Journal of Cryptographic Engineering (2020) 10:229–247
https://doi.org/10.1007/s13389-020-00223-w

REGULAR PAPER

Review of error correction for PUFs and evaluation on state-of-the-art FPGAs

Matthias Hiller[1] · **Ludwig Kürzinger**[2] · **Georg Sigl**[1,3]

Received: 30 April 2018 / Accepted: 22 March 2020 / Published online: 11 May 2020
© The Author(s) 2020

Abstract

Efficient error correction and key derivation is a prerequisite to generate secure and reliable keys from PUFs. The most common methods can be divided into linear schemes and pointer-based schemes. This work compares the performance of several previous designs on an algorithmic level concerning the required number of PUF response bits, helper data bits, number of clock cycles, and FPGA slices for two scenarios. One targets the widely used key error probability of 10^{-6}, while the other one requires a key error probability of 10^{-9}. In addition, we provide a wide span of new implementation results on state-of-the-art Xilinx FPGAs and set them in context to old synthesis results on legacy FPGAs.

Keywords Physical unclonable functions · Key generation · Fuzzy extractor · FPGA · Hardware implementation

1 Introduction

Over the last fifteen years, physical unclonable functions (PUFs) went a long way from an idea in fundamental research, e.g., [1–5], to commercial products that are deployed in a large scale by major semiconductor companies, e.g., [6–8]. PUFs are a promising replacement for secure key storage technologies in CMOS circuits in general, and for security applications in lightweight embedded devices in particular that do not have access to dedicated secure on-chip memory. While emerging nonvolatile memory technologies such as magnetic or resistive RAM [9] potentially provide new approaches to secure nonvolatile storage, PUFs have the advantage that they can also be built upon default semiconductor technologies like CMOS or implemented on FPGA fabric.

Silicon PUFs evaluate manufacturing variation inside a circuit to derive internal secrets [10,11]. As in any physical measurement, we have to address two effects: deterministic changes of the measurement and noise. The deterministic change can be caused by environmental effects on the measured sample and the measuring equipment, as well as aging. This can also change the distribution of the noise. As we focus on the error correction in this work, we will abstract from the peculiarities of specific PUF implementations and assume a random variable as input with a given distribution of the bit error rate of the individual PUF bits.

In the next step, algorithms for key derivation and error correction generate reliable and unpredictable keys from the PUF data. This is achieved by storing helper data that maps random PUF responses to codewords of an error-correcting code (ECC) where errors can be detected and corrected. However, this helper data has to be constructed in a way that it does not reveal significant information about the derived secret.

In the following, we will introduce the criteria that will be applied to assess the implementation efficiency of the compared approaches. Figure 1 shows the three main criteria under the assumption that the target security and reliability

L. Kürzinger: The majority of this work was carried out while the author was with Fraunhofer AISEC.

✉ Matthias Hiller
matthias.hiller@aisec.fraunhofer.de

Ludwig Kürzinger
ludwig.kuerzinger@tum.de

Georg Sigl
sigl@tum.de

[1] Fraunhofer Institute for Applied and Integrated Security AISEC, Lichtenbergstraße 11, 85748 Garching (near Munich), Germany

[2] Chair of Human-Machine Communication, Technical University of Munich, Arcisstraße 21, 80333 Munich, Germany

[3] Chair of Security in Information Technology, Technical University of Munich, Arcisstraße 21, 80333 Munich, Germany

Fig. 1 Evaluation criteria for secure key derivation with PUFs

level of the cryptographic key are specified by the system or the cryptographic application in which the key is processed.

We have a classical time-area trade-off with the helper data as a third criterion. A next step is to assign an application-specific cost function and define hard outer constraints that have to be met by a design. This is highly application-dependent so that we will assess the different criteria, but have to leave the evaluation of the cost function and thus the selection of specific solutions to the designer in the application.

Chip area For a silicon PUF, the implementation cost of the PUF and the post-processing are subsumed together as chip area, or FPGA slices. Depending on the available PUF types, the required area and also the quality of the produced PUF bits can vary, so that cost for a required PUF bit have to be weighted with its individual area requirement.

This leads directly to the trade-off between the number of PUF bits and the algorithmic complexity of the post-processing.

Helper data The helper data has to be stored permanently in unsecured storage, and the cost of this storage can vary greatly depending on the available technology. On-chip NVM is relatively expensive and inflexible and has to be considered at design time. In contrast, storing helper data on a remote server comes almost for free. In addition, the integrity of the helper data has an impact on potential helper data manipulation or side-channel attacks, e.g., [12].

Run time As third criterion, there can be very strict or up to virtually no real timing constraints, depending on the application. For example, assuming that a cryptographic key can be precomputed before the cryptographic algorithm runs does not pose strict timing constraints. Again, the run time of the PUF and the post processing have to be considered together. In particular, for slow PUFs like the RO PUF, it might also be relevant to consider pipelined approaches where the post-processing can start the error correction even if not all PUF bits are ready yet. There is typically a trade-off between area and time, where this work optimizes for area.

An additional constraint that has to be taken into account is the entropy of the PUF bits, that can be impaired, e.g., by bias or correlation [13]. If available, additional reliability information can improve the error correction notably.

Over the last years, several approaches were designed and implemented. Implementations based on linear schemes such as the code-offset fuzzy extractor [14–17], the syndrome construction [14,18] or systematic low leakage coding (SLLC) [19] operate on algebraic operations. In contrast, pointer-based schemes such as index-based syndrome coding (IBS) [20,21], differential sequence coding (DSC) [22–24] or the maximum-likelihood approach in [25] break up the linear structure by referencing specific PUF response bits or chunks of multiple bits. Later on, more complex code classes and concatenation schemes were introduced to further increase the efficiency of the post-processing, e.g., [26–28]. More details on the different approaches can also be found in the corresponding dissertations [10,29–33].

This work gives a broad overview over state-of-the-art algorithms and their hardware implementations, with a main focusing on FPGA implementations. Most previous work designed the error correction to meet key error probabilities of either 10^{-6} or 10^{-9}. To cover a wide span of possible underlying PUFs, we analyze schemes for average input bit error probabilities of 5%, 10%, 15%, 20%, and for the 10^{-6} scenario also 25%.

To give an overview over existing work and keep the results comparable, we present results for a secret key length of 128 bit. Physical attacks such as side-channel attacks [12,34–36] and invasive attacks [37–39] are not considered in this work.

Reusability [40,41] is another interesting and important aspect which is still lacking practical implementations so that we will not further address it in the following.
Four main criteria are applied for evaluating the different approaches:

– The number of PUF response bits directly contributes to the chip area or number of slices that have to be assigned for the key generation module and thus directly translates into cost. Some PUFs directly provide reliability information, while for others, it has to be obtained from multiple measurements.
– The size of the helper data that is stored to enable error correction. The cost for helper data bits is harder to generalize. If the helper data is transmitted from an external source and time is not a constraint, the helper data size is not a significant cost factor. In contrast, if the helper data is stored on the device in a memory of limited size, the cost can be significantly higher.
– The execution time in clock cycles is the third criterion. If the PUF is not in a critical path it can almost be neglected,

while it becomes a hard constraint if a key cannot be precomputed.
– The area occupied on an FPGA, given in slices or CLBs. Typically, various trade-offs between time and area are possible in digital design. In this work, we emphasize the size of the decoder and therefore put only a small weight on the execution time, as long as it does not approaches millions of clock cycles.

A variety of external influences poses significant variation on the outer constraints and limitations so that the results of this comparison can only provide a first orientation for a practical implementation. The results still hold but the implications have to be reassessed, e.g., depending on the constraints on the PUF, the available area and time, provisioning constraints, or the environment. Adding intermediate steps and quantization such as in [3,42–45] fundamentally changes properties again, which makes it even harder to provide general results.

So far, most reference approaches were implemented for 90 nm Xilinx Spartan 3 and 45 nm Xilinx Spartan 6 FPGAs. This work provides additional synthesis results for 28 nm, 20 nm and 16 nm FPGAs to provide up-to-date information for designs on recent FPGAs.

A recent focus was set on biased PUF responses [46–48]. As the ECC decoder implementations cover large parts of the algorithmic complexity of the approaches, the ECC implementation results for similar approaches without debiasing can also be used to estimate the implementation complexities of future work operating on similar codes.

While most work using ECC focuses on secret key generation, similar approaches can also be used for authentication [49]. Comprehensive analyses of PUF-based authentication can be found, e.g., in [30,50].

Our main contributions are

– Comprehensive listing and discussion of the current error correction approaches for unbiased PUFs for key lengths of 128 bit
– Comparison of the state of the art approaches in two reference scenarios with output key error probabilities of 10^{-6} and 10^{-9} and average input bit error probabilities between 5% and 25%
– Comparison of the designs on legacy Xilinx FPGAs and new synthesis results for state-of-the-art FPGAs

Outline:

Section 2 gives an overview over key derivation and error correction schemes for PUFs. Synthesis results for two reference scenarios are discussed in Sect. 3. In addition, new synthesis results are provided in Sect. 4. Design rationals for choosing a code are given in Sect. 5. Section 6 concludes this work. Details on the BCH decoder implementation can be found in the “Appendix”.

2 Error correction for PUFs

This section gives an overview over the state-of-the-art error correction schemes for PUFs. Overviews that discuss various aspects of error correction can be found in, e.g., in [10,11,13,30,51].

We divide the approaches in two different classes of schemes: Linear schemes are presented in Sect. 2.2, followed by the pointer-based schemes in Sect. 2.3. Section 2.4 briefly discusses different types of ECC implementations for PUFs.

If a PUF does not provide a sufficient reliability, the response can be reduced to a shorter sequence by removing unreliable PUF bits through so-called dark bit masking [52]. This reduces the information theoretically achievable performance of the system and trades it for a decreased practical decoder complexity. Taking the required area for the PUF and the error correction into account, this can be an approach, designers are willing to take.

There is also a wide body of work that discusses how to efficiently encode a physical PUF measurement to generate a binary sequence, addressing mainly reliability and entropy of the derived sequence, e.g., [3,43,44,53,54]. As this work focuses on the helper data generation and ECC parts, all bit mapping aspects are subsumed under the PUF and the output is interpreted as PUF response in the following. Incorporating the helper data into a specific ECC for each PUF, as it was shown in [55] for LDPC codes, would further increase the efficiency but is not possible for many use cases.

2.1 Notation

Functions are denoted by small letters, e.g., $f(\cdot)$. Line vectors are denoted by capital letters, e.g., X and matrices are provided in bold capital letters, e.g., $\mathbf{G}$.

$\mathbf{I}$ is the identity matrix. $\mathbf{G}$ defines the generator matrix of a code with the special case $\mathbf{G} = [\mathbf{I}\ \mathbf{P}]$ for a linear code in systematic form [56]. $\mathbf{P}$ is the part of the generator matrix of a linear code in standard form which creates the redundancy while $\mathbf{H}$ is the parity check matrix of a code.

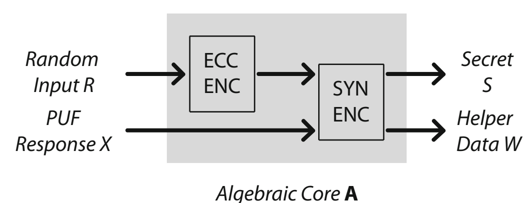

Fig. 2 Secret key and helper data generation with a PUF during enrollment

During generation, helper data W and secret S are computed from the PUF response X and an optional random number R, as shown in Fig. 2. In case of linear schemes, the computation of W and S can be summarized by an algebraic core $\mathbf{A}$ [57]. In the following, superscripts are added to indicate the lengths of the vectors.

2.2 Linear schemes

Several error correction schemes for PUFs can be subsumed under the label linear schemes. They all compute the helper data and the secret from linear operations on PUF response bits, as discussed in [57]. So far, all schemes are based on linear ECCs [56]. However, some approaches could also operate on nonlinear codes which was neither investigated nor implemented so far, to the best of our knowledge. In the following, we will use the parameter notation (n, k, d) for code length n, code size k and minimum distance d.

Fuzzy commitment In the fuzzy commitment [58] by Juels and Wattenberg, a random input R^k is stored with a modified error-correcting one-time pad such that it can be reproduced later from a noisy version of the original input. The codeword C^n of the ECC is computed according to $S^n = C^n = R^k\,\mathbf{G}$. Then, C^n and the PUF response X^n are XORed to compute helper data W^n.

$$S^n = R^k\,\mathbf{G} \tag{1}$$

$$W^n = \left(R^k\,\mathbf{G}\right) \oplus X^n \tag{2}$$

A slightly modified approach was discussed in [59] where the decoded random number R^k directly forms the secret instead of C^n.

$$S^k = R^k \tag{3}$$

$$W^n = \left(R^k\,\mathbf{G}\right) \oplus X^n \tag{4}$$

While the security of the original fuzzy commitment is proven in an information theoretical setting, the more recent computational fuzzy extractor [60] is based on a complexity theoretical argument. The hard problem of learning parity with noise [61] allows to create constructions without complexity theoretical leakage. The generator matrix $\mathbf{G}$ of the ECC, which is typically highly structured, is substituted by a random matrix. The decoding complexity increases significantly for higher noise levels. Therefore, a trapdoor is added in [62] to identify reliable PUF response bits in Y^n and mark unreliable ones as erasures. This reduces the decoding complexity on-chip, while the hardness of the problem remains unchanged for the attacker. Another implementation can be found in [63].

Code-offset fuzzy extractor The code-offset fuzzy extractor [14] is similar to the fuzzy commitment. While the fuzzy commitment derives the secret based on the random number R^k, the code-offset fuzzy extractor computes the secret from the PUF response X^n. However, this change causes secrecy leakage. Therefore, a hash function $f(\cdot)$ has to be added. The helper data computation is identical to the fuzzy commitment.

$$S^n = X^n \tag{5}$$

$$K^k = f\left(S^n\right) \tag{6}$$

$$W^n = \left(R^k\,\mathbf{G}\right) \oplus X^n \tag{7}$$

The code-offset fuzzy extractor was implemented several times: The implementation by Bösch et al. [15,64] is based on Reed–Muller, BCH and Golay Codes [56]. Later, Maes et al. discussed soft-decision decoding of Reed–Muller codes for an SRAM PUF on FPGAs [16,17]. Van der Leest et al. performed a single read-out during enrollment and also use Golay codes in their implementation [65]. More recently, constructions with generalized concatenated codes (GCC) were introduced by Müelich et al. for Reed–Muller [26] and Puchinger et al. for Reed–Solomon codes [27]. An implementation of the Reed–Muller construction can be found in [66].

Syndrome construction The syndrome construction discussed in [14] is based on [67] and requires linear codes. The syndrome of the PUF response is stored in the helper data so that no random number R^k is required. It is computed by multiplying the PUF response with the parity check matrix $\mathbf{H}$ of the code. Similar to the code-offset fuzzy extractor, the syndrome construction also leaks secret information so that a hash function is added again to compute a secure cryptographic key $K^k = f\left(S^n\right)$.

The AEGIS secure processor by Suh [68] uses a BCH code and the syndrome construction. Also the PUFKY implementation by Maes et al. [18] is based on a BCH code and the Syndrome Construction.

Parity construction The parity construction [69] computes the parity of the PUF response and stores it as helper data $W^{n-k} = X^k\mathbf{P}$. Again, PUF response X^k is hashed and output as key.

Please note that $\mathbf{P}$ is part of a generator matrix with $\mathbf{G} = (\mathbf{I}|\mathbf{P})$ of the code while PUF response is multiplied with the parity check matrix $\mathbf{H}$ in the syndrome construction. See, e.g., [29,57] for details. It was shown in [70] that this scheme shows a weaker performance compared to the other linear schemes. Therefore, it was not considered for

past implementations. However, the work in [54] built upon of this scheme as the other approaches are not suitable for the newly introduced insertion/deletion error patterns.

Systematic low leakage coding Systematic low leakage coding (SLLC) was introduced independently in [19,71] and also operates on codes with systematic encoding. Instead of storing the parity, SLLC masks the parity bits with fresh PUF bits to mitigate the helper data leakage. The secret and helper data are computed according to

$$S^k = X^k \tag{8}$$

$$W^{n-k} = X^k\mathbf{P} \oplus X_{k+1}^n \tag{9}$$

2.3 Pointer-based schemes

Pointer-based schemes replace the linear mapping through nonlinear indexing operations. This removes linear dependencies between the secret and the helper data and is beneficial for example for mitigating leakage from biased PUFs [46]. Considering reliability information at the input can give a bonus in efficiency at the expense that the enrollment requires extra effort if the PUF outputs only binary information.

Index-based syndrome coding Index-based syndrome coding (IBS) was introduced by Yu and Devadas in [20] and divides the PUF response into fixed-size blocks. One block is assigned to each bit of the codeword and IBS selects the PUF response bit that has the highest probability to be equal to the codeword bit. A pointer to this PUF response bit is stored in the helper data. Selecting reliable PUF response bits, IBS also reduces the average input error probability at the input of the ECC decoder. It was shown in [20] that IBS pointers do not leak information for i.i.d. PUF response bits.

The implementation in [72] evaluates an RO sum-PUF and introduces syndrome distribution shaping as countermeasure against machine learning attacks. An ASIC implementation of IBS is discussed [73]. An attack against IBS with non-i.i.d. PUF response bits can be found in [74].

Complementary index-based syndrome coding IBS only indexes a small fraction of the available PUF response while a large portion is discarded. Complementary IBS (C-IBS) [21] introduces an additional encoding step and encodes each codeword bit with a repetition-like code with equal Hamming weight for encoding zero and one. In the next step, each bit of the repetition code is stored as IBS pointer. A soft-decision Reed–Muller implementation was presented in [21].

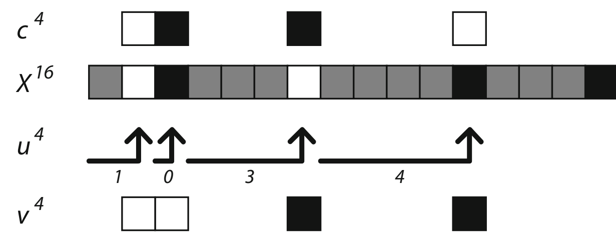

Fig. 3 Example for DSC encoding

Differential sequence coding All previous approaches have in common that compact implementations can only be achieved for small block sizes. Differential sequence coding (DSC) [24] operates on one large block that is searched sequentially for PUF response bits X that are reliable according to their individual reliability indicator $\mu(X)$. Each reliable PUF response bit points to one code sequence bit C^k. If the PUF response bit and the code sequence bit are identical, the inversion bit is set to zero, and otherwise to one. Figure 3 shows an example for DSC encoding.

The presented DSC implementation is concatenated with convolutional codes with parameters $(n, k, [m])$. k input streams are encoded to n output streams such that each output symbol is affected by the previous $[m]$ input symbols. So far, only $(2, 1, [m])$ codes were implemented in the PUF context.

Algorithmic descriptions of DSC can be found in [22,23]. The security of DSC was proven in [22] and the size of the helper data can be reduced effectively by compressing the pointers with run-length encoding (RLE) [23,75].

Maximum-likelihood symbol recovery While the previous three approaches compute pointers bit-wise, the ML symbol recovery by Yu et al. [25] computes indices to larger PUF response blocks. The PUF response is divided into multiple blocks and a part of the PUF response is selected depending on the secret.

During reproduction, the PUF response is divided into blocks again and then each block is compared to the helper data. The position of the block that is the most similar one to the helper data is output as secret. This approach shows a strong performance for high input bit error probabilities larger than 20%. Since the decoding complexity and the number of PUF bits increase exponentially with the number of embedded key bits, this approach is only suitable for low numbers of key bits per block.

2.4 Error-correcting code implementations

Typically, the ECC decoder is the most complex module in the post-processing of PUF responses such that it requires special effort in design and also promises large potential for optimization. Most communication use cases pose strict constraints on the latency and throughput of the ECC decoder. This is fundamentally different for the PUF case, where only one single key is reproduced such that the implementation size is a critical measure and time constraints are typically more relaxed.

Application-specific processors For a small implementation footprint, it is important to break down the complex decoding algorithm into small steps that can be executed sequentially. One strategy is to design a processor with an application-specific instruction set. The generic components can be reused several times, which leads to very compact implementations. The *Generalized Multiple Concatenated* (GMC) code decoder [76] for a Reed–Muller code in [17] and the BCH decoders in [18,77] and the "Appendix" were based on such a processor. Also the BCH decoder in this work is based on this design.

Register reduction A second design strategy is trying to minimize the number of registers, as a large number of registers also leads to a high slice count. There are two main objectives: The first is to reduce the number of intermediate registers by reusing them multiple times. The second strategy is to modify the algorithms such that the read and write operations match to access patterns of higher-density memory such as shift registers and RAM.

The RM decoder in [66] breaks down the Reed decoding [56] into small and regular operations using a ring buffer, and the concatenated codeword is stored in RAM. The Seesaw Viterbi decoder in [78] is also optimized for storing most information in RAM with a compact data path in between.

3 Evaluation

The previous sections presented several helper data generation schemes and ECCs that are designed for different scenarios. The scenario is defined by the characteristics of the PUF, implementation complexity constraints and the required output reliability of the key. Important criteria to evaluate an implementation for a given output error probability are the numbers for PUF bits, helper data bits, slices on the FPGA and clock cycles. In the following, we will compare designs and implementations for different average input bit error probabilities $\mu(p_{puf})$ for defined key output error probabilities. We assume bias-free SRAM PUF distributions discussed in [16] as underlying reliability distributions.

Designs are typically optimized for one use case with given input and output error probabilities. We discuss in Sect. 3.1 how the different implementations are compared and how the results were estimated if no dedicated results exist. The SLLC results discussed in [19] were generated for a completely different scenario with a significantly more reliable PUF [79]. Therefore, Sect. 3.2 puts the SLLC results into context with the other results.

In the following, we will analyze two scenarios: The results for a medium key output error probability of 10^{-6} are discussed in Sect. 3.3. Section 3.4 presents designs for a lower key error probability of 10^{-9}. The field started with the medium error probability and as it matured, also more designs were for the low error probability scenario were presented. Tables 1 and 2 show the precise numbers for the approaches analyzed in this work. We cover PUFs with average bit error probabilities $\mu(p_{puf})$ between 10^{-5} and 25% for the two scenarios.

3.1 Estimation of implementation complexity

To be able to present a wider range of results, numbers were extrapolated from reference implementations for additional parameter sets. Once a parameter set is chosen for an implementation, specific optimizations can be carried out to further improve the results.

For DSC, the results were obtained with the bounding technique presented in [22]. Only the 15% average input error probability and 10^{-6} output key error probability data point was obtained through simulation, as discussed in [24]. The compressed helper data sizes for DSC with run-length encoding were derived using m parameters as powers of 2 and the expected size. It was shown in [23] that only a small overhead is sufficient to achieve high yields. This overhead will not be considered in the following. $m = 2$ will be used to estimate the implementation sizes and cycle counts. Larger m values only cause a small implementation overhead. For results with convolutional codes, the (2, 1, [7]) Seesaw Viterbi decoder presented in [78] is used. For DSC, the clock cycle counts assume that a PUF response bit is read in two clock and helper data is written in a third. The $\mu(p_{puf}) = 15\%$ and 10^{-6} data point is taken as reference for the clock cycles and the other values are extrapolated from there.

We synthesized the BCH decoders with the area-optimized implementation discussed in the "Appendix". Note that the code parameters only have a minor influence on the size of the implementation but the number of clock cycles varies by orders of magnitude for increasing code length and code distance. The SPONGENT (128/128/8) implementation published in [80] has an estimated delay of 1, 120 clock cycles using 16 bytes with 70 rounds each and occupies 44

Table 1 Comparison of different approaches with target key error probability 10^{-6} synthesized for Xilinx Spartan 6 FPGAs

PUF err. prob.	Approach (+ Reference)	PUF bits	HD bits	Slices	Clock cycle est.	Comment
5%	CO + Rep(3,1,3) + BCH(255,171,23) [64]*	765	765	103	106,000	
10%	CO + Rep(7,1,7) + BCH(59,35,9) [64]*	2,065	2,065	89	68,000	
10%	DSC + Viterbi + SPONGENT 88 [24]	608	707	146	28,000	Binary reliability indicator
15%	CO + Rep(5,1,5) + BCH(226,86,43) [64]*	2,260	2,260	107	365,000	
15%	CO + Rep(3,1,3) + RM(64,22,16) [17]	1,536	13,952	–	10,000	Soft Input Information
15%	CO + Rep(8,1,8) + Golay(24,12,8) [65]	2,880	2,880	–	–	
15%	C-IBS(9,4) + RM(8,4,4) [21]	2,304	9,216	76	9,000	Soft Input Information
15%	DSC + Viterbi + SPONGENT 88 [24]	974	1,108	146	30,000	Binary Reliability Indicator
20%	CO + Rep(13,1,13) + BCH(255,171,23) [64]*	3,315	3,315	104	118,000	
20%	DSC + Viterbi + SPONGENT 88 [24]	3,780	1,575	146	38,000	Binary Reliability Indicator
24%	ML symbol approach + non-binary PC [25]	753,664	2,944	–	–	
25%	DSC + Viterbi + SPONGENT 88 [24]	15,120	1,977	146	72,000	Binary Reliability Indicator

For the implementations denoted with *, we used a custom BCH decoder instead of the originally published one

Table 2 Comparison of different approaches with target key error probability 10^{-9} synthesized for Xilinx Spartan 6 FPGAs

PUF err. prob.	Approach (+ Reference)	PUF bits	HD bits	Slices	Clock cycle est.	Comment
10^{-5}	SLLC + BCH(55,43,5) [19]*	165	36	43	23,000	
5%	CO + Rep(3,1,3) + BCH(127,57,23) [64]*	1143	1143	95	945,000	
10%	CO + Rep(7,1,7) + BCH(255,171,23) [64]*	1785	1785	116	1,843,000	
10%	DSC + Viterbi + SPONGENT 88 [24]	810	810	146	29,000	Binary reliability indicator
13%	CO + Rep(7,1,7) + BCH(318,174,34) [18]	2226	2052	243	55,000	
14%	CO + RM(GCC) [27,66]	2048	2048	179	109,000	Error probability 1.5×10^{-9}
14%	CO + RM + RS + SPONGENT 128 [27]	1152	1152	–	–	
15%	CO + Rep(3,1,3) + BCH(251,43,85) [64]*	3012	3012	116	1,843,000	
15%	DSC + Viterbi + SPONGENT 88 [24]	1890	1236	146	32,000	Binary reliability indicator
15%	Optimized polar codes [28]	974	896	–	–	
20%	CO + Rep(5,1,5) + BCH(243,43,85) [64]*	5020	5020	116	1,853,000	
20%	DSC + Viterbi + SPONGENT 88 [24]	7020	1852	146	48,000	Binary reliability indicator

For the implementations denoted with *, we used a custom BCH decoder instead of the originally published one

slices on a Xilinx Spartan 6 FPGA. To ensure comparable results, all RAMs were synthesized as distributed RAM.

3.2 Comments on SLLC and ML symbol recovery

Typically, approaches are designed to derive reliable keys for mean input error probabilities $\mu(p_{puf}) > 10\%$ and require more than $n = 700$ PUF bits to generate $k = 128$ key bits. SLLC reduces the number of helper data bits from n to $n - k$ so that it only gives a small improvement for high error rates. It was shown in [19] that SLLC shows its best performance for low input error probabilities. The data point from [19] is added in Sect. 3.4 for completeness. However, it is hard to make a meaningful comparison to the other implementations operating on completely different input data.

The ML symbol-based approach in [25] also follows a different optimization direction. It assumes a strong PUF with high error probability at the input that provides a large number of PUF response bits so that the approach cannot be directly compared to the other approaches in this comparison either.

3.3 Syndrome coding and ECC designs for medium key error probability

Several works such as [4,15,17,21,22,24,25] set a key error probability of 10^{-6} as target error probability for the output keys and presented corresponding implementations. Table 1 provides results for different average input error probabilities that follow the distribution in [16]. The numbers of PUF and

helper data bits serve as measure for the effectiveness while the complexity directly effects the number of slices and the run time of the algorithms.

The following candidates will be evaluated in detail in the following:

- A code-offset implementation with repetition and BCH codes was presented in [64]. The original paper only shows numbers for Spartan 3 FPGAs. New Spartan 6 results are based on our BCH decoder implementation. For SPONGENT 128/128/8 hash function [81], the implementation presented in [80] was used instead of the original the Toeplitz Hash [82]. Note that the code-offset construction could be replaced by SLLC to reduce the helper data size by the key length, i.e., 128 bit.
- A second code-offset implementation based on repetition and Reed–Muller codes is given in [16,17]. The helper data is extended by reliability information on each specific PUF response bit to improve the decoding performance by applying soft-decision decoding.
- The third code-offset candidate presented in [65] shows results for concatenated codes with extremely short block lengths using the extended binary Golay code, which is also related to BCH codes [83]. Creating soft information in the repetition decoder and considering it in the outer Golay decoder reduces the number of PUF bits by 38% compared to hard decision decoding.
- C-IBS with Reed–Muller codes was introduced in [21]. It is a pointer-based approach and again Reed–Muller decoders with soft input information are used. However, the block lengths are the shortest in this comparison.
- DSC and a Viterbi decoder are another approach in this comparison [22–24]. While the soft-decision RM designs [16,17] and C-IBS [21] store precise reliability information in the helper data, DSC only requires a binary reliability indicator for each PUF bit.

We will use the mean error probability of the PUF $\mu(p_{puf})$ as common reference to compare the approaches. Therefore, it is plotted on the x-axis of all following figures. The other measures are presented on the y-axis and given in detail in Columns 3 to 6 in Table 1, and later also in Table 2 for the second scenario.

PUF response bits The numbers of PUF bits are provided in Figure 4. For 15% input error probability, the black code-offset BCH point falls together with the code-offset RM data point. DSC is very efficient in the middle area. It can also be seen that C-IBS and the code-offset RM implementation show an improved performance due to soft-decision decoding. Compared with the code-offset BCH approach, C-IBS achieves a similar performance with only 4% of the code length in the outer code. The code-offset Golay design has the highest number of PUF bits, due to the small code length and the lack of the bit-specific soft information. The code-offset RM implementation is more efficient through the larger block size and by applying the more fine-grained soft decision decoding. However, DSC is even more efficient than the other references in the 15% input error data points.

Moving to 10% input error probability, DSC requires only 30% of the PUF response bits of the code-offset BCH value. Increasing the input error probability to 20%, the code-offset BCH approach with a BCH code of length 255 outperforms DSC with a (2, 1, [7]) convolutional code. Going to a more powerful but also more complex (2, 1, [8]) code will reduce the DSC error probability again.

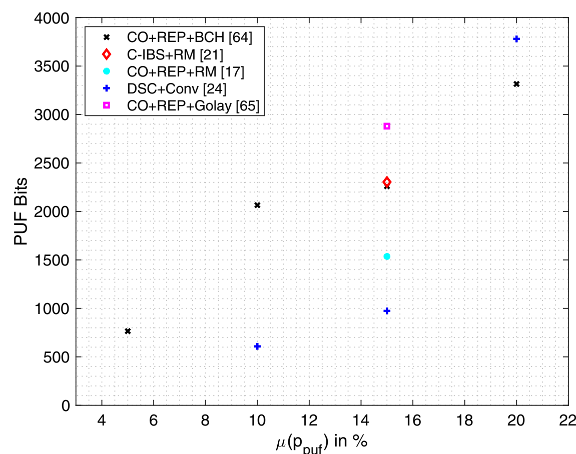

Fig. 4 Number of PUF bits of different syndrome coding and ECC approaches designed for a key error probability of 10^{-6}

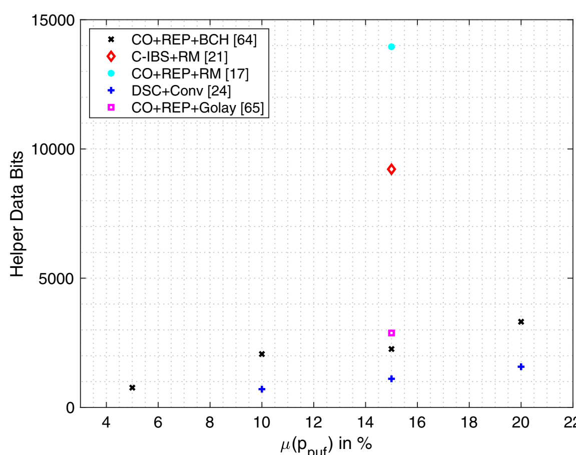

Fig. 5 Number of helper data bits of different syndrome coding and ECC approaches designed for a key error probability of 10^{-6}

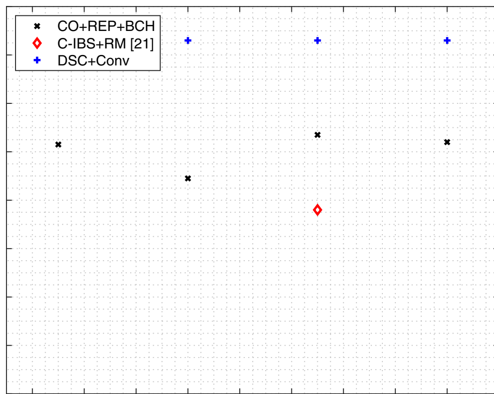

Fig. 6 Number of Spartan 6 slices of different syndrome coding and ECC approaches designed for a key error probability of 10^{-6}

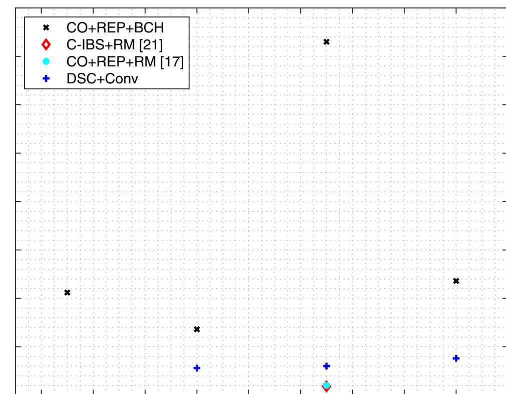

Fig. 7 Number of clock cycles of different syndrome coding and ECC approaches designed for a key error probability of 10^{-6}

Helper data Figure 5 shows the helper data sizes for the different schemes. Here, the hard-decision approaches lead to significantly lower numbers than the soft-decision approaches. The code-offset BCH and DSC designs have lower numbers that scale almost linearly with the input error probability. The helper data compression for DSC [23] is most efficient for high PUF bit to key bit ratios on the right of the x-axis. The code-offset Golay approach also has a relatively compact helper data.

In contrast, the code-offset and C-IBS soft-decision approaches [17,21] have to store reliability information in the helper data such that the overall helper data size is noticeably increased.

Slices The slice counts in Figure 6 show the first part of the implementation complexity. [17] only presents Spartan 3E results so that it cannot be compared to the other approaches in this comparison. Details on the corresponding architectures and implementations can be found [21,78] and in the "Appendix".

The BCH decoder is based on a processor architecture where the block size has only a minor impact on the size of the hardware architecture. The (2, 1, [7]) Seesaw Viterbi decoder is identical for all DSC results, keeping the implementation size constant. The seesaw decoder benefits from synthesizing the memory in Block RAM instead of using distributed RAM, since about half of the module is filled with distributed RAM. Therefore, the BCH decoder is roughly 25–30% smaller. If Block RAM is permitted, the more efficient results discussed in the Seesaw implementation [24] hold. Due to its small code length of only 8, the C-IBS implementation in [21] is both, compact and fast.

Figure 7 shows the cycle counts of the different implementations which differ by multiple orders of magnitude. This is due to different code classes, hardware architectures and code parameters. For example, the number of operations in the syndrome calculation of the processor-based BCH decoder highly depend on the code parameters leading to a significant variation in execution time. The 15% data point is the slowest in this comparison, because a length 255 BCH code with a low rate is used, where many syndrome equations have to be solved. For DSC, the decoding time increases linearly with the number of PUF bits since the Viterbi decoder remains unchanged. The Reed–Muller decoder in [21] is fastest due to its small code length.

Trade-offs The radar chart in Figure 8 compares the data points at 15% bit error probability. The values are normalized to the maximum value for each category in the comparison so that the most efficient approaches group around the center point of the plot.

The DSC implementation is efficient in terms of PUF response bits, helper data bits and clock cycles with a higher implementation complexity as long as no Block RAM is used. The code-offset implementation based on repetition and BCH codes requires a large number of PUF response bits and clock cycles but therefore provides fairly constant and relatively small slice counts. Providing reliability information on the input data as in code-o ffset RM and C-IBS RM implementation [17,21] leads to an increase in helper data size but helps to reduce either the number of PUF bits or the decoding complexity. The code-offset Golay approach promises a compact implementation because of the very short codes, but unfortunately only ASIC implementation results are available.

The precise numbers discussed in this section can be found in Table 1 at the beginning of the section.

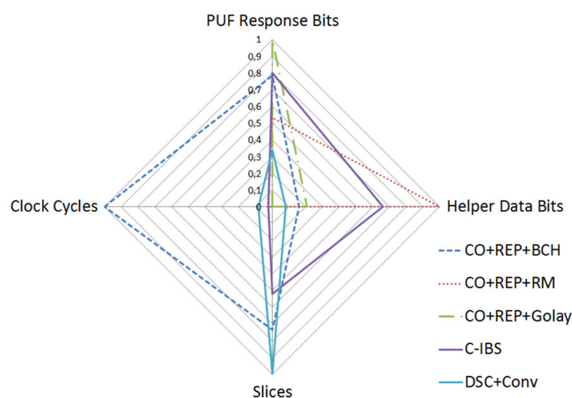

Fig. 8 Trade-offs between different syndrome coding and ECC approaches designed for a key error probability of 10^{-6}

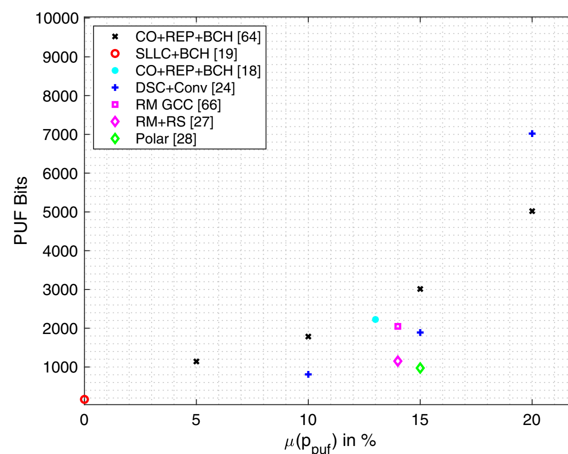

Fig. 9 Number of PUF bits of different syndrome coding and ECC approaches designed for a key error probability of 10^{-9}

3.4 Syndrome coding and ECC designs for low key error probability

In addition to the discussed results for a key error probability of 10^{-6}, several other implementation were designed for a lower key error probability of 10^{-9}. This improved reliability can only be achieved by increasing the error correction capabilities of the applied codes. These seven implementation candidates are compared in the following:

- A candidate based on SLLC and BCH codes was discussed in [19]. In contrast to the other candidates, it was designed as lightweight solution for the extremely low input error probabilities in [79]. Therefore, it is not directly comparable to the other designs in this comparison.
- The code-offset fuzzy extractor using BCH and repetition codes in [64] serves again as baseline reference.
- An optimized BCH and repetition code implementation integrated into a standalone PUF key generation module was published in [18].
- The design and implementation of a construction based on RM codes with generalized concatenated codes (GCC) [27,66] shows that the decoding complexity can be reduced with a GCC construction by using shorter concatenated block codes.
- The concatenated Reed–Muller and Reed–Solomon (RS) code construction in [27] is the first that shows the potential of RS codes in the PUF context. The designs in [27] and [84] also discuss incremental improvements through more sophisticated RS constructions that will not be taken into account in the following due to their increased implementation complexity.
- Bounded DSC and Viterbi results discussed in [22] also provide DSC results for a lower output key error probability.
- Recent work on Polar codes seems very promising [28], but since no implementation results are published so far, its practical applicability cannot be assessed yet.

PUF response bits Figure 9 shows similarities to Figure 4. DSC only requires a small number of PUF response bits for expected input error probabilities 10% and 15%. The 20% data point exceeds the code-offset Rep and BCH code approach due to the rather simple (2, 1, [7]) convolutional code. For 13–15% input error probability, the DSC results are slightly better than the code-offset approaches in [18,26]. They both use roughly two thirds of the PUF response bits of the oldest code-offset repetition and BCH implementation [64]. The Reed–Solomon construction in [27] uses only half of the PUF bits of the other constructions which makes it a very efficient approach for this input error range. The Polar codes in [28] provide even slightly lower results.

Helper data Again, the helper data size varies over the different approaches. The number of helper data bits in Figure 10 is equal to the number of PUF response bits in Figure 9 are equal for the code-offset approach. For the syndrome approach and SLLC, the helper data size is computed as difference between the number of PUF response bits and the size of the secret.

For large input error rates, the DSC helper data size would grow significantly. This is mitigated by helper data compression where the m parameter is increased in the RLE encoder for larger $\mu(p_{puf})$. The maximum helper data sizes for the 10^{-9} scenario are smaller than for the 10^{-6} setting because no reliability information is stored in the helper data for the discussed schemes.

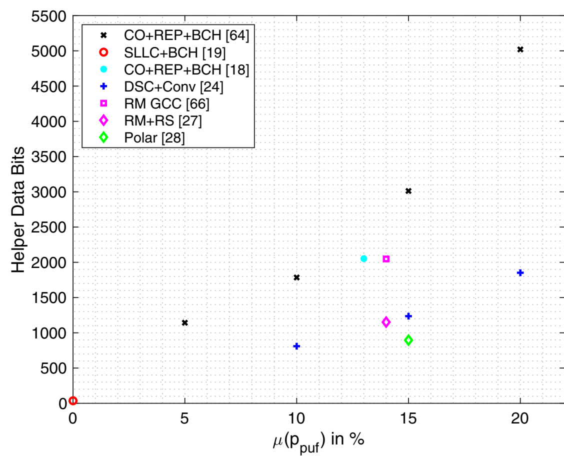

Fig. 10 Number of helper data bits of different syndrome coding and ECC approaches designed for a key error probability of 10^{-9}

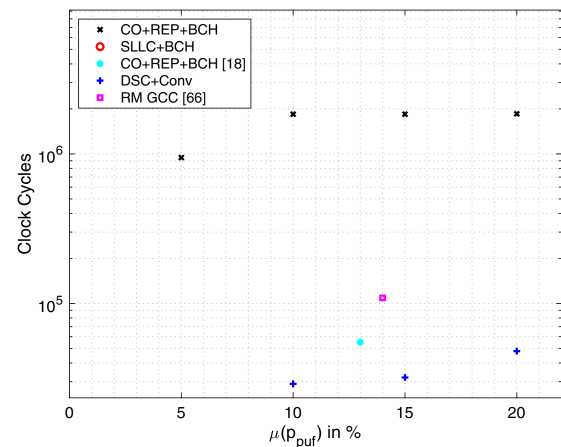

Fig. 12 Number of clock cycles of different syndrome coding and ECC approaches designed for a key error probability of 10^{-9}

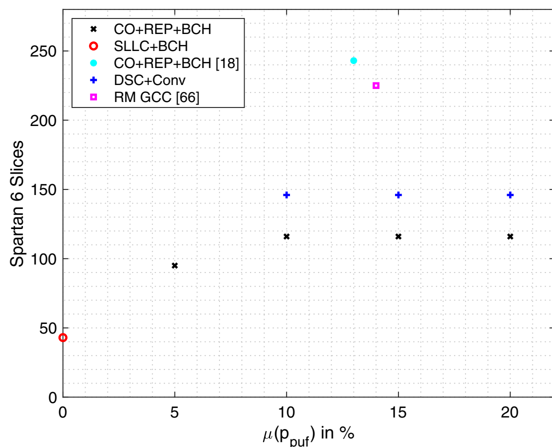

Fig. 11 Number of Spartan 6 slices of different syndrome coding and ECC approaches designed for a key error probability of 10^{-9}

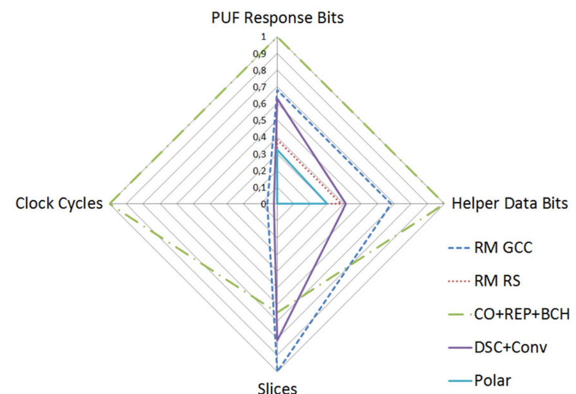

Fig. 13 Trade-offs between different syndrome coding and ECC approaches designed for a key error probability of 10^{-9}

Slices The slice counts of the different implementations are given in Figure 11. Again, the implementations of the code-offset repetition and BCH, and the SLLC construction are relatively compact due to the processor-based BCH decoder. DSC is slightly larger while the BCH [18] and RM [66] implementations with large code lengths require more than 200 slices. Again, details on the corresponding implementations can be found [19,66,77,78] and in the "Appendix".

The best of our knowledge, there is no implementation available published for the RS construction presented in [27]. A Reed decoder [56] was used in [66] instead of the original recursive decoder in [27] which increases the output error probability slightly to 1.5×10^{-9}.

Clock cycles Figure 12 shows the numbers of clock cycles on a logarithmic scale. The code-offset repetition and BCH implementations are more than one order of magnitude slower compared to the other candidates due to the compact BCH decoder design.

Figures 11 and 12 show that DSC is both, smaller and faster, than the code-offset and syndrome constructions in [18,66]. The pre-selection of reliable PUF bits pays off, since it reduces the required complexity of the subsequent ECC.

Trade-offs The radar chart in Figure 13 compares the data points at 14% and 15% bit error probability. The values are normalized to the maximum value for each category in the comparison so that the most efficient approaches group around the center point of the plot.

Looking over all categories, DSC provides a good performance across PUF bits, helper data, FPGA slices and clock

cycles. However, it requires that reliability information for each PUF response bit is available during generation. Focusing on specific categories, other approaches show a stronger performance with the downside of weaker numbers in other criteria. The numbers corresponding to the plots in this subsection are provided in Table 2. Optimized code constructions with RS and Polar codes promise a great decrease in PUF and helper data bits, e.g., in Figure 13, while the implementation overhead is still unclear.

3.5 On the blessing and curse of reliability information

In general, reliability information is helpful for decoding, as it provides additional input and increases the probability of making a correct decoding decision. The simplest way of using reliability information is to take the reliability of previous decoding steps into account as it was performed, e.g., in the repetition and Golay design in [65] for concatenated codes or in the Maximum-Likelihood approach [25] for input symbols. In [85], the reliability of a decoded key was considered to decide if the key should be output or the decoding process should be restarted with a fresh PUF readout and a possibly more favorable error pattern.

Reliability information can also already be considered at the input of the decoder by evaluating the reliability of individual PUF response bits, e.g., the counter differences in RO PUFs, or analog PUF values [3,44,86], or by evaluating the same PUF multiple times and applying a majority voting [52]. Additionally, in the decoding of concatenated codes the error correction decoder can annotate its decoded output with reliability information, e.g., in the form of erasure symbols for binary codes as in [66], to support later stages of decoding.

If no online reliability information is available, reliable bits can be labeled as, e.g., by applying dark bit masking [52], by the indexing operations in [20,23,24], or even by storing precise reliability values [16,17,21]. From a security perspective, it is important to check if the helper data leaks information, e.g., for biased PUF.

However, the extraction and use of reliability information adds additional overhead that resembles in the effort of deriving the PUF bits and the ECC decoders, as well. Using reliability information only adds a moderate amount of complexity to the error correction decoder in the best case, but might also require a full redesign of the decoder for others. Extracting the online reliability information of binary-output PUFs requires multiple read-outs, adding latency and additional requirements on memory. Depending on PUF type and scenario, the can add a significant delay to the extraction the key.

Generating the reliability information during enrollment can also cause an overhead in terms of initialization latency. This can have significant impact, for example when testing PUF ICs with low bandwidth wafer testing and limited time slots, or other hard provisioning constraints. Therefore, when using the soft-input approaches [16,17,21] and also DSC [24] with a binary indicator, the effort of generating the required reliability at the input side also has to be taken into account during system design.

This also resembles in possible code classes, as some naturally support soft decoding, such as RM, Golay or convolutional codes, while it is more difficult for other, such as BCH. The improved error correction performance can also come at an increased implementation complexity so that one has to compare the saving in PUF size to the increase in decoding and helper data effort. For example, implementing a recursive soft-decoder [87] as in [17] has a significantly more complex control flow than the Reed decoder [56] in [66].

3.6 Helper data manipulation

The helper data must not leak too much information about the derived secret, as it has to be assumed as public. However, it does not have to be assumed write proof, e.g., if it is loaded from an external source into the chip. In this case, helper data attacks can be mounted that can have devastating impact on the security of the error correction in multiple ways. Obviously, helper data manipulation can mount a denial-of-service attack and simply prevent the device from generating the correct key.

In addition, the attacker can vary the input of the decoder to mount differential side-channel attacks [12,35].

For DSC, pointer manipulation allows to reduce the key entropy to one bit in the worst case without error correction. This attack also relates to C-IBS, if more than one bit is extracted from a block and can be countered if the helper data directly contributes to the key, e.g., by XORing the derived key with a hash value of the helper data [22].

Linear constructions, especially the fuzzy commitment are susceptible to related key attacks [30]

For further reading, we also refer to [30,51] where helper data manipulation reduces the complexity of attacking constructions with code concatenation and various helper data generation schemes.

4 Implementation results on recent FPGAs

As the previously discussed implementations only provide results for legacy Spartan 6 or even Spartan 3 FPGAs, it is unclear how the implementations of the different error correction schemes will compare in current technology. Therefore, this section provides new implementation results for the previously discussed designs. To cover multiple technologies, we synthesized results for three additional FPGAs of different FPGA generations specified in Table 3.

Table 3 List of Xilinx FPGAs that were selected for comparison

FPGA Family	Device	Fab (nm)	Slice count
Spartan 6	xc6slx45	45	6.8 K Slices
Zynq 7000	xc7z020	28	85 K Slices
Kintex Ultrascale	xcku040	20	530 K CLBs
Zynq Ultrascale+	xczu9eg	16	600 K CLBs

The results were synthesized with Vivado 2015.2 or 2017.1, while the old Spartan 6 results were obtained with ISE. The optimization strategy *area_high* for synthesis yielded the smallest modules for the Ultrascale and Ultrascale+ devices. For the Zynq 7020 the optimization strategy *area_medium* was chosen, as in most cases this strategy yielded smaller modules than with the *area_high* setting.

The transition from logic cells in 28 nm and above to system logic cells in the Ultrascale devices added new functionality and flexibility and thus generally leads to a decrease in cell count.

Table 4 shows the cell counts over the different technologies and implementations. The BCH decoders in our code-offset implementations are varied in their parameters and therefore lead to a variety of results. For DSC, all designs have the roughly the same size. It can be seen that the number of slices is reduced significantly from Spartan 6 to Zynq and again to the Ultrascale FPGAs. As expected, this decrease is fairly homogenous over the different schemes such that no implementation benefits significantly and scales better than the other schemes.

In general, the Zynq slice counts are between 58% and 76% of the Spartan 6 slice counts. Most Ultrascale results are in the range of 37–56% of the Spartan 6 slices. We have two major outliers: The BCH (55,43,5) are almost double the other implementations while the BCH (318,174,34) is larger but therefore also orders of magnitude faster. This is due to the fact that the original implementation from [18] was used.

5 Choosing a suitable code

When it comes to creating a short-list of good ECC candidates for a PUF module, we give some practical design recommendations to come to an efficient ECC implementation.

Even if outside the specific scope of this work, a good characterization of the PUF concerning its entropy and reliability is the foundation for all later steps.

Recommendation 1: Use all available information. Reliability information helps to improve the efficiency. Using dynamic reliability information on the inputs, such as RO counter values or quantization information [44] allows to assess the reliability of a specific value, especially when it comes more or less for free at the input stage. Storing static reliability information on individual PUF bits, as it was done, e.g., in [16,17,21] also gives a notable benefit. Also, generating reliability information on an intermediate stage as in [65] helps improving the performance. Going beyond pure reliability information to higher-order alphabets and quantization helps extracting more secret information [44].

Recommendation 1a: Rule out unreliable bits. Correcting errors is expensive, especially if no reliability information is available. A wide portion of the efficiency of the DSC designs [24] is owed to the fact that a lower number of errors occurs because only reliable PUF bits are indexed. By definition, discarding information either trough indexing operations in the pointer-based approaches, e.g., IBS or DSC [20,24] or dark bit masking [52] cannot lead to theoretically optimal results but can bring down the implementation complexity in practice.

Recommendation 2: Large blocks improve the error correction performance. In general, larger code blocks achieve error correction rates closer to theoretical limits. Given a limited amount of PUF bits, small blocks reduce implementation complexity, but may not be sufficient to achieve the desired reliability. Therefore, we only observed few candidates with very small block lengths. For PUFs with very low error rates ($\leq 1\%$), already one single code block without concatenation can suffice, as in [19]. However, note that the implementation effort typically increases nonlinearly with block size.

Recommendation 2a: Combining multiple small blocks helps mimicking large blocks. Most discussed approaches use concatenated codes [56,76] to create larger blocks. The ensemble of designs shows a sweet spot between block lengths of roughly 64 and 256 where the codes already provide a good efficiency while the implementation overhead is still manageable. While simple concatenation as in older candidates such as [64] allows for a fast design, more advanced concatenation techniques help improving the performance further [26,27]. Therefore, it would be very interesting to also explore the potential of GCC constructions for the 10^{-6} scenario and multiple 10^{-9} data points in the future. Going to RM RS with GCC gives an additional boost in performance at the cost of complexity. On the other side, the short code length of the Golay code limits its efficiency as outer code.

Table 4 Comparison the slice counts of different approaches synthesized for state-of-the-art Xilinx FPGAs

Approach and original reference	Spartan 6 (Slices)	Zynq 7020(Slices)	Ultrascale (CLBs)	Ultrascale+ (CLBs)
SLLC + BCH(55,43,5) [19]	43	57	38	37
CO + Rep(3,1,3) + BCH(127,57,23)* [64]	95	55	35	35
CO + Rep(5,1,5) + BCH(127,57,23)* [64]	95	55	35	35
CO + Rep(3,1,3) + BCH(255,171,23)* [64]	103	67	57	54
CO + Rep(7,1,7) + BCH(59,35,9)* [64]	89	58	39	38
CO + Rep(5,1,5) + BCH(226,86,43)* [64]	107	67	57	54
CO + Rep(13,1,13) + BCH(255,171,23)* [64]	104	69	58	55
CO + Rep(3,1,3) + BCH(251,43,85)* [64]	116	86	62	59
CO + Rep(5,1,5) + BCH(243,43,85)* [64]	116	86	62	59
CO + Rep(7,1,7) + BCH(318,174,34) [18]	243	185	125	127
FC + RM(GCC) [27,66]	179	133	75	84
C-IBS(9,4) + RM(8,4,4) [21]	76	58	37	31
DSC + Viterbi + SPONGENT 88 [24]	146	101	64	64

For the implementations denoted with *, we used a custom BCH decoder instead of the originally published one

Recommendation 2b: Increasing the PUF size increases reliability. In principle, adding more redundancy to an error correction code increases the reliability. Simple code concatenation can be applied to increase code size with only minor increases in implementation complexity, such as [18,64]. In reverse, if PUF bits are costly, a higher reliability can also be achieved using stronger codes with better error correction performance but higher decoding complexity, e.g., with [27].

Recommendation 2c: Plan rather more than too little error correction. Error rates of real-world PUFs may divert from their stochastic approximation. For scenarios that are more focused on reliability, it is practical to add a small safety margin and choose a slightly more powerful error correction. As improvements of the error correction increase implementation complexity and are limited to the theoretical channel capacity, it is often preferable to increase PUF size instead, as pointed out in the previous paragraph. For generic scenarios with medium to high error rates ($\leq 14\%$), the RM GCC construction provides a good trade-off between implementation complexity and reliability.

Recommendation 3: Take advantage of existing resources. Some decoder implementations benefit greatly from special resources. For example, the Viterbi decoder in [78] benefits significantly from Block RAM. BCH decoders can benefit for example from existing GF multipliers.

Recommendation 4: Take advantage of code class properties. Encoders and decoders for repetition and RM codes have lower complexity resulting in faster or more compact implementations, but are restricted to certain parameter configurations. Such codes are also more often used as inner codes in concatenated code constructions. RS and BCH codes are more flexible, but may result in a more complex implementation. These can be used as outer code in a concatenated code construction, as they can be constructed to an arbitrary dimension that fits to any codeword size and only have to be decoded once.

Recommendation 4a: Consider trade-offs in decoding algorithms. For each code class there exist multiple decoding algorithms with different trade-offs. For example, RM codes can be decoded as generalized concatenated codes with higher reliability [76]. This algorithm is recursive and its decoding steps for a specific code parameter set can be stored as microcode [17]. The recursive structure stores soft information which results in a higher memory consumption than using the iteratively structured Reed decoding [66]. The middle way is to cut off some of the soft information with intermediate hard decisions, as proposed in [27] as recursive RM decoder.

Once a decoding algorithm is chosen, further implementation trade-offs can be made. While straight-forward BCH decoder implementations show a fairly low run-time, the application-specific processors in [18] or in the “Appendix”

have a fairly constant size over different parameters but vary significantly in run-time.

The output error probability of bounded-minimum distance decoders decoding up to $\lfloor d/2 \rfloor$ errors can be easily computed. The error probabilities of other decoders can be obtained through Monte-Carlo simulations.

Recommendation 4b: Be aware of data dependent run-times in decoding algorithms. Advanced decoding algorithms, e.g., list decoding or belief propagation can easily show data-dependent timing behavior, which typically speeds up decoding but creates timing side-channels.

Recommendation 5: Ensure that the derived secret has sufficient entropy. The helper data can leak different amounts of entropy depending on the properties of the PUF, e.g., bias or correlation, and the post-processing scheme. Therefore, it is important to assess the helper data leakage and potential losses [13,30,47,51] to ensure that the remaining entropy still matches the security target.

6 Conclusion

We present a comprehensive overview over the state of the art of error correction schemes for PUFs and their implementations, and provide new implementation results on recent FPGAs.

The evaluation shows that the field matured over time and more efficient solutions were introduced. However, the approaches still have individual strengths and weaknesses such that there is no one-size-fits-all solution.

The first implementations provide a solid foundation with several options for improvement. The implementation with DSC and a convolutional code provides a good general trade-off over different criteria, but requires a reliability indicator for the individual PUF bits. In general, reliability information improves the performance. More recent code constructions with RS and Polar codes promise a good error correction, especially for low output error probabilities, but still lack practical implementations to fully assess their performance.

We discuss design recommendations to support a designer in selecting efficient algorithms and implementation strategies under given design constraints.

Acknowledgements Open Access funding provided by Projekt DEAL. The authors would like to thank Milos Grujic from KU Leuven for providing updated results for recent FPGAs for the PUFKY implementation [18], Julian Leyh for his work on the BCH decoder implementation, and the anonymous reviewers for the several rounds of detailed and constructive feedback.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix

A Custom BCH decoder design and implementation

For the area-optimized BCH decoder, we implemented a soft-core similar to [18,77], defined an assembler and implemented the firmware [88]. Limiting the instruction set and the supported types of branches allows for area efficient implementation.

A.1 Architecture

Figure 14 shows the overall architecture of the decoder that follows a clear separation of addressing and data manipulation. Datapaths to the control block are not included in the diagram for better readability. Data is always addressed indirectly through the address block, which, in turn, performs calculations over the integers. The firmware is stored in ROM while the codeword and intermediate results are placed in RAM. To avoid obvious timing side channels, all instructions have data-independent cycle counts.

- The control block computes, which instruction to execute next based on the current (branch) instructions and contains of some comparators to provide signals for branching and the program counter which either increments by one in every clock cycle or is set to an immediate from the ROM in case of a branch.
- The instruction block then loads the requested code from its ROM and decodes it. The opcode and immediate field from the currently decoded instruction is forwarded to all other modules. The decoded instruction also contains two addresses, used to index two words in the address memory. These are used as address inputs to select two data words. Depending on the instruction to execute, either the data memory's write enable, the address memory's write enable or some control path is enabled and a calculation result or an immediate from the instruction ROM is stored.

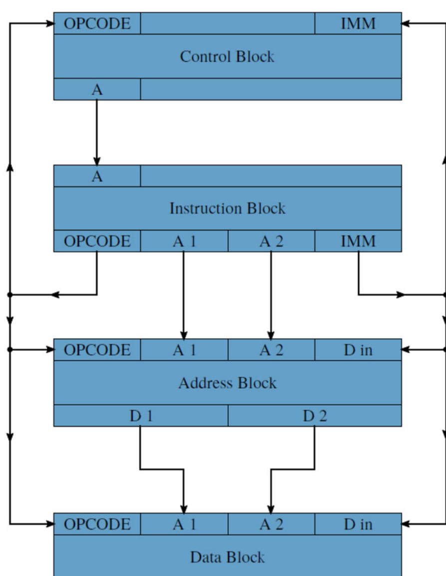

Fig. 14 Processor address and dataflow graph

- The address block, not displayed here, is very similar to the data block, with the difference of the arithmetic unit performing different operations and the outputs of the address memory also being outputs of the whole address block.
- Within the data block, an arithmetic unit performs multiplications or additions in the field F_2^m.

A significant part of the area went into the instruction block, so we designed a custom instruction set that directly interfaces the datapath multiplexers and write enables with the instruction word without the need of any decoding. Most datapath multiplexers are directly driven by the current instruction word.

There is no pipelining and almost everything is combinatorial to enable single cycle operation with most instructions and reduce control overhead. Data synchronization is mostly achieved by the synchronous write inputs of the RAMs, also leading to a low register count. The support for procedure calls has been dropped to eliminate the need for a stack and any other connections between the program counter to the datapath. Branching support is quite limited as well.

While PUFKY uses some additional means to reduce runtime, such as a multiply-accumulate-unit, no such efforts have been taken here for a low area footprint.

A.2 Firmware

The firmware is split into 5 algorithms, namely Input, Syndrome Calculation, Berlekamp- Massey- Algorithm, Search and Correct Errors and Output.

- Input and Output contain simple loops going over each codeword bit either writing or reading it from or to the memory.
- Syndrome Calculation selectively sums over powers of α. The result of the algorithm are the coefficients of the syndrome stored in the general purpose memory. Four additional memory words are used for buffers and powers of α during algorithm runtime.
- The Berlekamp- Massey- Algorithm for finding $\Lambda(x)$ is certainly the most complex part of the firmware. The algorithm is implemented after the one shown in [76] with some slight modifications. As many calculations as possible are moved out of the if-else construction to even out runtime in both paths. In some places, additional operations were introduced to guarantee constant timing and a loop was extended to make timing of all paths equal.
- The Search and Correct Errors part is very similar to the Syndrome Calculation algorithm. This architecture has been shown to be more compact than the Chien Search for the processor's instruction set. If any power of α is detected to be a root of $\Lambda(x)$, the corresponding bit in the codeword is flipped. Otherwise, stub operations are executed to not give away any information by the program's runtime.

References

1. Gassend, B., Clarke, D., van Dijk, M., Devadas, S.: Silicon physical random functions. In: ACM Conference on Computer and Communications Security (CCS), pp. 148–160 (2002)
2. Lim, D., Lee, J.W., Gassend, B., Suh, G.E., van Dijk, M., Devadas, S.: Extracting secret keys from integrated circuits. IEEE Trans. Very Large Scale Integr. VLSI Syst. **13**(10), 1200–1205 (2005)
3. Tuyls, P., Schrijen, G.-J., Skoric, B., van Geloven, J., Verhaegh, N., Wolters, R.: Read-proof hardware from protective coatings. In: Goubin, L., Matsui, M. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 4249, pp. 369–383. Springer, Berlin (2006)
4. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: FPGA intrinsic PUFs and their use for IP protection. In: Paillier, P., Verbauwhede, I. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007)
5. Suh, G.E., Devadas, S.: Physical unclonable functions for device authentication and secret key generation. In: ACM/IEEE Design Automation Conference (DAC), pp. 9–14 (2007)
6. Lu, T., Kenny, R., Atsatt, S.: Secure Device Manager for Intel Stratix 10 Devices Provides FPGA and SoC Security (WP-01252-1.2). Intel Corporation, Santa Clara (2018)
7. Peterson, E.: Developing Tamper-Resistant Designs with Zynq UltraScale+ Devices, XAPP1323 (v1.0). Xilinx, Inc., San Jose (2017)
8. NXP Semiconductors: SmartMX3 Family P71D320 Product Short Data Sheet (rev. 3.0). Report (2017)
9. Yu, S., Chen, P.-Y.: Emerging memory technologies: recent trends and prospects. IEEE Solid State Circuits Mag. **8**(2), 43–56 (2016)
10. Maes, R.: Physically Unclonable Functions: Constructions, Properties and Applications. Dissertation, KU Leuven (2012)

11. Herder, C., Yu, M., Koushanfar, F., Devadas, S.: Physical unclonable functions and applications: a tutorial. Proc. IEEE **102**(8), 1126–1141 (2014)
12. Tebelmann, L., Pehl, M., Sigl, G.: EM side-channel analysis of BCH-based error correction for PUF-based key generation. In: Workshop on Attacks and Solutions in Hardware Security (ASHES), pp. 43–52. ACM (2017)
13. Delvaux, J., Gu, D., Verbauwhede, I., Hiller, M., Yu, M.: Efficient fuzzy extraction of PUF-induced secrets: theory and applications. In: Gierlichs, B., Poschmann, A. (eds.) Conference on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS. Springer, Berlin (2016)
14. Dodis, Y., Reyzin, L., Smith, A.: Fuzzy extractors: how to generate strong keys from biometrics and other noisy data. In: Cachin, C., Camenisch, J.L. (eds.) Advances in Cryptology (EUROCRYPT), Series LNCS, vol. 3027, pp. 523–540. Springer, Heidelberg (2004)
15. Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., Tuyls, P.: Efficient helper data key extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 5154, pp. 181–197. Springer, Heidelberg (2008)
16. Maes, R., Tuyls, P., Verbauwhede, I.: A soft decision helper data algorithm for SRAM PUFs. In: IEEE International Symposium on Information Theory (ISIT), pp. 2101–2105 (2009)
17. Maes, R., Tuyls, P., Verbauwhede, I.: Low-overhead implementation of a soft decision helper data algorithm for SRAM PUFs. In: Clavier, C., Gaj, K. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), pp. 332–347. Springer, Heidelberg (2009)
18. Maes, R., Van Herrewege, A., Verbauwhede, I.: PUFKY: a fully functional PUF-based cryptographic key generat. In: Prouff, E., Schaumont, P. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 7428, pp. 302–319. Springer, Heidelberg (2012)
19. Hiller, M., Yu, M., Pehl, M.: Systematic low leakage coding for physical unclonable functions. In: ACM Symposium on Information, Computer and Communications Security (ASIACCS) (2015)
20. Yu, M., Devadas, S.: Secure and robust error correction for physical unclonable functions. IEEE Des. Test Comput. **27**(1), 48–65 (2010)
21. Hiller, M., Merli, D., Stumpf, F., Sigl, G.: Complementary IBS: application specific error correction for PUFs. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 1–6 (2012)
22. Hiller, M., Weiner, M., Rodrigues Lima, L., Birkner, M., Sigl, G.: Breaking through fixed PUF block limitations with differential sequence coding and convolutional codes. In: International Workshop on Trustworthy Embedded Devices (TrustED), pp. 43–54. ACM (2013)
23. Hiller, M., Sigl, G.: Increasing the efficiency of syndrome coding for PUFs with helper data compression. In: Design, Automation & Test in Europe Conference & Exhibition (DATE). ACM/IEEE (2014)
24. Hiller, M., Yu, M., Sigl, G.: Cherry-picking reliable PUF bits with differential sequence coding. IEEE Trans. Inf. Forensics Secur. **11**(9), 2065–2076 (2016)
25. Yu, M., Hiller, M., Devadas, S.: Maximum likelihood decoding of device-specific multi-bit symbols for reliable key generation. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (2015)
26. Müelich, S., Puchinger, S., Bossert, M., Hiller, M., Sigl, G.: Error correction for physical unclonable functions using generalized concatenated codes. In: International Workshop on Algebraic and Combinatorial Coding Theory (ACCT) (2014)
27. Puchinger, S., Müelich, S., Bossert, M., Hiller, M., Sigl, G.: On error correction for physical unclonable functions. In: International ITG Conference on Systems, Communications and Coding (SCC). IEEE (2015)
28. Chen, B., Ignatenko, T., Willems, F.M.J., Maes, R., van der Sluis, E., Selimis, G.: High-Rate Error Correction Schemes for SRAM-PUFs Based on Polar Codes. Report (2017)
29. Hiller, M.: Key Derivation with Physical Unclonable Functions, Dissertation, Technical University of Munich (2016)
30. Delvaux, J.: Security Analysis of PUF-Based Key Generation and Entity Authentication, Dissertation, KU Leuven (2017)
31. Yu, M.: PUF Constructions with Limited Information Leakage, Dissertation, KU Leuven (2018)
32. Müelich, S.: Channel Coding for Hardware-Intrinsic Security, Dissertation, Ulm University (2019)
33. Immler, V.: Higher-Order Alphabet Physical Unclonable Functions, Dissertation, Technical University of Munich (2019)
34. Karakoyunlu, D., Sunar, B.: Differential template attacks on PUF enabled cryptographic devices. In: IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1–6 (2010)
35. Merli, D., Schuster, D., Stumpf, F., Sigl, G.: Side-channel analysis of PUFs and fuzzy extractors. In: McCune, J.M., Balacheff, B., Perrig, A., Sadeghi, A.-R., Sasse, A., Beres, Y. (eds.) International Conference on Trust and Trustworthy Computing (TRUST), Series LNCS, vol. 6740, pp. 33–47. Springer (2011)
36. Merli, D.: Attacking and Protecting Ring Oscillator Physical Unclonable Functions and Code-Offset Fuzzy Extractors, Dissertation (2014)
37. Helfmeier, C., Boit, C., Nedospasov, D., Seifert, J.-P.: Cloning physically unclonable functions. In: IEEE International Workshop on Hardware-Oriented Security and Trust (HOST) (2013)
38. Tajik, S., Lohrke, H., Ganji, F., Seifert, J.-P., Boit, C.: Laser fault attack on physically unclonable functions. In: Fault Diagnosis and Tolerance in Cryptography Workshop (FDTC). IEEE (2015)
39. Tajik, S.: On the Physical Security of Physically Unclonable Functions, Series T-Labs Series in Telecommunication Services. Springer, Berlin (2019)
40. Boyen, X.: Reusable cryptographic fuzzy extractors. In: ACM Conference on Computer and Communications Security (CCS), pp. 82–91. ACM (2004)
41. Boyen, X., Dodis, Y., Katz, J., Ostrovsky, R., Smith, A.: Secure remote authentication using biometric data. In: Cramer, R. (ed.) Advances in Cryptology (EUROCRYPT), Ser. LNCS, vol. 3494, pp. 147–163. Springer, Heidelberg (2005)
42. Günlü, O., Iscan, O.: DCT based ring oscillator physical unclonable functions. In: IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), pp. 8248–8251 (2014)
43. Günlü, O., Iscan, O., Sidorenko, V., Kramer, G.: Reliable secret-key binding for physical unclonable functions with transform coding. In: IEEE Global Conference on Signal and Information Processing (GlobalSIP), pp. 986–991 (2016)
44. Immler, V., Hennig, M., Kürzinger, L., Sigl, G.: Practical aspects of quantization and tamper-sensitivity for physically obfuscated keys. In: Workshop on Cryptography and Security in Computing Systems (CS2), pp. 13–18. ACM (2016)
45. Immler, V., Uppund, K.: New insights to key derivation for tamper-evident physical unclonable functions. IACR Trans. Cryptogr. Hardw. Embedded Syst. **2019**(3), 30–65 (2019)
46. Maes, R., van der Leest, V., van der Sluis, E., Willems, F.: Secure key generation from biased PUFs: extended version. J. Cryptogr. Eng. **6**(2), 121–137 (2016)
47. Hiller, M., Önalan, A.G.: Hiding secrecy leakage in leaky helper data. In: Fischer, W., Homma, N. (eds.) Conference on Cryptographic Hardware and Embedded Systems, Series LNCS, vol. 10529. Springer, Berlin, pp. 601–619 (2017)
48. Suzuki, M., Ueno, R., Homma, N., Aoki, T.: Multiple-valued debiasing for physically unclonable functions and its application to fuzzy extractors. In: Huss, S., Schindler, W. (eds.) International

Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE), Series LNCS. Springer, Berlin (2017)
49. van Herrewege, A., Katzenbeisser, S., Maes, R., Peeters, R., Sadeghi, A.-R., Verbauwhede, I., Wachsmann, C.: Reverse fuzzy extractors: enabling lightweight mutual authentication for PUF-enabled RFIDs. In: Keromytis, A.D. (ed.) Financial Cryptography and Data Security (FC), Series LNCS, vol. 7397, pp. 374–389. Springer, Berlin (2012)
50. Delvaux, J., Peeters, R., Gu, D., Verbauwhede, I.: A survey on lightweight entity authentication with strong PUFs. ACM Comput. Surv: CSUR **48**(2), 1–42 (2015)
51. Delvaux, J., Gu, D., Schellekens, D., Verbauwhede, I.: Helper data algorithms for PUF-based key generation: overview and analysis. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **34**(6), 889–902 (2015)
52. Armknecht, F., Maes, R., Sadeghi, A.-R., Sunar, B., Tuyls, P.: Memory leakage-resilient encryption based on physically unclonable functions. In: Matsui, M. (ed.) Advances in Cryptology (ASIACRYPT), Series LNCS, vol. 5912, pp. 685–702. Springer, Berlin (2009)
53. Aysu, A., Wang, Y., Schaumont, P., Orshansky, M.: A new maskless debiasing method for lightweigth physical unclonable functions. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 54–59 (2017)
54. Immler, V., Hiller, M., Liu, Q., Lenz, A., Wachter-Zeh, A.: Variable-length bit mapping and error correcting codes for higher-order alphabet pufs. In: Danger, J.-L., Ali, S.S., Eisenbarth, T. (eds.) International Conference on Security, Privacy, and Applied Cryptography Engineering (SPACE), Series LNCS, pp. 190–209. Springer, Berlin (2017)
55. Müelich, S., Bossert, M.: A new error correction scheme for physical unclonable functions. In: International ITG Conference on Systems, Communications and Coding (SCC) (2017)
56. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland, Amsterdam (1977)
57. Hiller, M., Pehl, M., Kramer, G., Sigl, G.: Algebraic security analysis of key generation with physical unclonable functions. In: Security Proofs for Embedded Systems Workshop (PROOFS) (2016)
58. Juels, A., Wattenberg, M.: A fuzzy commitment scheme. In: ACM Conference on Computer and Communications Security (CCS), pp. 28–36 (1999)
59. Tuyls, P., Akkermans, A.H.M., Kevenaar, T.A.M., Schrijen, G.-J., Bazen, A.M., Veldhuis, R.N.J.: Practical biometric authentication with template protection. In: Kanade, T., Jain, A., Ratha, N. (eds.) Audio- and Video-Based Biometric Person Authentication (AVBPA), Series LNCS, vol. 3546, pp. 436–446. Springer, Berlin (2005)
60. Fuller, B., Meng, X., Reyzin, L.: Computational fuzzy extractors. In: Sako, K., Sarkar, P. (eds.) Advances in Cryptology (ASIACRYPT), Series LNCS, vol. 8269, pp. 174–193. Springer, Berlin (2013)
61. Blum, A., Kalai, A., Wasserman, H.: Noise-tolerant learning, the parity problem, and the statistical query model. J. ACM **50**(4), 506–519 (2003)
62. Herder, C., Ren, L., van Dijk, M., Yu, M.-D.M., Devadas, S.: Trapdoor computational fuzzy extractors and stateless cryptographically-secure physical unclonable functions. IEEE Trans. Dependable and Secure Comput. **14**, 65–82 (2016)
63. Huth, C., Becker, D., Guajardo, J., Duplys, P., Güneysu, T.: Securing systems with scarce entropy: LWE-based lossless computational fuzzy extractor for the IoT. IACR eprint Archive (2016)
64. Bösch, C.: Efficient Fuzzy Extractors for Reconfigurable Hardware, Master's Thesis, Ruhr-University Bochum (2008)
65. van der Leest, V., Preneel, B., van der Sluis, E.: Soft decision error correction for compact memory-based PUFs using a single enrollment. In: Prouff, E., Schaumont, P. (Eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 7428, pp. 268–282. Springer, Heidelberg (2012)
66. Hiller, M., Kürzinger, L., Sigl, G., Müelich, S., Puchinger, S., Bossert, M.: Low-area Reed decoding in a generalized concatenated code construction for PUFs. In: IEEE Computer Society Annual Symposium on VLSI (ISVLSI) (2015)
67. Bennett, C.H., Brassard, G., Crepeau, C., Skubiszewska, M.-H.: Practical quantum oblivious transfer. In: Feigenbaum, J. (Ed.) Advances in Cryptology (CRYPTO), Series LNCS, vol. 576, pp. 351–366. Springer, Berlin (1992)
68. Suh, G.E.: AEGIS: a single-chip secure processor, Dissertation, Massachusetts Institute of Technology (2005)
69. Davida, G.I., Frankel, Y., Matt, B.J.: On enabling secure applications through off-line biometric identification. In: IEEE Symposium on Security and Privacy (S&P), pp. 148–157 (1998)
70. Delvaux, J., Gu, D., Verbauwhede, I., Hiller, M., Yu, M.-D.M.: Secure sketch metamorphosis: tight unified bounds. IACR eprint Archive (2015)
71. Kang, H., Hori, Y., Katashita, T., Hagiwara, M., Iwamura, K.: Cryptographic key generation from PUF data using efficient fuzzy extractors. In: International Conference on Advanced Communication Technology (ICACT), pp. 23–26. IEEE (2014)
72. Yu, M., M'Raihi, D., Sowell, R., Devadas, S.: Lightweight and secure PUF key storage using limits of machine learning. In: Preneel, B., Takagi, T. (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 6917, pp. 358–373. Springer, Heidelberg (2011)
73. Yu, M., Sowell, R., Singh, A., M'Raihi, D., Devadas, S.: Performance metrics and empirical results of a PUF cryptographic key generation ASIC. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 108–115 (2012)
74. Becker, G.T., Wild, A., Güneysu, T.: Security analysis of index-based syndrome coding for PUF-based key generation. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST) (2015)
75. Golomb, S.W.: Run-length encodings (corresp.). IEEE Trans. Inf. Theory **12**(3), 399–401 (1966)
76. Bossert, M.: Channel Coding for Telecommunications. Wiley, New York (1999)
77. Van Herrewege, A., Verbauwhede, I.: Tiny application-specific programmable processor for BCH decoding. In: IEEE International Symposium on System on Chip (SoC), pp. 1–4 (2012)
78. Hiller, M., Rodrigues Lima, L., Sigl, G.: Seesaw: an area optimized FPGA Viterbi decoder for PUFs. In: Euromicro Conference on Digital System Design (DSD). IEEE (2014)
79. Hofer, M., Böhm, C.: An alternative to error correction for SRAM-like PUFs. In: Mangard, S., Standaert, F.-X., (eds.) Workshop on Cryptographic Hardware and Embedded Systems (CHES), Series LNCS, vol. 6225, pp. 335–350. Springer, Berlin (2010)
80. Jungk, B., Rodrigues Lima, L., Hiller, M.: A systematic study of lightweight hash functions on FPGAs. In: International Conference on Reconfigurable Computing and FPGAs (ReConFig). IEEE (2014)
81. Bogdanov, A., Knezevic, M., Leander, G., Toz, D., Varici, K., Verbauwhede, I.: SPONGENT: the design space of lightweight cryptographic hashing. IEEE Trans. Comput. **62**(10), 2041–2053 (2013)
82. Krawczyk, H.: LFSR-based hashing and authentication. In: Desmedt, Y. (ed.) Advances in Cryptology (CRYPTO), Series LNCS, vol. 839, pp. 129–139. Springer, Berlin (1994)
83. Solomon, G.: Golay encoding/decoding via BCH-Hamming. Comput. Math. Appl. **39**(11), 103–108 (2000)
84. Puchinger, S., Müelich, S., Wachter-Zeh, A., Bossert, M.: Timing attack resilient decoding algorithms for physical unclonable

functions. In: International ITG Conference on Systems, Communications and Coding (SCC) (2017) **(to appear)**
85. Hiller, M., Önalan, A.G., Sigl, G., Bossert, M.: Online reliability testing for PUF key derivation. In: International Workshop on Trustworthy Embedded Devices (TrustED), pp. 15–22. ACM (2016)
86. Immler, V., Obermaier, J., König, M., Hiller, M., Sigl, G.: B-TREPID: batteryless tamper-resistant envelope with a PUF and integrity detection. In: IEEE International Symposium on Hardware Oriented Security and Trust (HOST), pp. 49–56 (2018)
87. Schnabl, G., Bossert, M.: Soft-decision decoding of Reed–Muller codes as generalized multiple concatenated codes. IEEE Trans. Inf. Theory **41**(1), 304–308 (1995)
88. Leyh, J.: Lightweight BCH decoder architectures for PUF-based key generation, Bachelor's Thesis, Technical University of Munich (2015)

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.