Efficient Helper Data Key Extractor on FPGAs

Christoph Bösch[1,⋆], Jorge Guajardo[2], Ahmad-Reza Sadeghi[1],
Jamshid Shokrollahi[1,⋆⋆], and Pim Tuyls[2]

[1] Horst-Görtz-Institute for IT-Security, Ruhr-University Bochum, Germany
{christoph.boesch,jamshid.shokrollahi,ahmad.sadeghi}@trust.rub.de
[2] Philips Research Europe, Eindhoven, The Netherlands
{jorge.guajardo,pim.tuyls}@philips.com

Abstract. Physical Unclonable Functions (PUFs) have properties that make them very attractive for a variety of security-related applications. Due to their inherent dependency on the physical properties of the device that contains them, they can be used to uniquely bind an application to a particular device for the purpose of IP protection. This is crucial for the protection of FPGA applications against illegal copying and distribution. In order to exploit the physical nature of PUFs for reliable cryptography a so-called helper data algorithm or fuzzy extractor is used to generate cryptographic keys with appropriate entropy from noisy and non-uniform random PUF responses. In this paper we present for the first time efficient implementations of fuzzy extractors on FPGAs where the efficiency is measured in terms of required hardware resources. This fills the gap of the missing building block for a full FPGA IP protection solution. Moreover, in this context we propose new architectures for the decoders of Reed-Muller and Golay codes, and show that our solutions are very attractive from both the area and error correction capability points of view.

Keywords: Physical Unclonable Functions, Intrinsic PUF, Fuzzy Extractor, Helper Data Algorithm, FPGAs, Implementation.

1 Introduction

Virtually all keyed cryptographic primitives, regardless of whether they are based on public-key or private-key cryptography, assume the secrecy of the key used to encrypt/sign a given message. Since the late 90's, there has been a lot of interest in developing methods to guard against key compromise at the protocol level but also at the physical level [1,2,3]. By physical level, we mean mechanisms which can make the platform where cryptographic primitives run (more) secure to key compromise. One of the most interesting of these methodologies is the idea of Physical Unclonable Functions (PUFs) as introduced in [1]. A PUF is a primitive that maps challenges T_i to responses R_i, which are highly dependent on the physical properties of the device in which the PUF is contained or embedded.

[⋆] Part of this work was done while the author was at Philips Research Europe.

[⋆⋆] Current contact address: jamshid.shokrollahi@de.bosch.com

E. Oswald and P. Rohatgi (Eds.): CHES 2008, LNCS 5154, pp. 181–197, 2008.

We will write $R_i \leftarrow \mathsf{PUF}(T_i)$ to denote the response R_i of a PUF to a challenge T_i. Physical Unclonable Functions have essentially two parts: i) a physical part and ii) an operational part. The physical part is a physical system that is very difficult to clone. It inherits its unclonability from uncontrollable process variations during manufacturing. In the case of PUFs on an IC such process variations are typically deep-submicron variations such as doping variations in transistors. The operational part corresponds to the function. In order to turn the physical system into a *function* a set of challenges T_i (stimuli) has to be available to which the system responds with a set of sufficiently different responses R_i. Examples of PUFs include optical PUFs [1], silicon PUFs [4], coating PUFs [2], Intrinsic-PUFs [5], and LC-PUFs [6]. Regardless of their particular instantiation, their unclonability, and tamper evidence properties have made PUFs very useful tools in IP protection and secure key storage applications.

IP Protection For FPGAs. Field Programmable Gate Arrays (FPGAs) are gaining widespread acceptance as substitutes for ASICs in many applications. In fact their re-programmability has made them very attractive in the embedded market, where software and functionality updates can be common and desirable by customers. As a result of this shift, it is increasingly the case that the functionality of an embedded system is presented in the form of a bit configuration file or, in the case of microprocessors, in the form of a program. Thus, the very property that makes FPGAs so attractive (their programmability) also makes it very easy for counterfeiters to copy an IP developer's configuration file and create a similar product without the up-front cost of Intellectual Property (IP) development. This problem was introduced most recently[1] by Simpson and Schaumont [8]. In particular, the authors in [8] showed that by using a PUF on an FPGA they could develop protocols that allow binding of a particular IP to a particular FPGA. Their protocols also allow proving authenticity of the IP to the hardware platform. In [5], the authors further reduce the computation and communication complexity of the protocols in [8] and introduce the idea of Intrinsic-PUFs based on the start-up values of SRAM memory values. Both based their protocols on symmetric-key primitives. In [9], the authors observe that by introducing public-key cryptography, the corresponding private-key does not need to ever leave the FPGA, even during the enrollment stage, thus increasing the security of the overall system. A common characteristic of all PUF-based protocols in [5,8,9] is the derivation of a key(s) from the PUF, which is used to encrypt a piece of IP and authenticate its origin. In the remainder of the paper, we will refer to the encrypting operation for ease of presentation but it is clear that our discussion extends to the computation of Message Authentication Codes (MACs) and/or signatures on a particular IP block.

The Need for a Helper Data Algorithm. Notice that PUF responses are noisy by nature. In other words, two calls to the PUF with the same challenge T_i will produce two different but closely related responses R_i, R_i', where the measure of closeness can be defined via a distance function. We will make the distance

[1] See [7] for earlier references to the problem.

function more explicit in Sect. 2. Intuitively, the distance function should be small among responses originating from the same device and very large for PUF responses originating from different devices. Nevertheless, it is clear that the plain PUF response can not be used as the key, since this would mean that the data encrypted under response R_i could not be decrypted with response R'_i, even if both responses originate from the same PUF embedded in the same device[2]. In order to derive reliable and uniform strings from (imperfect) sources of randomness, such as a PUF, the concept of a fuzzy extractor or helper data algorithm were introduced in [10,11].

Related Work. To our knowledge, there is no previous description of the complexities and design choices made to implement a helper data algorithm on hardware and, more specifically, on FPGAs. In both [12,13] the noisy nature of a PUF is acknowledged. Their solution to the problem is to add an error correcting stage based on BCH codes. Other codes are not considered and no detailed explanation of how to choose the code is given. Gassend [14] also considers the problem of noisy measurements in PUFs by considering Hamming codes and product codes. The solutions based on product codes in [14] is only able to correct up to two errors. Gassend gets around this problem by trying different challenges until the response has a sufficiently small number of errors that they can be solved. It is worth noticing that a similar problem to the one we are considering is present in biometrics. In fact, the first fuzzy extractor construction [15] was aimed at biometric applications. Dodis et al. [11] also describe a software implementation of a fuzzy extractor based on BCH codes. Somewhat related to our construction is the construction of Hao et al. [16] where they implement a two stage error correcting scheme for biometric applications (iris recognition). The scheme in [16] uses first a Hadamard code and then a Reed-Solomon code in a concatenated manner. Notice that the authors in [16] do not consider the hardware implementation of their schemes. In addition, Reed-Solomon codes are optimized for burst errors and thus, are not applicable to our solution since errors present in PUF responses tend to be random.

Our Contributions. In this paper, we focus on the study and implementation of fuzzy extractors on FPGAs, as [5,8,9] assume the existence of such a block but do not provide explicit constructions nor investigate the hardware costs of fuzzy extractors on FPGAs. Our work can be seen as the final block necessary to generate cryptographic keys and, thus, allows for the construction of full IP-protection solutions on FPGAs. We focus on making an efficient choice of code. By efficient, we mean two things. First, we aim to be able to reconstruct the same key with high probability. In other words, given a code we want to achieve an error probability (the probability that an error pattern happens that can not be corrected by the chosen error correcting code) of at least $10^{-6} \approx 2^{-20}$. We argue in Sect. 3.2 that this is a conservative estimate, which can be applied for most applications. In this respect, we show empirically that from the codes considered,

[2] This would only work if $R_i = R'_i$, which in general is highly unlikely.

the best codes (meaning those that can achieve a low error probability) are BCH codes [17,18].

Our second efficiency measure refers to hardware resources. In particular, once we have achieved a certain error probability, we desire that the error correcting decoding algorithm implementation be as area efficient as possible. This, in fact, is a key requirement and makes our work fundamentally different from other helper data algorithm implementations. In particular, the aim of our solution is not the implementation of a helper data algorithm on an FPGA by itself. Rather, our aim is to implement a helper data algorithm in as little hardware as possible and, in the process, allow for the secure deployment of IP. The IP block, in fact, is the one that should determine the FPGA resources, not the helper data algorithm. In our search for an area efficient solution, we turned to different code constructions. We find that concatenation of codes as introduced by Forney [19], allows for the use of codes that are much simpler to implement and possibly more area efficient than BCH codes. In particular, an odd repetition code followed by a Reed-Muller code or a Golay code can satisfy our error probability requirements. We expect that such construction will incur in considerable area savings with respect to a construction based on BCH codes only. We also propose new architectures for the decoders of Reed-Muller and Golay codes, which are of independent interest. In addition, we identify which universal hash function constructions from those already known in the literature are most suitable for small area implementations. These results are described with focus on an implementation results targeting a Spartan-3E Xilinx FPGA, which is a typical FPGA used in low cost applications.

Notation. Algebraically a binary linear code $\mathcal{C}$ with message length k and codeword length n is a k-dimensional subspace of $\mathbb{F}_2^n$. The messages specify each element of the subspace and the codewords are their representations in $\mathbb{F}_2^n$. Given two codewords $\mathbf{v} = (v_1, v_2, \ldots, v_n)$, and $\mathbf{w} = (w_1, w_2, \ldots, w_n)$, with $v_i, w_i \in \mathbb{F}_2$, the Hamming distance between the two words, denoted by d_H, is the number of coordinates in which $\mathbf{v}$ and $\mathbf{w}$ differ. The minimum distance d_{min} of a linear code $\mathcal{C}$ is the smallest Hamming distance between any two different codewords in $\mathcal{C}$. For linear codes the minimum distance is equal to the minimum non-zero weight in $\mathcal{C}$. We write an $[n, k, d]$-code to mean a binary code $\mathcal{C}$ of length n, cardinality 2^k (encoding messages of length k), and minimum distance d. A linear code with minimum distance d has error correcting capability or error correcting distance $t = \left\lfloor \frac{d_{\text{min}}-1}{2} \right\rfloor$. An important data structure related to the linear code is the generator matrix G whose rows are elements of a basis for the linear code. For a binary linear $[n, k, d]$-code, we can write the generator matrix in the standard form as $G = (I_k | P)$, where I_k is the $k \times k$ identity matrix and P is a $k \times (n-k)$ matrix. The parity check matrix is then found as $H = (P^T | I_{n-k})$. This is an $n-k$ by n matrix such that the inner product of any codeword with any column of H equals zero. Encoding a message $\mathbf{m}$ is accomplished by computing $\mathbf{v} = \mathbf{m}G$. The syndrome of a received word $\mathbf{r} = \mathbf{v} + \mathbf{e}$, where $\mathbf{v}$ and $\mathbf{e}$ are a codeword and error, respectively, is defined as $S_r = H\mathbf{r} = H\mathbf{e}$. We refer the reader to [20,21] as standard references for error correcting codes.

2 Helper Data Algorithms

PUF responses can not be used as a key (as in e.g. [2]) in a cryptographic primitive for two reasons. First, PUF responses are obtained through measurements on physical systems, which are typically noisy. This leads to a problem since cryptographic functions are very sensitive to noise on their inputs. Second, PUF responses are not uniformly distributed. Hence, even if there was no noise, the response would not form a cryptographically secure key. In order to deal with both issues a Helper Data Algorithm (HDA) or Fuzzy Extractor or has to be used. In the remainder of this paper, we will use the two terms interchangeably. For the precise definition of a Fuzzy Extractor and Helper Data algorithm we refer to [10,11].

In general a helper data algorithm deals with both issues (noise and non-uniformity of keys) by implementing first an *information reconciliation phase* and second, by applying a *privacy amplification* or randomness extraction primitive. In order to implement those two primitives, helper data W are generated during the *enrollment phase*. During this phase, carried out in a trusted environment, a probabilistic procedure called Gen is run. Later, during the *key reconstruction* or authentication phase, the key is reconstructed based on a noisy measurement R_i' and the helper data W. During this phase, a procedure called Rep is performed. We present one of the constructions for such procedures previously described in [11]. Other constructions as well as constructions for other metrics can be found in [11]. Notice that all constructions have an error correcting stage. Optimizing such stage will be our focus in the next sections.

Construction Based on Code Offset. In order to implement the procedures Gen and Rep an error correction code $\mathcal{C}$ and a set $\mathcal{H}$ of universal hash functions [22] is required. The parameters $[n, k, d]$ of the code $\mathcal{C}$ are determined by the length of the responses R and the number of errors t that have to be corrected. The distance d of the code is chosen such that t errors can be corrected. The Gen-procedure takes as input a PUF response(s) R and produces as output a key K and helper data $W = (W_1, W_2)$, i.e., $(K; W) = (K; (W_1, W_2)) \leftarrow \mathsf{Gen}(R)$. This is achieved as follows. First, a code word $C_S \leftarrow \mathcal{C}$ is chosen at random from $\mathcal{C}$. Then, a first helper data vector equal to $W_1 = C_S \oplus R$ is generated. Furthermore, a hash function h_i is chosen at random from $\mathcal{H}$ and the key K is defined as $K \leftarrow h_i(R)$. The helper data W_2 is set to i. During the key reconstruction phase the procedure Rep is run. It takes as input a noisy response R' from the same PUF and helper data W and reconstructs the key K *i.e.* $K \leftarrow \mathsf{Rep}(R', W)$. This is accomplished according to the following steps: (1) *Information Reconciliation*: Using the helper data W_1, $W_1 \oplus R'$ is computed. Then, the decoding algorithm of $\mathcal{C}$ is used to obtain C_S. From C_S, R is reconstructed as $R = W_1 \oplus C_S$; and (2) *Privacy amplification*: The helper data W_2 is used to choose the correct hash function $h_i \in \mathcal{H}$ and to reconstruct the key as $K = h_i(R)$. Notice that we have implicitly assumed the use of a binary code. The security of the above constructions has been established in [10,11].

3 Searching for Good Linear Codes and Efficient HDAs

We will model the noise present in PUF responses as a binary symmetric channel (BSC). In particular, in a BSC, the bit error probability p_b specifies the probability that a transmitted information bit is received in error. Then, with probability $1 - p_b$ the sent information bit equals the received one. We assume that all bits are independent, which turns out to be a good assumption as shown in [9,5]. In [9], the authors also show that the probability that a string of n bits has more than t errors is given by:

$$P_{total} = \sum_{i=t+1}^{n} \binom{n}{i} p_b^i (1-p_b)^{n-i} = 1 - \sum_{i=0}^{t} \binom{n}{i} p_b^i (1-p_b)^{n-i} \qquad (1)$$

Notice that the value of P_{total} will determine the minimum distance of the code and thus, the size of the code. We argue that a conservative value is $P_{total} \leq 10^{-6}$. To see this, we relate the failure error probability of our system to the error probability of the hardware platform (in this case FPGAs), which is given in terms of the Failure-In-Time (FIT) unit. Given a FIT rate λ, the probability that a failure will occur until time t is given by $P_{\text{Failure until time }t}(t) = 1 - \exp(-\lambda t)$. For both Xilinx and Altera devices, the lowest FIT rate that we found in [23,24] was five for older devices. All device families manufactured in newer process technology have a FIT rate higher than twelve. Assuming conservatively a FIT rate equal to five, for a 15-year period the resulting failure probability is 6.610^{-4}. Thus, it is clear that assuming $P_{total} \leq 10^{-6}$ is quite conservative for any realistic application (i.e. applications that should last more than a month).

3.1 The Naive Approach: Simple Codes

We consider codes which can be used to correct random errors (as opposed to burst errors) in a received word $\mathbf{r}$. Thus, we do not consider Reed-Solomon codes, which have very good burst error correcting capabilities, as well as convolutional codes for similar reasons. We also discarded LDPC codes [25], which are very efficient but require very large and sparse binary matrices, thus making them resource intensive in hardware applications (see e.g. [26,27] for designs targeting FPGAs, which occupy more than 50% of a high-end FPGA). We do not consider Hadamard codes explicitly but notice that they are equivalent to first-order Reed-Muller codes. Similarly, the Hamming code is a $[2^m - 1, 2^m - m - 1, 3]$-code and can therefore correct only one error. Thus, it is not very useful to decode received vectors with high error rates as in the applications we are considering. In order to attain the desired error probability of 10^{-6}, we simply start by looking at what can be achieved via straight forward use of the following codes: repetition code, Reed-Muller codes of the first order[3], binary Golay code, and binary BCH codes. We refer to [28] for extensive tables for different error probabilities. For our particular case, we summarize the relevant code parameters in Table 1. In Table 1,

[3] Reed-Muller codes of higher order offer better error correction performance at a considerable higher cost in decoder complexity. Thus, we do not consider them.

Table 1. Results for different codes with $p_b = 0.15$. A code denoted with a star (*) means it has been shortened.

Code	$[n, k, d]$	$t = \lfloor d/2 \rfloor$	P_{total}	source bits for 171 bits
Repetition	$[33, 1, 33]$	16	1.0010^{-6}	5643
Reed-Muller	$[256, 9, 128]$	63	2.0410^{-5}	4864
Reed-Muller	$[512, 10, 256]$	127	2.5410^{-9}	9216
Golay	$[23, 12, 7]$	3	0.4604	345
BCH	$[511, 19, 239]$	119	2.9710^{-7}	4599
BCH	$[1023, 46, 439]$	219	1.8510^{-8}	4092
BCH	$[1020, 43, 439]^*$	219	1.4410^{-8}	**4080**

the last column refers to the number of SRAM source bits required to obtain 171 "error-free" bits which can then be hashed to obtain 128 random and uniformly distributed bits. This is based on the amount of entropy in SRAM PUFs reported in [5]. Then, the last column in Table 1 can be computed as $n\lceil 171/k \rceil$, where n and k, refer to the code parameters. We observe that in this table, we have only considered the code as stated in the first column or a shortened version of it in the case of the $[1020, 43, 439]$-BCH code. Shortening a code[4] is one of many different code modifications and one which we found useful empirically. We also notice that the number of errors that a shortened code can correct is at least t. However, correcting the additional error patterns enabled by the code shortening results in additional decoder complexity. Notice that neither the Golay code nor the $[256, 9, 128]$-Reed-Muller code provide our desired error probability and that the BCH codes provide a very low error rate. Thus, the question that we ask is if we can extract a 128-bit cryptographically secure key with less than 4080 bits of SRAM.

3.2 A New Construction Based on Concatenated Codes

Notice that the previous scheme has several disadvantages. First, a scheme based on the repetition code alone, although low complexity, requires more than 5000 bits of SRAM. Second, although we have not discussed the complexity of the decoders yet, BCH decoder algorithms are very complex and, thus, we expect it to be expensive in terms of area. Thus, in this section we propose a new scheme based on concatenation of two error correcting codes $\mathcal{C}_1$ and $\mathcal{C}_2$ [19] to get less complexity while achieving the same or comparable error probabilities. Given two codes $\mathcal{C}_1$ and $\mathcal{C}_2$ with parameters $[n_1, k_1, d_1]$ and $[n_2, k_2, d_2]$ respectively, the concatenated code $\mathcal{C}_c$ is a $[n_c, k_c, d_c]$-code with $n_c = n_1 n_2, k_c = k_1 k_2$ and $d_c = d_1 d_2$. Notice that very long codes can be constructed from considerably shorter ones. Furthermore, a concatenated code can correct (depending on construction) random and burst errors simultaneously and the decoding complexity of two short codes is lower than the complexity of the entire code. Based

[4] One way to shorten an $[n, k, d]$-code, is to set i information symbols to zero to obtain an $[n - i, k - i, d]$-code.

Algorithm 1. Gen_c Algorithm for Concatenated Codes

```
Require: An $[n_1, k_1, d_1]$-code $\mathcal{C}_1$, an $[n_2, k_2, d_2]$-code $\mathcal{C}_2$, a family $\mathcal{H}$ of universal hash functions, and a PUF response $R$ of size $s_R = l_2 \cdot n_2$ bits, where $l_2 = \lceil \frac{l_1 \cdot n_1}{k_2} \rceil$ and $l_1 = \lceil \frac{s_K}{k_1} \rceil$ or $l_1 = \lceil \frac{s_K}{k_1} \rceil + 1$.
Ensure: Helper data $(W_1, W_2)$ and a key $K$ of size $s_K$
Set $l_1 \leftarrow \lceil \frac{s_K}{k_1} \rceil$ if $k_2$ divides $l_1 \cdot n_1$, otherwise $l_1 \leftarrow \lceil \frac{s_K}{k_1} \rceil + 1$
Generate uniformly at random code words $\mathbf{v_{1i}}$ from $\mathcal{C}_1$, for $i = 1, 2, \ldots, l_1$.
Form the string $u$ by concatenating the binary representation of $\mathbf{v_{1i}}$ for $i = l_1, \ldots, 2, 1$. At the end $u = (u_1, u_2, \cdots, u_{l_u})$, where $u_i \in \{0,1\}$ and $l_u = l_1 \cdot n_1$.
Set $l_2 \leftarrow \lceil \frac{l_u}{k_2} \rceil$. If $k_2$ does not divide $l_u$, extend $u$ by adding $l_2 k_2 - l_u$ zero bits to it. The resulting string $u'$ is of size $l_2 \cdot k_2$ bits.
Write $u' = (U'_1, U'_2, \ldots, U'_{l_2})$, where $U'_i$ are words of size $k_2$ bits.
For $i = 1, 2, \ldots, l_2$, compute $\mathbf{v_{2i}} \leftarrow \mathsf{Encode}_{\mathcal{C}_2}(U'_i)$, where $\mathsf{Encode}_{\mathcal{C}_2}$ is the encoding algorithm for the code $\mathcal{C}_2$.
Form the string $w$ by concatenating the binary representation of $\mathbf{v_{2i}}$ for $i = 1, 2, \ldots, l_2$. At the end $w = (w_1, w_2, \cdots, w_{l_w})$, where $w_i \in \{0,1\}$ and $l_w = l_2 \cdot n_2$.
Set $W_1 \leftarrow w \oplus R$, where $\oplus$ is the bitwise logical XOR operation.
Choose a random hash function $h_i \in \mathcal{H}$
Set $W_2 \leftarrow i$
Set $R_K$ to the first $(l_2 - 1) \cdot n_2$ bits of $R$. This essentially ignores $n_2$ bits of the response $R$.
Set $K \leftarrow h_i(R_K)$
```

on our discussion on fuzzy extractors in Sect. 2, our new constructions for the procedures Gen and Rep are described in Algorithms 1 and 2, respectively.

Before considering specific examples, we discuss what the error probability will be for our concatenated codes. Intuitively, the main idea of our construction is to first use a rather simple code, let's say $\mathcal{C}_2$ to bring the number of errors down. Then, with the second code, $\mathcal{C}_1$, the remaining errors are corrected. By a clever choice of the first code also the second code, has to correct only a few errors making the scheme more efficient. A similar idea is presented in [29] as a way to cope with extremely noisy channels. In particular, [29] reduced the error probability by using a repetition code and then combine the first stage with a more powerful code. For example, a simple calculation with (1) will demonstrate that just using the $[3, 1, 3]$ repetition code, it is possible to bring the error probability from 15% to 6%. In general, the resulting error probability can be estimated as follows. Given two codes $\mathcal{C}_1, \mathcal{C}_2$ with parameters $[n_1, k_1, d_1; t_1 = \lfloor (d_1 - 1)/2 \rfloor]$ and $[n_2, k_2, d_2; t_2 = \lfloor (d_2 - 1)/2 \rfloor]$, respectively, the $\mathsf{Rep_c}$-procedure will first decode with $\mathsf{Decode}_{\mathcal{C}_2}$ and the result will be decoded with $\mathsf{Decode}_{\mathcal{C}_1}$. Thus, the error probabilities P_2 and P_1 after decoding with the decoding algorithms of $\mathcal{C}_2$ and $\mathcal{C}_1$, respectively, correspond to:

$$P_2 = \sum_{i=t_2+1}^{n_2} \binom{n_2}{i} p_b^i (1 - p_b)^{n_2 - i} = 1 - \sum_{i=0}^{t_2} \binom{n_2}{i} p_b^i (1 - p_b)^{n_2 - i} \qquad (2)$$

Algorithm 2. Rep_c Algorithm for Concatenated Codes

```
Require: Helper data $(W_1, W_2)$, the decoding algorithms $\mathsf{Decode}_{\mathcal{C}_1}$ and $\mathsf{Decode}_{\mathcal{C}_2}$ corresponding to the $[n_1, k_1, d_1]$-code $\mathcal{C}_1$ and $[n_2, k_2, d_2]$-code $\mathcal{C}_2$, respectively, and a noisy PUF response $R'$ of size $s_R = l_2 \cdot n_2$ bits, where $l_2$ and $l_1$ are as determined in Algorithm 1.
Ensure: A key $K$ of size $s_K$
1: Set $\widetilde{w} \leftarrow W_1 \oplus R'$, where $\oplus$ is the bitwise logical XOR operation. This results in a bit string $\widetilde{w} = (\widetilde{w}_1, \widetilde{w}_2, \cdots, \widetilde{w}_{l_w})$, where $\widetilde{w}_i \in \{0,1\}$ and $l_w = l_2 \cdot n_2$.
2: Set $\widetilde{\mathbf{v}}_{\mathbf{2i}} = (\widetilde{w}_{(i-1)n_2+1}, \widetilde{w}_{(i-1)n_2+2}, \ldots, \widetilde{w}_{in_2})$ for $i = 1, 2, \ldots, l_2$.
3: Compute $\mathbf{v'_{2i}} \leftarrow \mathsf{Decode}_{\mathcal{C}_2}(\widetilde{\mathbf{v}}_{\mathbf{2i}})$ and recover the perturbed string $\widetilde{u}' = (\widetilde{U}'_1, \widetilde{U}'_2, \ldots, \widetilde{U}'_{l_2})$, where $\widetilde{U}'_i$ are words of size $k_2$ bits.
4: If zero bits were added during the $\mathsf{Gen}_c$ procedure, delete them and obtain a string $\widetilde{u} = (\widetilde{u}_1, \widetilde{u}_2, \cdots, \widetilde{u}_{l_u})$, where $\widetilde{u}_i \in \{0,1\}$ and $l_u = l_1 \cdot n_1$. Otherwise (if no zero bits were added) set $\widetilde{u} \leftarrow \widetilde{u}'$.
5: Set $\mathbf{v'_{1i}} = (\widetilde{u}_{(i-1)n_1+1}, \widetilde{u}_{(i-1)n_1+2}, \ldots, \widetilde{u}_{in_1})$ for $i = 1, 2, \ldots, l_1$.
6: Compute $\mathbf{v_{1i}} \leftarrow \mathsf{Decode}_{\mathcal{C}_1}(\mathbf{v'_{1i}})$ and recover the original string $u = (u_1, u_2, \cdots, u_{l_u})$, where $u_i \in \{0,1\}$ and $l_u = l_1 \cdot n_1$.
7: Perform Steps 4 through 12 of Algorithm 1 to recover $K$.
```

$$P_1 = \sum_{i=t_1+1}^{n_1} \binom{n_1}{i} P_2^i (1 - P_2)^{n_1 - i} = 1 - \sum_{i=0}^{t_1} \binom{n_1}{i} P_2^i (1 - P_2)^{n_1 - i}$$

where p_b is the bit error probability of the source, in our case the noise in the PUF response. Notice that P_i is the *word* error probability. In other words, the probability that a word will be in error after decoding. This is equal to the bit error probability in the case of the repetition code but not in the case of Golay, Reed-Muller or BCH codes. However, it is well known [30] that the resulting *bit* error probability is always less or equal to the word error probability as estimated in (2). Thus, we are being conservative in our estimates and the results in Table 2 correspond to worst case error probabilities. We report in Table 2 the best constructions that we found. Extensive tables for different PUF bit-error probabilities can be found in [28]. Our first observation about Table 2 is that we can achieve probabilities $\leq 10^{-6}$ not using BCH codes. This is a nice outcome of the construction, since this allows us to consider other codes which accept a more efficient implementation. Nevertheless, BCH codes error correction capabilities are indisputable. A second and perhaps more important observation is that when we combine a repetition code with a BCH code in a concatenated construction, the code size and thus, the number of SRAM source bits required decreases considerably (compare 4080 in Table 1 with the shortened $[226, 86, 43]$-BCH code at 2260 bits). Thus, it is clear that our construction offers considerable advantages.

Table 2. Output error probabilities for several concatenated codes with an input bit error probability of $p_b = 0.15$. Codes denoted with a star (*) have been shortened.

$\mathcal{C}_2$ $[n_2, k_2, d_2]$	$\mathcal{C}_1$ $[n_1, k_1, d_1]$	P_1	source bits for 171 bits
repetition $[3, 1, 3]$	$BCH[127, 29, 43]$	8.48 E-06	2286
	$RM[64, 7, 32]$	1.02 E-06	4800
	$BCH[63, 7, 31]$	8.13 E-07	4725
repetition $[5, 1, 5]$	$RM[32, 6, 16]$	1.49 E-06	4640
	$BCH[226, 86, 43]^*$	2.28 E-07	2260
repetition $[7, 1, 7]$	$G_{23}[23, 12, 7]$	1.58 E-04	2415
	$G_{23}[20, 9, 7]^*$	8.89 E-05	2660
	$BCH[255, 171, 23]$	8.00 E-05	1785
	$RM[16, 5, 8]$	3.47 E-05	3920
	$BCH[113, 57, 19]^*$	1.34 E-06	2373
repetition $[9, 1, 9]$	$BCH[121, 86, 11]$	6.84 E-05	2178
	$G_{23}[23, 12, 7]$	8.00 E-06	3105
	$RM[16, 5, 8]$	1.70 E-06	5040
repetition $[11, 1, 11]$	$G_{24}[24, 13, 7]$	5.41 E-07	3696
	$G_{23}[23, 12, 7]$	4.52 E-07	3795

4 HDA Architecture and Implementation Results

In addition to the error correcting properties of the codes that we previously considered, we also consider their performance from a hardware perspective. In particular, in this work we aim to make designs as small as possible in order to reserve space for the actual IP block to be implemented on the FPGA. We propose decoder architectures for first order Reed-Muller codes and for the binary Golay codes. For hash functions, we take the architecture proposed by Krawczyk [31] since this family accepts a more efficient implementation than all the other ones proposed or described in [32,31,33]. Due to space constraints, we refer to [28] for an exact analysis of their complexity.

Reed-Muller Codes. In this work we only consider first-order Reed-Muller codes because of their simple decoding algorithms. The procedure for decoding these codes is shown in Algorithm 3. To describe the process of generating the characteristic vectors let us denote the row of the generator matrix corresponding with the variable $\mathbf{v}_i$ by $\mathbf{v}_i^{(1)}$ and its logical negation by $\mathbf{v}_i^{(0)}$. Then the characteristic vectors of $\mathbf{v}_i$ are different vectors

$$\prod_{j=1,\cdots,m,j\neq i} \mathbf{v}_j^{(k_j)}, \tag{3}$$

where k_j is either 0 and 1. The values of k_j for each of the $m-1$ values of j assign to each characteristic vector for each variable a number between 0 and $2^{m-1}-1$. Thus an $m-1$ bit counter can be used to enumerate all of the characteristic

Algorithm 3. Decoding $\mathcal{R}(1, m)$ codes using Majority logic

```
Require: $x = (x_0, x_1, \ldots, x_{2^m-1})$ (the received vector) and $G \in \mathbb{F}_2^{(m+1)\times 2^m}$ the generator matrix of $\mathcal{R}(1, m)$
Ensure: $\hat{u} = (u_0, u_1, \cdots, u_m)$ the original message
for $i = 1$ to $m$ do do
    Find $2^{m-1}$ characteristic vectors for the row $i$ of $G$ (the index of the rows of $G$ begin with zero).
    Compute the dot product of each of these vectors with the received message.
    Compute the majority of the values of the dot products and assign it to $u_i$.
end for
Multiply $(u_1, \cdots u_m)$ by the sub-matrix consisting of the last $m$ rows of $G$ to get the vector $s$ with $2^m$ entries.
Assign the majority of the entries in $s + x$ to $u_0$.
```

vectors corresponding with a variable. We are aware of two different hardware structures proposed in the literature for hardware based decoding of $\mathcal{R}(1, m)$ codes which are also described in [21, Chapter 13]. However, since we target a low resource implementation, we propose an architecture that is a factor of m (asymptotically) smaller than those proposed in [21] at the cost of additional processing time. We defer a more detailed comparison of the architectures to the full version of the paper.

The newly proposed design is based on Algorithm 3 and is shown in Fig. 1. The most important parts of this circuit are the GM-generator and the CV-generator modules. The output of the GM-module periodically consists of each column of the generator matrix. Due to the structure of the $\mathcal{R}(1, m)$ generator matrix and considering the fact that only the last m rows of the $(m+1) \times 2^m$ generator matrix are required, it is easy to verify that GM-generator can be realized using an m-bit counter. The CV-generator module generates the bits of each of the characteristic vectors using the current column of the generator matrix, based on the index of the characteristic vector, the $m - 1$ values of k_j from (3), which is the output of CV Index and the variable being currently decoded which is the output of Variable counter. CV-generator consists of several multiplexers, which output an input value or its logical inverse, and the required circuitry to compute the product in (3). The Majority Decoder is a counter which can be either m bit or $m+1$ bit wide and its content must be compared with 2^{m-1} or 2^m, respectively. Hence its output is its $(m - 1)$th or mth bit, respectively. The output of Algorithm 3 is the message corresponding to the appropriate codeword whereas for PUF applications the codeword is needed. After the message is correctly decoded it is multiplied, again using the inner product module, by the generator matrix and the result is stored in the Received Word module. Our decoder requires $m2^{m-1}2^m = m2^{2m-1}$ iterations to process each row of the generator matrix with 2^m columns for each of the 2^{m-1} characteristic vectors and each of the m decoded messages. To this time the 2^m iterations for processing each of the 2^m columns of the generator matrix should be added. The Received Word module is used for both input and output of the values. The overall complexity of the decoder is shown in Table 3.

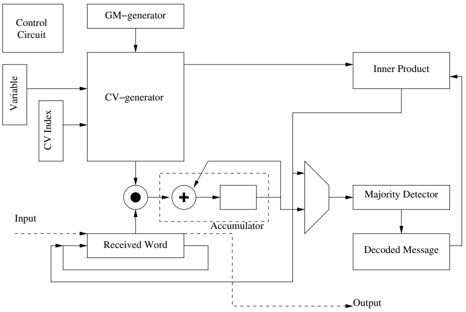

Fig. 1. Block diagram of our Reed-Muller decoder

Golay Code. An arithmetic decoder as seen in Algorithm 4 uses the weight structure of the syndrome to determine the error patterns [34]. P is the non-identity part of the generator matrix. The vector cp_i is the ith column of P and rp_i the ith row, respectively. The error vector is denoted as $e = (x, y)$ where x and y are vectors of length 12. A vector x_i is the zero vector with a 1 at the ith position. Our proposed circuit shown in Figure 2 can be derived from Algorithm 4. The main steps in the decoding process are the computation of the syndrome and helper syndrome, the determination of the Hamming weight, $GF(2)$ addition and the comparison of the Hamming weights with a constant.

We use a dot product block, which consists of 1 AND, 1 XOR and 1 FF for the serial computation of the syndrome and the helper syndrome. These syndromes are of length 12 and therefore the Hamming weight is at most 12 which can be represented with 4 bits. The result of the dot product or the $GF(2)$ addition respectively is loaded into a 12-bit shift register and a simple 4-bit counter counts the number of ones. The 4-bit counter requires 8 XOR, 8 AND and 4 OR gates. The result is then compared with a constant depending on the step of Algorithm 4. This step requires a 4-bit comparator and therefore 12 AND, 8 OR and 8 NOT gates. The constants are stored in memory and need 3 FFs.The $GF(2)$ addition is done with an XOR gate. Furthermore we need 24 FF for the error vector e, 24 FF to store the received vector and 288 FF for the generator matrix. The gate complexity without the control circuit is shown in Table 3. The above circuit can be optimized by removing the shift register and constructing the generator matrix on the fly, so we need to store only 24 bits for the first column and 23 bits for the second column. The other columns are determined by a simple shift operation. The complexity of the circuit without control can be seen in Table 3.

Universal Hashing. As previously mentioned, we use a construction due to Krawczyk [31]. This construction makes use of random binary matrices, where the hash value $h_A(x)$ is the Boolean multiplication of the matrix A by the message x. Krawczyk [31] shows how this can be implemented using a simple LFSR. We only need to store the first column of the matrix and the next columns are generated by the LFSR. The circuit is shown in Fig. 3. For a 128-bit key

Algorithm 4. Arithmetic Decoding of the Golay G_{24} code [34]

```
Require: $r$ (the received vector), $G = [I|P]$ (the generator matrix)
Ensure: $v$ (the encoded message)
Compute the syndrome $s = Gr$
if $wt(s) \leq 3$ then
    $e = (s^T, 0)$
else if $wt(s + cp_i) \leq 2$ for a column vector $cp_i$ then
    $e = ((s + cp_i)^T, y_i)$
else
    Compute the helper syndrome $z = P^T s$
    if $wt(z) \leq 3$ then
        $e = (0, (z)^T)$
    else if $wt(z + rp_i^T) \leq 2$ for a row vector $rp_i$ then
        $e = (x_i, (z)^T + rp_i)$
    else
        Too many errors
    end if
end if
$v = r + e$
```

Table 3. Area complexity of a serial implementation of arithmetic Golay decoding and low resources Reed-Muller decoder

Decoder Variant	FF	XOR	AND	OR	NOT
Golay store matrix	352	10	21	12	8
Golay generate matrix	99	10	21	12	8
Low resource $\mathcal{R}(1,m)$ decoder	$2^m + 6m - 1$	$\frac{m^2}{2} + \frac{13m}{2} - 2$	$9m$	$\frac{5m^2}{2} + \frac{m}{2} - 1$	m

the LFSR and the register for the accumulator need to be of size 128, thus requiring 256 FF. In addition the LFSR requires about $\frac{128}{2}$ XOR gates and the accumulator 128 XOR gates. The size of the shift register in Fig. 3 depends on the parameter n of the error correcting code used. Thus, for the Reed-Muller codes and Golay code it needs between 16 and 64 FFs depending on the code. Altogether without control the circuit requires 272-320 FFs and 192 XOR gates.

Implementation Results. We have implemented the repetition code, Golay and Reed-Muller code decoders ($\mathcal{R}(1,4), \mathcal{R}(1,5), \mathcal{R}(1,6)$) as well as the Toeplitz design for a universal hash function based on [31] in VHDL. We target the Spartan-3E 500 FPGA and use Xilinx ISE v9.2 for our tooling. We synthesized and mapped all designs. The Spartan-3E family of devices corresponds to low end FPGAs typically used in automotive and consumer electronic applications. The results are shown in Table 4. We observe that our designs our very space efficient. In particular, combining any repetition code decoder (a counter plus logic), a Reed-Muller or Golay code, any of the designs for the hash function our utilization is not greater than 10%. It seems clear that the Reed-Muller code is both superior in area but also in error correction performance (based on Table 2). Al-

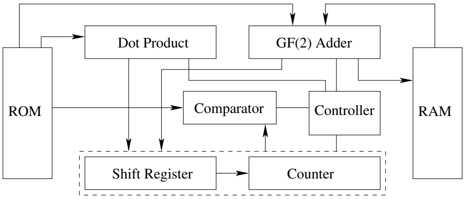

Fig. 2. Block diagram of our arithmetic Golay decoder

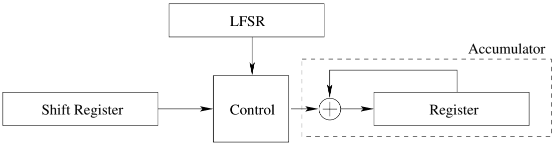

Fig. 3. LFSR-based Toeplitz hashing

Table 4. Implementation Results on Xilinx Spartan-3E-500 FPGA

Code / Hash	Output (bits)	Slices	Latency (cycles)	Critical Path (nsec)	Performance for 128-bit key@50 MHz (sec)
Repetition $[3, 1, 3]$	3	41 (1%)	6	5.3	$2.1\ 10^{-5}$
Repetition $[5, 1, 5]$	5	41 (1%)	10	5.3	$3.4\ 10^{-5}$
Repetition $[7, 1, 7]$	7	41 (1%)	14	5.3	$4.8\ 10^{-5}$
Repetition $[9, 1, 9]$	9	41 (1%)	18	5.3	$6.2\ 10^{-5}$
Repetition $[11, 1, 11]$	11	41 (1%)	22	5.3	$7.5\ 10^{-5}$
$\mathcal{R}(1, 4)$	16	69 (1%)	503	5.5	$3.5\ 10^{-4}$
$\mathcal{R}(1, 5)$	32	90 (1%)	1743	5.6	$1.0\ 10^{-3}$
$\mathcal{R}(1, 6)$	64	127 (1%)	6495	5.6	$3.2\ 10^{-3}$
Golay G_{24}	24	539 (5%)	1188	6.6	$3.6\ 10^{-4}$
Toeplitz Hash 16 [31]	128	319 (3%)	64	5.7	$1.0\ 10^{-5}$
Toeplitz Hash 24 [31]	128	327 (3%)	96	5.7	$1.2\ 10^{-5}$
Toeplitz Hash 32 [31]	128	335 (3%)	128	5.7	$1.0\ 10^{-5}$
Toeplitz Hash 64 [31]	128	367 (3%)	256	5.7	$1.0\ 10^{-5}$

though, the Golay code decoder can be optimized by generating the parity check matrix on the fly, in terms of area it is still outperformed by the Reed-Muller codes. We can estimate the complexity of the overall helper data algorithm by assuming a concatenated construction with a Repetition code, a Reed-Muller code, a Toeplitz-based hash function and 100% overhead for control. Even then, the overall fuzzy extractor requires less than 10% of the FPGA resources. Unfortunately, we have not found any BCH code implementations on FPGAs to which we can compare. However, the BCH decoding algorithms themselves are much more complex, thus, it is expected that their hardware complexity will be similarly higher. In the full version of the paper, we expect to have a full BCH decoder and thus, be able to fully compare all our constructions.

5 Conclusion

We present the first efficient implementations of helper data algorithms on FPGAs. Helper data algorithms are used to extract cryptographic keys from the noisy response of, e.g., a Physical Unclonable Function (PUF). PUFs have become an attractive subject of research due to their nice properties for unforgeable authentication and secure key storage purposes. In particular, one can deploy them to securely bind applications to the underlying hardware, a mechanism that has various applications and most prominently IP protection. Our solution offers the last missing building block toward real world IP protection on FPGAs. Our helper data algorithms are efficient with regard to the required hardware resources, which is important for hardware design. In the design of our helper data algorithms, we make use of various linear codes constructions, each with own advantages and shortcomings. These constructions are then compared in terms of error correction capabilities and hardware resource usage, giving the designer the necessary tools to make an informed decision when implementing a helper data algorithm.

References

1. Pappu, R.S., Recht, B., Taylor, J., Gershenfeld, N.: Physical one-way functions. Science 297(6), 2026–2030 (2002)
2. Tuyls, P., Schrijen, G.-J., Škorić, B., van Geloven, J., Verhaegh, N., Wolters, R.:: Read-Proof Hardware from Protective Coatings. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 369–383. Springer, Heidelberg (2006)
3. Trusted Computing Group: TPM main specification. Technical Report Version 1.2 Revision 94 (March 2006)
4. Gassend, B., Clarke, D.E., van Dijk, M., Devadas, S.: Silicon physical unknown functions. In: Atluri, V. (ed.) ACM Conference on Computer and Communications Security — CCS 2002, pp. 148–160. ACM, New York (2002)
5. 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.) CHES 2007. LNCS, vol. 4727, pp. 63–80. Springer, Heidelberg (2007)
6. Škorić, B., Bel, T., Blom, A., de Jong, B., Kretschman, H., Nellissen, A.: Randomized resonators as uniquely identifiable anti-counterfeiting tags. Technical report, Philips Research Laboratories (January 28, 2008)
7. Kean, T.: Cryptographic rights management of FPGA intellectual property cores. In: ACM/SIGDA International Symposium on Field-Programmable Gate Arrays — FPGA 2002, pp. 113–118 (2002)
8. Simpson, E., Schaumont, P.: Offline Hardware/Software Authentication for Reconfigurable Platforms. In: Goubin, L., Matsui, M. (eds.) CHES 2006. LNCS, vol. 4249, pp. 311–323. Springer, Heidelberg (2006)
9. Guajardo, J., Kumar, S.S., Schrijen, G.J., Tuyls, P.: Physical Unclonable Functions and Public Key Crypto for FPGA IP Protection. In: International Conference on Field Programmable Logic and Applications — FPL 2007, August 27-30, 2007, pp. 189–195. IEEE, Los Alamitos (2007)
10. Linnartz, J.P.M.G., Tuyls, P.: New Shielding Functions to Enhance Privacy and Prevent Misuse of Biometric Templates. In: Kittler, J., Nixon, M.S. (eds.) AVBPA 2003. LNCS, vol. 2688, pp. 393–402. Springer, Heidelberg (2003)

11. Dodis, Y., Reyzin, M., Smith, A.: Fuzzy extractors: How to generate strong keys from biometrics and other noisy data. In: Cachin, C., Camenisch, J.L. (eds.) EUROCRYPT 2004. LNCS, vol. 3027, pp. 523–540. Springer, Heidelberg (2004)
12. Suh, G.E., O'Donnell, C.W., Devadas, S.: AEGIS: A Single-Chip Secure Processor. IEEE Design & Test of Computers 24(6), 570–580 (2007)
13. Dijk, M.v., Lim, D., Devadas, S.: Reliable Secret Sharing With Physical Random Functions. Computation Structures Group Memo 475, CSAIL — Massachusetts Institute of Technology (2004)
14. Gassend, B.: Physical Random Functions. Master's thesis, Computer Science and Artificial Intelligence Laboratory, MIT Computation Structures Group Memo 458 (February 2003)
15. Juels, A., Wattenberg, M.: A Fuzzy Commitment Scheme. In: Motiwalla, J., Tsudik, G. (eds.) ACM Conference on Computer and Communications Security — ACM CCS 1999, November 1-4, 1999, pp. 28–36. ACM, New York (1999)
16. Hao, F., Anderson, R., Daugman, J.: Combining Crypto with Biometrics Effectively. IEEE Transactions on Computers 55(9), 1081–1088 (2006)
17. Hochquenghem, A.: Codes Correcteurs D'erreurs. Chiffres 2, 147–156 (1959)
18. Bose, R.C., Ray-Chaudhuri, D.K.: On a Class of Error-Correcting Binary Group Codes. Information and Control 3, 68–79 (1960)
19. Forney Jr., G.D.: Concatenated Codes. Research Monograph No. 37. MIT Press, Cambridge (1966)
20. Blahut, R.E.: Theory and Practice of Error Control Codes, 1st edn. Addison-Wesley Publishing Company, Reading (1985)
21. MacWilliams, F.J., Sloane, N.J.A.: The Theory of Error-Correcting Codes. North-Holland Mathematical Library, vol. 16. North-Holland/Elsevier, Amsterdam (1977)
22. Carter, L., Wegman, M.N.: Universal Classes of Hash Functions. J. Comput. Syst. Sci. 18(2), 143–154 (1979)
23. Xilinx: Device Reliability Report — Fourth Quarter 2007. Technical Report UG116 (v4.3) (February 6, 2008), http://www.xilinx.com/support/documentation/
24. Altera: Reliability Report 45 — Q2 2007. Technical report (2007), http://www.altera.com/literature/lit-index.html.
25. MacKay, D.J.C.: Good Error-Correcting Codes Based on Very Sparse Matrices. IEEE Transactions on Information Theory 45(2), 399–431 (1999)
26. Levine, B.A., Reed Taylor, R., Schmit, H.: Implementation of Near Shannon Limit Error-Correcting Codes Using Reconfigurable Hardware. In: IEEE Symposium on Field-Programmable Custom Computing Machines — FCCM 2000, April 17-19, 2000, pp. 217–226. IEEE Computer Society, Los Alamitos (2000)
27. Brack, T., Kienle, F., Wehn, N.: Disclosing the LDPC code decoder design space. In: Gielen, G.G.E. (ed.) Conference on Design, Automation and Test in Europe — DATE 2006, European Design and Automation Association, Leuven, Belgium, March 6-10, 2006, pp. 200–205 (2006)
28. Bösch, C.: Efficient fuzzy extractors for reconfigurable hardware. Master's thesis, Chair for System Security, Department of Electrical Engineering and Information Science, Ruhr-Universität Bochum (March 2008)
29. Desset, C., Macq, B., Vandendorpe, L.: Block error-correcting codes for systems with a very high BER: Theoretical analysis and application to the protection of watermarks. Signal Processing: Image Communication 17(5), 409–421 (2002)
30. Desset, C., Macq, B.M., Vandendorpe, L.: Computing the word-, symbol-, and bit-error rates for block error-correcting codes. IEEE Transactions on Communications 52(6), 910–921 (2004)

31. Krawczyk, H.: LFSR-based Hashing and Authentication. In: Desmedt, Y.G. (ed.) CRYPTO 1994. LNCS, vol. 839, pp. 129–139. Springer, Heidelberg (1994)
32. Nevelsteen, W., Preneel, B.: Software Performance of Universal Hash Functions. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 24–41. Springer, Heidelberg (1999)
33. Kaps, J.P., Yüksel, K., Sunar, B.: Energy Scalable Universal Hashing.. IEEE Trans. Computers 54(12), 1484–1495 (2005)
34. Vanstone, S.A., van Oorschot, P.C.: An Introduction to Error Correcting Codes with Applications. Kluwer Academic Publishers, Dordrecht (1989)