Chapter 1

Secret-key techniques

We now study the case of symmetric crypto. In symmetric crypto, Ali and Bachar have one key each, and this key is private. This is very important to hold into account. In asymmetric crypto, each one has a public key, a private key, must know the public key of his collegues, etc. But here, none of that : an encryption scheme works under a private key. This is why this is the simplest case to study and the first one we study.

We are thus interested here in building a secure encryption scheme, that gets as input a message in plaintext and outputs a ciphertext, as well as a decryption algorithm that allows us to retrieve the message. The definitions of IND-CPA was made very clear : if the system is secure, an adversary that does not know the key has better chance of going random instead of trying to guess !

There are many functions needed to build to *in fine* build a secure symmetric encryption scheme. There is the mechanism of the key, and some more functions such as block ciphers, permutations, etc. All these are useless separately, but at the end they build an encryption scheme. In this course, we will cover :

- Keystream generators and stream ciphers
- Block ciphers
- Permutations

1.1 Basic schema

The basic schema of the communication is the following : Ali wants to send a message to Bachar. Then :

- One key is generated and given to Ali, and to Bachar, under a **secure channel**.
- Ali encrypts the message with the key k,
- transmits it to Bachar in a public channel, in clear : any outsider Omar[1] can observe the thing sent.
- Bachar receives the ciphertext, and decrypts it with the same key k.

[1] Instead of Oscar.

1.2 Stream ciphers vs block ciphers

There are two ways of doing symmetric crypto : with stream ciphers of with block ciphers.

- Stream ciphers process a message bit by bit, XORing every bit with a key stream s_i built from the original key by a keystream generator.
- Block ciphers process a message by separating it into blocks, and processing block by block. Within a block, each bit value has an impact on the output for the block.

1.3 Keystream generators and stream ciphers

Stream ciphers are simple : they need a key as long as the plain text and they XOR it with the message. For this, we take a private key k that is not necessarly very long, and we extend it, building a key stream $s = (s_i)$. This key stream generation process is the main focus for stream ciphers.

The encryption and decryption algorithm are as follows, with the generated key stream $s = (s_i)$. We notice that the two algorithms are the same. It indeed works.

Keystream generators and stream ciphers

What is a stream cipher?

Keystream generator: $G : K \times \mathbb{N} \to \mathbb{Z}_2^\infty$

- inputs a secret key $k \in K$ and a diversifier $d \in \mathbb{N}$
- outputs a long (potentially infinite) keystream $(s_i) \in \mathbb{Z}_2^\infty$

To encrypt plaintext $(m_0, \ldots, m_{|m|-1})$:

$$c_i = m_i + s_i \quad (\text{in } \mathbb{Z}_2)$$

To decrypt ciphertext $(c_0, \ldots, c_{|c|-1})$:

$$m_i = c_i + s_i \quad (\text{in } \mathbb{Z}_2)$$

The diversifier d is public and must be a **nonce**, i.e., unique per encryption for a given key.

6 / 69

1.4 Block ciphers

A block cipher is basically a mapping from :

- Input : secret key $k \in \mathbb{Z}_2^m$, and **input block** $x \in \mathbb{Z}_2^n$
- Output block $y = E_k(x) \in \mathbb{Z}_2^n$.

And we ensure that for each key k, the function has an inverse : $x = E_k^{-1}(y)$.

Note that as it is, we can not use it as encryption scheme, it is completely deterministic so it loses the IND-CPA game.

1.4.1 DES

DES is a block cipher : it takes a block as input, a key, and generates an output block. For a classical DES algorithm, the input block has 64 bits and the key is 56-bits long. Actually, 64 bits come as input for the key, but every $7th$ bit is used as a parity bit, not a key bit. DES is a 56-bit cipher.

$$x \in \mathbb{Z}_2^{64} \qquad k \in \mathbb{Z}_2^{56}$$

With the input block x and the key k, DES works by computing 16 rounds of its underlying network. For each round, it needs a key, that is derived from the original key. Those are subkeys K_i.

$$k \Rightarrow K_1, K_2, \ldots K_{16} \in \mathbb{Z}_2^{48}$$

The section below explains how the subkeys (also called *round keys* are generated).

Key schedule : round keys generation

Input : the 56-bits key. It is all based on permutations, and more precisely on two permutations : PC-1 and PC-2. Here is how 16 48-bits keys are generated.

1. The initial key goes under a bit transposition, through the table of permutation of PC-1 (see figure below). So at the exit of PC-1, we still have a 56-bit key. We separate it into two 28-bits blocks : C_0 and D_0.

$$\text{PC1} \ : \ \mathbb{Z}_2^{56} \rightarrow \mathbb{Z}_2^{56}$$

2. For i going from 1 to 16 :

   - We build C_i from C_{i-1}, D_i from D_{i-1}, thanks to a permutation LS_i
   - LS_i is a circular shift to the left : by one when $i = 1, 2, 9, 16$, by two positions otherwise
   - K_i is obtained by the concatenation $C_i \| D_i$ but again under a permutation. This time, it is PC-2 that handles it, and ignore some bits to make it fit in 48 bits :

$$\text{PC2} \ : \ \mathbb{Z}_2^{56} \rightarrow \mathbb{Z}_2^{48}$$

Block ciphers DES

DES: Tables for PC1 and PC2

PC1						
57	49	41	33	25	17	9
1	58	50	42	34	26	18
10	2	59	51	43	35	27
19	11	3	60	52	44	36
above for C_i; below for D_i						
63	55	47	39	31	23	15
7	62	54	46	38	30	22
14	6	61	53	45	37	29
21	13	5	28	20	12	4

PC2					
14	17	11	24	1	5
3	28	15	6	21	10
23	19	12	4	26	8
16	7	27	20	13	2
41	52	31	37	47	55
30	40	51	45	33	48
44	49	39	56	34	53
46	42	50	36	29	32

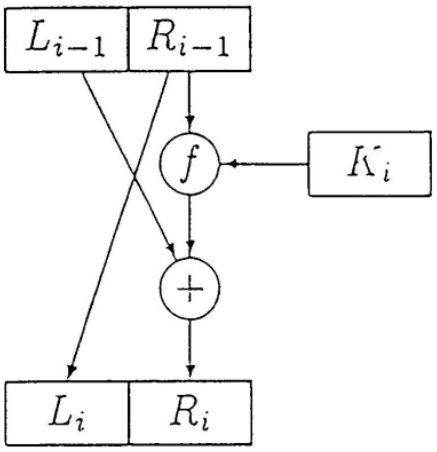
Table 7.4: DES key schedule bit selections (PC1 and PC2).

20 / 69

DES's f function

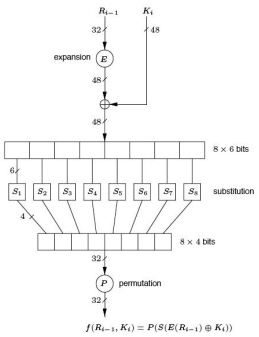
At each round, the current state is processed, divided into two blocks, L_{i-1} and R_{i-1}, using K_i and a function called f. DES's f function works as follows : it takes a 32-bit input, which is the half-message block, and the subkey.

$$f : \mathbb{Z}_2^{32} \times \mathbb{Z}_2^{48} \to \mathbb{Z}_2^{32}$$



Block ciphers DES

Inside DES' f function



$f : \mathbb{Z}_2^{32} \times \mathbb{Z}_2^{48} \to \mathbb{Z}_2^{32}$

1. Expand R through E from 32 to $48 = 16 + 2 \times 16$ bits
2. Compute $E(R) \oplus K_i$
3. Cut in 6-bit blocs B_j
4. $C_j = S_j(B_j)$, 8 different S-Boxes $S_j : \mathbb{Z}_2^6 \to \mathbb{Z}_2^4$
5. Bit transposition $P : \mathbb{Z}_2^{32} \to \mathbb{Z}_2^{32}$

Figure 7.10: DES inner function f.

16 / 69

The $S-$boxes are what make DES non linear, crucial for cryptanalysis.

Non-ideal properties

- There are some weak keys (4) for which DES is an involution
- There are some weak pairs (6) of keys for which DES is an involution
- Complementarity property : $E_k(x) = y \iff E_{\bar{k}}(\bar{x}) = \bar{y}$ reduces the Exhaustive search by 2 (one bit).

Flaws of DES

- Size of the key space : 56 bits is very small... Exhaustive search can be done in 2^{55} operations, which is nowadays feasible.
- Susceptible to mathematic cryptanalysis : differential, linear.

Triple-DES

An improvement is then to take larger keys, for example three keys and building Triple-DES with 168 bits ($= 3 \times 56$), with two keys, 112 bits. Note that 3DES is retro-compatible with single DES, we just have to huse thrice the same key.

$$3\text{DES}_{k_1\|k_2\|k_3} = \text{DES}_{k_3} \circ \text{DES}_{k_2}^{-1} \circ \text{DES}_{k_1}$$

$$3\text{DES}_{k_1\|k_2} = \text{DES}_{k_1} \circ \text{DES}_{k_2}^{-1} \circ \text{DES}_{k_1}$$

Why not simple Double-DES ? Because with two rounds of DES, one can make two times a brute force, one for each system, and breaking the system, in a time of 2^{57} and using memory 2^{56} : storing every attempt on the first system (intercepting information before it comes to the second DES layer), then trying to decrypt the second DES with the saved attempts.

1.4.2 Rijndael and AES

With such flaws for DES, it stinks. Rijndael comes to help. We talk of Rijndael as AES, you should know it, and if you don't, Google the origins. Anyway, AES takes as input a 128-bits block, and there are multiple cases for the key, and for each case ther is a number of rounds of the cipher :

- AES-128 : 128-bits key : 10 rounds
- AES-192 : 192-bits key : 12 rounds
- AES-256 : 256-bits key : 14 rounds

Representation of x in the Galois Field

x is represented as a 4×4 array in the Galoid Field. It is called at any point the **state** of the algorithm. The key, $k \in \mathbb{Z}_2^{128}$.

Round of AES

It works by **layers**. Each round contains 3 layers :

- Key addition layer : a 128-bit round key, or subkey, which has been derived from the main key in the key schedule, is XORed to the state
- Byte Substitution layer (S-Box) Each element of the state is nonlinearly transformed using lookup tables with special mathematical properties. This introduces *confusion* to the data, i.e., it assures that changes in individual state bits propagate quickly across the data path.
- Diffusion layer : it consists of two sublayers, both of which perform linear operations, in order to provide diffusion over all state bits :
  1. *ShiftRows* : permute the data on a **byte** level
  2. *MixColumns* : matrix multiplication which combines, mixes, blocks of 4 bytes $(s_1, ..., s_4)$. Here, if one of the values change, the hole state changes.

Block ciphers | Rijndael and AES

Rijndael data path

Input $x \in \mathbb{Z}_2^{128}$ and $k \in \mathbb{Z}_2^{128\text{ (or 192 or 256)}}$

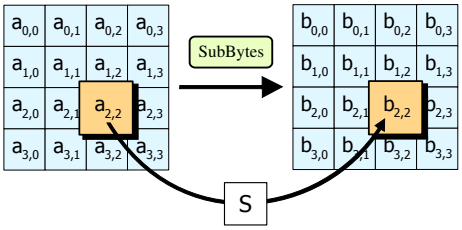
- Key schedule: $(K_i) \leftarrow \texttt{KeyExpansion}(k)$
- state $\leftarrow x$ (but represented as $\mathrm{GF}(2^8)^{4\times 4}$)
- $\texttt{AddRoundKey}(\text{state}, K_0)$
- For each round $i = 1$ to 9 (or 11 or 13):
  - $\texttt{SubBytes}(\text{state})$
  - $\texttt{ShiftRows}(\text{state})$
  - $\texttt{MixColumns}(\text{state})$
  - $\texttt{AddRoundKey}(\text{state}, K_i)$
- And for the last round:
  - $\texttt{SubBytes}(\text{state})$
  - $\texttt{ShiftRows}(\text{state})$
  - $\texttt{AddRoundKey}(\text{state}, K_{10\text{ (or 12 or 14)}})$

Output $y \leftarrow$ state back in $\mathbb{Z}_2^{128}$

32 / 69

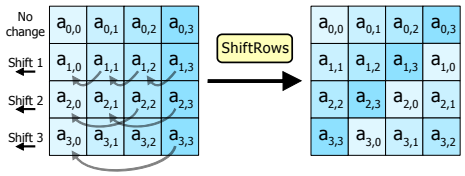
Block ciphers | Rijndael and AES

Rijndael – SubBytes



Block ciphers | Rijndael and AES

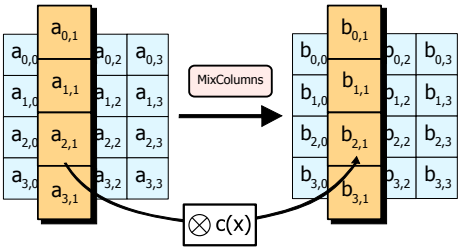
Rijndael – ShiftRows



34 / 69

Block ciphers | Rijndael and AES

Rijndael – MixColumns



37 / 69

36 / 69

Block ciphers | Rijndael and AES

Rijndael – MixColumns

Each column undergoes the following matrix multiplication:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

The inverse of MixColumns uses the inverse matrix:

$$\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = \begin{pmatrix} 0E & 0B & 0D & 09 \\ 09 & 0E & 0B & 0D \\ 0D & 09 & 0E & 0B \\ 0B & 0D & 09 & 0E \end{pmatrix} \begin{pmatrix} s_0 \\ s_1 \\ s_2 \\ s_3 \end{pmatrix}$$

38 / 69

Quick chat about Galois Field

In AES the finite field contains 256 elements and is denoted as $GF(2^8)$. This field is chosen because then each element of the field can be represented, in binary, by one byte. This is why in the S-box and MixColumns transforms, AES treats every byte of the internal data path as an element of the field $GF(2^8)$ and manipulates is by performing arithmetics in this finite field.

Inverting AES

As AES is not based on a Feistel network, all layers must be inverted for decryption. We begin by inverting last round, round n_r, then n_{r-1} until round 1.

- We first go through the same key expansion algorithm to get the subkeys K'_i from k if needed (remember, we are dealing with symmetric crypto, encryption and decryption are with the same key!)
- With the ciphertext y as input, we compute XOR with the last key chunk $K_{10,12\text{ or}14}$.
- Then, for each round, we compute the inverse of each step.
  - SubBytes becomes InvSubBytes
  - InvShiftRows
  - InvMixColumns
  - AddRoundKey stays the same (XOR)

1.4.3 Modes of operation

Okay we've seen the famous block ciphers DES, AES, but what to do with them ? As we have already said, it can not constitute an encryption scheme that is IND-CPA, because of their deterministic kind. We must build another architecture using those.

Electronic codebook (ECB)

A first example of architecture using block ciphers is the **electronic codebook** (ECB). It takes as input a message p, a secret key, and a number n which corresponds to the desired block length. So, p is first padded to reach a length that is a multiple of the block size n, and then each block is processed the same way using the block cipher encryption with the key. It is represented at the left figure below.

$$c_i = E_k(p_i)$$

The problem here is of course that every similar block of plaintext is encoded the same way.

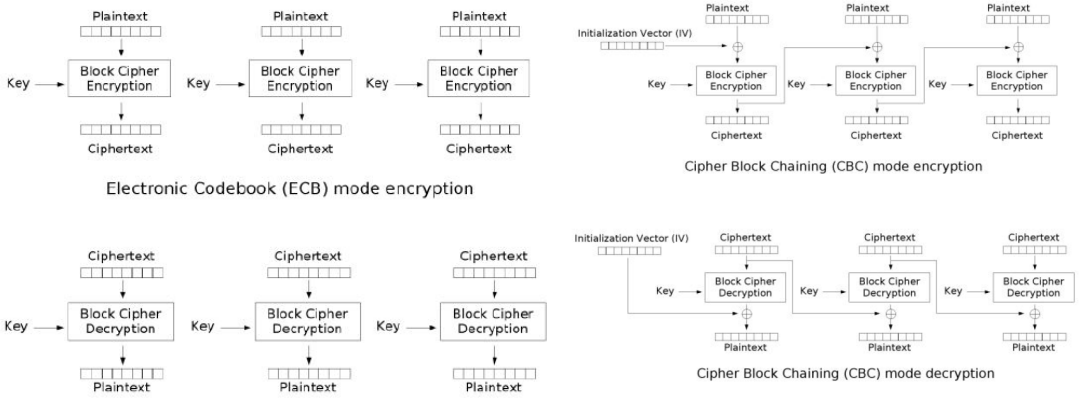
Ciphertext block chaining (CBC)

We take the same scheme as the ECB but at each round, the plaintext block is XORed with a vector before output. The first vector is an initialization vector (IV), and the vector to XOR with p_i is c_{i-1}, nothing but the ciphertext of previous encryption. Note that we here also add **diversification** : the initialization vector is **random**, and it must be a **nonce**. Then :

$$\begin{aligned} c_0 &= E_k(p_0 \oplus \text{IV}) \\ c_i &= E_k(p_i \oplus p_{i-1}) \end{aligned}$$

And to decrypt :

$$p_i = E_k^{-1}(c_i) \oplus c_{i-1}$$



Limits of CBC

What are the limits of the CBC ? Here, we present the probability that for encryption with a same key, $c_i = c'_j$ for different ciphertext blocks.

$$\begin{aligned} c_i &= c'_j \\ E_k(p_i \oplus c_{i-1}) &= E_k(p'_j \oplus c'_{j-1}) && \text{By def.} \\ p_i \oplus c_{i-1} &= p_j \oplus c'_{j-1} && \text{Because } E \text{ invertible} \\ p_i \oplus p'_j &= c_{i-1} \oplus c'_{j-1} \end{aligned}$$

Then, like we saw for the OTP, this means information is revealed on the plaintext ! But how likely is it that $c_i = c'_j$? This is related to the birthday paradox. Indeed, here, the ciphertexts of different blocks and different encryptions (same key, but other diversifier) are supposed to be completely random in respect with each other. Then, the probability of having such a *collision* is left to randomness. This is why it is related to the birthday paradox.

Birthday paradox

With a block size of n, the probability to get a collision after L attempts is

$$\Pr[\text{coll.}] = \frac{L^2}{2^{n+1}}$$

which means that it tends to $1/2$ if

$$L \sim 2^{n/2}$$

.

- For DES, the block size is 64 ($n = 64$), so the number of blocks resulting from the message must be of 2^{32} in order to have a problem. This is a large number, but not really.
- For AES, the block size is 128, which gives us 2^{64} blocks before having a big probability of collision.

The birthday paradox is hence more a problem for DES.

1.4.4 Counter mode

The Counter mode (CTR) is also an encryption mode that uses block cipher as stream cipher.

- Inputs :
  - Secret key $k \in \mathbb{Z}_2^m$
  - Plaintext $p \in \mathbb{Z}_2^*$
  - Diversifier $d \in \mathbb{N}$
- First, we put a target, the desired block size : n.
- Then, we cut our message $p \longrightarrow (p_1, p_2, \ldots, p_x)$. Each block is of n bits, except p_x that does not even require padding.
- We can now begin. The key stream is computed as followed, for i going from 1 to x :
  $$k_i = E_k(d \| i)$$
  The last key chunk, p_x is truncated to the length of p_x
- Encryption of block i :
  $$c_i = k_i \oplus p_i$$
- Output : $c \in \mathbb{Z}_2^*$

To decrypt, we can use the **same keystream** (remember that the nonce is public!) :

$$p_i = k_i \oplus c_i$$

1.4.5 Build MAC from block cipher : CBC-MAC

Aaaah, a bit of authentication ! Yes, block ciphers can also build MACs. Using the CBC architecture, but changing our goals, we can have :

- Inputs :
  - Secret key $k \in \mathbb{Z}_2^m$
  - Message $m \in \mathbb{Z}_2^*$
- Prepend m with its length : $m \Rightarrow |m| \ \|m$
- Pad this with 1 and 0s, cut into blocks $(m_1, m_2, \ldots, m_x)$
- Define $z_0 = 0^n$ and compute
  $$z_i = E_k(m_i \oplus z_{i-1})$$
- Output : MAC is z_x, the last chunk.

As we can see, we only keep the **last chunk** as output, in contrary with classical CBC. Verification goes by doing the same operation at home, with our key k which is the same because of symmetric crypto, and see if we get the same MAC.

1.4.6 Authenticated encryption

As already said, we've seen primitives. Then we can combine them to create a scheme that is secure and provides authentication ! For example, we can encrypt with one key, and compute a MAC with another key (using CBC-MAC for example). Or there are also some modes that provide encryption and authentication with the same key.

1.5 Permutations

Permutations are also primitives, and they are very important. A **permutation is basically** a bijective mapping in $\mathbb{Z}_2^b$:

$$f : \mathbb{Z}_2^b \to \mathbb{Z}_2^b$$

A As it is injective, is has an inverse.

1.5.1 The sponge construction

With the help of permutations, we can build the following architecture, that is called a **sponge function**.

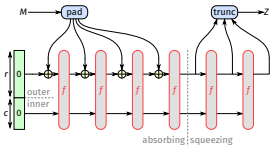
Two parameters, they are natural numbers :

- r : bits of rate
- c : bits of capacity (security parameter)

and of course the permutation f. Together, they form a **sponge function** :

Permutations

The sponge construction



A sponge function implements a mapping from $\mathbb{Z}_2^*$ to $\mathbb{Z}_2^\infty$ (truncated at an arbitrary length)

- $s \leftarrow 0^b$
- $M \| 10^*1$ is cut into r-bit blocks
- For each M_i do (absorbing phase)
  - $s \leftarrow s \oplus (M_i \| 0^c)$
  - $s \leftarrow f(s)$
- As long as output is needed do (squeezing phase)
  - Output the first r bits of s
  - $s \leftarrow f(s)$

55 / 69

The output can be of any length that we want, we just need to add more rounds of f, this is why we see the ∞ symbol.

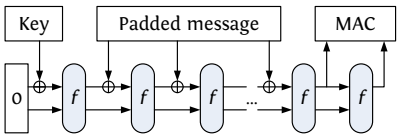
A keyed-sponge structure can be obtained if the message m is prefixed with the secret key K.

Keyed sponge for authentication

Again, here, the goal is not to encrypt the text, but to build a MAC from the key and the message. When Bachar receives the message with the MAC, he has the secret key and can thus compute the same operation at home to verify that no Omar intercepted the message and changed it authenticity.

Permutations

Keyed sponge for authentication



$\text{sponge}_K(M)$ gives a MAC on M under key K

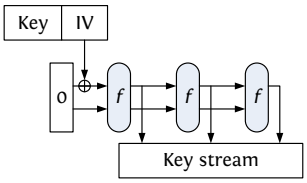
57 / 69

Keyed sponge as a stream cipher

Used here to build an encryption scheme : the sponge function acts as a key stream generator, and the encryption goes as it always do for stream ciphers.

Permutations

Keyed sponge as a stream cipher



$s_i \leftarrow \text{sponge}_K(d)$ produces keystream from key K and diversifier d

$$c_i = m_i + s_i \quad (\text{in } \mathbb{Z}_2)$$

Note: IV = d

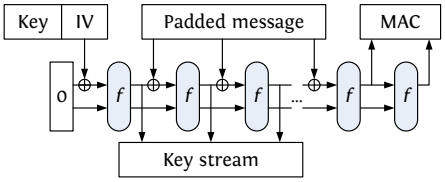
58 / 69

Authenticated encryption : spongeWrap

Ewa c'est quoi la prochaine étape, spongeBob ?

Permutations

Authenticated encryption: spongeWrap



Variant of the sponge construction:

- Keystream: $s_i \leftarrow \text{sponge}_K(d \| \text{previous plaintext blocks})$
- MAC: $\text{sponge}_K(d \| \text{plaintext})$

59 / 69

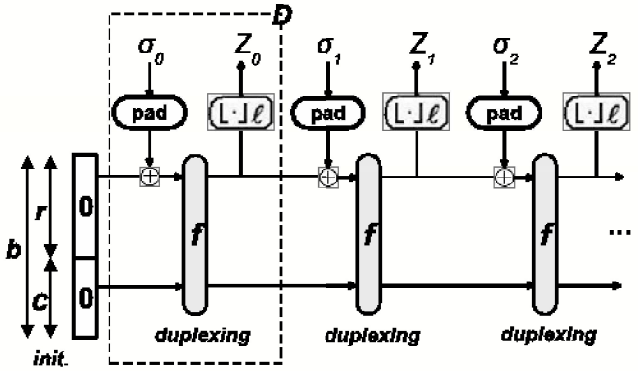
1.5.2 The duplex construction

Pobrane z czasopisma Annales AI- Informatica **http://ai.annales.umcs.pl**
Data: 28/12/2021 22:14:07

and sponge constructions in a coherent and rather simple and compact way. Original results are presented in Section 4. Security of authenticated encryption schemes defined for block ciphers and schemes based on the duplex construction is reviewed in Section 4.1. Section 4.2 shows comparison of some key wrapping schemes. The assessment proves that cryptographic schemes based on the duplex construction can be used for protection of the classified information up to the "TOP SECRET" level and unclassified information of different sensitivity levels. Section 4.3 briefly presents the design and evaluation of a cheap but very fast pseudo-random sequences generator based on the duplex construction and a slow random bit generator developed by Military Communication Institute. Finally, Section 5 concludes the paper.

2 The Duplex Construction

The duplex construction (Fig. 1), like the sponge construction [**4**], [**8**], uses a fixed length transformation or permutation f operating on a fixed number b of bits, a padding rule "pad" to build a cryptographic scheme. The duplex construction operates on a state of $b = r + c$ bits. The value b is called the width, the value r is called the bitrate and the value c the capacity. Different values for a bitrate and a capacity give the trade-off between speed and security. The higher bitrate gives the faster cryptographic function that is less secure. It is important that the last c bits of the b-bit state are never directly affected by the input blocks and are never output during the output producing. The capacity c, the most important security parameter, determines the attainable security level of the constructions, as proven in chapter 3. The duplex construction results in an object that accepts calls that take an input string and return an output string that depends on all inputs received so far. An instance of the duplex construction is called a *duplex object* and is denoted by D.



Rysunek 1. The duplex construction

A duplex object D has a state of b bits. Upon initialization all the bits of the state are set to zero. From then on one can send to it D.duplexing (σ, ℓ) calls, with σ as an input string and the requested number of bits. The maximum number of bits ℓ one

1.6 Pseudo-random functions