 is still useful for applications depending only on its preimage resistance.

SHA-2 has some variants: SHA-256 with $m = 256$ (8 32-bit words), and SHA-512 with $m = 512$ (8 64-bit words), with a similar structure as SHA-1, but with more complex round functions.

Detailed decription:

`https://en.wikipedia.org/wiki/SHA-1` (SHA-1 at Wikipedia)
`https://en.wikipedia.org/wiki/SHA-2` (SHA-2 at Wikipedia)

#### 3.2.2 Sponge Construction

In the Sponge construction an internal state register sequentially accumulates information about each block of the message. Once the message is exhausted, the final hash value is extracted from the internal state in an iterative way.

The internal state is stored into an accumulator, that is divided into two parts: $(R, S)$. Only the R part interacts directly with the message and the final hash value, while a "stirring" function $F$ is applied to the whole accumulator $(R, S)$ in each iteration, so that the information flows between $R$ and $S$.

$m' = \mathrm{Pad}(m)$,
$m' = (m'_1, m'_2, ..., m'_{n'})$
$(R_0, S_0) = IV$,
$(R_i, S_i) = F(R_{i-1} \oplus m_i, S_{i-1})$, for $i = 1, ..., n'$,
$(R_i, S_i) = F(R_{i-1}, S_{i-1})$, for $i = n' + 1, ..., n' + u$,
$H(m) = (R_{n'+1}, ..., R_{n'+u})$.

In the last $u$ iterations the hash value is extracted block by block from the $R$ part of the accumulator.

##### 3.2.2.1 SHA-3 (2015)

SHA-3 is a sponge construction, in which $IV = 0$ and the internal state $(R, S)$ is 1600 bit long.

There are different SHA-3 versions depending on the size of the final hash value. For instance, SHA3-512 uses a hash value of 512 bits. The size of the $S$ part is twice as long as the final hash value. Thus, the $R$ part has $1600 - 2 \times 512 = 576$ bits. As a consequence, only one final iteration (that is, $u = 1$) is needed to extract the hash value. The remaining $576 - 512 = 64$ bits of $R_{n'+1}$ are discarded.

The 1600-bit internal state is organized into a $5 \times 5$ array of 64-bit words, and the stirring function $F$ consists of 24 rounds. Each round consists of:

- XOR (bitwise) each matrix entry with the XOR of all entries in the two adjacent columns.
- Apply a different (but fixed) bit-rotation to each of the 25 words.
- Apply a specific permutation of the 25 words.
- XOR each row with a nonlinear function of its two next rows.
- XOR one of the 25 words (depending on the round) with the output of an auxiliary LFSR.

Missing details:

`https://en.wikipedia.org/wiki/SHA-3` (SHA-3 at Wikipedia)

### 3.3 Applications

Hash functions are commonly applied to any context that requires dealing with objects of arbitrary length, where preimage or collision resistance is necessary.

Message authentication codes can be constructed from hash functions. Also, standard hash functions can be adapted to "structured messages" like trees.

#### 3.3.1 HMAC (1996)

HMAC is a generic construction of a message authentication code from any hash function. Given a long term secret key $k$ and a hash function $H$, the MAC of a message $m$ is computed as follows:

$\mathrm{MAC}(k,m) = H(k_{\text{out}}, H(k_{\text{in}}, m))$,
where the inner and outer auxiliary keys $k_{\text{in}}$ and $k_{\text{out}}$ are derived from $k$:

If $k$ is longer than the block size of $H$, then replace it by $H(k)$.
If $k$ is shorter that the block size of $H$, then pad it with trailing zeros.
Compute $k_{\text{in}} = k \oplus P_{\text{in}}$ and $k_{\text{out}} = k \oplus P_{\text{out}}$, where $P_{\text{in}}$ and $P_{\text{out}}$ are constants.

Missing details:

`https://en.wikipedia.org/wiki/HMAC` (HMAC at Wikipedia)

The two-round design of HMAC fixes the "length-extension attacks" suffered by some existing practical hash functions (mainly, those based on the Merkle-Damgård construction).

In Merkle-Damgård construction $H(m)$ is obtained as the last output of the compression function, computed on the block sequence $m' = (m'_1, ..., m'_{n'})$ of the padded message. Thus, the value of $H(m)$ could be used to continue the hash evaluation of additional blocks appended to $m'$, even if some parts of $m'$ are not known. As a consequence, given a message $m$ and the value $H(k,m)$ for unknown $k$, an attacker can compute $H(k,m,x)$ for certain values of $x$ (namely, $m'$ is a prefix of $(m,x)$).

Considering that SHA-3 seems to be immune to such attacks, HMAC-SHA3 needs only one round:

$\text{HMAC-SHA3}(k,m) = \text{SHA3}(k,m)$.

### 3.3.2 Merkle Trees (1979)

A Merkle Tree is a tree of hash values that allows to authenticate a collection of objects with a single hash value (the root of the tree), and has only logarithmic complexity in the verification that a given object belongs to the collection.

The objects in the collection are arranged as the leaves of a binary tree. Each node in the tree is associated to a binary string, indicating the path from the root to the node ("0" = left, "1" = right). The hash values of the nodes are computed as follows:

$h_\emptyset = H(1, h_0, h_1)$, for the root node
$h_x = H(1, h_{x0}, h_{x1})$, for any intermediate node $x$, if $h_{x1}$ exists.
$h_x = H(1, h_{x0})$, otherwise.
$h_x = H(0, m_x)$, for any leaf,
where $m_x$ is the object at leaf node $x$.

The verification that an object $m_x$ is at node $x$ of a tree with root hash $h_\emptyset$, only requires to check the hash equations along the path to $x$. Therefore, a proof of the above fact consists of the hashes $h_y$ for all nodes $y$ that are siblings of the nodes of the path to $x$.

For instance, a proof that $m$ belongs to the collection with root hash $h_\emptyset$ and it is at node $x = 10010$, consists of the hashes $(h_0, h_{11}, h_{101}, h_{1000}, h_{10011})$. The proof verification consists of checking the system of equations:

$h_{10010} = H(0, m)$
$h_{1001} = H(1, h_{10010}, h_{10011})$
$h_{100} = H(1, h_{1000}, h_{1001})$
$h_{10} = H(1, h_{100}, h_{101})$
$h_1 = H(1, h_{10}, h_{11})$
$h_\emptyset = H(1, h_0, h_1)$

The length of the proof and the number of hash evaluations performed in the verification depend logarithmically on the size of the collection.

### 3.3.3 Hash Puzzles (1993)

The preimage resistance property of a hash function can be used to define families of computational problems parameterized by the computational cost they require. This cost ranges from the very easy problems to the

infeasible ones.

Providing the solution to a given problem (or puzzle) can be seen as a "proof of work" done to get it. The simplest example is the following problem:

Given a difficulty parameter $t$, find $m$ such that $H(m)$ has (at least) $t$ trailing zeros.

Assuming that $H$ behaves as a random function, the expected number of trials (hash evaluations) to get a valid solution is $2^t$, which is nowadays considered as an infeasible task if, for instance, $t = 128$.

The problem can be modified to require some particular information in the prefix of $m$, like a timestamp or the hash of the solution to a previously posed puzzle.

The first hash-based puzzles were proposed in 1993 as a countermeasure against e-mail spam, and they are used nowadays mainly in cryptocurrencies like Bitcoin.

# 4 Authenticated Encryption

Message authentication provides data integrity and origin authentication, independently of any confidentiality guarantee. Therefore, most cryptographic applications use a combination of symmetric encryption and message authentication.

Even if encryption and message authentication can be secure when used alone, a careless combination of the two protocols can enable dramatic attacks. For instance, using the same secret key for a block cipher operating in CBC mode and a CBC-MAC enables an easy MAC forgery attack.

On the other hand, the similarity between the computations performed during a CBC-mode encryption and a CBC-MAC suggests that both things could be computed together in a faster and compact way. The schemes providing the two functionalities at once are called **authenticated encryption** schemes.

## 4.1 Encrypt-then-MAC Generic Construction

There are different approaches to combine two independent schemes (one encryption scheme and one message authentication code) in a secure way. The most reliable one is first encrypting and then computing the authentication tag on the ciphertext (including the IV, if any), with an independent key.

$c = E_{k_1}(m)$,
$t = \text{MAC}(k_2, c)$.

## 4.2 Particular Constructions

In this section two particular constructions using a single key for both encryption and authentication are described.

### 4.2.1 CCM (2003)

CCM is a specific construction that combines the CTR mode for encryption and the CBC-MAC for authentication, using the same key k. The scheme requires two block cipher encryption calls per message block.

The authentication tag $t = \text{CBC-MAC}(k, m)$ is computed first. Then $(m, t)$ is encrypted in CTR mode. The initialization value IV for the CBC-MAC must never be used as the counter value in the CTR mode.

### 4.2.2 OCB (2001)

OCB is an authenticated encryption scheme optimized for hardware and software implementations. It uses about one block cipher encryption call per message block, and the ciphertext length is minimal, since no

bytes are added because of padding.

- Given a key $k$, a table of shift constants $(L_*, L_\$, L_1, ..., L_u)$ is precomputed.
- The message block $m_i$ is encrypted as $c_i = E_k(m_i \oplus \Delta_i) \oplus \Delta_i$, where $\Delta_i$ is computed from the shift constants in the table.
- If the last message block is incomplete, it is encrypted as $c_* = E_k(\Delta_*) \oplus m_*$, truncated to the length of $m_*$.
- The authentication tag is computed from the XOR of all message blocks: $t = E_k(m_1 \oplus ... \oplus m_* \oplus \Delta_\$)$, truncated to the desired length of the tag.

The scheme can include some unencrypted but authenticated extra information, which is hashed (using the same block cipher and key and some $\Delta$ constants) and XORed with the tag.

More information:

`http://web.cs.ucdavis.edu/~rogaway/ocb/ocb-faq.htm#describe-ocb` (OCB FAQs at author's page.)

---