Key reconciliation protocol application to error correction in silicon PUF responses

Brice Colombier*, Lilian Bossuet*, David Hély+

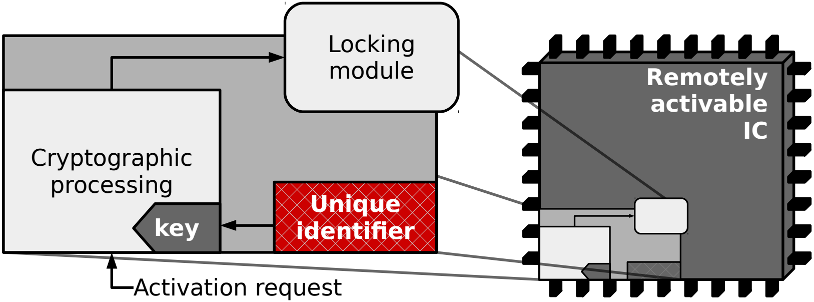
*Laboratoire Hubert Curien
Saint-Étienne — France
+LCIS, Grenoble Institute of Technology
Valence — France

May 31, 2016

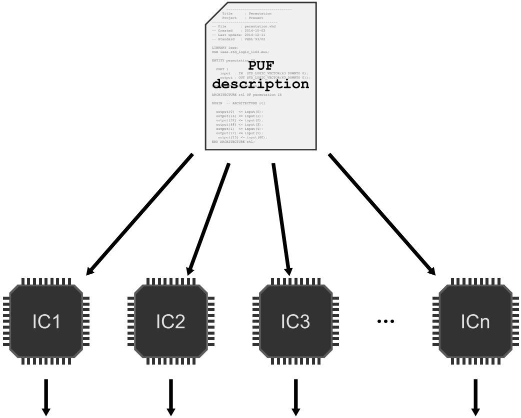
Journée Sécurité Numérique du GDR SoC-SiP : 11ème édition
La génération d'aléa dans le matériel : TRNG & PUF

LABORATOIRE HUBERT CURIEN UMR • CNRS • 5516 • SAINT-ETIENNE RhôneAlpes Région SALWARE French ANR Project

SALWARE project[1]



[1] http://www.univ-st-etienne.fr/salware/



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.

The instability problem

Problem:

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

This variation depends on multiple parameters:

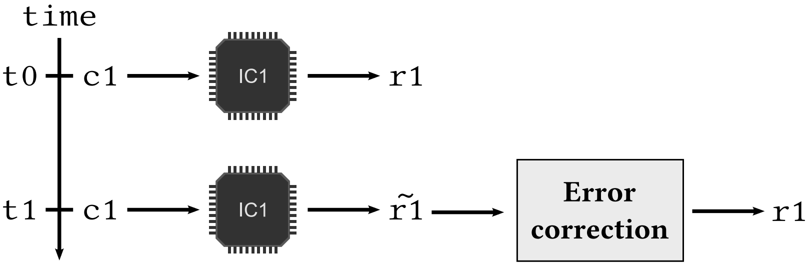
- PUF architecture,
- Process node,
- Aging,
- Temperature,
- Environment...

→ It prevents the PUF response from being used as a **key**.

Assumptions and requirements

Solution:

Correct the PUF response.



Requirements for the error correction module:

- Low area,
- High correction probability.

State-of-the-art error correction for PUF responses

Several error-correcting code implementations exist:

Article	Construction and code(s)	Logic resources (Xilinx Slices) Xilinx Spartan 3	Xilinx Spartan 6
[2]	Concatenated: Repetition and BCH		**221**
[3]	Reed-Muller		**179**
[4]	BCH		**>59**
[5]	Concatenated: Repetition and Reed-Muller	**168**	

[2] R. Maes et al. "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator". *CHES*. 2012.

[3] M. Hiller et al. "Low-Area Reed Decoding in a Generalized Concatenated Code Construction for PUFs". *ISVLSI*. 2015.

[4] A. V. Herrewege et al. "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs". *FC*. 2012.

[5] C. Bösch et al. "Efficient Helper Data Key Extractor on FPGAs". *CHES*. 2008.

Several error-correcting code implementations exist:

Article	Construction and code(s)	Logic resources (Xilinx Slices) Xilinx Spartan 3	Xilinx Spartan 6
[2]	Concatenated: Repetition and BCH		221
[3]	Reed-Muller		179
[4]	BCH		>59
[5]	Concatenated: Repetition and Reed-Muller	168	
This work	CASCADE protocol	69	19

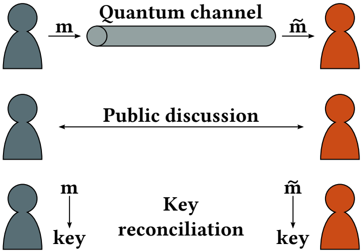
[2] R. Maes et al. "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator". *CHES*. 2012.

[3] M. Hiller et al. "Low-Area Reed Decoding in a Generalized Concatenated Code Construction for PUFs". *ISVLSI*. 2015.

[4] A. V. Herrewege et al. "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-Enabled RFIDs". *FC*. 2012.

[5] C. Bösch et al. "Efficient Helper Data Key Extractor on FPGAs". *CHES*. 2008.

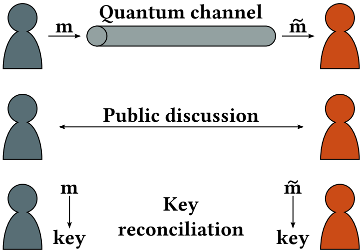
CASCADE introduced in 1993 by Brassard and Salvail[6]



The final key is **shorter** than the original message.

[6] G. Brassard et al. "Secret-Key Reconciliation by Public Discussion". *EUROCRYPT*. 1993.

CASCADE introduced in 1993 by Brassard and Salvail[6]



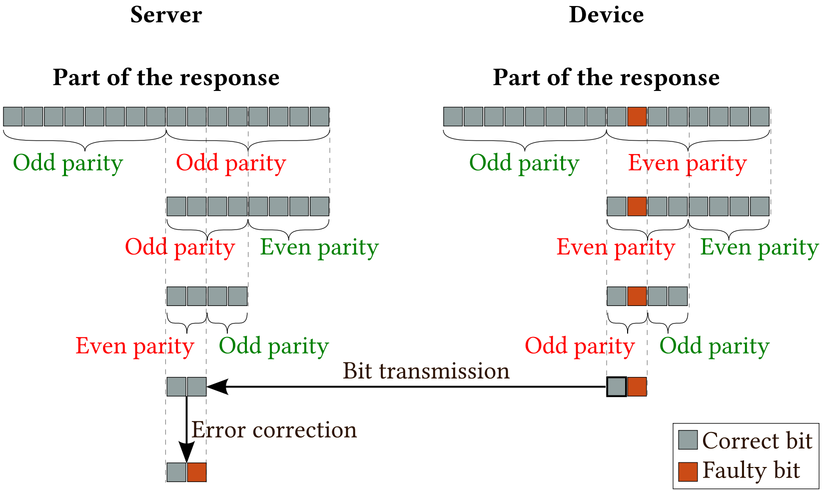
The final key is **shorter** than the original message.

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

[6] G. Brassard et al. "Secret-Key Reconciliation by Public Discussion". *EUROCRYPT*. 1993.

Dichotomous error detection and correction

Works on **parts** of the responses that have a **different parity**.



Allows to correct **one error**.

Algorithm 1: BINARY

Input: r_0, r_t, n_{passes}

for $i = 1$ **to** n_{passes} **do**

  Shuffle r_0 and r_t using a public permutation σ_i

  Split r_0 and r_t in blocks of size s_b

  forall *blocks* **do**

    Compute the relative parity $P_r(b_{0,i}, b_{t,i})$ // Detection

    if $P_r(b_{0,i}, b_{t,i}) = 1$ **then**

      CONFIRM($b_{0,i}, b_{t,i}$) // Correction

  Double the block size $s_b = 2 * s_b$

Un-shuffle r_0 and r_t with inverse permutations $\sigma_1^{-1}, \sigma_2^{-1}, ..., \sigma_{n_{passes}}^{-1}$

return r_0, r_t

Example: 32-bit responses, 5 errors.

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | **8** | 9 | 10 | 11 | 12 | 13 | 14 | **15** | 16 | **17** | 18 | **19** | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | **29** | 30 | 31 |

Shuffling and splitting into blocks

[31 1 7 9] [26 13 3 11] [30 18 5 14] [22 2 20 10] [**19** **17** 23 **8**] [24 27 21 0] [16 **29** 28 **15**] [25 12 4 6]

Error correction using CONFIRM

[31 1 7 9] [26 13 3 11] [30 18 5 14] [22 2 20 10] [**19** 17 23 **8**] [24 27 21 0] [16 **29** 28 **15**] [25 12 4 6]

Pass 1

Shuffling and splitting into blocks

[**8** 0 14 31 28 23 11 27] [5 16 21 **19** 9 1 **15** 20] [24 4 2 12 7 13 **29** 3] [30 6 26 18 25 10 22 17]

Error correction using CONFIRM

[8 0 14 31 28 23 11 27] [5 16 21 **19** 9 1 **15** 20] [24 4 2 12 7 13 29 3] [30 6 26 18 25 10 22 17]

Pass 2

Shuffling and splitting into blocks

[10 13 16 1 31 25 **15** 21 18 6 12 4 7 23 26 22] [5 29 30 24 17 20 9 27 **19** 2 28 11 0 8 3 14]

Error correction using CONFIRM

[10 13 16 1 31 25 15 21 18 6 12 4 7 23 26 22] [5 29 30 24 17 20 9 27 19 2 28 11 0 8 3 14]

Pass 3

Un-shuffling

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 | 30 | 31 |

Associated information leakage

Two ways of leaking information:

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

Associated information leakage

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**.

Associated information leakage

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**.

How can it be improved?

Backtracking

Backtracking can be used to leak fewer bits.

After a pass, all the blocks have an **even** relative parity.

Backtracking

Backtracking can be used to leak fewer bits.

After a pass, all the blocks have an **even** relative parity.
→ if an error is corrected on a bit from this block in a subsequent pass, then its relative parity becomes **odd** again.

Backtracking

Backtracking can be used to leak fewer bits.

After a pass, all the blocks have an **even** relative parity.
→ if an error is corrected on a bit from this block in a subsequent pass, then its relative parity becomes **odd** again.
→ **one more** error from this block can be corrected.

Backtracking can be used to leak fewer bits.

After a pass, all the blocks have an **even** relative parity.
→ if an error is corrected on a bit from this block in a subsequent pass, then its relative parity becomes **odd** again.
→ **one more** error from this block can be corrected.

Example:

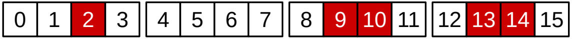
12	14	4	7	9	0	13	5

Parity check does not detect these errors.
If, **in a subsequent pass**, the error 9 is corrected:
→ The block can be **processed again** to correct error 13.

Backtracking

Required:
Two lists storing blocks according to their relative parity.

- Correcting an error at index i makes blocks containing index i move from one list to the other
  → (**their relative parity changed**).
- Error correction is carried out until there are **no more blocks of odd parity**.
- At the end of each pass, the blocks are added to the list of blocks of **even relative parity**.



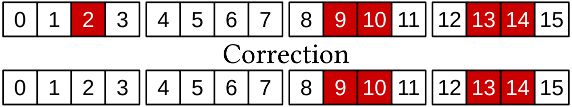
Blocks of even relative parity:

$\varnothing$

Blocks of odd relative parity:

$\varnothing$

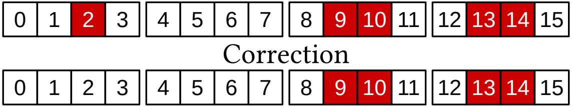
The CASCADE protocol



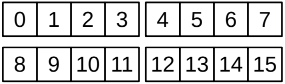
Blocks of even relative parity:
$\varnothing$

Blocks of odd relative parity:
$\varnothing$

The CASCADE protocol



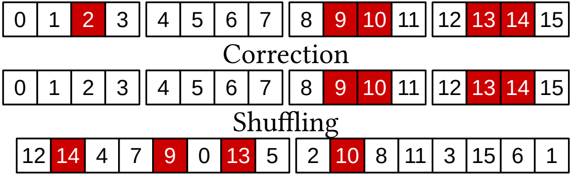
Blocks of even relative parity:



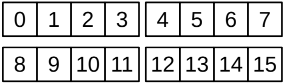
Blocks of odd relative parity:

$\varnothing$

The CASCADE protocol



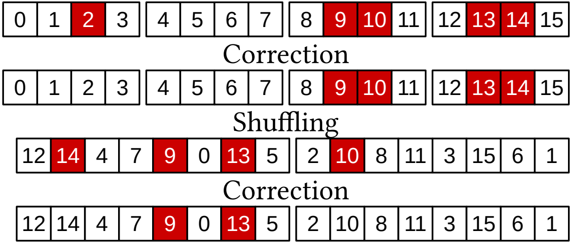
Blocks of even relative parity:



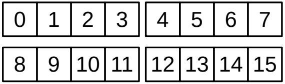
Blocks of odd relative parity:

$\varnothing$

The CASCADE protocol



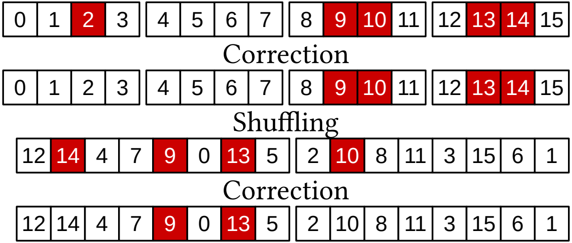
Blocks of even relative parity:



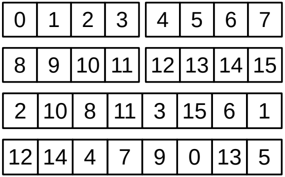
Blocks of odd relative parity:

$\varnothing$

The CASCADE protocol



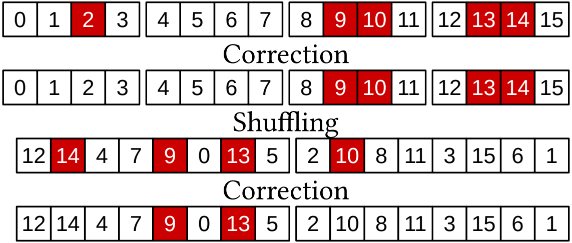
Blocks of even relative parity:



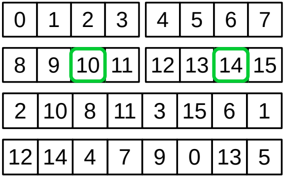
Blocks of odd relative parity:

$\varnothing$

The CASCADE protocol



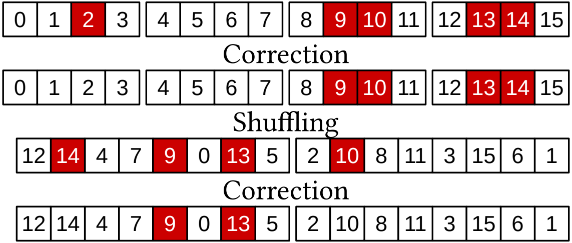
Blocks of even relative parity:



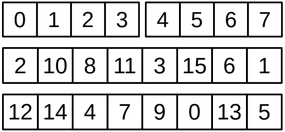
Blocks of odd relative parity:

$\varnothing$

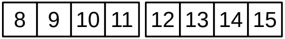
The CASCADE protocol



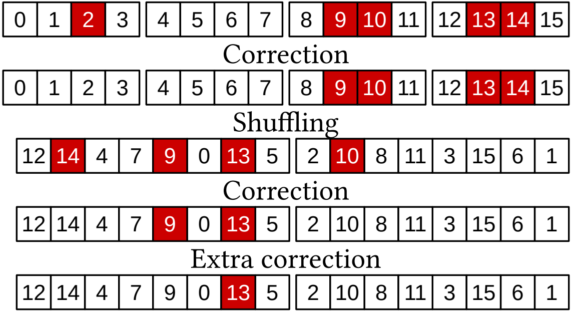
Blocks of even relative parity:



Blocks of odd relative parity:



The CASCADE protocol



Blocks of even relative parity:

0 1 2 3 | 4 5 6 7

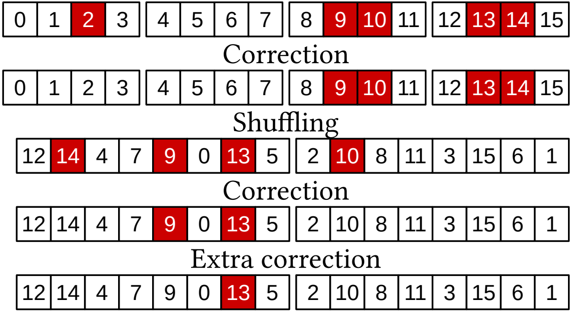
2 10 8 11 3 15 6 1

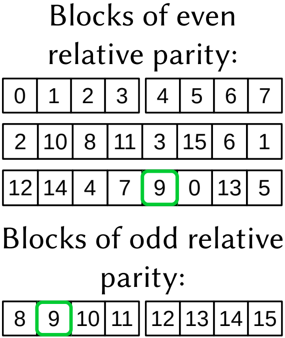
12 14 4 7 9 0 13 5

Blocks of odd relative parity:

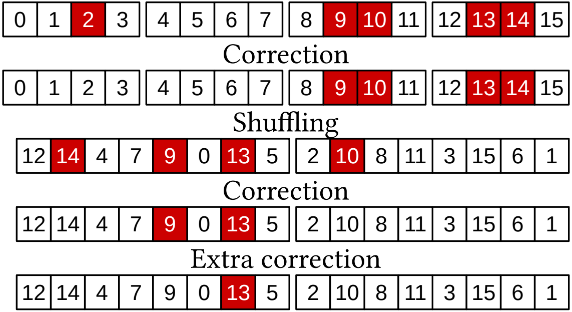
8 9 10 11 | 12 13 14 15

The CASCADE protocol

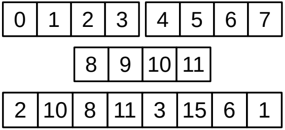




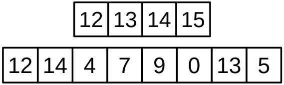
The CASCADE protocol



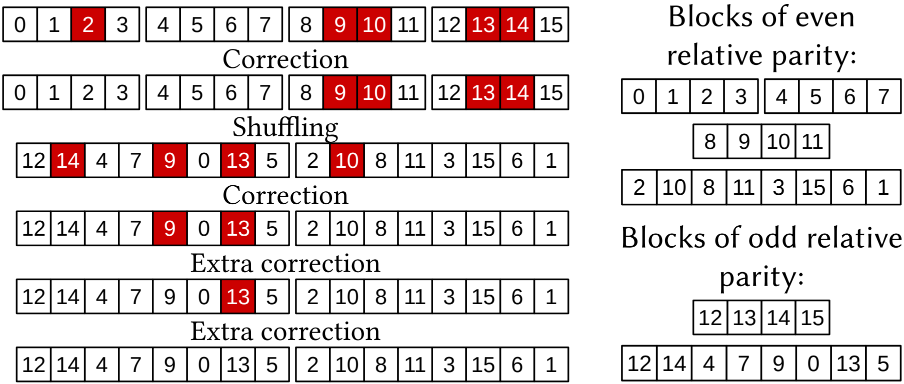
Blocks of even relative parity:



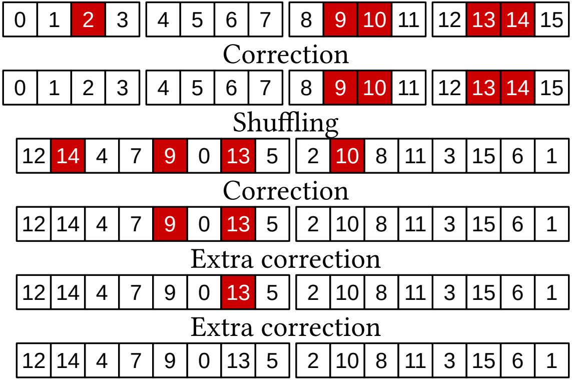
Blocks of odd relative parity:



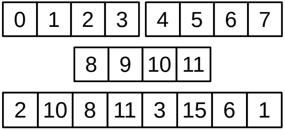
The CASCADE protocol



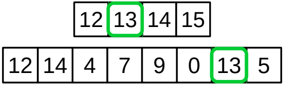
The CASCADE protocol



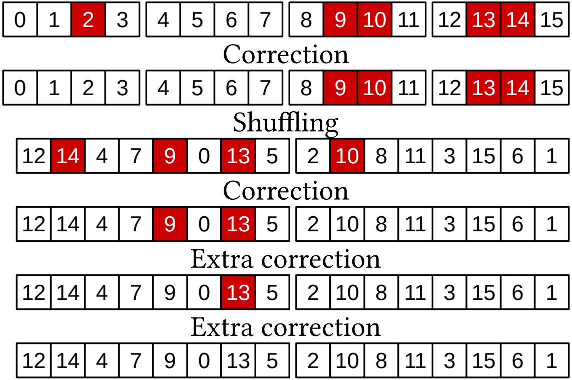
Blocks of even relative parity:



Blocks of odd relative parity:



The CASCADE protocol



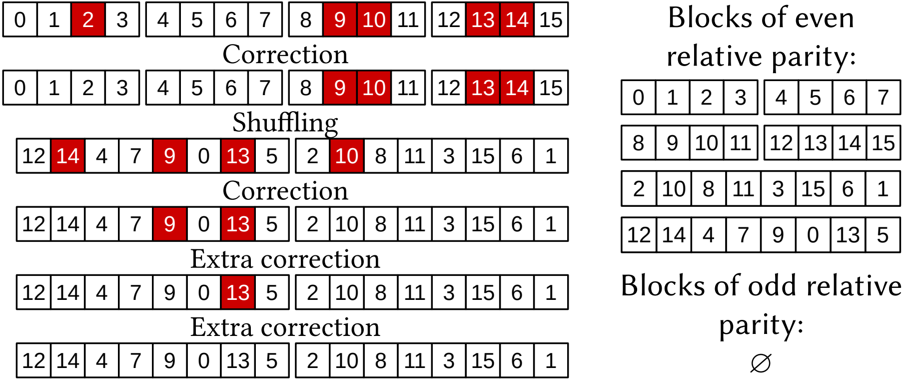
Blocks of even relative parity:

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

Blocks of odd relative parity:

$\varnothing$

The CASCADE protocol



For the **same number of passes**, the CASCADE protocol allows to correct **more errors** than BINARY.
→ The information leakage is **lower**.

Information leakage

What is the lower bound on the information leakage?

It is related to the conditional entropy[7] $H(r_t|r_0) = nh(\varepsilon)$ 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.

[7] J. Martinez-Mateo et al. "Demystifying the Information Reconciliation Protocol CASCADE". (2015).

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 at each pass.

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 at each pass.

⚠ ⚠ ⚠

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

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 at each pass.

⚠ ⚠ ⚠

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

Solution

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

Several realistic PUF references:

- RO PUF in FPGA $\varepsilon = 0.9\%$[8].
- TERO PUF in FPGA $\varepsilon = 1.8\%$[9].
- SRAM PUF in ASIC $\varepsilon = 5.5\%$[10].

256-bit responses, aim for 128-bit security

Simulation carried out on 2 500 000 responses.

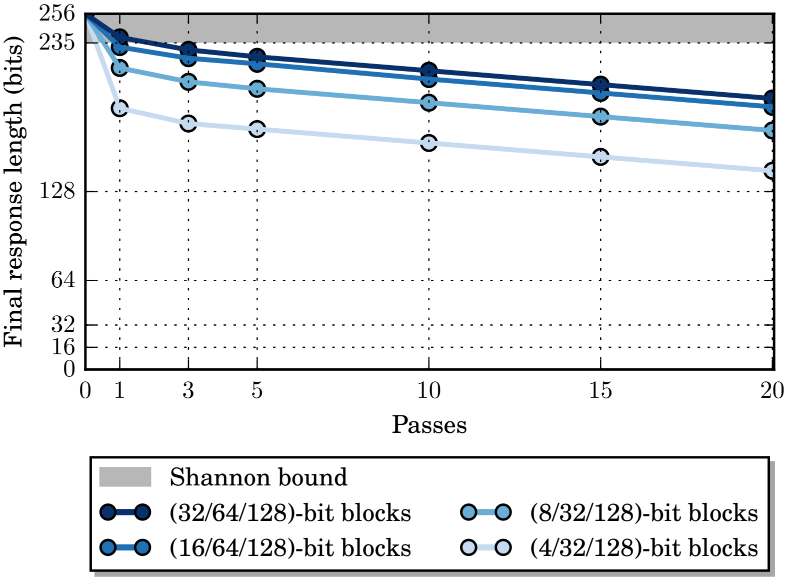
---

[8] A. Maiti et al. "A large scale characterization of RO-PUF". . *HOST*. 2010.

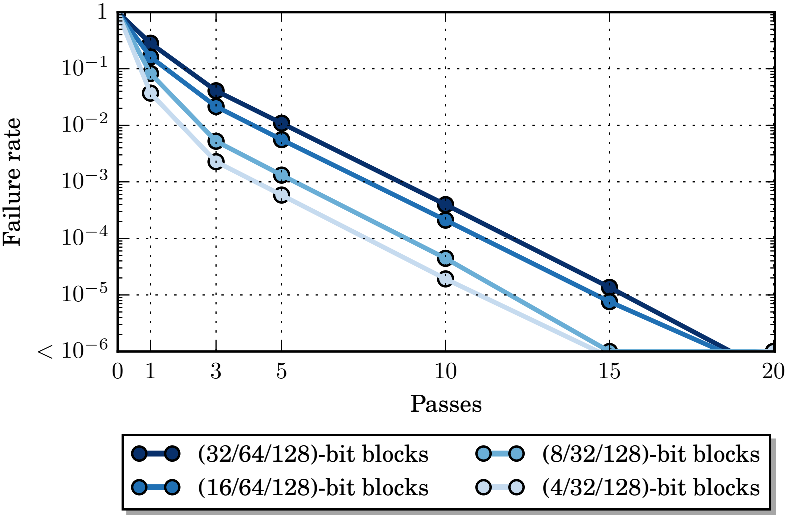
[9] C. Marchand et al. "Enhanced TERO-PUF Implementations and Characterization on FPGAs". *International Symposium on FPGAs*. ACM, 2016.

[10] M. Claes et al. "Comparison of SRAM and FF-PUF in 65nm Technology". *Nordic Conference on Secure IT Systems*. 2011.

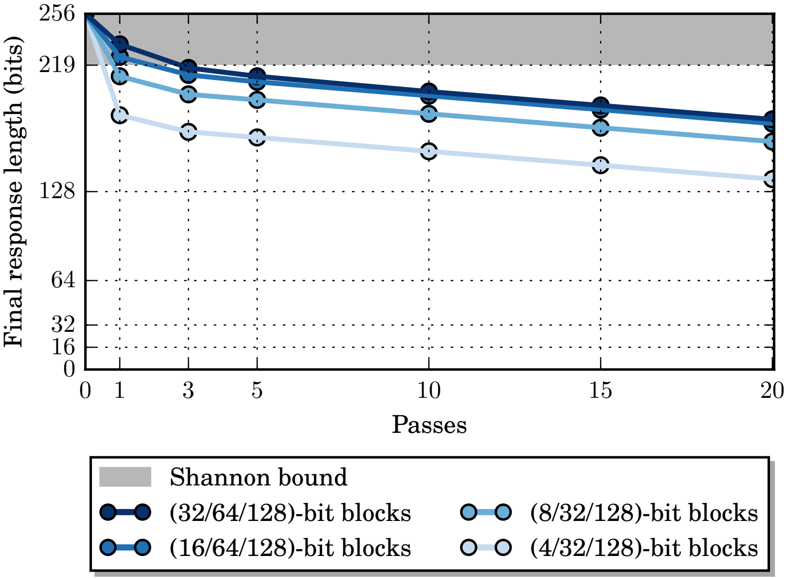
Leakage for $\varepsilon = 1\%$, (RO-PUF)



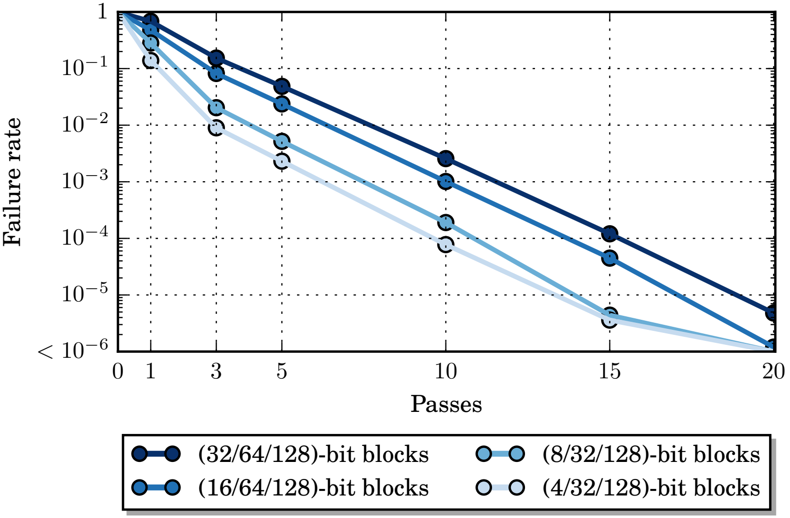
Failure rate for $\varepsilon = 1\%$, (RO-PUF)



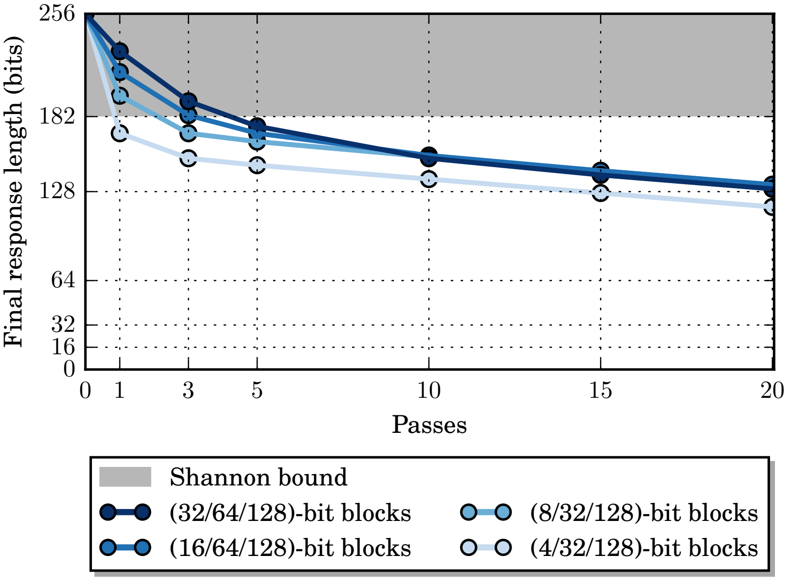
Leakage for $\varepsilon = 2\%$, (TERO-PUF)



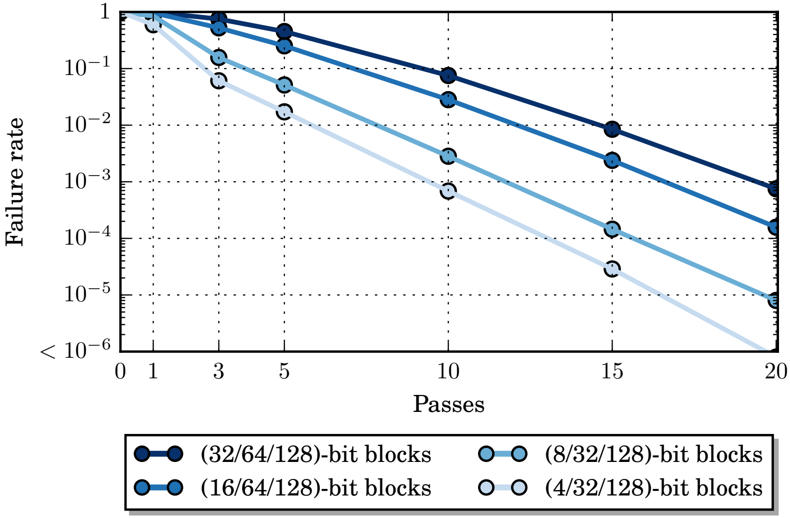
Failure rate for $\varepsilon = 2\%$, (TERO-PUF)



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

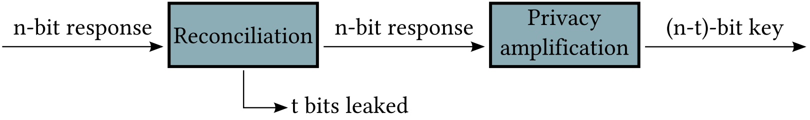


Failure rate for $\varepsilon = 5\%$, (SRAM-PUF)



Privacy amplification

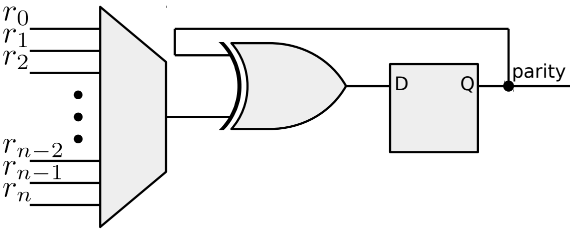
From an *n*-bit response, if *t* bits are leaked, it is possible to obtain an $(n-t)$-bit secret key.



A **hash function** can be used for privacy amplification[11].

[11] R. Impagliazzo, L.A. Levin and M. Luby, *Pseudo-random Generation from one-way functions*, **21st Annual Symposium on Theory of Computing**, 1989.

Only **parity computations** are embedded.
All other computations can be done **on the server**.



Requirements:

- Multiplexer to select the bits to XOR,
- One XOR gate,
- One D flip-flop.

Latency

- Compute the parity of an *n*-bit block: **n cycles.**
- Correct one error in an *n*-bit block: $\sum_{i=1}^{log_2(n)} \frac{n}{2^i} = \mathbf{n - 1}$ **cycles**.

Latency

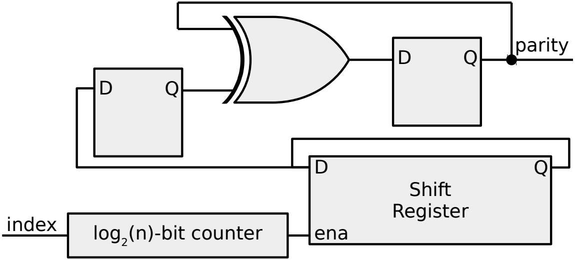
- Compute the parity of an n-bit block: **n cycles.**
- Correct one error in an n-bit block: $\sum_{i=1}^{log_2(n)} \frac{n}{2^i} = \mathbf{n-1}$ **cycles**.

Example:

256-bit response, $\varepsilon = 2\%$, 20 passes, $k_1 = 8$ bits:

- Best case (3%): 1 error, corrected AEAP.
  → Latency: **5 127** clock cycles.
- Worst case (0.05%): 14 errors, corrected ALAP.
  → Latency: **8 690** clock cycles.

Implementation 2nd option



Requirements:

- Circular shift register to select the bits to XOR,
- One counter,
- One XOR gate,
- Two D flip-flops.

Latency

- Compute the parity of an *n*-bit block: $\frac{n^2}{2}$ **cycles**.
- Correct one error in an *n*-bit block: $\frac{n(n-1)}{2}$ **cycles**.

Latency

- Compute the parity of an n-bit block: $\mathbf{\frac{n^2}{2}}$ **cycles**.
- Correct one error in an n-bit block: $\mathbf{\frac{n(n-1)}{2}}$ **cycles**.

Example:

256-bit response, $\varepsilon = 2\%$, 20 passes, $k_1 = 8$ bits:

- Best case (3%): 1 error, corrected AEAP.
  - → Latency: **656 256** clock cycles.
- Worst case (0.05%): 14 errors, corrected ALAP.
  - → Latency: **1 112 320** clock cycles.

Latency

- Compute the parity of an n-bit block: $\mathbf{\frac{n^2}{2}}$ **cycles**.
- Correct one error in an n-bit block: $\mathbf{\frac{n(n-1)}{2}}$ **cycles**.

Example:

256-bit response, $\varepsilon = 2\%$, 20 passes, $k_1 = 8$ bits:

- Best case (3%): 1 error, corrected AEAP.
  → Latency: **656 256** clock cycles.
- Worst case (0.05%): 14 errors, corrected ALAP.
  → Latency: **1 112 320** clock cycles.

Trade-off: area/latency.

IP core activation procedure:

	Server		Device i
at $t=0$	Generates challenge c_i		
		$\xrightarrow{c_i}$	
enrolment			$r_0 \leftarrow PUF(c_i)$
		$\xleftarrow{r_0}$	
	Stores r_0		
at $t=t_1$			Requests activation
		$\xrightarrow{c_i}$	
			$r_{t_1} \leftarrow PUF(c_i)$
activation	r_0	$\xleftrightarrow{CASCADE}$	r_{t_1}
	$K \leftarrow PA(r_{t_1})$	*Privacy amplification*	$K \leftarrow PA(r_{t_1})$
	Encrypts UW with K		
		$\xrightarrow{[UW]_K}$	
			Decrypts UW Activates by unlocking

Conclusion

Compared to existing methods:

→ few on-chip logic resources,

→ can reach very low failure rates,

→ very tunable depending on the expected error-rate

Conclusion

Compared to existing methods:

→ few on-chip logic resources,

→ can reach very low failure rates,

→ very tunable depending on the expected error-rate

DONE/TO-DO:

✓ Software model,

✓ Implementation in VHDL,

× Tests with a real PUF: TERO-PUF

× Integration in the overall module.

Conclusion

Compared to existing methods:

→ few on-chip logic resources,

→ can reach very low failure rates,

→ very tunable depending on the expected error-rate

DONE/TO-DO:

✓ Software model,

✓ Implementation in VHDL,

× Tests with a real PUF: TERO-PUF

× Integration in the overall module.

— Questions? —