PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator

Roel Maes, Anthony Van Herrewege, and Ingrid Verbauwhede

KU Leuven Dept. Electrical Engineering-ESAT/SCD-COSIC and IBBT
Kasteelpark Arenberg 10, B-3001 Leuven-Heverlee, Belgium
{roel.maes,anthony.vanherrewege,ingrid.verbauwhede}@esat.kuleuven.be

Abstract. We present PUFKY: a practical and modular design for a cryptographic key generator based on a Physically Unclonable Function (PUF). A fully functional reference implementation is developed and successfully evaluated on a substantial set of FPGA devices. It uses a highly optimized ring oscillator PUF (ROPUF) design, producing responses with up to 99% entropy. A very high key reliability is guaranteed by a syndrome construction secure sketch using an efficient and extremely low-overhead BCH decoder. This first complete implementation of a PUF-based key generator, including a PUF, a BCH decoder and a cryptographic entropy accumulator, utilizes merely 17% (1162 slices) of the available resources on a low-end FPGA, of which 82% are occupied by the ROPUF and only 18% by the key generation logic. PUFKY is able to produce a cryptographically secure 128-bit key with a failure rate $< 10^{-9}$ in 5.62 ms. The design's modularity allows for rapid and scalable adaptations for other PUF implementations or for alternative key requirements. The presented PUFKY core is immediately deployable in an embedded system, e.g. by connecting it to an embedded microcontroller through a convenient bus interface.

Keywords: Physically Unclonable Functions (PUFs), Cryptographic Key Generation, Fuzzy Extractors.

1 Introduction

An indispensable premise for the majority of cryptographic implementations is the ability to securely generate, store and retrieve keys. The required effort to meet these conditions is often underestimated in the algorithmic description of cryptographic primitives. The minimal common requirements for a secure key generation and storage are *i)* a source of true randomness that ensures unpredictable and unique fresh keys, and *ii)* a protected memory which reliably stores the key's information while shielding it completely from unauthorized parties. From an implementation perspective, both requisites are non-trivial to achieve. The need for unpredictable randomness is typically filled by applying a seeded pseudo-random bit generator (PRNG). However, the fact that such generators are difficult to implement properly was just recently made clear again by the observation [13] that a large collection of "random" public RSA keys contains many

E. Prouff and P. Schaumont (Eds.): CHES 2012, LNCS 7428, pp. 302–319, 2012.

pairs which share a prime factor, which is immediately exploitable. Implementing a protected memory is also a considerable design challenge, often leading to increased implementation overhead and restricted application possibilities, to enforce the physical security of the stored key. Countless examples can be provided of broken cryptosystems due to poorly designed or implemented key storages, or bad handling of keys. Moreover, even high-level physical protection mechanisms are often not sufficient to prevent well-equipped and motivated adversaries from discovering stored secrets [24, 25].

PUF-based key generators try to tackle both requirements at once by harvesting static, device-unique randomness and processing it into a cryptographic key. This avoids the need for both a PRNG, since the randomness is already intrinsically present in the device, and the need for a protected non-volatile memory, since the used randomness is static over the lifetime of the device and can be measured again and again to regenerate the same key from otherwise illegible random features. Since PUF responses are generally noisy and of low-entropy, a PUF-based key generator faces two main challenges: increasing the reliability to a practically acceptable level and compressing sufficient entropy in a fixed length key. Fuzzy extractors [7] perform exactly these two functions and can be immediately applied for this purpose, as suggested in a number of earlier PUF key generator proposals. In [10], Guajardo et al. propose to use an SRAM PUF for generating keys, using a fuzzy extractor configuration based on linear block codes. This idea was extended and optimized by Bösch et al. [4] who propose a concatenated block code configuration, and Maes et al. [14] who propose to use a soft-decision decoder. Yu et al. [28] propose a configuration based on ring oscillator PUFs and apply an alternative error-correction method.

Contribution. Our main contribution is a highly practical PUF-based cryptographic key generator design (PUFKY), and an efficient yet fully functional FPGA reference implementation thereof. The proposed design comprises a number of major contributions based on new insights: *i)* we propose a novel variant of a ring oscillator PUF based on very efficient Lehmer-Gray order encoding; *ii)* we abandon the requirement of information-theoretical security in favor of a much more practical yet still cryptographically strong key generation; *iii)* we counter the widespread belief that code-based error-correction, BCH decoding in particular, is too complex for efficient PUF-based key generation, by designing a highly resource-optimized BCH decoder; and *iv)* we present a global optimization strategy for PUF-based key generators based on well-defined design constraints.

Structure. In Section 2 we provide necessary background information on the individual elements of the proposed key generator. Section 3 describes the design stage, putting all these elements together in the PUFKY architecture and Section 4 provides concrete results on an optimized reference implementation of the proposed PUF and the full PUFKY design. In Section 5, we discuss some interesting details of our design and hint at possible future improvements and applications. Finally, we conclude in Section 6.

2 Background

2.1 Notation

We briefly introduce the notational conventions used throughout this work. A random variable is denoted by a capital letter X and a particular outcome thereof by a lower case letter x. A vector of length n is written as $X^n = (X_1, \ldots, X_n)$ and $\mathrm{HW}(X^n)$ is the Hamming weight of X^n. A matrix is represented by a bold faced symbol $\mathbf{A}$. $H(X)$ is the Shannon entropy of the random variable X and $H_\infty(X)$ is its min-entropy. For a random binary vector $X^n \in \{0,1\}^n$, we respectively define $R(X^n) \equiv \frac{H(X^n)}{n}$ and $R_\infty(X^n) \equiv \frac{H_\infty(X^n)}{n}$. By $B_{n,p}(t)$ we denote the binomial cumulative distribution function with parameters n and p evaluated in t, and $B^{-1}_{n,p}(q)$ is its inverse. By $\mathcal{C}(n,k,t)$ we denote a binary block code of length n, dimension k and minimal distance $2t+1$ which is hence able to correct up to t bit errors. When $\mathcal{C}(n,k,t)$ is linear it is defined by a generator and a parity-check matrix, respectively denoted by $\mathbf{G}^{k \times n}$ and $\mathbf{H}^{n-k \times n}$, satisfying the property $\mathbf{G}\mathbf{H}^T = \mathbf{0}$.

2.2 Physically Unclonable Functions (PUFs)

PUFs are hardware primitives which produce unpredictable and instantiation-dependent outcomes. A silicon PUF is implemented on a silicon chip and uses the intrinsic device randomness caused by chip manufacturing process variations to generate a device-unique response. Due to their physical nature, PUF responses are generally not perfectly reproducible (noisy) and not perfectly random. If we consider the response of a particular PUF instance as a binary vector X^n, the unreliability is expressed by the expected bit error rate between two evaluations x^n and x'^n of the same response: $\Pr(x_i \neq x'_i)$. The entropy density $R(X^n)$ of a response expresses its relative amount of randomness. We will refer to a PUF with a maximal bit error rate p_e and an entropy density of at least ρ as a (p_e, ρ)-PUF.

A Ring Oscillator PUF (ROPUF) is a silicon PUF which generates a response based on the frequencies of on-chip digital ring oscillators. Since the exact frequency of a such oscillators is noticeably affected by process variations, an accurate measurement thereof will contain unpredictable and device-unique information. The first concept of a ROPUF was proposed by Gassend et al. [9], based on a single configurable oscillator. Concerns about predictability and robustness led to the proposal of an improved ROPUF structure by Suh and Devadas [23], which uses a number of fixed oscillators and considers the relative frequencies of oscillator pairs instead of their absolute values. Yin and Qu [27] further explored this technique by considering the frequency ordering of larger groups of oscillators which is able to produce longer bit responses. Maiti et al. [15] performed an extensive characterization of ROPUFs on a large FPGA population, justifying their qualities as silicon PUFs.

2.3 Secure Sketching

The notion of a secure sketch was proposed by Dodis et al. [7] and provides a method to reliably reconstruct the outcome of a noisy variable in such a way that the entropy of the outcome remains high. A number of possible constructions based on error-correcting codes was also proposed in [7]. In this work, we will focus on the *syndrome construction* for binary vectors.

We describe the operation of a syndrome construction secure sketch which uses a binary linear block code $\mathcal{C}(n, k, t)$ with parity-check matrix $\mathbf{H}$. The *sketch procedure* takes as input an outcome of $X^n \to x^n$ and produces a *sketch* $h^{n-k} = x^n\mathbf{H}^T$. The *recovery procedure* takes as input a different (possibly noisy) outcome of $X^n \to x'^n (= x^n \oplus e^n$ with e^n a bit error vector) and the previously generated sketch h^{n-k}, and calculates the *syndrome* $s^{n-k} = x'^n\mathbf{H}^T \oplus h^{n-k}$. Because of the linearity of the code, it is easy to show that $s^{n-k} \equiv e^n\mathbf{H}^T$. If $\mathrm{HW}(e^n) \leq t$ then e^n can be decoded from s^{n-k}, which is equivalent to a decoding operation for $\mathcal{C}(n, k, t)$, and x^n can be recovered as $x^n = x'^n \oplus e^n$.

The sketch h^{n-k} needs to be stored in between sketching and recovering. The key point is that knowledge of h^{n-k} does not fully disclose the entropy of X^n, but at most $n-k$ bits thereof. This means that h^{n-k} can be stored and communicated publicly and there will still be at least $H(X^n) - (n-k)$ bits of entropy left in X^n. In the setting of cryptographic key generation, the term *helper data* is used to refer to such public information which is produced by the initial key extraction and used by subsequent key regenerations.

The design parameters of the syndrome construction are mainly determined by the selection of an appropriate linear block code $\mathcal{C}(n, k, t)$. In order to yield a meaningful secure sketch, $\mathcal{C}(n, k, t)$ needs to meet some constraints determined by the available (p_e, ρ)-PUF and by the required remaining entropy m and reliability $1 - p_{\mathit{fail}}$ of the output of the secure sketch. These constraints are listed in the first column of Table 1. The *practicality* constraint restricts the possible codes to ones for which a practical decoding algorithm exists. The *rate* and *correction* constraints further bound the possible code parameters as a function of the available input (p_e, ρ) and the required output (m, p_{fail}). They respectively express the requirement of not disclosing the full entropy of the PUF through the helper data, and the minimally needed bit error correction capacity in order to meet the required reliability. Bösch et al. [4] demonstrated that code concatenation offers considerable advantages when used in secure sketch constructions. Notably the use of a simple repetition code as an inner code significantly relaxes the design constraints. The parameter constraints for a syndrome construction based on the concatenation of a repetition code $\mathcal{C}_1(n_1, 1, t_1 = \frac{n_1-1}{2})$ as an inner code and a second linear block code $\mathcal{C}_2(n_2, k_2, t_2)$ as an outer code, are given in the second column of Table 1.

2.4 BCH Decoding

BCH codes are particularly performant cyclical linear block codes for which efficient error-decoding algorithms exist. A binary BCH code $\mathcal{C}_{\mathit{BCH}}(n_{\mathit{BCH}}, k_{\mathit{BCH}}, t_{\mathit{BCH}})$

Table 1. Parameter constraints for the syndrome construction of secure sketches, depending on the type of code construction used

	$\mathcal{C}(n,k,t)$	$\mathcal{C}_2(n_2,k_2,t_2) \circ \mathcal{C}_1(n_1,1,t_1 = \frac{n_1-1}{2})$
Practicality	$\mathcal{C}(n,k,t)$ is efficiently decodable	$\mathcal{C}_2(n_2,k_2,t_2)$ is efficiently decodable
Rate	$\frac{k}{n} > 1-\rho$	$\frac{k_2}{n_1 n_2} > 1-\rho$
Correction	$t \geq B^{-1}_{n,p_e}\left((1-p_{fail})^{\frac{1}{r}}\right)$, with $r = \lceil \frac{m}{k-n(1-\rho)} \rceil$	$t_2 \geq B^{-1}_{n_2,1-p'_e}\left((1-p_{fail})^{\frac{1}{r}}\right)$, with $p'_e = 1 - B_{n_1,p_e}(t_1)$ $r = \lceil \frac{m}{k_2-n_1 n_2(1-\rho)} \rceil$

is defined for $n_{BCH} = 2^u - 1$, but BCH codes of any code length can be constructed by also considering shortened versions: $\mathcal{C}_{BCH}(n_{BCH} - v, k_{BCH} - v, t_{BCH})$.

Decoding a BCH syndrome into the most-likely bit error vector is typically performed in three steps. First, so called *syndrome evaluations* z_i are calculated by evaluating the *syndrome* s^{n-k} as a polynomial for $\alpha, \ldots, \alpha^{2t_{BCH}}$, with α a generator for $\mathbb{F}_{2^u}$. The next step is using these z_i to generate an error location polynomial Λ. This is generally accomplished with the Berlekamp-Massey (BM) algorithm. First published by Berlekamp [2] and later optimized by Massey [16], this algorithm requires the inversion of an element in $\mathbb{F}_{2^u}$ in each of its $2t_{BCH}$ iterations. In order not to have to do this costly calculation, many authors have come up with modified versions of the algorithm, e.g. [20–22]. However, these are all time-memory tradeoffs of the original inversionless BM algorithm by Burton [5], which we prefer due to its lower storage requirements. Finally, by calculating the roots of Λ, one can find the error vector e^n. This is done with the Chien search algorithm [6] by evaluating Λ for $\alpha, \ldots, \alpha^{t_{BCH}}$. If Λ evaluates to zero for α^i then the corresponding error bit $e_{n_{BCH}-i} = 1$.

2.5 Cryptographic Key Generation

To ensure their unpredictability, cryptographic keys should be generated from a random source. Recommendations for appropriate sources and best practice extraction methods can be found, e.g. in [1, 8, 12], and are used heavily in practical implementations. In addition to these best practice methods, *strong extractors* [18] have been proposed as unconditionally secure extractors of uniform randomness. However they generally induce a large entropy loss, i.e. the output length is much smaller than the entropy of the input, which is undesirable since high-entropy randomness is scarce in most implementations. To generate reliable keys from noisy non-uniform sources like PUFs, Dodis et al. [7] introduced the concept of a *fuzzy extractor*. This is basically a concatenation of a secure sketch, as described in Sect. 2.3, with a strong extractor and is able to generate information-theoretically secure keys. To obtain this very high security level, one still has to make a strong assumption about the min-entropy of the randomness source, which is often impossible. Moreover, due to the use of a strong extractor, large entropy losses need to be taken into account here, which often makes the overall key generation very impractical[1].

Another approach is considered in key generation based on PRNGs seeded from an entropic source, as described in [1, 8, 12]. Such generators obtain their initial internal state by accumulating entropy from a, usually low-quality, entropic source using an entropy accumulation function. In [1, Sect. 10.4], constructions for entropy accumulators based on a generic cryptographic hash function or a block cipher are provided. Kelsey et al. [12] also strongly recommend a cryptographic hash function for this purpose. Following this motivation, we opt for a hash function to accumulate entropy in our design. The amount of data to be accumulated to reach a sufficient entropy level, depends on the (estimated) entropy rate of the considered source. For PRNGs which produce large quantities of output data, the source entropy estimates are usually very conservative. For PUFs, entropy comes at a high implementation cost and being too conservative leads to an excessively large overhead. For this reason we are forced to consider relatively tight estimates on the remaining entropy in a PUF response after secure sketching. On the other hand, the output length of a PUF-based key generator is very limited (a single key) compared to PRNGs. In any case, the total amount of entropy which needs to be accumulated should at least match the length of the generated key.

3 Design

3.1 PUFKY Architecture

The top-level architecture of our PUFKY PUF-based key generator is shown in Fig. 1. As a PUF, we use an ROPUF which produces high-entropy outputs based on the frequency ordering of a selection of ring oscillators, as described in Section 3.2. To account for the bit errors present in the PUF response, we use a secure sketch construction based on the concatenation of two linear block codes, a repetition code $\mathcal{C}_{REP}(n_{REP}, 1, \frac{n_{REP}-1}{2})$ with n_{REP} odd and a BCH code $\mathcal{C}_{BCH}(n_{BCH}, k_{BCH}, t_{BCH})$. The design of the syndrome generation and error decoder blocks used in the secure sketching is described in Section 3.3. To accumulate the remaining entropy after secure sketching, we apply the recently proposed light-weight cryptographic hash function SPONGENT [3].

3.2 ROPUF Design

Our ROPUF design is inspired by the design from Yin and Qu [27] which generates a response based on the frequency ordering of a set of oscillators. A measure of the frequency of an oscillator is obtained by counting the number of oscillations in a

[1] In earlier work on PUF-based key generation with fuzzy extractors, e.g. [4, 10, 14], the additional entropy loss by the strong extractor is ignored and the resulting keys can not be considered information-theoretically secure.

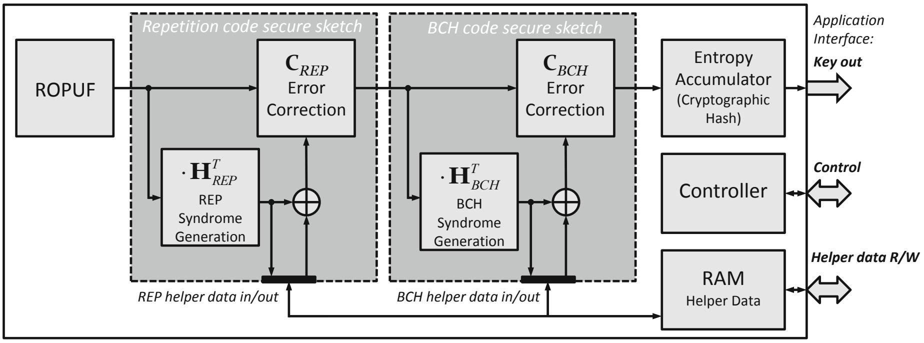

Repetition code secure sketch
BCH code secure sketch
ROPUF
$\mathbf{C}_{REP}$ Error Correction
$\cdot\mathbf{H}^T_{REP}$ REP Syndrome Generation
$\mathbf{C}_{BCH}$ Error Correction
$\cdot\mathbf{H}^T_{BCH}$ BCH Syndrome Generation
Entropy Accumulator (Cryptographic Hash)
Controller
RAM Helper Data
REP helper data in/out
BCH helper data in/out
Application Interface:
Key out
Control
Helper data R/W

Fig. 1. PUFKY: PUF-based cryptographic key generator architecture

fixed time interval. To amortize the overhead of the frequency counters, oscillators are ordered in b batches of a oscillators sharing a counter. In total, our ROPUF design contains $b \times a$ oscillators of which sets of b can be measured in parallel. The measurement time is determined as a fixed number of cycles of an independent on-chip ring oscillator and is fixed at 87 µs. After some post-processing, an ℓ-bit response is generated based on the relative ordering of b simultaneously measured frequencies. A total of $a \times \ell$-bit responses can be produced by the ROPUF in this manner. Note that, to ensure the independence of different responses, each oscillator is only used for a single response generation. The architecture of our ROPUF design is shown in Fig. 2.

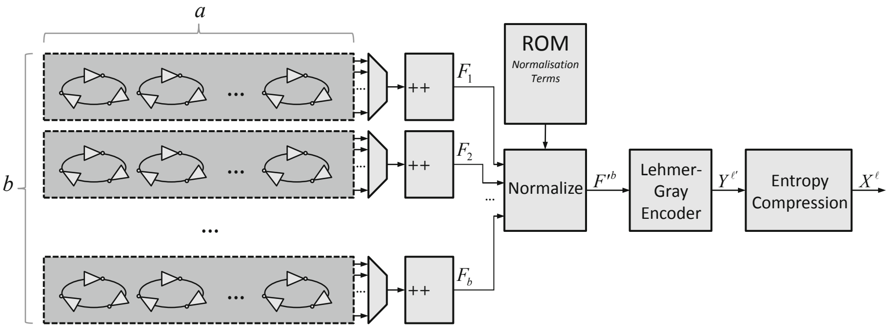


Fig. 2. ROPUF architecture

Encoding the ordering of b frequency measurements $F^b = (F_1, \ldots, F_b)$ in an ℓ-bit response $X^\ell = (X_1, \ldots, X_\ell)$, turns out to be the main design challenge for this type of ROPUF. As discussed in Section 2.3, the *quality* of the PUF responses, expressed by (p_e, ρ), will be decisive for the design constraints of the secure sketch, and by consequence for the key generator as a whole. The details of the post-processing will largely determine the final values for (p_e, ρ). We propose a three-step encoding for $F^b \to X^\ell$:

1. *Frequency Normalization:* remove structural bias from the measurements.
2. *Order Encoding:* encode the normalized frequency ordering to a stable bit vector in such a way that all ordering entropy is preserved.
3. *Entropy Compression:* compress the order encoding to maximize the entropy density without significantly increasing the bit error probability.

Frequency Normalization. Only a portion of a measured frequency F_i will be random, and only a portion of that randomness will be caused by the effects of process variations on the considered oscillator. The analysis from [15] demonstrates that F_i is subject to both device-dependent and oscillator-dependent *structural* bias. Device-dependent bias does not affect the ordering of oscillators on a single device, so we will not consider it further. Oscillator-dependent structural bias on the other hand is of concern to us since it has a potentially severe impact on the randomness of the frequency ordering. From a probabilistic viewpoint, it is reasonable to assume the frequencies F_i to be independent, but due to the oscillator-dependent structural bias we can not consider them to be *identically* distributed since each F_i has a different expected value μ_{F_i}. The ordering of F_i will be largely determined by the deterministic ordering of μ_{F_i} and not by the effect of random process variations on F_i. Fortunately, we are able to obtain an accurate estimate $\tilde{\mu}_{F_i}$ of μ_{F_i} by averaging F_i over many measurements on many devices. Subtracting this estimate from the measured frequency gives us a *normalized frequency* $F'_i = F_i - \tilde{\mu}_{F_i}$. Assuming $\tilde{\mu}_{F_i} \approx \mu_{F_i}$, the resulting normalized frequencies F'_i will be independent *and* identically distributed (i.i.d.). Calculating $\tilde{\mu}_{F_i}$ needs to be performed only once for a single design after the oscillator implementations are fixed, preferably over an initial test batch of ROPUF instances. When these normalization terms are known with high accuracy, they are included in the design, e.g. using a ROM.

Order Encoding. Sorting a vector F'^b of normalized frequencies, e.g. in ascending order, amounts to rearranging its elements in one of $b!$ possible ways. The goal of the order encoding step is to produce an ℓ'-bit vector $Y^{\ell'}$ which uniquely encodes the ascending order of F'^b. Since the elements of F'^b are i.i.d., each of the $b!$ possible orderings is equally likely to occur [26], leading to $H(Y^{\ell'}) = \log_2 b! = \sum_{i=2}^{b} \log_2 i$. An optimal order encoding has a high entropy density but a minimal sensitivity to noise on the F'_i values. We propose a Lehmer encoding of the frequency ordering, followed by a Gray encoding of the Lehmer coefficients. A Lehmer code is a unique numerical representation of an ordering which is moreover efficient to obtain since it does not require explicit value sorting. It represents the sorted ordering of F'^b as a coefficient vector $L^{b-1} = (L_1, \ldots, L_{b-1})$ with $L_i \in \{0, 1, \ldots, i\}$. It is clear that L^{b-1} can take $2 \times 3 \times \ldots \times b = b!$ possible values which is exactly the number of possible orderings. The Lehmer coefficients are calculated from F'^b as $L_j = \sum_{i=1}^{j} gt(F'_{j+1}, F'_i)$, with $gt(x, y) = 1$ if $x > y$ and

0 otherwise. The Lehmer encoding has the nice property that a minimal change in the sorted ordering caused by two neighboring values swapping places only changes a single Lehmer coefficient by ±1. Using a binary Gray encoding for the Lehmer coefficients, this translates to only a single bit difference as preferred. The length of the binary representation becomes $\ell' = \sum_{i=2}^{b} \lceil \log_2 i \rceil$ yielding $R\left(Y^{\ell'}\right) = \frac{\sum_{i=2}^{b} \log_2 i}{\sum_{i=2}^{b} \lceil \log_2 i \rceil}$ which is close to optimal.

Entropy Compression. $R\left(Y^{\ell'}\right)$ is already quite high, but can be increased further by compressing it to X^{ℓ} with $\ell \leq \ell'$. Note that $Y^{\ell'}$ is not quite uniform over $\{0,1\}^{\ell'}$ since some bits of $Y^{\ell'}$ are biased and/or dependent. This results from the fact that most of the Lehmer coefficients, although uniform by themselves, can take a range of values which is not an integer power of two, leading to a suboptimal binary encoding. We propose a simple compression by selectively XOR-ing bits from $Y^{\ell'}$ which suffer the most from bias and/or dependencies, leading to an overall increase of the entropy density. Note that XOR-compression potentially also increases the bit error probability, but at most by a factor $\frac{\ell'}{\ell}$.

3.3 Syndrome Generation and Error Decoding for $\mathcal{C}_{\text{REP}}$ and $\mathcal{C}_{\text{BCH}}$

Repetition Code $\mathcal{C}_{\text{REP}}$. The syndrome generation of $x^{n_{\text{REP}}}$ consists of pairwise XOR-ing x_1 with each remaining bit of $x^{n_{\text{REP}}}$, or $h_i = x_1 \oplus x_{i+1}$. Error decoding is based on a Hamming weight check of the syndrome $s^{n_{\text{REP}}-1}$, which immediately yields the value for the first error bit e_1. The remaining error bits are again obtained by a pairwise XOR of e_1 with each of the syndrome bits, but this step is discarded in the syndrome construction. In our design, both syndrome generation and error decoding of a repetition code are fully combinatorial.

BCH Code $\mathcal{C}_{\text{BCH}}$. Since BCH codes are cyclical codes, their syndrome generation is a finite field division by the code's generator polynomial. This is efficiently implemented in hardware as an LFSR evaluation of length $(n_{\text{BCH}} - k_{\text{BCH}})$.

The error decoding step of a BCH code is more complex and requires the largest design effort of all elements in our secure sketch. Most BCH decoders are designed with a focus on throughput and use systolic array designs, e.g. [19, 20, 22]. Aiming for a size-optimized implementation, we propose a serialized, minimalistic coprocessor design with a 10-bit application-specific instruction set and limited conditional execution support. Although highly optimized towards BCH decoding, the architecture is generic in the sense that it can decode any BCH code, including shortened versions, requiring only a slight change of firmware and memory size. The datapath consists of two blocks: an address and a data block. To optimize array indexing, all addressing is done indirectly using a five element address RAM, which is efficiently updated by a dedicated address ALU. The output of the address RAM is directly connected to the data RAM.

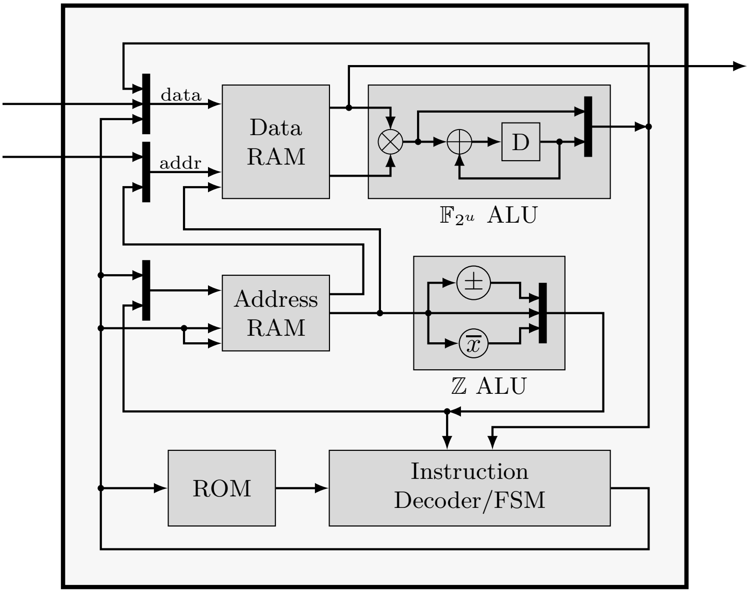


Fig. 3. BCH decoder architecture

The data block consists of data RAM and an ALU which is used mainly for multiply-accumulate operations over $\mathbb{F}_{2^u}$. To minimize the size, this ALU contains only a single register. All other necessary operands come directly from the data RAM. A high-level overview of the coprocessor architecture is shown in Fig. 3.

BCH error decoding is done in the three steps elaborated in Section 2.4. A listing of each used algorithm and their approximate runtimes can be found in Appendix A. The performance of the algorithm execution is heavily optimized using branch removal and loop unrolling. The coprocessor's instruction set can be found in Appendix B.

4 Implementation

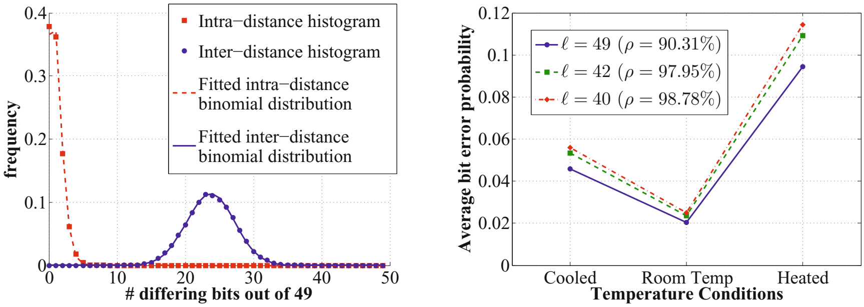
We now present the implementation results of our PUFKY design as described in Section 3. The implementation was synthesized, configured and tested on a Xilinx® Spartan®-6 FPGA (XC6SLX45) which is a low-end FPGA in 45 nm technology, specifically targeted for embedded system solutions.

4.1 PUF Implementation and Characterization

We first test our ROPUF implementation separately to obtain its quality parameters (p_e, ρ). This characterization also produces the $\tilde{\mu}_{F_i}$ normalization terms required in the final key generator implementation as detailed in Section 3.2.

We configured and tested exactly the same PUF implementation on 10 identical FPGAs, using an ROPUF design with $b = 16$ batches of $a = 64$ oscillators each.

The frequency measurements are outputted directly and we perform all post-processing described in Section 3.2 offline, using MATLAB[2]. To characterize the noise, the frequency of every loop is measured 25 times. For the moment we don't consider entropy compression, so the PUF response X^{ℓ} has length $\ell = \ell' = \sum_{i=2}^{b} \lceil \log_2 i \rceil = 49$ bits with an assumed entropy of $H(X^{\ell}) = H(Y^{\ell'}) = \log_2 b! = 44.25$ bits, yielding an entropy density of $\rho = 90.31\%$. In Fig. 4(a), the inter- and intra-distance histogram plots of these responses are presented. The average inter-distance between responses on different devices is about 23.7 in 49 bits or about 48.4%. The small deviation from the ideal of 50% is representative for the responses only having 90% entropy. At room temperature, the average intra-distance between measurements of the same response on a single device is just below 1 in 49 bits or merely 2.0%. ROPUFs are known to become more unstable under temperature changes. To estimate this effect, we performed a rough temperature test using a thermoelectric element to heat the FPGA's die temperature to about 80°C and cool it to about 10°C. We measured the intra-distances with respect to a room temperature reference. We also studied the effect of the XOR-compression on the ROPUF's response robustness, by compressing the response lengths to $\ell = 42$ (ρ becomes 97.95%) and $\ell = 40$ (ρ becomes 98.78%). Fig. 4(b) shows the effect of both temperature and XOR-compression on the average bit error probability. Heating the FPGA die has the most severe impact on the stability of the ROPUF's responses. As expected, XOR-compression also slightly increases the bit error probability, approximately by a factor $\frac{\ell'}{\ell}$. Taking into account a 2% safety margin on the observed bit error rates, our ROPUF implementation yields a $(p_e = 12\%, \rho = 90.31\%)$-PUF for $\ell = 49$, or a $(13\%, 97.95\%)$-PUF for $\ell = 42$, or a $(14\%, 98.78\%)$-PUF for $\ell = 40$.



(a) Inter- and intra-distance histogram plots at room temperature, for $\ell = 49$.

(b) Scaling of p_e under heating/cooling and entropy compression to $\ell = 42$ and $\ell = 40$.

Fig. 4. Characterization of our ROPUF implementation

[2] In the final PUFKY implementation, all post-processing is done on the device.

4.2 Full Key Generator Implementation

Now we can start optimizing the full PUFKY design according to the constraints as expressed in Section 2.3. The main cost variable for implementation size is the number of required oscillators ($a \times b$), and for performance the number of errors the BCH decoder needs to correct (t_{BCH}). Since we target embedded systems, we aim for an as small as possible implementation at a practically acceptable performance. The optimization parameters depend on our ROPUF, expressed by the triplet (ℓ, p_e, ρ) for which concrete values are provided at the end of Section 4.1, and on the requirements for the generated key, expressed by (m, p_{fail}). For our reference implementation, we aim for a key length $m = 128$ with failure rate $p_{fail} \leq 10^{-9}$. After a thorough exploration of the design space with these parameters, we converge on the following PUFKY reference implementation:

- We select the $(p_e = 13\%, \rho = 97.95\%)$-ROPUF variant with $\ell = 42$, implementing $b = 16$ batches of $a = 53$ oscillators each.
- A secure sketch applying a concatenation of $\mathcal{C}_{REP}(7, 1, 3)$ and $\mathcal{C}_{BCH}(318, 174, 17)$. The repetition block generates 36 bits of helper data for every 42-bit PUF response and outputs 6 bits to the BCH block. The BCH block generates 144 bits of helper data once and feeds 318 bits to the entropy accumulator.
- The ROPUF generates in total $a \times \ell = 2226$ bits containing $a \times \ell \times \rho = 2180.4$ bits of entropy. The total helper data length is $53 \times 36 + 144 = 2052$. The remaining entropy after secure sketching is at least $2180.4 - 2052 = 128.4$ bits which are accumulated in an $m = 128$-bit key by a SPONGENT-128 hash function implementation.

The total size of our PUFKY reference implementation for the considered FPGA platform is 1162 slices, of which 82% is taken up by the ROPUF block. Table 2(a) lists the size of each submodule used in the design. The total time spend to extract the 128-bit key is approximately 5.62 ms (at 54 MHz). Table 2(b) lists the number of cycles spend in each step of the key extraction.

Table 2. Area consumption and runtime of our reference PUFKY implementation on a Xilinx Spartan-6 FPGA. Due to slice compression and glue logic the sum of module sizes is not equal to total size. The PUF runtime is independent of clock speed.

(a) Area consumption

Module	**Size [slices]**
ROPUF	952
REP decoder	37
BCH syndrome calc.	72
BCH decoder	112
SPONGENT-128	22
helper data RAM	38
Total	1162

(b) Runtimes

Step of extraction	**Time [cycles]**
PUF output	4.59 ms
REP decoding	0
BCH syndrome calc.	511
BCH decoding	50320
SPONGENT hashing	3990
control overhead	489
Total @ 54 *MHz*	5.62 ms

5 Discussion

5.1 Some Notes on Security

Our reference PUFKY implementation uses a best-practice entropy accumulation function based on a cryptographically secure hash to generate a key from an amount of entropic data, instead of an information-theoretically secure fuzzy extractor. The large majority of currently existing key generators based on PRNGs also use the best-practice cryptographic approach. We note that, due to the modularity of the PUFKY design, it is possible to obtain an information-theoretically secure extraction with minor replacements: *i)* one needs to consider min-entropy instead of Shannon entropy in all design constraints, *ii)* one needs to replace the entropy accumulation function by a strong extractor, and *iii)* one needs to collect more (min-)entropy than the key length to account for the additional losses induced by the strong extractor. Note that all three changes do come at a rather large implementation overhead, which is the cost one pays for obtaining information-theoretical security.

From a physical security perspective, PUFs and PUF-based key generators can be assumed, like any implementation of a cryptographic primitive, to be vulnerable to side-channel attacks when no appropriate countermeasures are taken, see e.g. [11, 17]. Since our PUFKY reference implementation is a fully functional PUF-based key generator, it is the ideal test subject for side-channel analysis to identify and protect against possible side-channel leakages in a next version. Such analysis is a logical future work which we are considering. In this light, we do want to mention the inherent side-channel resistance of the error decoding blocks in syndrome-construction secure sketches. This results from the fact that no data processed by these blocks contains any information about the PUF output nor about the extracted key, but only about the public syndrome and the error on the PUF output.

5.2 Application Possibilities

The key generated by our PUFKY key generator can basically be used in any conceivable key-based security application. In its current form, the reference implementation produces cryptographically strong 128-bit keys with a failure rate $< 10^{-9}$, but similar implementations for other key parameters (or alternative PUF designs) can be produced rapidly based on our modular PUFKY architecture. Using a PUF-based key offers a number of advantages over traditional key generation, the most noteworthy being: *i)* one does not need protected non-volatile memory to permanently store the key since it can be regenerated at any time, and *ii)* the key is intrinsically bound to a particular platform instantiation which is very useful, e.g. in anticounterfeiting or HW/SW binding applications. We note that both advantages are of particular interest in the context of an FPGA-based embedded system. To demonstrate the ease of integrating a PUFKY implementation in an embedded design, we developed a bus wrapper and a software driver for connecting it to a Xilinx® MicroBlaze® embedded processor. The PUFKY interface then becomes as simple as calling the driver's `getKey()` function from one's embedded software application.

6 Conclusion

Developing a PUF-based cryptographic key generator is a process involving many parameters, constraints and trade-offs. In this work, we identified and formalized the generic design constraints and integrated them in a practical key generator design. We propose a complete implementation of this design based on a ring-oscillator PUF, a specialized error-correcting BCH decoder and a cryptographic entropy accumulator. Our ring-oscillator PUF produces high-entropy responses (up to 99%) based on actual physical randomness. The proposed BCH decoder design is very efficient and scalable, yet occuppies only a minimal amount of resources. As our implementation results demonstrate, the induced overhead of this BCH decoder in a PUF based key generator is certainly justifiable. Finally, the choice for a cryptographic entropy accumulator, motivated by their wide-spread use in PRNG based key generators, offers a considerable efficiency gain compared to the much more stringent design constraints for information-theoretically secure key extraction. Due to its completeness and efficieny, our PUFKY reference implementation is the first PUF-based key generator to be immediately deployable in an embedded system.

Acknowledgements. This work was supported in part by the Research Council KU Leuven: GOA TENSE (GOA/11/007), by the IAP Programme P6/26 BCRYPT of the Belgian State (Belgian Science Policy) and by the European Commission through the ICT programme under contract ICT-2007-216676 ECRYPT II. In addition, this work was supported by the Flemish Government, FWO G.0550.12N and by the European Commission through the ICT programme under contract FP7-ICT-2011-284833 PUFFIN and FP7-ICT-2007-238811 UNIQUE. Roel Maes is funded by a research grant (073369) of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen).

References

[1] Barker, E., Kelsey, J.: Recommendation for Random Number Generation Using Deterministic Random Bit Generators. NIST Special Publication 800-90A (January 2012), `http://csrc.nist.gov/publications/nistpubs/800-90A/SP800-90A.pdf`

[2] Berlekamp, E.: On Decoding Binary Bose-Chadhuri-Hocquenghem Codes. IEEE Transactions on Information Theory 11(4), 577–579 (1965)

[3] Bogdanov, A., Knežević, M., Leander, G., Toz, D., Varıcı, K., Verbauwhede, I.: SPONGENT: A Lightweight Hash Function. In: Preneel, B., Takagi, T. (eds.) CHES 2011. LNCS, vol. 6917, pp. 312–325. Springer, Heidelberg (2011)

[4] Bösch, C., Guajardo, J., Sadeghi, A.-R., Shokrollahi, J., Tuyls, P.: Efficient Helper Data Key Extractor on FPGAs. In: Oswald, E., Rohatgi, P. (eds.) CHES 2008. LNCS, vol. 5154, pp. 181–197. Springer, Heidelberg (2008)

[5] Burton, H.: Inversionless Decoding of Binary BCH codes. IEEE Transactions on Information Theory 17(4), 464–466 (1971)

[6] Chien, R.: Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. IEEE Transactions on Information Theory 10(4), 357–363 (1964)

[7] Dodis, Y., Ostrovsky, R., Reyzin, L., Smith, A.: Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data. SIAM Journal on Computing 38(1), 97–139 (2008)

[8] Eastlake, D., Schiller, J., Crocker, S.: Randomness Requirements for Security. RFC 4086 (Best Current Practice) (June 2005), http://www.ietf.org/rfc/rfc4086.txt

[9] Gassend, B., Clarke, D., van Dijk, M., Devadas, S.: Silicon Physical Random Functions. In: ACM Conference on Computer and Communications Security, pp. 148–160. ACM Press (2002)

[10] 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)

[11] Karakoyunlu, D., Sunar, B.: Differential Template Attacks on PUF Enabled Cryptographic Devices. In: 2010 IEEE International Workshop on Information Forensics and Security (WIFS), pp. 1–6 (December 2010)

[12] Kelsey, J., Schneier, B., Ferguson, N.: Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator. In: Heys, H.M., Adams, C.M. (eds.) SAC 1999. LNCS, vol. 1758, pp. 13–33. Springer, Heidelberg (2000)

[13] Lenstra, A.K., Hughes, J.P., Augier, M., Bos, J.W., Kleinjung, T., Wachter, C.: Ron was wrong, Whit is right. Cryptology ePrint Archive, Report 2012/064 (2012), http://eprint.iacr.org/

[14] 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.) CHES 2009. LNCS, vol. 5747, pp. 332–347. Springer, Heidelberg (2009)

[15] Maiti, A., Casarona, J., McHale, L., Schaumont, P.: A Large Scale Characterization of RO-PUF. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 94–99 (June 2010)

[16] Massey, J.: Shift-Register Synthesis and BCH Decoding. IEEE Transactions on Information Theory 15(1), 122–127 (1969)

[17] 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.) TRUST 2011. LNCS, vol. 6740, pp. 33–47. Springer, Heidelberg (2011)

[18] Nisan, N., Zuckerman, D.: Randomness is Linear in Space. Journal of Computer and System Sciences 52, 43–52 (1996)

[19] Park, J.I., Lee, H., Lee, S.: An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders. In: IEEE International Symposium on Circuits and Systems (ISCAS), pp. 2693–2696 (May 2011)

[20] Park, J.I., Lee, K., Choi, C.S., Lee, H.: High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm. In: International SoC Design Conference (ISOCC), pp. 452–455 (November 2009)

[21] Reed, I., Shih, M.: VLSI Design of Inverse-Free Berlekamp-Massey Algorithm. IEEE Proceedings on Computers and Digital Techniques 138(5), 295–298 (1991)

[22] Sarwate, D., Shanbhag, N.: High-Speed Architectures for Reed-Solomon Decoders. IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9(5), 641–655 (2001)

[23] Suh, G.E., Devadas, S.: Physical Unclonable Functions for Device Authentication and Secret Key Generation. In: Design Automation Conference (DAC), pp. 9–14. ACM Press (2007)

[24] Tarnovsky, C.: Deconstructing a 'Secure' Processor. In: Black Hat Federal 2010 (2010)

[25] Torrance, R., James, D.: The State-of-the-Art in IC Reverse Engineering. In: Clavier, C., Gaj, K. (eds.) CHES 2009. LNCS, vol. 5747, pp. 363–381. Springer, Heidelberg (2009)

[26] Wong, K., Chen, S.: The Entropy of Ordered Sequences and Order Statistics. IEEE Transactions on Information Theory 36(2), 276–284 (1990)

[27] Yin, C.E.D., Qu, G.: LISA: Maximizing RO PUF's Secret Extraction. In: IEEE International Symposium on Hardware-Oriented Security and Trust (HOST), pp. 100–105 (June 2010)

[28] Yu, M.-D(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.) CHES 2011. LNCS, vol. 6917, pp. 358–373. Springer, Heidelberg (2011)

A BCH Decoding Algorithms

Listed below are the three algorithms that we use for BCH decoding. More information on these algorithms and how they are used can be found in Section 2.4. We denote an array A of b elements, with each element in $\mathbb{N}$ as $A[b] \in \mathbb{N}$. Array indices start at 0, unless specifically mentioned otherwise. Both Algorithm 1 and 2 amount to polynomial evaluation, however, in the former, heavy optimization is possible since we know that every coefficient must be either 0 or 1. Algorithm 3 is the same as the one presented in [5], with a few modifications to better fit the architecture of our coprocessor.

Algorithm 1: Syndrome calculation

```
Input: $s^{n-k}[n-k] \in \mathbb{F}_2$
Output: $z[2t] \in \mathbb{F}_{2^u}$
Data: $\mathsf{curArg}, \mathsf{evalArg} \in \mathbb{F}_{2^u}$
      $i, j \in \mathbb{N}$
$\mathsf{curArg} \leftarrow \alpha$
for $i \leftarrow 0$ to $2t-1$ do
    $z[i] \leftarrow 0$
    $\mathsf{evalArg} \leftarrow 1$
    for $j \leftarrow 0$ to $n-k-1$ do
        if $s^{n-k}[j] = 1$ then
            $z[i] \leftarrow z[i] \oplus \mathsf{evalArg}$
        $\mathsf{evalArg} \leftarrow \mathsf{evalArg} \otimes \mathsf{curArg}$
    $\mathsf{curArg} \leftarrow \mathsf{curArg} \otimes \alpha$
```

Algorithm 2: Chien search

```
Input: $\Lambda[t+1] \in \mathbb{F}_{2^u}$
Output: $\mathsf{errorLoc}[n] \in \mathbb{F}_2$
Data: $\mathsf{curAlpha}, \mathsf{curEval} \in \mathbb{F}_{2^u}$
      $i, j \in \mathbb{N}$
for $i \leftarrow n-1$ to $0$ do
    $\mathsf{curEval} \leftarrow \Lambda[0]$
    $\mathsf{curAlpha} \leftarrow \alpha$
    for $j \leftarrow 1$ to $t$ do
        $\Lambda[j] \leftarrow \Lambda[j] \otimes \mathsf{curAlpha}$
        $\mathsf{curEval} \leftarrow \mathsf{curEval} \oplus \Lambda[j]$
        $\mathsf{curAlpha} \leftarrow \mathsf{curAlpha} \otimes \alpha$
    if $\mathsf{curEval} = 0$ then
        $\mathsf{errorLoc}[i] \leftarrow 1$
    else
        $\mathsf{errorLoc}[i] \leftarrow 0$
```

Algorithm 3: Inversionless Berlekamp-Massey

```
Input: $z[2t] \in \mathbb{F}_{2^u}$
Output: $\Lambda[t+1] \in \mathbb{F}_{2^u}$
Data: $\mathsf{b}[t+2], \delta, \gamma \in \mathbb{F}_{2^u}; \mathsf{flag} \in \mathbb{F}_2; \mathsf{k} \in \mathbb{Z}; i, j \in \mathbb{N}$
$\mathsf{b}[-1] \leftarrow 0$
$\mathsf{b}[0] \leftarrow 1$
$\Lambda[0] \leftarrow 1$
for $i \leftarrow 1$ to $t$ do
    $\mathsf{b}[i] \leftarrow 0$
    $\Lambda[i] \leftarrow 0$
$\gamma \leftarrow 1$
$\mathsf{k} \leftarrow 0$
for $i \leftarrow 0$ to $2t-1$ do
    $\delta \leftarrow 0$
    for $j \leftarrow 0$ to $\mathtt{min}(i, t)$ do
        $\delta \leftarrow \delta \oplus (z[i-j] \otimes \Lambda[j])$
    $\mathsf{flag} \leftarrow (\delta \neq 0)$ & $(\mathsf{k} \geq 0)$
    if $\mathsf{flag} = 1$ then
        for $j \leftarrow t$ to $0$ do
            $\mathsf{b}[j] \leftarrow \Lambda[j]$
            $\Lambda[j] \leftarrow (\Lambda[j] \otimes \gamma) \oplus (\mathsf{b}[j-1] \otimes \delta)$
        $\gamma \leftarrow \delta$
        $\mathsf{k} \leftarrow -\mathsf{k} - 1$
    else
        for $j \leftarrow t$ to $0$ do
            $\mathsf{b}[j] \leftarrow \mathsf{b}[j-1]$
            $\Lambda[j] \leftarrow (\Lambda[j] \otimes \gamma) \oplus (\mathsf{b}[j-1] \otimes \delta)$
        $\mathsf{k} \leftarrow \mathsf{k} + 1$
```

Table 3 lists formulas for the ideal and actual runtime of each algorithm. We define the ideal algorithm runtime as the total number of (not unrolled) loop iterations. Note that runtime is mainly determined by t_{BCH} in all three algorithms.

Our obtained runtimes of the syndrome and error-location calculation are particularly efficient requiring only 3–5 cycles per loop iteration with well chosen parameters for $\mathcal{C}_{BCH}$.

Table 3. The ideal and actual runtimes for the BCH decoding algorithms. The formulas for actual runtime are highest order approximations.

Algorithm	**Runtime [cycles]**	
	Ideal	**Actual (approx.)**
Syndrome calculation	$2t_{BCH} \cdot (n_{BCH} - k_{BCH})$	$40t_{BCH} \cdot \lceil \frac{n_{BCH}-k_{BCH}}{u} \rceil$
Berlekamp-Massey	$3.5 \cdot (t_{BCH}^2 + t_{BCH})$	$36t_{BCH}^2$
Error loc. calculation	$n_{BCH} \cdot t_{BCH}$	$3.6n_{BCH} \cdot t_{BCH}$

B BCH Decoder Instruction Set

Table 4 gives an overview of the instructions implemented on the BCH decoding coprocessor, their result and the number of cycles needed to execute each instruction.

Table 4. Instruction set of the BCH decoding coprocessor

Opcode	**Result**	**Cycles**
`jump`	PC $\leftarrow$ *value*	2
`cmp_jump`	PC $\leftarrow$ *value* **if** (*comp* = **true**)	3
`stop`	PC $\leftarrow$ PC	1
`comp`	$\text{cond}_i \leftarrow$ (*comp* = **true**)	2
`set_cond`	$\text{cond}_i \leftarrow$ *value*	1
`load_reg`	reg $\leftarrow \text{data}[\text{addr}_i]$	1
`load_fixed_reg`	reg $\leftarrow$ *value*	2
`load_fixed_addr`	$\text{addr}_i \leftarrow$ *value*	2
`mod_addr`	$\text{addr}_i \leftarrow f(\text{addr}_i)$	1
`copy_addr`	$\text{addr}_i \leftarrow \text{addr}_j$	1
`store_reg`	$\text{data}[\text{addr}_i] \leftarrow$ reg	1
`store_fixed`	$\text{data}[\text{addr}_i] \leftarrow$ *value*	2
`rotr`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \circlearrowleft 1$	1
`shiftl_clr`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \ll 1$	1
`shiftl_set`	$\text{data}[\text{addr}_i] \leftarrow (\text{data}[\text{addr}_i] \ll 1) \mid 1$	1
`gf2_add_mult`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j]$ $\text{reg} \leftarrow \text{reg} \oplus (\text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j])$	1