Key Reconciliation Protocols for Error Correction of Silicon PUF Responses

Brice Colombier[1], **Lilian Bossuet**[2], **David Hély**[3]

[1]CEA-Tech DPACA, Gardanne — France
[2]Laboratoire Hubert Curien, Saint-Étienne — France
[3]LCIS, Grenoble Institute of Technology, Valence — France

30 mai 2018

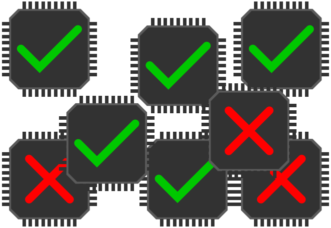
Journées Nationales 2018 Pré-GDR Sécurité Informatique

IoT devices

- Mutual identification
- Authentication

IP protection

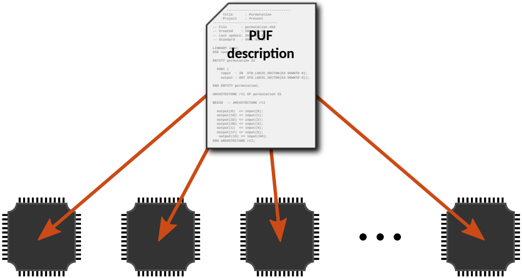
- ICs identification
- IP cores identification



Need for a hardware identifier as root of trust

Physical Unclonable Functions

PUFs as unique identifiers

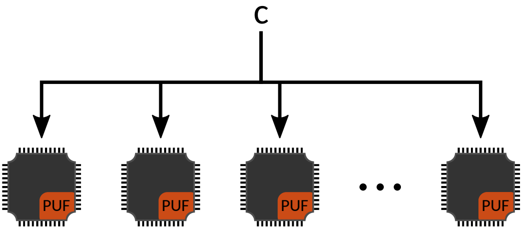


Principle:

Extract entropy from **process variations**.

Aim:

Provide a unique, per-device ID, thanks to the **inter-device** uniqueness.

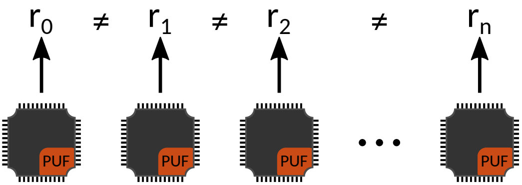


Principle:

Extract entropy from **process variations**.

Aim:

Provide a unique, per-device ID, thanks to the **inter-device** uniqueness.



Different responses to the **same** challenge.

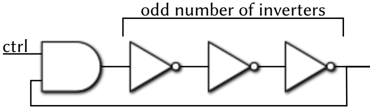
Principle:

Extract entropy from **process variations**.

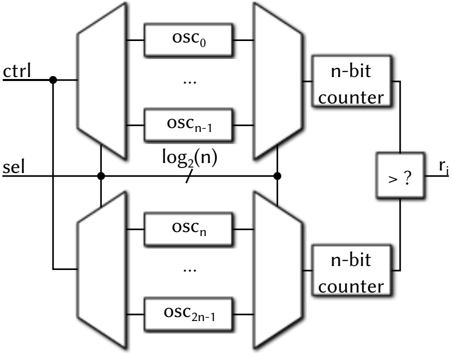
Aim:

Provide a unique, per-device ID, thanks to the **inter-device** uniqueness.

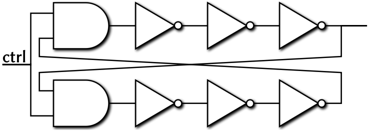
RO cell



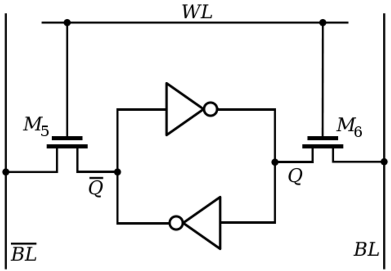
PUF architecture

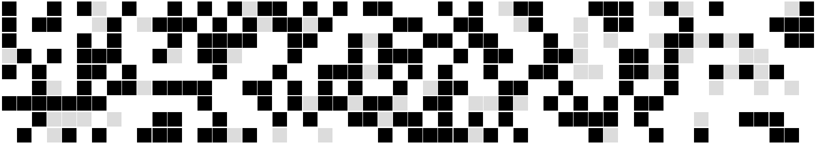


TERO cell



SRAM PUF





Problem:

PUF responses to the **same** challenge **change** over time.

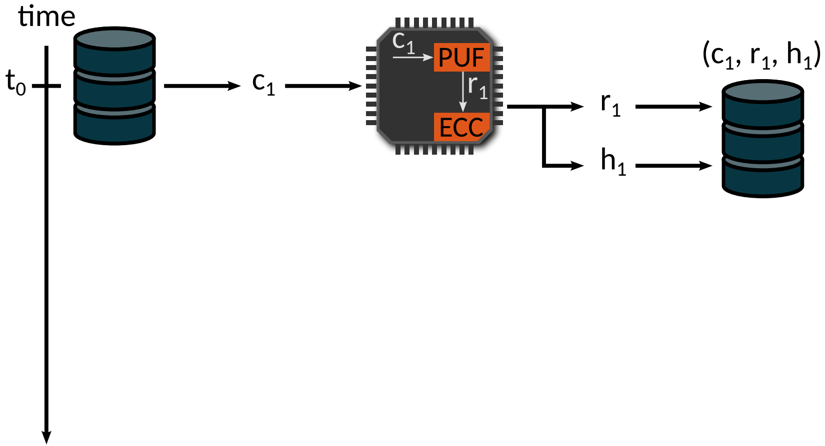
This variation depends on multiple parameters:

- PUF architecture,
- Process node,
- Aging,
- Temperature,
- Environment...

➔ The PUF response cannot be used as a **reliable identifier**.

Solution

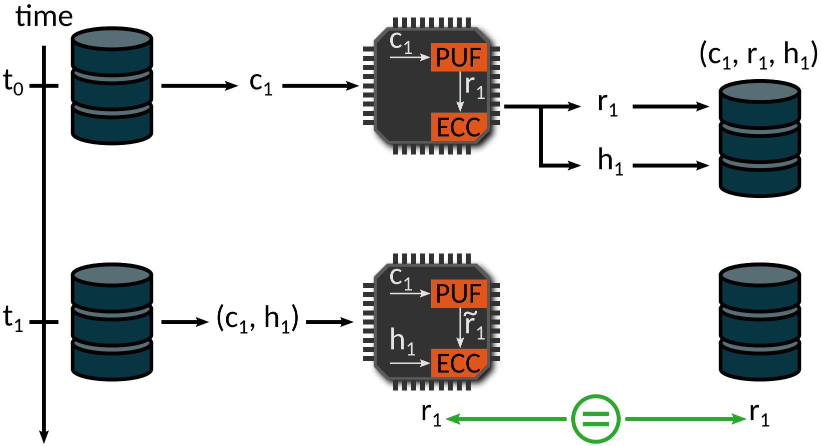
Apply a technique of **error correction** to the PUF response



Error correction for PUF responses

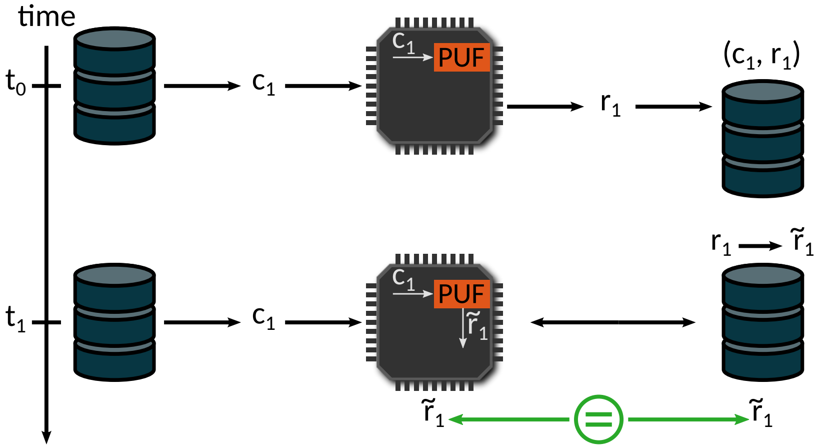
Solution

Apply a technique of **error correction** to the PUF response



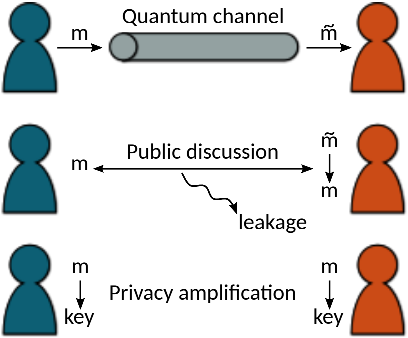
Solution

Apply a technique of **error correction** to the PUF response



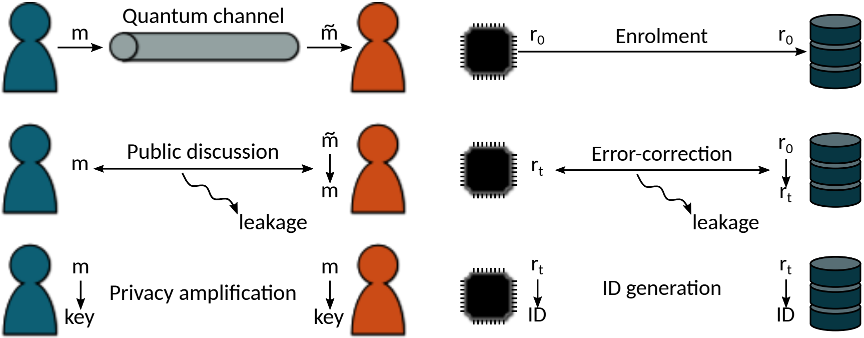
The CASCADE key reconciliation protocol

CASCADE introduced in 1993 by Brassard and Salvail [1]



[1] Gilles Brassard and Louis Salvail. "Secret-Key Reconciliation by Public Discussion". *EUROCRYPT*. 1993, pp. 410–423.

CASCADE introduced in 1993 by Brassard and Salvail [1]



This could be used to derive keys
from slightly different PUF responses.

[1] Gilles Brassard and Louis Salvail. “Secret-Key Reconciliation by Public Discussion”. *EUROCRYPT*. 1993, pp. 410–423.

One pass

- Perform **parity checks** on **blocks** of the PUF response,
- **Isolate** the errors using **binary search** and correct them,
- Check **current** parity of blocks and **backtrack**,
- **Increase** the block size and **shuffle** the response **randomly**.

One pass

- Perform **parity checks** on **blocks** of the PUF response,
- **Isolate** the errors using **binary search** and correct them,
- Check **current** parity of blocks and **backtrack**,
- **Increase** the block size and **shuffle** the response **randomly**.

Parameters

- Initial block size,
- Number of passes,
- Block size multiplier.

One pass

- Perform **parity checks** on **blocks** of the PUF response,
- **Isolate** the errors using **binary search** and correct them,
- Check **current** parity of blocks and **backtrack**,
- **Increase** the block size and **shuffle** the response **randomly**.

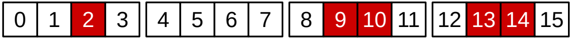
Parameters

- Initial block size,
- Number of passes,
- Block size multiplier.

Information leakage associated with the public discussion

For an *n*-bit response split into *k*-bit blocks:

- Parity checks: n/k-bit leakage.
- Binary search: $\log_2(k)$-bit leakage.



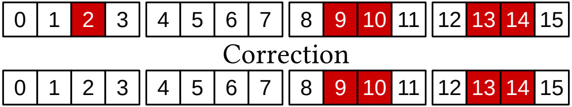
Blocks of even relative parity:

$\varnothing$

Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



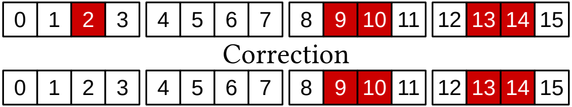
Blocks of even relative parity:

$\varnothing$

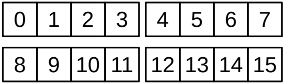
Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



Blocks of even relative parity:

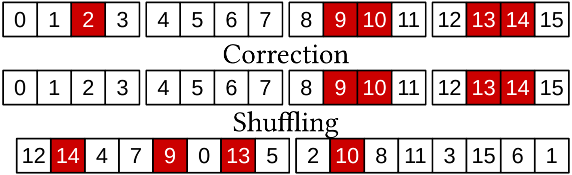


Blocks of odd relative parity:

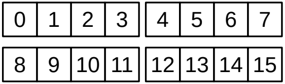
$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

The CASCADE protocol



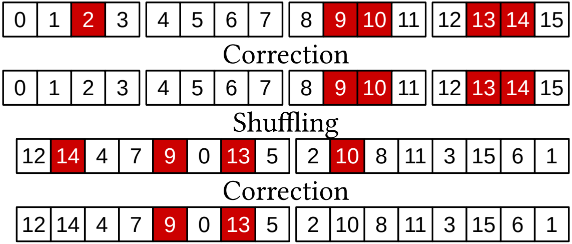
Blocks of even relative parity:



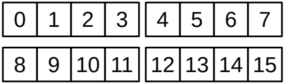
Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



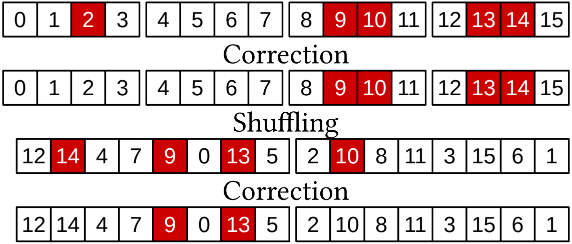
Blocks of even relative parity:



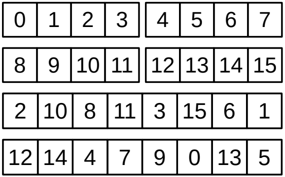
Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



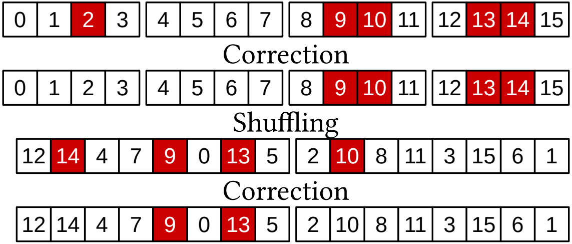
Blocks of even relative parity:



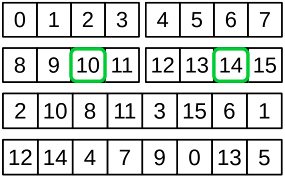
Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



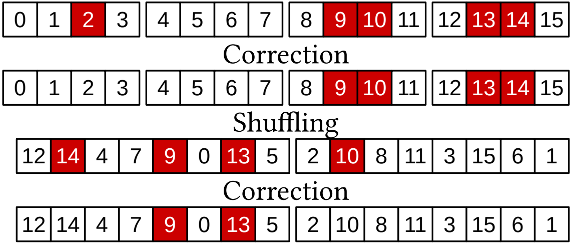
Blocks of even relative parity:



Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



Blocks of even relative parity:

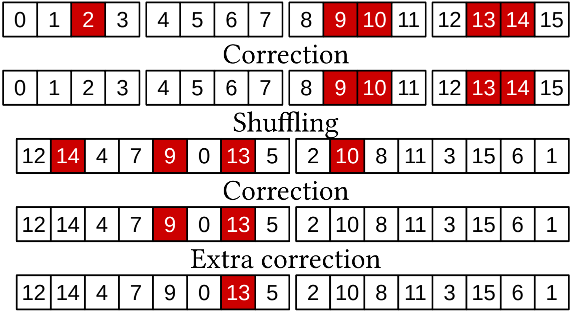
0	1	2	3	4	5	6	7
2	10	8	11	3	15	6	1
12	14	4	7	9	0	13	5

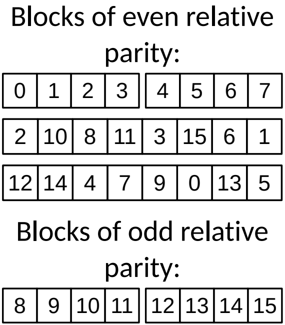
Blocks of odd relative parity:

8	9	10	11	12	13	14	15

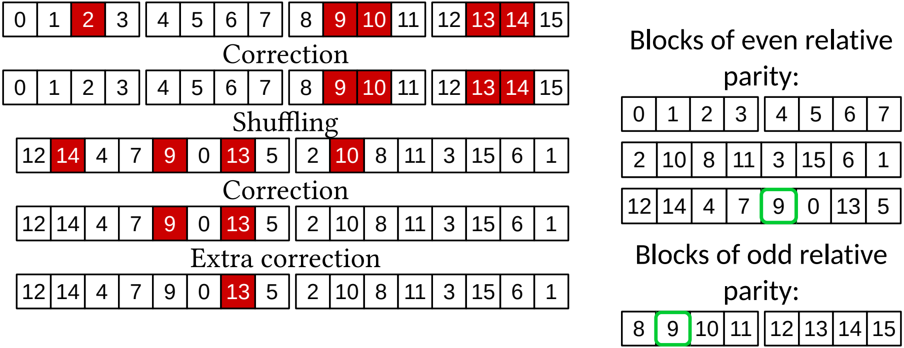
Relative parity: $$P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$$

The CASCADE protocol

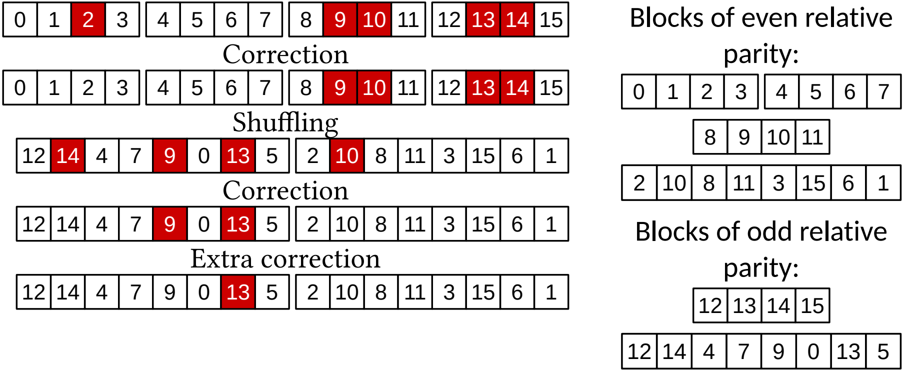




Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

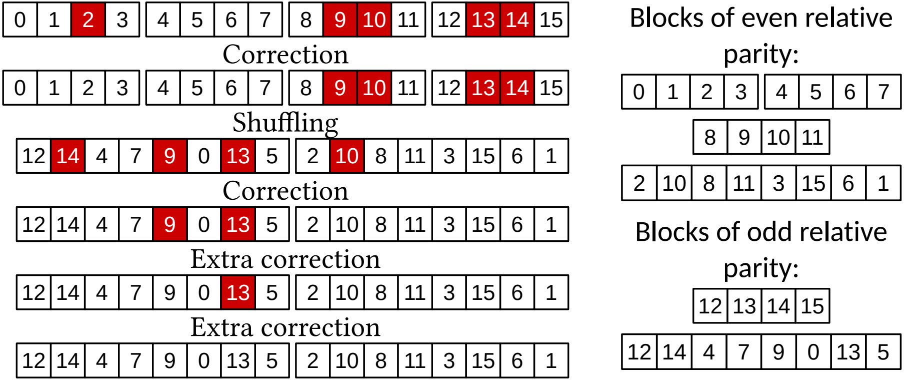


Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

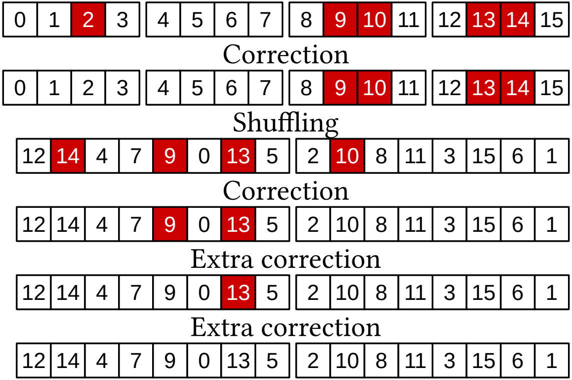


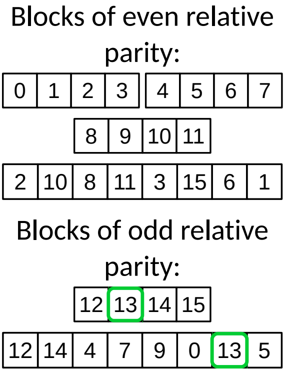
Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

The CASCADE protocol

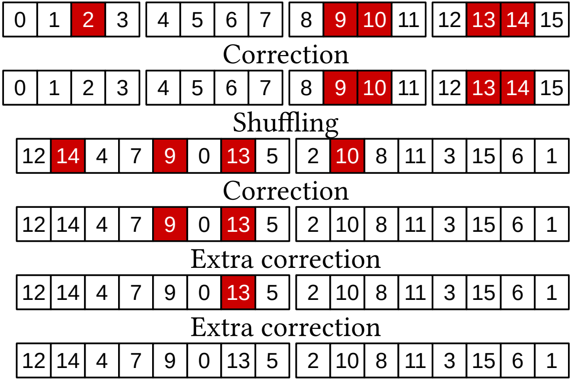


Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

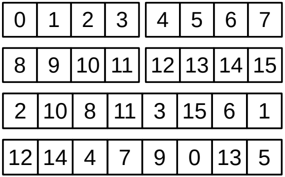




Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$



Blocks of even relative parity:



Blocks of odd relative parity:

$\varnothing$

Relative parity: $P_r(B_0, B_t) = \underbrace{\left(\bigoplus_{i=0}^{m-1} r_0[B_0[i]]\right)}_{\text{Parity of } B_0} \oplus \underbrace{\left(\bigoplus_{i=0}^{m-1} r_t[B_t[i]]\right)}_{\text{Parity of } B_t}$

Associated information leakage

Two ways of leaking information:

- Relative parity computations,
  - 1 bit.
- `CONFIRM` executions on an *n*-bit block.
  - $log_2(n)$ bits.

Two ways of leaking information:

- Relative parity computations,
  - 1 bit.
- `CONFIRM` executions on an *n*-bit block.
  - $log_2(n)$ bits.

Example:

128-bit response, $\varepsilon = 0.05 \rightarrow 7$ errors.

- 1^{st} pass: 8-bit blocks, 4 errors corrected.
- 2^{nd} pass: 16-bit blocks, 3 errors corrected.

Leakage: $\frac{128}{8} + 4 \times log_2(8) + \frac{128}{16} + 3 \times log_2(16) = 48$ bits.

The final effective length of the response is 128 - 48 = **80 bits**.

What is the lower bound on the information leakage?

It is related to the conditional entropy [2] $H(r_t|r_0) = nh(\varepsilon)$ where ε is the error rate and n is the response length.

$$h(\varepsilon) = -\varepsilon.log_2(\varepsilon) - (1-\varepsilon).log_2(1-\varepsilon)$$

The best length we can expect for the final response is then:

$$n - nh(\varepsilon) = n(1 - h(\varepsilon))$$

Examples:

With a 128-bit response and a 5% error rate: 91 bits.
With a 128-bit response and a 10% error rate: 67 bits.

[2] Jesus Martinez-Mateo et al. "Demystifying the Information Reconciliation Protocol CASCADE". . *Quantum Information & Computation* 15.5&6 (2015), pp. 453–477.

Parameters to tune to limit the leakage

How to set the CASCADE parameters?

- **Initial block size**: depends on the error rate.
- **Number of passes**: depends on the required correction success rate.
- **Block size multiplier**: x2/x4 at each pass.

How to set the CASCADE parameters?

- **Initial block size**: depends on the error rate.
- **Number of passes**: depends on the required correction success rate.
- **Block size multiplier**: x2/x4 at each pass.

Problem

The block size **cannot** exceed $n/2$.
The **failure rate** remains **too high**.

How to set the CASCADE parameters?

- **Initial block size**: depends on the error rate.
- **Number of passes**: depends on the required correction success rate.
- **Block size multiplier**: x2/x4 at each pass.

Problem

The block size **cannot** exceed $n/2$.
The **failure rate** remains **too high**.

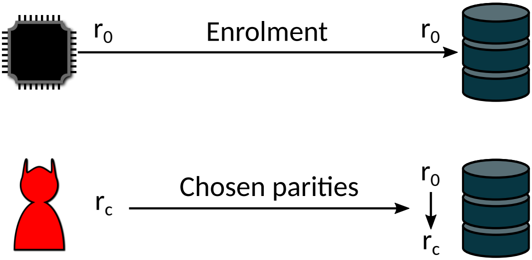
Solution

Add extra passes **without increasing** the block size.

Attacks and countermeasures

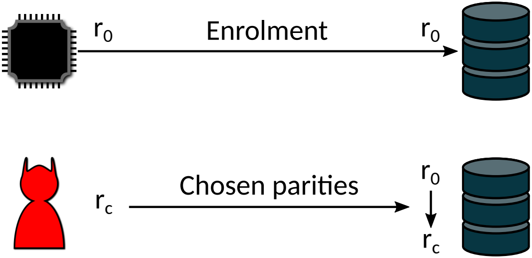
Threat: chosen parities scenario

An attacker wants to set a chosen response value on the server side by sending chosen parities.



Threat: chosen parities scenario

An attacker wants to set a chosen response value on the server side by sending chosen parities.

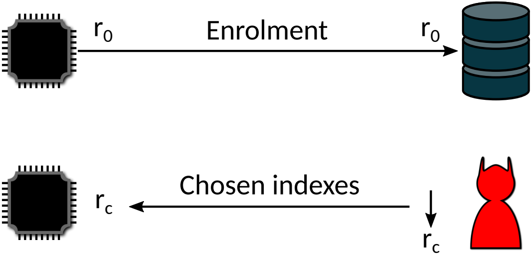


Countermeasure:

Limit the number of modifiable bits on the server side.

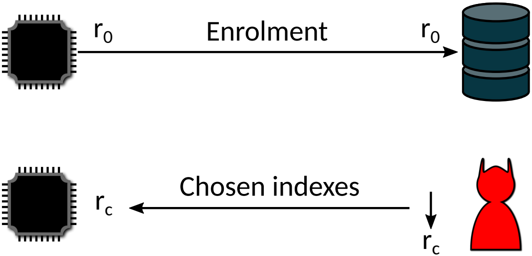
Threat: chosen indexes scenario

An attacker wants to **recover the PUF response** by building a sufficiently determined system of equations.



Threat: chosen indexes scenario

An attacker wants to **recover the PUF response** by building a sufficiently determined system of equations.



Countermeasures:

- Limit the number of parity values that can be sent out.
- Regenerate a new response at every protocol execution.

Experimental results

Several realistic PUF references:

- RO PUF in FPGA $\varepsilon = 0.9\%$ [3].
- TERO PUF in FPGA $\varepsilon = 1.8\%$ [4].
- SRAM PUF in ASIC $\varepsilon = 5.5\%$ [5].

Keep **128 bits secret** from a 256-bit response with **failure rate < 10^{-6}**.

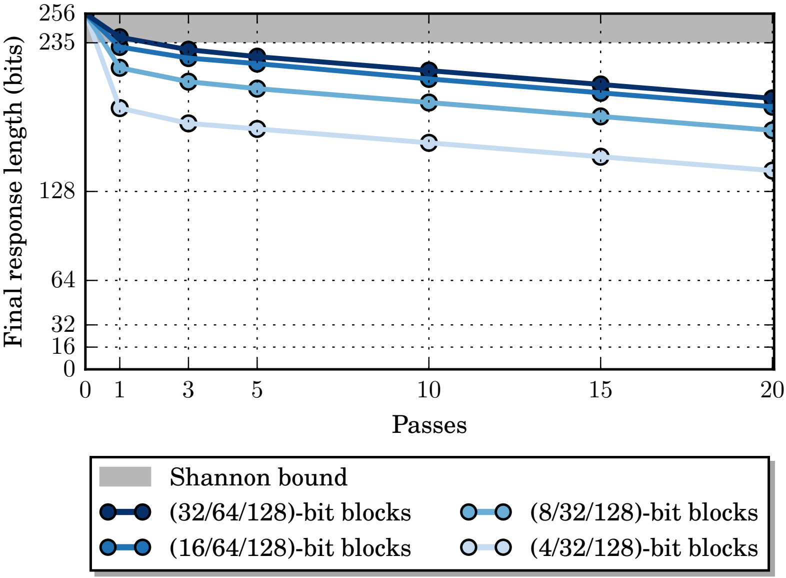
Simulation carried out on 2 500 000 responses.

[3] Abhranil Maiti, Jeff Casarona, Luke McHale, and Patrick Schaumont. “A large scale characterization of RO-PUF”. . *HOST*. 2010, pp. 94–99.

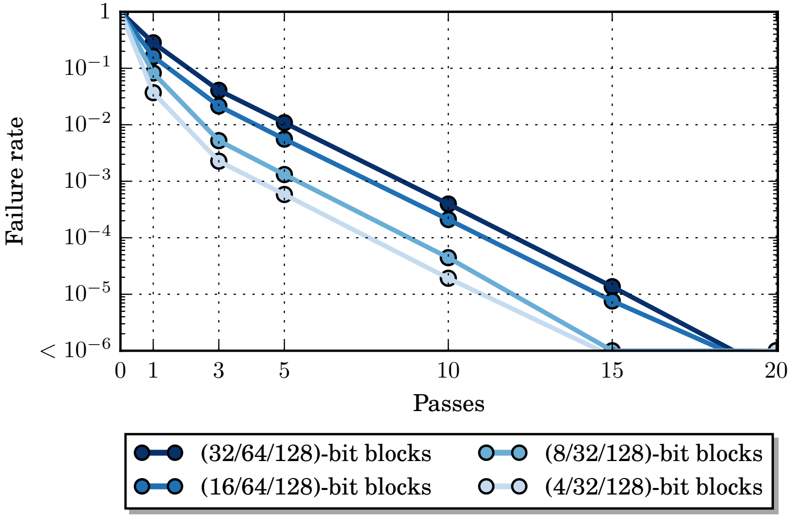
[4] Cédric Marchand, Lilian Bossuet, and Abdelkarim Cherkaoui. “Enhanced TERO-PUF Implementations and Characterization on FPGAs”. *International Symposium on FPGAs*. 2016, p. 282.

[5] Mathias Claes, Vincent van der Leest, and An Braeken. “Comparison of SRAM and FF-PUF in 65nm Technology”. *Nordic Conference on Secure IT Systems*. Vol. 7161. 2011, pp. 47–64.

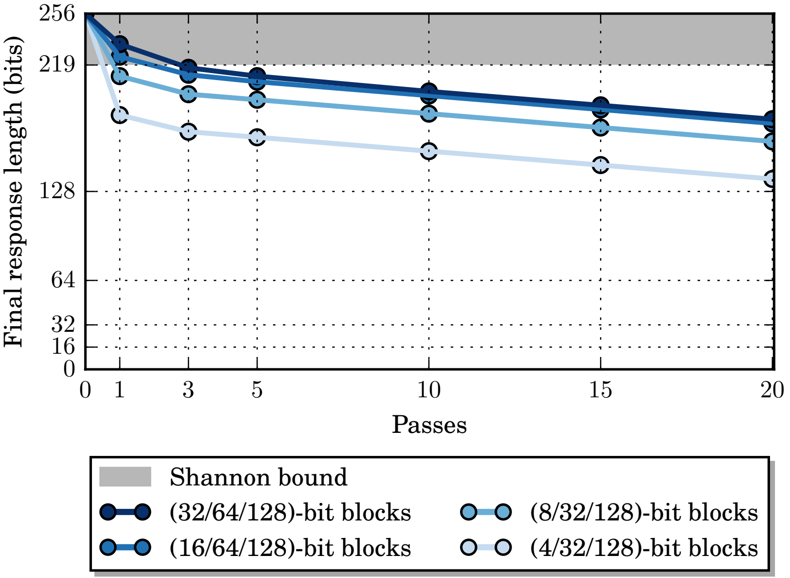
Leakage for $\varepsilon = 1\%$, (RO-PUF)



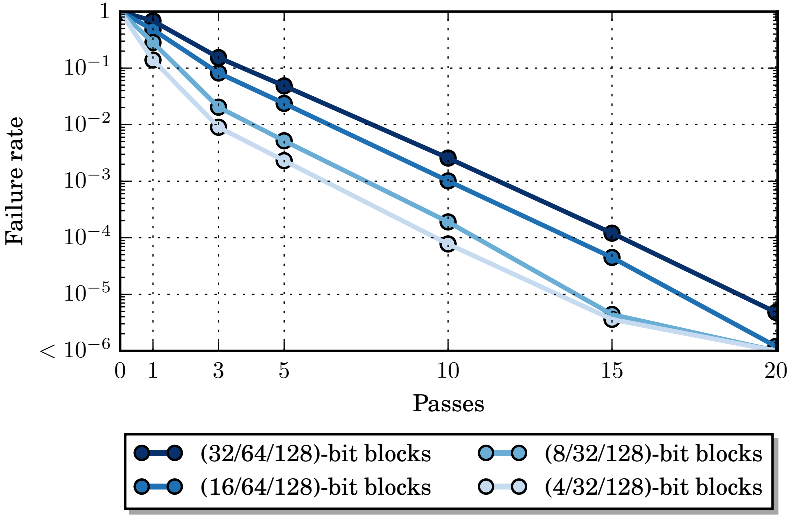
Failure rate for $\varepsilon = 1\%$, (RO-PUF)



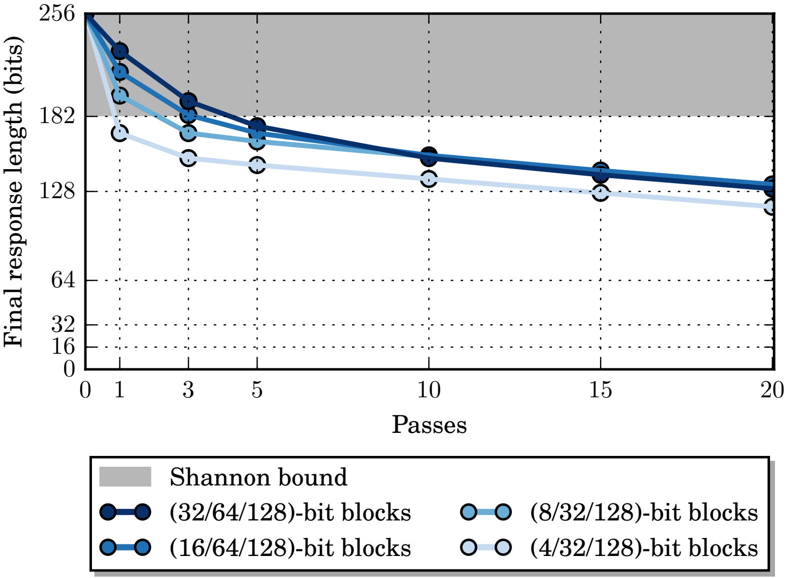
Leakage for $\varepsilon = 2\%$, (TERO-PUF)



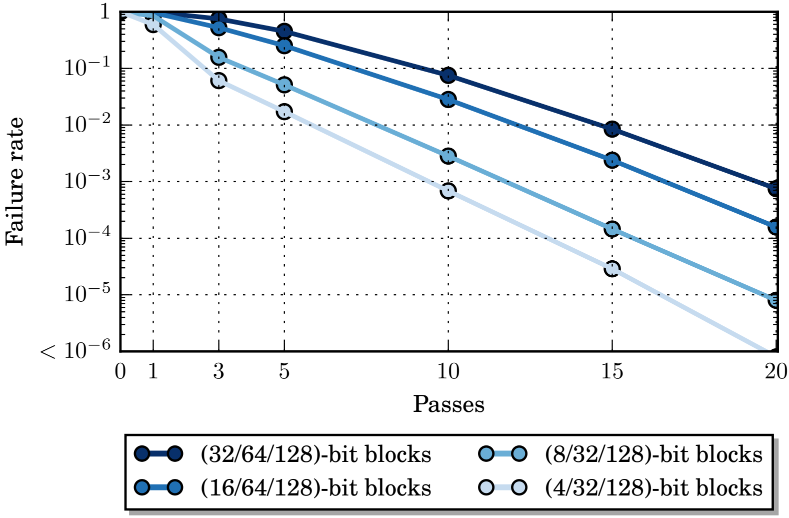
Failure rate for $\varepsilon = 2\%$, (TERO-PUF)



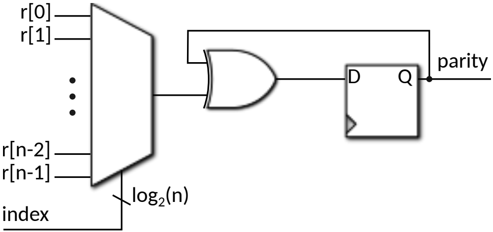
Leakage for $\varepsilon = 5\%$, (SRAM-PUF)



Failure rate for $\varepsilon = 5\%$, (SRAM-PUF)

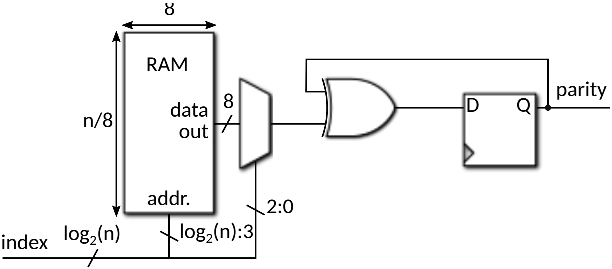


Hardware implementation



Logic resources:

- Spartan 3: 67 Slices
- Spartan 6: 19 Slices
- 0 RAM bits



Logic resources:

- Spartan 3: 3 Slices
- Spartan 6: 1 Slice
- 256 RAM bits

Classical error-correcting codes for PUFs

Article	Construction and code(s)		Logic resources (Slices) Spartan 3	Spartan 6	Block RAM bits
[6]	Reed-Muller (4, 7)			179	0
[7]	Reed-Muller (2, 6)		164		192
[8]	Concatenated: Repetition and Reed Muller		168		0
[9]	Differential Sequence Coding and Viterbi		75	27	10752
This work: CASCADE protocol		logic only	67	19	0
		with RAM	3	1	256

[6] Matthias Hiller et al. "Low-Area Reed Decoding in a Generalized Concatenated Code Construction for PUFs". *ISVLSI*. 2015, pp. 143–148

[7] Roel Maes, Pim Tuyls, and Ingrid Verbauwhede. "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs". *CHES*. 2009, pp. 332–347

[8] Christoph Bösch et al. "Efficient Helper Data Key Extractor on FPGAs". *CHES*. 2008, pp. 181–197

[9] Matthias Hiller, Meng-Day Yu, and Georg Sigl. "Cherry-Picking Reliable PUF Bits With Differential Sequence Coding". *IEEE Trans. Information Forensics and Security* 11.9 (2016), pp. 2065–2076

Conclusion

Compared to existing methods:

- ✔ **most lightweight** error-correction solution of state-of-the-art,
- ✔ can reach **very low** failure rates (down to 10^{-8}),
- ✔ leakage is **limited** and **easy** to estimate,
- ✔ **parameterizable** and can be changed **on the fly**.

All code available on Gitlab:
https://gitlab.univ-st-etienne.fr/b.colombier/cascade

Compared to existing methods:

- ✔ **most lightweight** error-correction solution of state-of-the-art,
- ✔ can reach **very low** failure rates (down to 10^{-8}),
- ✔ leakage is **limited** and **easy** to estimate,
- ✔ **parameterizable** and can be changed **on the fly**.

All code available on Gitlab:
https://gitlab.univ-st-etienne.fr/b.colombier/cascade

— Questions? —