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

September 29, 2016

Abstract

Physical Unclonable Functions (PUFs) are promising primitives for lightweight integrated circuit authentication. 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, but need to be corrected first. This is currently achieved using error correcting codes, which generate helper data through a one-time process. 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 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 few logic resources on the device side, at least three times fewer than existing error correcting codes. The leakage caused by the key reconciliation process is limited and easily computable. Results of implementation on various FPGA targets are presented.

1 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-proof nature. Indeed, they usually require only a small area on the device, 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, any attempt 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: the instability of several responses to the same challenge over time. 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, 2, 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 require significant area overhead on the IC. This is contrasted with the lightweight nature of PUFs, and prevents large adoption by industry.

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 important 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 specifically focuses on correcting errors only.

Over existing error-correcting codes used for PUFs, the `CASCADE` protocol has two main advantages. Since only parity computations are done on the circuit, the area overhead is very limited. We propose two implementations that balance area overhead and latency. As a second advantage, `CASCADE` is greatly parameterisable and can accommodate various error-rate and failure rates. The parameters can be dynamically changed after the circuit has been built. This makes it a very good candidate for integration alongside PUFs.

1.1 Notations

PUF responses are r, of size n. The reference response, obtained during enrolment, is written r_0. The response obtained later on, which contains errors with an error rate ε, is r_t. Bit found at index i of the response is written $r[i]$. 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 written σ_i.

1.2 Overview

The rest of this paper is organised as follows. Section 3 presents PUFs, error-correcting codes and key reconciliation protocols. Section 4 explains how key reconciliation protocols can be adapted to correct errors in PUF responses. Section 5 gives the results of implementation on various FPGA targets. Finally, Section 6 discusses other aspects such as secret key generation, implementation variations and security.

1.3 Reproducibility

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

2 Motivation

2.1 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 semi-conductors market, and provide *fabless* designers with manufacturing facilities [7]. However, in order to have their designs manufactured, integrated circuits 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 integrated circuits [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 loses are worth hundreds of billions of dollars [11]. The target audience of this work are then designers of integrated circuits and intellectual property (IP) cores who wish to protect their designs against such threats.

There has been several propositions aiming at mitigating these threats [12, 13] 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. PUFs are great candidates for the generation of such identifiers.

2.2 Overall scheme

After is has been embedded in the integrated circuit, the unique device identifier is used to remotely identify the integrated circuit. Typically, integrated circuit 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.

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

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 basic protocol is shown in Figure 1.

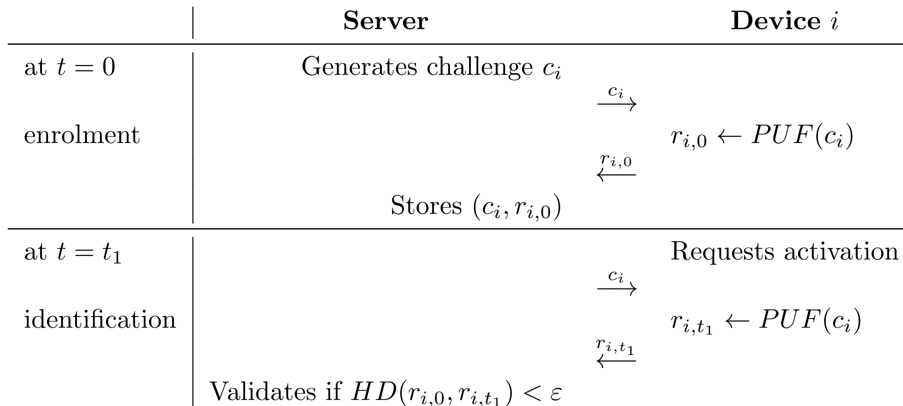

Figure 1: Basic protocol for integrated circuit remote identification using a PUF.

However, PUF responses are not perfectly stable and contain errors, so the maximum Hamming distance value ε can be hard to choose. Usually, error-correcting codes are associated to the PUF to correct those errors.

Key reconciliation protocols [5] can be used instead, in order to make identical the 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 [14] 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 integrated circuit and the server own an identical response, the integrated circuit 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 integrated circuit lifetime, when an activation request is issued. The adapted basic protocol is depicted in Figure 2.

In order to implement the previously described protocol for intellectual property protection, an activation module must be added to the integrated circuit. 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, the overhead is directly related to an increase in manufacturing costs for the designer. This increase in cost cannot be greater than the losses due to counterfeiting. The activation module must then be as lightweight as possible.

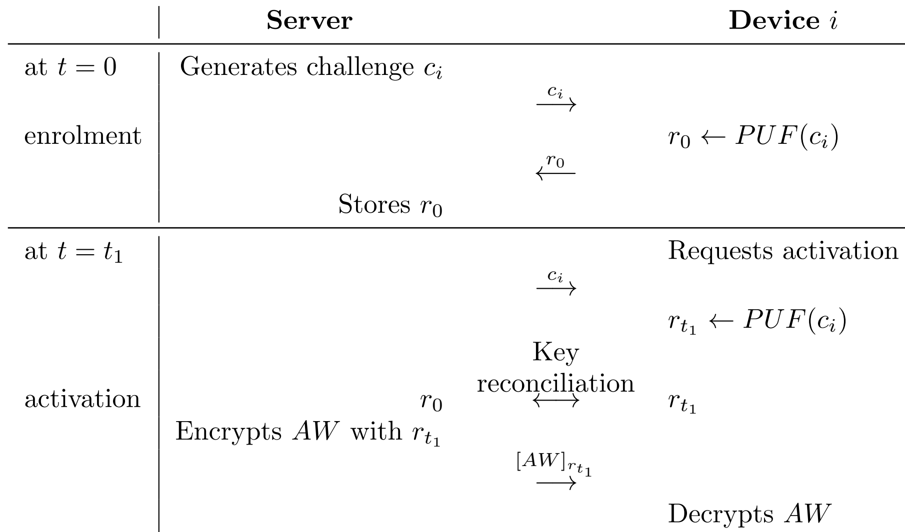

Figure 2: Overview of a typical protocol for integrated circuit remote activation using a key reconciliation protocol.

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 [15], 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.

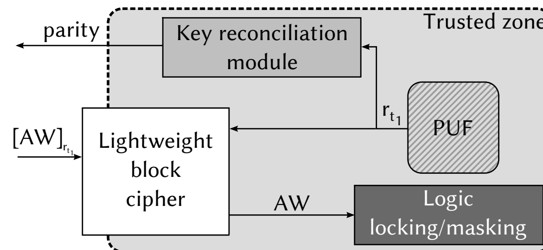

Figure 3: Activation module added to the integrated circuit.

The logic resources required 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 5. 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.

3 Related work

3.1 Physical Unclonable Functions

Physical Unclonable Functions (PUFs) are hardware primitives which 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 integrated circuits. Indeed, since the response results from random process variations, every integrated circuit embedding a PUF will generate a different response to the same challenge.

In 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 focuses only on error-correction.

3.2 Error-correcting codes for PUFs

An overview of helper data algorithms for PUF-based key generation has recently been published [16]. 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 [16], 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 code-offset or syndrome construction. Different types of codes are used, including repetition, BCH, Reed-Muller or Golay codes. A comparison is shown in Table 1, 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 will be 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 1, 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 last row in Table 1 gives the logic resources required by the key reconciliation protocol presented. Compared to the smallest error correcting codes [6, 17, 18], it requires almost three times fewer resources.

Table 1: 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.

Article	Construction and code(s)	Logic resources (Slices) Spartan 3	Spartan 6	Failure rate	Acceptable error-rate	PUF bits required for 128-bit entropy
[1]	Concatenated: Repetition (7, 1, 3) and BCH (318, 174, 17)		221	10^{-9}	13%	2226
[2]	Complementary IBS with Reed-Muller (2, 6)	250		10^{-6}	15%	>1536 (12×128)
[3]	Reed-Muller (4, 7)		179	1.48×10^{-9}	14%	130
[6]	BCH (255, 21, 55)		>59	$10^{-6.97}$	21.6%	1785
[17]	Reed-Muller (2, 6)	164		10^{-6}	15%	1536 (12×128)
[18]	Concatenated: Repetition (5, 1, 5) and Reed-Muller (1, 6)	168 (41+ 127)		1.49×10^{-6}	15%	4640
[18]	Concatenated Repetition (11, 1, 11) Golay $G_{24}(24, 13, 7)$	570 (41+ 539)		5.41×10^{-7}	15%	3696
This work	`CASCADE` protocol	**67**	**19**	*tunable* $10^{-4} \rightarrow 10^{-6}$	*tunable* 0.5% → 15%	*Depends on the failure and error rates* 256 → 2048

An interesting approach called *reverse fuzzy extractor* is presented in [6]. 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.

3.3 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

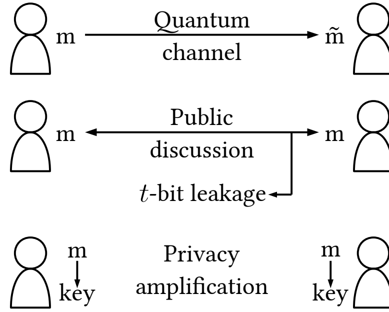

Figure 4: Key reconciliation protocol.

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) .

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

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.

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.

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 × 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}, ..., σ_{n_passes}^{-1}
return r_0, r_t
```

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.

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 `BINARY` 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. 1. Our arguments are detailed in the following section.

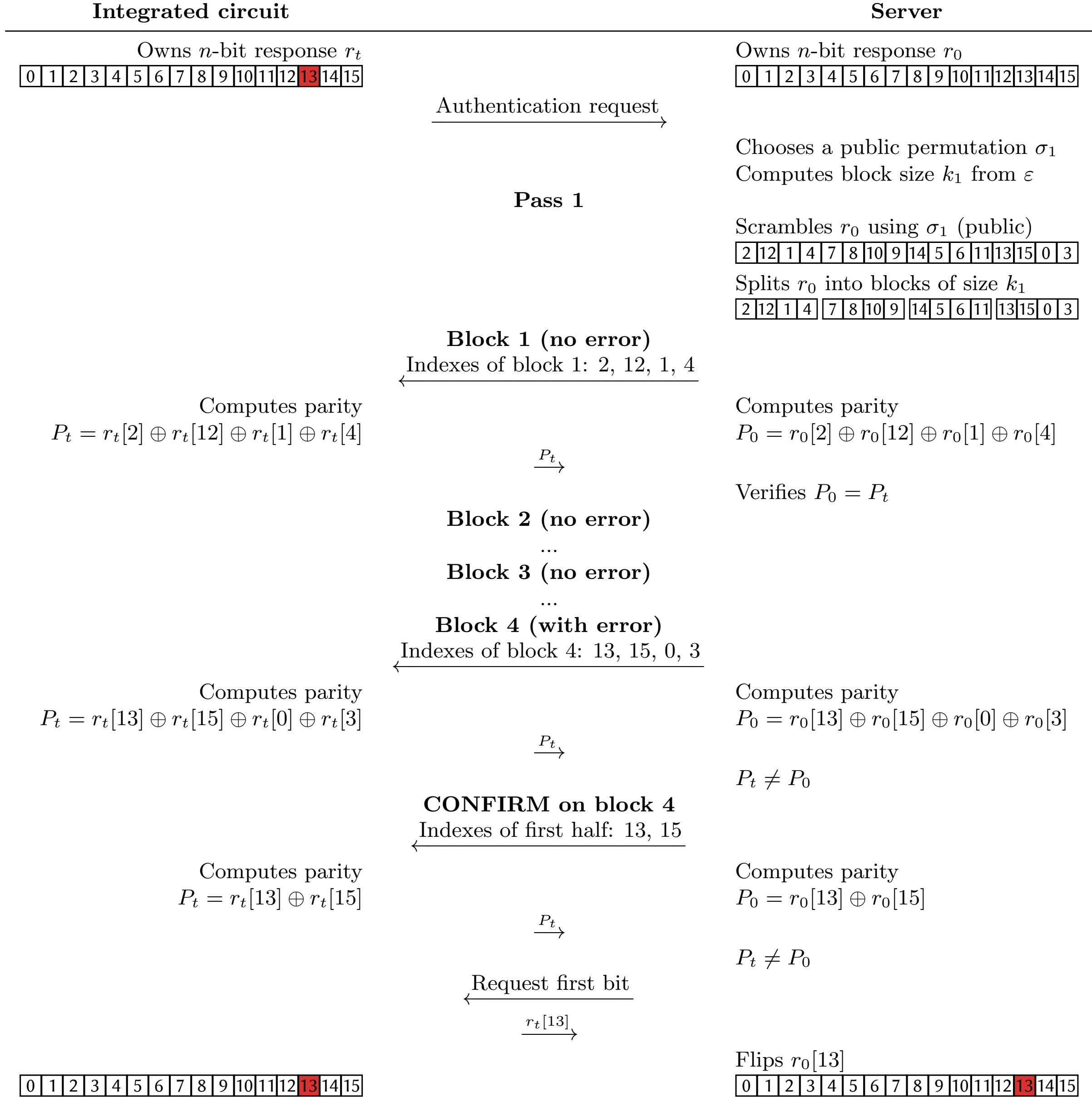

Figure 5: Toy-example of executing the BINARY protocol on 16-bit responses with one error.

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 × 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}, ..., σ_{n_passes}^{-1}
return r_0, r_t
```

4 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.

4.1 Protocol parameters

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

4.1.1 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 [19] tend to increase the initial block size up to $1/\varepsilon$. Moreover, [19] states that the block size should be a power of two to

achieve to the best reconciliation efficiency. This is emphasised in [20], in which the initial block size is given in Equation (2):

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

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.

4.1.2 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 [19, 20] 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.

4.1.3 Block size multiplier

In [19, 21], it is said that, from one pass to the next one, the block size could be multiplied by another factor than 2. Values ranging from 2 to 5 are explored for the block size multiplier. In our case, we chose to use multipliers that are powers of two. This led to the block sizes shown in Table 2.

Table 2: 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

4.1.4 Example

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. 6.4.

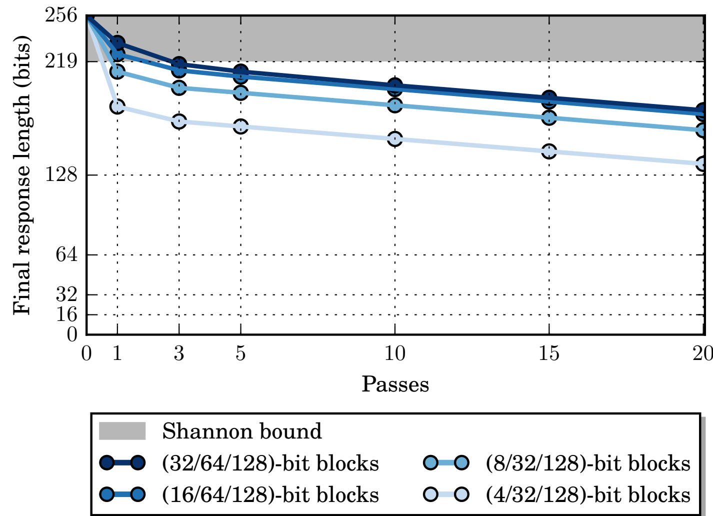

Figure 6: Final response length after executing the `CASCADE` protocol on a 256-bit response with different numbers of passes and initial block sizes. Here, the error rate is 2%.

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.

4.1.5 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

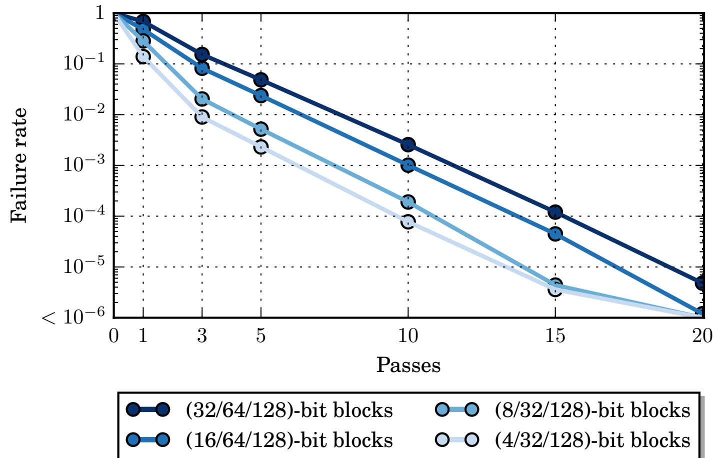

Figure 7: Failure rate for a 256-bit response with different numbers of passes and initial block sizes. Here, the error rate is 2%.

the leakage. They give the number of bits that can be kept secret. If the number of 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 3 and 4 show which parameters can be chosen for the `CASCADE` protocol in real-life examples to achieve a failure rate of 10^{-4} and 10^{-6} respectively 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. Two failure rates, 10^{-4} and 10^{-6}, are considered.

Table 3: Parameters chosen to achieve a failure-rate of 10^{-4} for different PUF architectures, aiming for at least 128 bits of final entropy.

PUF	Article	Target	Technology node	Error rate ε	k_1 [bits]	n_{passes}	Number of PUF bits required
RO	[22]	FPGA	90 nm	0.9%	32	15	256
	[23]	ASIC	65 nm	2.8%	32	20	256
TERO	[24]	FPGA	90 nm	1.7%	128	20	256
	[25]	FPGA	28 nm	1.8%	128	20	256
	[26]	ASIC	350 nm	0.6%	32	15	256
SRAM	[27]	FPGA	—	4%	64	15	512
	[28]	FPGA	—	10%	8	15	512
	[29]	ASIC	65 nm	5.5%	32	20	512

Table 4: Parameters chosen to achieve a failure-rate of 10^{-6} for different PUF architectures, aiming for at least 128 bits of final entropy.

PUF	Article	Target	Technology node	Error rate ε	k_1 [bits]	n_{passes}	Number of PUF bits required
RO	[22]	FPGA	90 nm	0.9%	64	25	256
	[23]	ASIC	65 nm	2.8%	32	15	512
TERO	[24]	FPGA	90 nm	1.7%	32	25	256
	[25]	FPGA	28 nm	1.8%	32	25	256
	[26]	ASIC	350 nm	0.6%	64	25	256
SRAM	[27]	FPGA	—	4%	16	25	512
	[28]	FPGA	—	10%	8	25	512
	[29]	ASIC	65 nm	5.5%	8	20	512

4.2 Leakage estimation

As highlighted in [19], 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 response and $h(\varepsilon)$ is the Shannon entropy.

$$nh(\varepsilon) = n.(-\varepsilon log_2 \varepsilon - (1-\varepsilon) log_2(1-\varepsilon)) \tag{3}$$

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 [30]. 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 need to be carefully chosen.

4.3 Implementation

This section presents the hardware implementation of the `CASCADE` protocol on the device side and its software implementation on the server side. We assume that the server has high computational capabilities, whereas the implementation should be as lightweight as possible on the device.

4.3.1 Device side

The only computation that needs to be carried out on the device is the parity computation. It must be done on blocks of variable length. This is simply achieved by multiplexing the PUF response bits to an XOR gate one after the

other. After the XOR gate, the intermediate result is sampled by a D flip-flop. This is illustrated by the diagram in Figure 8.

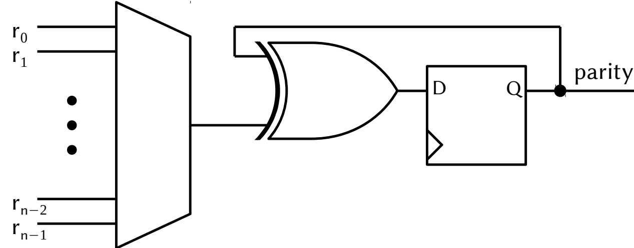

Figure 8: Parity computation module on the device-side.

4.3.2 Server side

All the other computations are handled by the server. 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. Table 5 summarises how computations are distributed between the device and the server.

Table 5: Distribution of features between device and server.

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

5 Implementation results

For the device side implementation of `CASCADE`, we used different target FPGAs to provide the most accurate comparison possible. We integrated the module computing the parity in a simple controller with three states: *idle*, *compute_parity* and *send_parity*. On the server side, we used Python for development.

5.1 Logic resources

Table 6 and 7 shows the implementation results in terms of LUTs, D flip-flops and Slices/ALMs/LCs[2]. As already shown in Table 1, the hardware implementation

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

of CASCADE is very lightweight. It is at least two to three times smaller than state-of-the-art error-correcting codes. Again, a 256-bit response is assumed here. Increasing the initial response size would require a larger multiplexer, and hence more logic resources.

Table 6: Implementation on different target FPGAs for the parity computation module on a 256-bit response.

Target device	#LUTs	#DFFs	#Slices/ALMs/ LCs
Xilinx Spartan 3 [a]	132	1	67 Slices
Xilinx Spartan 6 [b]	68	1	19 Slices
Altera Cyclone III [a]	170	1	171 LCs
Altera Cyclone V [c]	86	1	56 ALMs

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

Table 7: Implementation on different target FPGAs for the parity computation module on a 512-bit response.

Target device	#LUTs	#DFFs	#Slices/ALMs/ LCs
Xilinx Spartan 3 [a]	264	1	133 Slices
Xilinx Spartan 6 [b]	170	1	95 Slices
Altera Cyclone III [a]	340	1	342 LCs
Altera Cyclone V [c]	170	1	109 ALMs

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

A more device-specific FPGA implementation could also be achieved by instantiating a dedicated vendor primitive for the multiplexer.

5.2 Latency

Another criterion used to evaluate the different error correcting codes is their latency. For the CASCADE protocol, we distinguish two border cases. The actual latency of one execution of the protocol lies between those two cases.

The latency 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. Computing the parities for all the blocks of an n-bit response requires n clock cycles with the module shown in Figure 8. The number of clock cycles required for parity computations when executing the

CONFIRM method on a t-bit block is given by Equation (4). This corresponds to computing parities on blocks of sizes starting at $t/2$ down to 1.

$$\sum_{i=1}^{log_2(t)} \frac{t}{2^i} = t - 1 \tag{4}$$

Let us consider the previous case of a 256-bit response and a 2% error-rate. On average, five bits are flipping. We assume that the protocol starts with 32-bit blocks and runs for 15 passes.

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 latency 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 latency is:

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

Table 8 gives a comparison with existing error-correcting codes in terms of latency. 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. This leads to a low latency of 3933 clock cycles, comparable to the fastest existing codes. In the worst case, we considered a 512-bit response, in which the errors were corrected as late as possible. This gives a much higher latency of 33280 clock cycles, which is still much lower than [3].

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

The logic function is very simple here, since it consists only of one XOR gate and a D flip-flop, as shown in Figure 8. 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.

However, due to the great interactivity of the CASCADE protocol, the main latency bottleneck is the communication between the device and the serve. It can be order of magnitude slower that intra-device communication. Thus the associated latency depends to a great extent on the target platform. However, recent devices usually embed very high speed communication channels.

Table 8: Latency in clock cycles of different codes with different constructions.

Article	Construction and code(s)	Latency
[1]	Concatenated: Repetition (7, 1, 3) and BCH (318, 174, 17)	5831
[2]	Complementary IBS with Reed-Muller (2, 6)	—
[3]	Reed-Muller (4, 7)	108000
[6]	BCH (255, 21, 55)	—
[17]	Reed-Muller (2, 6)	1248
[18]	Concatenated: Repetition (5, 1, 5) and Reed-Muller (1, 6)	6505
[18]	Concatenated Repetition (11, 1, 11) Golay G_{24}(24, 13, 7)	1210
`CASCADE`	on 256-bit responses and $\varepsilon = 1\%$, 15 passes, starting with 32-bit blocks (errors corrected as early as possible)	3933
`CASCADE`	on 512-bit responses and $\varepsilon = 10\%$, 25 passes, starting with 8-bit blocks (errors corrected as late as possible)	33280

5.3 Alternative device-side implementation

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 `CASCADE` protocol implementation. However, it comes at the expense of increasing the protocol latency.

The architecture of the parity computation module is detailed in Figure 9. The bottom left D flip-flop samples the response bit once it is available at the shift register output. 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.

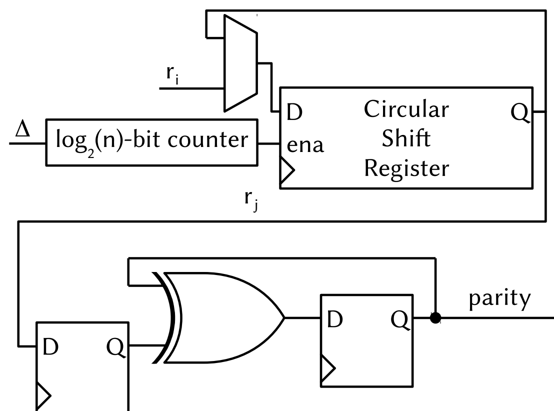

Figure 9: Alternative architecture for the parity computation module on the device-side.

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 $r[j]$. The response $r[j]$ is then the response bit to be selected after $r[i]$. Two cases can occur when selecting these response bits.

- 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.

The associated logic resources overhead is very low. Table 9 gives the implementation results on the four types of FPGAs previously considered with the same metrics: number of LUTs, number of D flip-flops and number of Slices/ALMs/LCs. We considered a 256-bit response here.

As seen in Table 9, the logic resources overhead drops significantly when an existing shift register can be reused. Depending on the type of PUF, this alternative implementation is an interesting solution. However, this comes at the price of increased latency.

5.3.1 Associated latency

With the original device-side implementation shown in Figure 8, one response bit can be selected for parity computation every clock cycle. Indeed, in order to

Table 9: Implementation on different target FPGAs for the alternative parity computation module on a 256-bit response.

Target device	#LUTs	#DFFs	#Slices/ALMs/ LCs
Xilinx Spartan 3 [a]	26	12	17 Slices
Xilinx Spartan 6 [b]	17	12	7 Slices
Altera Cyclone III [a]	25	20	26 LCs
Altera Cyclone V [c]	23	20	15 ALMs

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

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, 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 for the original parity computation module, for which only n clock cycles are necessary.

The other step which increases in latency 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 (6).

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

Like previously, we need to consider the best and worst case scenario for when the `CONFIRM` method is applied. 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 latency 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 latency is:

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

After comparing with the numbers given in Table 8, it is clear that the latency here exceeds by far the one of existing error-correcting codes. However, the logic resources overhead is also greatly reduced. This trade-off provides the designer with the ability to chose the most suited solution for the target application.

6 Discussion

6.1 Privacy amplification

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. The next step is thus to shorten the response in order to get all the bits with maximum entropy. This is called *privacy amplification*, and can be achieved by using a universal hash function, thanks to the leftover hash lemma [31]. In practise, a hash function is used for this purpose. Figure 10 shows how the number of bits varies at different steps.

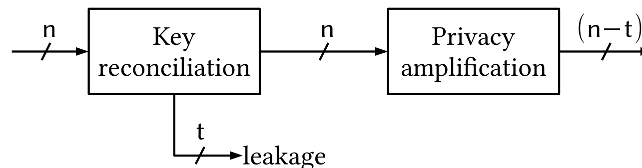

Figure 10: Changes in the number of bits in the response at different steps.

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 [32] was used in [1], and requires only 22 Slices on a Xilinx Spartan 6 FPGA for the 128-bit output block option. Another alternative is to implement Toeplitz hashing [33], which was chosen in [34]. Based on an LFSR, this construction occupies 59 Slices on a Xilinx Spartan 3 FPGA. Other low-area oriented hash functions can be found on the SHA-3 competition webpage, in the "low-area implementations" section[3].

6.2 Replacing parity check with hashing

Some works [35, 36] 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. Hence it does not suit our use case.

For example, if the the `CASCADE` protocol starts with 8-bit blocks, then it is easy for an attacker to 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.

6.3 Security analysis

6.3.1 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

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

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.

6.3.2 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.

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 (7), where X is the number of bits modified by executing the `CASCADE` protocol.

$$m : P(X = m) < f \tag{7}$$

Over this limit, the probability that an attacker is trying to modify the response on server-side is higher than the failure rate of the protocol. Therefore, further modifications are not allowed and the protocol is stopped. This allows to prevent device impersonation.

6.3.3 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 integrated circuit 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, in order 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 6.4, demonstrates another way to avoid server impersonation.

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 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. This however, can be circumvented by resetting the device and executing the `CASCADE` protocol multiple times, to obtain more linear equations. Therefore, we propose an additional counter-measure to address this problem.

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 equivalent 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 [37]. Learning parities with noise is equivalent in complexity to decoding from a random linear code [38], which is known to be an NP-hard problem.

6.3.4 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 has been exceeded. 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 this way.

6.3.5 Helper data manipulation

Recent works highlight the fact that helper data can be manipulated [39]. Since `CASCADE` only requires exchanging simple parity values, manipulation is not a threat and is handled like impersonation.

6.4 Deterministic scrambling

The first step of the key reconciliation protocol is scrambling. The aim is to spread the errors evenly over the blocks. However, 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 [40]. At that point, the chosen permutation can assign one unstable bit per response block, so that the errors on those bits are easily detected and corrected in the first passes of the 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.

7 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 overhead compared to existing error correcting codes, its main advantage is requiring very low area overhead on the device and small latency for the computations. Another advantage of this solution is its great flexibility. Parameters can be very easily tuned, and adapted to the constraints of the application. 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 integrated circuits.

Acknowledgements

The work presented in this paper was realized 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*".

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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-33027-8_18

[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. [Online]. Available: http://dx.doi.org/10.1109/HST.2012.6224310

[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. [Online]. Available: http://dx.doi.org/10.1109/ISVLSI.2015.31

[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. [Online]. Available: http://dx.doi.org/10.1007/BF00191318

[5] G. Brassard and L. Salvail, "Secret-key reconciliation by public discussion," in *EUROCRYPT*, Lofthus, Norway, May 1993, pp. 410–423. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.42.9686&rep=rep1&type=pdf

[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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32946-3_27

[7] D. A. Hodges, "Building the fabless/foundry business model," *IEEE Solid-State Circuits Magazine*, vol. 3, no. 4, pp. 7–44, 2011. [Online]. Available: http://ieeexplore.ieee.org/ielx5/4563670/6069769/06069787.pdf?arnumber=6069787&origin=publication_detail

[8] C. Gorman, "Counterfeit chips on the rise," *IEEE Spectrum*, vol. 49, no. 6, pp. 16–17, 2012. [Online]. Available: http://spectrum.ieee.org/computing/hardware/counterfeit-chips-on-the-rise

[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. [Online]. Available: http://dx.doi.org/10.1109/JPROC.2014.2332291

[10] AGMA, "Managing the risks of counterfeiting in the information technology industry," Alliance for Gray Market and Counterfeit Abatement, Tech. Rep., 2005. [Online]. Available: http://www.agmaglobal.org/press_events/press_docs/Counterfeit_WhitePaper_Final.pdf

[11] IHS Technology. (2012, Apr.) Top 5 most counterfeited parts represent a $169 billion potential challenge for global semiconductor market. [Online]. Available: https://technology.ihs.com/405654/top-5-most-counterfeited-parts-represent-a-169-billion-potential-challenge-for-global-semiconductor-market

[12] U. Guin, D. Forte, and M. Tehranipoor, "Anti-counterfeit techniques: from design to resign," in *IEEE Microprocessor Test & Verification*, Austin TX, USA, Dec. 2013. [Online]. Available: http://www.engr.uconn.edu/~tehrani/publications/mtv13.pdf

[13] 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. [Online]. Available: http://dx.doi.org/10.1049/iet-cdt.2014.0028

[14] B. Colombier, L. Bossuet, and D. Hély, "From secured logic to IP protection," *Elsevier Microprocessors and Microsystems*, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0141933116000417

[15] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74735-2_31

[16] 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. [Online]. Available: http://dx.doi.org/10.1109/TCAD.2014.2370531

[17] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-04138-9_24

[18] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85053-3_12

[19] 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. [Online]. Available: http://arxiv.org/abs/1407.3257v2

[20] 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. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7282551

[21] H. Yan, T. Ren, X. Peng, X. Lin, W. Jiang, T. Liu, and H. Guo, "Information reconciliation protocol in quantum key distribution system," in *International Conference on Natural Computation*, vol. 3, Jinan, China, Oct. 2008, pp. 637–641.

[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. [Online]. Available: http://dx.doi.org/10.1109/HST.2010.5513108

[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. [Online]. Available: http://dx.doi.org/10.1109/ESSCIRC.2012.6341361

[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. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TETC.2013.2287182

[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. [Online]. Available: http://doi.acm.org/10.1145/2847263.2847298

[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. [Online]. Available: http://eprint.iacr.org/2015/623

[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. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74735-2_5

[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. [Online]. Available: http://dx.doi.org/10.1007/978-3-662-48324-4_28

[29] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-29615-4_5

[30] R. L.-Y. Ng, "A probabilistic analysis of CASCADE," in *International conference on quantum cryptography*, 2014. [Online]. Available: http://home.uchicago.edu/~ruthfrancisng/cascade.pdf

[31] R. Impagliazzo, L. A. Levin, and M. Luby, "Pseudo-random generation from one-way functions," in *Annual Symposium on Theory of Computing*, Seattle, Washington, USA, May 1989, pp. 12–24. [Online]. Available: http://doi.acm.org/10.1145/73007.73009

[32] 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. [Online]. Available: http://link.springer.com/chapter/10.1007%2F978-3-642-23951-9_21#

[33] H. Krawczyk, "LFSR-based hashing and authentication," in *Annual International Cryptology Conference*, vol. 839, Santa Barbara, California, USA, Aug. 1994, pp. 129–139. [Online]. Available: http://dx.doi.org/10.1007/3-540-48658-5_15

[34] 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. [Online]. Available: http://www.cosic.esat.kuleuven.be/publications/article-1251.pdf

[35] 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. [Online]. Available: http://dx.doi.org/10.1137/0217014

[36] 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. [Online]. Available: http://dx.doi.org/10.1007/3-540-47719-5_22

[37] 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, January 2012, pp. 99–114. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-27660-6_9

[38] 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. [Online]. Available: http://dx.doi.org/10.1109/TIT.1978.1055873

[39] 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. [Online]. Available: http://dx.doi.org/10.7873/DATE.2014.085

[40] 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. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40349-1_5