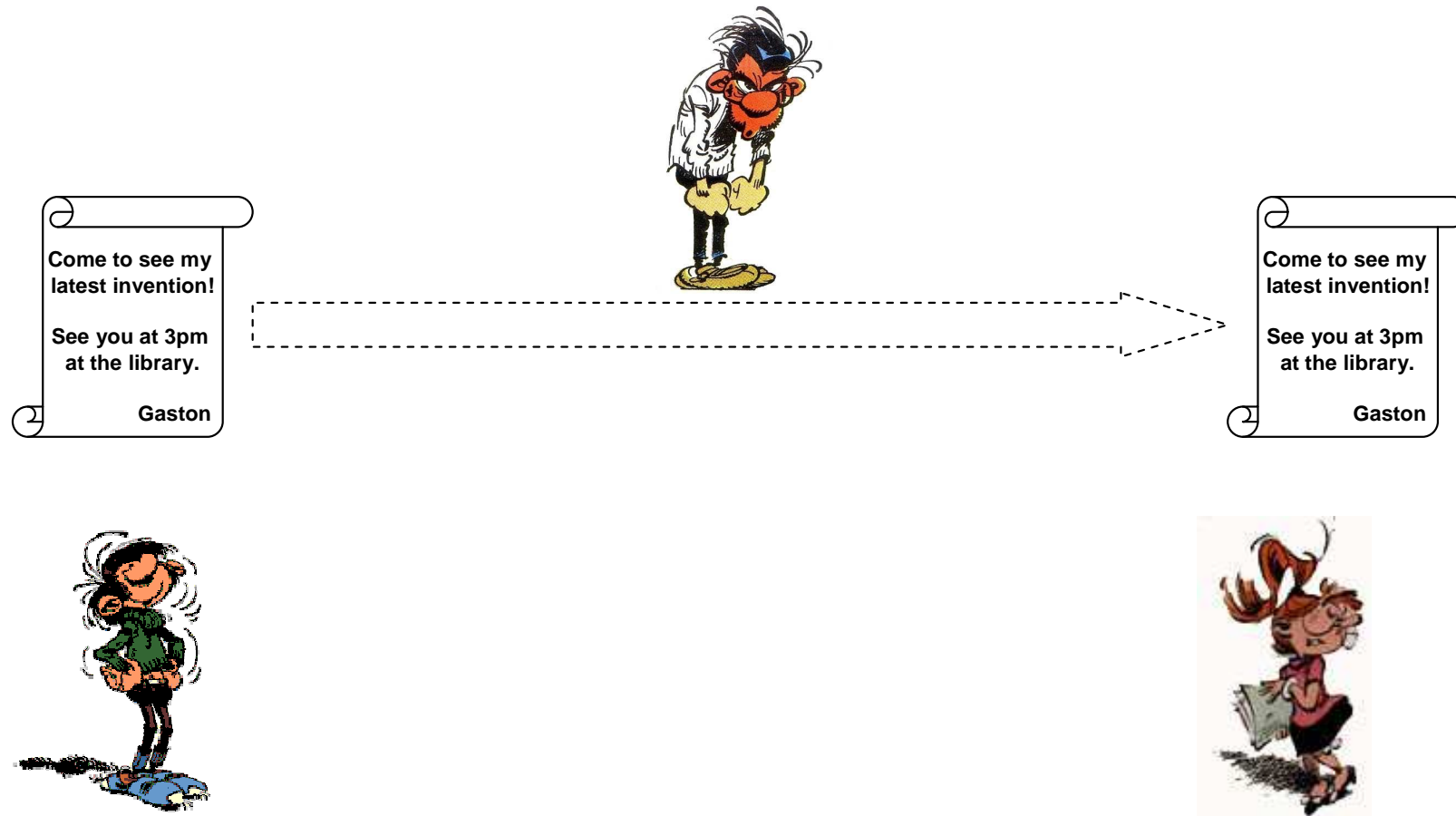
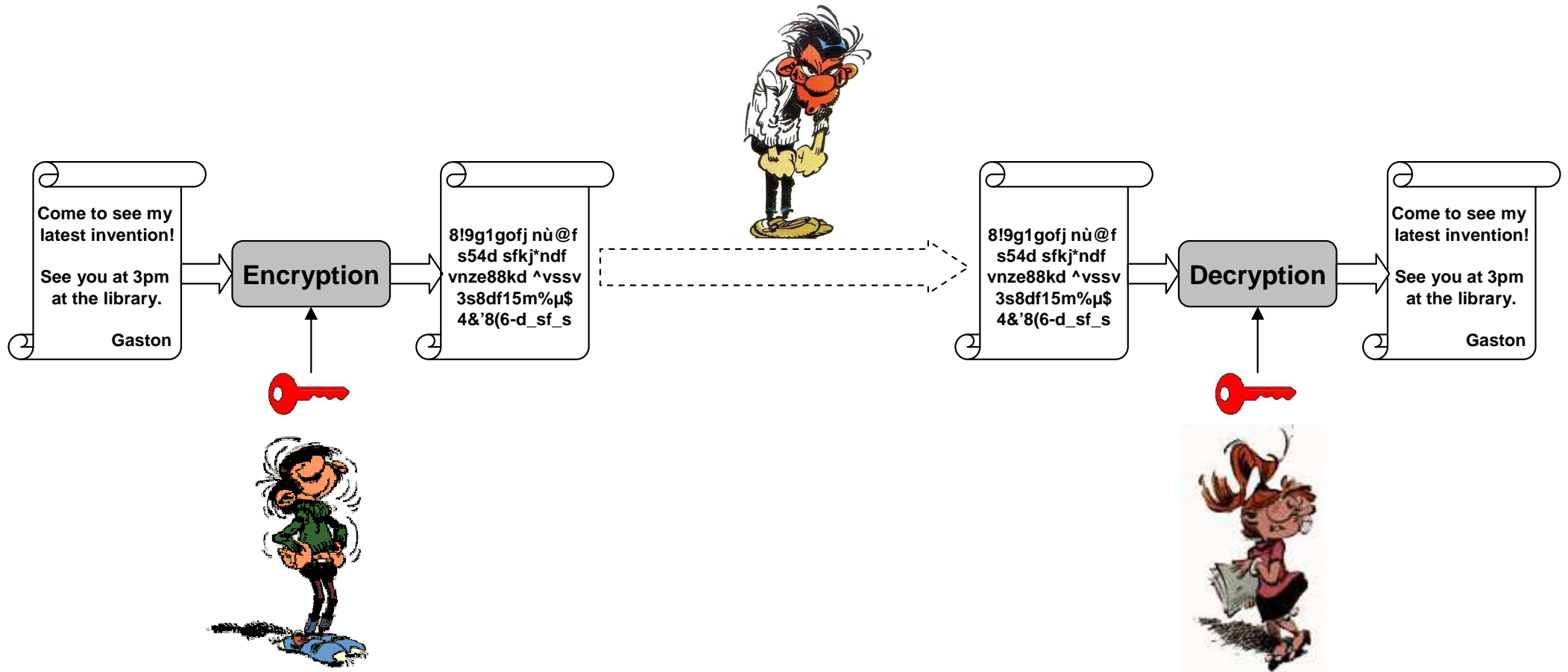
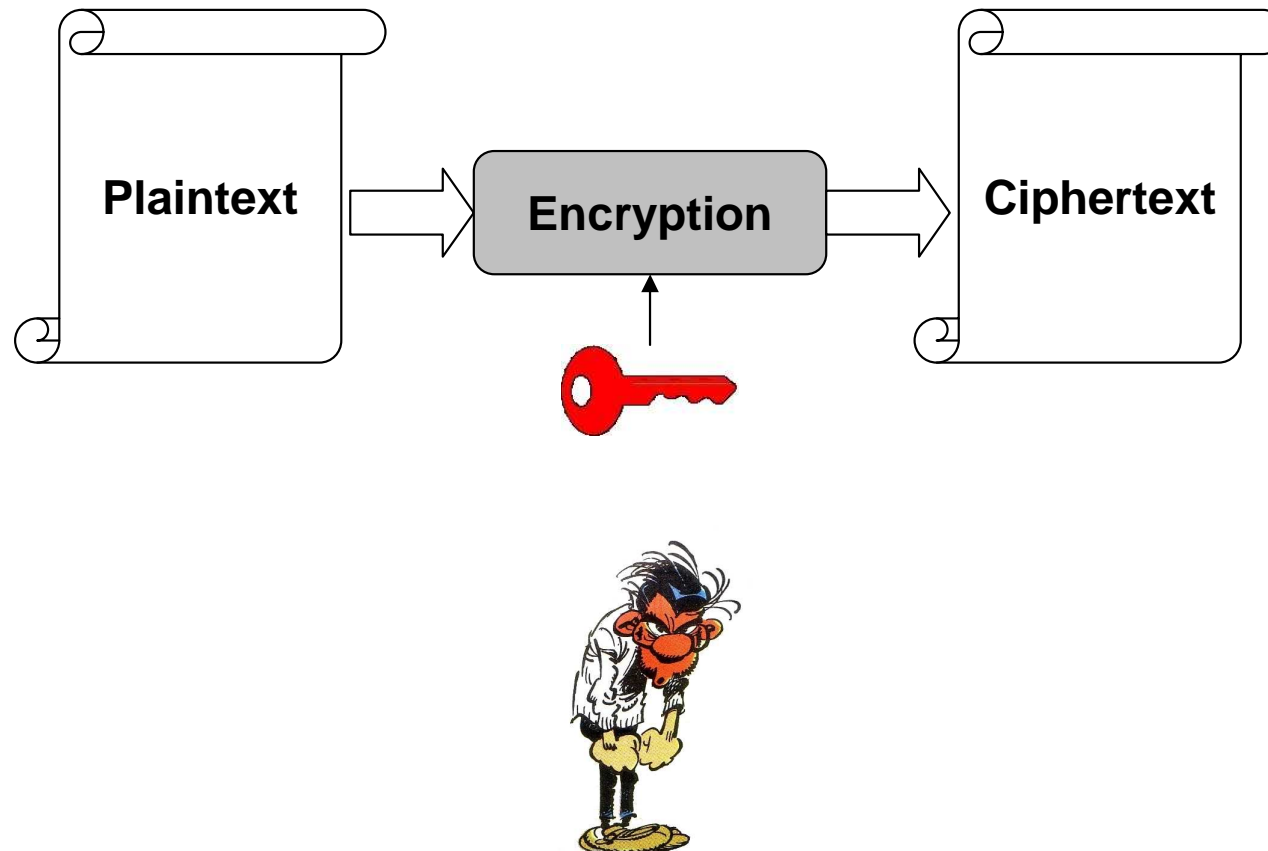


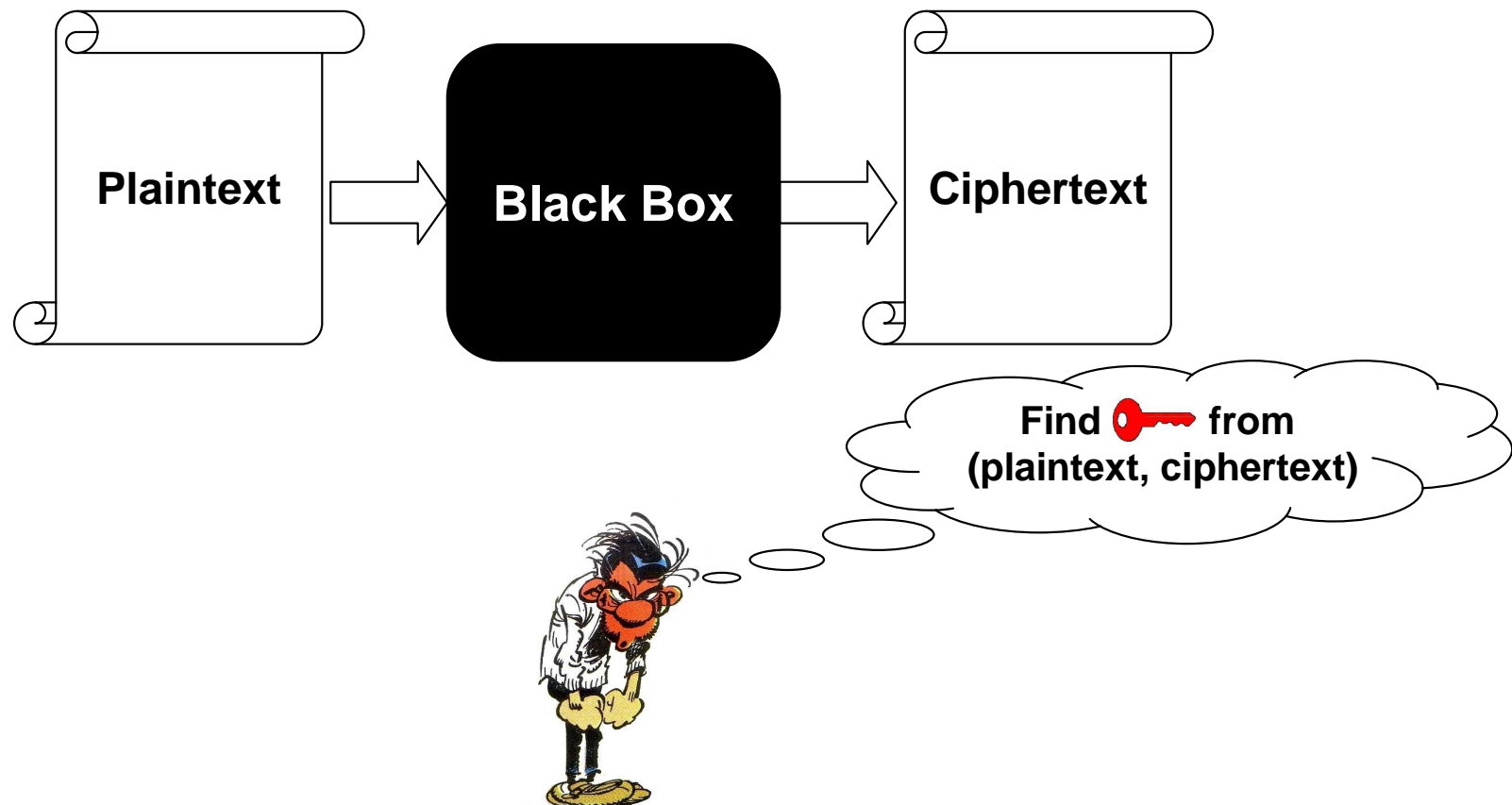
- **Introduction**
- **Countermeasures Description**
- **Implementation Difficulties**
- **(Not So) Futuristic Attacks**

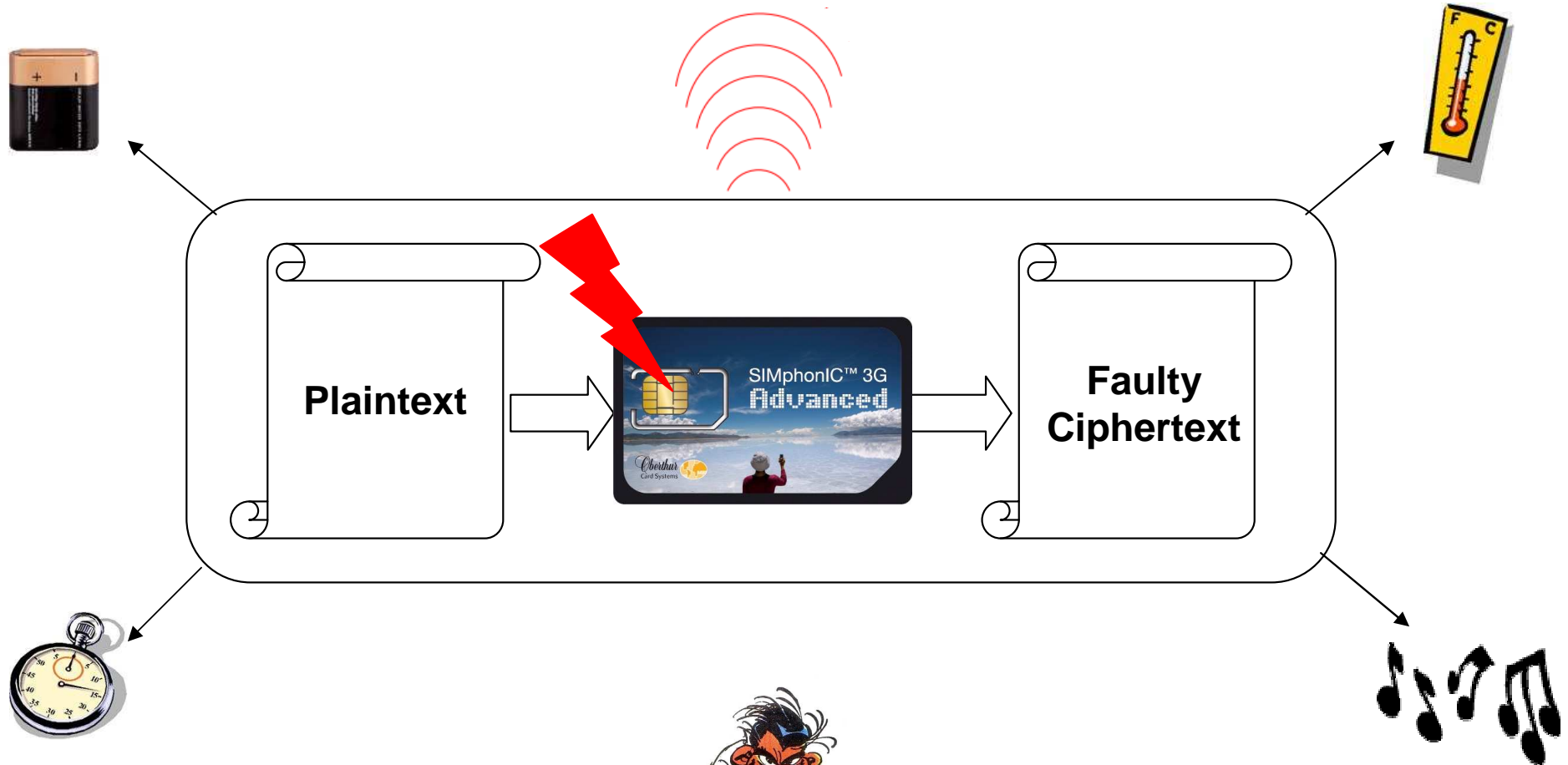
- **Introduction**
- Countermeasures Description
- Implementation Difficulties
- (Not So) Futuristic Attacks











Side-Channel Analysis
Fault Attacks

- First published usage: 1956 : MI5 tries to break encryption used by Egyptian embassy
- First publication for smart card environment : 1996
- Principle: Side-Channel leakages can be dependant on :
 - The operations performed by the device
 - The values of the variables manipulated by these operations
- Three main kinds of leakages:
 - Timing execution
 - Power consumption
 - Electromagnetic radiation

- Objective: find information on the secret key by using only one measurement

Algorithm 1 RSA signature.

 INPUTS: $m, d = (d_{n-1}, \dots, d_0)_2, N$

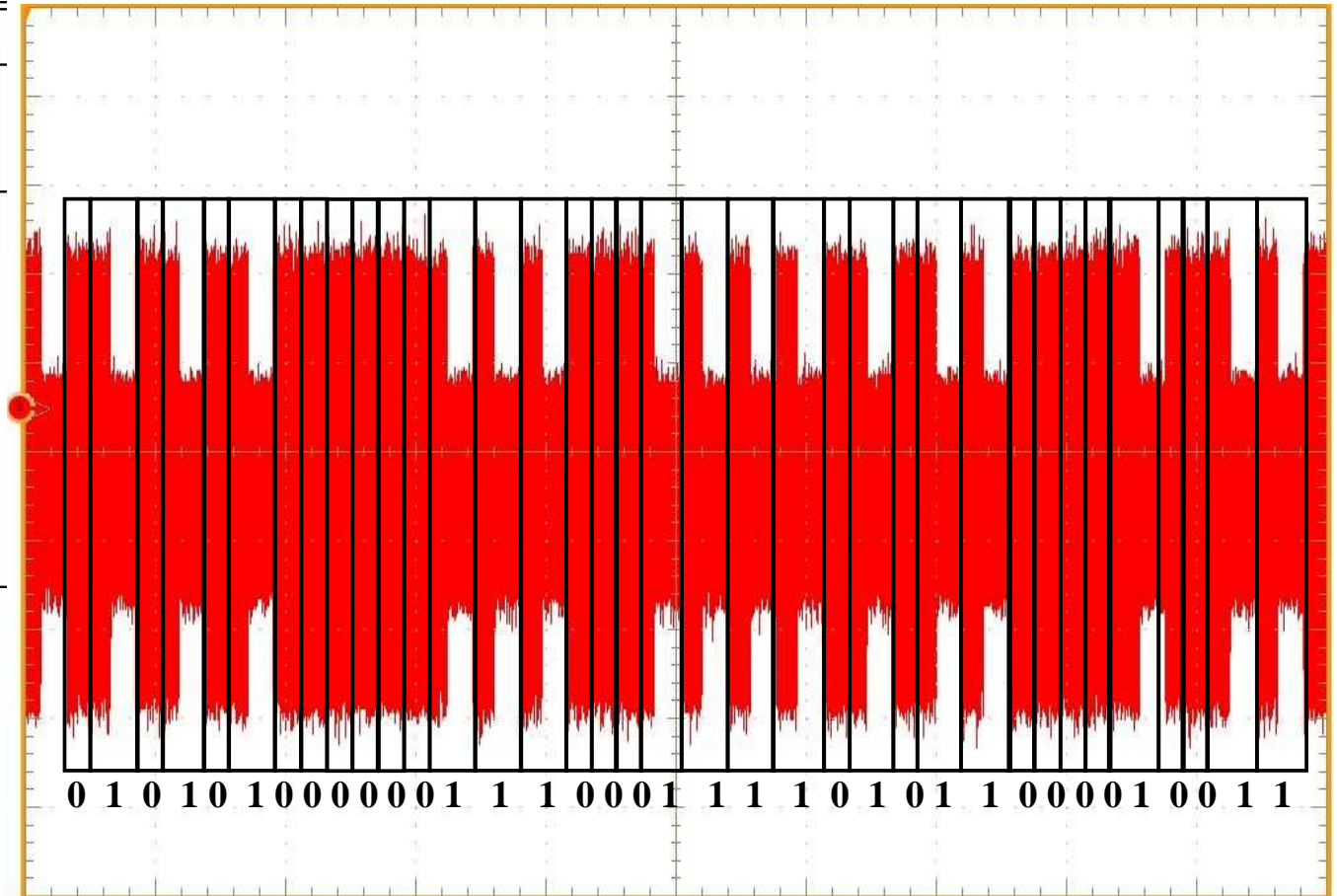
 OUTPUT: $m^d \bmod N$

 $T \leftarrow 1$
for i **from** $n - 1$ **to** 0

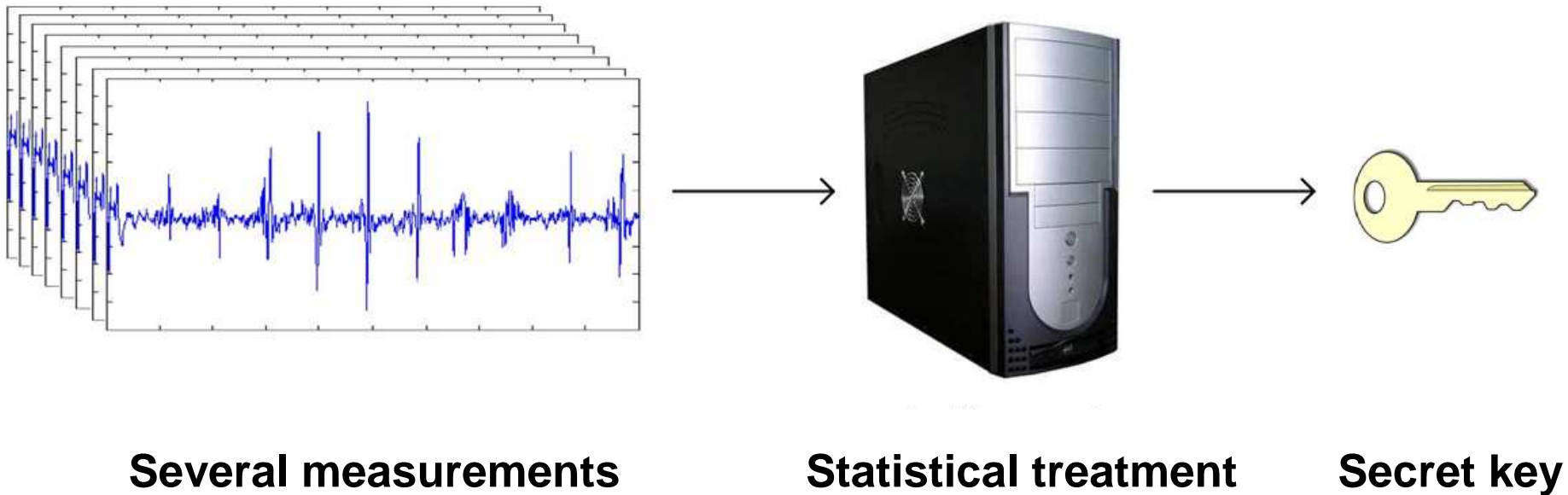
 $T \leftarrow T^2 \bmod N$

 if $d_i = 1$ **then**

 $T \leftarrow T \cdot m \bmod N$
return T



- If too much noise for Simple Analysis to succeed, one can apply Differential Analysis



- Examples of statistical treatments:
 - Distance of Means
 - Pearson correlation coefficient

- May have been used for a long time but no proof until 1993
- Some teenagers used



- Also works on video game stations
- But manufacturers included piezo detectors ☹️
- Nowadays, it could work on food distributors, coffee machines, electrical gates, ...

- First publication for embedded environment : 1996
- Objectives :
 - Recover secret keys
 - Obtain advantageous outputs (higher balance, wrong PIN successfully verified, etc...)
 - Execute the application with granted privileges
- How : Generate faults during code execution
 - Electrical disturbances (glitches)
 - Electromagnetic disturbances
 - Light disturbances

- **Error:**
 - Location : bit / byte / word (16-32 bits) / full



1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

Can impact :

1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



- **Error:**

- Location : bit / byte / word (16-32 bits) / full
- Modification : stuck-at 0 / stuck-at 1 / random



1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

With an error on 1 byte, we can obtain :

0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	1	1	1	1	1	1	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

0	1	0	1	0	0	0	1	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



- **Error:**

- Location : bit / byte / word (16-32 bits) / full
- Modification : stuck-at 0 / stuck-at 1 / random
- Time : chosen / period of time more or less precise



1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

With an stuck-at 0 error on 1 byte, we can obtain :

0	0	0	0	0	0	0	0	1	0	0	0	1	0	1	1	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	1	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0	1	0	1
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---

1	0	0	1	1	0	1	0	1	0	0	0	1	0	1	1	0	0	0	0	0	0	0	0	0
---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---	---



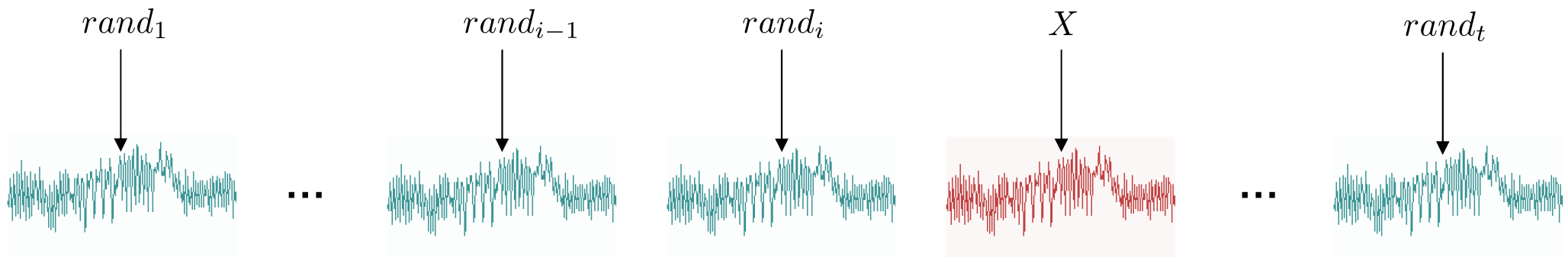
- Introduction
- **Countermeasures Description**
 - **SCA-countermeasures**
 - FA-countermeasures
- Implementation Difficulties
- (Not So) Futuristic Attacks

- Add Noise: Increase the global consumption / radiation of the chip:
 - Activate the various hardware modules even if they are not used: DES, AES, RSA, RNG

- Add Desynchronisation:
 - Hardware: variable clock, random dummy instructions, ...
 - Software: random loops, ...

Use Shuffling method:

- **Principle:** Spread the sensitive signal related to X over t different signals S_1, \dots, S_t leaking at different times.
- Handle the sensitive variable at a random index:

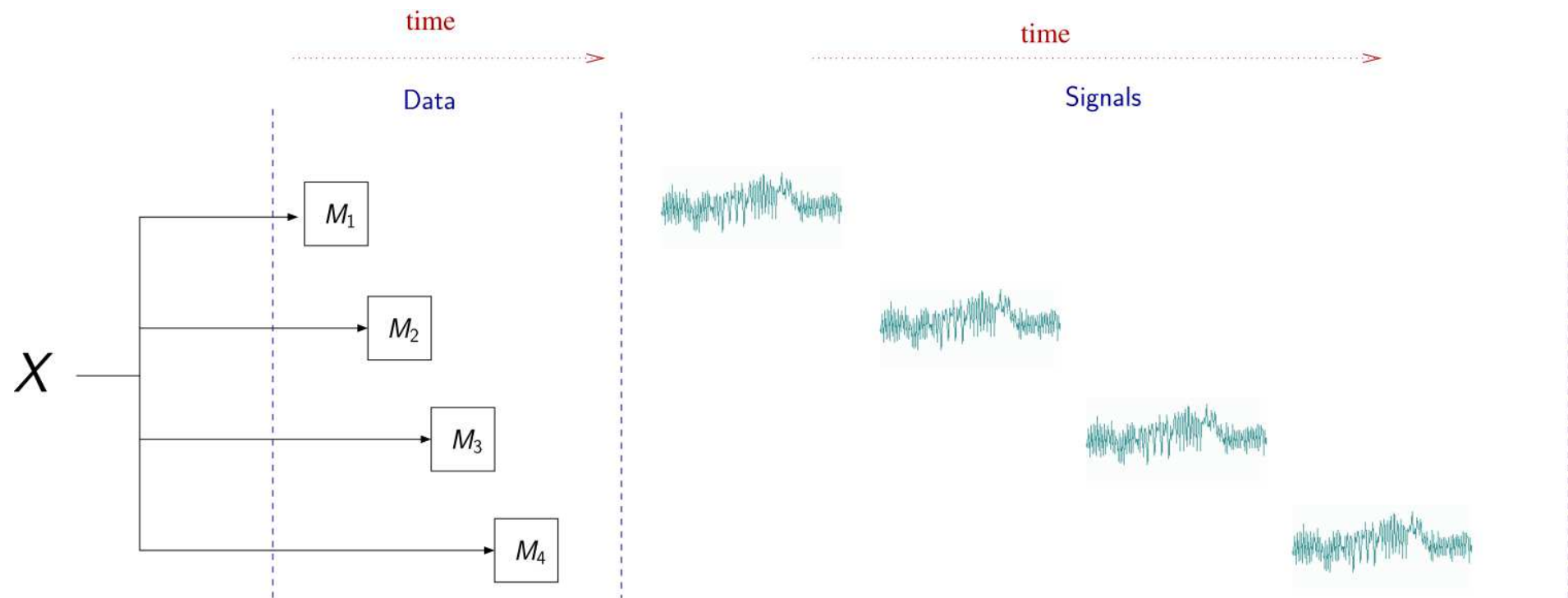


- **Impact:** Decrease the SNR by a factor of t
- **Asset:** Can protect any operation on X .
- **Issue:** t must be very large to have satisfying security.

Use Masking method:

- Principle: Randomly split X into $d + 1$ shares M_0, \dots, M_d s.t.

$$M_0 \star \dots \star M_d = X \quad .$$



Use Masking method:

- **Principle:** Randomly split X into $d + 1$ shares M_0, \dots, M_d s.t.

$$M_0 \star \dots \star M_d = X \ .$$

- **Impact:** d^{th} -order SCA complexity grows exponentially with d .
- **Asset:** Masks propagation is easy for linear operations.
- **Issue:** Masks propagation is a very difficult for non-linear operations.

Use Masking method:

- Boolean masking:

- $X = M_0 \oplus \dots \oplus M_d$
- Symmetric cryptosystems

- Arithmetic masking:

- $X \bmod N \rightarrow X + k_0N \bmod k_1N$
- Asymmetric cryptosystems:

$$m^d \bmod N \rightarrow (m + k_0N)^{d+k_1\varphi(N)} \bmod k_2N$$

Exemple of a 2O-countermeasure to protect symmetric cryptosystems :

- Input : $\tilde{X} = X \oplus t_1 \oplus t_2$
- Output: $F(X) \oplus s_1 \oplus s_2$
- Algorithm idea:

For a from 0 to $size(Sbox) - 1$, compute :

$$F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2$$

$$\begin{array}{ll} \text{if } a = t_1 \oplus t_2 & R_0 \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2 = F(X) \oplus s_1 \oplus s_2 \\ \text{else} & R_1 \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2 \end{array}$$

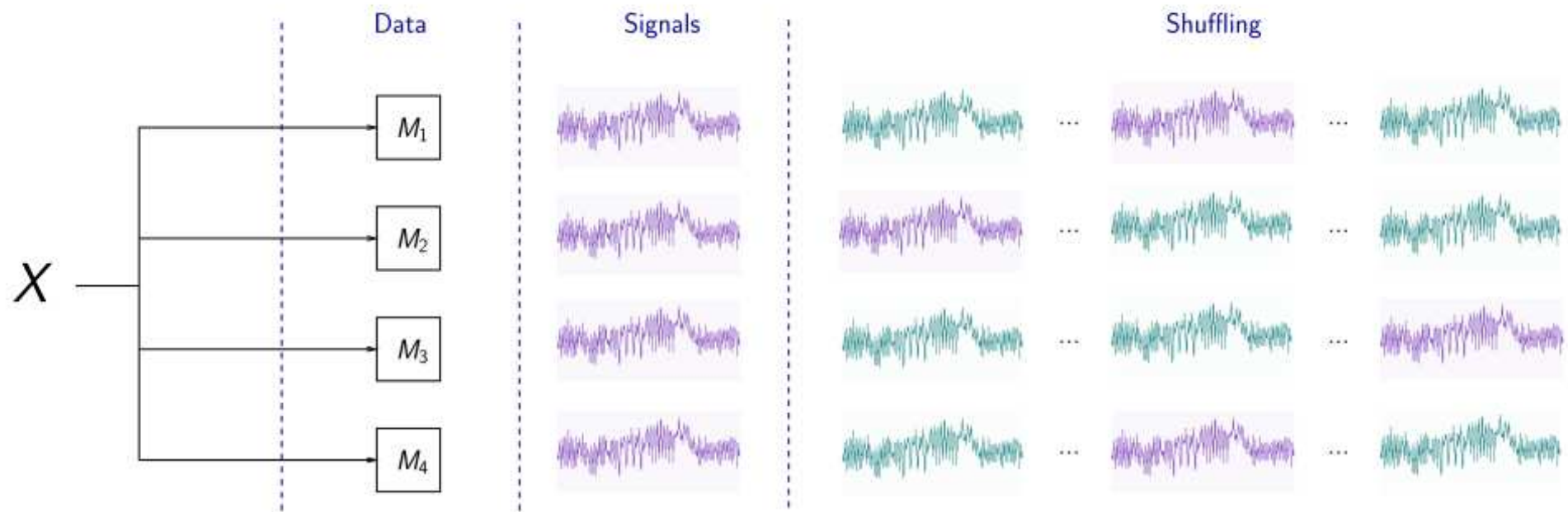
Algorithm 1 Computation of a 2O-masked SBox output from a 2O-masked input

INPUTS: $\tilde{X} = X \oplus t_1 \oplus t_2$, (t_1, t_2) , (s_1, s_2) , a SBox function F OUTPUT: $F(X) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
 2. **for** $a = 0$ **to** $2^n - 1$ **do**
 3. $cmp \leftarrow compare_b(t_1 \oplus a, t_2)$
 4. $R_{cmp} \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2$
 5. **return** R_b
-

- Step 3 : $compare_b$ tests if $a = t_1 \oplus t_2$
- Step 4 : Storage in R_b if equality or in $R_{\bar{b}}$ otherwise.

Combine Masking and Shuffling:



- Countermeasure security is proved in a leakage model

- ODL model : *Only manipulated Data Leak*

$$L \sim \varphi(Z) + B$$

- MTL model : *Memory Transition Leak*

$$L \sim \varphi(Y \oplus Z) + B$$

where:

- Z the manipulated value,
- Y the previous manipulated value,
- φ a non constant function (example: HW or identity function),
- B an independent gaussian noise with zero mean.

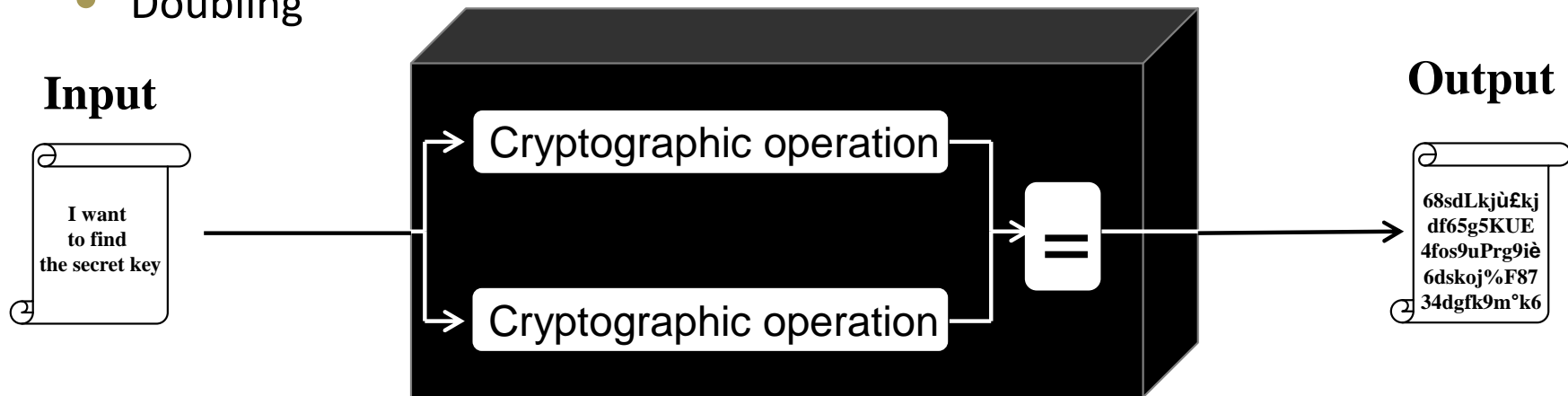
- To simplify, if $\varphi = HW$:

- ODL model \sim *Hamming Weight (HW) model*
- MTL model \sim *Hamming Distance (HD) model*

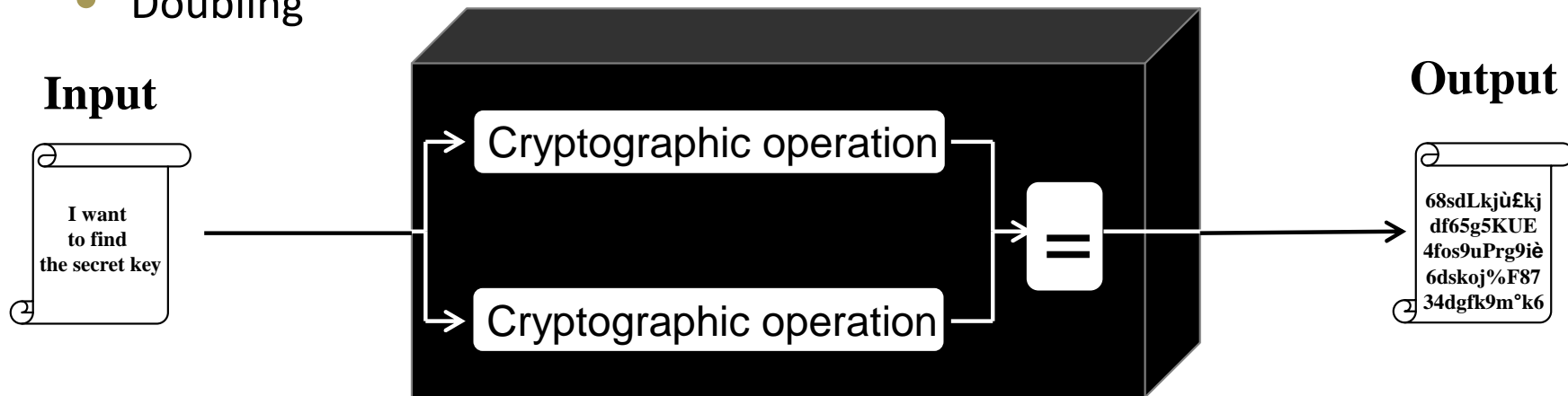
- Introduction
- **Countermeasures Description**
 - SCA-countermeasures
 - **FA-countermeasures**
- Implementation Difficulties
- (Not So) Futuristic Attacks

- Injecting a fault requires SCA of the command to know when to disturb the chip execution
- ➔ Use SCA countermeasures :
 - Add noise
 - Add desynchronisation

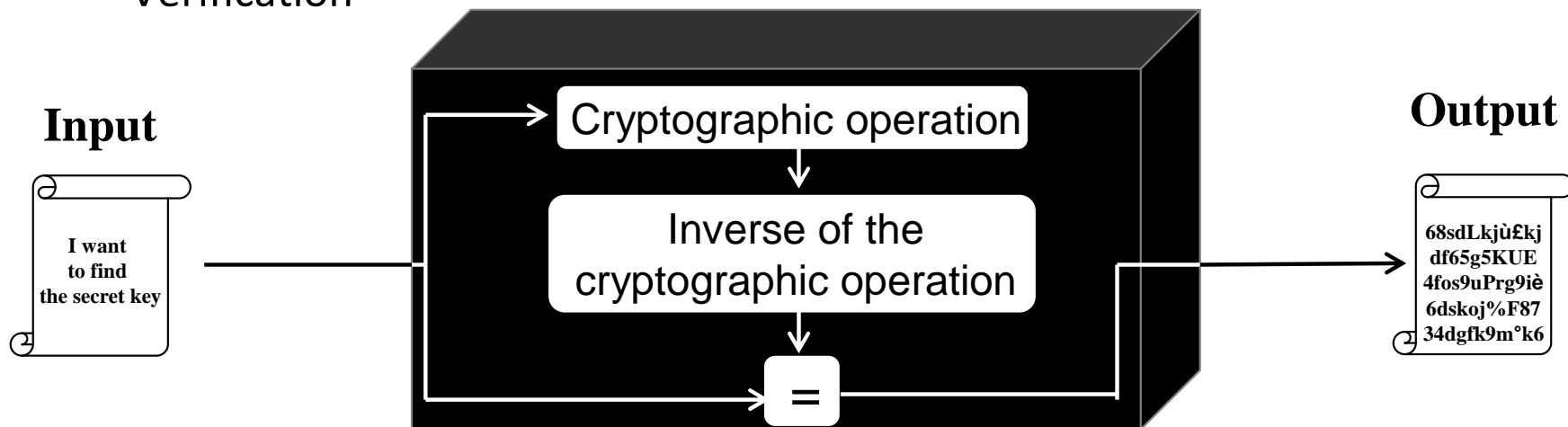
- Main countermeasure against FA : Redundancy
 - Doubling



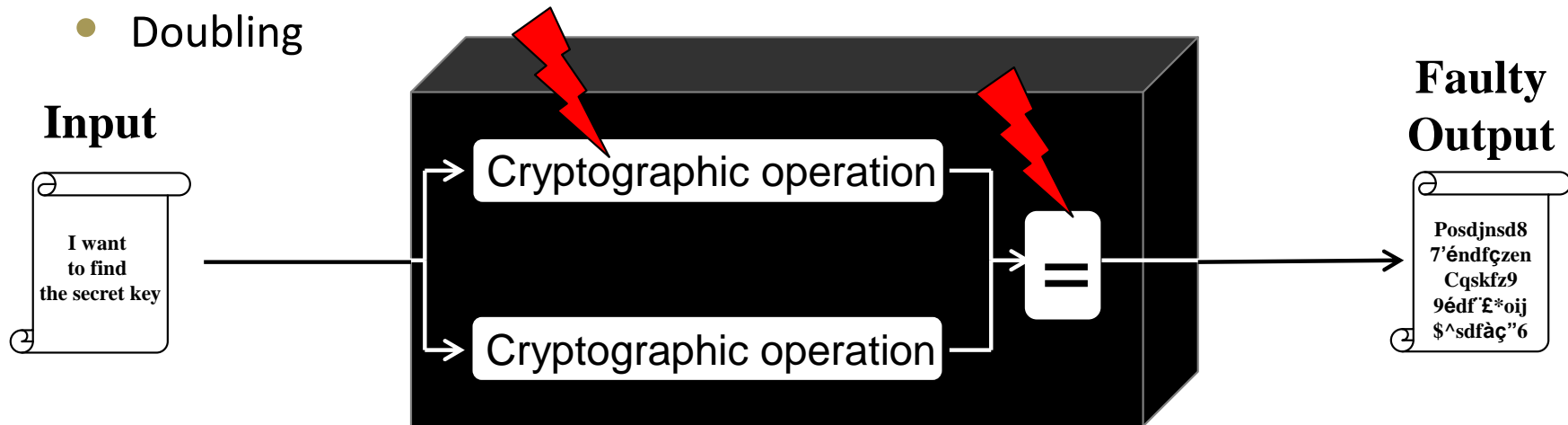
- Main countermeasure against FA : Redundancy
 - Doubling



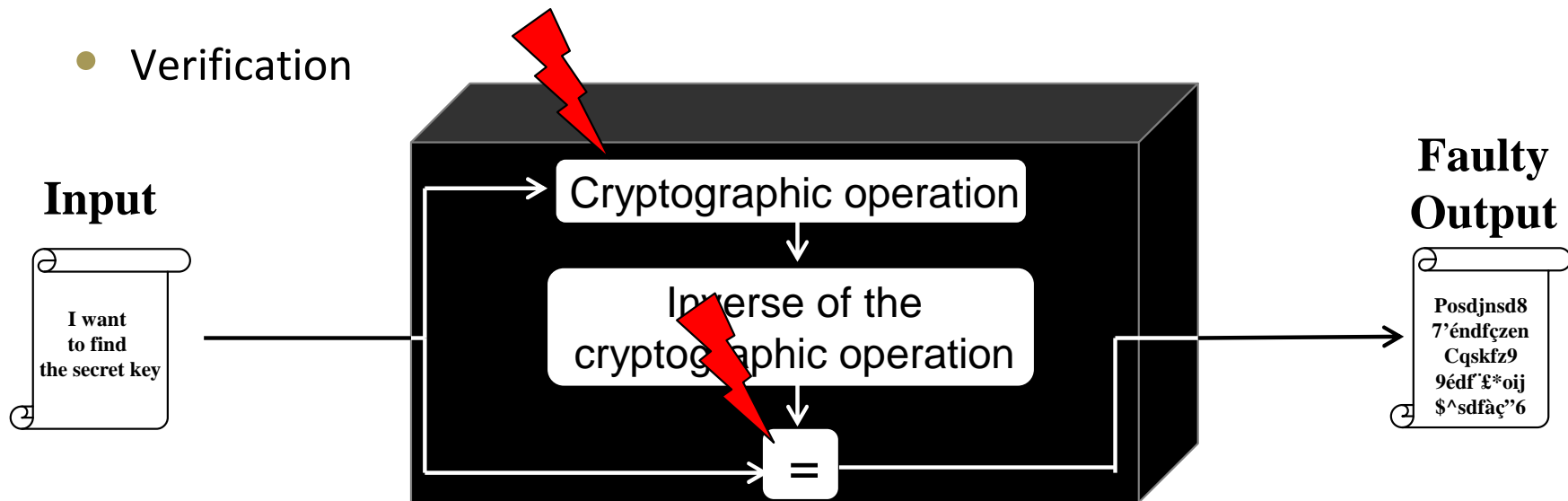
- Verification



- Application of a generic 2O-FA:
 - Doubling

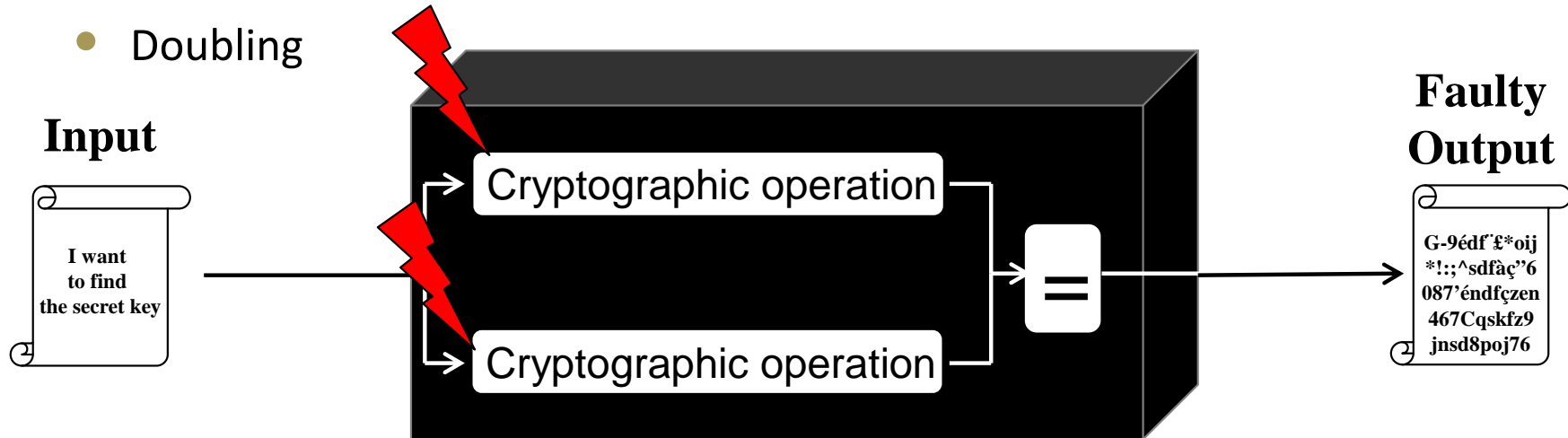


- Verification

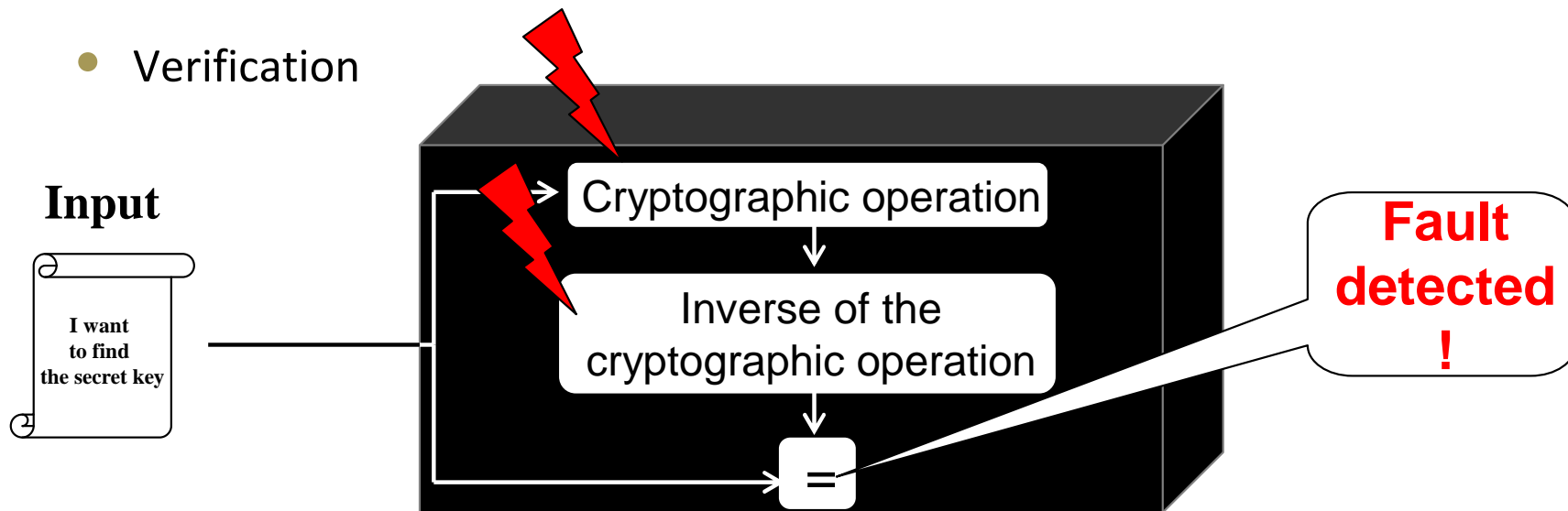


- Application of a 2O-FA inducing the same stuck-at error:

- Doubling

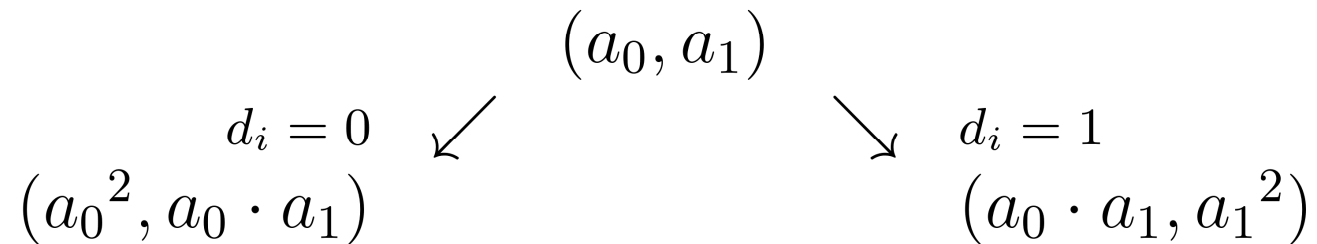


- Verification



More sophisticated methods exist for asymmetric cryptosystems

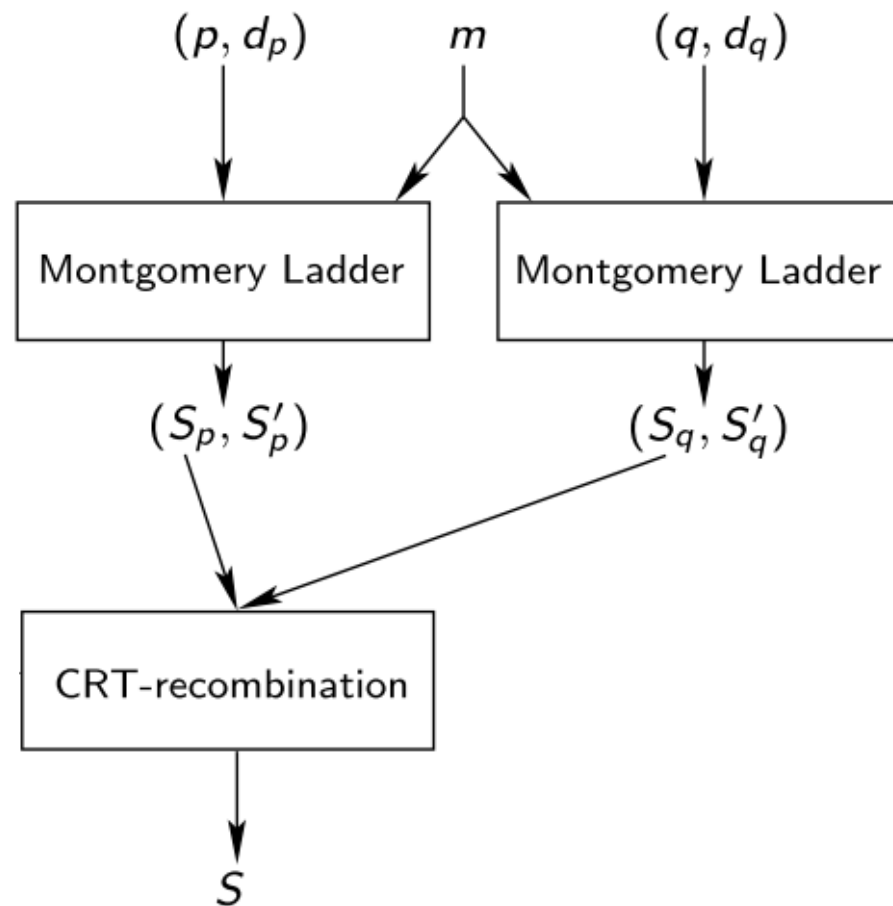
- Example on RSA signature :
 - Use Montgomery Ladder to protect the exponentiation $m^d \bmod N$
 - Initialise (a_0, a_1) to (m, m^2)
 - Then loop on exponent bits:



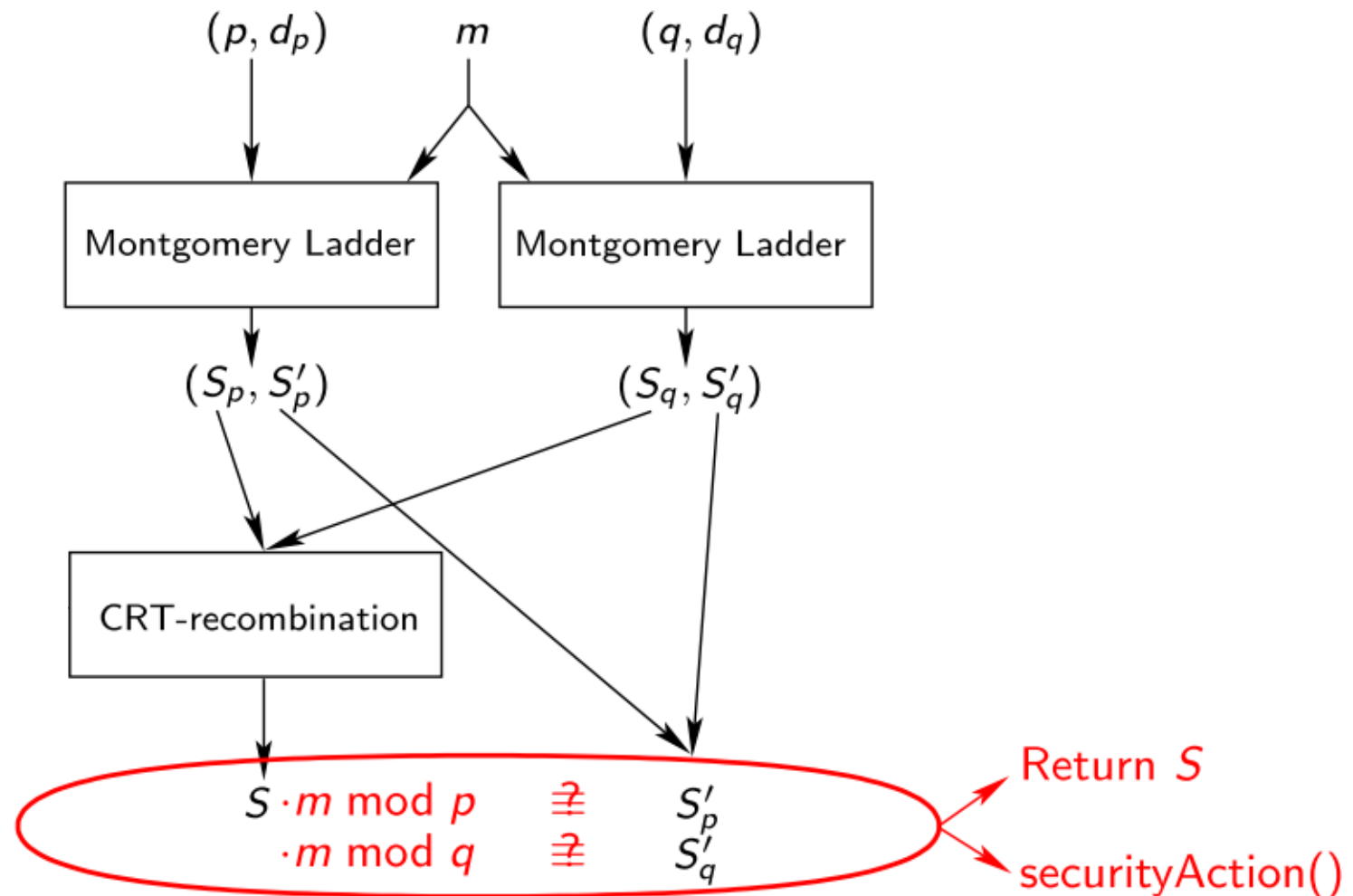
- Remark: (a_0, a_1) is always of the form $(m^\alpha, m^{\alpha+1})$.
 \Rightarrow If a fault is induced during a computation then the coherence is lost!
- Fault detection: Check if $m \cdot a_0 \bmod N = a_1$

- Adaptation to protect CRT-RSA :
 - Parameters:
 - Modulus p and q
 - Exponents $d_p = d \bmod (p - 1)$ and $d_q = d \bmod (q - 1)$
 - Signature of a message m :
 - $S_p = m^{d_p} \bmod p$
 - $S_q = m^{d_q} \bmod q$
 - $S = ((S_p - S_q) \cdot q^{-1} \bmod p) \cdot q + S_q$

Compute S_p and S_q : Montgomery Ladder returning a_0 and a_1 :



Compute S_p and S_q : Montgomery Ladder returning a_0 and a_1 :



- Introduction
- Countermeasures Description
- **Implementation Difficulties**
 - **Countermeasure Conversion**
 - On the Leakage Models
- (Not So) Futuristic Attacks

- Leakage Models :
 - In the literature: generic use of the HW model $L \sim \varphi(Z) + B$.
 - In practice : HD model is more pertinent $L \sim \varphi(Y \oplus Z) + B$.
- Description of a first-order HW-countermeasure:
 - X : sensitive value
 - t : mask
 - $Z = X \oplus t$ the manipulated value,
 \Rightarrow In the HW model: $L \sim \varphi(X \oplus t) + B$
- Description of a first-order attack in the HD model:
 - $Z = X \oplus t$ the manipulated value,
 - $Y = t$ the previous manipulated value,
 \Rightarrow In the HD model: $L \sim \varphi(X) + B$

Algorithm 1 Computation of a 2O-masked SBox output from a 2O-masked input

INPUTS: $\tilde{X} = X \oplus t_1 \oplus t_2$, (t_1, t_2) , (s_1, s_2) , a SBox function F OUTPUT: $F(X) \oplus s_1 \oplus s_2$

1. $b \leftarrow rand(1)$
 2. **for** $a = 0$ **to** $2^n - 1$ **do**
 3. $cmp \leftarrow compare_b(t_1 \oplus a, t_2)$
 4. $R_{cmp} \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2$
 5. **return** R_b
-

- 2O-countermeasure proved secure in the HW model
- What about in the HD model ?

- Implementation of Step 4:

$$R_{cmp} \leftarrow (F(\tilde{X} \oplus a) \oplus s_1) \oplus s_2$$

- Hypotheses:

- R_{cmp} initially at 0
- $R_{cmp} \sim$ accumulator registers

Step 4 is thus implemented such as:

$$4.1 \quad R_{cmp} \leftarrow \tilde{X} \oplus a$$

$$4.2 \quad R_{cmp} \leftarrow F(R_{cmp})$$

$$4.3 \quad R_{cmp} \leftarrow R_{cmp} \oplus s_1$$

$$4.4 \quad R_{cmp} \leftarrow R_{cmp} \oplus s_2$$

Step 4.1 : $R_{cmp} \leftarrow \tilde{X} \oplus a$

- Round targeted: $a = 1 \Rightarrow R_{cmp} \leftarrow \tilde{X} \oplus 1 (= X \oplus t_1 \oplus t_2 \oplus 1)$
- HD model: $L \sim \varphi(Y \oplus Z) + B$
 - Y the previous value contained in R_{cmp}
 - Z the manipulated value in R_{cmp}

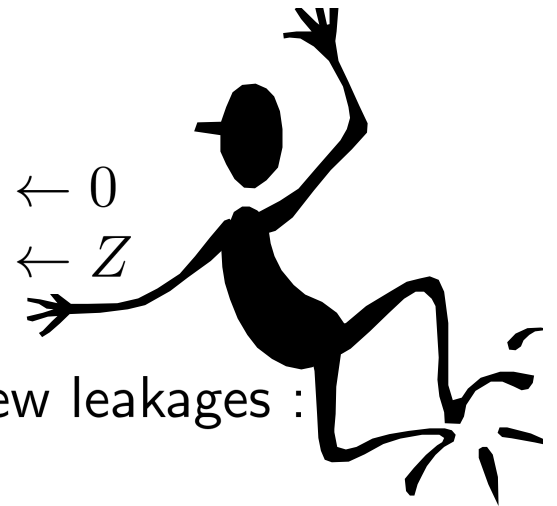
	Initial State	Operation	Y	Z	L
$t_1 \oplus t_2 = 0$	$R_b = F(\tilde{X}) \oplus s_1 \oplus s_2$ $R_{\bar{b}} = 0$	$R_{\bar{b}} \leftarrow \tilde{X} \oplus 1$	0	$X \oplus 1$	$\varphi(X \oplus 1) + B$
$t_1 \oplus t_2 = 1$	$R_b = 0$ $R_{\bar{b}} = F(\tilde{X}) \oplus s_1 \oplus s_2$	$R_b \leftarrow \tilde{X} \oplus 1$	0	X	$\varphi(X) + B$
$t_1 \oplus t_2 > 1$	$R_b = 0$ $R_{\bar{b}} = F(\tilde{X}) \oplus s_1 \oplus s_2$	$R_{\bar{b}} \leftarrow \tilde{X} \oplus 1$	$F(\tilde{X}) \oplus s_1 \oplus s_2$	$\tilde{X} \oplus 1$	$\varphi(rand) + B$

⇒ Dependence between L and X

- A 2O HW-countermeasure is broken by a 1O HD-attack...
- What a challenge for a developer who wants to use such a method on a component leaking in the HD model !!
- Proposition : Erasing of memory before each writing

Step i : $R_j \leftarrow Z$ becomes Step $i.1$: $R_j \leftarrow 0$

Step $i.2$: $R_j \leftarrow Z$



In the HD model, $L \sim \varphi(Y \oplus Z) + B$ is replaced by 2 new leakages :

$$L \sim \begin{cases} \varphi(Y \oplus 0) + B_1 & \text{of Step 1.1} \\ \text{and} \\ \varphi(0 \oplus Z) + B_2 & \text{of Step 1.2} \end{cases}$$

⇒ A countermeasure proved to be secure in HW model seems also to be secure in HD model...

- Step 4 $R_{cmp} \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2$ becomes:

$$4.1 \quad R_{cmp} \leftarrow 0$$

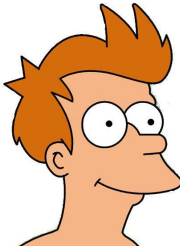

$$4.2 \quad R_{cmp} \leftarrow F(\tilde{X} \oplus a) \oplus s_1 \oplus s_2$$

- The two leakages considered:
 - $L_1 \sim \varphi(\tilde{X}) + B_0 = \varphi(X \oplus t_1 \oplus t_2) + B_0$
 - $L_2 \sim \varphi(Y \oplus 0) + B_1$ (leakage during the erasing step 4.1)
- Round targeted: $a = 1$

	$L_1 \sim \varphi(\tilde{X}) + B_0$	Y	$L_2 \sim \varphi(Y) + B_1$
$t_1 \oplus t_2 = 0$	$\varphi(X) + B_0$	0	$\varphi(0) + B_1$
$t_1 \oplus t_2 = 1$	$\varphi(X \oplus 1) + B_0$	0	$\varphi(0) + B_1$
$t_1 \oplus t_2 > 1$	$\varphi(X \oplus t_1 \oplus t_2) + B_0$	<i>rand</i>	$\varphi(\text{rand}) + B_1$

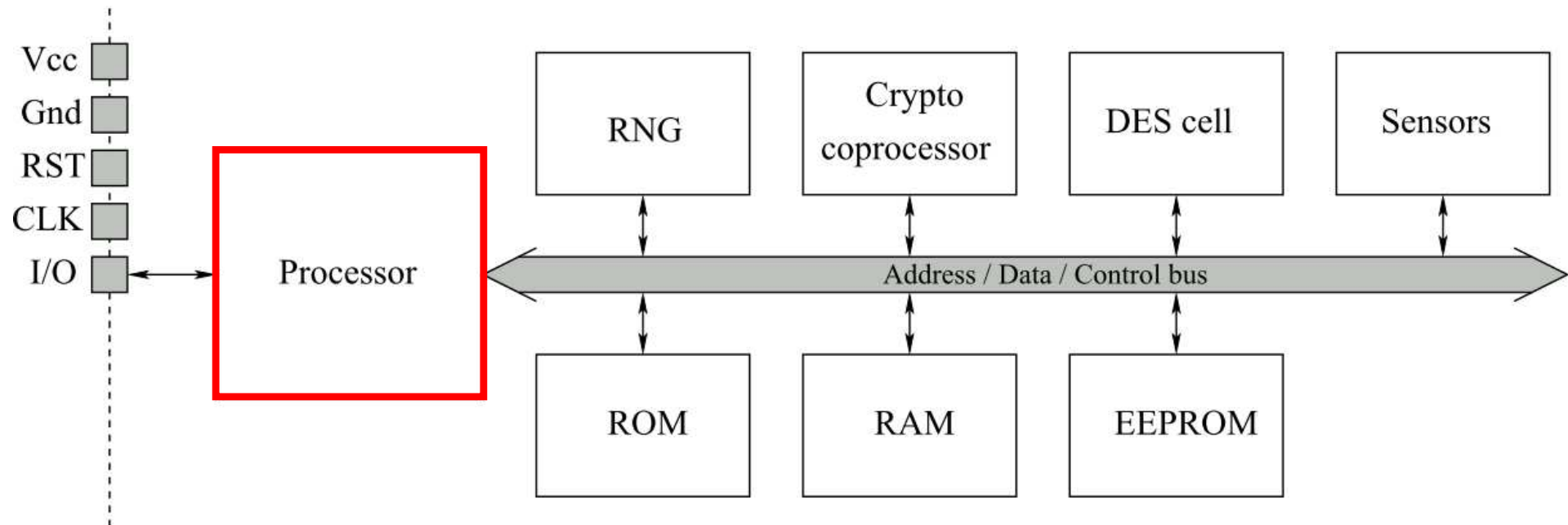
⇒ Dependence between the pair (L_1, L_2) and X !

- First Conclusion:
 - It is very difficult for a developer to adapt the security of a scheme proved to be secure in the HW-model into a more realistic HD-model

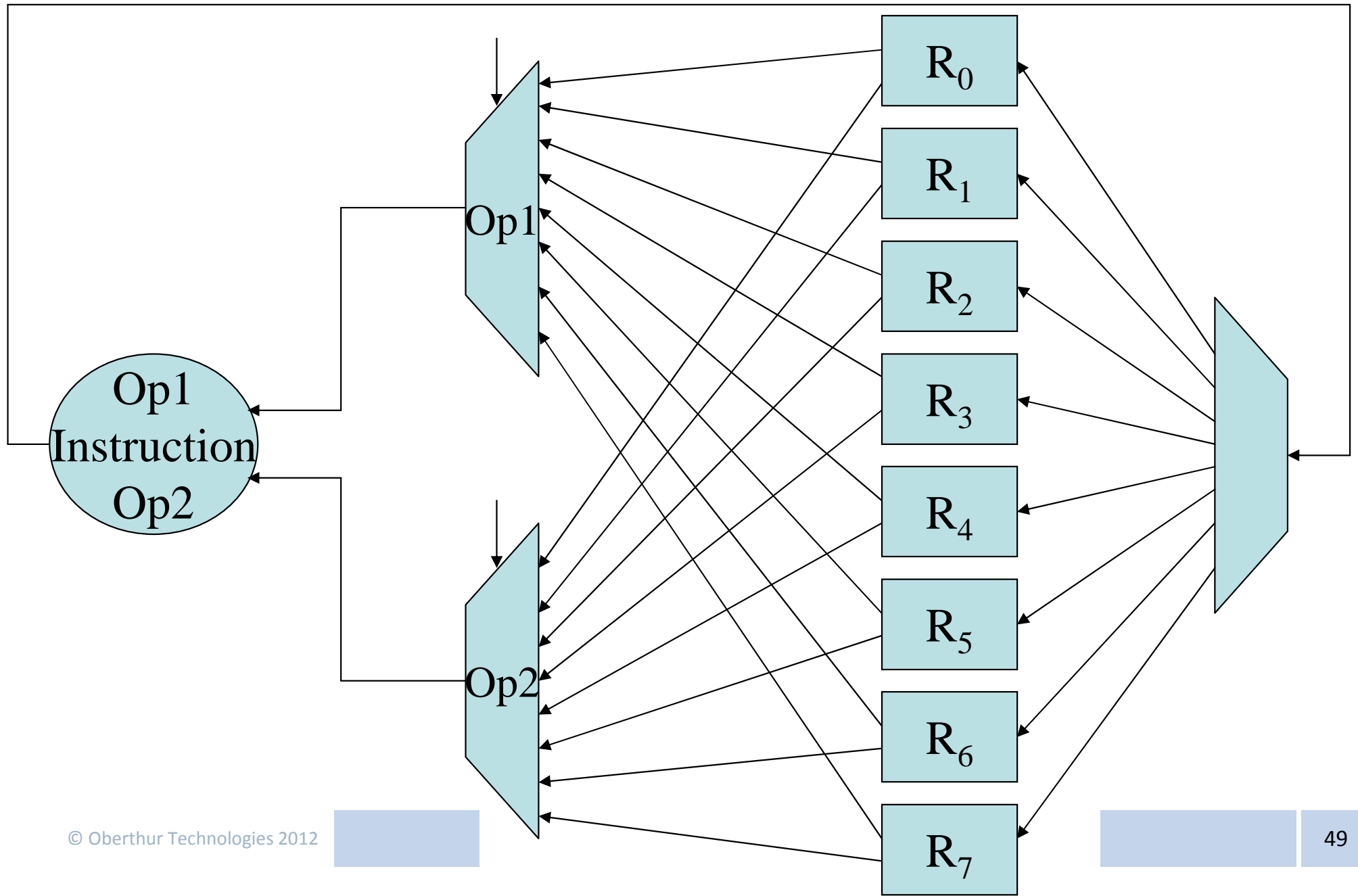
Leakage model	HW	HD
Developer		

- First Challenge:
 - Need of a generic method to convert a scheme resistant in the HW-model into a scheme resistant in the HD-model
- Moreover, practice is not so easy...

- Introduction
- Countermeasures Description
- **Implementation Difficulties**
 - Countermeasure Conversion
 - **On the Leakage Models**
- (Not So) Futuristic Attacks

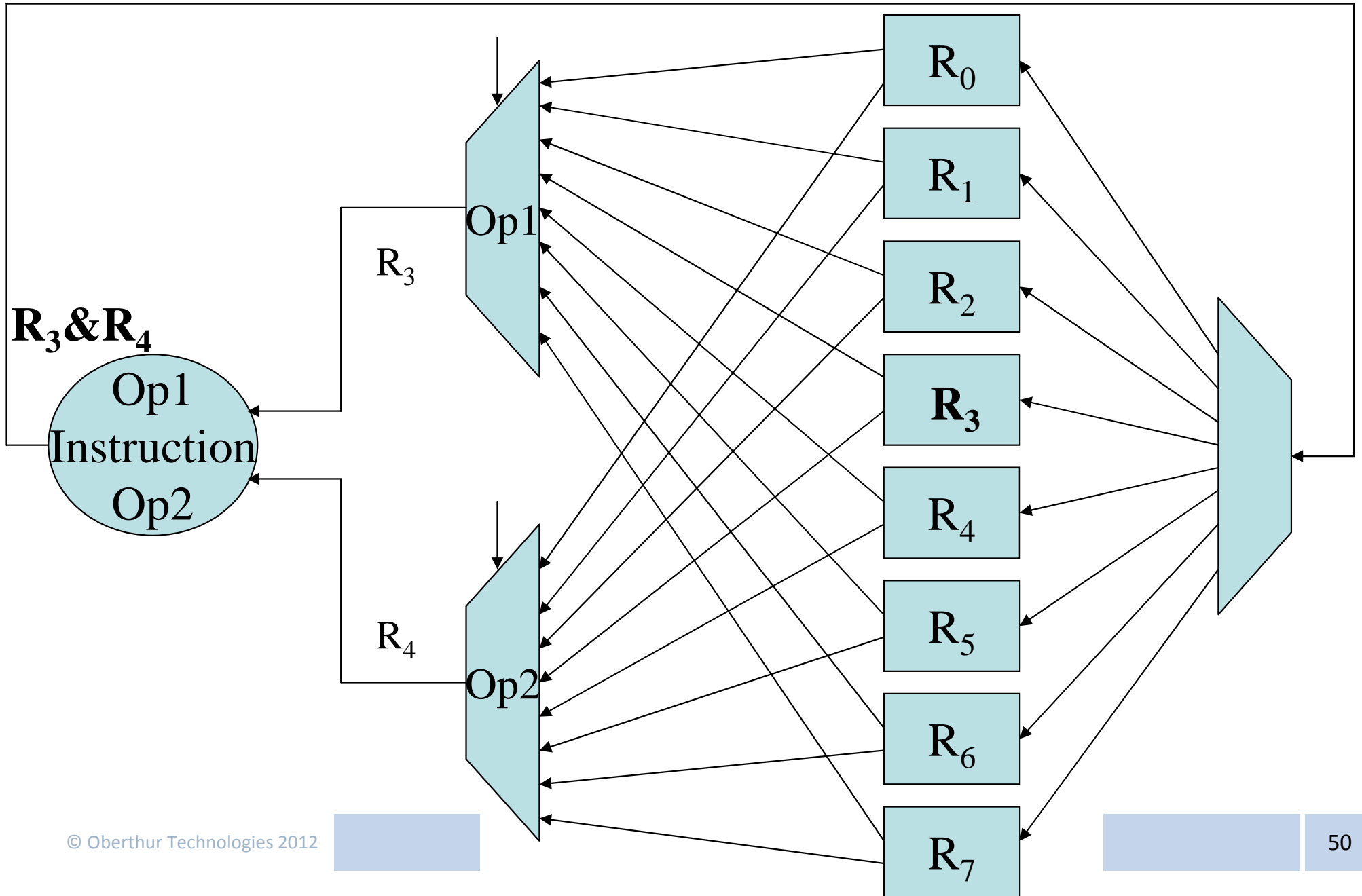


- Leakage model of the processor:
 - Access bus leaking in the HD model
 - Registers leaking in the HW model
- Code example :
 - $R_3 \leftarrow R_3 \& R_4$
 - $R_1 \leftarrow R_1 \oplus R_6$
- What are the leakages observed during the XOR operation?



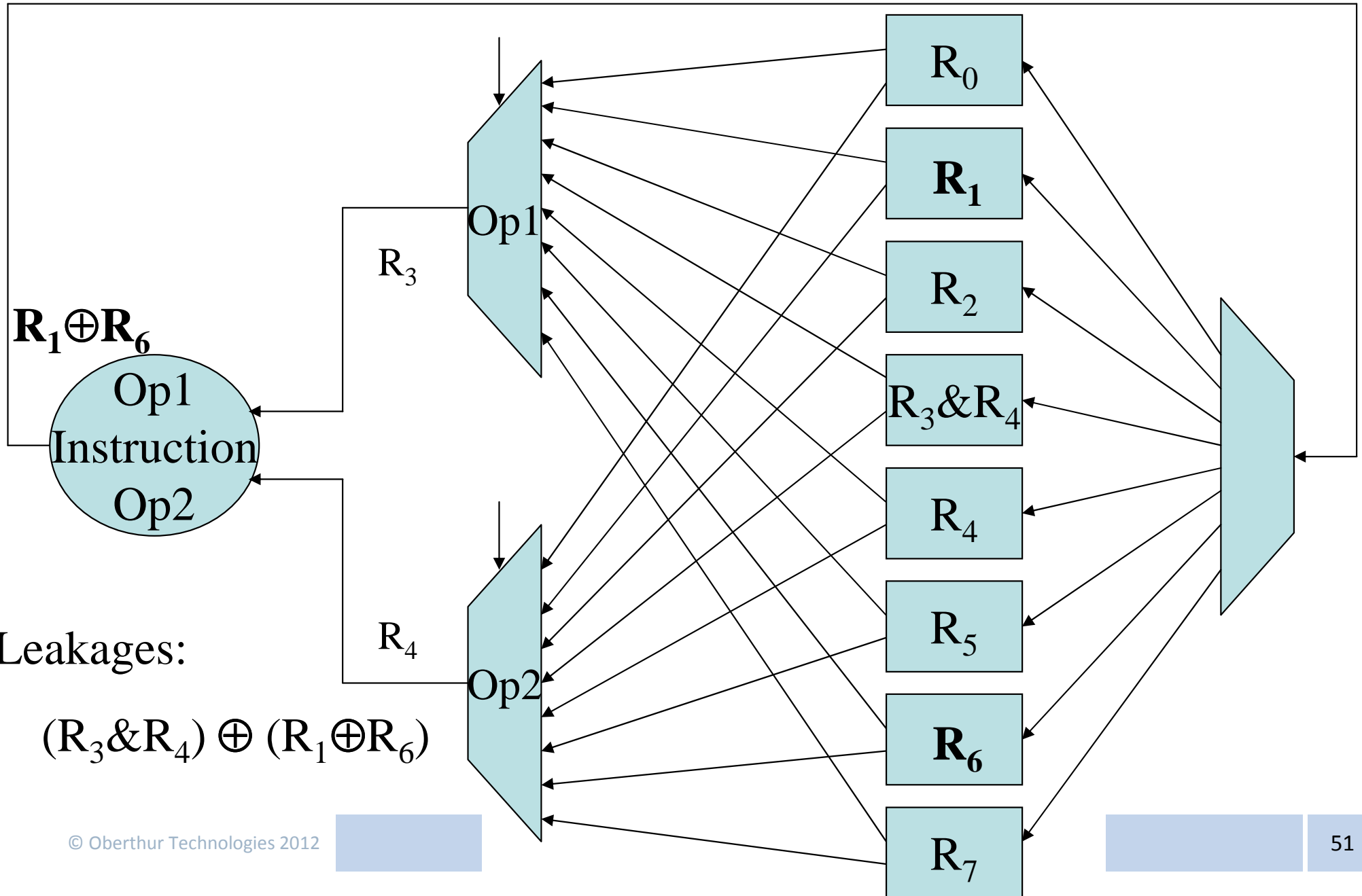


R₃&R₄

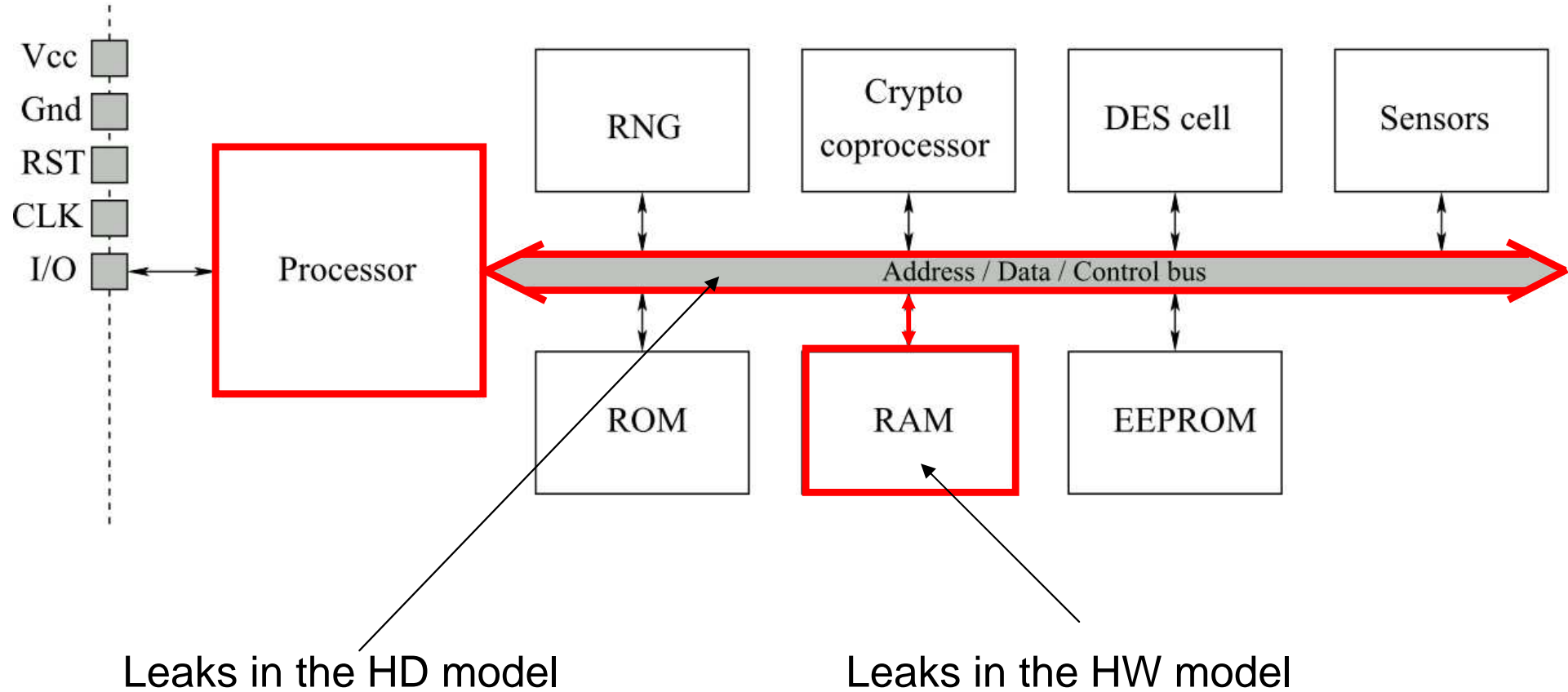




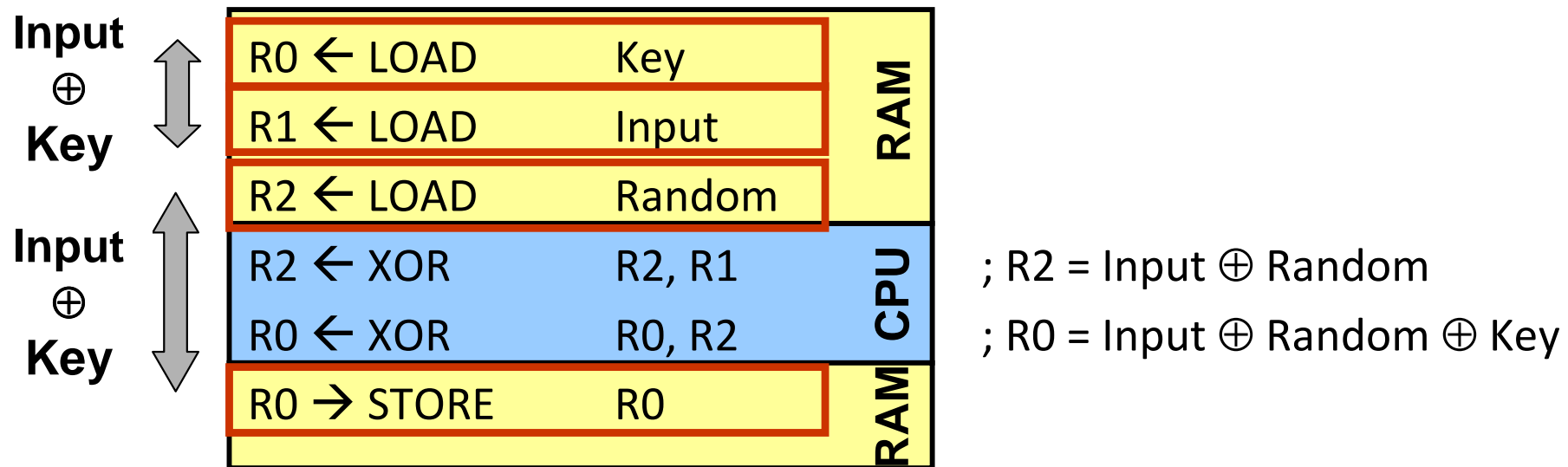
R₃&R₄



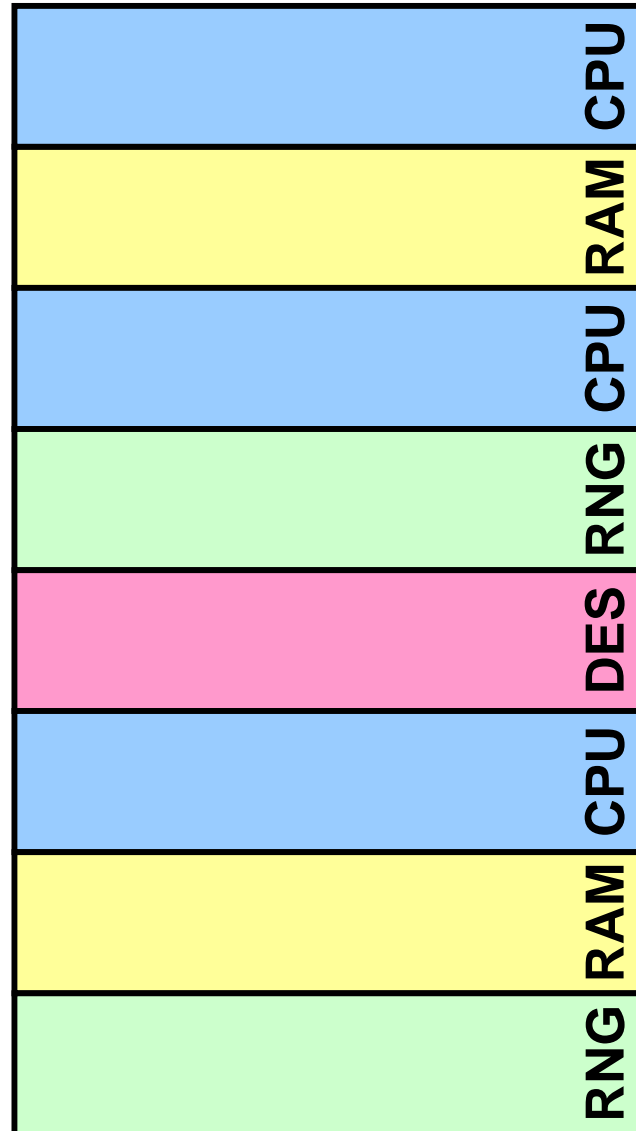
- Code example :
 - $R_3 \leftarrow R_3 \& R_4$
 - $R_1 \leftarrow R_1 \oplus R_6$
 - What are the leakages observed during the XOR operation?
 - $R_1 \oplus R_3$
 - $R_4 \oplus R_6$
 - $(R_3 \& R_4) \oplus (R_1 \oplus R_6)$
 - $R_1 \oplus R_6$
- ⇒ Complex leakages



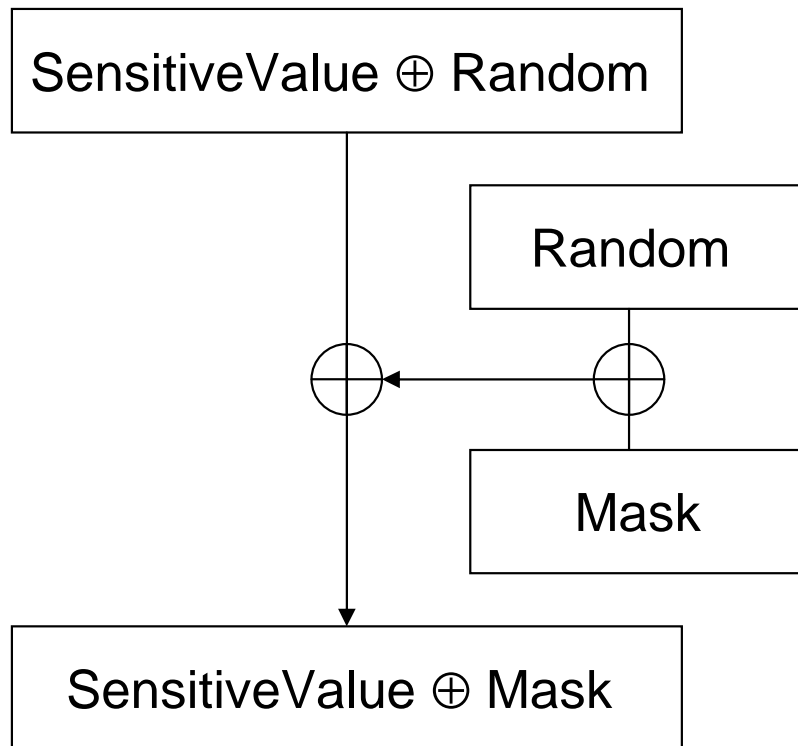
- Now let us observe a piece of code using RAM access:



- Identify leakages when using interlaced dedicated code is a difficult task



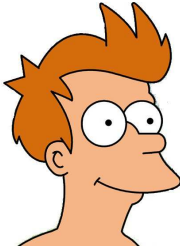


- An example from real life:



- $R_1 \leftarrow R_1 \oplus R_3$
 $R_0 \leftarrow R_0 \oplus R_1$
- Analysis:
 - $R_1 \leftarrow \text{XOR } R_1, R_3$
 $R_1 = \text{Mask}$
 $R_3 = \text{Random}$
 $\rightarrow R_1 = \text{Mask} \oplus \text{Random}$
 - $R_0 \leftarrow \text{XOR } R_0, R_1$
 $R_0 = \text{SensitiveValue} \oplus \text{Random}$
 $R_1 = \text{Mask} \oplus \text{Random}$
 $\rightarrow R_0 = \text{SensitiveValue} \oplus \text{Mask}$

SensitiveValue

- First Conclusion:
 - It is very difficult for a developer to adapt the security of a scheme prove to be secure in the HW-model into a more realistic HD-model

Leakage model	HW	HD	Real Device
Developer			

- First Challenge:
 - Need of a generic method to convert a scheme resistant in the HW-model into a scheme resistant in the HD-model
- Moreover, practice is not so easy...
 - The different modules can have different leakage models
 - The leakages of the low-level hardware design can be very complex

- Second Conclusion:
 - Implementing efficient software side-channel countermeasures requires:
 - The precise architecture of the chip
 - A deep analysis of the possible leakages
 - If the architecture is unknown:
 - Implement second-order countermeasures to resist first-order attacks
 - ➔ This does not prevent each and every kind of Hamming distance leakages!
- Second Challenge: How to characterise each and every leakages of a chip ?

- What about the validation?
 - Code review?



Good Code



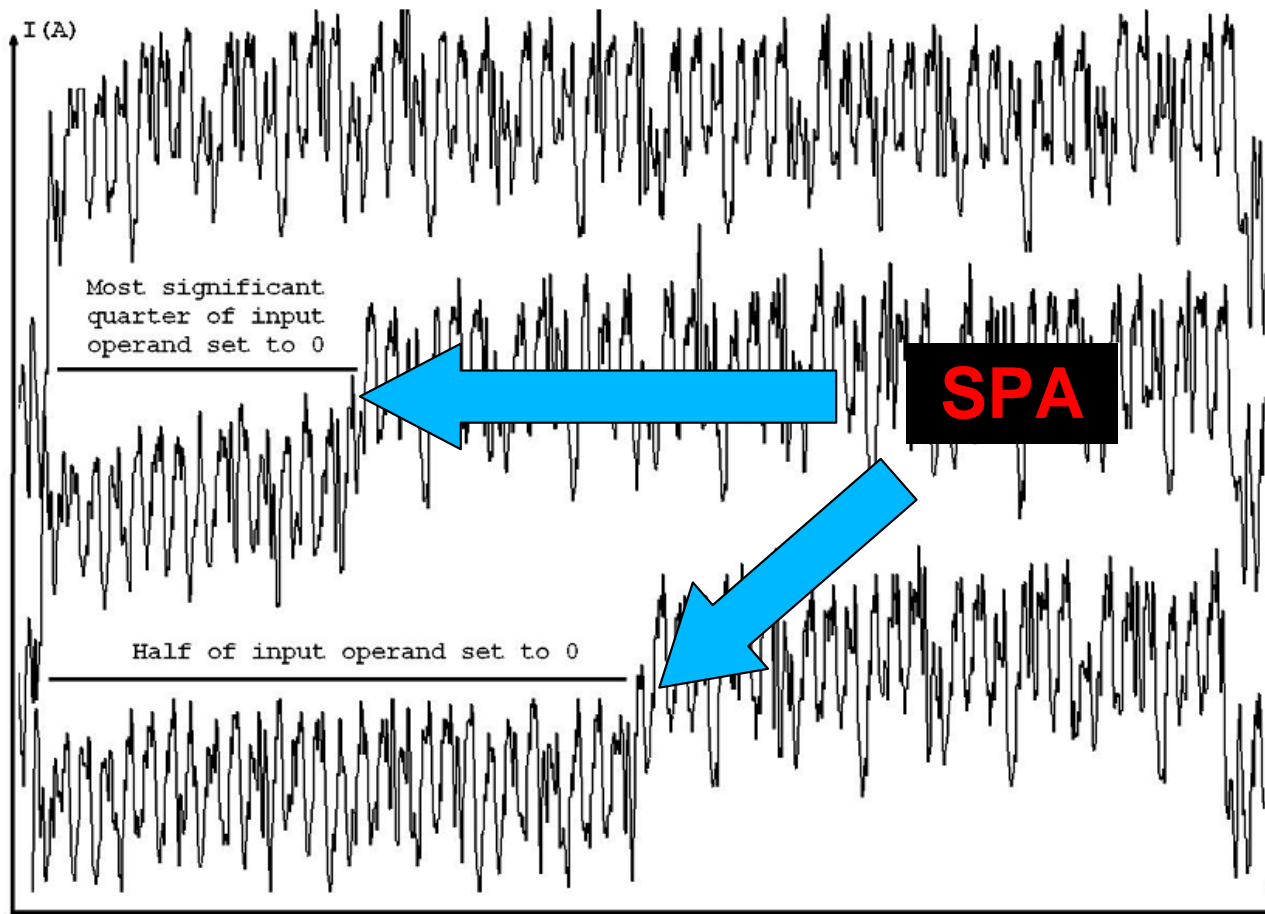
Bad Code

➔ Difficult not to miss anything amongst 10 000 source code lines...

- Practical validation?
 - ➔ Very time-costly

- Introduction
- Countermeasures Description
- Implementation Difficulties
- **(Not So) Futuristic Attacks:**
 - **On the need of FA-resistance for SCA-countermeasure**
 - **And Vice Versa**

- Fact: Each countermeasure is usually focused to prevent either SCA or FA
- Combined attacks: combine both kind of attacks to defeat a classical set of countermeasures
- Let us see an example on a “fully” secure RSA [AVFM07]



Square or Mult

Random values for Operands

Mult A x B

Quarter of A set to 0

Mult A x B

Half of A set to 0

Let us analyse a RSA implementation secure against

- SCA: by using Atomicity principle and Masking method
- FA: by using Signature verification

Algorithm 1 RSA signature implementation.

INPUTS: $m, d = (d_{n-1}, d_{n-2}, \dots, d_0)_2, N$

OUTPUT: $m^d \bmod N$

Get r_0, r_1 two random values

$R_0 \leftarrow 1 + r_0N$

$R_1 \leftarrow m + r_0N \bmod r_1N$

$(k, i) \leftarrow (0, n - 1)$

while $i \geq 0$ **do**

$R_0 \leftarrow R_0 \cdot R_k \bmod r_1N$

$k \leftarrow k \oplus d_i$

$i \leftarrow i - \neg k$

$S \leftarrow R_0 \bmod N$

$m' \leftarrow S^e \bmod N$

if $m \neq m'$ **then**

 securityAction()

else

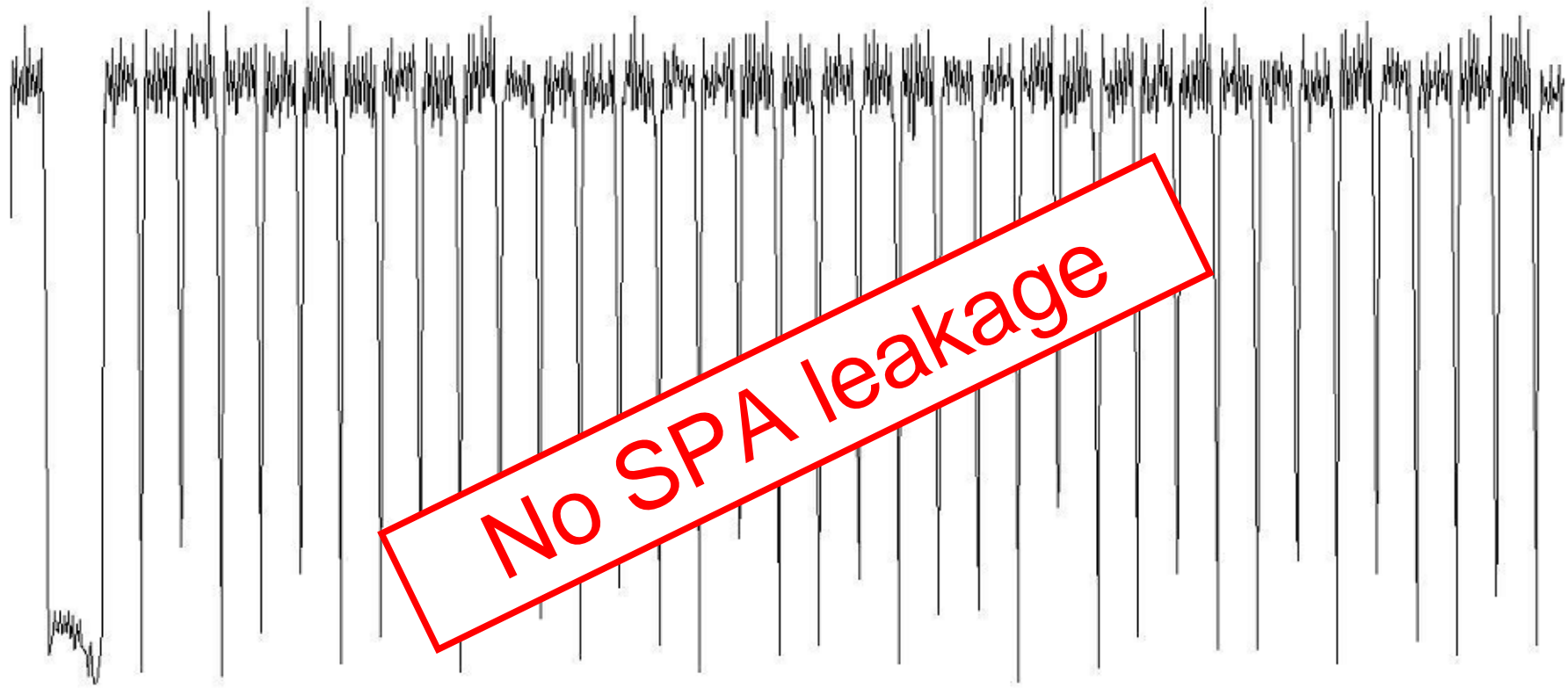
return S

Masking

Atomic loop

Signature verification

- It seems good...



Let us analyse a RSA implementation secure against

- SCA: by using Atomicity principle and Masking method
- FA: by using Signature verification

Algorithm 1 RSA signature implementation.

INPUTS: $m, d = (d_{n-1}, d_{n-2}, \dots, d_0)_2, N$

OUTPUT: $m^d \bmod N$

Get r_0, r_1 two random values

$R_0 \leftarrow 1 + r_0N$

$R_1 \leftarrow m + r_0N \bmod r_1N$

$(k, i) \leftarrow (0, n - 1)$

while $i \geq 0$ **do**

$R_0 \leftarrow R_0 \cdot R_k \bmod r_1N$

$k \leftarrow k \oplus d_i$

$i \leftarrow i - \neg k$

$S \leftarrow R_0 \bmod N$

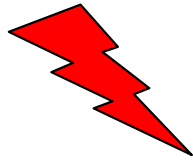
$m' \leftarrow S^e \bmod N$

if $m \neq m'$ **then**

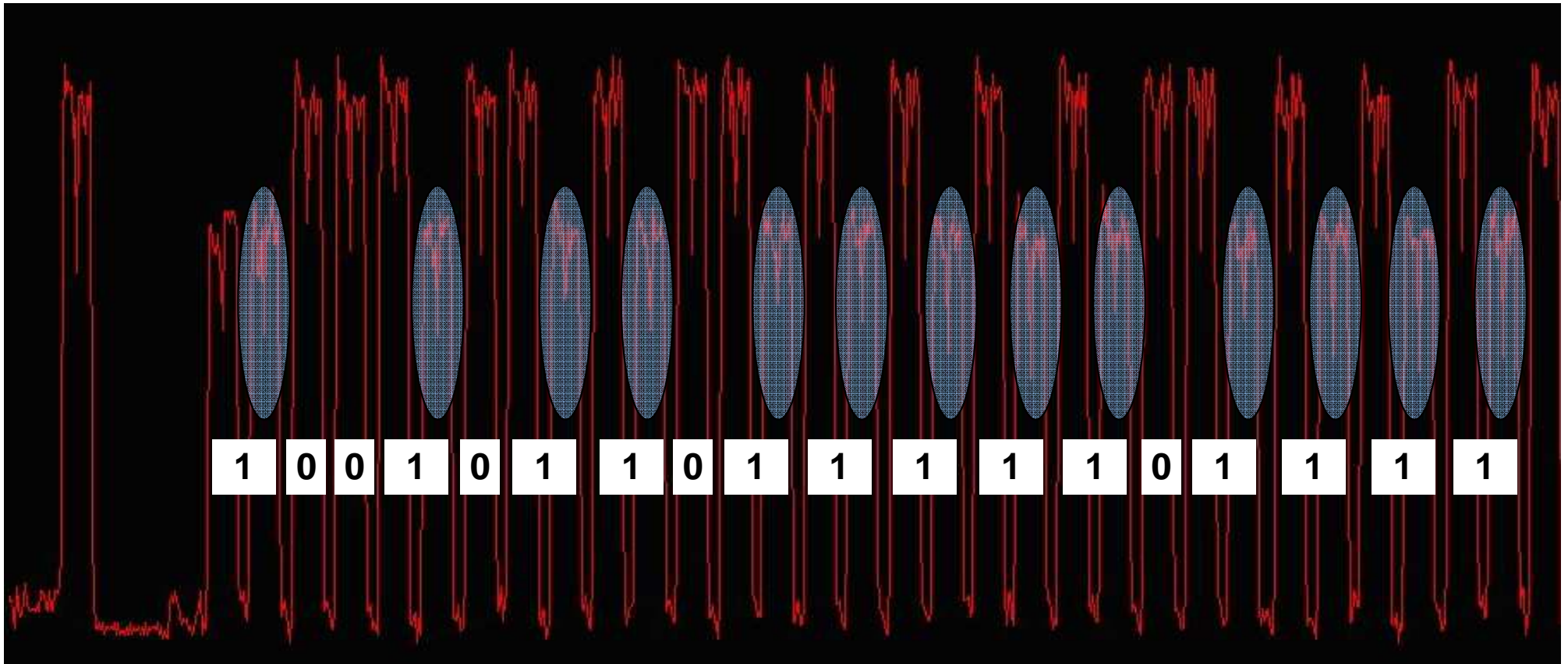
 securityAction()

else

return S



If this operation is disturbed
such that R_1 is set to 0 ?



- Multiplication operations by R_1 are now revealed!
- Full secret exponent obtained with only one successful fault!

Let us analyse an RSA implementation secure against


- SCA: by using Atomicity principle and Masking method
- FA: by using Signature verification

Algorithm 1 RSA signature implementation.

 INPUTS: $m, d = (d_{n-1}, d_{n-2}, \dots, d_0)_2, N$

 OUTPUT: $m^d \bmod N$

 Get r_0, r_1 two random values

 $R_0 \leftarrow 1 + r_0 N$
 $R_1 \leftarrow m + r_0 N \bmod r_1 N$
 $(k, i) \leftarrow (0, n - 1)$
while $i \geq 0$ **do**
 $R_0 \leftarrow R_0 \cdot R_k \bmod r_1 N$
 $k \leftarrow k \oplus d_i$
 $i \leftarrow i - \neg k$
 $S \leftarrow R_0 \bmod N$
 $m' \leftarrow S^e \bmod N$
if $m \neq m'$ **then** 

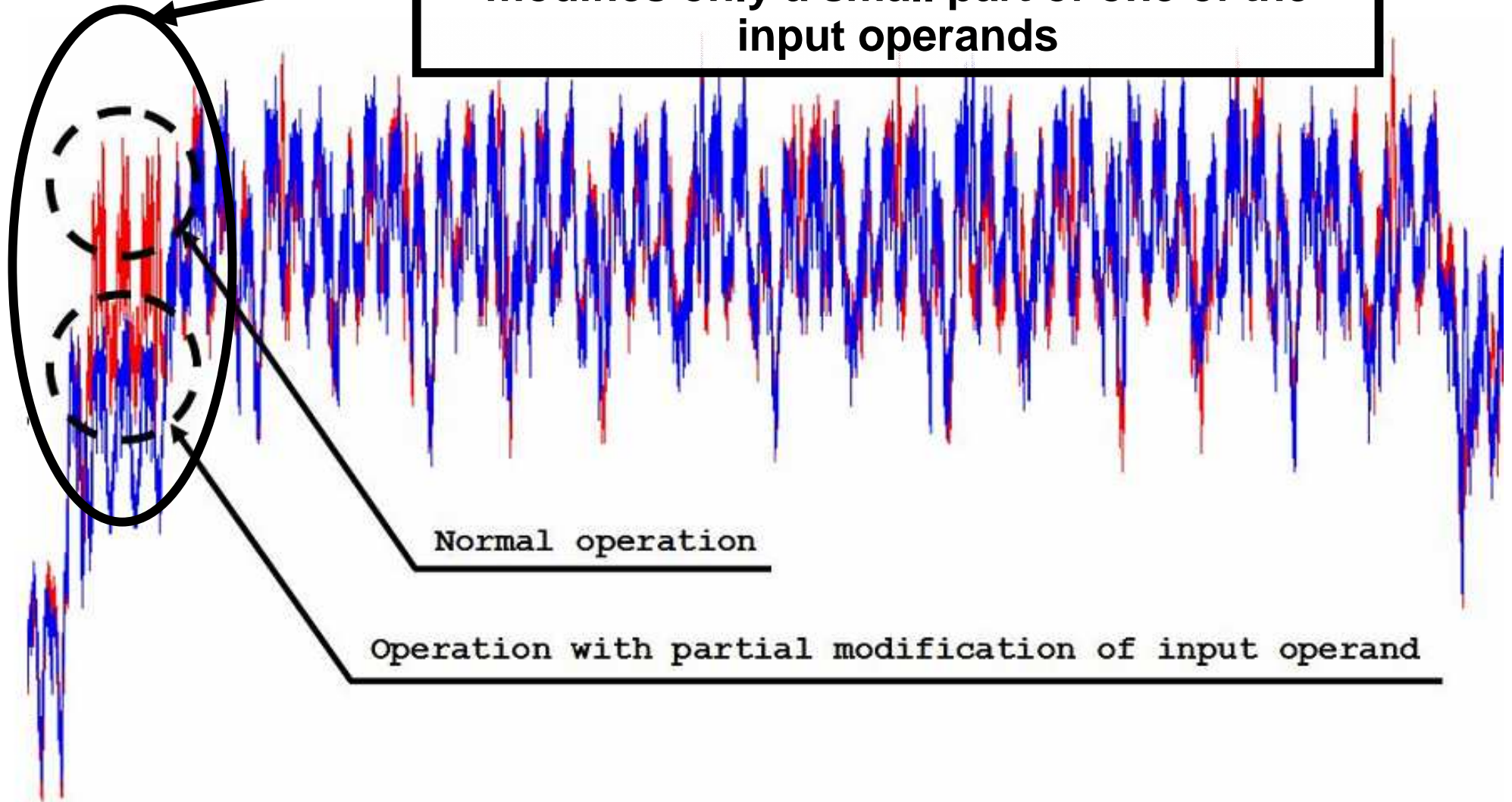
 securityAction()

else

 return S

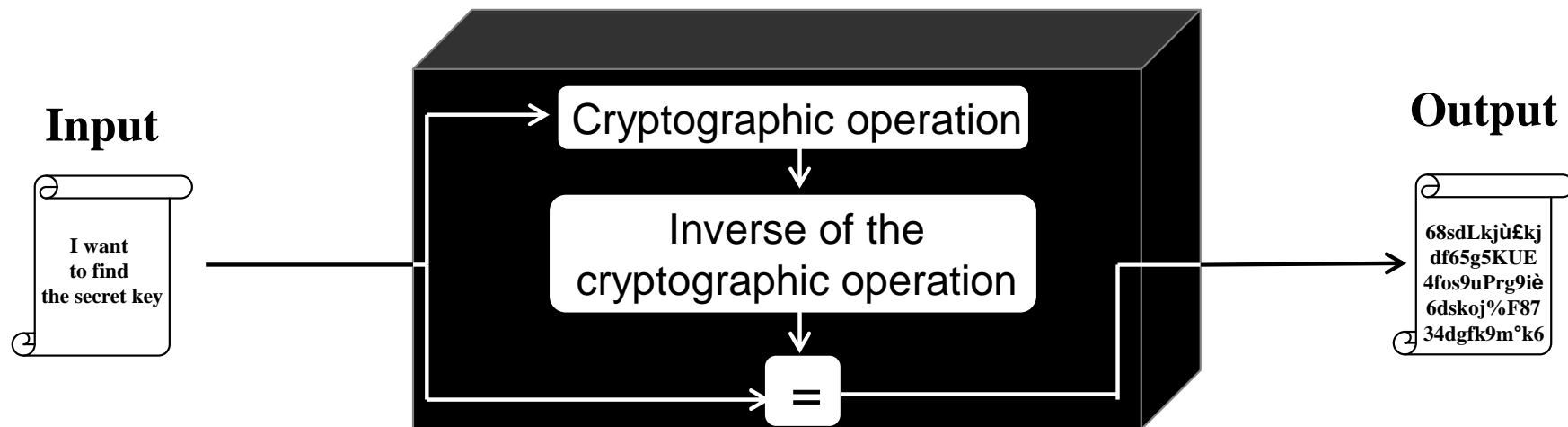
The fault will be detected but
 it is too late!!

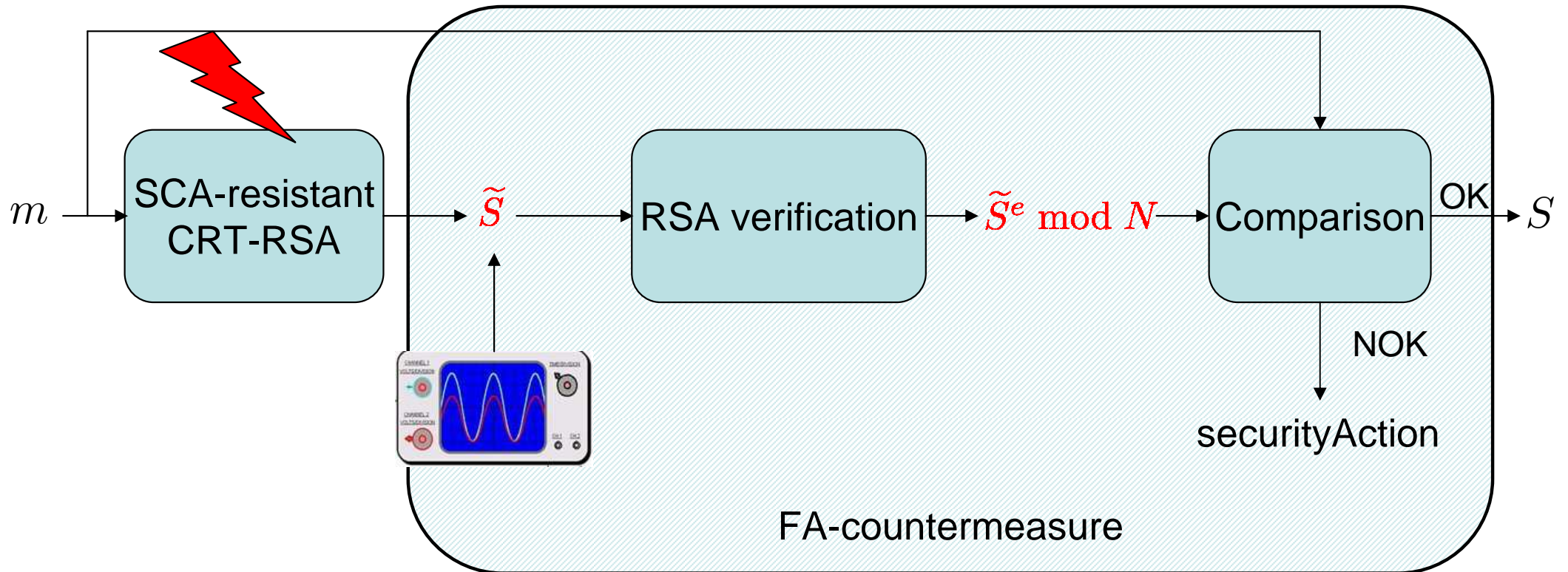
The fault could be exploited even if it modifies only a small part of one of the input operands

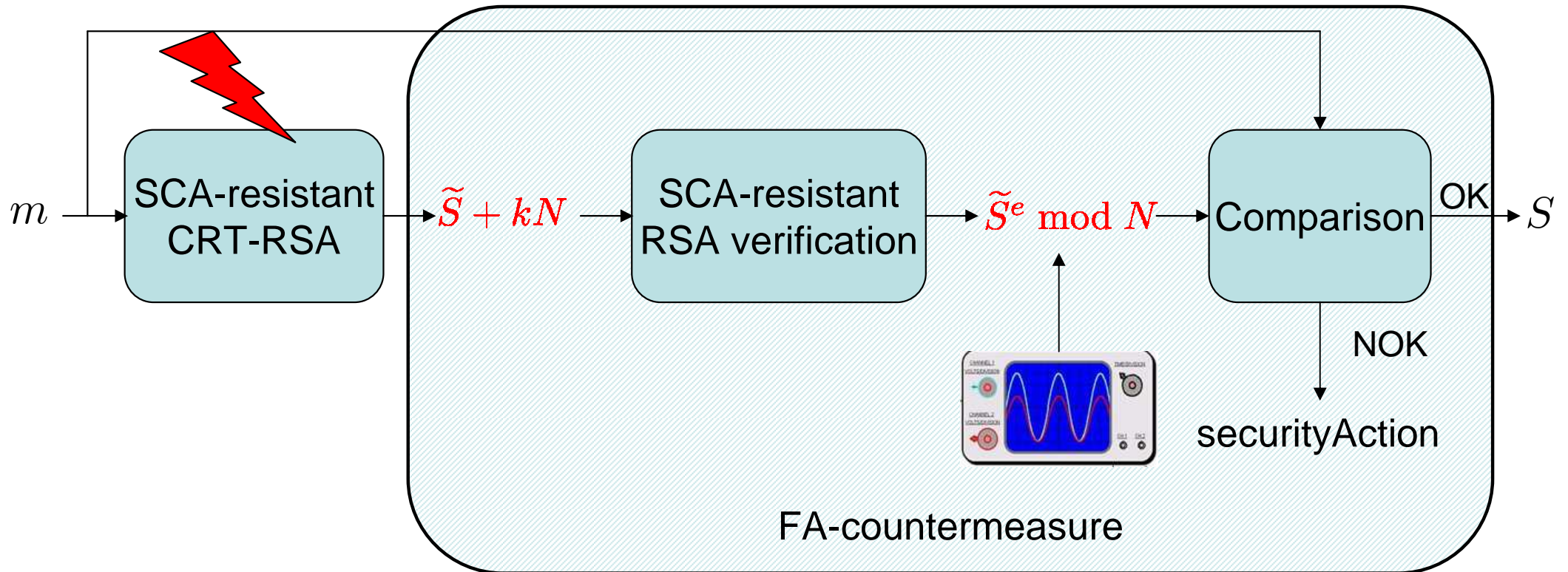


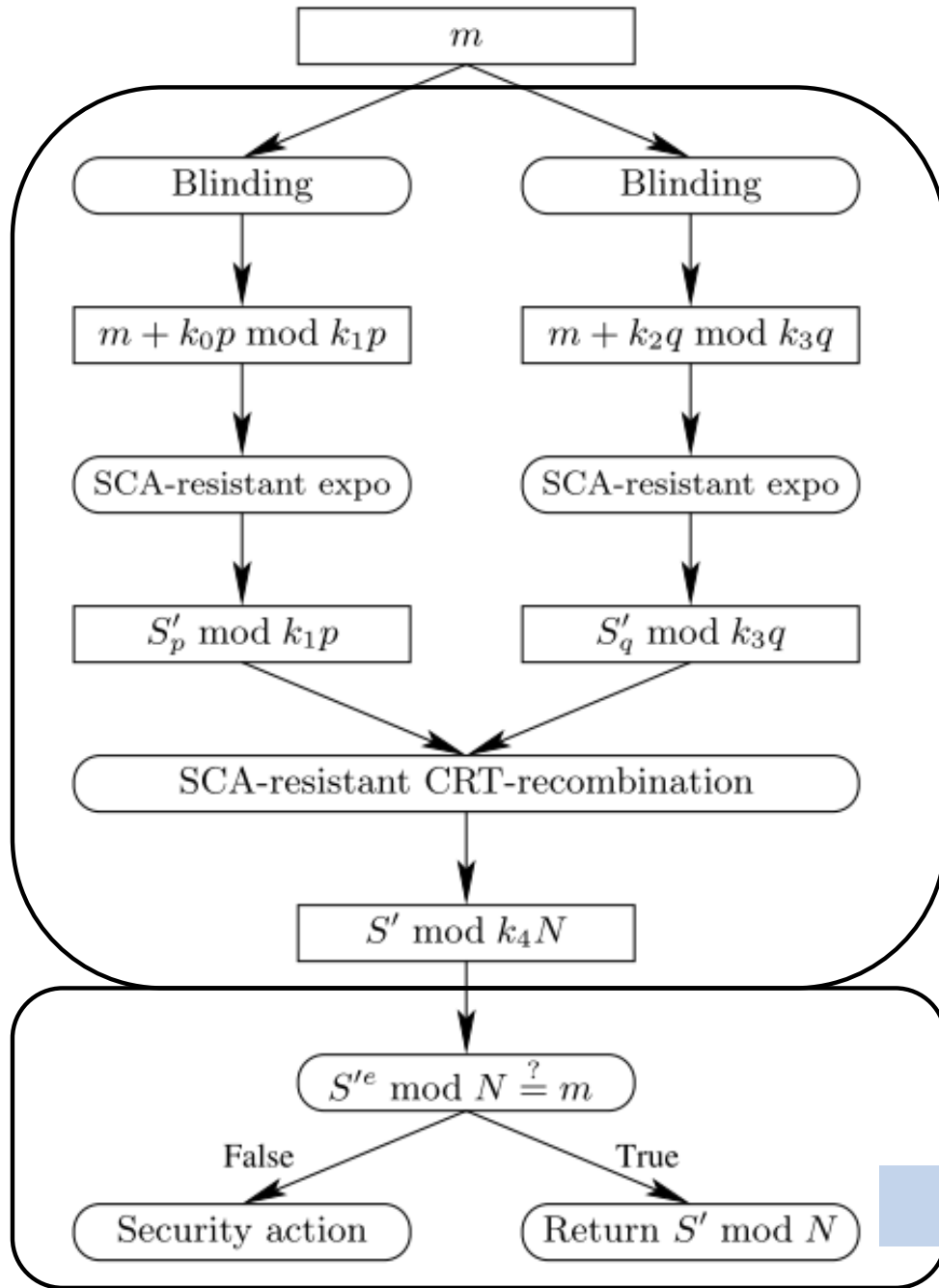
- Introduction
- Countermeasures Description
- Implementation Difficulties
- **(Not So) Futuristic Attacks:**
 - On the need of FA-resistance for SCA-countermeasure
 - **And Vice Versa**

- We proved that a SCA-countermeasure must resist Fault Attacks
- Let us see that the opposite is also true

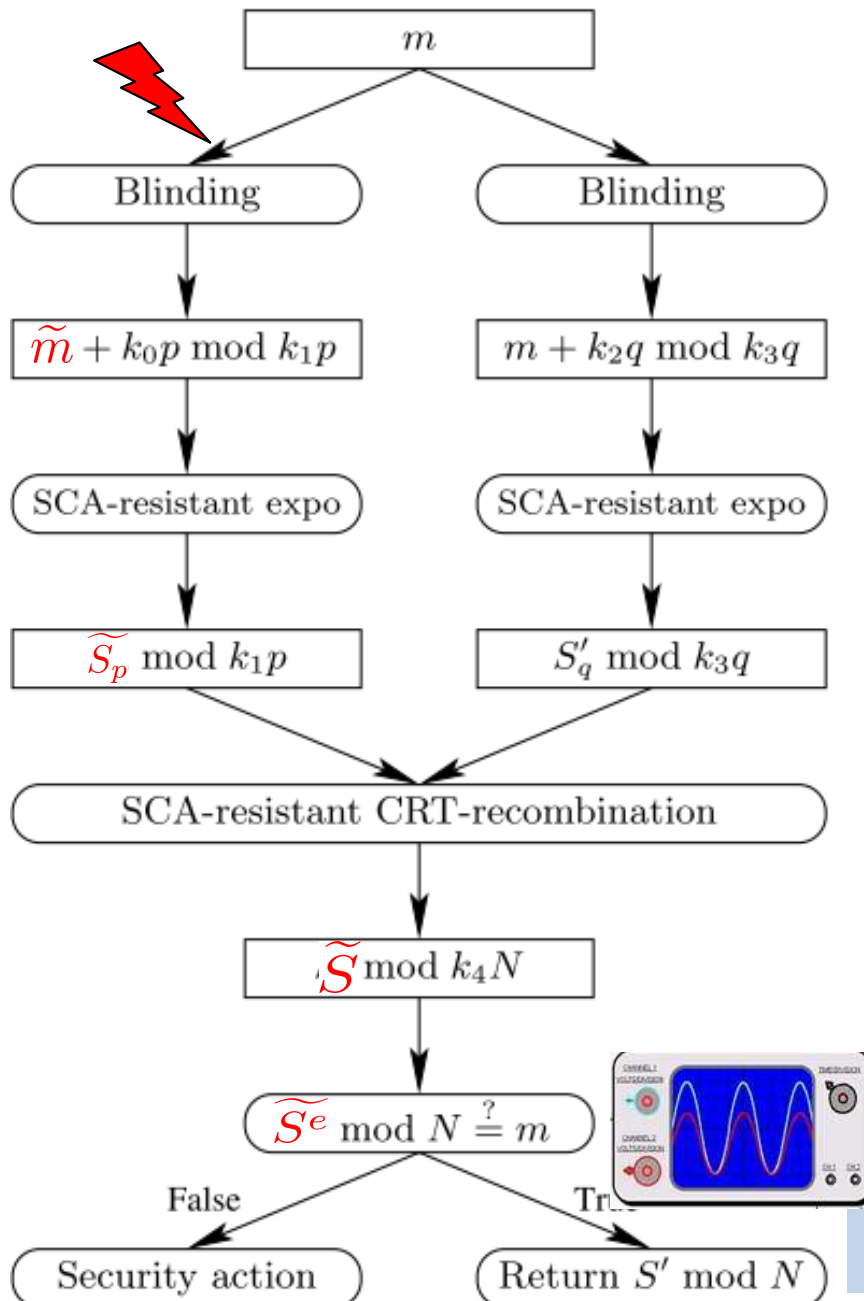







 SCA-resistant
 CRT-RSA

FA-verification



- If m is disturbed such that $\tilde{m} = m + \epsilon$ then we have:

$$\tilde{S}^e \equiv m + \epsilon q i_q \pmod{N}$$
- Therefore if ϵ is constant the attacker can mount a DPA on $\tilde{S}^e \pmod{N}$ during the signature verification.
- In this case, signature verification must also resist SCA !!

- Conclusion:
 - Adapt countermeasures proved to be secure in one model into another model could be a very difficult task
 - A chip does not have only one leakage model!
 - Countermeasures must take into account Combined Attacks
- Present challenges:
 - Need for a generic method to convert a scheme resistant in the HW-model into a scheme resistant in the HD-model
 - How to characterise each and every leakage of a chip ?
- Future challenges:
 - Design countermeasures which are proved to be resistant not only to one kind of attack but to both SCA and FA !!

