Modular PUF Coding Chain with High-Speed Reed-Muller Decoder

Holger Mandry*, Andreas Herkle*, Ludwig Kürzinger†, Sven Müelich⋆,
Joachim Becker*, Robert F.H. Fischer⋆ and Maurits Ortmanns*
*Institute of Microelectronics, ⋆Institute of Communications Engineering, Ulm University,
Albert-Einstein-Allee 43, Ulm, Germany, holger.mandry@uni-ulm.de
†Institute for Human-Machine Communication, Technical University Munich, Germany, ludwig.kuerzinger@tum.de

***Abstract*—Physical Unclonable Functions (PUFs) offer the possibility to produce unique fingerprints for integrated circuits. As raw PUF responses are affected by noise, some post-processing steps are necessary. We present a coding chain test framework for PUFs on Field Programmable Gate Arrays. The framework allows easy exchange, evaluation and comparison of different PUF implementations, coding algorithms and other chain modules. For a testing framework, the execution time of the evaluated algorithm is a bottleneck, since a huge amount of runs are supposed to be done. Hence, we additionally present a new type of Reed-Muller decoder hardware architecture using parallel modules to speed up the decoding process. The decoding time could be decreased by 95 % in comparison to existing implementations at the cost of 41 times higher slice count.**

***Index Terms*—Physical Unclonable Function (PUF), FPGA, generalized concatenated codes, Reed-Muller**

I. Introduction

With the increasing spread of Internet-Of-Things (IoT) based devices during the last years additional gateways to the infiltration of personal privacy were opened. Therefore authentication and secure communication became a more relevant issue and hardware embedded security got a significant boost. Especially in hardware cryptography, secret keys are required, which should neither be accessible nor modifiable as they can not be exchanged in-field. So called Physical Unclonable Functions (PUFs) offer the ability to generate individual fingerprints of hardware components without storing them. PUFs utilize small manufacturing variations to produce *unpredictable but repeatable* responses to a given challenge, which usually are bitstrings of given length. Additionally, PUFs are *easy to evaluate* and *hard to characterize* [1]. This means, that a response must be created very fast and it should be impossible to calculate the function with a given number of known challenge-response-pairs.

For Field Programmable Gate Arrays (FPGAs), different implementations have been proposed, e.g. the ringoscillator (RO) PUF, which was proven to be the best implementation regarding noise immunity so far [2]. Environmental influences can affect the *reproducibility* [3] of PUFs such that their response might be altered over multiple measurements. Several PUF specific error correcting techniques have already been published [4], [5], [6], [7] to correct for this. However, only very few publications take a whole system implementation into consideration.

Therefore, we present a modular test framework for FPGAs to allow the comparison of existing and new PUF FPGA implementations and algorithms as a whole. In the framework, the creation of a PUF response and post-processing like source coding and error correction are combined to a chain-like structure. A modular approach allows an easy exchange of specific modules and their evaluation without the need to adapt the remaining system. An additional Central Processing Unit (CPU) is available that allows to analyze and debug intermediate steps of the processing.

The complete PUF coding chain is described in sec. II. Different recursive Reed-Muller ($\mathcal{RM}$) decoder implementations are presented and a new one proposed in sec. III, serving as examples of interchangeable modules-under-test. In sec. IV, we compare these to an existing decoder [7] and highlight the tradeoffs regarding processing time and area consumption on the FPGA. Section V finally concludes the paper.

II. Modular PUF Coding Chain

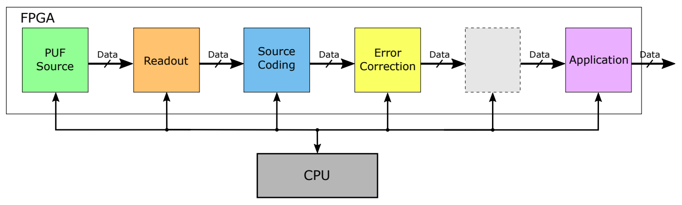


Fig. 1: Modules of the coding chain.

In most cases a raw PUF response from the original source of randomness needs some post-processing steps to be usable by an application. A full PUF coding chain was presented as PUFKY [5], where the RO PUF responses were encoded with a Lehmer-Gray Code followed by error correction using secure sketch and stored helper data. PUFKY was kept modular to allow quick adaptations to surrounding circuits using a bus interface. Even though, the design itself is compact and highly optimized for its specific application, unfortunately this advantage also makes testing of different algorithms a challenging task as all systems modules and connections have to be adapted. In order to tackle this gap in the state of the art and to be able to test various PUF sources with various coding algorithms, we designed a flexible and modular framework for testing on FPGAs. The test-framework is divided into the following sub-modules: *PUF-Source*, *readout*, *source coding*, *error correction* and *application*, as illustrated in Fig. 1. Each module has a defined and mostly similar interface to the previous and following one, which allows the modules to interchange data via a defined and module independent handshake protocol. This generic setup makes it possible to exchange the algorithms of each module while leaving the remaining system untouched and even allows additional modules to be integrated as indicated in the grayish box in Fig. 1.

The platform we used is the System-on-Chip (SoC) *Xilinx ZYNQ 7000 XC7Z010*. Every module is bidirectionally connected to the CPU not only for debugging and analysis: by replacing a module with a redirect placeholder, its functionality can be provided by a software function running on the CPU, thus speeding up prototyping and hardware-software co-design. Obviously this offends against the principle of a secure system, where no secret information can leave the circuit, however our system is designed as a test platform to compare and evaluate different algorithms before an isolated and dedicated system is designed.

The first module of the PUF coding chain is the *PUF-Source* itself, which contains the physical components generating the random signals. On FPGAs this components are mostly flip flops (FFs) and look-up tables (LUTs) that provide boolean functions. In our exemplary implementation, 80 ROs were placed. The next adjacent module *readout* is responsible to process the PUF signals and convert them into digital values. Our example entity uses DSP48 based counters, which count the number of rising clock edges in a given time interval. A bit is then generated in the source coding module by 2nd order quantization [8]: two selected counter values are subtracted from each other, giving a bit value of 0 if the result is positive and a bit value of 1 if negative. These digital values can further be compressed and mapped to codewords. To remove the influence of circuit noise and other non-idealities of the reproducibility of the response, the subsequent module applies *error correction*. In our exemplary implementation, a fuzzy extractor with generalized concatenated codes as described in [6] is used. By using the helper data and the corrected key, which has an error probability of $1.49 \cdot 10^{-9}$ [9], it is possible to restore the original PUF response. For this decoding algorithm, an implementation for low-area constraints was published [7]. With the help of our framework, we further explore the parameter space of speed and area and present our different implementations in-depth in sec. III. After the error correction some additional modules can be inserted, e.g. for *privacy amplification* or a *hash function* which are not explicitly shown in Fig. 1 and their discussion is omitted in the following. As a last step in the coding chain the application module can use the post-processed PUF response.

III. Various types of a Plotkin Reed-Muller Decoder

Reed-Muller codes are a class of codes suited for PUF bit error correction as decoding is easily implementable [6]. In this section, we will show how Reed-Muller ($\mathcal{RM}$) codes, in the following called super-codes, are constructed from sub-codes via the Plotkin construction. We will show how this approach perfectly fits the requirement for parallel instantiation in FPGAs, and present one stacked based and two new recursive implementations of a decoder in hardware and highlight their benefits and trade-offs.

$\mathcal{RM}(r,m)$ codes are characterized by the two parameters $m > 0$ and $0 \leq r < m$. The length of the created binary linear code is $n = 2^m$. With the Plotkin construction it is possible to create a super-code out of the sub-codes [10] as follows:

$$\mathcal{RM}(r,m) = \{(u|u \oplus v) : u \in \mathcal{RM}(r, m\text{-}1), \quad v \in \mathcal{RM}(r\text{-}1, m\text{-}1)\} \tag{1}$$

$\mathcal{RM}$ codes with $r = 0$ are repetition codes and with $r = m-1$ are parity-check codes, which are both easy to decode. This recursive code construction allows it to see $\mathcal{RM}$ codes as generalized concatenated codes [11]. In order to decode a codeword y and correct errors e, three recursive $\mathcal{RM}$ decoding steps have to be made as shown in Algorithm 1 [9].

Algorithm 1 Recursive $\mathcal{RM}(r,m)$ Decoder

```
Input: $y = (y_u|y_v) = (u \oplus e_u|u \oplus v \oplus e_v)$
1: $\hat{v}$ = decode $\tilde{v} = y_u \oplus y_v$ in $\mathcal{RM}(r\text{-}1, m\text{-}1)$
2: $\hat{u}_1$ = decode $\tilde{u}_1 = y_v \oplus \hat{v}$ in $\mathcal{RM}(r, m\text{-}1)$
3: $\hat{u}_2$ = decode $\tilde{u}_2 = y_u$ in $\mathcal{RM}(r, m\text{-}1)$
4: Find $i \in \{1, 2\}$ so that Hamming distance
   $d_H(y, (\hat{u}_i|\hat{u}_i \oplus \hat{v}))$ is minimal
5: return $(\hat{u}_i|\hat{u}_i \oplus \hat{v})$
```

Recursive functions are very easy to implement in software. The main challenges for recursive algorithms in hardware are the unknown depth of recursion as well as the different length of resulting codewords. This makes it very hard to develop an efficient general hardware setup. Nevertheless we designed three different variants of the Plotkin $\mathcal{RM}$ decoder in hardware which are explained in the following.

a) Fully Recursive Decoder: The approach of the fully recursive decoder is to decode all three sub-codewords in parallel. The Steps 1 and 3 of Algorithm 1 only depend on the input y and can start immediately. Step 2 needs the result $\hat{v}$ from Step 1 but is completely independent of Step 3, thus Step 2 and 3 can also run in parallel. Consequently, the fully recursive decoder consists of many sub-decoders that are instantiated recursively. Depending of the value of r and m, either a repetition decoder, a parity-check decoder or a combination module with individual bit width is created. A combination module instantiates three additional recursive decoders with r and m of the required sub-codes and combines their corrected sub-codewords $\hat{v}, \hat{u}_1$ and $\hat{u}_2$. $\tilde{u}_1$ and $\tilde{u}_2$ are decoded with one set of sub-decoders each to allow real parallel decoding. Because the bit width of each sub-module is known, the combination can be done by wiring and no shifting is necessary. However this setup requires many hardware components as a huge amount of adapted decoders are instantiated.

b) Stack Decoder: The stack based decoder uses Block-RAMs (BRAMs) to save data in a last in, first out queue, which is called stack. If the current r and m values do not correspond to repetition or parity-check codes, all current values and intermediate results are saved on the stack. Next new sub-codeword and r, m values are calculated as described in Algorithm 1. Then the sub-code can start to decode. After the combination of all three results, the stack pointer is decreased so that the super-code can continue its decoding. $\mathcal{RM}(0,m)$ and $\mathcal{RM}(m\text{-}1,m)$ codes are decoded by specific repetition and parity-check decoders. When the stack pointer is zero again, the decoding process is finished. Due to the fact, that the bit width of the codewords depends on the parameter m the sub-codewords have to be shifted before combination. A simple wiring solution is not possible since all occurring bit widths are handled on the same hardware components. This needs extra time, which increases with m. Additionally, the three decoding steps must be calculated in series. Furthermore, loading and storing of intermediate results from and into BRAMs needs extra time.

c) Serial Recursive Decoder: The serial recursive decoder is a hybrid of serial and parallel decoding. As the decoding Step 2 can not start until Step 1 is finished, we decode $\tilde{u}_1$ and $\tilde{u}_2$ in series on the same hardware. Steps 3 and 1 run in parallel as in the fully recursive decoder to keep the time benefit. The serial decoding of $\tilde{u}_i$ allows us to reduce the used amounts of hardware components. In the fully recursive decoder two identical sub-decoder sets for $\tilde{u}_1$ and $\tilde{u}_2$ exist. By using serial decoding, it is possible to save one complete set of sub-decoders, which should decrease the required area significantly.

IV. Comparison of the Different Decoders

In this section, we analyze the quality of these three different types of decoders by means of decoding time and required area on the FPGA.

a) Decoding Time: To analyze the decoding time Δt for one $\mathcal{RM}$ decoding, all three decoders had to decode the same test sequences. This was done for all possible $\mathcal{RM}$ codes with $m \in \{1, \ldots, 7\}$. Fig. 2 shows exemplarily the decoding time Δt for $m = 5$ and increasing r. As can be seen, the decoding time Δt for $r = 0$ and $r = 4$ are very similar. This is an expected behavior as for both r, only repetition and parity-check codes are directly used. The stack based variation just needs some more clock cycles because it has to load the corresponding bit widths beforehand.

For the other $\mathcal{RM}$ codes, the difference in decoding time between stack and recursive decoders is significantly larger. Here, many sub-decoding steps have to be calculated, which the stack decoder can not run in parallel. Additionally, it needs time to load and store values to the BRAMs for each decoding step and the shift operations take more time.

If we compare the decoding time of $\mathcal{RM}(1,5)$ with $\mathcal{RM}(3,5)$, we can see that the $\mathcal{RM}(1,5)$ needs a lot longer using the stack decoder. This can be explained since the amount of sub-steps in $\mathcal{RM}(r,m\text{-}1)$ is higher than the amount of sub-steps in $\mathcal{RM}(r\text{-}1,m\text{-}1)$. Due to the fact that $\tilde{u}_1$ and $\tilde{u}_2$ both are decoded in $\mathcal{RM}(r,m\text{-}1)$, the bare amount of decoding steps is a lot bigger. Therefore super-codes with low r need more time compared to super-codes with the same amount of sub-codes but bigger r. This performance is also noticeable for the serial recursive decoder.

Only for the fully recursive decoder, the amount of decoding steps in the $\mathcal{RM}(r,m\text{-}1)$ does not impact the execution time, because they are calculated in parallel. Therefore, decoding of super-codes with different r and same amount of sub-codes, as e.g. $\mathcal{RM}(1,5)$ and $\mathcal{RM}(3,5)$, need the same time.

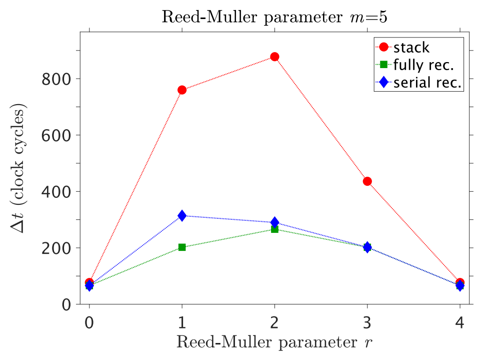
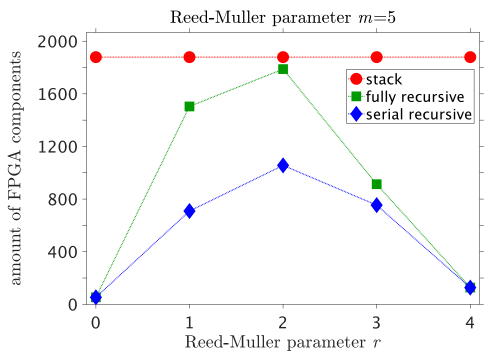
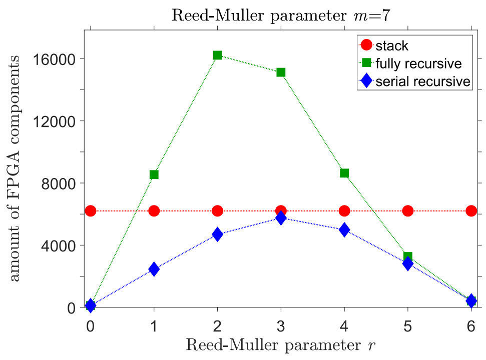


Fig. 2: Average calculation time Δt of all three $\mathcal{RM}$ decoders with $m = 5$ over r.



(a)



(b)

Fig. 3: Required LUTs and FFs of all three $\mathcal{RM}$ decoders with $m = 5$ (a), $m = 7$ (b) over r.

b) Area: The second criteria for the applicability of a decoder, especially on a resource limited device, is the amount of required hardware components, particularly LUTs and FFs. Fig. 3 shows the synthesis results of all different $\mathcal{RM}$ decoders with $m \in \{5, 7\}$ and $r \in \{0, \dots, m\text{-}1\}$.

The stack decoder requires the same amount of components for all possible values of r, because an increasing r has no influence on the bit width. Conversely, the recursive decoders' area consumption always depends on r. For $m \leq 5$, the stack decoder needs more components than the recursive decoders but this relation begins to change for $m = 7$.

c) Time vs Area: In Fig. 4, the relation of area vs time is plotted on a logarithmic scale for the three analyzed implementation variants, as well as the *low area* implementation from [7] and a comparison to a CPU implementation. The *low area* decoder works on the error correcting principle described in [6] as well but uses Reed decoding. Reed decoding is a very memory and therefore area efficient strategy to decode $\mathcal{RM}$ codes [12]. Apart from that it is very time-consuming as every bit is calculated separately. For the CPU implementation, a bare metal C code with 50.6 kB code size was written. This software algorithm is rather slow since a CPU is not adapted to a specific task. On the other hand it requires only few FPGA components to access the readout values.

As can bee seen, there is a clear trade-off between area and speed for recursive implementations, which follows a regression curve, whereby area consumption and decoding time are linked exponentially. The stack based variant is always outperformed by the recursive variants. Although it seems that the stack decoder is a bad choice, one have to keep in mind that it can also decode all $\mathcal{RM}(r', m')$ codes with $m' \leq m$, while the recursive variants are bound to a specific $\mathcal{RM}(r, m)$.

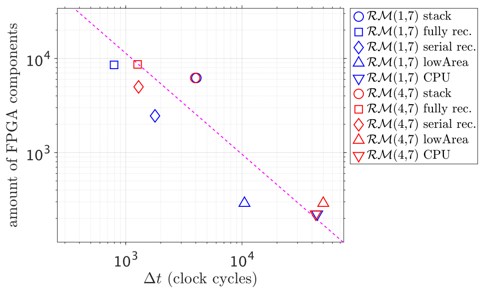


Fig. 4: Calculation time vs required area with logarithmic scale on both axes for $\mathcal{RM}(1, 7)$ (blue) and $\mathcal{RM}(4, 7)$ (red).

Table I lists all criteria of all implementations for a complete error correction, which consists of one $\mathcal{RM}(4, 7)$ and four $\mathcal{RM}(1, 7)$ decoding steps [7]. The fully recursive decoder is 20.7 times faster than the *low area* decoder from [7], which is the slowest variant. On the other hand, it consumes 59.2 times more components of the FPGA. The serial recursive decoder only requires 25.6 times more components and is still 10.9 times faster. It only needs 1.2 times more components than the stack decoder and is 2.3 times faster. In conclusion, the serial recursive decoders has a very good trade-off between area and speed for fixed $\mathcal{RM}(r, m)$ codes.

Especially, for a testing platform as the one presented, execution time of the error correction is highly needed, since trillions of readouts are supposed to be done for code and BER testing. With a clock of $50\,MHz$, the proposed serial recursive decoder only consumes $169\,\mu s$ per decoding, while the low area implementation of [7] would need $1840\,\mu s$. Obviously, the speed advantage of the proposed algorithm is of utmost importance for an evaluation setup as proposed.

TABLE I: Required FPGA components and calculation time for used error correction.

	stack	fully rec.	serial rec.	low Area
LUTs $\mathcal{RM}(1, 7)$	3256	4468	1388	184
LUTs $\mathcal{RM}(4, 7)$	-	4258	2578	-
LUTs total	3256	8726	3966	184
FFs $\mathcal{RM}(1, 7)$	2950	4072	1068	106
FFs $\mathcal{RM}(4, 7)$	-	4378	2411	-
FFs total	2950	8450	3479	106
components[a]	6206	17176	7445	290
slices total[b]	982	3011	1338	73
Δt (clock cycles)				
$\mathcal{RM}(1, 7)$	4054	794	1786	10500
$\mathcal{RM}(4, 7)$	3955	1266	1290	50000
total	20171	4442	8434	92000

[a]after synthesis
[b]after implementation

V. Conclusion

A modular and easily adaptable test framework for creation and post-processing of PUF responses is presented, in which sub-modules can easily be exchanged for testing and comparing of different implementations both in hardware and in software. A connection to the CPU on the same chip was used to analyze intermediate results like bit-error calculation and efficiency testing of error-correction algorithms.

Based on this framework, we presented implementation types of Reed-Muller decoders, with our recursive instantiated decoders representing a new type of hardware setup compared to existing technologies. We showed that our recursive decoders are much faster than stack based ones at the cost of more area of the FPGA. Additionally, we showed that a serial decoding of $\tilde{u}_i$ increases the decoding time only a little but reduces the necessary components significantly, which is the sweet spot for a time-area trade-off. We showed that the stack based variant performs slower than its recursive counterparts yet is the most flexible one as it can also decode $\mathcal{RM}$ codes with smaller m and every valid r. In future work, our test framework can be used to test additional enhancements of PUF post processing.

Acknowledgment

This work was funded by the German National Science Foundation DFG under grant number FI 982/15-1.
We acknowledge Prof. G. Sigl and Dr. M. Hiller from TU Munich for providing us the Python code of their RM decoder.

References

[1] Blaise Gassend, Dwaine Clarke, Marten Van Dijk, and Srinivas Devadas. Silicon physical random functions. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 148–160. ACM, 2002.

[2] Alexander Wild, Georg T. Becker, and Tim Güneysu. A fair and comprehensive large-scale analysis of oscillation-based PUFs for FPGAs. In *27th International Conference on Field Programmable Logic and Applications (FPL), 2017*, pages 1–7. IEEE, 2017.

[3] Roel Maes. Physically unclonable functions: Constructions, properties and applications. 2012.

[4] Dominik Merli, Dieter Schuster, Frederic Stumpf, and Georg Sigl. Side-channel analysis of PUFs and fuzzy extractors. In *International Conference on Trust and Trustworthy Computing*, pages 33–47. Springer, 2011.

[5] Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede. PUFKY: A fully functional PUF-based cryptographic key generator. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 302–319. Springer, 2012.

[6] Sven Müelich, Sven Puchinger, Martin Bossert, Matthias Hiller, and Georg Sigl. Error correction for physical unclonable functions using generalized concatenated codes. In *Fourteenth International Workshop on Algebraic and Combinatorial Coding Theory September , 2014, Svetlogorsk (Kaliningrad region), Russia*, pages 253–258, 2014.

[7] Matthias Hiller, Ludwig Kürzinger, Georg Sigl, Sven Müelich, Sven Puchinger, and Martin Bossert. Low-area reed decoding in a generalized concatenated code construction for PUFs. In *IEEE Computer Society Annual Symposium on VLSI (ISVLSI), 2015*, pages 143–148. IEEE, 2015.

[8] Vincent Immler, Matthias Hiller, Johannes Obermaier, and Georg Sigl. Take a moment and have some t: Hypothesis testing on raw PUF data. In *IEEE International Symposium on Hardware Oriented Security and Trust (HOST), 2017*, pages 128–129. IEEE, 2017.

[9] Sven Puchinger, Sven Müelich, Martin Bossert, Matthias Hiller, and Georg Sigl. On error correction for physical unclonable functions. In *SCC 2015; 10th International ITG Conference on Systems, Communications and Coding; Proceedings of*, pages 1–6. VDE, 2015.

[10] Martin Bossert. *Einführung in die Nachrichtentechnik*. Oldenburg Verlag, 2012.

[11] Martin Bossert. *Kanalcodierung*. Oldenburg Verlag, 3. Auflage, 2013.

[12] I. S. Reed. A class of multiple-error-correcting codes and the decoding scheme. *IRE Trans. on Inf. Th.*, vol. 4, no. 4, pp. 38–49, 1954.