Security Analysis of Extended Sponge Functions

Hash functions in cryptology: theory and practice
Leiden, Netherlands

Thomas Peyrin

Orange Labs

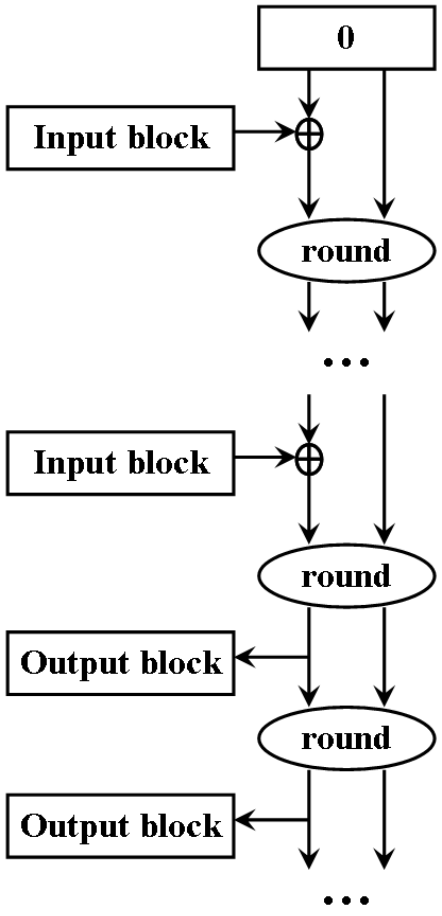
University of Versailles

June 4, 2008

Outline

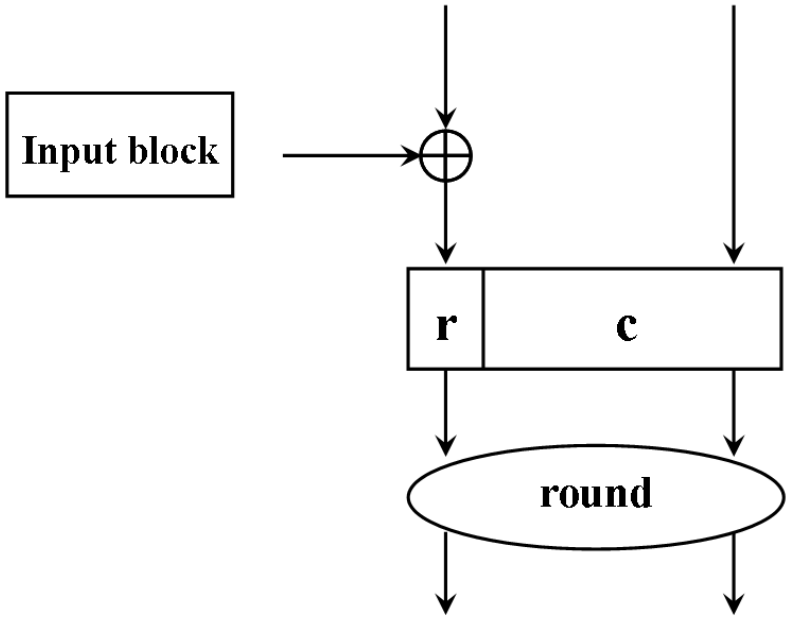
Outline

Sponge Functions



- sponge functions: a new iterative hash function (or stream cipher) framework.
- introduced by Bertoni, Daemen, Peeters and Van Assche in 2007.
- **idea:** absorb the message blocks (with padding) and squeeze the hash output blocks.
- use a unique fixed length round transformation (or a permutation).
- handy: variable output length.

Sponge Functions



- **c** represents the capacity.
- **r** represents the bit-rate.

Security Bounds (Ecrypt Hash Workshop 2007)

- **black box model**: the attacker has no access to the internal round function.
- the random sponge model is similar to the random oracle model (but also takes in account the internal finite state).
- when the round function is a random transformation (or permutation), the random sponge cannot be distinguished from a random oracle, unless there is an inner collision.
- it can be proven that generic attacks on a random sponge are as hard as on a random oracle, up to a limit of about $2^{c/2}$ in complexity (when the number of oracle queries is much lower than $2^{c/2}$, inner collisions are very unlikely).
- gives upper bounds for the attacks (in practice the attacker has access to the round function).

Security Bounds (Eurocrypt 2008)

- **white box model**: the attacker has access to the internal round function.
- use the indifferentiability framework from Maurer *et al.* (2007).
- **Theorem:** a random sponge can be differentiated from a random oracle only with probability $\simeq N(N+1)/2^{c+1}$, with $N < 2^c$, where N is the total number of calls to the internal round function.
- generic attacks require $2^{c/2}$.
- better results if the internal round function is a random permutation.
- gives lower bounds for the attacks.

Proof of Security

The proof has to be taken carefully:

- **for Merkle-Damgård**, we require a pseudo-collision resistant hash function. When one breaks the pseudo-collision resistance of the compression function, we consider the whole hash function as broken (even if no complete collision attack is found).
- **for sponge functions**, we require an ideal primitive that will operate on a big internal state. For practical reasons, the round function will be quite weak.
- the security assumptions for the internal function of RADIOGATÚN are clearly not met.
- the bounds give the designers an easy method to evaluate realistic security claims for a sponge function.

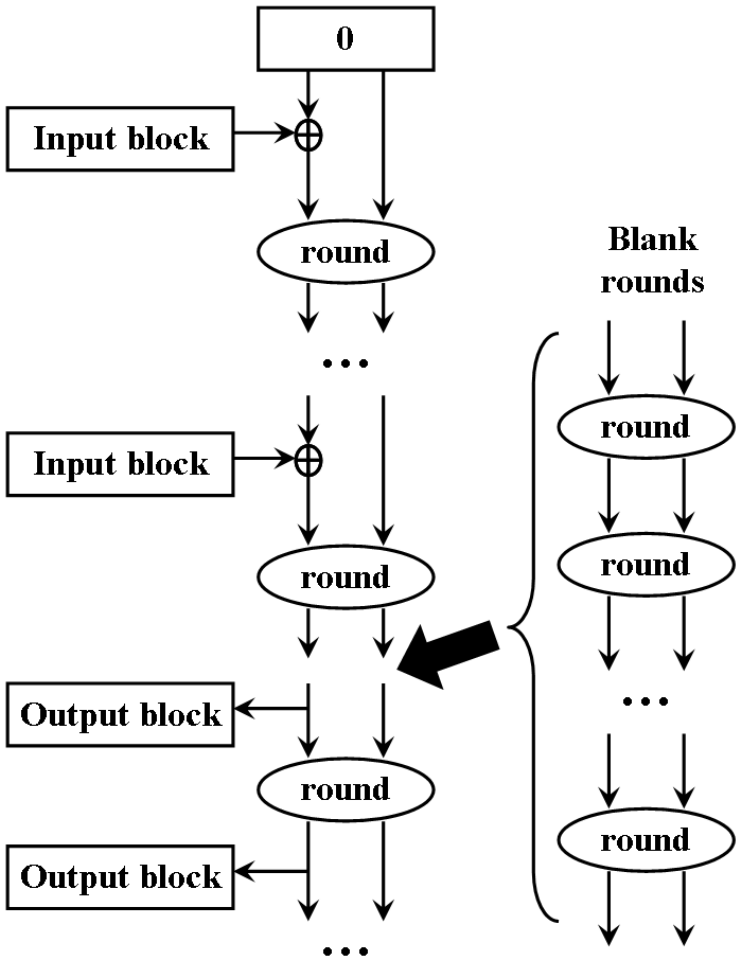
Why Sponge Functions ?

Sponge functions are interesting because:

- they allow new directions to build hash functions (no more MD-SHA-like or Merkle-Damgård constructions).
- they are built upon a permutation and not a compression function.
- they give an easy way to thwart the generic attacks on iterated hash functions such as multicollisions, long 2nd-preimages, ...

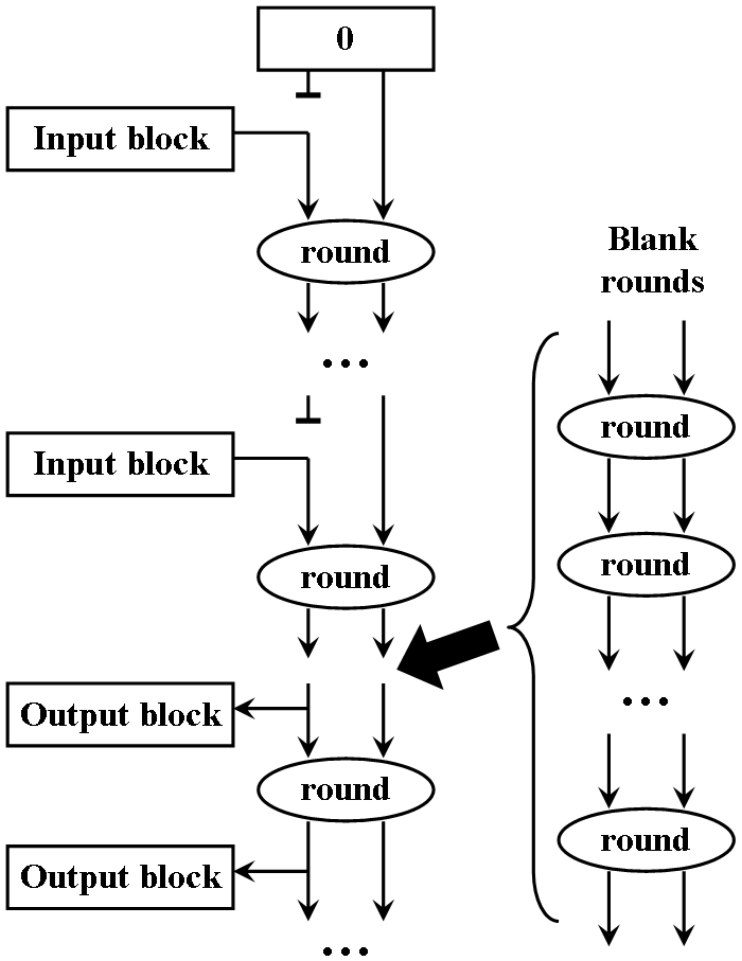
The idea of using a big internal state to avoid generic attacks on iterated hash functions was already pointed out by Joux and Lucks.

Practical Sponge Functions



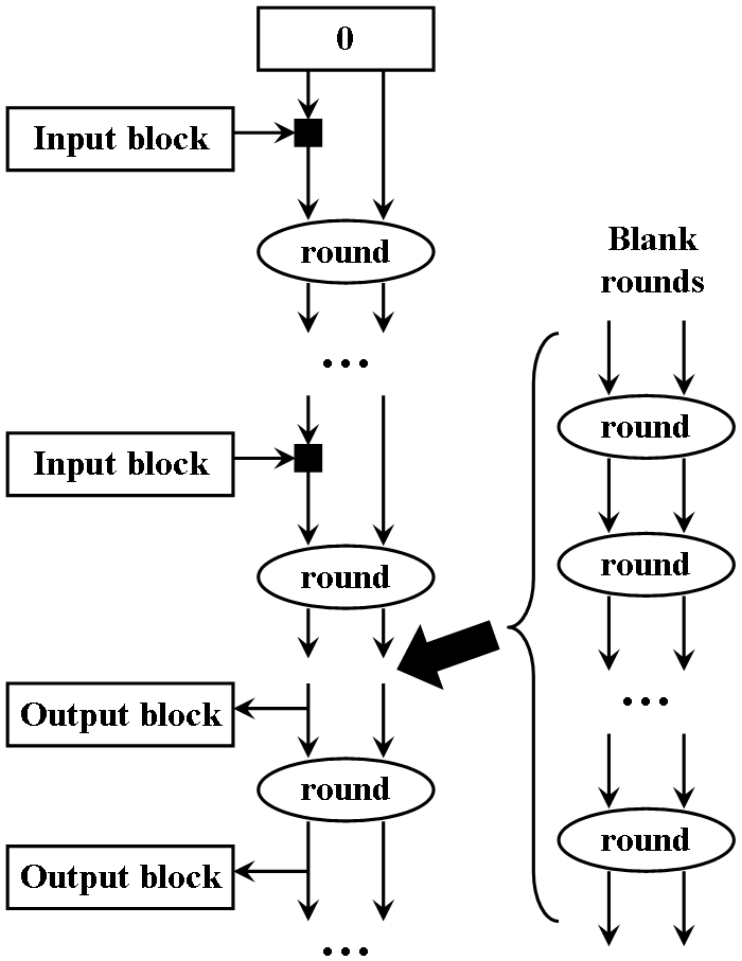
- in theory, the round function is ideal ... but not in practice.
- seems relatively ok for collision resistance but seems weak for preimage resistance.
- in practice, **we add blank rounds** (rounds without incoming message blocks).
- theory probably still applicable.

Extended Sponge Functions



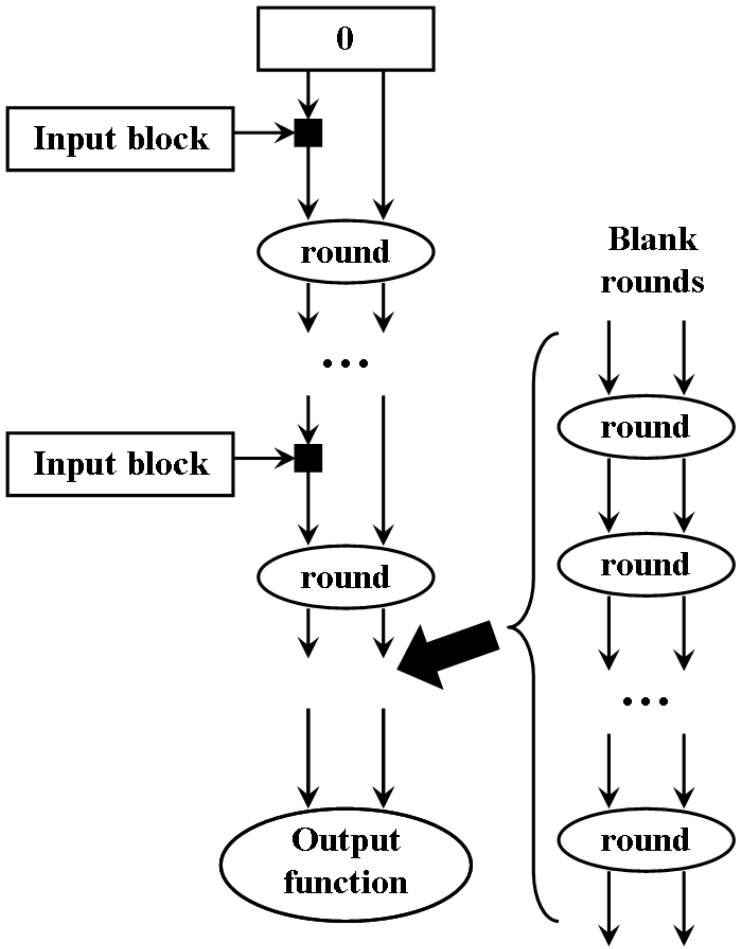
- we extend the original sponge functions framework.
- instead of XORing the message blocks to the internal state, **we allow them to overwrite the corresponding areas**.
- theory is no more applicable !
- an example : GRINDAHL.

Extended Sponge Functions



- we extend the original sponge functions framework.
- instead of XORing the message blocks to the internal state, **we allow them to overwrite the corresponding areas**.
- theory is no more applicable !
- an example : GRINDAHL.

Extended Sponge Functions



- we further extend the original sponge functions framework.
- instead of squeezing the sponge to get output blocks, **we can directly truncate it**.
- theory is no more applicable !
- an example : GRINDAHL.

Outline

Slide Attacks for Block Ciphers

- slide attacks were introduced for block ciphers by Biryukov and Wagner in 1999.
- **efficient against block cipher with a weak and periodic key schedule** (self-similarity of the cipher).
- independent of the number of rounds.
- allows to mount **distinguishing attacks** or even **key recovery attacks**.
- many improvements were later introduced.

Slide Attacks for Block Ciphers

A n-bit block cipher E with r rounds is split into b identical rounds of the same keyed permutation F^i for $i = \{1, \ldots, b\}$:

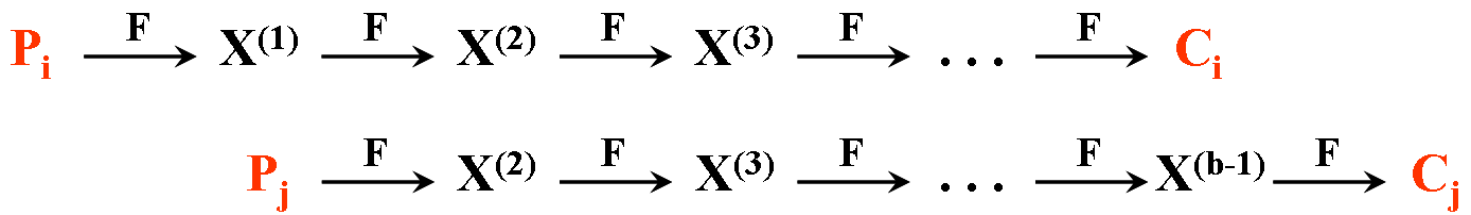
$$\begin{aligned} E &= F^1 \circ F^2 \circ \cdots \circ F^b \\ &= F \circ F \circ \cdots \circ F \end{aligned}$$

A plaintext P_j is then encrypted as:

$$P_j \xrightarrow{F} X^{(1)} \xrightarrow{F} X^{(2)} \xrightarrow{F} \ldots \xrightarrow{F} X^{(b-1)} \xrightarrow{F} C_j.$$

Slide Attacks for Block Ciphers

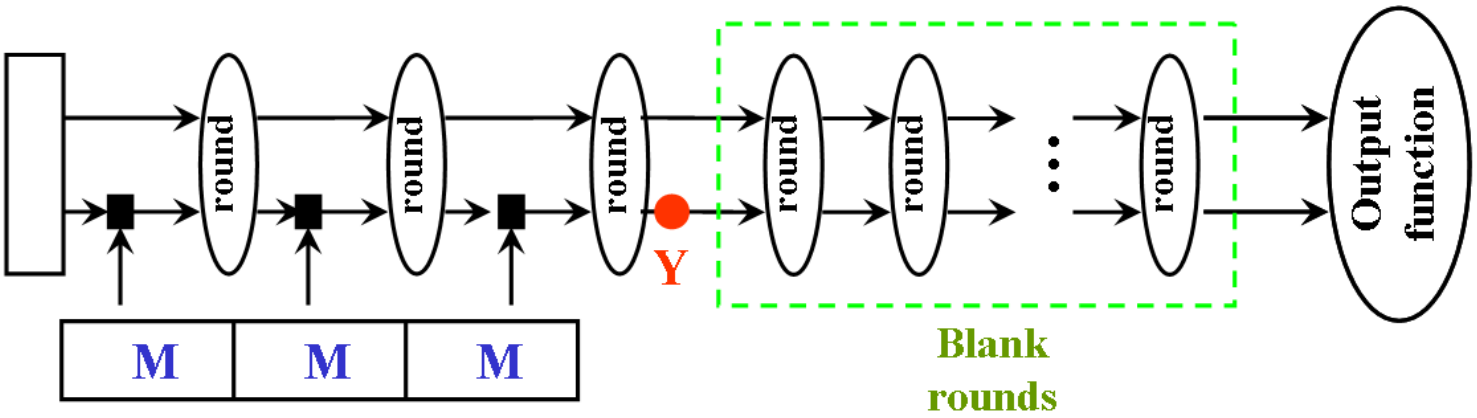
To mount a slide attack one has to find a slid pair of plaintexts (P_i, P_j), such that $P_j = F(P_i)$ and $C_j = F(C_i)$ holds.



With the birthday paradox, only $2^{n/2}$ plaintexts are required to find a slid pair.

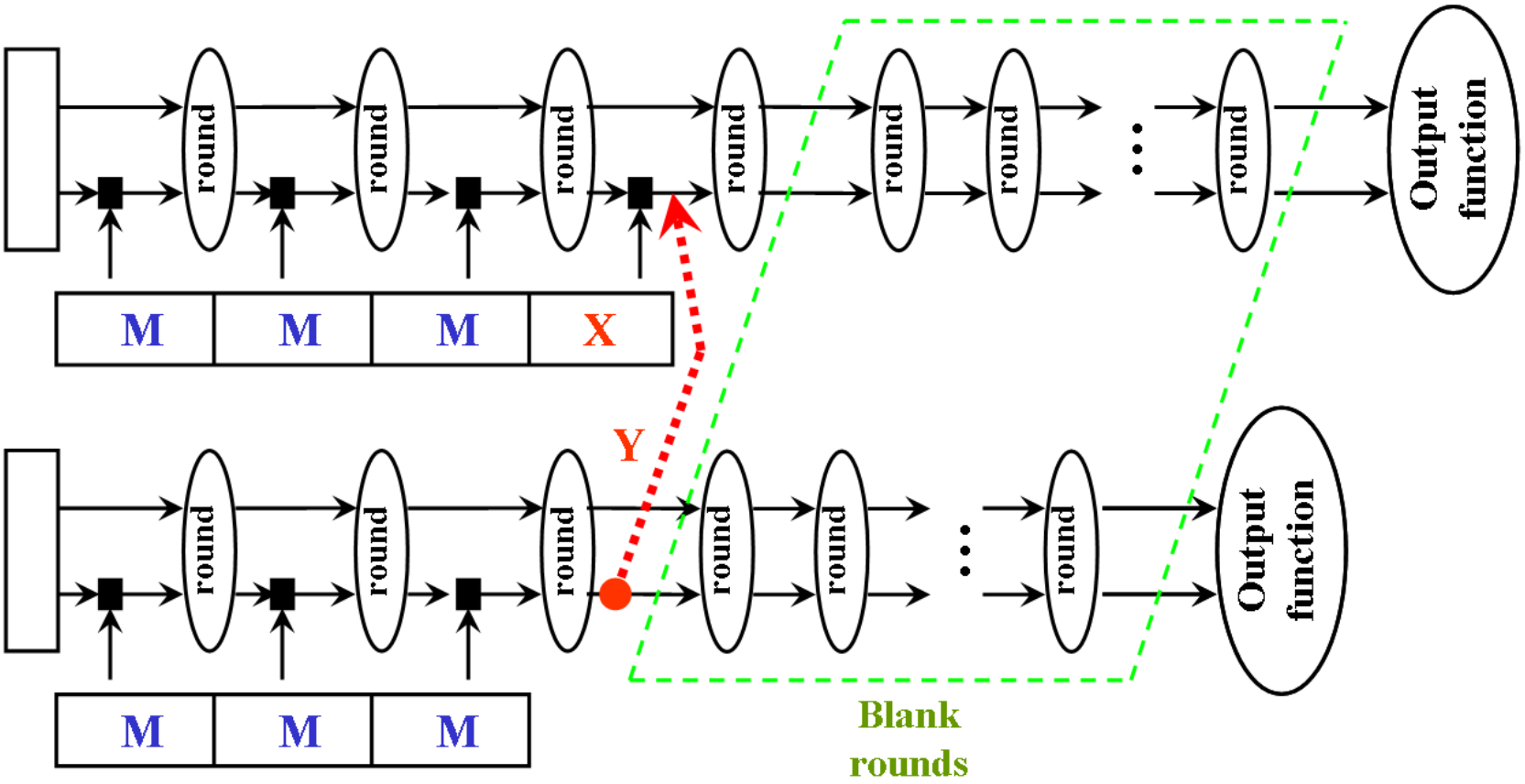
Application of slide attacks against hash functions were very few studied (Saarinen applied slide attacks against the inner cipher of SHA-1).

Slide Attacks on Sponge Functions



Slide Attacks on Sponge Functions

If the addition of X is neutral, then $output1 = round(output2)$.



Slide Attacks for Hash Functions

What can we obtain from slide attacks ?

- slide attacks are a typical block cipher cryptanalysis technique.
- doesn't seem useful for collision or preimage attacks ...
- ... but we can "distinguish" the hash function from a random oracle.
- the key recovery attack may also be useful if some secret is used in the hash function: we can attack a MAC construction using a hash function.

We'll try to attack the following MAC construction:

$$\mathrm{MAC}(K, M) = H(K||M).$$

Slide Attacks for Hash Functions

We'll try to attack the following MAC construction:

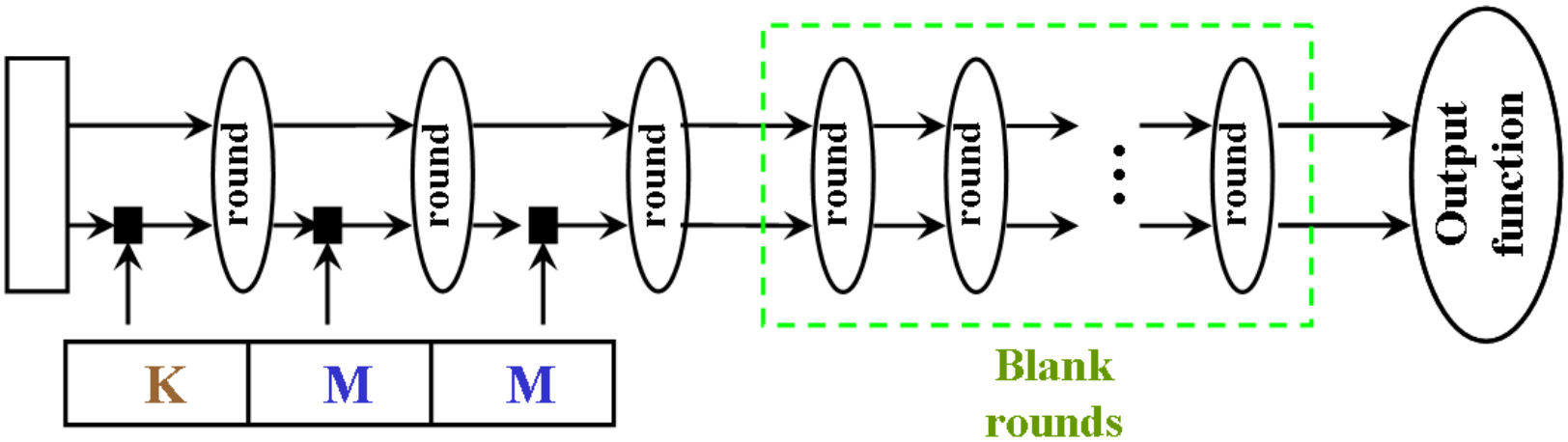
$$\mathrm{MAC}(K, M) = H(K||M).$$

- **... which is secure if the hash function is modeled as a random oracle.**
- **Merkle-Damgård already known to be weak against this construction**: given $\mathrm{MAC}(K, M) = H(K||M)$, compute $\mathrm{MAC}(K, M||Y) = H(K||M||Y)$ without knowing the secret key K.
- patch provided in Coron *et al.*'s paper from Crypto 2005.

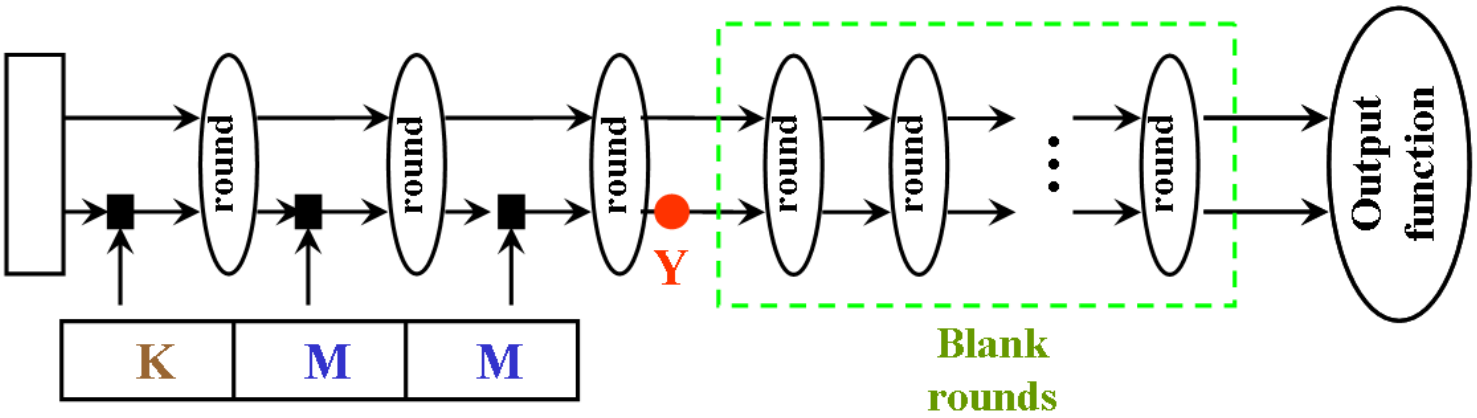
Why Slide Attacks for Sponge Functions

$$\mathrm{MAC}(K, M) = H(K||M).$$

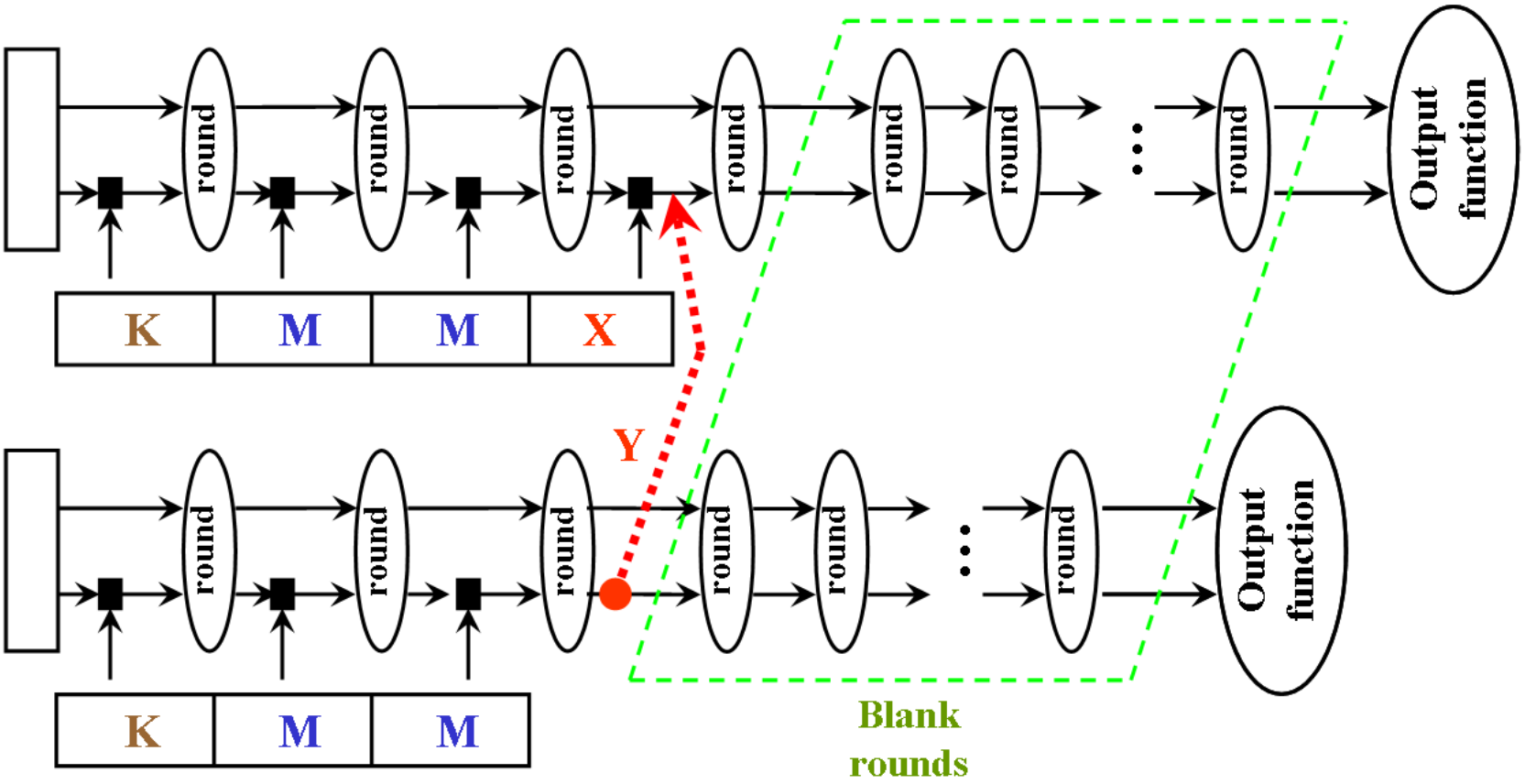
HMAC would be very slow with a sponge function, due to the blank rounds. Thus, the authors advised the following MAC construction:



Slide Attacks on Sponge Functions



Slide Attacks on Sponge Functions



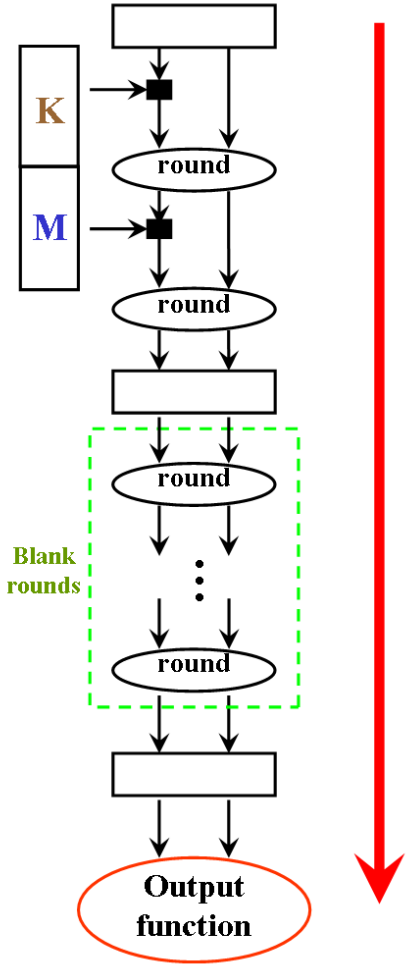
Slide Attacks for Sponge Functions

The Attack Scenario: the attacker makes queries M_i and receive replies $H(K||M)$. He then tries to get some non trivial information from the secret K or manage to forge another MAC with good probability.

The attack will be in three steps:

- Find and detect slid pairs of messages.
- Recover the internal state.
- Uncover some part of the secret key (or forge a new MAC).

The padding must also be taken in account !



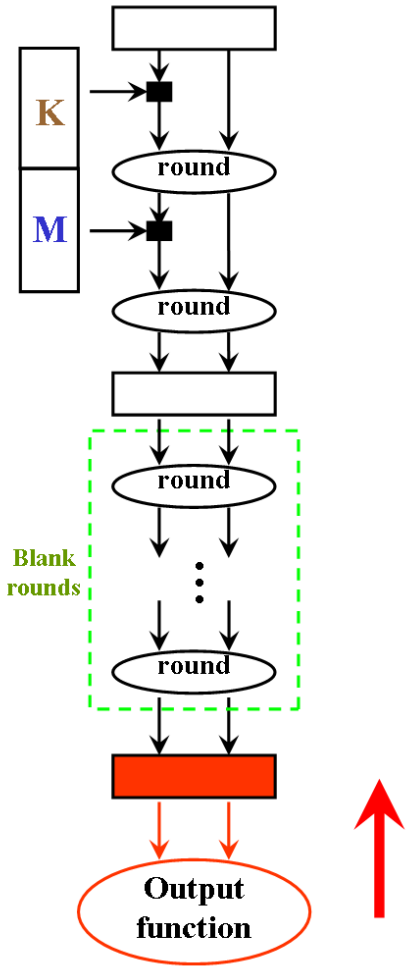
Slide Attacks for Sponge Functions

The Attack Scenario: the attacker makes queries M_i and receive replies $H(K||M)$. He then tries to get some non trivial information from the secret K or manage to forge another MAC with good probability.

The attack will be in three steps:

- Find and detect slid pairs of messages.
- Recover the internal state.
- Uncover some part of the secret key (or forge a new MAC).

The padding must also be taken in account !



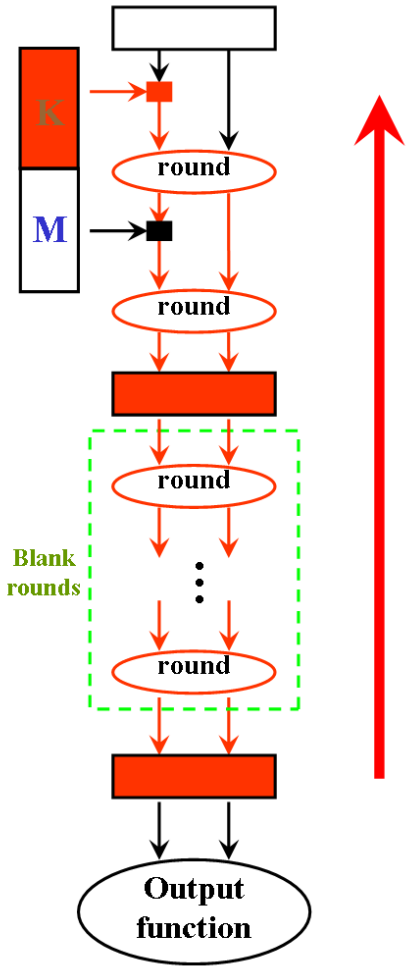
Slide Attacks for Sponge Functions

The Attack Scenario: the attacker makes queries M_i and receive replies $H(K||M)$. He then tries to get some non trivial information from the secret K or manage to forge another MAC with good probability.

The attack will be in three steps:

- Find and detect slid pairs of messages.
- Recover the internal state.
- Uncover some part of the secret key (or forge a new MAC).

The padding must also be taken in account !



Find and detect slid pairs of messages.

Find a slid pair of messages:

- depends on the message insertion function.
- impossible in the original sponge framework (in which the last inserted word must be different from 0) ...
- ... but possible if a different padding is used !
- possible if the insertion function overwrites the corresponding internal state words (as in GRINDAHL) with $P = 2^{-r}$.

Detect a slid pair of messages:

- depends on the output function.
- very easy with the sponge squeezing process (all the output words are shifted by one iteration position).
- more complicated with a direct truncation after the blank rounds.

Recovering the internal state and **uncovering the secret key** both depend on the whole hash function (require a case by case analysis).

Slide Attacks for Sponge Functions

Why not attacking

- HMAC ?
- or $\mathrm{MAC}(K, M) = H(M||K)$?
- or $\mathrm{MAC}(K, M) = H(K||M||K)$?

Because we need direct access to the last inserted word in order to get a slid pair.

Patches

It is very easy (and costless) for the designers to protect themselves against slide attacks.

If you're inserting message blocks with a XOR:

- just use exactly the sponge framework and make sure that the last inserted message work is different from zero.

If you're inserting message blocks by overwriting the corresponding internal state words:

- add a constant to the internal state just before the blank rounds to clearly separate them from the normal rounds.
- use a different transformation during the blank rounds.

GRINDAHL (Knudsen, Rechberger, Thomsen - 2007)

- 256-bit output (a 512-bit version is also defined).
- fits the framework of extended sponge functions.
- **based on AES:** faster than SHA-256 and low memory requirements (can benefit from the fast/small AES implementations).
- collision resistance, 2nd preimage and preimage resistance in $2^{n/2}$ function calls (possibility of meet-in-the-middle attacks for (2nd)-preimage).

GRINDAHL (Knudsen, Rechberger, Thomsen - 2007)

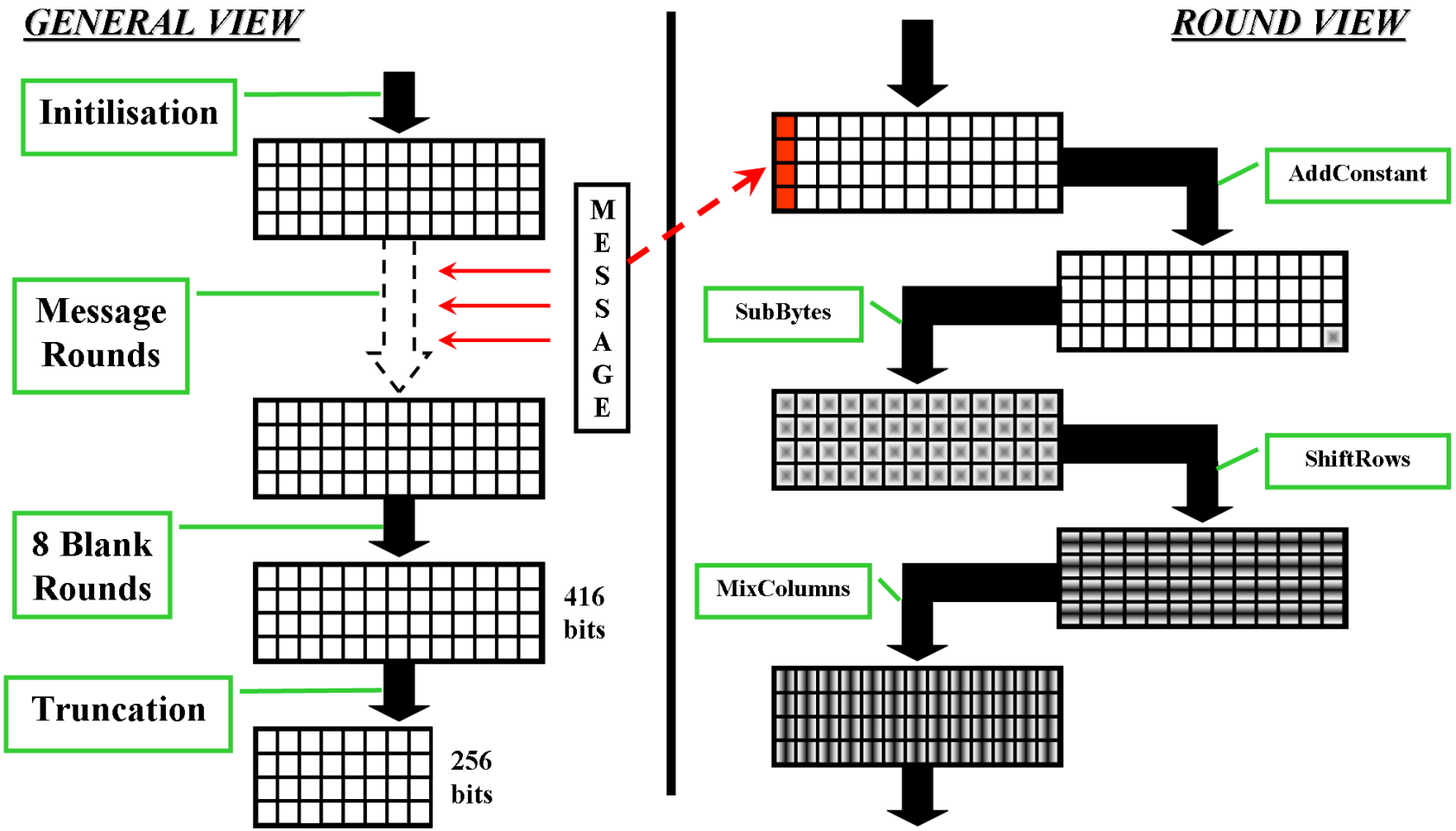
- use a **big internal state S**: 4×13 **matrix of bytes**.
- process 4 new bytes of message each round.
- $r = 4 \times 8 = 32$ bits and $c = 12 \times 4 \times 8 = 384$ bits.
- a round uses **Rijndael** parts: MixColumns, SubBytes, ShiftRows (with rotations 1, 2, 4, 10 for better diffusion) and AddRoundKey is replaced by the addition of a constant.
- **8 blank rounds** and then **truncation of S** for a 256-bit output.

GRINDAHL (Knudsen, Rechberger, Thomsen - 2007)

The whole process:

- **initialize** the internal state bytes with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **overwrite** the first column of S with 4 new message bytes.
  - do **AddConstant**
  - do **SubBytes**
  - do **ShiftRows**
  - do **MixColumns**
- do **8 blank rounds** without incoming message byte.
- **truncate S**: the output is the 8 first columns of the matrix.

High-level view of GRINDAHL



The padding of GRINDAHL

In GRINDAHL we have 10-padding and length-padding:

- **10-padding** appends a "1"-bit to the message, followed by as many "0"-bits as needed to complete the last message block.
- **length-padding** then appends the number of message blocks (not bits!) for the entire padded message as a 64-bit value (two blocks of message for GRINDAHL-256, one for GRINDAHL-512).

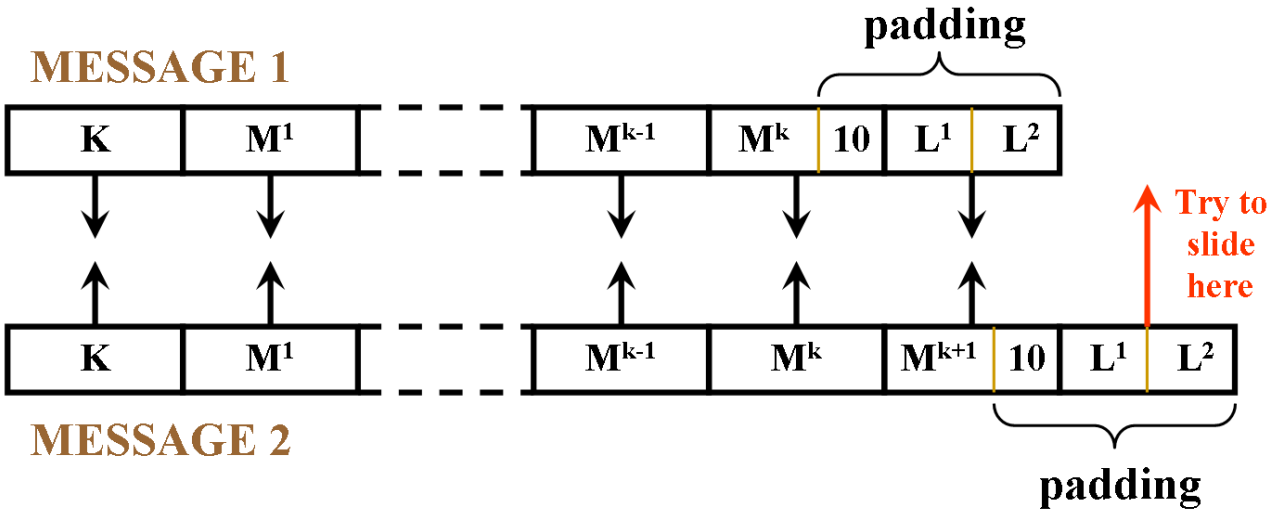
One effect of the 10-padding is that the last message block before the length-padding can be any value, except for the all-zero block.

Message $M = M^1||\ldots||M^l$ of 32-bit blocks M^i (already 10-padded) will be padded to

$$Pad(M) = M^1||\ldots||M^l||M^{l+1}||M^{l+2}.$$

Finding slid pairs for GRINDAHL

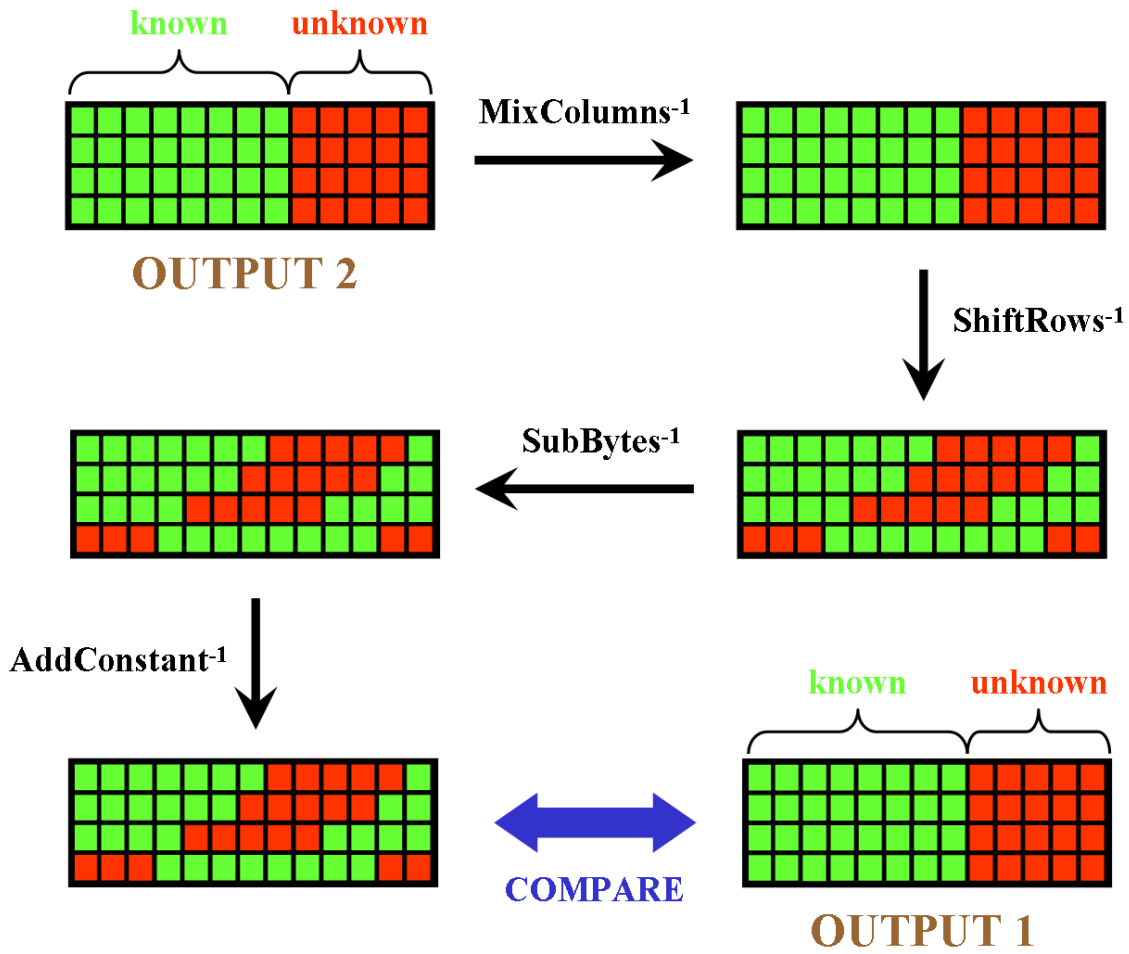
Suppose the length padding fits in one message block (true for 512-bit version, a little bit more complicated technique is required for the 256 version).



We have a probability of $2^{-r} = 2^{-32}$ to get a slid pair (for 512-bit version $P = 2^{-64}$).

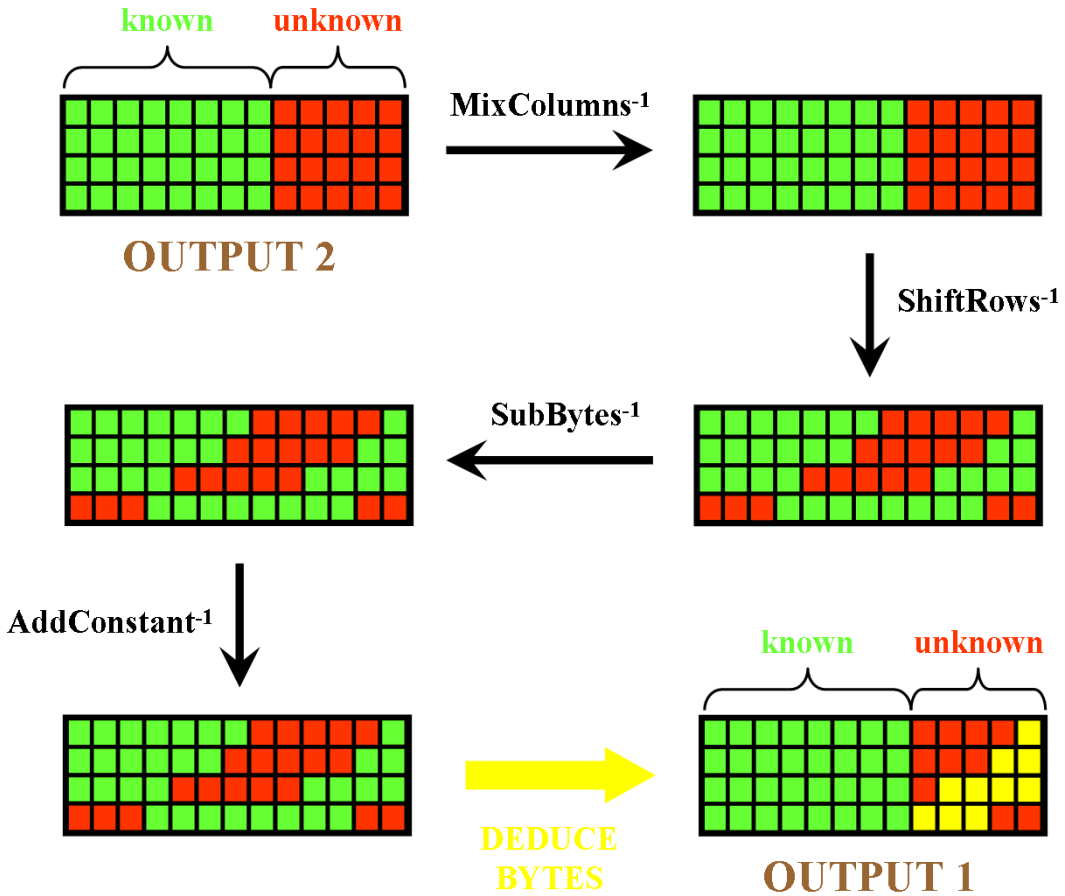
Detecting slid pairs for GRINDAHL

Assume two messages M_1 and M_2 that are slid. **How to detect them, just by looking at the corresponding outputs ?**



Recovering the internal state for GRINDAHL

Assume we found and detected a slid pair (M_1, M_2), **we already know the truncated internal state from M_1 and we'll try to recover the rest thanks to the truncated output of M_2.**



Once the internal state before the truncation is fully recovered, **one can completely invert the blank rounds**.

Uncovering some part of the secret for GRINDAHL

First note that **we can now forge valid MACs without even knowing the secret key** K.

In order to get some part of the secret, we need to invert the function. Inverting the blank rounds is easy, but inverting the message rounds is much more tricky: **at each iteration, we don't know what was overwritten by the message blocks**.

It is possible to deduce the overwritten columns by checking that we come to the good insertion of the known message blocks.

We continue this technique up to the first unknown block we meet: we can recover some bytes of the last **unknown** secret block.

More results

For GRINDAHL**-256**, the attack allows to:

- distinguish from RO with 2^{64} queries and computation time.
- forge valid MACs or to recover 1 new byte of the secret with 2^{64} queries and 2^{80} computations.

For GRINDAHL**-512:** the attack allows to (**first cryptanalytic results on this version**):

- distinguish from RO with 2^{64} queries and computation time.
- forge valid MACs or to recover 4 new bytes of the secret with 2^{64} queries and 2^{80} computations.

For RADIOGATÚN**: attack don't apply**, but would work on an overwrite version of it.

Outline

Properties of GRINDAHL

Main security arguments:

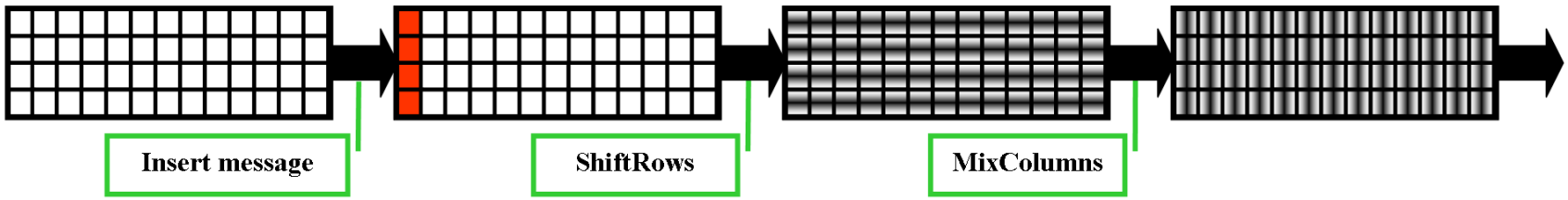
- a collision requires intermediate states with at least half of the bytes active.
- an internal collision requires at least 5 rounds.

It is very hard to find a low-weight and-or a small differential path for GRINDAHL.

Truncated differentials

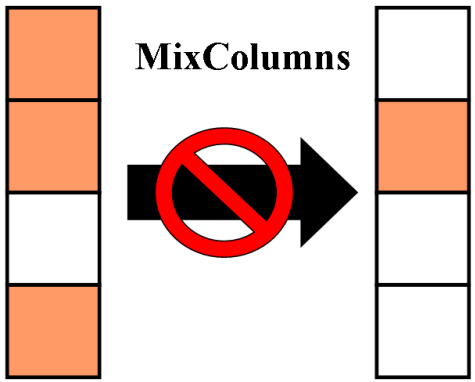
- the scheme is byte oriented.
- let's deal with **truncated differences**: only check if there is a difference in a byte, but don't care about the actual value of the difference.
- we can forget about SubBytes and the constant addition (transparent for truncated differentials).
- *we only deal with ShiftRows, MixColumns and truncation.*

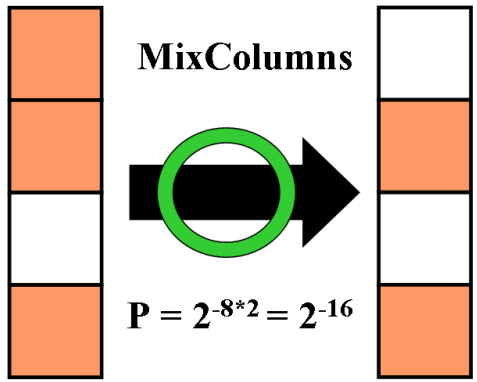
The simplified scheme we consider:



The MixColumns function

- How do the truncated differentials react with the MixColumns function ?
- **Property of MixColumns:**
  $\sharp\{\text{input byte-differences}\} + \sharp\{\text{output byte-differences}\} \geq 5$.
- **P[valid transitions] =** $2^{-8\times(4-\sharp\{\text{output byte-differences}\})}$.



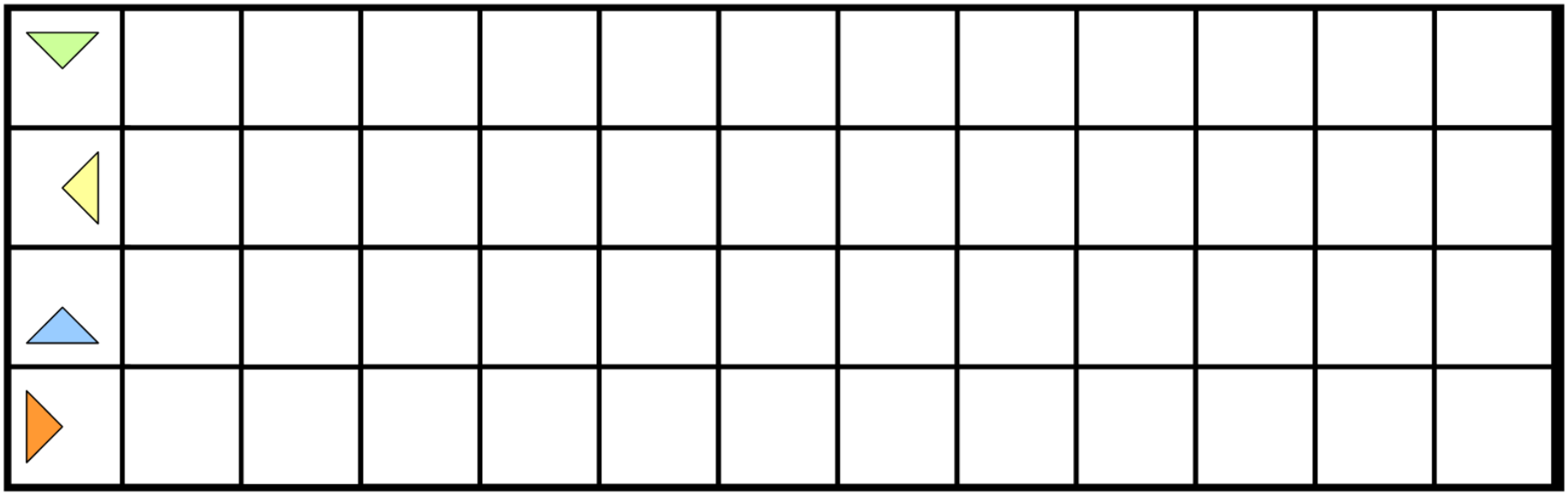


The control bytes (1)

- ShiftRows modified (1, 2, 4, 10) for better diffusion: every state byte depends on every message byte after 4 rounds.
- ... but what happens before those 4 rounds ?
- each message byte inserted affect some subset of the internal state S.
- **this will allow us to control a little bit the difference spreading by forcing some MixColumns differential transitions independently**.
- we call them **control bytes**.

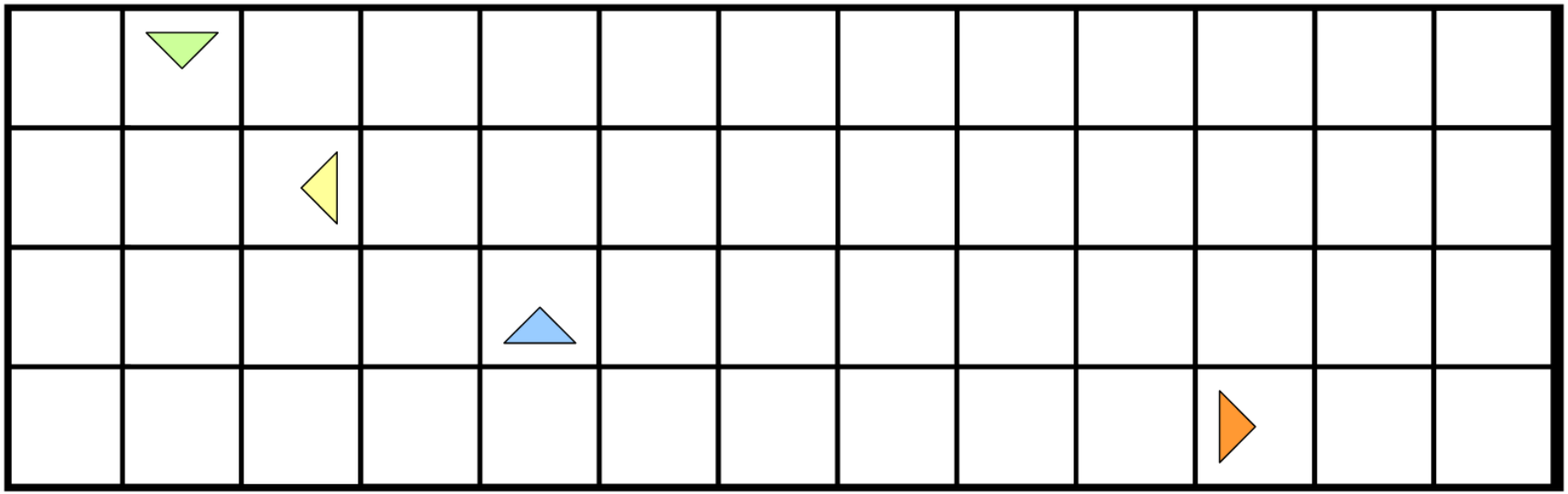
The control bytes (2)

- **Insert the message bytes.**



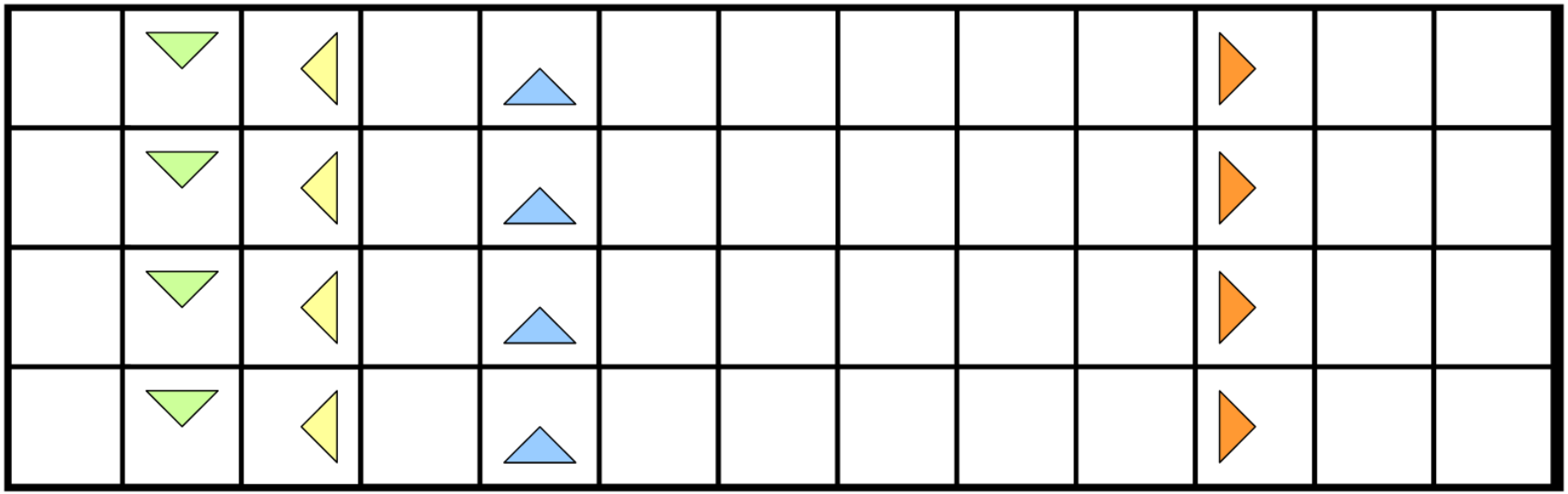
The control bytes (2)

- Do **ShiftRows** (1^{st} round).



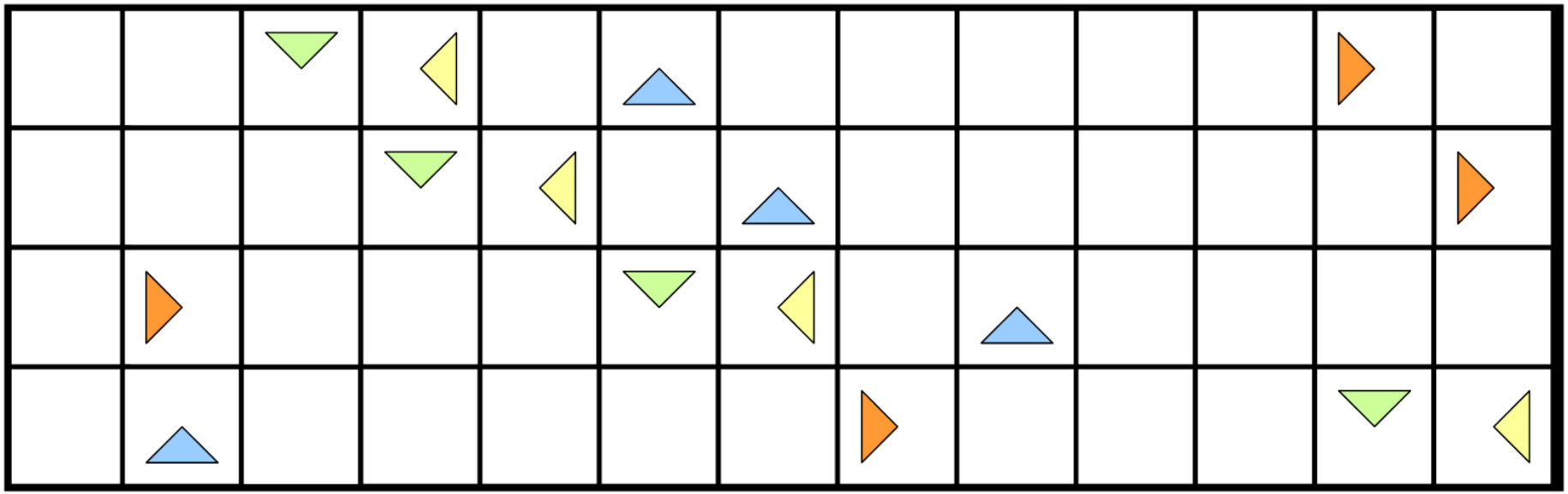
The control bytes (2)

- Do **MixColumns** (1^{st} round).



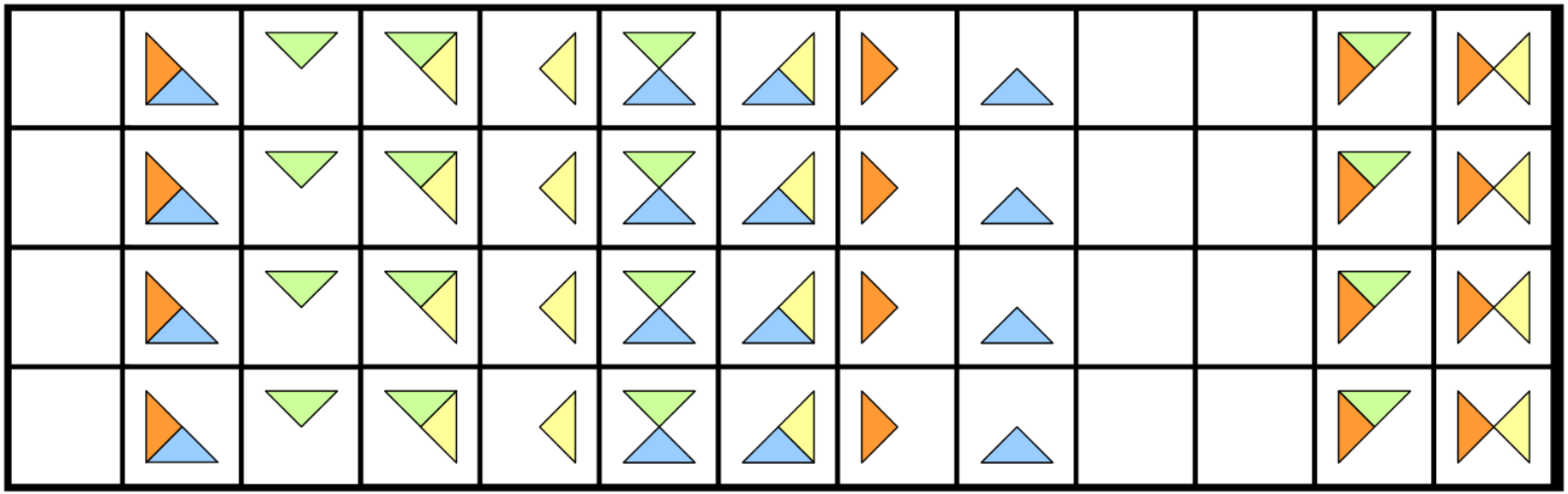
The control bytes (2)

- Do **ShiftRows** (2^{nd} round).



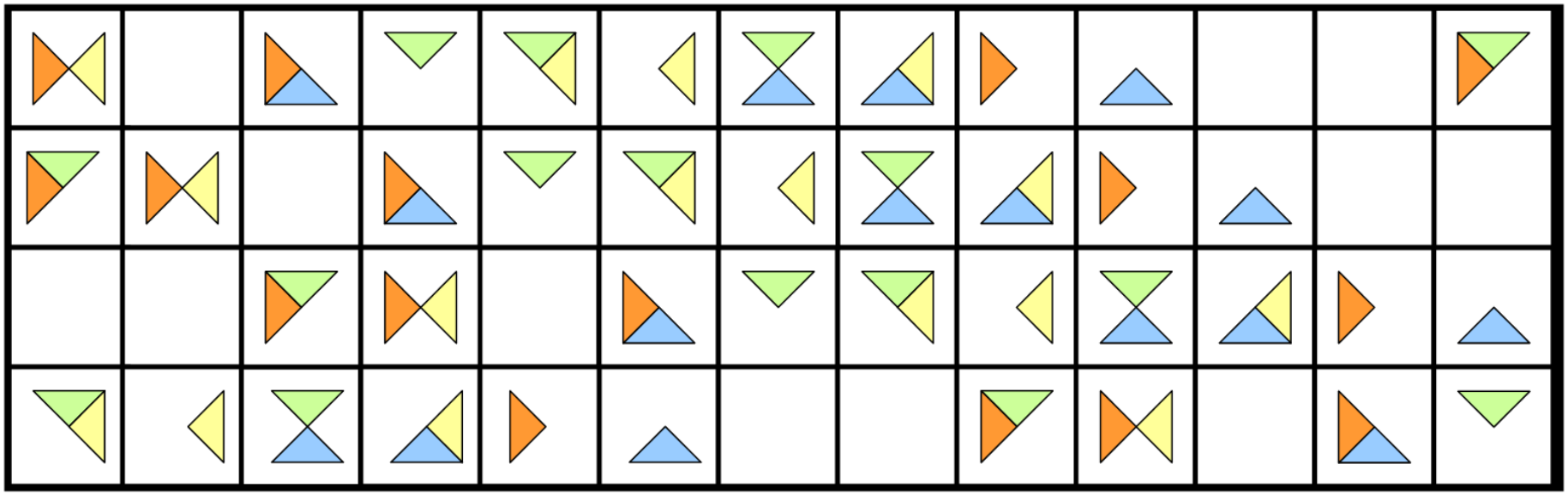
The control bytes (2)

- Do **MixColumns** (2^{nd} round).



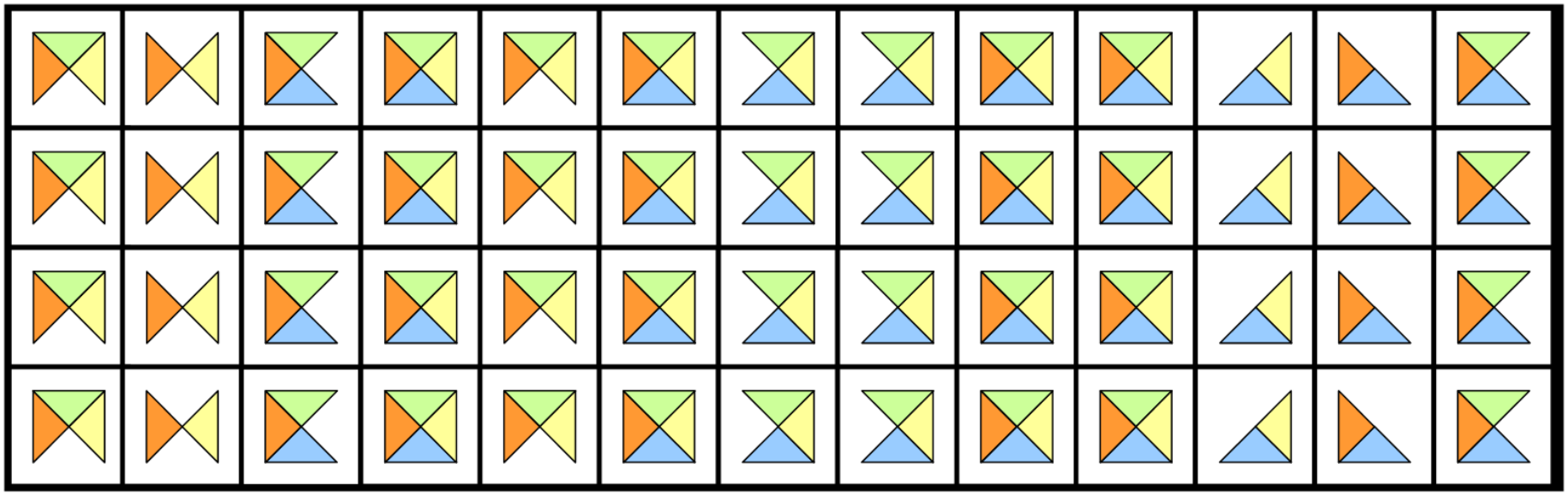
The control bytes (2)

- Do **ShiftRows** (3^{rd} round).



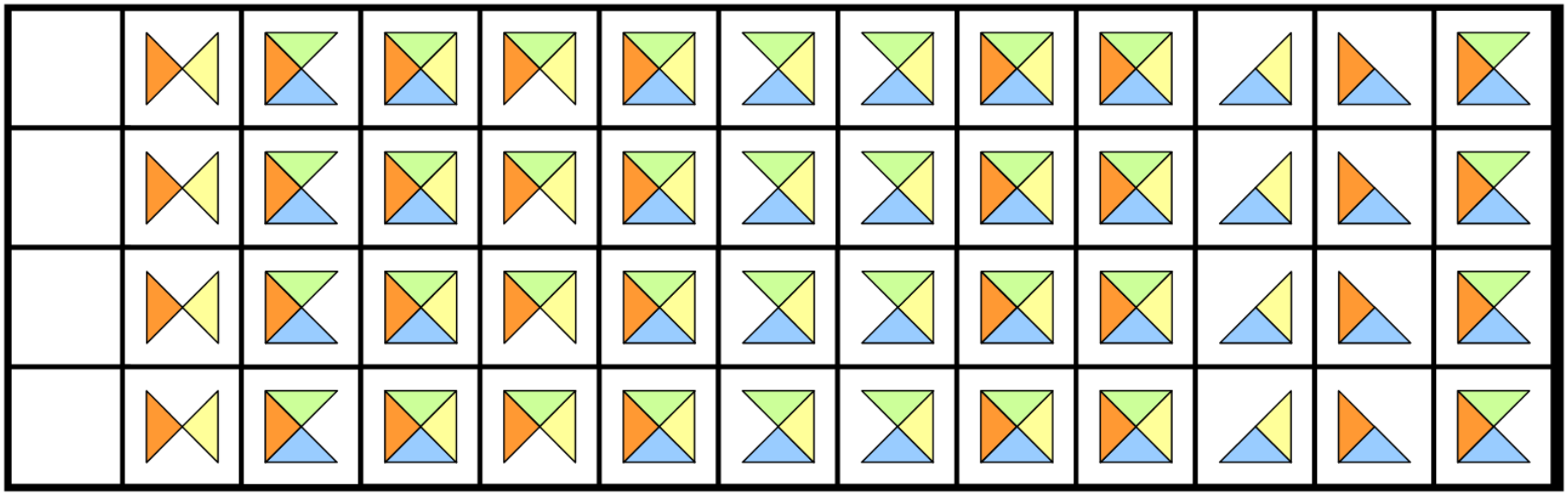
The control bytes (2)

- Do **MixColumns** (3^{rd} round).



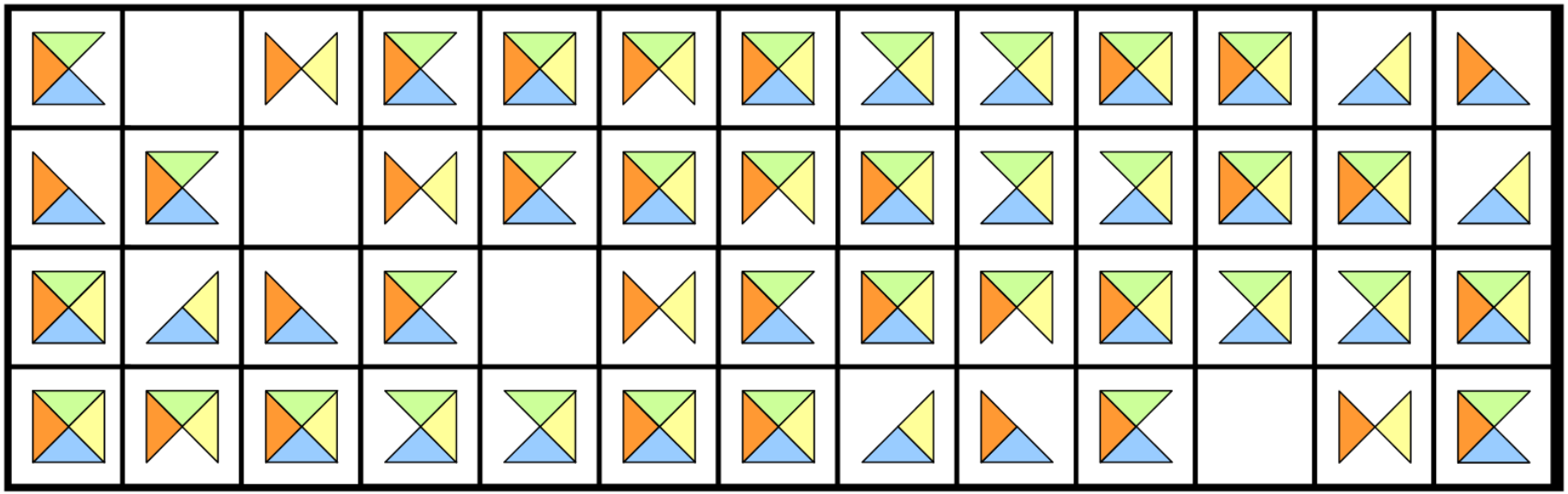
The control bytes (2)

- **Truncation of the first column** (new message bytes).



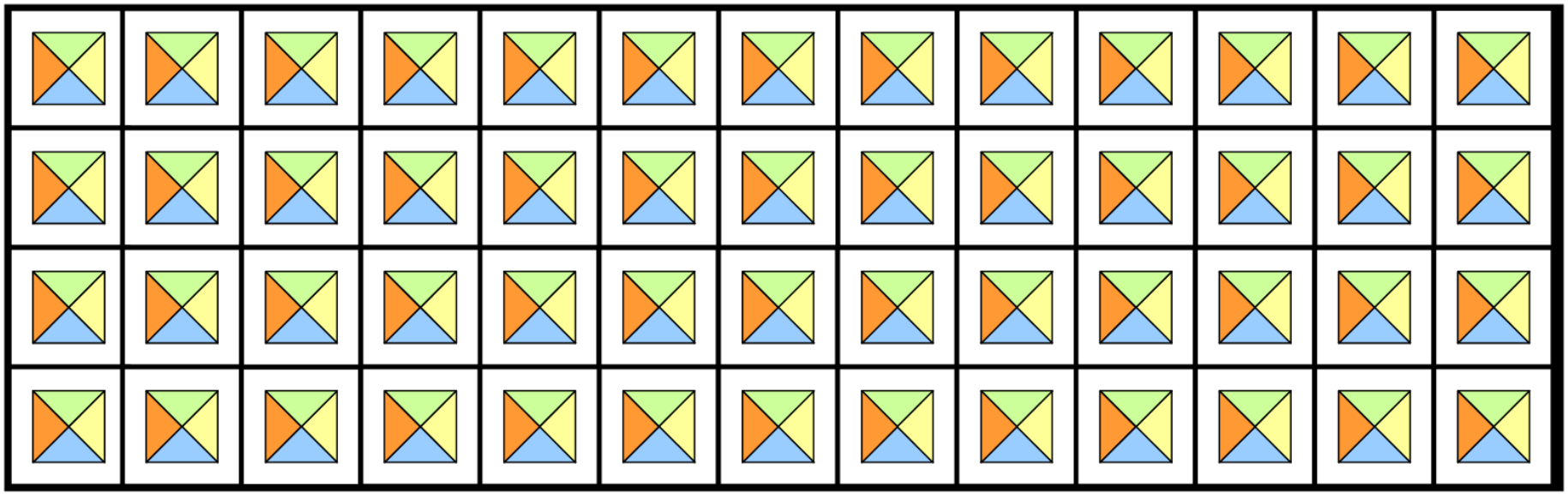
The control bytes (2)

- Do **ShiftRows** (4^{th} round).



The control bytes (2)

- Do **MixColumns** (4^{th} round).



Internal collisions are better

- 2 possibilities for a collision: internal or not.
- the blank rounds would make things really hard since we have no more control (no more message byte inserted).
- an **internal collision** seems easier, even if we can not use the final truncation anymore (we'll have a bigger internal state to make collide).
- **2 possibles ways to erase a truncated difference**: with a MixColumns transition (for a cost P^{-1}) or thanks to the overwriting during a message insertion (no cost since already planed in the differential path).

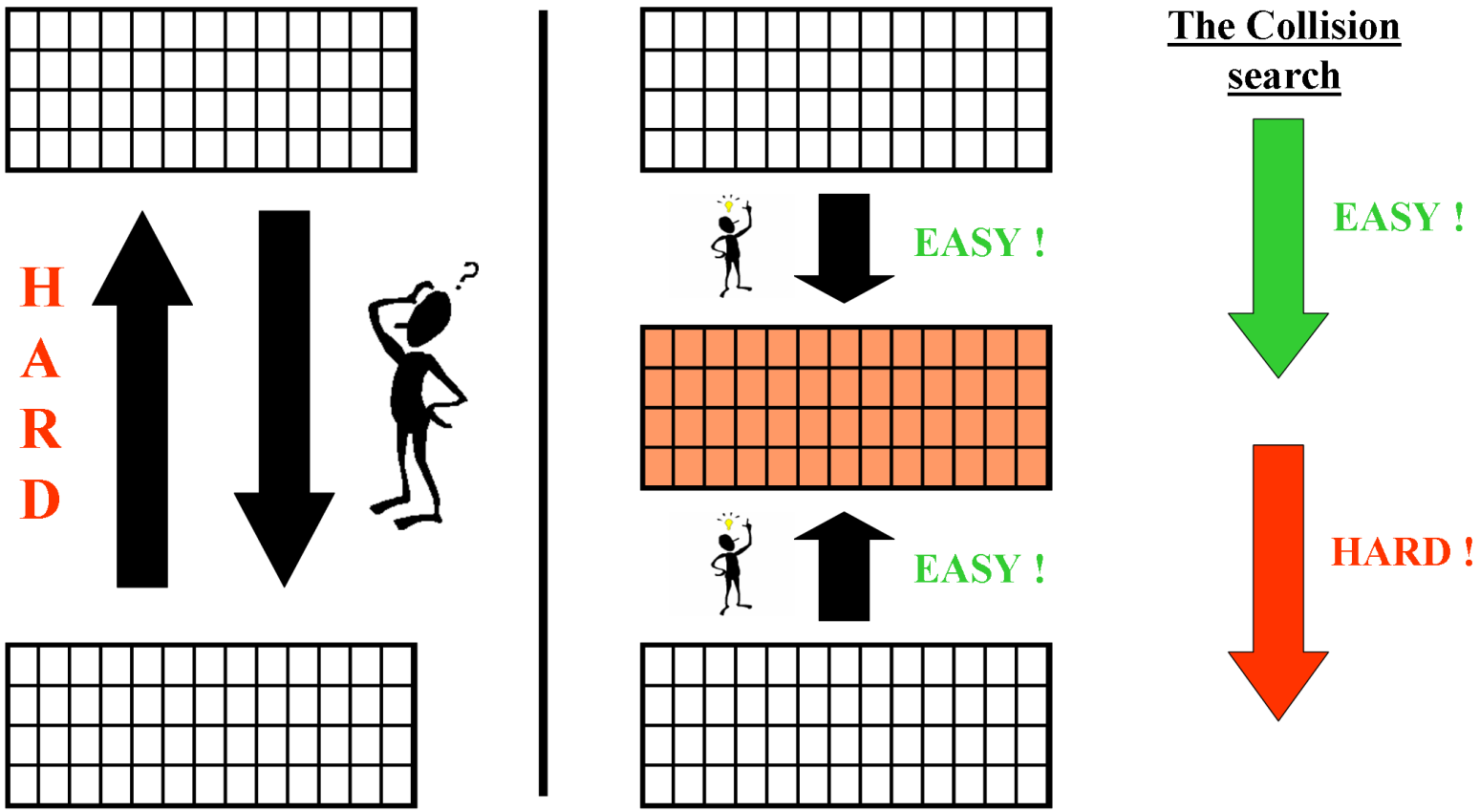
An unintuitive strategy

- Building a differential path is really hard because of the two security properties.
- **idea - take the all-difference state as a check point:**
  - from a no-difference state to an all-difference state: hopefully very easy ! No need for a differential path here.
  - from an all-difference state to a no-difference state: harder ! Build the differential path backward and search for a collision onward.
- the costly part is obviously the second stage !

That is an unintuitive strategy for a hash function cryptanalyst: we deliberately let all the differences spread in the whole state before beginning the collision search !

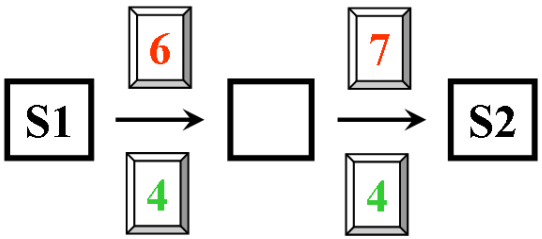
How to build a differential path

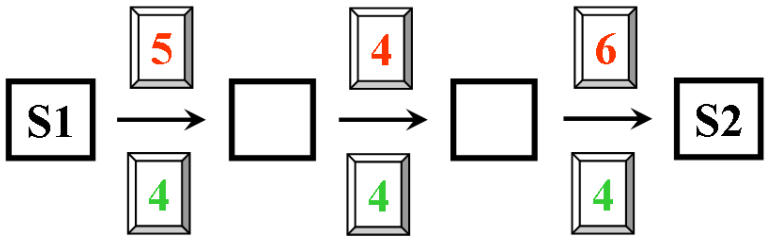
Building a differential path is really hard !



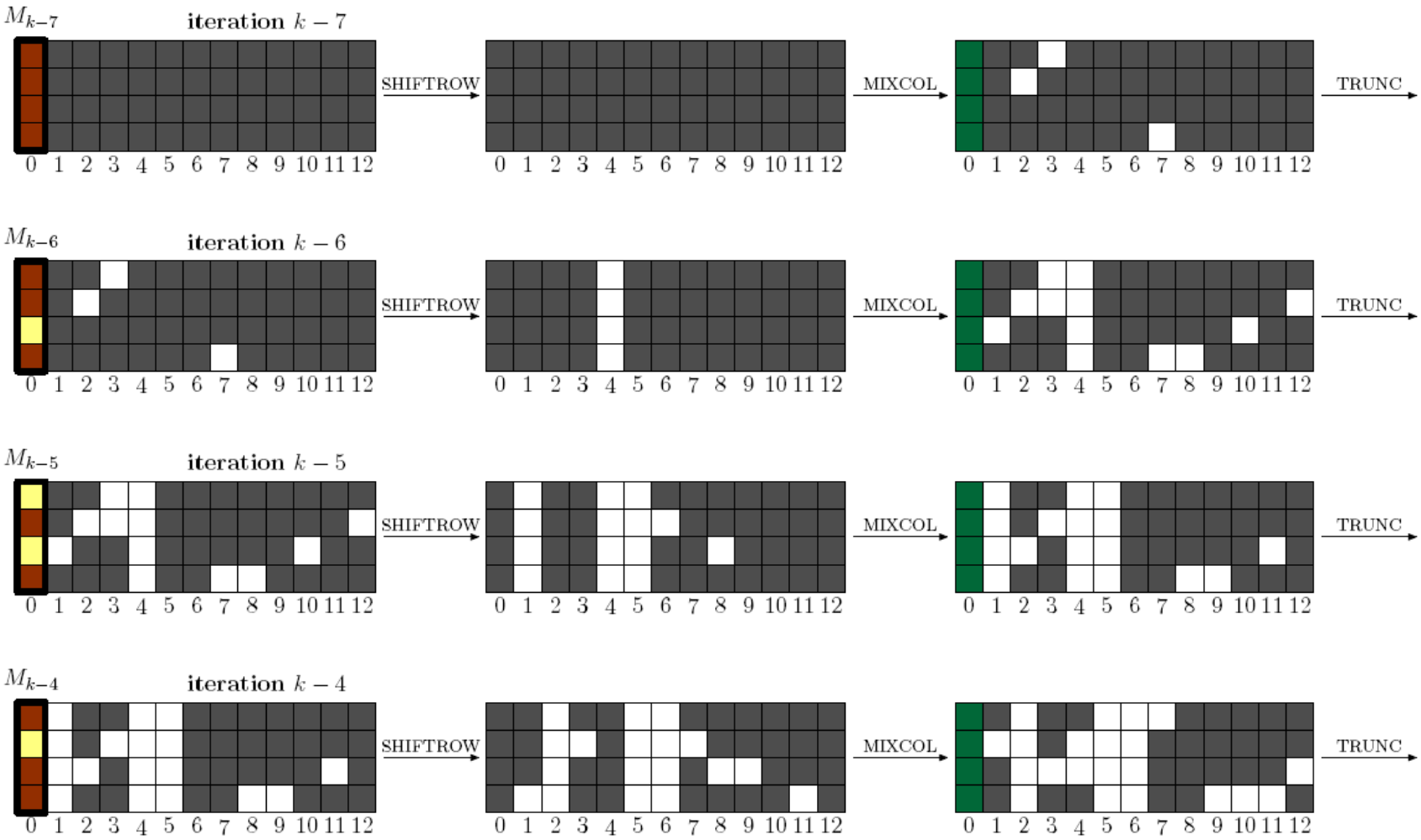
Differential path and control bytes

- several differential paths are possibles.
- some give better probability of success than others ... but we will use the control bytes to force some MixColumns independently.
- **dilution effect: it may be better to use less probable paths but longer ones (more message/control bytes gained than probability decrease).**
- this whole differential path trade-off search can be automated.

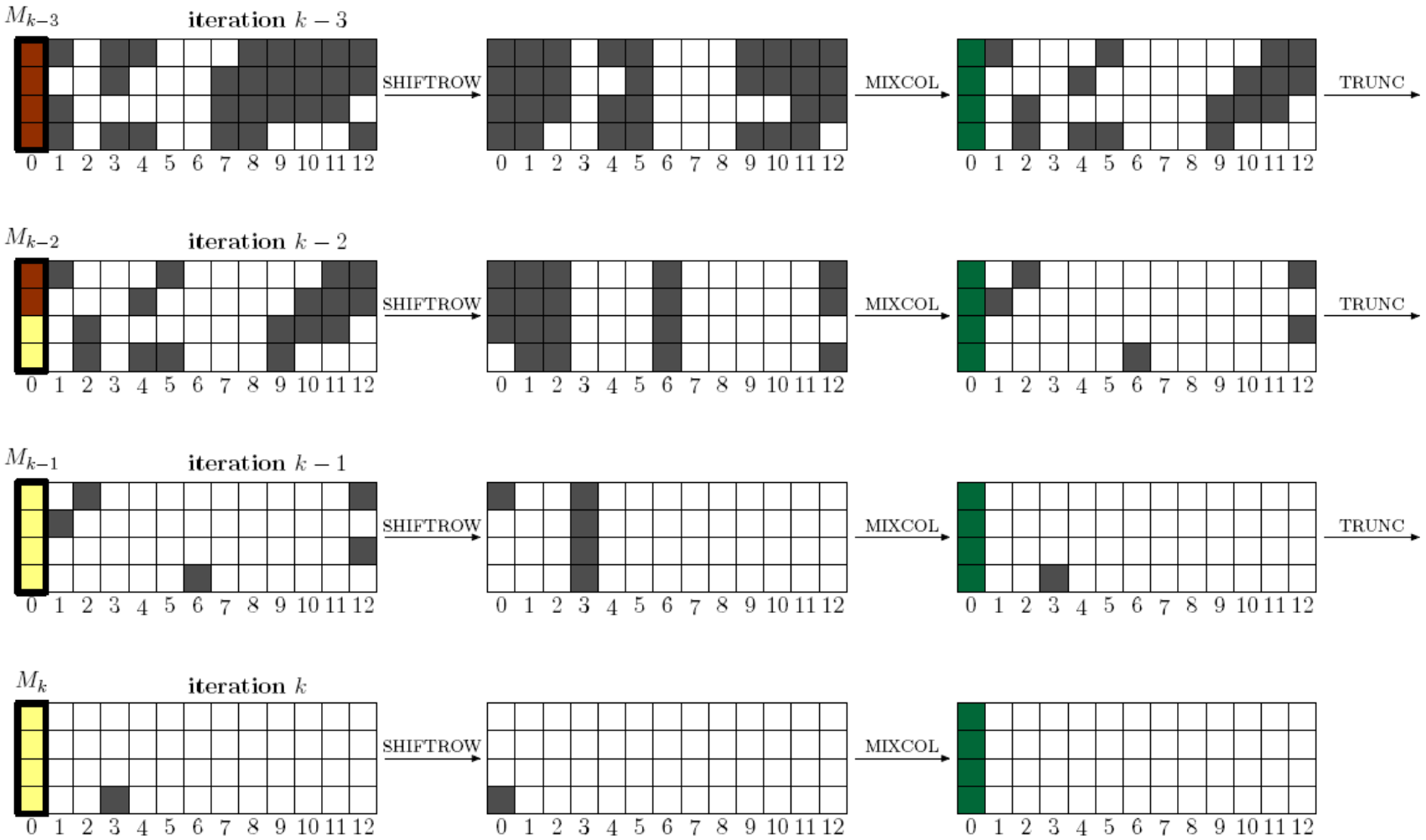




Our truncated differential path (1)



Our truncated differential path (1)



The collision attack

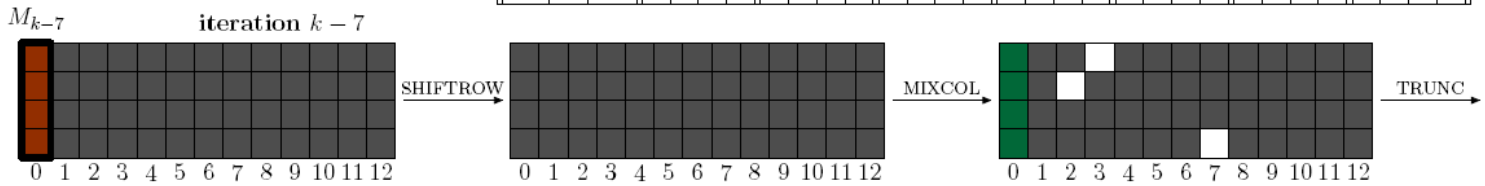
The attack is in three steps:

- 1^{st} **step:** reach an all-difference state (for example by adding a lot of differences very quickly) and generate $K = 2^{112}$ other all-difference states from it.
  - P[all-difference state to all-difference state]$\simeq 2^{-0,27}$.
- 2^{nd} **step:** for each all-difference state, check if one can find a message pair following the differential path.
  - P[without control bytes]=2^{-440}.
  - P[with control bytes]=2^{-112}.
- 3^{rd} **step:** once a valid message pair found, add a random message block without difference in order to force the first column overwriting in the last step.

Choosing the message bytes

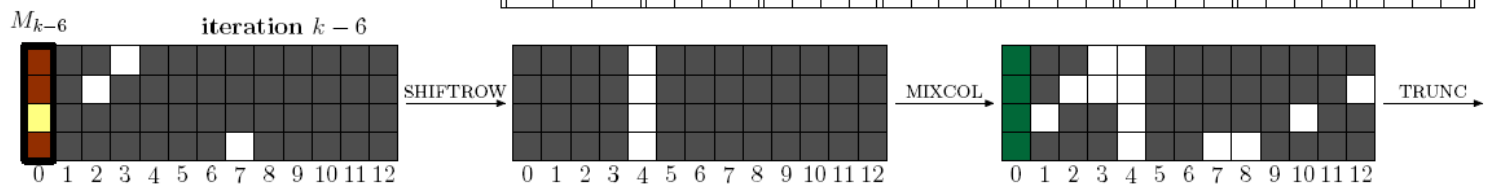
1st block

it	col	cost	message blocks inserted																							
			$k-8$				$k-7$				$k-6$				$k-5$				$k-4$				$k-3$			
			A	B	C	D	A	B	C	D	A	B	c	D	a	B	c	D	A	b	C	D	A	B	C	D
k-7	2	1	–					×																		
	3	1	×	×																						
	7	1				×																				



2nd block

it	col	cost	message blocks inserted																							
			$k-8$				$k-7$				$k-6$				$k-5$				$k-4$				$k-3$			
			A	B	C	D	A	B	C	D	A	B	c	D	a	B	c	D	A	b	C	D	A	B	C	D
k-6	1	1		–		–			–	–	×															
	2	1	–		–	–	–					×														
	3	2	–	–	–	–	×	×																		
	7	1	–	–	–					×																
	8	1	–	–	–	–			×																	
	10	1		–	–									×												
	12	1	–		–	–		×		×																



Outline

RADIOGATÚN (Bertoni *et al.* - 2007)

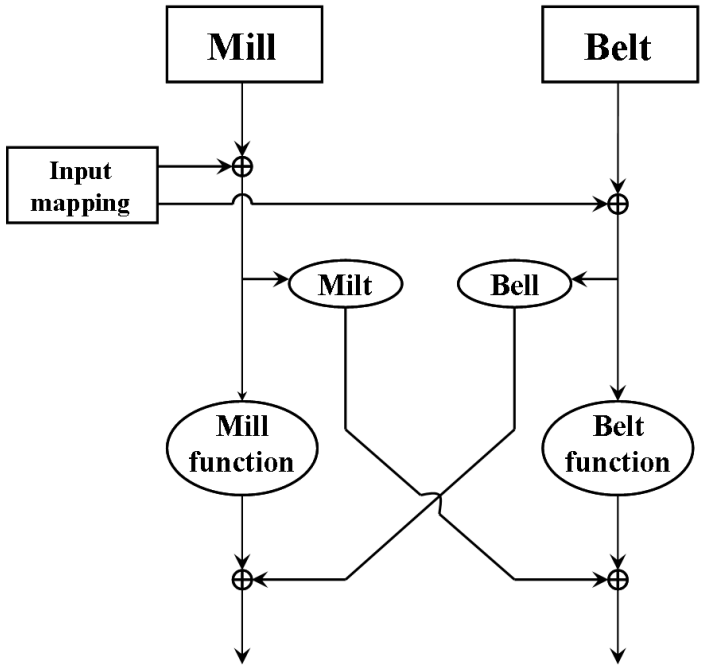
- fits the original framework of sponge functions.
- variable output length with squeezing process.
- RADIOGATÚN-*w* stands for RADIOGATÚN with words of *w* bits.
- RADIOGATÚN-32 faster than SHA-1 or SHA-256, but requires more memory (big internal state).

RADIOGATÚN (Bertoni *et al.* - 2007)

- use a **big internal state S** of 58 words of w bits ...
- ... divided in two parts: the Mill and the Belt.
- process 3 words of w bits of message each round.
- $r = 3 \times w$ bits and $c = 55 \times w$ bits.
- **basic idea:** make hashing fast by applying a strong function on the Mill part only.
- **16 blank rounds** and then extract output words.

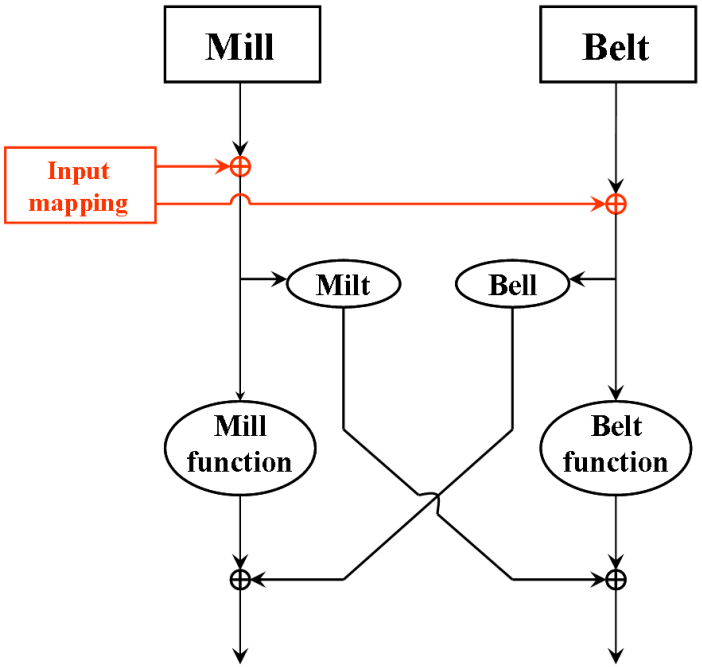
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.



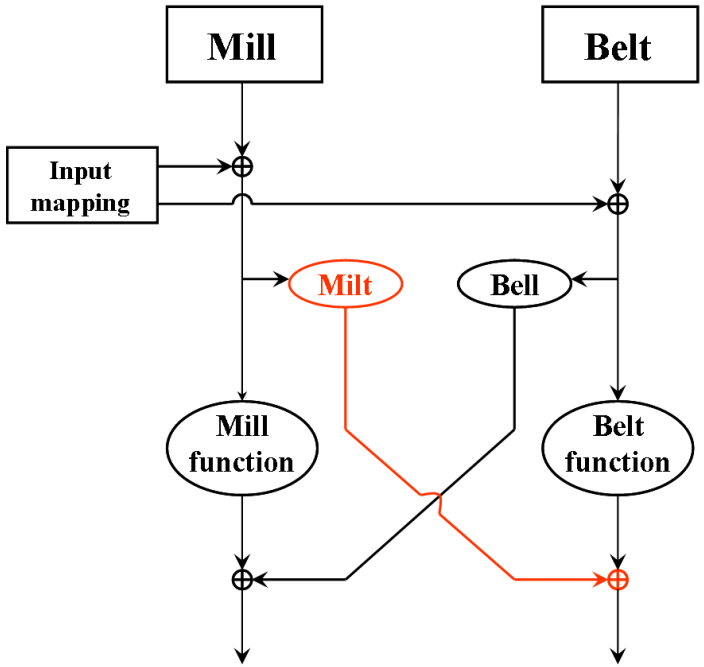
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.



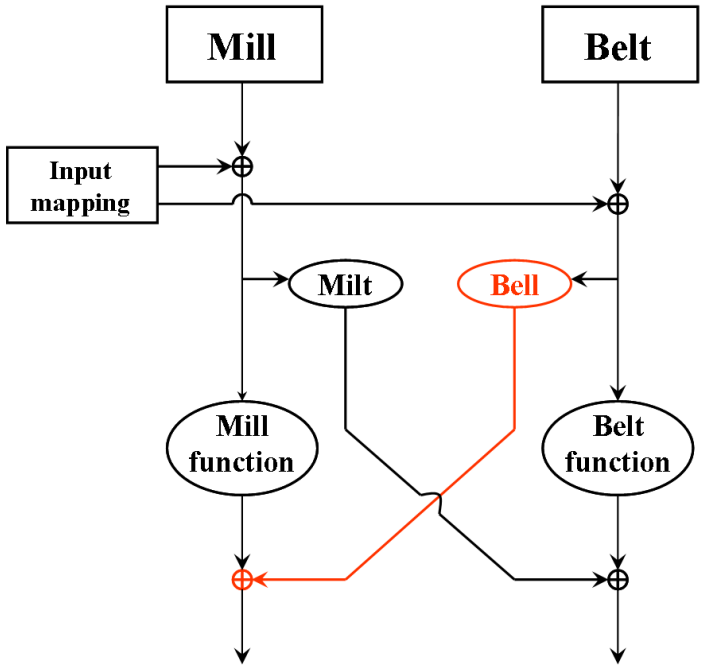
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.



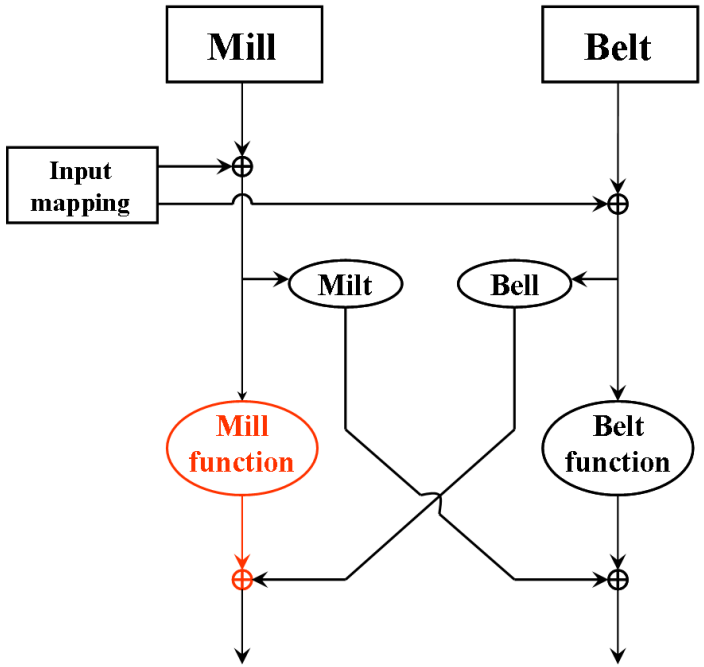
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.



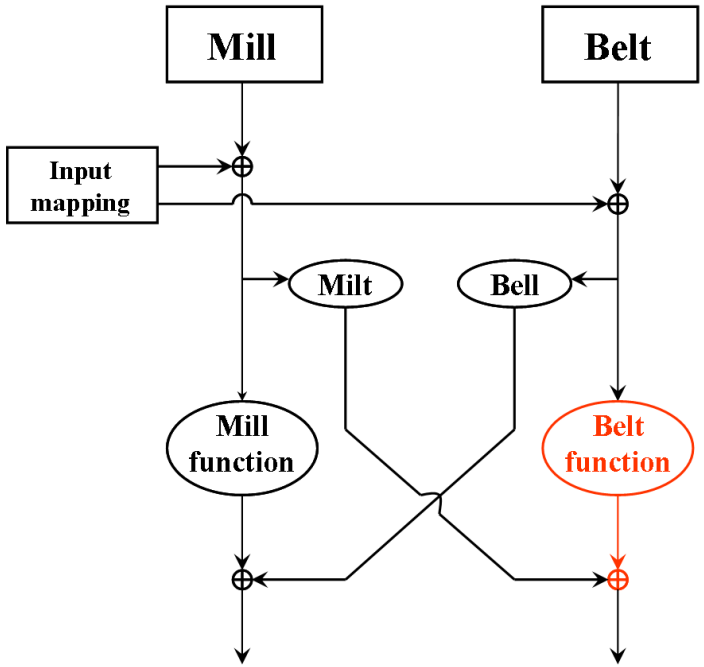
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.

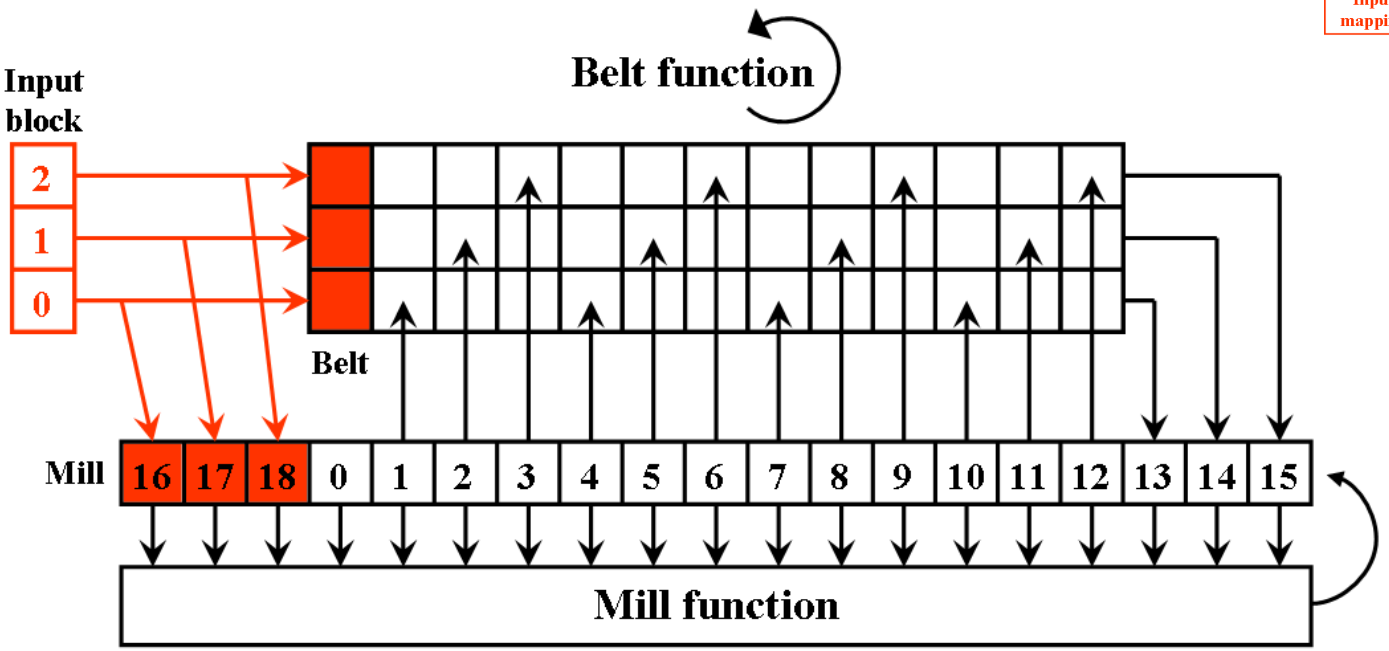


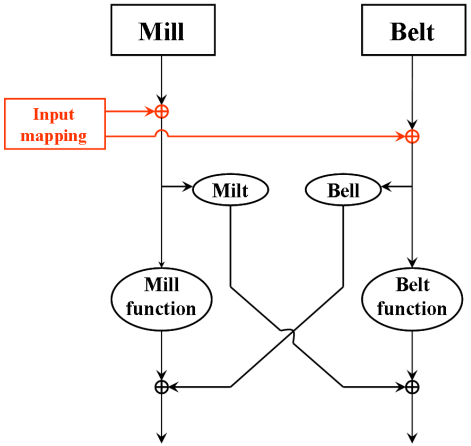
RADIOGATÚN (Bertoni *et al.* - 2007)

- **initialize** the state with zeros.
- **for each round** do (while all the padded message hasn't been processed):
  - **XOR** 3 words of the Mill and 3 of the Belt to 3 new message words.
  - do **Milt**.
  - do **Bell**.
  - do **Mill function**.
  - do **Belt function**.
- do **16 blank rounds**.
- **do** (until we reach the good output size): a blank iteration and output 2 words from the Mill.

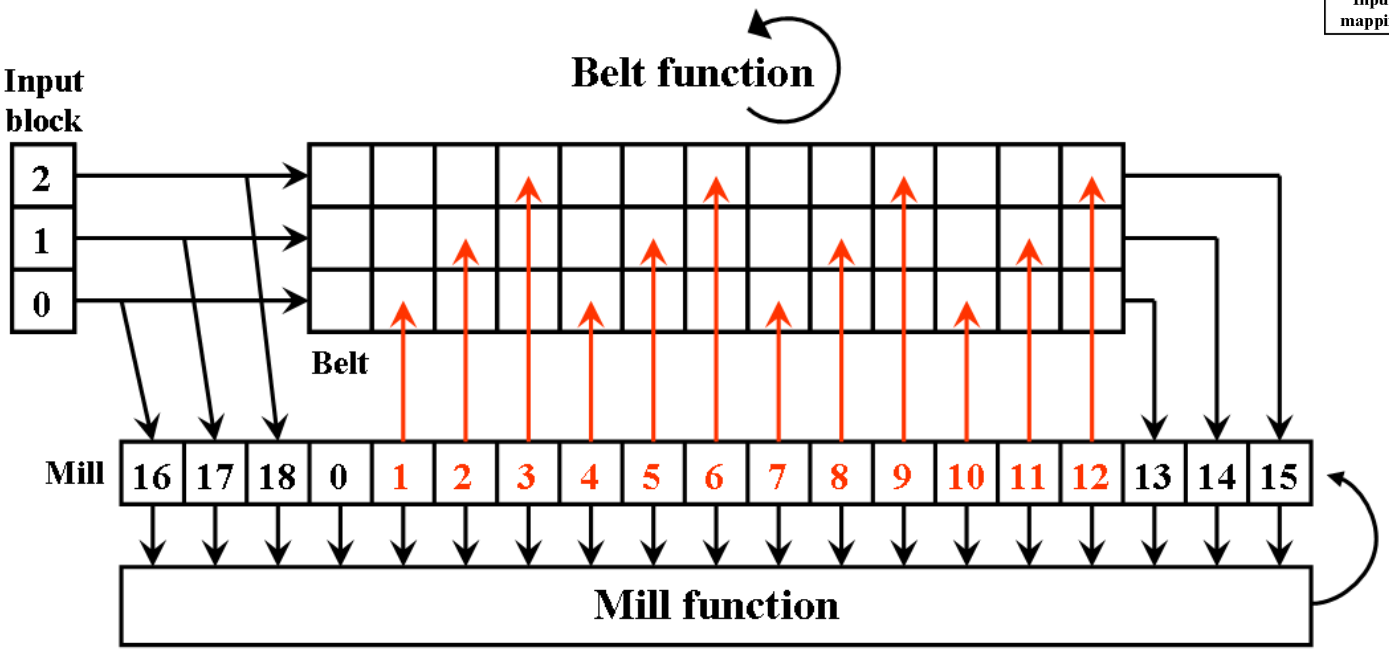


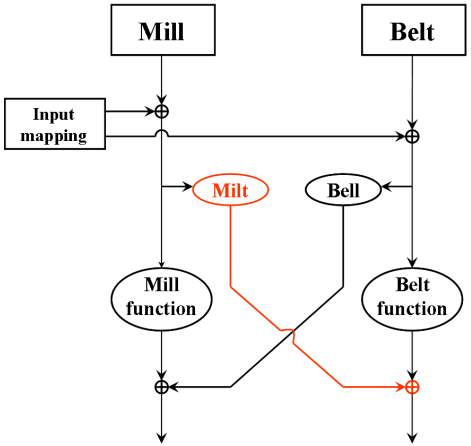
RADIOGATÚN (Bertoni *et al.* - 2007)



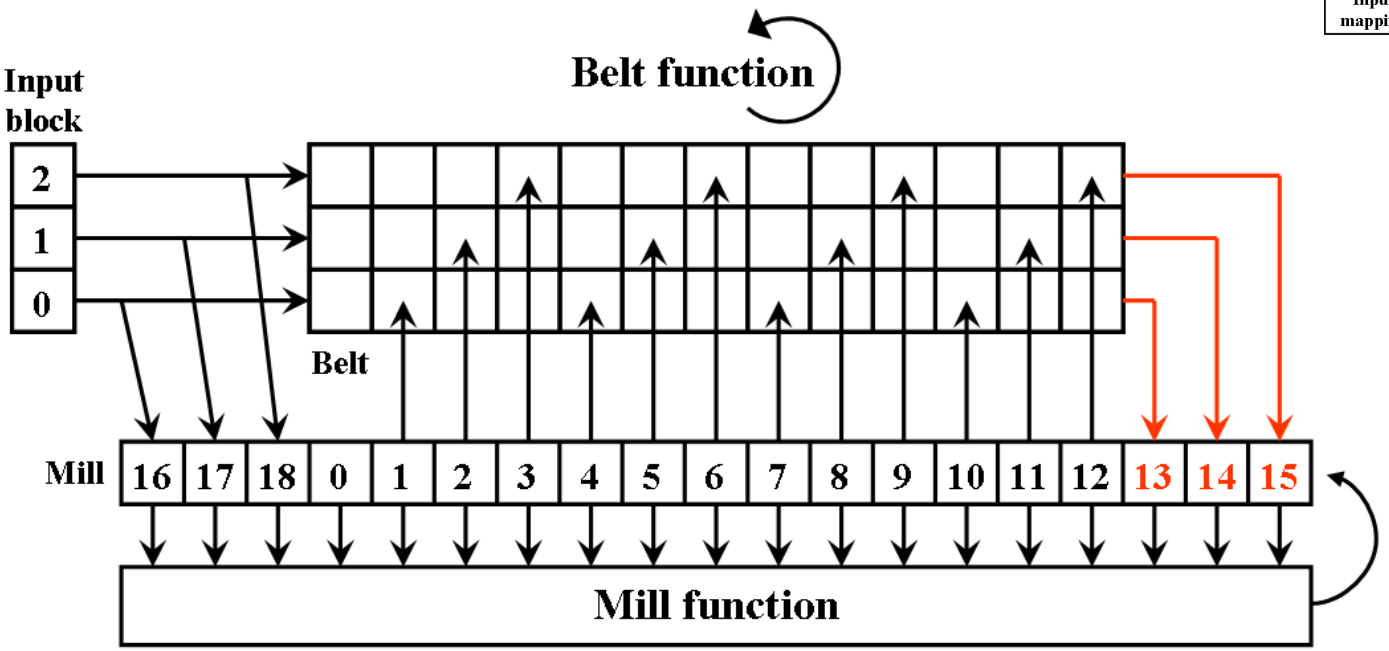


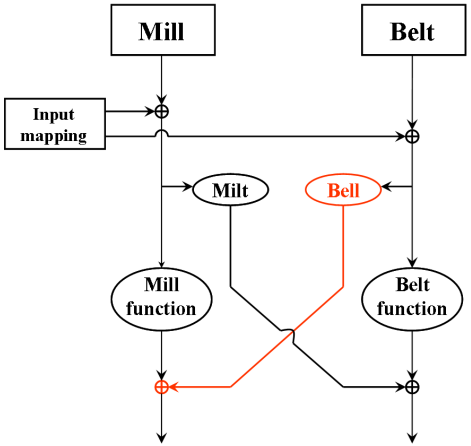
RADIOGATÚN (Bertoni *et al.* - 2007)



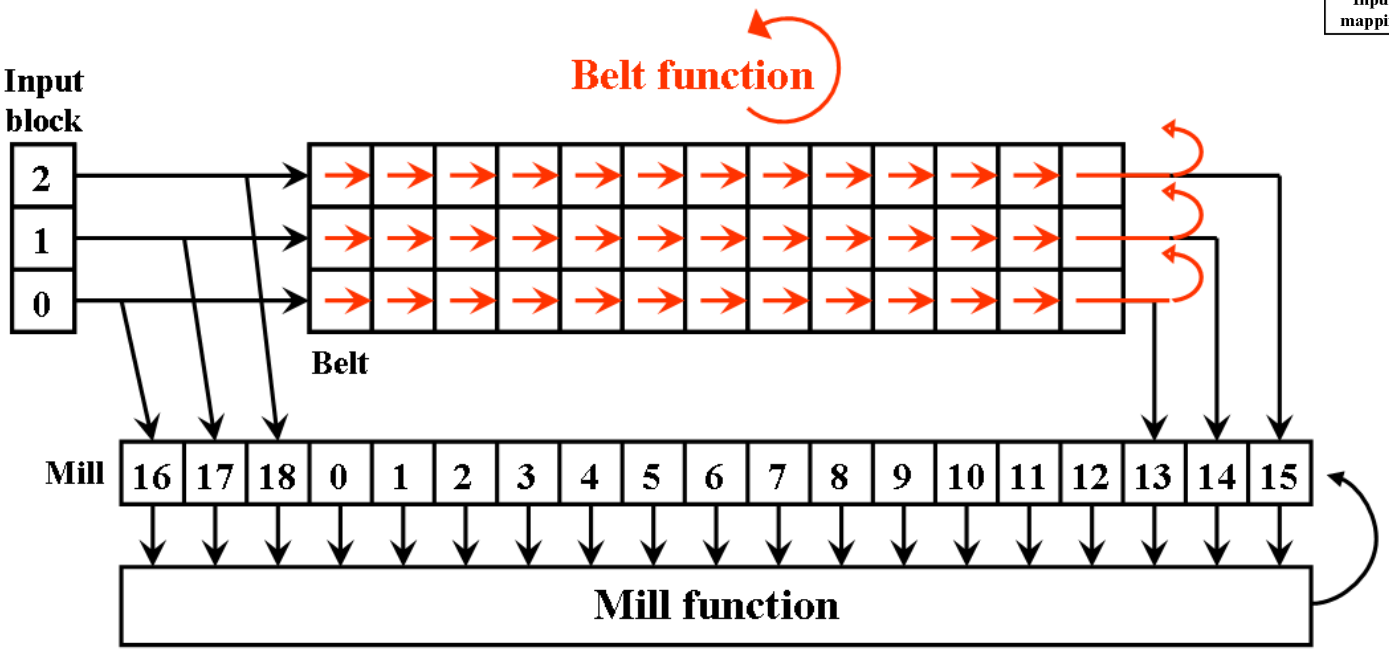


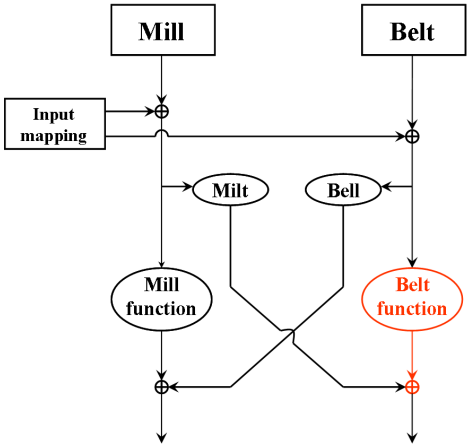
RADIOGATÚN (Bertoni *et al.* - 2007)



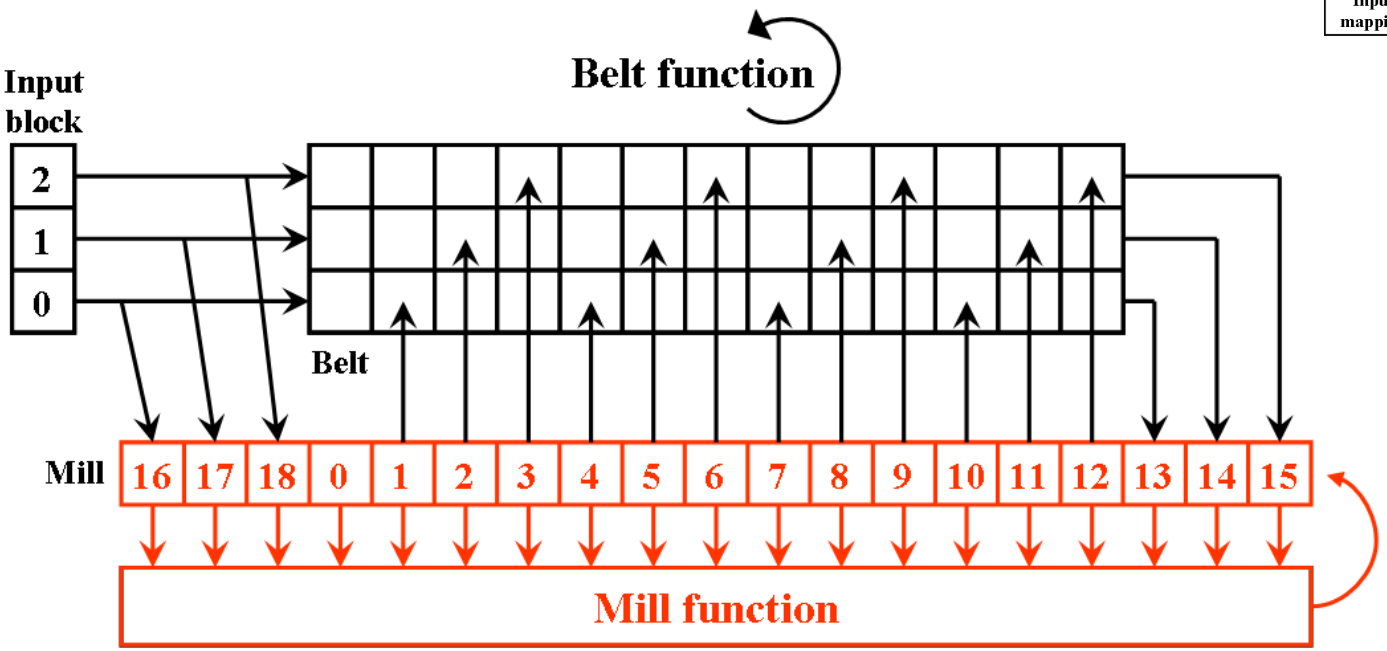


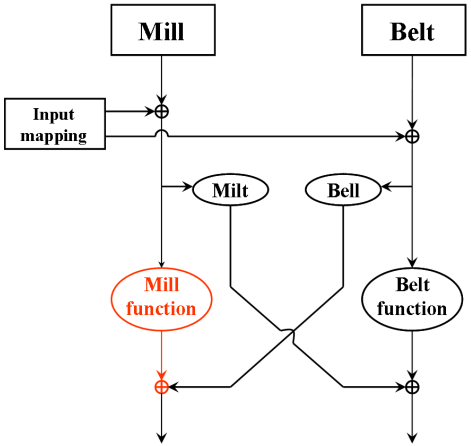
RADIOGATÚN (Bertoni *et al.* - 2007)



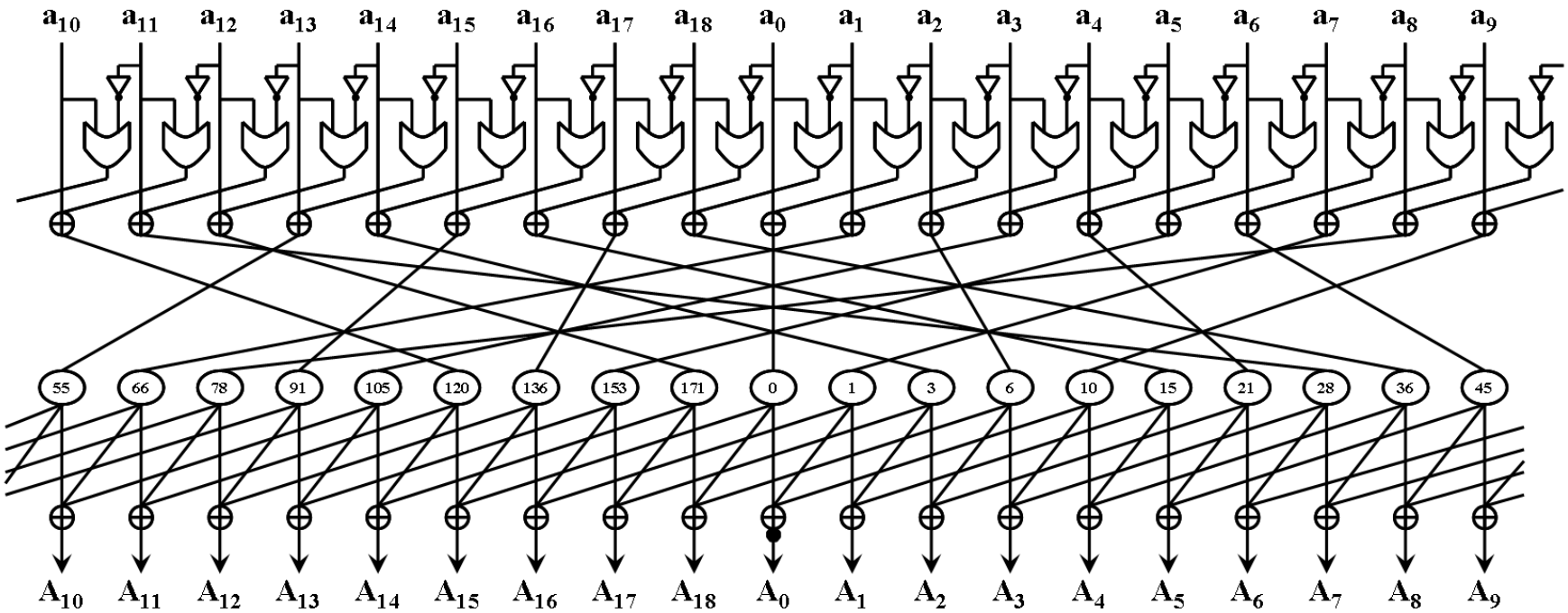


RADIOGATÚN (Bertoni *et al.* - 2007)





RADIOGATÚN (Bertoni *et al.* - 2007)



Security claims

In theory, we should be able to output $c/2 = 55 \times w/2 = 27,5 \times w$ bits, but the internal permutation is not ideal.
The authors claimed that they build a ideal hash function up to 19 words of w bits of output.

The best attack so far for collision resistance (except the birthday attack) is the ***trail backtracking*** with ***symmetric differences***:

- fix the maximum number of iterations (7 steps) of the differential path (to avoid a too big search tree).
- search exhaustively all the differential paths with symmetric differences that start from no difference and lead to a collision.
- for each trail, compute the trail cost by considering a perfect use of the freedom degrees.

Best trail found requires $2^{46 \times w}$ inputs.

GRINDAHL-like attacks

The GRINDAHL attacks tricks don't work here:

- **truncated differences:** when considering $w = 32$, each control step cost a lot.
- **all difference state:** easy to get to, but VERY costly to come to a collision from it.

More generally, the internal state is too big for the GRINDAHL attack to work. However, the round function is quite weak, we should go further in truncated differences and use symmetric differences.

New Idea

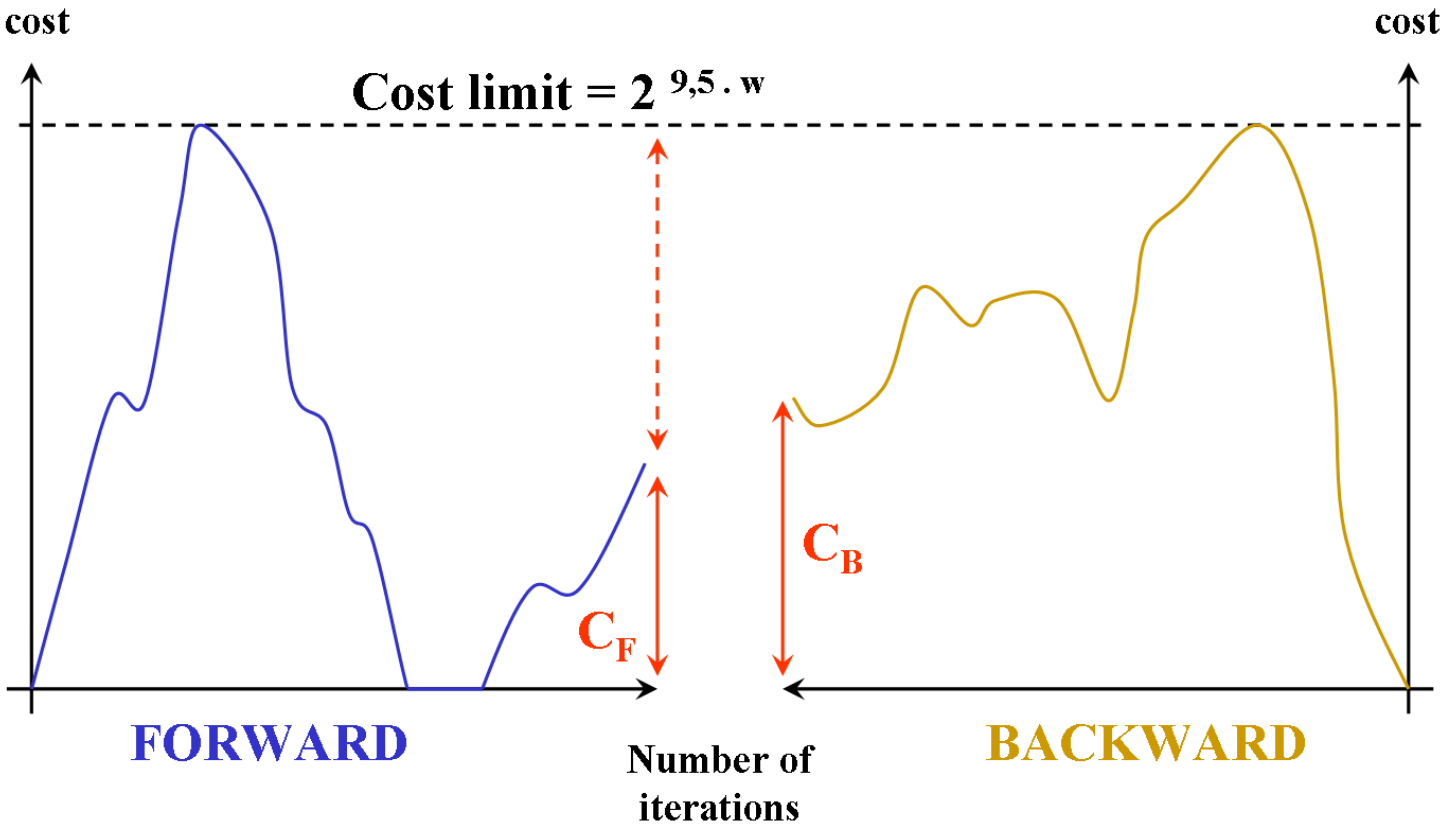
Consider **symmetric differences** for each word: only "all-different" or "equal".

- all the complexity comes from the non-linear part in the Mill function.
- each event you want to force costs you 2^w message freedom.
- the conditions can sometime be compressed (two same conditions on the same word).
- be careful with contradicting conditions.

The **trail backtracking is not optimal**:

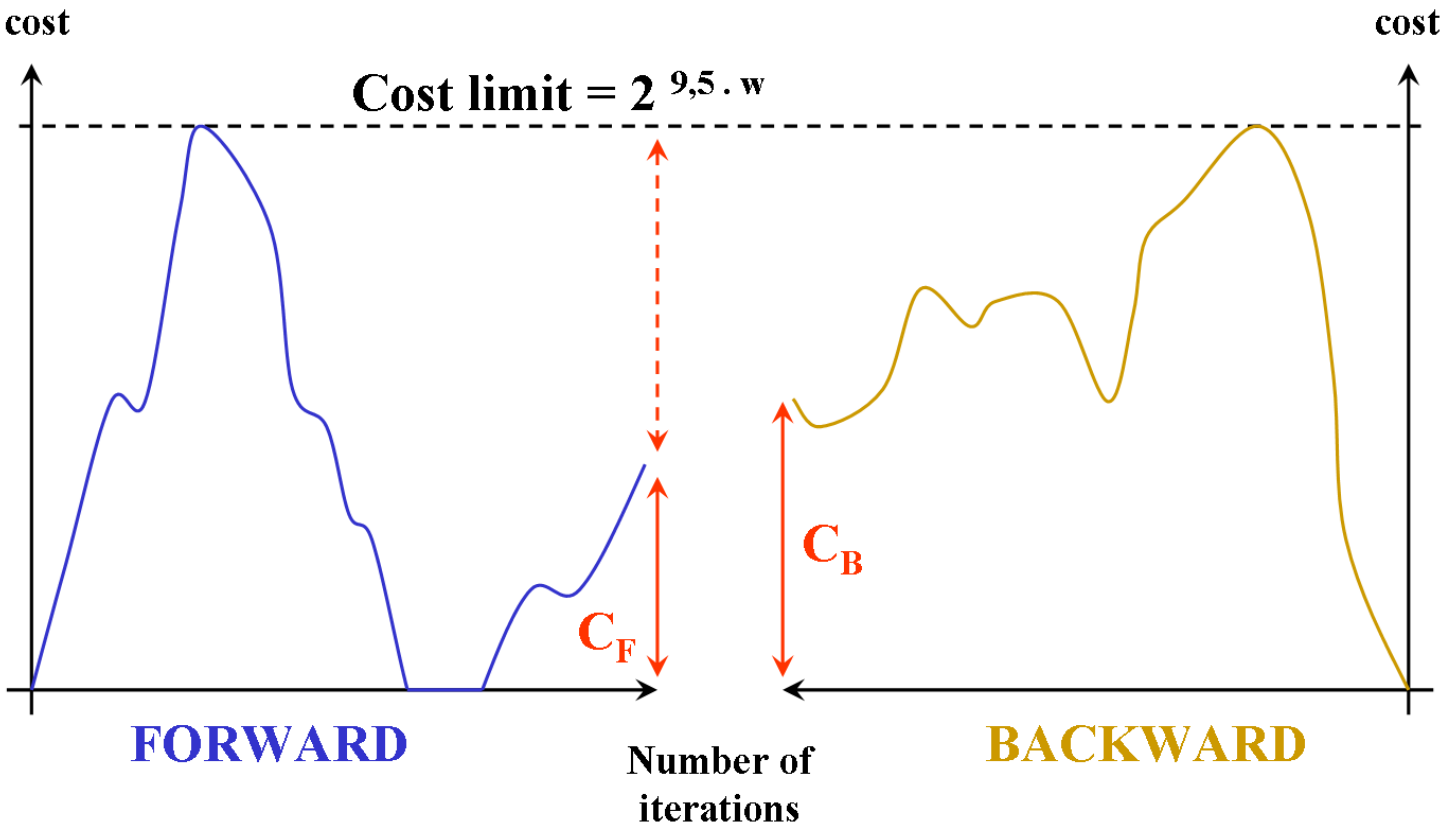
- it continues to run in the search tree, even if the cost is already too big in one branch.
- idea: cut the branch if the cost is already too big, this allows to go much further than 7 steps.
- search for a collision trail with a meet-in-the-middle method ($2^{55/2} = 2^{27,5}$ candidates forward and backward required).

The differential trail search



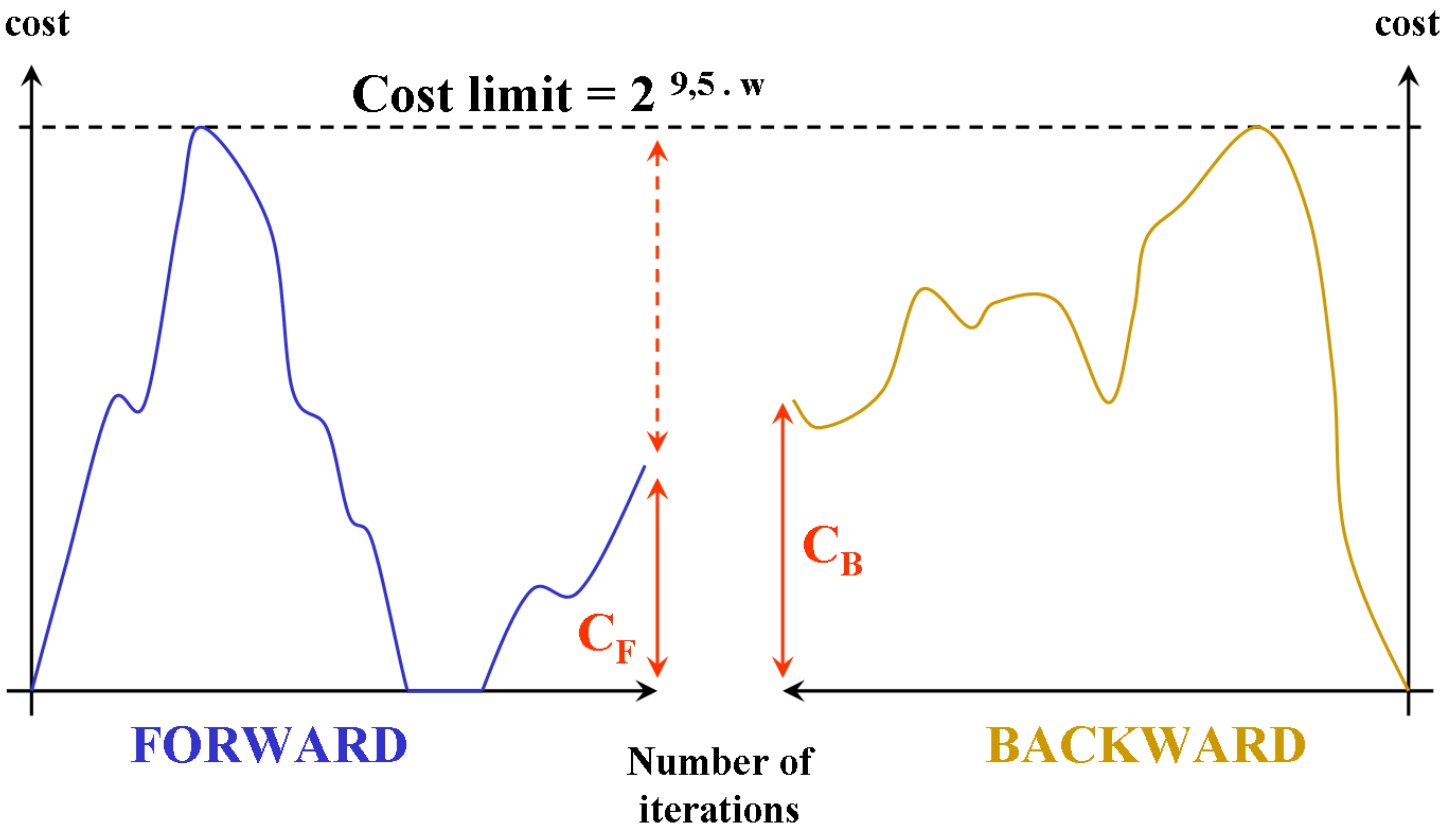
The forward and backward trails must never cost more than: $2^{\frac{19}{2} \times w} = 2^{9.5 \times w}$.

The differential trail search



Joining the two trails is also important: the cost C_F at the end the forward trail and the cost C_B at the end the backward trail must verify $C_F \times C_B \leq 2^{9.5 \times w}$.

The differential trail search



During the meet-in-the-middle phase, we need to run through much more than $2^{27,5}$ candidates for each trail in total.

Results

Results:

- experimentation tends to show that a trail leading to an attack can be found (the search tree is big enough with our restrictions) ...
- ... but the time and memory required to find it is too big for our small computers !!
- same kind of security as GRINDAHL-512 ? **a collision attack surely exists, but the cost for FINDING it is too big for computers (but smaller than the ideal collision bound !)**.

Ongoing work:

- optimize the trail search in order to find better differential path.
- precisely study the freedom degrees use (we assumed a perfect utilisation of them).
- use other symmetric differences (i.e. $\{01, 10, 00, 11\}$).

Conclusion

Sponge functions are great, but ...

- be careful with **slide attacks** when designing a sponge-like function, very simple and costless countermeasures exist.
- be careful with GRINDAHL**-like attacks**, seems powerful against sponge-like functions (try a lot of possible kind of differences, remember that you can use control bytes).
- be careful with ”security” based on the hardness to FIND an attack (and not to RUN it).

Thank you!