Key Reconciliation Protocols for Error Correction of Silicon PUF Responses

Brice Colombier, Lilian Bossuet, Viktor Fischer, David Hely

Key Reconciliation Protocols for Error Correction of Silicon PUF Responses

Brice Colombier, Lilian Bossuet, Viktor Fischer
Univ Lyon, UJM-Saint-Etienne, CNRS
Laboratoire Hubert Curien UMR 5516
F-42023, Saint-Étienne, France
{b.colombier, lilian.bossuet, fischer}@univ-st-etienne.fr

David Hély
Univ. Grenoble Alpes, LCIS
F-26000, Valence - France
david.hely@lcis.grenoble-inp.fr

***Abstract*—Physical Unclonable Functions (PUFs) are promising primitives for the lightweight authentication of an integrated circuit (IC). Indeed, by extracting an identifier from random process variations, they allow each instance of a design to be uniquely identified. However, the extracted identifiers are not stable enough to be used as is, and hence need to be corrected first. This is currently achieved using error-correcting codes in secure sketches, that generate helper data through a one-time procedure. As an alternative, we propose key reconciliation protocols. This interactive method, originating from quantum key distribution, allows two entities to correct errors in their respective correlated keys by discussing over a public channel. We believe that this can also be used by a device and a remote server to agree on two different responses to the same challenge from the same PUF obtained at different times. This approach has the advantage of requiring very few logic resources on the device side. The information leakage caused by the key reconciliation process is limited and easily computable. Results of implementation on FPGA targets are presented, showing that it is the most lightweight error-correction module to date.**

I. Introduction

Physical Unclonable Functions (PUFs) have emerged in the last two decades as a root of trust and a way to provide identifiers for integrated circuits (ICs). They rapidly gained attention thanks to their lightweight and tamper-evident nature. Indeed, they usually require only a small area on the device and do not require a dedicated technology process, compared to non-volatile memory which could be used to store a unique identifier. Moreover, since they rely on *physical* characteristics to derive the identifier, most attempts to tamper with the PUF modifies the responses and makes the PUF useless. This justifies the term *unclonable*. These two characteristics made PUFs a convincing candidate for lightweight and secure IC authentication.

However, PUFs do have one major drawback: two responses obtained at different times from the same PUF using an identical challenge are different. This instability is caused by environmental parameters, aging of the device, PUF architecture, etc. For that reason, PUF responses are not reliable enough to be directly used as cryptographic keys and require error-correction.

The current way to address this issue is to implement error-correcting codes with the PUF [1]–[3]. When the PUF is first challenged, so-called *helper data* are generated from the response. Later on, if the PUF is challenged again with an identical challenge, these related helper data are exploited by the error correction module to regenerate the original response from the inaccurate one. Several types of error-correcting codes can be used to this end, but they all induce significant area overhead on the IC. This is contrasted with the lightweight nature of PUFs, and prevents widespread adoption by industry.

A. Contribution

In this paper, we propose to use a key reconciliation protocol instead. This interactive method, proposed in [4] and improved in [5], is called the `CASCADE` protocol. It is the main protocol for key reconciliation in a quantum key distribution context. It allows two parties who exchanged a stream of bits through an insecure and noisy quantum channel to discuss about it publicly and derive a secret key from it. We believe that this protocol can also be used to reconcile two PUF responses obtained from the same challenge but at a different time. The `CASCADE` protocol mainly consists in interactively exchanging parity values of different blocks of the responses. Therefore, only parity computations need to be carried out on-chip, which requires very few logic resources. This minimal area overhead comes at the cost of heavy communication between the IC and the server. However, in the context of intellectual property protection of ICs, device authentication occurs rarely in the IC lifetime. Moreover, existing error-correcting codes are also time consuming. The parity values are then exploited to modify the response bits on the server side, like the reverse fuzzy extractor [6]. When the protocol terminates, it is highly probable that the two parties will own an identical response. These two identical responses could then be further processed to generate a cryptographically strong secret key. The `CASCADE` protocol then specifically focuses on correcting errors only, and does not address further processing which is required to make the generated keys uniformly distributed for instance.

The `CASCADE` protocol has two main advantages over existing error-correcting codes used for PUFs integrated in ICs. First, since only parity computations need to be embedded on the circuit, the area overhead is very limited. We then propose two implementations that balance area overhead and execution time. As a second advantage, the `CASCADE` protocol is greatly parameterisable and can accommodate various error-rates and failure rates. Moreover, the parameters can be dynamically modified after the IC has been built. This makes it a very good candidate for integration alongside PUFs in order to protect ICs from counterfeiting.

B. Notations

PUF responses are r, of size n. The reference response, obtained during enrolment, is denoted by r_0. The response obtained later on, which contains errors with an error rate ε, is r_t. The bit found at index i of the response is denoted by $r[i]$. In the key reconciliation protocol, responses r_0 and r_t are split into blocks $B_{0,0}, B_{0,1}, ..., B_{0,\frac{n}{k_i}}$ and $B_{t,0}, B_{t,1}, ..., B_{t,\frac{n}{k_i}}$ of size k_i. Random permutations used in the protocol are denoted by σ_i.

C. Overview

The rest of this paper is organised as follows. Section II presents the motivation for this work. In particular, it focuses on how the CASCADE protocol can be used alongside a PUF to achieve intellectual property protection for ICs. Section III presents PUFs, error-correcting codes and key reconciliation protocols. Section IV explains how key reconciliation protocols can be adapted to correct errors in PUF responses. Section V gives the results of implementation on various FPGA targets. Finally, Section VI discusses other aspects such as secret key generation, implementation variations and security.

D. Reproducibility

We have made our implementation of the CASCADE protocol on the device and server sides, i.e. hardware and software, available online[1]

II. Motivation

A. Economic context

Following Moore's law, electronic systems are becoming increasingly complex. Such an exponential increase in complexity comes with an associated rise of manufacturing costs. Overseas foundries are now major players in the semiconductors market, and provide *fabless* designers with manufacturing facilities [7]. However, in order to have their designs manufactured, IC designers must disclose it completely to the foundry. Moreover, once the foundry owns the design for manufacturing, the designer has no control on his intellectual property anymore. In particular, the designer has no way of knowing how many instances of the design are actually built.

This situation has led to the rise of counterfeiting and illegal copying of ICs [8], [9], even though the vast majority of the actual incidents are never reported to legal authorities. According to the *Alliance for Gray Market and Counterfeit Abatement*, approximately 10% of the semiconductor products are counterfeited [10]. The associated losses are worth hundreds of billions of dollars. The target audience of this work are both IC designers and intellectual property (IP) core providers who wish to protect their designs against such threats.

There have been several propositions aiming at mitigating these threats [11]. Most of them have in common the following requirement: every instance of a design must be absolutely distinguishable from the others. Therefore, a unique, per-device identifier is necessary. In the case of IC activation, the identifier needs to be derived only once. Therefore, protocol reusability is not a requirement in this specific use case. As a hardware root of trust, a PUF is a great candidate for the generation of such identifiers.

[1]https://gitlab.univ-st-etienne.fr/b.colombier/cascade

B. Overall scheme

After it has been embedded within the IC, the unique device identifier is used to remotely identify the IC. Typically, IC remote identification requires two phases. The first one is the enrolment phase. It is usually carried out after manufacturing. The aim is to assign a challenge-response pair to every circuit, obtained from the PUF, so that it can be identified later on.

The second is the identification phase. Upon device request, the server sends the challenge of a known challenge-response pair to the device. The device generates the associated response by questioning the PUF again and sends it back to the server. The server then owns two responses to the same challenge from the same PUF. If the Hamming distance between those two responses is low enough, the circuit is identified. This protocol is known to have flaws [12], and is only used for illustration purpose here. Figure 1 depicts the protocol.

	Server		**Device** i
at $t = 0$	Generates challenge c_i		
		$\xrightarrow{c_i}$	
enrolment			$r_{i,0} \leftarrow \text{PUF}(c_i)$
		$\xleftarrow{r_{i,0}}$	
	Stores $(c_i, r_{i,0})$		
at $t = t_1$			Requests activation
		$\xrightarrow{c_i}$	
identification			$r_{i,t_1} \leftarrow \text{PUF}(c_i)$
		$\xleftarrow{r_{i,t_1}}$	
	Validates if $\text{HD}(r_{i,0}, r_{i,t_1}) < \varepsilon$		

Fig. 1: Basic protocol for IC remote identification using a PUF.

However, PUF responses are not perfectly stable and contain errors.Usually, error-correcting codes are associated to the PUF to correct those errors.

Key reconciliation protocols [5] can be used instead, in order to correct the differences between two responses obtained at different times. They are presented in details in the following section. Once the two response are *reconciled*, they are identical with a very high probability. Therefore, they can be used to encrypt an activation word (AW in Figure 2). Such activation word is not a cryptographic key, but controls a logic masking or logic locking module [13] embedded in the circuit. This module controllably disrupts the outputs of the circuit if a wrong activation word is sent on its inputs. Since both the IC and the server own an identical response, the IC can then internally decrypt the activation word. If the correct activation word is fed to the logic masking/locking module, the circuit operates correctly. This protocol is used only once in the IC lifetime, when an activation request is issued. The modified activation step of the basic protocol is depicted in Figure 2.

In order to implement the previously described protocol for intellectual property protection, an activation module must be

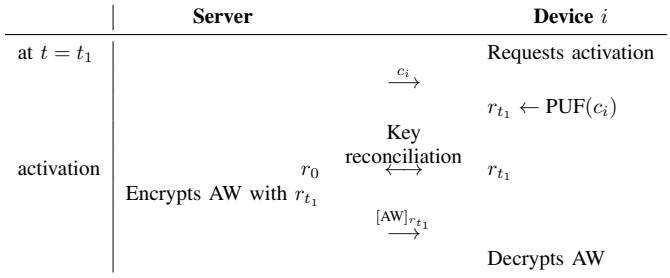

Fig. 2: Overview of a typical activation protocol for IC remote activation using a key reconciliation protocol.

added to the IC. As stated before, the activation module is used only once, in order to make the circuit usable if it is not an illegal copy. Therefore, the main evaluation criterion for this module is its area overhead. Indeed, such overhead is directly related to an increase in manufacturing costs for the designer. The activation module must then be as lightweight as possible, so that its cost does not exceed the losses due to counterfeiting.

It is composed of the following components:

- **a lightweight block cipher**: it allows to decrypt the cipher-text $[AW]_{r_{t_1}}$ using r_{t_1} as a key in order to obtain the activation word AW. A lightweight block cipher, such as PRESENT [14], can be used to this end.
- **a PUF**: it provides the unique identifier, which is then used as a key to encrypt the activation word.
- **a logic locking/masking module**: it makes the circuit unusable by disrupting the outputs if the wrong activation word is sent to its inputs.
- **a key reconciliation module**: it computes the parity values exploited in the key reconciliation protocol, and allows to obtain an identical response on the server and in the circuit.

Figure 3 depicts the activation module integrated in the IC.

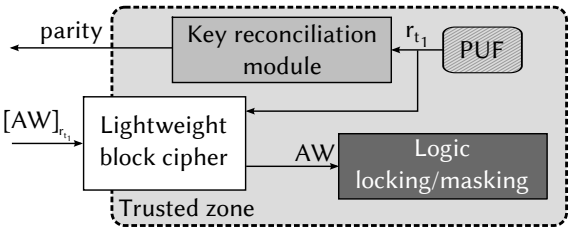

Fig. 3: Activation module added to the IC.

The required logic resources must then be as low as possible. This is the main advantage of key reconciliation protocols over existing error-correcting codes. This detailed in Section V. However, this is only possible by relaxing other design constraints. In particular, the communication overhead can be more important, since the protocol is meant to be executed only once in the circuit lifetime.

For the activation to be carried out properly, the availability of a server with high computational power is also assumed.

III. Related work

A. Physical Unclonable Functions

Physical Unclonable Functions (PUFs) are hardware primitives that are capable of extracting a binary string from random process variations. Strong PUFs can be challenged with an $m-$bit word called the challenge. The associated $n-$bit string obtained from the PUF is called the response. Those two form challenge-response pairs, which can be used as identifiers for ICs. Indeed, since the response results from random process variations, every IC embedding a PUF will generate a different response to the same challenge.

For the use case we consider, we assume the responses have full entropy. Making the original response have full entropy is outside the scope of this work, which targets error-correction.

B. Error-correcting codes for PUFs

An overview of helper data algorithms for PUF-based key generation has recently been published [15]. Helper data algorithms have several purposes, such as correcting the errors in the PUF response, making the response bits independent or ensuring they are uniformly distributed. Following their classification in [15], we consider only the so-called *Reproducibility* requirement. Related to the error correction step, this requirement guarantees that the corrected response has a very high probability of being identical to the reference response.

State-of-the-art error-correcting codes adapted to PUF responses employ the code-offset or the syndrome construction. Different types of codes are used, including repetition, BCH, Reed-Muller or Golay codes. A comparison is shown in Table I, in which logic resources, failure rate, acceptable error-rate and number of PUF bits required to reach 128-bit entropy are compared. The logic resources overhead should be as low as possible. For the failure rate, the typical value found in most articles is 10^{-6}. The acceptable error-rate depends on the PUF type which is used and the environmental conditions in which the PUF is used. Finally, the number of PUF bits required to reach 128-bit entropy is also an important criterion. Indeed, the more bits required from the PUF the larger the PUF implementation. For lightweight applications, requiring less bits from the PUF is then an advantage. As shown in Table I, the overhead of classic error-correcting codes exceeds hundred of slices when implemented on Xilinx Spartan FPGAs, with 4-input LUTs on Xilinx Spartan 3 and 6-input LUTs on Xilinx Spartan 6. The numbers we report here correspond to the error-correction core only. It does not include the controllers or the PUF response storage. The last row in Table I gives the logic resources required by the key reconciliation protocol presented in this work. Compared to the smallest error-correcting codes [6], [16], [17], it requires six to ten times less logic resources. Another method based on differential sequence coding was proposed in [18]. Even though it is also very lightweight, it requires an additional convolutional code to reach low failure rates of 10^{-6}. In [18], the Viterbi decoder is implemented in Block RAM, which does not increase the Slices count although it actually takes logic resources on the device. The implementation part of our work is detailed in Sect. V.

TABLE I: Logic resources required to implement different codes with different constructions. The failure rate, acceptable error-rate and number of PUF bits required to obtain 128-bit entropy are also given. For CASCADE protocol implementation, we provide results for two implementations, one using only logic (LUTs and D flip-flops) and one using RAM blocks.

Article	Construction and code(s)		Logic resources (Slices) Spartan 3	Spartan 6	Block RAM Bits	Failure rate	Acceptable error-rate	PUF bits required for 128-bit entropy
[1]	Concatenated: Repetition (7, 1, 3) and BCH (318, 174, 17)			221	0	10^{-9}	13%	2226
[2]	Complementary IBS with Reed-Muller (2, 6)		250		0	10^{-6}	15%	>1536 (12×128)
[3]	Reed-Muller (4, 7)			179	0	1.48×10^{-9}	14%	130
[6]	BCH (255, 21, 55)			>59	0	$10^{-6.97}$	21.6%	1785
[16]	Reed-Muller (2, 6)		164		192	10^{-6}	15%	1536 (12×128)
[17]	Concatenated: Repetition (5, 1, 5) and Reed-Muller (1, 6)		168 (41+ 127)		0	1.49×10^{-6}	15%	4640
[17]	Concatenated Repetition (11, 1, 11) Golay G_{24}(24, 13, 7)		570 (41+ 539)		0	5.41×10^{-7}	15%	3696
[18]	Differential Sequence Coding		75	27	0	10^{-3}	15%	974
	DSC + Viterbi decoder		75	27	10752	10^{-6}	15%	974
This work	CASCADE protocol	logic only	**18**	**11**	**0**	10^{-6}	10%	512
		with RAM	**3**	**1**	**512**			
		logic only	**18**	**10**	**0**	10^{-6}	15%	1024
		with RAM	**3**	**1**	**1024**			

An interesting approach called *reverse fuzzy extractor* is presented in [6], which is the most fair comparison to our work. It is called *reverse* because instead of correcting the errors on the device side, the reference response stored on the server side is modified. This makes it possible to transfer the computationally expensive workload of error correction from the device to the server. When requested, the PUF generates helper data from a noisy response. This helper data is then sent to the server, which uses it to modify the reference response and get both responses to match. We suggest using this *reverse* principle along with key reconciliation protocols.

A key reconciliation approach was taken very recently in [19]. However, the non-interactive low leakage coding they proposed has a high area overhead too.

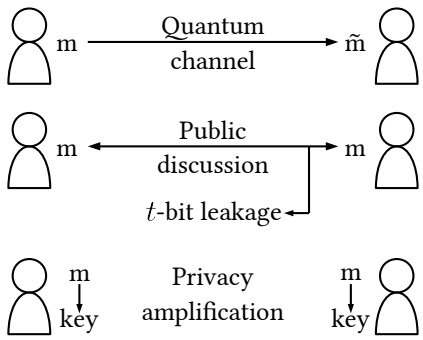

Fig. 4: Key reconciliation protocol.

C. Key reconciliation protocols

Key reconciliation protocols have been developed in the context of quantum key exchange [4], [5]. They allow two parties who exchanged a message over a quantum channel to discuss it publicly, locate the errors, and correct them. The errors can originate from noise in the channel or eavesdropping, which are usual characteristics of quantum channels. Obviously, the public discussion comes with associated leakage, which should be kept as small as possible so that most of the message is kept secret. Depending on the number of bits leaked, an appropriate privacy amplification method is used later to extract a secret key with the appropriate amount of entropy per bit. The overall protocol is depicted in Figure 4.

The public discussion step can be implemented as the BINARY protocol, described in [4] and shown in Algorithm 2. First, the original message is scrambled to spread the errors over the whole message in case they occur in bursts, which is often the case in quantum key distribution. The permutation used here is public, and so are the subsequent ones. The message is then split into blocks of size k_1. The initial block size k_1 is derived from the expected error rate in the quantum channel, so there is approximately one error per block.

Then the parity is computed for all the blocks, and exchanged over a public channel. The relative parity, i.e. the exclusive-OR of the parities of the blocks from each of the two responses is then computed too (see Equation (1) in which B_1 and B_2 are

the blocks containing bits from identical indexes from the two responses) .

$$P_r(B_1, B_2) = \underbrace{\left(\bigoplus_{i=1}^{|B_1|} B_1\right)}_{\text{Parity of } B_1} \oplus \underbrace{\left(\bigoplus_{i=1}^{|B_2|} B_2\right)}_{\text{Parity of } B_2} \quad (1)$$

The relative parity is used to detect the errors. If it is odd, that means that there is an odd number of errors. Thus there is at least one error to correct. At this point, an error-correction step is carried out, called CONFIRM. It proceeds with a binary search in order to isolate the errors. This is shown in Algorithm 1. This method enables the detection of an odd number of errors and the correction of one error per execution. The block is split into two parts of equal size, and the parity of the first half is exchanged. If the relative parity is odd, then the error is located in the first half. If the relative parity is even, then the error is located in the second half. The message is then split into two parts again and the process is repeated until the parts are only two bits long. By convention, the first bit is then sent. Knowing the parity of the two-bit block, the error has been located and corrected.

Algorithm 1: CONFIRM

Input: B_0, B_t
while *size*(B_0) > 1 **do**
  Split B_0 into two parts $B_{0,0}$ and $B_{0,1}$
  Split B_t into two parts $B_{t,0}$ and $B_{t,1}$
  if $P_r(B_{0,0}, B_{t,0}) = 1$ **then**
    // The error is located
    // in the first half
    $B_0 = B_{0,0}$
  else
    // The error is located
    // in the second half
    $B_0 = B_{0,1}$
return B_0

Algorithm 2: BINARY

Input: r_0, r_t, ε, n_{passes}
Scramble r_0 and r_t using a public permutation σ_0
Estimate the initial block size k_1 from the error rate ε
for *i = 1* **to** n_{passes} **do**
  Split r_0 and r_t into blocks of size k_i
  forall *blocks* **do**
    Compute the relative parity $P_r(B_{0,i}, B_{t,i})$
    if $P_r(B_{0,i}, B_{t,i}) = 1$ **then**
      CONFIRM$(B_{0,i}, B_{t,i})$
  Double the block size $k_{i+1} = 2 \times k_i$
  Scramble r_0 and r_t using a public random permutation σ_i
Unscramble r_0 and r_t with σ_0^{-1}, σ_1^{-1}, ..., $\sigma_{n_{passes}}^{-1}$
return r_0, r_t

After the CONFIRM method has been executed on all the blocks for which the relative parity is odd, the BINARY protocol is resumed. The block size is then doubled. The message is scrambled again using a public random permutation. The process starts again for the subsequent pass by splitting the message into blocks. After a sufficient number of passes, the probability that an error is still present in the message should be sufficiently low. A toy example of using the BINARY with 16-bit responses is shown in Figure 5.

CASCADE improved on BINARY by adding a backtracking step to the protocol. At the end of each pass, all the blocks have an even relative parity, since all detected errors have been corrected. Then, in the subsequent passes, if an error is corrected as index i, that means that all blocks from previous passes that contain index i are now of odd relative parity. Therefore, CONFIRM can be applied to them again.

Algorithm 3: CASCADE

Input: r_0, r_t, ε, n_{passes}
Scramble r_0 and r_t using a public permutation σ_0
Estimate the initial block size k_1 from the error rate ε
Create a list of blocks of even relative parity: L_{even}
Create a list of blocks of odd relative parity: L_{odd}
for *i = 1* **to** n_{passes} **do**
  Split r_0 and r_t into blocks of size k_i
  forall *blocks* **do**
    Compute the relative parity $P_r(B_{0,i}, B_{t,i})$
    if $P_r(B_{0,i}, B_{t,i}) = 1$ **then**
      CONFIRM$(B_{0,i}, B_{t,i})$: correct an error at index j
      Move all blocks containing j from L_{even} to L_{odd} or from L_{odd} to L_{even}
  Add all blocks to L_{even}
  while L_{odd} is not empty **do**
    // Backtracking step
    Find the smallest block B from L_{odd}
    CONFIRM(B_0, B_t): correct an error at index j
    Move all blocks containing j from L_{even} to L_{odd} or from L_{odd} to L_{even}
  Double the block size $k_{i+1} = 2 \times k_i$
  Scramble r_0 and r_t using a public random permutation σ_i
Unscramble r_0 and r_t with σ_0^{-1}, σ_1^{-1}, ..., $\sigma_{n_{passes}}^{-1}$
return r_0, r_t

First, two lists storing the blocks of even and odd relative parity are required. The backtracking step starts with the smallest block of odd relative parity, in which one error is corrected. All the blocks from the even and odd relative parity lists that contained this error are moved from one list to the other. This process is carried out until the list of blocks of odd relative parity is empty, which means all detected errors have been corrected. Finally, this allows more errors to be corrected in the same number of passes than by using the BINARY method alone.

We believe that the framework in which key reconciliation protocols are currently used, i.e. quantum key exchange, shares considerable similarity with the use case of PUFs requiring error correction presented in Sect. I. Our arguments are detailed in the following section.

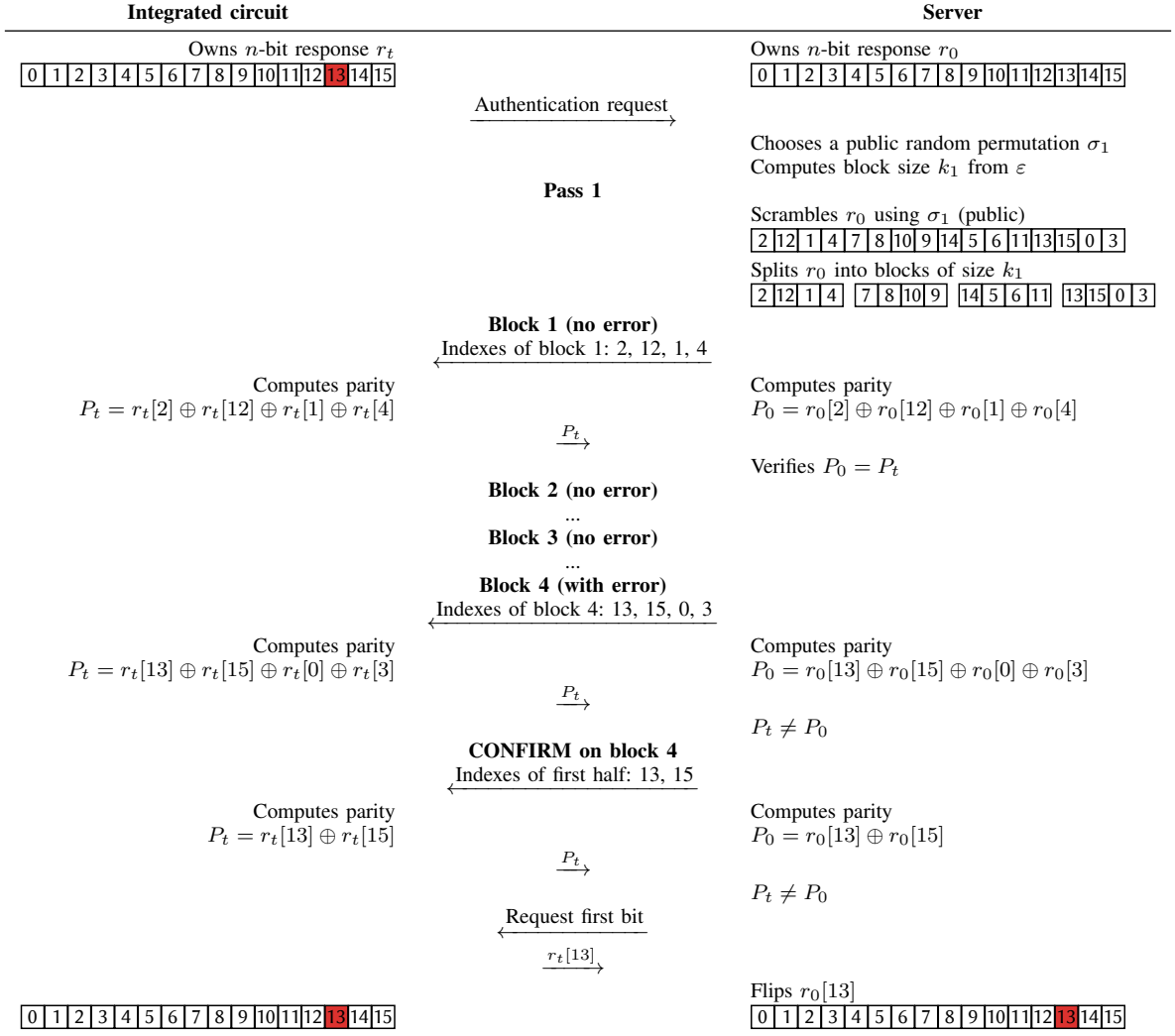

Fig. 5: Toy-example of executing the BINARY protocol on 16-bit responses with one error.

IV. Key reconciliation for error correction in PUF responses

Over time, the PUF responses to the same challenge differ as if they were altered by transmission over a binary symmetric channel (BSC) under the assumption that all the response bits have the same flipping probability. Therefore, using public discussion, the errors in the PUF response could be corrected. Following the *reverse* principle of [6], the response is modified on the server side according to the parity values received from the device.

A. Protocol parameters

There are two parameters to tune for the CASCADE protocol: the initial block size and the number of passes.

1) Initial block size: The initial block size k_1 is the block size used in the first pass, which is then doubled for every subsequent pass. It should be set so that there is one error per block on average after scrambling. This will make the error detectable by the parity check. Thus the initial block size is derived from the error rate ε. In the original article [5], the initial block size is: $k_1 \approx 0.73/\varepsilon$. However, optimised versions of CASCADE presented in [20] tend to increase the initial block size up to $1/\varepsilon$. Moreover, [20] states that the block size should be a power of two to achieve to the best reconciliation efficiency. This is emphasised in [21], in which the initial block size is given in Equation (2).

$$k_1 = min(2^{\lceil log_2(\frac{1}{p})\rceil}, \frac{n}{2}) \tag{2}$$

However, this initial block size makes it possible to correct enough errors only for very long bit frames, which is typically the case in quantum key distribution. For PUF responses, however, using k_1 from Equation (2) does not allow enough errors to be corrected. Therefore, in the following subsections, we explore different values for k_1, from 4 to 32 bits.

2) Number of passes: The number of passes depends on the acceptable number of responses left uncorrected after the protocol has been executed. By increasing the number of passes, more errors can be detected and corrected. However, each pass

implies parity exchanges, so increasing the number of passes also increases the leakage. Yet the final passes leak less since the blocks on which the parity is computed are larger. On the other hand, the block size is limited by the size of the response and cannot exceed half the response length: $\forall i, k_i \leq n/2$. This limitation is already present for frames of 2^{14} bits found in quantum key distribution, but is much more problematic when dealing with PUF responses, that are much shorter. For instance, if $n = 256$, then the passes must stop when the block size reaches $k_i = 128$ bits.

However, one approach proposed in [20], [21] is to add extra passes with block size $n/2$. This makes it possible to overcome the limitation previously mentioned. It then increases the success rate without leaking too much additional information. Indeed, each extra pass requires only two parity checks. Therefore, for each extra pass, only two bits of the response are leaked. Up to twenty passes are performed in the following example. The corresponding block sizes are shown in Table II.

TABLE II: Block sizes used here for passes 1 to 20.

k_1	k_2	k_3	...	k_{20}
4	32	128	...	128
8	32	128	...	128
16	64	128	...	128
32	64	128	...	128
64	128	128	...	128
128	128	128	...	128

3) Examples:

a) 2% error-rate: Figure 6 shows how the number of passes influence leakage. The values were obtained by simulation on 2,500,000 random responses. We chose to overestimate the leakage here by considering that one bit is leaked for every parity bit that is transmitted. When considering all the bits, the errors were assumed to be independent and identically distributed, although this might not be the case for practical PUF responses. Different distributions are discussed in Sect. VI-D.

As mentioned above, the number of passes is limited by the block size, which cannot exceed $n/2$. The Shannon bound, i.e. the maximum number of secret bits that can be achieved with optimal error correction is in grey. As can be seen in Figure 6, the best strategy to remain close to the Shannon bound appears to start with large blocks. For instance here, with $n = 256$ and $\varepsilon = 0.02$, starting with 32-bit blocks makes it possible to stay close to the Shannon bound.

A second metric that has to be taken into account is the failure rate. It is defined as the ratio of responses that could not be corrected after executing the protocol. Since increasing the number of passes enables more errors to be corrected, it also reduces the failure rate. Similarly, using smaller initial blocks makes it possible to detect more errors, which can then be corrected, thereby reducing the failure rate. Reusing previous parameters, Figure 7 shows how the failure rate is influenced by the number of passes and the initial block size.

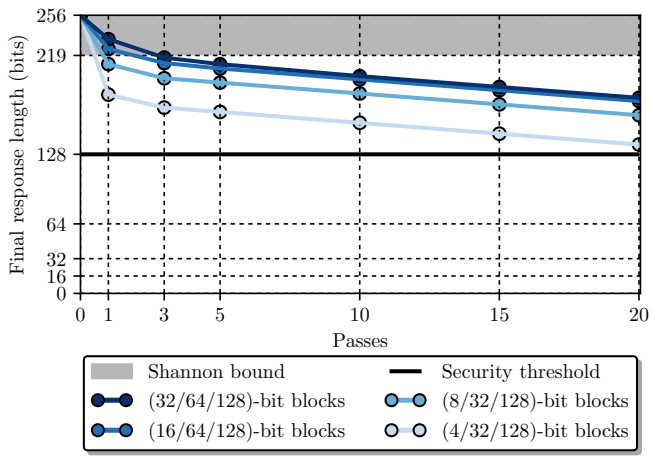

Fig. 6: Final response length after executing the CASCADE protocol on a 256-bit response with different numbers of passes and initial block sizes. Here, the error rate is 2%.

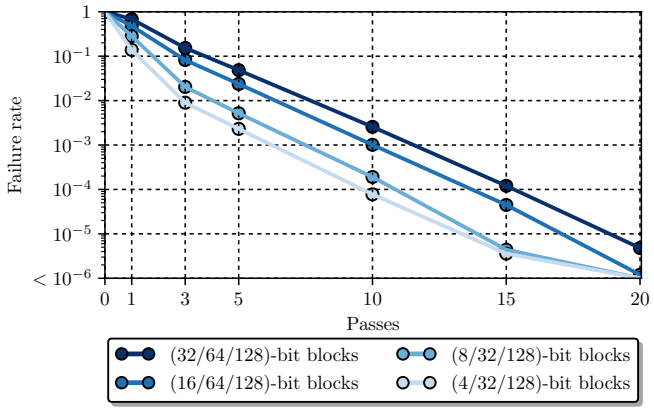

Fig. 7: Failure rate for a 256-bit response with different numbers of passes and initial block sizes. Here, the error rate is 2%.

b) 15% error-rate: For comparison we provide the same graphs but this time we consider a PUF with a 15% error-rate. 1024-bit responses are extracted, so the number of errors is 154 on average. The associated leakage is shown in Figure 8. Since the error-rate is higher, the Shannon bound is lower. Therefore, more bits are extracted from the PUF to be able to obtain a reliable and secret 128-bit response.

Compared to Figure 6, the leakage pattern is different here. Starting with small blocks of 4 bits, the leakage is very high in the first ten passes but then extra passes leak much less. After 40 passes, leakage is above the 128-bit threshold. Conversely, when large initial blocks are chosen, the leakage is less important in the first passes but stays high in the next ones. This is due to the fact that since the average number of errors is high, parity checks on large blocks fail to isolate them fast enough. Therefore, those errors are detected in subsequent passes with a larger block size. Such a large block size leads to high leakage when running the CONFIRM method on them. Therefore, in

this case, 4-bit initial blocks must be chosen.

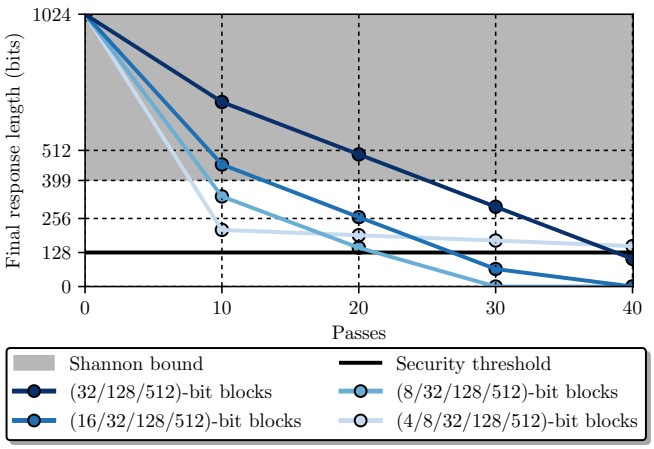

Fig. 8: Final response length after executing the CASCADE protocol on a 1024-bit response with different numbers of passes and initial block sizes. Here, the error rate is 15%.

The failure rates observed using this configuration are shown in Figure 9. The only solution to obtain a realistic failure rate is to start the reconciliation protocol with 4-bit blocks. The other configurations include too few blocks per pass to be able to detect and correct the 154 errors on average.

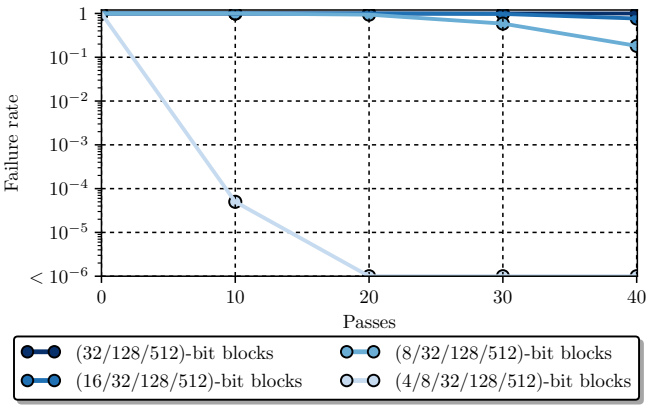

Fig. 9: Failure rate for a 1024-bit response with different numbers of passes and initial block sizes. Here, the error rate is 15%. The first two curves are really close to the top.

The examples given above show how a designer can easily balance the leakage and the failure rate by tuning the CASCADE parameters like the number of passes and the initial block size. It can be tempting to start with large blocks to limit the leakage induced by parity checks, but then executing the CONFIRM method on these blocks leaks much more. On the other hand, adding passes with a block size of $n/2$ reduces the failure rate efficiently without leaking too many bits.

The following section presents the typical design flow that a designer should follow to tune the CASCADE parameters to a particular use case.

4) Design flow: After characterisation of the PUF, the error rate can be estimated. Depending on the target application, the designer can then select a failure rate, and estimate the protocol parameters that have to be chosen to achieve it: the initial block size and the number of passes. These two parameters make it possible to compute the leakage. They give the number of bits that can be kept secret. If the number of secret bits is too low for the target application, the designer can request more bits from the PUF in order to obtain a final secret key of sufficient length.

Table III shows which parameters can be chosen for the CASCADE protocol in real-life examples to achieve a failure rate of 10^{-4}, 10^{-6} 10^{-8} and a security of 128 bits, which is the usual value for the length of a symmetric encryption key. Several types of PUF architectures are considered: a ring-oscillator (RO), transient-effect ring-oscillator (TERO) and SRAM PUFs. The associated error rate provided in the original articles is used to evaluate the initial block size k_1, the number of passes n_{passes} and the number of bits required from the PUF. The number of bits from the PUF is set to be a power of two. Three different failure rates, 10^{-4}, 10^{-6} and 10^{-8} are considered.

B. Leakage estimation

As highlighted in [20], the minimum information required to recover a variable X when an altered version Y is known is given by the conditional entropy $H(X|Y)$. When considering a BSC of error rate ε, the conditional entropy is related to the error rate. Y is then an instance of the X variable, in which every bit has a flipping probability of ε. The minimum information to be exchanged between the two parties in order to reconcile their respective responses is given in Equation (3), where n is the length of the response and $h(\varepsilon)$ is the Shannon entropy.

$$nh(\varepsilon) = n(-\varepsilon log_2(\varepsilon) - (1-\varepsilon)log_2(1-\varepsilon)) \quad (3)$$

Therefore, the maximum number of PUF response bits kept secret after the reconciliation protocol is carried out is given in Equation (4).

$$n - nh(\varepsilon) = n(1 - h(\varepsilon)) \quad (4)$$

For example, for a 5% error rate, one cannot expect to keep secret more than 182 bits from an initial 256-bit response. However, if the error rate is lower, 2% for instance, then up to 219 bits can be kept secret. This is an overestimation, and tighter bounds can be found in the literature [31]. However, there is no analytical value for the exact leakage associated with the execution of CASCADE. In practise, at most one bit is leaked each time a parity value is transmitted over the public channel. Therefore, the leakage mainly depends on the number of passes and the size of the block. In order to limit leakage, the protocol parameters must be carefully chosen.

TABLE III: Examples of parameters to achieve failure-rates of 10^{-4}, 10^{-6} and 10^{-8} for different PUF architectures, aiming at keeping at least 128 bits secret.

					Target failure rate								
					10^{-4}			10^{-6}			10^{-8}		
PUF	Article	Target	Technology node	Error rate ε	k_1 [bits]	#passes	PUF bits required	k_1 [bits]	#passes	PUF bits required	k_1 [bits]	#passes	PUF bits required
RO	[22]	FPGA	90 nm	0.9%	8	10	256	8	20	256	8	30	256
	[23]	ASIC	65 nm	2.8%	8	15	256	8	25	256	8	30	256
TERO	[24]	FPGA	90 nm	1.7%	8	15	256	8	25	256	8	30	256
	[25]	FPGA	28 nm	1.8%	8	15	256	8	25	256	8	30	256
	[26]	ASIC	350 nm	0.6%	8	10	256	8	20	256	8	30	256
SRAM	[27]	FPGA	—	4%	8	15	256	8	20	512	8	30	512
	[28]	FPGA	—	10%	4	15	512	8	25	512	4	44	512
	[29]	FPGA	65 nm	15%	4	15	1024	4	20	1024	4	50	1024
	[30]	ASIC	65 nm	5.5%	8	18	256	8	20	512	8	30	512

V. Implementation

A. Implementation principle

The implementation of the CASCADE protocols consists in both a device-side and server-side implementation. We assume that the server has high computational capabilities, whereas the implementation should be as lightweight as possible on the device. Table IV summarises how computations are distributed between the device and the server.

TABLE IV: Distribution of features between device and server.

Feature	Device side	Server side
Block-size computation		×
Parity computations	×	×
Permutations		×
Error detection		×
Error correction		×

1) Device side: For the device side implementation of CASCADE, we chose cost-optimised FPGA devices from the Xilinx Spartan and Intel Cyclone families, since those are typically the ones used in applications requiring a lightweight root of trust like a PUF. We selected two FPGAs per family, namely Spartan 3, Spartan 6, Cyclone III and Cyclone V. We integrated the module computing the parity in a simple controller with three states: *idle*, *compute_parity* and *send_parity*. The only computation carried out on the device is the parity computation. It must be done on blocks of variable length.

a) Implementation option 1: Multiplexing the response bits: The first option is to multiplex the PUF response bits to an XOR gate one after the other. We assume the PUF response is stored in an n-bit register. After the XOR gate, the intermediate result is sampled by a D flip-flop. This is illustrated in Figure 10.

However, one can make use of an existing shift-register to reduce the overhead. This is detailed in the next paragraph.

b) Implementation option 2: Making an existing shift register circular: Among the PUF architectures we considered, both the RO and TERO PUFs have the characteristic to not derive the full response immediately. Instead, the RO PUF generates one response bit per challenge, as the result of the comparison between the frequencies of two ring oscillators. Similarly, the TERO-PUF generates from one up to three bits of response per challenge. In both cases, in order to obtain the full response, the response bits must be stored in a shift register. Such existing shift register can be leveraged to further reduce the logic resources overhead of the implementation.

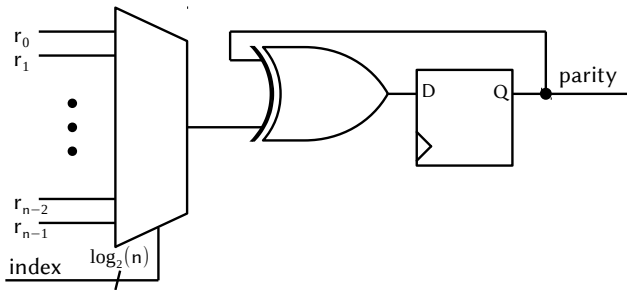

Fig. 10: Hardware architecture of the parity computation module when multiplexing the response bits stored in registers.

The architecture of the parity computation module in this case is detailed in Figure 11. The bottom left D flip-flop samples the response bit once it is available at the output of the shift register. The XOR gate computes the parity value, which is then sampled by another D flip-flop.

Therefore, the only additional components to add to the PUF are the following, since the shift-register is already present:

- One $log_2(n)$-bit counter,
- One XOR gate,
- Two flip-flops.

The shift register is made circular by connecting its output to its input. Depending by how much the data in the circular shift register is shifted, the appropriate response bit is selected and sent to the parity computation module. The amount of shifting required to select a specific response bit is controlled by a counter, connected to the Δ input.

Let us identify two consecutively selected response bits as $r[i]$ and $r[j]$. $r[j]$ is then the response bit to be selected after $r[i]$. Two cases can occur when selecting these response bits.

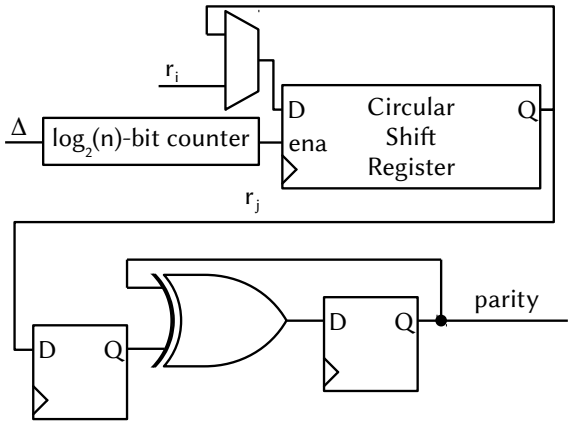

Fig. 11: Hardware architecture of the parity computation module where an existing shift-register storing the response is reused and made circular.

- If $j > i$, the counter must be set to count for $j - i$ clock cycles, which is the difference of indexes.
- If $j < i$, the counter must be set to count for $n + j - i$ clock cycles, which is the difference of indexes when wrapping beyond the response size n.

The counter must then count for Δ clock cycles (see Equation (5)).

$$\Delta = (j - i) \ mod \ n \tag{5}$$

Therefore, the counter must be $log_2(n)$-bit wide so that it can index all response bits.

c) Implementation option 3: Storing the response in RAM: Finally, the last implementation alternative we explored is when the response is stored in RAM. In order to store 2256, 512 and 1024-bit responses, 32x8, 64x8 and 128x8-bit RAM blocks are used respectively, and the response is split into bytes. In this case, the multiplexing capability is intrinsic to the RAM, and response bytes can be accessed directly. In order to select the response bits individually, a 8:1 multiplexer is used. When an index is sent by the server, the three least significant bits are used as a selection signal by the multiplexer, to select the response bit in the response byte. The most significant bits down to the third are used as the address signal by the RAM. The remaining part of the parity computation module is the same, comprising one D flip-flop and one XOR gate. The hardware architecture is shown in Figure 12.

2) Server side: All the other computations are handled by the server, where Python was used for development. The communication between the server and the device consists in a list of indexes sent by the server to the device and a parity value sent back to the server from the device. The permutations are selected by the server, and only individual indexes are transmitted to the device. Therefore, the permutation layer is entirely implemented on the server side. After the error has been located using binary search, it is corrected on the server.

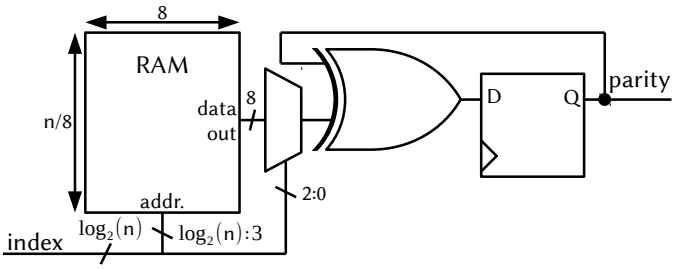

Fig. 12: Hardware architecture of the parity computation module where the PUF response is stored in RAM

B. Implementation results

1) Logic resources: We give the implementation results on the four FPGA targets previously considered with the same metrics: number of LUTs, number of D flip-flops, number of RAM bits and number of Slices/ALMs/LCs[2]. We provide the number of Slices/ALMs/LCs as a mean of comparison with existing work, but only LUTs and D flip-flops figures should be taken into account when comparing between different families. Moreover, more recent FPGAs like Spartan 6 or Cyclone V embed LUTs with a greater number of inputs. Therefore, implementing the same logic function takes less LUTs on those. Similarly to the method used for Table I, we only report the logic resources used by the parity computation module. Thus for implementation option 1 and 2, we do not report the registers used to store the PUF response. However, for implementation option 3, we report the RAM bits used to store the PUF response. This is shown in Table V.

As we can see when comparing with the results obtained for existing error-correcting codes presented in Table I, the CASCADE protocol is very lightweight.

For implementation option 1, where a multiplexer is used, most of the resources are required to implement the large n:1 multiplexer. Thus the number of LUTs required to implement it increases linearly with the response length. Even though we reported implementation results on FPGAs, we can note that such an implementation option is much more suited for ASICs. Indeed, large multiplexers are not efficiently implemented on FPGAs because there are too few D flip-flops per logic element compared to the number of LUT entries.

The second implementation option, reusing a shift register to make it circular, is much more lightweight. The main advantage here is that since the number of flip-flops required by the counter grows logarithmically with the counter size, so does the number of LUTs. Therefore, when the response length is doubled, only one extra flip-flop is needed. This implementation is suited for either ASICs or FPGAs.

Finally, the third implementation option is clearly oriented toward FPGAs. Indeed, on such devices, RAM is available and easily usable. By taking advantage of the intrinsic multiplexing capability of the RAM, the amount of logic resources required drops significantly compared to the two previous implementa-

[2]ALM: Adaptive Logic Module LC: Logic Cell

256-bit response

Target device	Option 1: Registers and multiplexer				Option 2: Circular shift register				Option 3: RAM			
	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic
Xilinx Spartan 3[a]	133	1	0	67 Slices	26	12	0	17 Slices	5	1	256	3 Slices
Xilinx Spartan 6[b]	67	1	0	19 Slices	17	12	0	7 Slices	3	1	256	1 Slice
Intel Cyclone III[a]	170	1	0	170 LCs	25	20	0	26 LCs	6	1	256	6 LCs
Intel Cyclone V[c]	86	1	0	46 ALMs	23	20	0	13 ALMs	4	1	256	3 ALMs

512-bit response

Target device	Option 1: Registers and multiplexer				Option 2: Circular shift register				Option 3: RAM			
	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic
Xilinx Spartan 3[a]	265	1	0	133 Slices	26	13	0	18 Slices	5	1	512	3 Slices
Xilinx Spartan 6[b]	171	1	0	92 Slices	25	13	0	11 Slices	3	1	512	1 Slice
Intel Cyclone III[a]	342	1	0	342 LCs	28	22	0	29 LCs	6	1	512	6 LCs
Intel Cyclone V[c]	171	1	0	87 ALMs	26	22	0	14 ALMs	4	1	512	3 ALMs

1024-bit response

Target device	Option 1: Registers and multiplexer				Option 2: Circular shift register				Option 3: RAM			
	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic	LUTs	DFFs	RAM bits	Logic
Xilinx Spartan 3[a]	529	1	0	265 Slices	28	14	0	18 Slices	5	1	1024	3 Slices
Xilinx Spartan 6[b]	341	1	0	182 Slices	27	14	0	10 Slices	3	1	1024	1 Slice
Intel Cyclone III[a]	683	1	0	683 LCs	30	24	0	31 LCs	6	1	1024	6 LCs
Intel Cyclone V[c]	342	1	0	176 ALMs	28	24	0	15 ALMs	4	1	1024	3 ALMs

[a] 4-input LUTs
[b] 6-input LUTs
[c] 7-input LUTs

TABLE V: Logic resources required for three implementation options of the parity computation module and three response sizes.

tion options presented. Moreover, since the 8:1 multiplexer only selects one specific bit in the response byte, its size remains constant when the length of the response increases. Since the RAM is used to store the PUF response, the number of RAM bits required grows linearly with the response length. With implementation figures between 3 and 6 LUTs and 1 D flip-flop, this FPGA implementation of the `CASCADE` protocol using RAM is by far the most lightweight error-correction module to date.

2) Execution Time: Another criterion used to evaluate the different error-correcting codes is their execution time. We give the execution times in number of clock cycles to be device-independent. Implementation options 1 and 3 have an identical way of selecting the PUF bits, therefore they have the same execution time. Implementation option 2 on the other hand has a longer execution time since it requires to shift the circular shift register to select the appropriate response bit. The execution time of the protocol can be split into a fixed and a variable portion. The fixed portion includes the parity computations aimed at detecting the errors, which are executed after the scrambling step of each pass. The variable portion is related to the execution of the `CONFIRM` method. If the errors are detected in the initial passes, then the `CONFIRM` method will be applied to small blocks. On the other hand, if errors remain until the last passes, correcting them means `CONFIRM` will have to be executed on large blocks. The larger the blocks, the longer the parity computations. This is detailed in the following paragraphs.

a) Implementation options 1 & 3: These two implementations multiplex the response bits directly, in one clock cycle. In this case, accessing the response bits has $\mathcal{O}(1)$ time complexity for an n-bit response.

Computing the parities for all the blocks of an n-bit response requires n clock cycles with the module shown in Figure 10. The number of clock cycles required for parity computations when executing the `CONFIRM` method on a t-bit block is given by Equation (6). This corresponds to computing parities on blocks of sizes starting at $t/2$ bits down to 1 bit.

$$\sum_{i=1}^{log_2(t)} \frac{t}{2^i} = t - 1 \tag{6}$$

Let us consider the previous case of a 256-bit response and a 2% error-rate. On average, five bits are flipping. We assume the protocol starts with 32-bit blocks and runs for 15 passes. We distinguish two border cases. The actual execution time of one execution of the protocol lies between those two cases.

In the best case, the errors are corrected as soon as possible. The binary search is conducted on smaller blocks and is shorter. The five errors are corrected in the first pass. Therefore, the device-side execution time is:

$$256 \times 15 + 5 \times (32 - 1) = 3,995 \text{ clock cycles}$$

In the worst case, there are more than five errors. For example, let us assume that 15 bits are faulty. This occurs with a probability of 2.10^{-4} Moreover, since we are in the worst case scenario, the errors are corrected as late as possible. The binary search is then conducted on larger blocks and is longer. The errors are corrected in the last passes.

In this case, the execution time is:

$$256 \times 15 + 15 \times (128 - 1) = 5,745 \text{ clock cycles}$$

b) Implementation option 2: In order to select a response bit, the circular shift register must be shifted by an amount δ, with $\delta \in [1; n-1]$. On average, reaching the next response bit requires $n/2$ shifts. Therefore, accessing the response bits has $\mathcal{O}(n)$ time complexity for an n-bit response.

Therefore, computing the parity on a t-bit block taken out of an n-bit response is done in $(t.n)/2$ clock cycles on average. Since there are n/t of these blocks in the response, computing the parity of all the blocks of an n-bit response requires $n^2/2$ clock cycles on average. This is much longer than with the other options, for which only n clock cycles are necessary.

The other step which increases in execution time with this implementation is the `CONFIRM` method. The number of clock cycles required to execute the `CONFIRM` method on a t-bit block is given in Equation (7).

$$\sum_{i=1}^{log_2(t)} \frac{t.\frac{n}{2}}{2^i} = \frac{n.(t-1)}{2} \tag{7}$$

We need to consider the best and worst case for the `CONFIRM` method. A 256-bit response with a 2% error rate is studied, with five bits flipping on average. We assume the protocol starts with 32-bit blocks and runs for 15 passes.

In the best case, the errors are corrected as early as possible, in the first pass where blocks are 32 bits long. Therefore, the associated device-side execution time is:

$$\frac{256^2}{2} \times 15 + 5 \times \frac{256 \times (32-1)}{2} = 511,360 \text{ clock cycles}$$

In the worst case, the errors are corrected as late as possible, when the blocks are 128 bits long. Moreover, there are 15 of them instead of 5. In this case, the execution time is:

$$\frac{256^2}{2} \times 15 + 15 \times \frac{256 \times (128-1)}{2} = 735,360 \text{ clock cycles}$$

c) Comparison to existing codes: Table VI gives a comparison with existing error-correcting codes in terms of execution time. We considered two corner cases for `CASCADE`. First, the protocol was executed on a 256-bit response with a 1% error rate. The errors were corrected as early as possible, making it the best case scenario. In the worst case, we considered a 1024-bit response with $\varepsilon = 15\%$, in which the errors were corrected as late as possible.

The execution time of the `CASCADE` protocol is very dependent on the size of the response to correct. It also depends on the error rate, since the error-correction steps achieved by the `CONFIRM` method are also a source of execution time. Depending on when the errors are corrected, the execution time also varies to a great extent.

Implementation options 1 and 3 have execution times between 4,000 and 200,000 clock cycles. This is comparable to the range observed for existing codes, between 1,210 and 108,000 clock cycles. However, when implementation option 2 is selected, the execution time increases dramatically. This is compliant with the $\mathcal{O}(n)$ time complexity. Therefore, for high error-rates, options 1 and 3 should be preferred.

TABLE VI: Execution Time in clock cycles of different codes with different constructions.

Article	Construction and code(s)	Execution time (cycles)
[1]	Concatenated: Repetition (7, 1, 3) and BCH (318, 174, 17)	50,831
[2]	Complementary IBS with Reed-Muller (2, 6)	—
[3]	Reed-Muller (4, 7)	108,000
[6]	BCH (255, 21, 55)	—
[16]	Reed-Muller (2, 6)	10,298
[17]	Concatenated: Repetition (5, 1, 5) and Reed-Muller (1, 6)	6,505
[17]	Concatenated: Repetition (11, 1, 11) Golay G_{24}(24, 13, 7)	1,210
[18]	Differential Sequence Coding	29,243
`CASCADE` options 1 & 3	on 256-bit responses and $\varepsilon = 1\%$, 15 passes, starting with 32-bit blocks (errors corrected as early as possible)	3,933
`CASCADE` options 1 & 3	on 1024-bit responses and $\varepsilon = 15\%$, 45 passes, starting with 4-bit blocks (errors corrected as late as possible)	203,622
`CASCADE` option 2	on 256-bit responses and $\varepsilon = 1\%$, 15 passes, starting with 32-bit blocks (errors corrected as early as possible)	503,424

On the implementation side, we can note that the logic function is very simple here, and has a very short critical path. A higher clock frequency than the one used by the other logic of the circuit could then be used to reduce the delay required by the `CASCADE` protocol.

Due to the great interactivity of the `CASCADE` protocol, the main execution time bottleneck is the communication between the device and the server. It can be order of magnitude slower that intra-device communication. Thus the associated execution time depends to a great extent on the target platform.

VI. Discussion

A. Privacy amplification

For this analysis, we assume to be in the random oracle model. The number of bits leaked during the error correction step can be estimated. However, the remaining entropy is not only concentrated in the non-leaked bits. Instead, it is evenly spread over all the bits of the response. Moreover, the initial

responses usually have poor statistical properties, such as not being independent and identically distributed. The next step is thus to shorten the response in order to get all the bits with maximum entropy. This is called *privacy amplification*. In practise, a cryptographic hash function is used, and is assumed to behave as a random oracle [32]. Figure 13 shows how the number of bits varies at different steps.

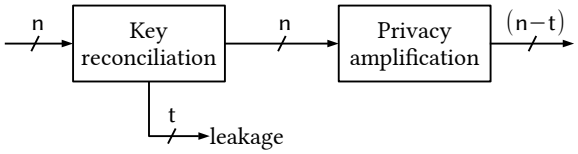

Fig. 13: Changes in the number of bits in the response at different steps.

After key reconciliation, t bits are leaked. Therefore, the hash function used for privacy amplification should have an output of size inferior or equal to $(n-t)$ bits so that all the output bits have maximum entropy. A lightweight hash function can be used to achieve privacy amplification with low area penalty. SPONGENT [33] was used in [1], and requires only 22 Slices on a Xilinx Spartan 6 FPGA for the 128-bit output block option. An alternative is to implement Toeplitz hashing [34], which was chosen in [35]. Based on an LFSR, this construction occupies 59 Slices on a Xilinx Spartan 3 FPGA. Other low-area hash functions can be found on the SHA-3 webpage, in the "low-area implementations" section[3].

B. Replacing parity check with hashing

Some works [36], [37] suggested using a hash function instead of a parity check to detect the errors in corresponding response blocks. This would enable detection of two errors in the same block, which is not possible using the simple parity check. However, this error detection method cannot be used with small blocks. For example, if the CASCADE protocol starts with 8-bit blocks, then an attacker can pre-compute a 2^8-bit look-up table containing the hashes if the hash function is public. By observing the successive hash values sent by the device, the attacker could easily recover the PUF response.

C. Security analysis

In this section, we assume again that the PUF responses have full entropy.

1) Brute force attack: By observing the indexes sent to the PUF and the associated parity value that it returns, an attacker can build a system of linear equations describing the parity relations between the indexes. This system can then be solved to obtain the PUF response by Gaussian elimination. However, the system of linear equations does not fully specify the values of the variables, and multiple responses can satisfy these equations. Therefore, an attacker would have to exhaustively explore the remaining space until the correct response is revealed.

Assuming t parity bits have been leaked during the protocol execution on n-bit responses, 2^{n-t} possible responses are still to explore for the attacker. Therefore, after executing the CASCADE protocol, taking the conservative estimation that one bit of information is leaked every time a parity value is sent, the security level drops from 2^n to 2^{n-t}. As detailed before, it is up to the PUF designer to tune the protocol parameters so that t remains as low as possible in order to limit the leakage.

2) Device impersonation: chosen parity values scenario: An attacker could impersonate the PUF and return parity values of his choice to the server, with the aim of setting the reference response r_0 to a chosen value. This corresponds to a *chosen parity values* scenario. We propose the following counter-measure to address this threat.

a) Counter-measure: Device impersonation is thwarted by limiting the number of modifiable bits on the server side. Since response bits have probability ε of flipping, the total number of bits that flipped in a n-bit response follows the binomial distribution $\mathcal{B}(n,\varepsilon)$. We chose to allow up to m bits to be modified. m was chosen so that the probability that m bits flip is lower than the expected failure rate. For example, for a 256-bit response and a 2% error rate, if a failure rate of 10^{-6} is specified, then we search for the number of bits flipping m such that $Pr(X = m) < 10^{-6}$. Therefore, the maximum number of bit modifications we would allow for on the server side in this configuration is $m = 20$.

The general value for the number of modifiable bits m on the server side with respect to the failure rate f is given in Equation (8), where X is the number of bits modified by executing the CASCADE protocol.

$$m : P(X = m) < f \tag{8}$$

Beyond this limit, the probability that an attacker is trying to modify the response is higher than the failure rate of the protocol. Thus further modifications are not allowed and the protocol is stopped.

3) Server impersonation: chosen indexes scenario: Following our use case, the main threat here is server impersonation. Indeed, this would allow an attacker to unlock an IC by sending the activation word encrypted with a chosen PUF response. In order to do so, an attacker must construct a PUF response. He can choose the indexes sent to the PUF, and obtain the associated parity values. This corresponds to a *chosen indexes* scenario. The point here is to obtain a sufficient number of parity relations between the PUF bits, to forge a PUF response.

Considering a set of parity relations as a sufficiently determined system of equations over $GF(2)$, Gaussian elimination can be used to recover the PUF response bits. However, this requires the attacker to be able to build a sufficiently determined system of equations. Therefore, we propose the two following counter-measures against server impersonation. In addition, deterministic scrambling, presented in Subsection VI-D, demonstrates another way to avoid server impersonation.

a) Counter-measure 1: This countermeasure comes in two aspects. The point is to prevent the attacker from building a sufficiently determined system of equations. First, a hard limit is set on the device-side for the number of parity values

[3]http://ehash.iaik.tugraz.at/wiki/SHA-3_Hardware_Implementations

which can be sent when the protocol is executed. By setting it at the security requirement, 128-bit in our case, the designer can be sure that at least 128 bits are kept secret. However, this can be circumvented by resetting the device and executing the CASCADE protocol multiple times, to obtain more linear equations. Therefore, we propose to add an extra counter-measure.

b) Counter-measure 2: At the beginning of each execution of the CASCADE protocol, a new PUF response must be requested. By doing so, the attacker can obtain more parity relations between the PUF response bits. However, since a new response has been generated, some bits might have flipped to an erroneous value or flipped back to a correct value with respect to the reference response r_0. Therefore, the parity relations do not correspond to the same PUF response, and cannot be used to forge a response by Gaussian elimination. This problem is similar to the *Learning parities with noise* problem, which is considered a hard problem and has been used as the hardness assumption in constructing cryptographic schemes [38]. Learning parities with noise is equivalent in complexity to decoding from a random linear code [39], which is known to be an NP-hard problem. Proving rigorously the equivalence between LPN and the case we study here requires further investigation.

4) Single index request: By challenging the system with only one index i, an attacker could obtain the value of the PUF response at index i. Doing so sequentially for all indexes would allow the attacker to recover the whole response. A simple and not costly secure controller should thus be implemented in the system to avoid this type of manipulation. For example, single index requests can be counted and not allowed anymore once a specific threshold is reached. Indeed, knowing the error-rate, one can fix a threshold on the number of faulty bits. Above this threshold, single index requests are not allowed anymore. It prevents an attacker from recovering the entire response. Such a counter is very lightweight. Determining the size of the counter can be done using Equation (8). For the example given above, counting up to $m = 20$ requires only 5 extra D flip-flops. All those counter-measures are fully compatible with the industrial use case we consider here.

5) Helper data manipulation: Recent works highlight the fact that helper data can be manipulated [40]. Since the CASCADE protocol only requires exchanging simple parity values, manipulation is not a threat and is handled like impersonation.

D. Deterministic scrambling

The first step of the key reconciliation protocol is scrambling. Depending on the PUF architecture used, characterisation can be done right after the PUF is implemented on the device. This will detect which bits of the response are the most unstable, i.e. the ones whose value is the most likely to change over time [41]. At that point, the chosen permutation can assign one unstable bit per response block, so that the potential errors are easily detected and corrected in the first passes of the CASCADE protocol.

Another point of using deterministic scrambling is to thwart attacks which aim at fully determining the system of parity equations in order to solve it and recover the response, *i.e.* server impersonation. This could be achieved for example by executing the CASCADE protocol on the same circuit multiple times, since random permutations are normally used. If deterministic scrambling is used instead, the same fixed set of permutations is used for all protocol executions. Therefore, running the protocol multiple times does not help an attacker in building a sufficiently determined system of parity equations, hence avoiding the previously described threat.

E. Dynamic parameterisability

Tables III gives the parameters of the CASCADE protocol for several error-rates and failure rates. However, those parameters are not set in stone and can be modified later. This can be necessary if the error rate increases under poor environmental conditions. The server can then increase the number of passes or start the protocol with smaller blocks, so that more errors are detected. This must be taken into consideration when choosing the nominal parameters, so that these modifications do not lower the security level too much. This is visible in Table III, in the SRAM row: starting with 8-bit blocks can correct the errors when $\varepsilon = 5.5\%$ or $\varepsilon = 10\%$ depending on the number of passes, 20 or 25. Hence if the PUF is expected to exhibit an error rate of 5% but it increases to 10% later, the number of passes can be changed to handle it.

VII. Conclusion

This article proposes to use key reconciliation protocols for error correction of PUF responses. We show that this interactive method is efficient at reaching very low failure rates while requiring less bits from the PUF than existing error-correcting codes. Although it incurs significant communication compared to existing error-correcting codes, its main advantage is requiring very low area overhead on the device and small execution time for the computations. Another advantage of this solution is its great flexibility. Parameters can be easily tuned to the design constraints. This makes it a suitable option for resource constrained applications, which are the ones targeted by PUFs in the first place. Our work clearly points toward using silicon PUFs and key reconciliation protocols in an industrial context for intellectual property protection and authentication of ICs.

Acknowledgements

The work presented in this paper was realised in the frame of the SALWARE project number ANR-13-JS03-0003 supported by the French "*Agence Nationale de la Recherche*" and by the French "*Fondation de Recherche pour l'Aéronautique et l'Espace*", funding for this project was also provided by a grant from "*La Région Rhône-Alpes*".

This work has also received funding from the European Union's Horizon 2020 research and innovation programme in the framework of the project HECTOR (Hardware Enabled Crypto and Randomness) under grant agreement No 644052.

REFERENCES

[1] R. Maes, A. V. Herrewege, and I. Verbauwhede, "PUFKY: A fully functional PUF-based cryptographic key generator," in *International Workshop on Cryptographic Hardware and Embedded Systems*, vol. 7428, Leuven, Belgium, Sep. 2012, pp. 302–319.

[2] M. Hiller, D. Merli, F. Stumpf, and G. Sigl, "Complementary IBS: application specific error correction for PUFs," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, San Francisco, CA, USA, Jun. 2012, pp. 1–6.

[3] M. Hiller, L. Kurzinger, G. Sigl, S. Müelich, S. Puchinger, and M. Bossert, "Low-area reed decoding in a generalized concatenated code construction for PUFs," in *IEEE Computer Society Annual Symposium on VLSI*, Montpellier, France, Jul. 2015, pp. 143–148.

[4] C. H. Bennett, F. Bessette, G. Brassard, L. Salvail, and J. A. Smolin, "Experimental quantum cryptography," *Journal of Cryptology*, vol. 5, no. 1, pp. 3–28, 1992.

[5] G. Brassard and L. Salvail, "Secret-key reconciliation by public discussion," in *EUROCRYPT*, Lofthus, Norway, May 1993, pp. 410–423.

[6] A. V. Herrewege, S. Katzenbeisser, R. Maes, R. Peeters, A. Sadeghi, I. Verbauwhede, and C. Wachsmann, "Reverse fuzzy extractors: Enabling lightweight mutual authentication for PUF-enabled RFIDs," in *International Conference on Financial Cryptography and Data Security*, Kralendijk, Bonaire, Feb. 2012, pp. 374–389.

[7] D. A. Hodges, "Building the fabless/foundry business model," *IEEE Solid-State Circuits Magazine*, vol. 3, no. 4, pp. 7–44, 2011.

[8] C. Gorman, "Counterfeit chips on the rise," *IEEE Spectrum*, vol. 49, no. 6, pp. 16–17, 2012.

[9] U. Guin, K. Huang, D. DiMase, J. M. Carulli, M. Tehranipoor, and Y. Makris, "Counterfeit integrated circuits: A rising threat in the global semiconductor supply chain," *Proceedings of the IEEE*, vol. 102, no. 8, pp. 1207–1228, 2014.

[10] AGMA, "Managing the risks of counterfeiting in the information technology industry," Alliance for Gray Market and Counterfeit Abatement, Tech. Rep., 2005.

[11] B. Colombier and L. Bossuet, "Survey of hardware protection of design data for integrated circuits and intellectual properties," *IET Computers & Digital Techniques*, vol. 8, no. 6, pp. 274–287, Nov. 2014.

[12] J. Delvaux, D. Gu, R. Peeters, and I. Verbauwhede. (Jan. 2015). A survey on lightweight entity authentication with strong PUFs.

[13] B. Colombier, L. Bossuet, and D. Hély, "From secured logic to IP protection," *Elsevier Microprocessors and Microsystems*, vol. 47, pp. 44–54, 2016.

[14] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. B. Robshaw, Y. Seurin, and C. Vikkelsoe, "PRESENT: an ultra-lightweight block cipher," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Vienna, Austria, Sep. 2007, pp. 450–466.

[15] J. Delvaux, D. Gu, D. Schellekens, and I. Verbauwhede, "Helper data algorithms for PUF-based key generation: Overview and analysis," *IEEE Transactions on CAD of Integrated Circuits and Systems*, vol. 34, no. 6, pp. 889–902, 2015.

[16] R. Maes, P. Tuyls, and I. Verbauwhede, "Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Lausanne, Switzerland, Sep. 2009, pp. 332–347.

[17] C. Bösch, J. Guajardo, A. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient helper data key extractor on FPGAs," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Washington, D.C., USA, Aug. 2008, pp. 181–197.

[18] M. Hiller, M. Yu, and G. Sigl, "Cherry-picking reliable PUF bits with differential sequence coding," *IEEE Trans. Information Forensics and Security*, vol. 11, no. 9, pp. 2065–2076, 2016.

[19] M. Hiller, M. Yu, and M. Pehl, "Systematic low leakage coding for physical unclonable functions," in *ACM Symposium on Information, Computer and Communications Security*, Singapore, Apr. 2015, pp. 155–166.

[20] J. Martinez-Mateo, C. Pacher, M. Peev, A. Ciurana, and V. Martin, "Demystifying the information reconciliation protocol CASCADE," *Quantum Information & Computation*, vol. 15, no. 5&6, pp. 453–477, 2015.

[21] C. Pacher, P. Grabenweger, J. Martinez-Mateo, and V. Martin, "An information reconciliation protocol for secret-key agreement with small leakage," in *IEEE International Symposium on Information Theory*, Hong Kong, Hong Kong, Jun. 2015, pp. 730–734.

[22] A. Maiti, J. Casarona, L. McHale, and P. Schaumont, "A large scale characterization of RO-PUF," in *IEEE International Symposium on Hardware-Oriented Security and Trust*, Anaheim CA, USA, Jun. 2010, pp. 94–99.

[23] R. Maes, V. Rozic, I. Verbauwhede, P. Koeberl, E. van der Sluis, and V. van der Leest, "Experimental evaluation of physically unclonable functions in 65 nm CMOS," in *European Solid-State Circuit Conference*, Bordeaux, France, Sep. 2012, pp. 486–489.

[24] L. Bossuet, X. T. Ngo, Z. Cherif, and V. Fischer, "A PUF based on transient effect ring oscillator and insensitive to locking phenomenon," *IEEE Transaction on Emerging Topics in Computing*, vol. 2, no. 1, pp. 30–36, 2014.

[25] C. Marchand, L. Bossuet, and A. Cherkaoui, "Enhanced TERO-PUF implementations and characterization on FPGAs," in *International Symposium on Field-Programmable Gate Arrays*, Monterey, CA, USA, Feb. 2016, p. 282.

[26] A. Cherkaoui, L. Bossuet, and C. Marchand, "Design, evaluation and optimization of physical unclonable functions based on transient effect ring oscillators," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1291–1305, 2016.

[27] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA intrinsic PUFs and their use for IP protection," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Vienna, Austria, Sep. 2007, pp. 63–80.

[28] A. Aysu, E. Gulcan, D. Moriyama, P. Schaumont, and M. Yung, "End-to-end design of a PUF-based privacy preserving authentication protocol," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Saint-Malo, France, Sep. 2015.

[29] R. Maes, P. Tuyls, and I. Verbauwhede, "A soft decision helper data algorithm for SRAM pufs," in *IEEE International Symposium on Information Theory*, Seoul, Korea: IEEE, Jun. 2009, pp. 2101–2105.

[30] M. Claes, V. van der Leest, and A. Braeken, "Comparison of SRAM and FF-PUF in 65nm technology," in *Nordic Conference on Secure IT Systems*, vol. 7161, Tallinn, Estonia, Oct. 2011, pp. 47–64.

[31] R. L.-Y. Ng, "A probabilistic analysis of CASCADE," in *International conference on quantum cryptography*, 2014.

[32] B. Barak, Y. Dodis, H. Krawczyk, O. Pereira, K. Pietrzak, F. Standaert, and Y. Yu, "Leftover hash lemma, revisited," in *Annual Cryptology Conference*, vol. 6841, Springer, 2011, pp. 1–20.

[33] A. Bogdanov, M. Knezevic, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, "SPONGENT: A lightweight hash function," in *International Workshop on Cryptographic Hardware and Embedded Systems*, Nara, Japan, Sep. 2011, pp. 312–325.

[34] H. Krawczyk, "LFSR-based hashing and authentication," in *Annual International Cryptology Conference*, vol. 839, Santa Barbara, California, USA, Aug. 1994, pp. 129–139.

[35] R. Maes, D. Schellekens, P. Tuyls, and I. Verbauwhede, "Analysis and design of active IC metering schemes," in *IEEE International Workshop on Hardware-Oriented Security and Trust*, San Francisco CA, USA, Jul. 2009, pp. 74–81.

[36] C. H. Bennett, G. Brassard, and J. Robert, "Privacy amplification by public discussion," *SIAM Journal on Computing*, vol. 17, no. 2, pp. 210–229, 1988.

[37] A. Yamamura and H. Ishizuka, "Error detection and authentication in quantum key distribution," in *Australasian Conference on Information Security and Privacy*, vol. 2119, Sydney, Australia, Jul. 2001, pp. 260–273.

[38] K. Pietrzak, "Cryptography from learning parity with noise," in *38th Conference on Current Trends in Theory and Practice of Computer Science*, Špindlerův Mlýn, Czech Republic, Jan. 2012, pp. 99–114.

[39] E. R. Berlekamp, R. J. McEliece, and H. C. A. van Tilborg, "On the inherent intractability of certain coding problems," *IEEE Transactions on Information Theory*, vol. 24, no. 3, pp. 384–386, 1978.

[40] J. Delvaux and I. Verbauwhede, "Key-recovery attacks on various RO PUF constructions via helper data manipulation," in *Design, Automation & Test in Europe Conference*, Dresden, Germany, Mar. 2014, pp. 1–6.

[41] R. Maes, "An accurate probabilistic reliability model for silicon PUFs," in *International Workshop on Cryptographic Hardware and Embedded Systems*, vol. 8086, Santa Barbara, CA, USA, Aug. 2013, pp. 73–89.