3. Hashing

3-Hashing.

3.1 Definitions and requirements

3.1.1 Cryptographic hash functions

A #hash-function is a **deterministic** function that takes an **input message**, hash it and outputs a #digest. It **can not be reversed** and **always returns the same digest for a given input**.

$$h : \{0,1\}^* \to \{0,1\}^n$$

There are multiples **applications**:

- *Signatures*: input the hash of the message in the signature instead of the message! $\to sign_{RSA}(h(M))$ instead of $sign_{RSA}(M)$.
- *Key derivation*: a master key K to derived keys ($K_i = h(K \mid\mid i)$)
  - Hash the master key concatenated with the number of the key to have a derived key.
- *Bit commitment, predictions*: let's take the example that we want to vote for someone, we hash our vote and store it somewhere. Then, when everyone has voted, if you rehash your vote and the digest is the same as the one stored, you can then prove that you told it earlier!
- *Message authentication*: $h(K \mid\mid M)$

3.1.2 Generalised: extendable output function (XOF)

An **extendable output function** #XOF is a function in which the *output can be extended to any length*.

$$h : \{0,1\}^* \to \{0,1\}^\infty$$

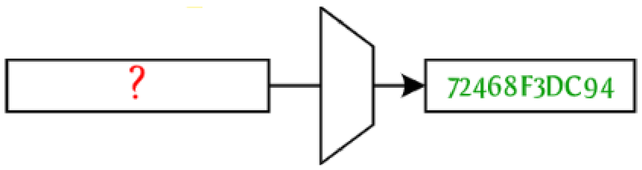
There are multiple applications:

- *Signatures*: full domain hashing, mask generating function
- *Key derivation*: as many/long derived keys as needed
- *Stream cipher*: $C = P \oplus h(K \mid\mid nonce)$. #stream-cipher

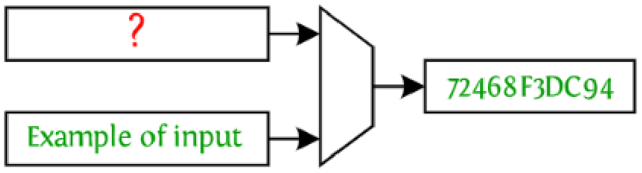
3.1.3 Requirements of a hash function

A #hash-function has several #requirements :

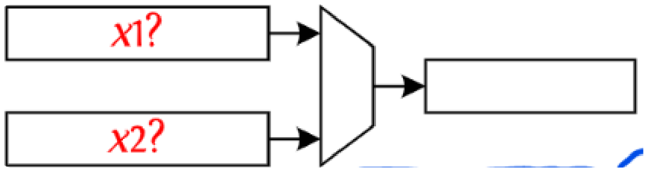
- #Preimage-resistance : *find back the input from the digest*
  - **Given** $y \in Z_2^n$**, find** $x \in Z_2^*$ **such that** $h(x) = y$



  - If h is a random function, about 2^n attempts are needed to retrieve the input from the digest
  - *Example*: From a derived key $K_1 = h(K \mid\mid 1)$ find back the master key K.
- #Second-preimage-resistance : *find another input that gives the same digest as an input that we have.*
  - *Given \$x \in Z_{2}^, $find\$x \neq x'$ such that $h(x) = h(x')$



  - If h is a random function, about 2^n attempts are needed to find another input that gives the same digest
  - *Example*: Signature forging, given M and $sign(h(M))$, find $M' \neq M$ with equal signature
- #Collision-resistance : *find 2 inputs that gives the same output*
  - **Find $x_1 \neq x_2$ such that $h(x_1) = h(x_2)$**



  - If h is a random function, about $2^{n/2}$ attempts are needed (see #birthday-paradox).
  - *Example*: secretary signature forging
    - Set of good messages $\{M_{good}i\}$
    - Set of bad messages $\{M_{bad}i\}$
    - Find $h(M_{good}i) = h(M_{bad}j)$
    - Boss signs M_{good}, but valid also for M_{bad}

These where the **3 most importants requirements**, but other properties are quite important too:

- *Correlation free*
- *Resistant again length-extension attacks* (for MAC).

A **good hash function** should behave like a **random mapping**.

What implies what?

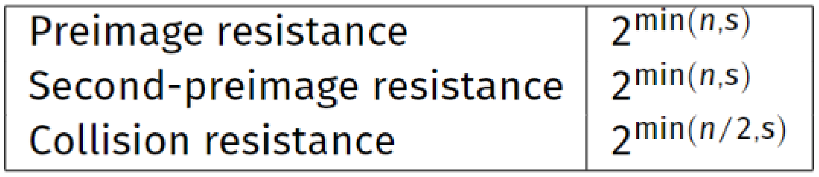
Second preimage resistance $\implies$ Collision resistance

- Let us consider that an adversary is able to find a second preimage of $y = h(x)$.
  This means that he can find an $x' \neq x$ s.t. $h(x') = y = h(x)$
- By definition of the second preimage, he is given x. Therefore he is able to produce a conflict.
  #CQFD
  Collision resistance $\implies$ Second preimage resistance
- Assume that a hash function is collision resistant (infeasible to find a collision)
- Then if an adversary could find a second preimage, he would also be able to find a collision. Which was assumed to be infeasible!
- Hence it is also infeasible to find a second preimage.
  #CQFD
  Collision resistance $\not\Leftarrow$ Second preimage resistance
- If a hash function is second preimage resistant, a collision could nevertheless be found by other means than the second preimage attack.
- Intuitively, a collision attack does not require a specific image y
- So a collision attack in general does not help finding a preimage for the given image y
  #CQFD #TP3-ex1

Security requirements summarised

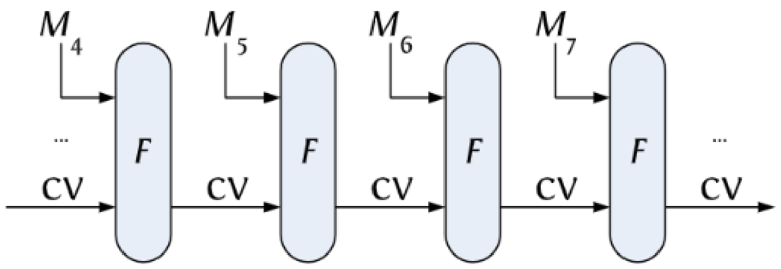
- #hash-function / #XOF h with $n-$bit output
- Modern security requirements
  - h behaves like a random mapping *up to security strength* s.

- Classical security requirements derived from it:



Preimage resistance	$2^{\min(n,s)}$
Second-preimage resistance	$2^{\min(n,s)}$
Collision resistance	$2^{\min(n/2,s)}$

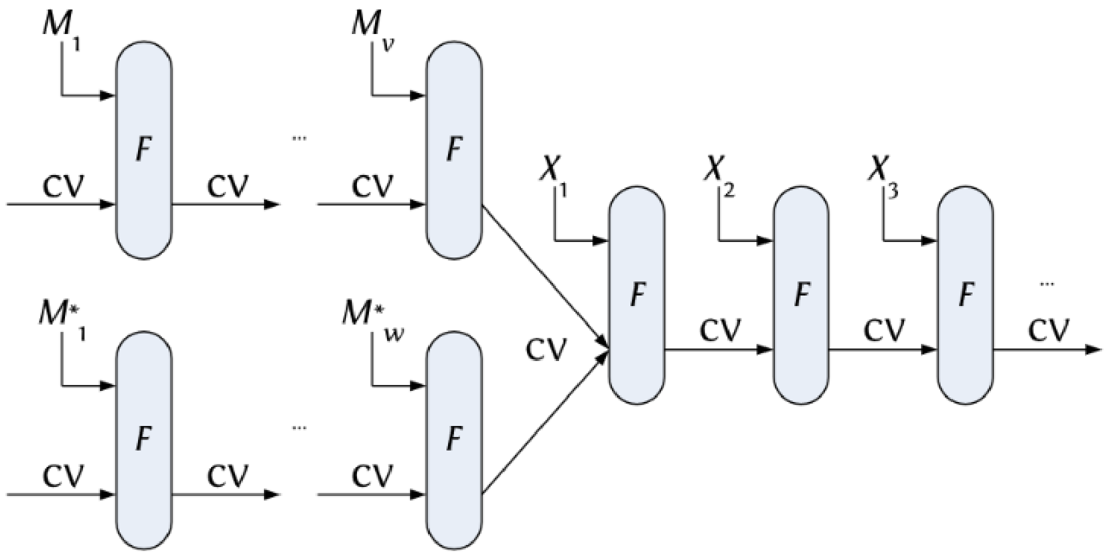
3.1.4 Iterated functions



All practical #hash-function are **iterated**, meaning that the **message M is cut into blocks** $M_1 \dots M_l$ and that we have **a $q-$bit chaining value**.

The **output** is function of the final chaining value.

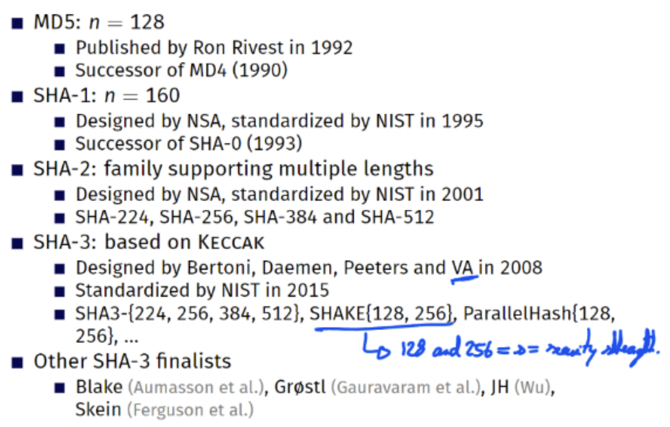
Internal collisions



We can have #internal-collision when different inputs M and M^* gives the same #chaining-value . The messages $M \,||\, X$ and $M^* \,||\, X$ always collide for any string X!

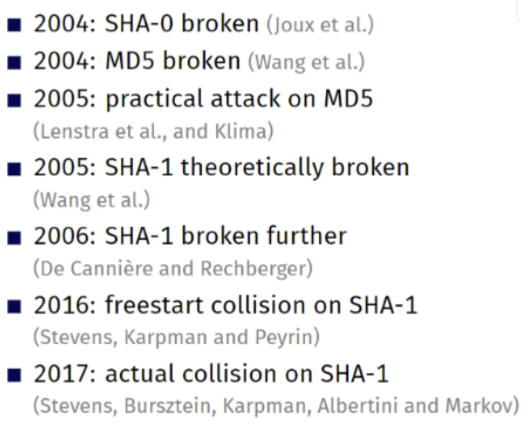
$\rightarrow$ This is a problem! It does not occur in a random mapping.

3.1.5 Examples of Hash functions



- MD5: $n = 128$
  - Published by Ron Rivest in 1992
  - Successor of MD4 (1990)
- SHA-1: $n = 160$
  - Designed by NSA, standardized by NIST in 1995
  - Successor of SHA-0 (1993)
- SHA-2: family supporting multiple lengths
  - Designed by NSA, standardized by NIST in 2001
  - SHA-224, SHA-256, SHA-384 and SHA-512
- SHA-3: based on KECCAK
  - Designed by Bertoni, Daemen, Peeters and VA in 2008
  - Standardized by NIST in 2015
  - SHA3-{224, 256, 384, 512}, SHAKE{128, 256}, ParallelHash{128, 256}, ...

    ↳ 128 and 256 = s = security strength.
- Other SHA-3 finalists
  - Blake (Aumasson et al.), Grøstl (Gauravaram et al.), JH (Wu), Skein (Ferguson et al.)

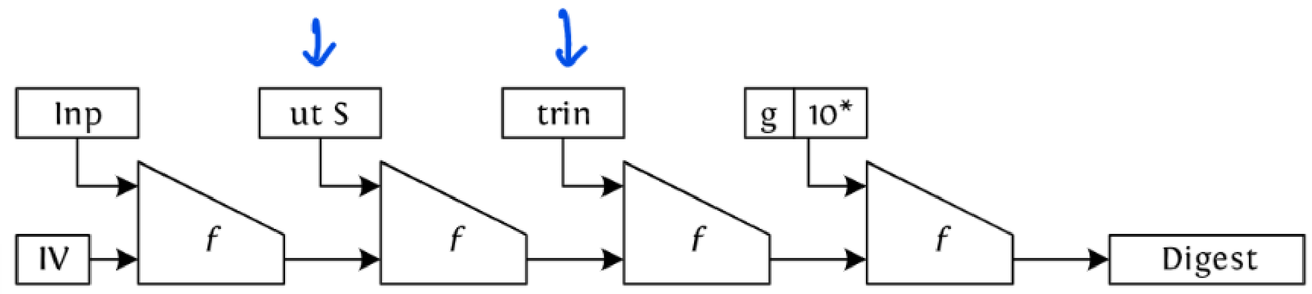


- 2004: SHA-0 broken (Joux et al.)
- 2004: MD5 broken (Wang et al.)
- 2005: practical attack on MD5 (Lenstra et al., and Klima)
- 2005: SHA-1 theoretically broken (Wang et al.)
- 2006: SHA-1 broken further (De Cannière and Rechberger)
- 2016: freestart collision on SHA-1 (Stevens, Karpman and Peyrin)
- 2017: actual collision on SHA-1 (Stevens, Bursztein, Karpman, Albertini and Markov)

3.2 Zooming from Merkle-Damgard into SHA-1

3.2.1 Merkle-Damgard

The #Merkle-Damgard-construction is a **method** for **building cryptographic** #hash-function . It **divides the input messages into fixed-size blocks**, **processes each block** using a #Collision-resistance **compression function** and **chains the results**. The **final state** serves has the **hash value** for the entire message. It is used in MD5, SHA-1 and SHA-2.



1. **Initialization:**
   - The *input* message is *divided into fixed-size blocks*.
   - An *initial value*, known as the "**IV**" (Initialization Vector), is defined.
2. **Compression Function:** from $n + m$ to n bits
   - A compression function takes a fixed-size input block and the current state as input.
   - It *compresses the input block and updates the state*.
   - The compression function is designed to be **collision-resistant**.
3. **Chaining:**
   - The *output of the compression function becomes the new state for the next iteration*.
   - The chaining of states continues until all blocks are processed.
4. **Padding:**
   - If necessary, padding is added to the last block to ensure it has the required size.
   - The length of the original message might also be appended to ensure uniqueness.
5. **Finalisation:**
   - The final state (output of the last compression function) serves as the hash value for the entire message.

This respect #Collision-resistance , if you give two different inputs, but keep the same compression function, the digests will be different.

You can also observe **recurrence** (modulo the padding):

- $h(M_1) = f(IV, M_1) = CV_1$
- $h(M_1 \mid\mid \cdots \mid\mid M_i) = f(CV_{i-1}, M_i) = CV_i$

This can be a problem because a #forgery on *naïve* #MAC can be done:

- $MAC(M) = h(Key \mid\mid M) = CV$
- $MAC(M \mid\mid suffix) = h(Key||(M||suffix)) = f(CV \mid\mid suffix)$
- This means that an **attacker** can **create a valid MAC for a new, extended message without knowledge of the secret key**. Which is a problem!
  $\implies$ The **solution** is #HMAC

$$HMAC(M) = h(Key_{out} \mid\mid h(Key_{in} \mid\mid M)) = h(Key_{out}||MAC(M))$$

  It consists of **hashing the output key concatenated to the MAC**.

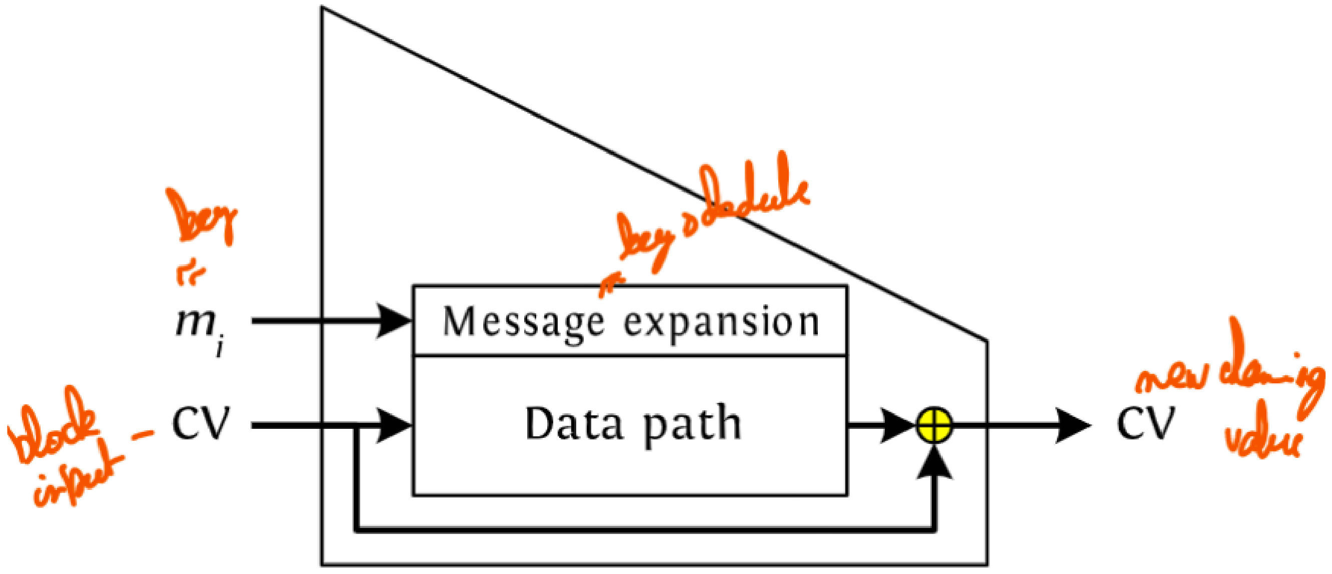
Strengthening of Merkle-Damgard

There are some ways to **strengthen** this construction method such as:

1. **Padding Schemes:** Implementing more secure padding schemes to prevent attacks like length extension. This involves carefully designing how the message is padded before processing.
2. **Finalization Steps:** Adding specific steps during the finalization process to ensure that certain attacks, such as collision attacks, are more difficult to execute successfully.
3. **Iterative Designs:** Some hash functions use an iterative design, where the Merkle-Damgård construction is applied multiple times in a nested or interleaved manner. This can provide additional security against certain types of attacks.
4. **Use of Dedicated Functions:** Introducing dedicated functions within the construction that further strengthen the overall security, such as incorporating specialized mixing or diffusion functions.

3.2.2 Davies-Meyer

The #Davies-Meyer-construction is a method used to build **collision resistant** cryptographic #hash-function. It's particularity is that it **transforms a** **#Block-ciphers** **into a compression function** that **creates a hash function**.

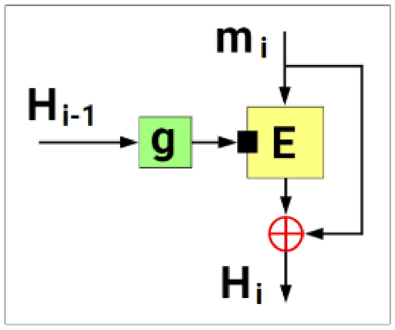


1. The **block cipher** (that can be AES, CBC,...) as a **compression function**
   - The input is a chaining value, a *fixed-size block* of data
   - The *key* is the *message block*
   - Produces a fixed-size output
2. **Message processing**
   - The *input message* is **divided into blocks**
   - Each block is ***XORed** with the current state* before being encrypted with the block cipher. (This is to avoid the reverse operation).

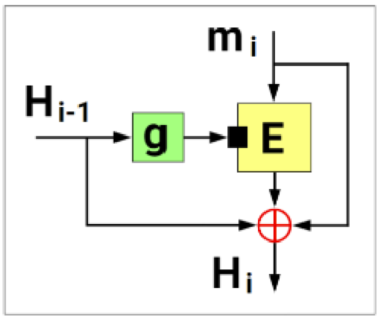
- The result becomes the new state for the next iteration.

3. The **final state**, after processing all messages blocks, serves as the **hash values**

3.2.3 Other constructions using block ciphers



Matyas-Meyer-Oseas
[Matyas et al., IBM Tech. D. B., 1985]



Miyaguchi-Preneel
[Miyaguchi et al., NTT Rev., 1990], [Preneel, PhD th., 1993]

3.2.4 SHA-1

#SHA-1 or **Secure Hash Algorithm 1** is a #hash-function that takes an input message (of **variable length**) and produces a **160-bits hash value**.

It uses #Davies-Meyer-construction with a data path of 160-bits (5 states of 32 bits) and a **message expansion** $m = 512 = 16 \times 32$ (16 words of 32 bits).

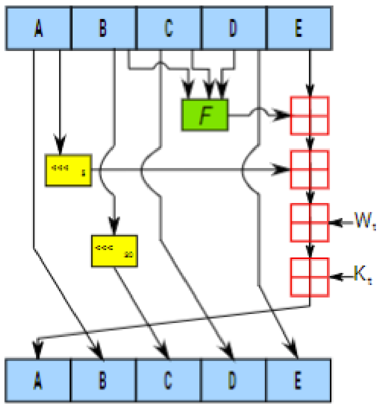
The states (5 words of 32 bits words) are initialised - this is a constant that will always be the same :

$$(A, B, C, D, E) = (67452301, EFCDAB89, 98BADCFE, 10325476, C3D2E1F0)$$

The 16 words of 32 bits are expanded to form 80 words (t is the number of turns) $(w_0, \dots, w_{15})$

$$w_t = (w_{t-3} \oplus w_{t-8} \oplus w_{t-14} \oplus w_{t-16}) \lll 1 \; (16 \leq t \leq 79)$$

The **data path is composed of 80 steps**, at each step, the word changes.



details for F

$0 \leq t \leq 19$	$f(B,C,D) = (B \odot C) \oplus (\bar{B} \odot D)$	$K_t = 5A827999$
$20 \leq t \leq 39$	$f(B,C,D) = B \oplus C \oplus D$	$K_t = 6ED9EBA1$
$40 \leq t \leq 59$	$f(B,C,D) = (B \odot C) \oplus (B \odot D) \oplus (C \odot D)$	$K_t = 8F1BBCDC$
$60 \leq t \leq 79$	$f(B,C,D) = B \oplus C \oplus D$	$K_t = CA62C1D6$

→ selection function

→ XOR

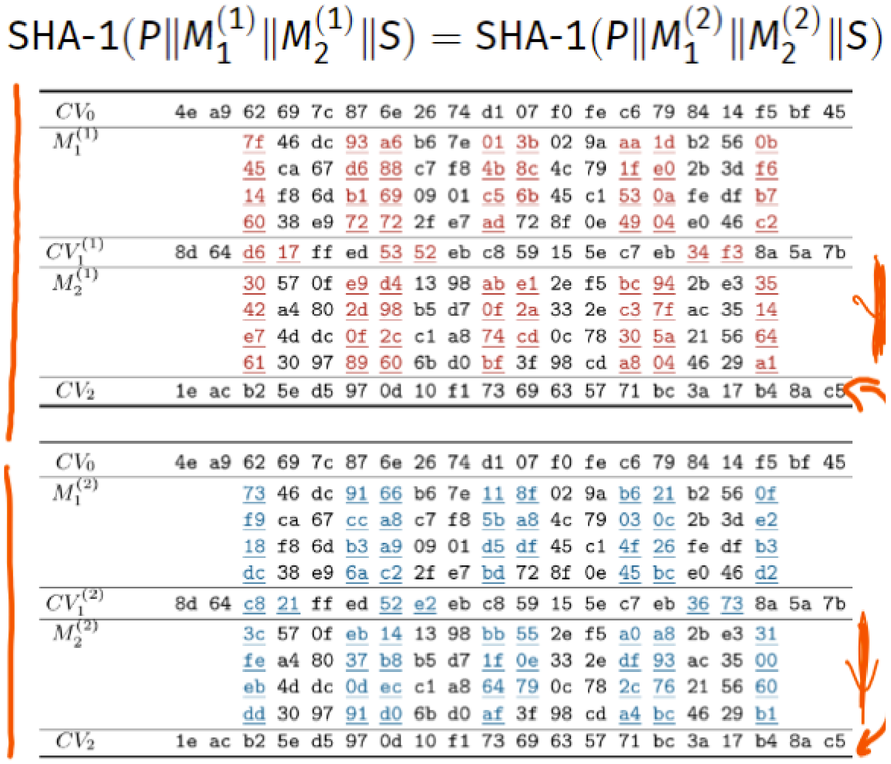
→ majority function

Here we can see that for each step:

- $A = E + F(B,C,D) + shift_5(A) + W_t + K_t$
  - F being first the #selection function, then the #parity function, then the #majority function and finally the #parity function again.
- $B = A$
- $C = shift_{30}(B)$
- $D = C$
- $E = D$
  At the end of the 80 rounds, the result is added to the initial vector.

Once all blocks have been computed, all 5 states are concatenated (32bits x5) to form the **160 bits signature/digest**.

One sad day of 2005, a collision has been found in SHA-1. So they made it stronger! But in 2017, SHA-1 died... They found an #internal-collision , the estimated complexity was thus of $2^{63} \ll 2^{80}$.



$$\text{SHA-1}(P \| M_1^{(1)} \| M_2^{(1)} \| S) = \text{SHA-1}(P \| M_1^{(2)} \| M_2^{(2)} \| S)$$

CV_0	4e	a9	62	69	7c	87	6e	26	74	d1	07	f0	fe	c6	79	84	14	f5	bf	45
$M_1^{(1)}$			7f	46	dc	93	a6	b6	7e	01	3b	02	9a	aa	1d	b2	56	0b		
			45	ca	67	d6	88	c7	f8	4b	8c	4c	79	1f	e0	2b	3d	f6		
			14	f8	6d	b1	69	09	01	c5	6b	45	c1	53	0a	fe	df	b7		
			60	38	e9	72	72	2f	e7	ad	72	8f	0e	49	04	e0	46	c2		
$CV_1^{(1)}$	8d	64	d6	17	ff	ed	53	52	eb	c8	59	15	5e	c7	eb	34	f3	8a	5a	7b
$M_2^{(1)}$			30	57	0f	e9	d4	13	98	ab	e1	2e	f5	bc	94	2b	e3	35		
			42	a4	80	2d	98	b5	d7	0f	2a	33	2e	c3	7f	ac	35	14		
			e7	4d	dc	0f	2c	c1	a8	74	cd	0c	78	30	5a	21	56	64		
			61	30	97	89	60	6b	d0	bf	3f	98	cd	a8	04	46	29	a1		
CV_2	1e	ac	b2	5e	d5	97	0d	10	f1	73	69	63	57	71	bc	3a	17	b4	8a	c5

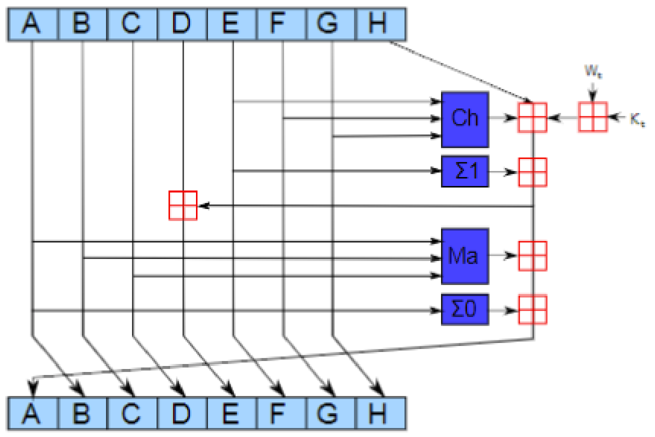
CV_0	4e	a9	62	69	7c	87	6e	26	74	d1	07	f0	fe	c6	79	84	14	f5	bf	45
$M_1^{(2)}$			73	46	dc	91	66	b6	7e	11	8f	02	9a	b6	21	b2	56	0f		
			f9	ca	67	cc	a8	c7	f8	5b	a8	4c	79	03	0c	2b	3d	e2		
			18	f8	6d	b3	a9	09	01	d5	df	45	c1	4f	26	fe	df	b3		
			dc	38	e9	6a	c2	2f	e7	bd	72	8f	0e	45	bc	e0	46	d2		
$CV_1^{(2)}$	8d	64	c8	21	ff	ed	52	e2	eb	c8	59	15	5e	c7	eb	36	73	8a	5a	7b
$M_2^{(2)}$			3c	57	0f	eb	14	13	98	bb	55	2e	f5	a0	a8	2b	e3	31		
			fe	a4	80	37	b8	b5	d7	1f	0e	33	2e	df	93	ac	35	00		
			eb	4d	dc	0d	ec	c1	a8	64	79	0c	78	2c	76	21	56	60		
			dd	30	97	91	d0	6b	d0	af	3f	98	cd	a4	bc	46	29	b1		
CV_2	1e	ac	b2	5e	d5	97	0d	10	f1	73	69	63	57	71	bc	3a	17	b4	8a	c5

As we can see here, the CV1 of the two different calls of the function are different, so this is normal! But then both CV2 are the same. Hence, proving that a collision occurred.

3.2.5 From SHA-1 to SHA-2

The #SHA-2 comes in *multiple variants with different hash lengths*, including:

- SHA-224, SHA-256:
  - $n = 256 = 8 \times 32$ and $m = 512 = 16 \times 32$
- SHA-384, SHA-512
  - $n = 256 = 8 \times 64$ and $m = 512 = 16 \times 64$
  The **message expansion of SHA-2** is **non linear**! The one from SHA-1 was linear.



The **data path mixing is stronger!**

3.3 Modern generic security

3.3.1 Generic security: indifferentiability

#Indifferentiability is a concept that is **used to analyse and define the security of hash functions**, encryption schemes, and other cryptographic primitives.

It states that a **cryptographic construction is secure** if an **adversary cannot distinguish** between **the idealized version of the construction and a truly #random-oracle** (or random process) in a meaningful way.

Here are the key points related to indifferentiability:

1. **Indistinguishability:**
   - Indifferentiability focuses on whether an adversary can distinguish between the idealised scenario (Random Oracle) and the real-world scenario.
   - If an adversary cannot distinguish between the two scenarios with a probability significantly better than random chance, the construction is considered **indifferentiable**.
2. **Generic Security:**
   - Indifferentiability is a form of generic security analysis, meaning it provides a framework for proving the security of a broad class of constructions rather than analysing each specific construction separately.
   - It allows for the analysis of the security of a cryptographic primitive by comparing it to an idealised model without getting into the details of the specific algorithms used.
3. **Applicability:**
   - Indifferentiability is commonly applied to the analysis of hash functions, block ciphers, and other cryptographic primitives.
   - It provides a formal and abstract way to reason about the security properties of these primitives without being overly specific about the implementation details.

Security of RO and similar to RO

#TP3-ex8

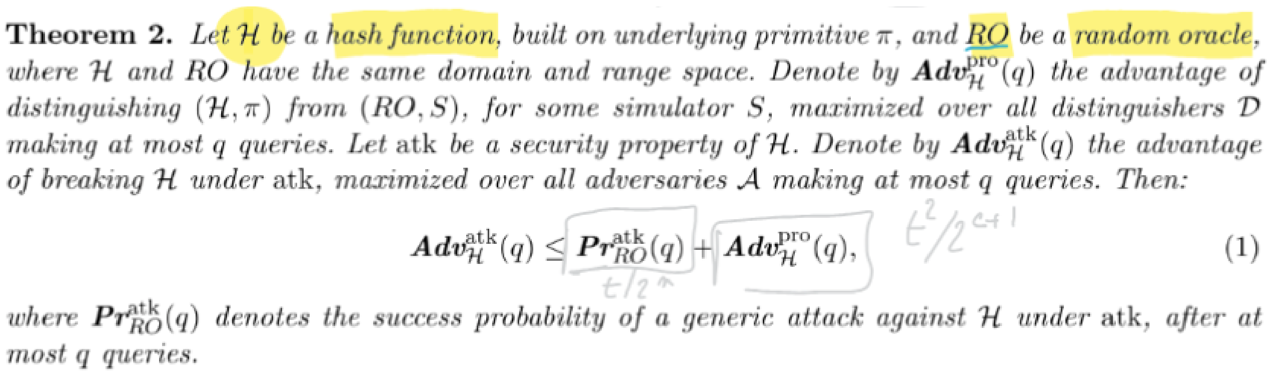
To see the security of a #random-oracle , we do the following computation:

- Note that the advantage (*Adv*) can be a known attack that has a certain security parameter c
- t is the number of tries done by the attacker
- n is the length of the digest (number of bits)
- $Pr(UP \cap \text{ mode behaves as an RO}) = \frac{\#tries}{\#possibilies} = \frac{t}{2^n}$
- $Pr(\text{distinguish mode from RO}) = \frac{t^2}{2^{c+1}}$ *for a sponge!*, otherwise it should be given in the enonce.

  Pr(UP on a given mode/construction)

= Pr(UP ∩ mode behaves as RO) + Pr(UP ∩ not Mode behalves as RO)

$\leq$ **Pr(UP on a RO) + Pr(distinguish mode from RO)** $= \frac{t}{2^n} + \frac{t^2}{2^{c+1}}$



Theorem 2. *Let $\mathcal{H}$ be a hash function, built on underlying primitive π, and RO be a random oracle, where $\mathcal{H}$ and RO have the same domain and range space. Denote by $\mathbf{Adv}^{\text{pro}}_{\mathcal{H}}(q)$ the advantage of distinguishing $(\mathcal{H}, \pi)$ from (RO, S), for some simulator S, maximized over all distinguishers $\mathcal{D}$ making at most q queries. Let* atk *be a security property of $\mathcal{H}$. Denote by $\mathbf{Adv}^{\text{atk}}_{\mathcal{H}}(q)$ the advantage of breaking $\mathcal{H}$ under* atk*, maximized over all adversaries $\mathcal{A}$ making at most q queries. Then:*

$$\mathbf{Adv}^{\text{atk}}_{\mathcal{H}}(q) \leq \underset{t/2^n}{\boxed{\mathbf{Pr}^{\text{atk}}_{RO}(q)}} + \boxed{\mathbf{Adv}^{\text{pro}}_{\mathcal{H}}(q),} \quad t^2/2^{c+1} \qquad (1)$$

where $\mathbf{Pr}^{\text{atk}}_{RO}(q)$ denotes the success probability of a generic attack against $\mathcal{H}$ under atk*, after at most q queries.*

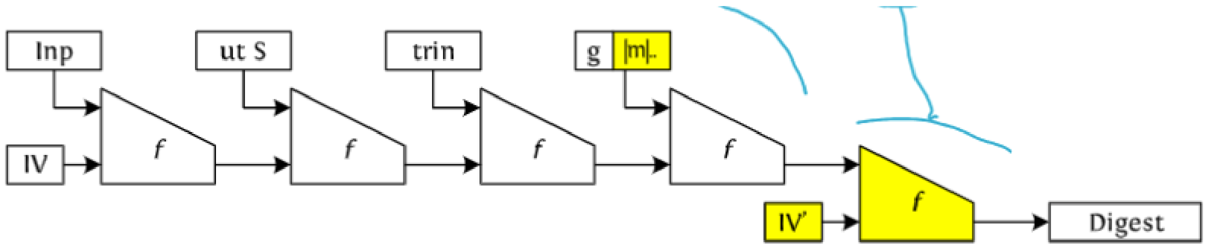
Limitations of indifferentiability

#Indifferentiability is limited because:

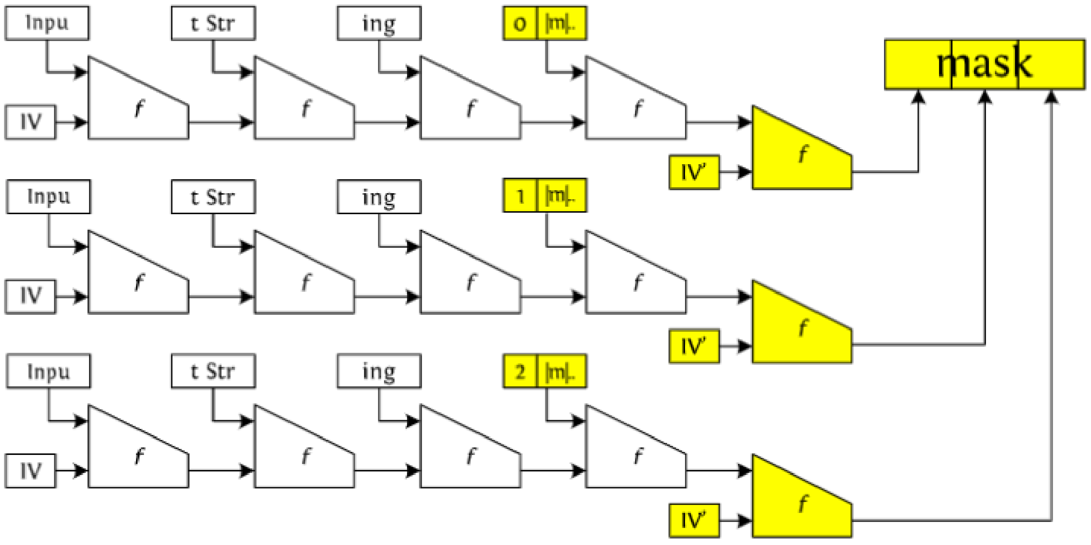
- It is only **about the mode**, there is no security proof with a concrete primitive.
- It is only about single-stage games
  - *Example*: hash-based storage auditing $Z = h(File \parallel C)$

Making Merkle-Damgard indifferentiable

To make #Merkle-Damgard-construction #Indifferentiability , envelop it: **hash the input and hash the hash**!

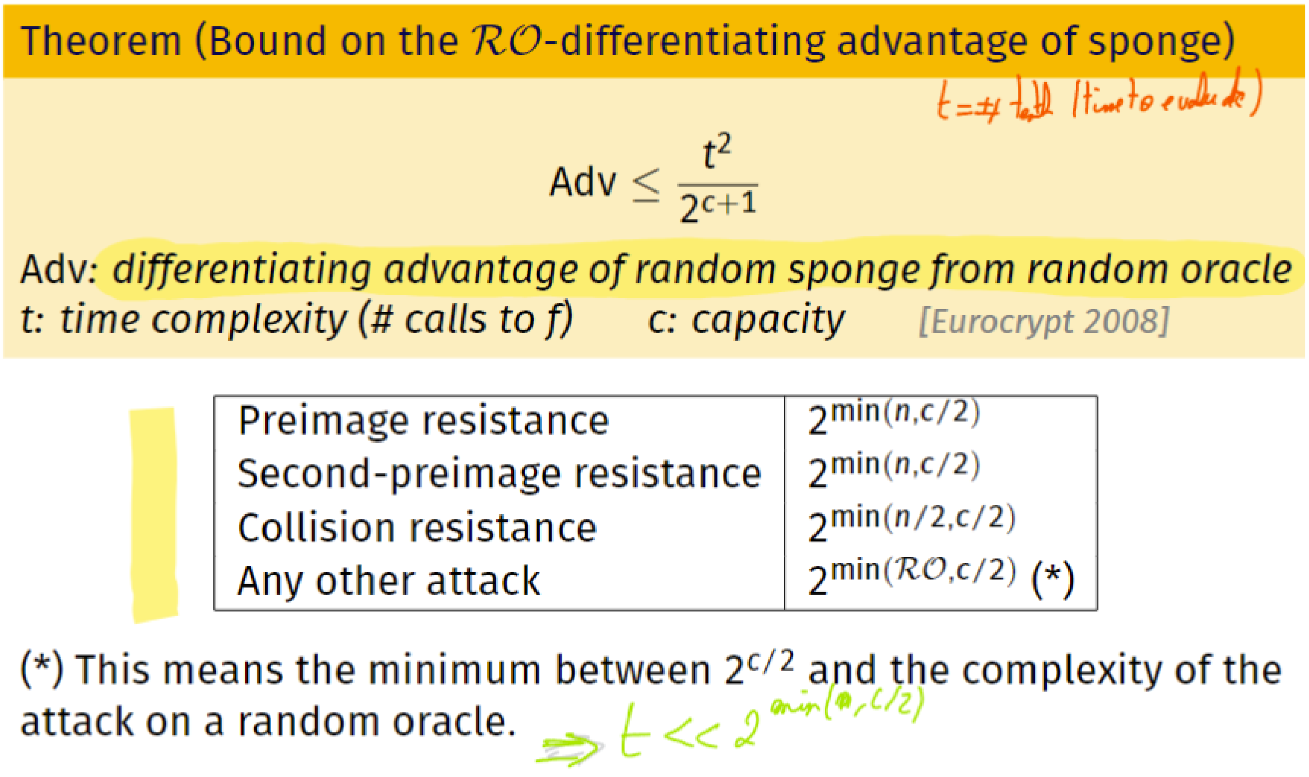


To make it suitable for #XOF : mask the generating function construction "MGF1":



3.3.2 The sponge construction

The #sponge construction is used in SHA-3 and here is the information about its security:



Theorem (Bound on the $\mathcal{RO}$-differentiating advantage of sponge)

$t = \#$ total (time to evaluate)

$$\text{Adv} \leq \frac{t^2}{2^{c+1}}$$

Adv: *differentiating advantage of random sponge from random oracle*
t: *time complexity (# calls to f)* c: *capacity* [Eurocrypt 2008]

Preimage resistance	$2^{\min(n,c/2)}$
Second-preimage resistance	$2^{\min(n,c/2)}$
Collision resistance	$2^{\min(n/2,c/2)}$
Any other attack	$2^{\min(\mathcal{RO},c/2)}$ (*)

(*) This means the minimum between $2^{c/2}$ and the complexity of the attack on a random oracle. $\Rightarrow t \ll 2^{\min(\mathcal{RO}, c/2)}$

Here is an **example** for a hash function that takes an **9 bits message** and outputs an **8 bit digest**.

- $r = 3, c = 1, b = r + c = 4$

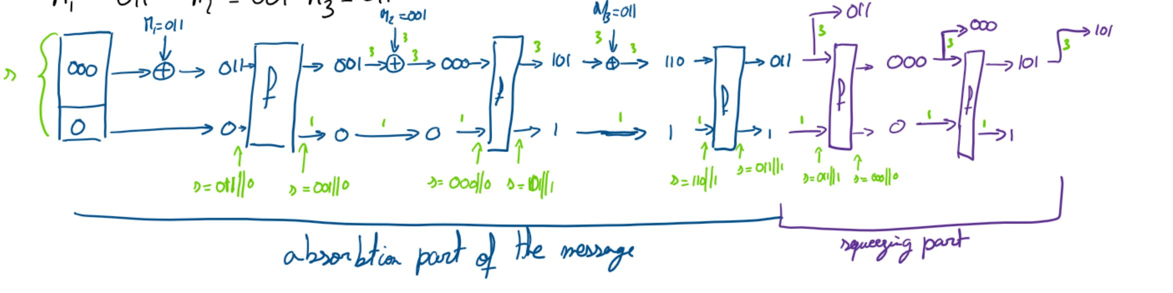
$f : \{0,1\}^4 \rightarrow \{0,1\}^4$

0000	→ 1011
0001	→ 1110
0010	→ 1111
0011	→ 0001
0100	→ 0100
0101	→ 1000
0110	→ 0010
0111	→ 0000
1000	→ 0011
1001	→ 1101
1010	→ 0101
1011	→ 0110
1100	→ 1001
1101	→ 0111
1110	→ 1010
1111	→ 1100

$r + c = 4$ $r = 3$ $c = 1$

$M = 011\ 0\ 01011$

$M_1 = 011$ $M_2 = 001$ $M_3 = 011$ 3 bc $r=3$



$\text{hash}(m) = 011 \| 000 \| 10$ ⟶ discard the last bit to have an 8 bits output

3.4 Inside SHA-3

3.4.1 KECCAK-f

The seven permutation army. Repetitions of a simple round function that operates on a 3D state (**5x5 lanes** up to **64-bit** each).

```
KECCAK-F[b](A) {
  forall i in 0…n_r-1
    A = Round[b](A, RC[i])
  return A
}

Round[b](A,RC) {
  θ step
  C[x] = A[x,0] xor A[x,1] xor A[x,2] xor A[x,3] xor A[x,4],  forall x in 0…4
  D[x] = C[x-1] xor rot(C[x+1],1),                             forall x in 0…4
  A[x,y] = A[x,y] xor D[x],                                    forall (x,y) in (0…4,0…4)

  ρ and π steps
  B[y,2*x+3*y] = rot(A[x,y], r[x,y]),                          forall (x,y) in (0…4,0…4)

  χ step
  A[x,y] = B[x,y] xor ((not B[x+1,y]) and B[x+2,y]),           forall (x,y) in (0…4,0…4)

  ι step
  A[0,0] = A[0,0] xor RC

  return A
}
```

χ - the **non-linear mapping** flip bit if neighbors exhibit 01 pattern.

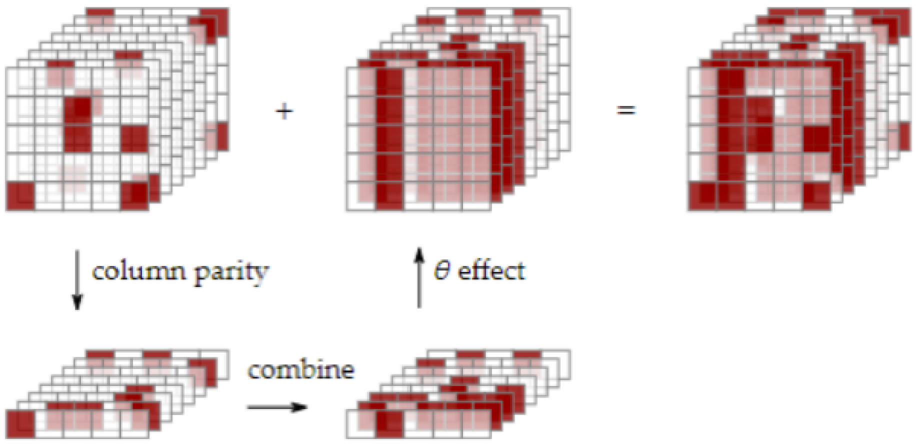
- operates independently and in parallel on **5-bit rows**
- **Cheap**: small number of operations per bit
- Algebraic degree 2, inverse has degree 3.

θ - **mixing bits**

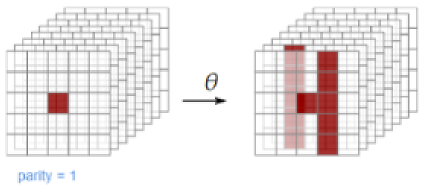
- Compute the parity $c_{x,z}$ of each column
- Add each cell **parity** of neighboring columns

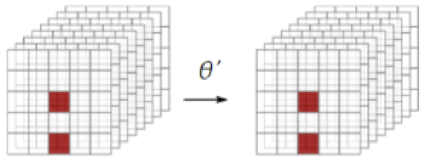
$$b_{x,y,z} = a_{x,y,z} \oplus c_{x-1,z} \oplus c_{x+1,z-1}$$

- **Cheap**: two XORs per bit and allows a good diffusion.



Diffusion of θ:





$$1+\left(1+y+y^2+y^3+y^4\right)\left(x+x^4z\right)$$
$$\left(\bmod \left\langle 1+x^5, 1+y^5, 1+z^w\right\rangle\right)$$

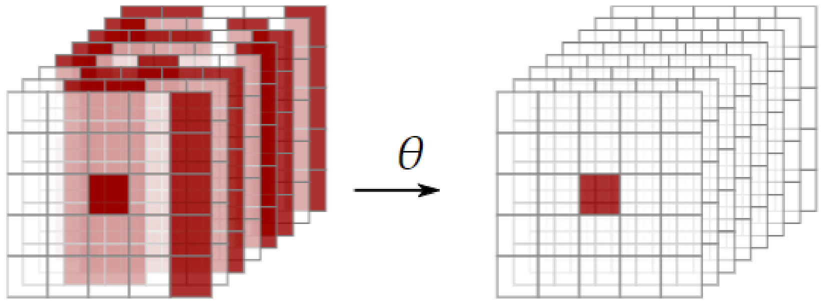
Here, the parity of the column is 1 so output
-> right column is full
-> column in the back and left is full

$$1+\left(1+y+y^2+y^3+y^4\right)\left(x+x^4z\right)$$
$$\left(\bmod \left\langle 1+x^5, 1+y^5, 1+z^w\right\rangle\right)$$

Here, the parity of the column is 0
so output doesn't change!

Diffusion of θ^{-1} - Q is dense so

- The diffusion from a single bit output to input is very high
- Increases resistance against LC/DC and algebraic attacks



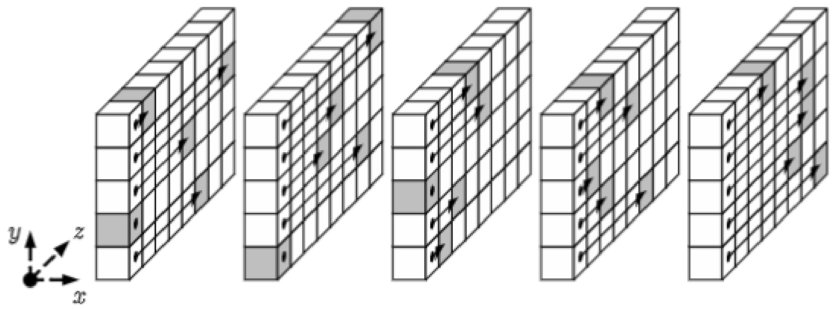
$$1+\left(1+y+y^2+y^3+y^4\right)\mathbf{Q},$$
$$\text{with } \mathbf{Q}=1+\left(1+x+x^4z\right)^{-1} \bmod \left\langle 1+x^5, 1+z^w\right\rangle$$

ρ for inter-slice dispersion

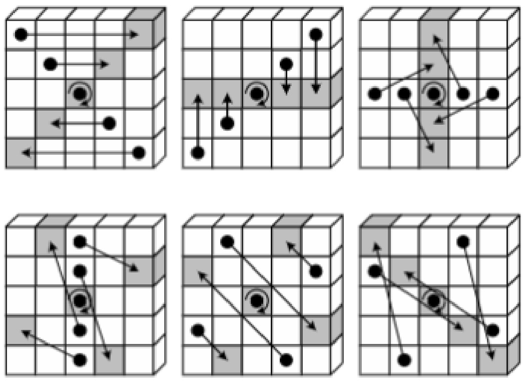
- To add some diffusion between the slices, do some **cyclic shifts of lanes with offsets**, *every lane is shifted by a different amount*:

$$i(i+1)/2 \bmod 2^{\ell}, \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix}^{i-1} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

- The offsets cycle through all values below 2^l



π for disturbing horizontal/vertical alignment:



$$a_{x,y} \leftarrow a_{x',y'} \text{ with } \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 2 & 3 \end{pmatrix} \begin{pmatrix} x' \\ y' \end{pmatrix}$$

ι to break symmetry

- XOR of round-dependent constant to lane in origin
- Without this, the round mapping would be symmetric (invariant to translation in the z-direction susceptible to rotational cryptanalysis)
- *All rounds would have been the same without it.*

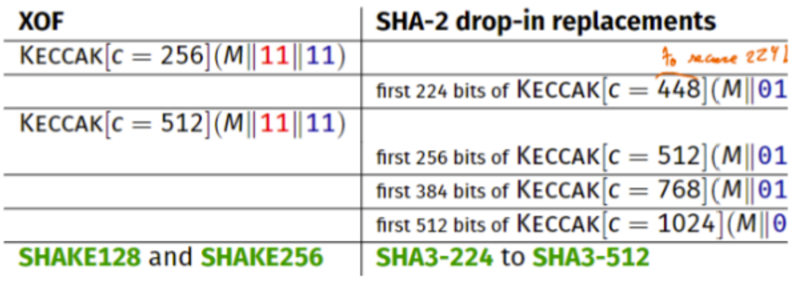
Round summary

$$R = \iota \circ \chi \circ \pi \circ \rho \circ \theta$$

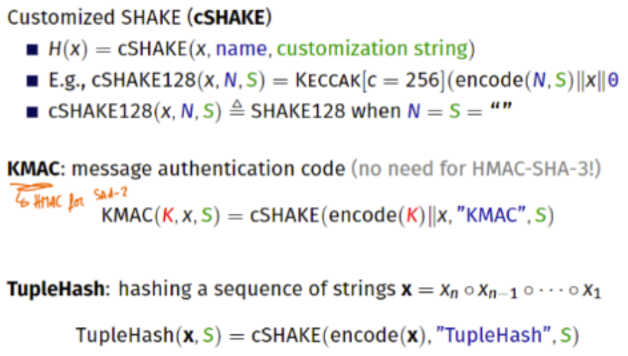
R is the #round-function .
There are $12 + 2l$ rounds:
- KECCAK-f$[25]$ has 12 rounds
- KECCAK-f$[1600]$ has 24 rounds

3.4.2 NIST FIPS 202

Four drop in replacements to SHA-2 (two #XOF)



XOF	**SHA-2 drop-in replacements**
$\textsc{Keccak}[c = 256](M\|11\|11)$	to secure 224!
	first 224 bits of $\textsc{Keccak}[c = 448](M\|01$
$\textsc{Keccak}[c = 512](M\|11\|11)$	first 256 bits of $\textsc{Keccak}[c = 512](M\|01$
	first 384 bits of $\textsc{Keccak}[c = 768](M\|01$
	first 512 bits of $\textsc{Keccak}[c = 1024](M\|0$
SHAKE128 and **SHAKE256**	**SHA3-224** to **SHA3-512**



Customized SHAKE (**cSHAKE**)

- $H(x) = \text{cSHAKE}(x, \text{name}, \text{customization string})$
- E.g., $\text{cSHAKE128}(x, N, S) = \textsc{Keccak}[c = 256](\text{encode}(N, S)\|x\|0$
- $\text{cSHAKE128}(x, N, S) \triangleq \text{SHAKE128}$ when $N = S =$ “”

KMAC: message authentication code (no need for HMAC-SHA-3!)

→ HMAC for SHA-2

$$\text{KMAC}(K, x, S) = \text{cSHAKE}(\text{encode}(K)\|x, \text{"KMAC"}, S)$$

TupleHash: hashing a sequence of strings $\mathbf{x} = x_n \circ x_{n-1} \circ \cdots \circ x_1$

$$\text{TupleHash}(\mathbf{x}, S) = \text{cSHAKE}(\text{encode}(\mathbf{x}), \text{"TupleHash"}, S)$$