KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Lightweight PUF-based Key and Random Number Generation

Anthony Van Herrewege

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering.

January 2015

Lightweight PUF-based Key and Random Number Generation

Anthony VAN HERREWEGE

Supervisor:
Prof. dr. ir. Ingrid Verbauwhede

Members of the
Examination Committee:
Prof. dr. ir. Pierre Verbaeten
Chair
Prof. dr. ir. Jean Berlamont
Chair
Prof. dr. ir. Bart Preneel
Prof. dr. ir. Wim Dehaene
Prof. dr. ir. Frank Piessens
Prof. dr. Stefan Katzenbeisser
Technische Universität Darmstad, DE
Prof. dr. ing. Patrick Schaumont
Virginia Polytechnic Institute and State University, USA

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering.

January 2015

Uitgegeven in eigen beheer, Anthony Van Herrewege, Kasteelpark Arenberg 10 bus 2452, 3001 Heverlee (Belgium).

ISBN 978-94-6018-947-0

D/2015/7515/4

§ Preface

This is how you do it: you sit down at the keyboard
and you put one word after another until it's done.
It's that easy, and that hard.
— Neil Gaiman, Personal journal (2 May 2004)

Writing quickly is surely not my forte, but fortunately now only this one page remains unfinished. After what seems like ages, I can almost put down the proverbial pen! So let me use this opportunity to thank all of those who have been influential in the making of this thesis and all that lead up to it.

Not a single word of this work would have been written, were it not for my supervisor, prof. Ingrid Verbauwhede, who graciously offered me a position as PhD researcher. For that, and for her letting me find my own direction in research, I am very grateful.

I would also like to thank my assessors, prof. Bart Preneel and prof. Frank Piessens, the other members of my jury, prof. Wim Dehaene, prof. Stefan Katzenbeisser and prof. Patrick Schaumont, and the jury chairmen prof. Pierre Verbaeten and prof. Jean Berlamont. They sacrificed part of their valuable time to sit on my examination committee, and their remarks have certainly helped improve the quality of this thesis.

In particular, I would like to thank my many colleagues at COSIC, not only for contributing to a great research environment, but also for Alma lunches, karting events, the COSIC weekend, summer BBQs, and even the occasional climbing outing.

To the people who have co-authored papers with me over the years or whom I have worked together with on projects, such as UNIQUE and PUFFIN: it has been an honor.

Finally, an utmost thanks to my family and all of my friends for always being there when they are most needed. With you around, life is never boring. You are fantastic!

Anthony Van Herrewege
Leuven, January 2015

§ Abstract

As embedded electronics continue to be integrated into our daily lives at such a pace that there are nowadays more cellphones than people on the planet, security is becoming ever more crucial. Unfortunately, this is all too often realized as an afterthought and thus the security implementations in many embedded devices offer little to no practical protection. Security does not require only cryptographic algorithms; two other critical modules in a secure system are a key generation module and a random number generator (RNG). The lack of well thought-out implementations of these modules has been the downfall of the security in many devices, many of them high-profile.

In this thesis, we look into ways of constructing secure versions of both of these building blocks in embedded devices. Towards this end, we turn our attention to physically unclonable functions (PUFs). A PUF is a promising, relatively novel primitive that functions as a fingerprint for electronic devices. In our research, we have combined PUFs with custom hardware modules, such as a BCH error correcting code decoder, to create the first "black box" PUF-based key generation module. Our implementation requires very little real estate, proving that very efficient BCH error correcting codes, which are normally written off as being unwieldy and complex, are in fact feasible for use in PUF-based systems.

We furthermore investigate the presence of PUFs in commercial off-the-shelf (COTS) microcontrollers. A thorough investigation of the usability of SRAM as PUFs and RNGs in a handful of the most prominent microcontroller families on the market is presented. We discuss the practical use of the measured microcontrollers in light of our findings, and show that there are large differences between the various families. Our study is the first of its kind, and clearly displays the need for continued work in this fashion on other microcontrollers.

Finally, we develop a system for a secure RNG on COTS embedded devices, leveraging errors in available PUFs as a source of entropy. Building upon the findings of our microcontroller study, we successfully implement this system onto various ARM Cortex-M microcontrollers. Part of this result is an implementation of the Keccak algorithm, the smallest published to date.

§ Samenvatting

In onze moderne leefwereld, waarin er meer smartphones dan mensen op de planeet zijn en de opmars van geïntegreerde elektronica gestaag blijft toenemen, is beveiliging onontbeerlijk. Helaas wordt dat bij het ontwerpen van apparaten vaak te laat gerealiseerd, waardoor de beveiliging van vele geïntegreerde toestellen vaak amper tot geen bescherming biedt. Beveiliging bestaat uit meer dan cryptografie, twee andere noodzakelijke bouwblokken zijn een sleutelgeneratie module en een willekeurige nummergenerator (RNG). Door een gebrek aan weldoordachte implementaties van deze modules werd de afgelopen jaren de beveiliging van vele apparaten gebroken, wat in verschillende gevallen tot grote kosten leidde.

In deze thesis onderzoeken we methodes om veilige versies te maken van beide bovengenoemde bouwblokken, voor gebruik in geïntegreerde elektronica. Hiervoor richten we onze aandacht op fysisch onkloonbare functies (PUFs). PUFs zijn veelbelovende, redelijk moderne cryptografische primitieven die functioneren als een vingerafdruk voor elektronica. In ons onderzoek combineren we PUFs met op maat gemaakte hardware, zoals een BCH foutcorrectie module, om zo te komen tot 's wereld's eerste "black box" PUF-gebaseerde sleutelgeneratiemodule. Onze implementatie vereist zeer weinig oppervlakte, wat bewijst dat BCH foutcodes van praktisch nut zijn voor PUF-gebaseerde systemen, hoewel zulke codes doorgaans afgeschreven worden als zijnde te complex.

Verder onderzoeken we de aanwezigheid van PUFs in commercieel verkrijgbare microcontrollers. We presenteren een grondig onderzoek naar het gebruik van SRAM als PUFs en RNGs in een aantal van de belangrijkste microcontrollerfamilies. Onze resultaten tonen aan dat er drastische verschillen zijn tussen deze families, ze zijn niet allen geschikt voor veilige implementaties. Ons onderzoek is het eerste in zijn soort en demonstreert een duidelijke nood aan voortgezet werk van deze aard.

Ten slotte ontwikkelen we een systeem voor een veilige, software-gebaseerde RNG op microcontrollers, gebruikmakend van fouten in aanwezige PUFs als bron van entropie. Voortbouwend op de bevindingen van onze microcontrollerstudie, implementeren we dit systeem succesvol op verscheidene ARM Cortex-M microcontrollers. Een deel van dit werk is de, tot nu toe, kleinste implementatie van het Keccak algoritme.

§ Contents

List of Figures

§ List of Tables

§ List of Code Listings

§ List of Abbreviations

CBC	cipher-block chaining
CMOS	complementary metal-oxide-semiconductor
COTS	commercial off-the-shelf
DRNG	RNG module in Intel's modern processors
ECB	electronic codebook
ECC	error correcting code
ECC	elliptic curve cryptography
i.i.d.	independent and identically distributed
ISA	instruction set architecture
IV	initialization vector
LFSR	linear feedback shift register
MAC	message authentication code
MOSFET	metal-oxide-semiconductor field-effect-transistor
MUX	multiplexer
PRNG	pseudo-random number generator
PUF	physically unclonable function
RDF	random dopant fluctuation
RNG	random number generator
RSA	public key encryption algorithm by Rivest, Shamir, and Adleman

SRAM	static random-access memory
SSL	Secure Sockets Layer
TRNG	true random number generator

§ List of Symbols

$A[i]$	element i of array A
$\mathrm{B}_{n,p}(t)$	binomial cumulative distribution function with a number of trials n and trial success probability p evaluated in t
$\mathrm{B}^{-1}_{n,p}(q)$	inverse of binomial cumulative distribution function
$\otimes$	bitwise AND
$\sim$	bitwise inversion
$\lvert a \rvert$	length of bit string a
$A \lll i$	bitwise rotation of A i positions to the left
$\oplus$	bitwise XOR
$\rho_\infty(X)$	fractional min-entropy or entropy density of X
$\rho_1(X)$	fractional Shannon entropy or entropy density of X
$H_\infty(X)$	min-entropy of X
$H_1(X)$	Shannon entropy of X
$\mathbb{F}_{2^n}$	binary Galois field
$\oplus$	addition in Galois field
$\otimes$	multiplication in Galois field
$\mathrm{ord}(A)$	order of polynomial A
$\mathrm{HD}(a,b)$	Hamming distance between bitstrings a and b
$\mathrm{HW}(a)$	Hamming weight of bitstring a
$C(n,k,t)$	linear error-correcting code with length n, dimension k and number of correctable errors t
$\mathcal{G}$	generator polynomial for polynomial error-correcting code

$\mathcal{P}$	PUF design
puf_i	instance i of PUF design
$\boldsymbol{Y}_i(x)$	group of PUF response measurements on puf_i
$Y_i(x)$	response of puf_i to challenge x
$Y_{i;\alpha}(x)$	response of puf_i to challenge x under operating condition α
$Y_i^{(j)}(x)$	response number j of puf_i to challenge x
$\mathcal{X}_\mathcal{P}$	set of all challenges of $\mathcal{P}$
$\mathcal{Y}_\mathcal{P}$	set of all responses of $\mathcal{P}$

1 Introduction

This method, seemingly very clever, actually played into our hands!
And so it often happens that an apparently ingenious idea is in fact a weakness
which the scientific cryptographer seizes on for his solution.
— Herbert Yardley, The American Black Chamber (1931)

Modern societies see an ever increasing use of portable computing devices, the adoption of a growing number of smart appliances, and a pervasive presence of electronics. This creates a strong need for security solutions tailored to embedded devices. Because most of these devices are required to be either as small, as power- or as energy-efficient as possible, suitable cryptographic implementation techniques differ from those used in e.g. powerful desktop computers.

Crucial to the security of a cryptographic systems are the quality of the keys, random numbers, nonces and initialization vectors. We start this chapter with use cases of three cryptographic primitives used in most cryptographic applications. These examples illustrate the importance of the aforementioned keys, etc. This point is driven home further by a discussion on recent cryptographic security failures of systems lacking exactly those components.

The second part of the chapter outlines the contents of this thesis which focuses on efficient implementations for the generation of high quality keys and random numbers on embedded devices. The contributions in this context are *i*) an elaborate set of measurements on memory behavior for PUF purposes in the most popular families of off-the-shelf microcontrollers; *ii*) the, at time of writing, smallest software implementation of the hash function Keccak for microcontrollers; *iii*) a lightweight, practical implementation for strong random number generation on microcontrollers; and *iv*) an area efficient design and implementation for high quality PUF-based key generation.

1.1 Cryptographic primitives

In this first section, we explain three cryptographic primitives: symmetric cryptography, asymmetric cryptography, and hash functions. Most, if not all, cryptographic protocols require one or more of these primitives. For each of these primitives, we give a typical use case example, which demonstrates the importance of two components that are often overlooked or (falsely) assumed present: secure keys and a strong random number generator.

1.1.1 Symmetric key cryptography

Symmetric key cryptography is the workhorse of modern cryptography, and was the only known type of crypto until the invention of asymmetric cryptography.

Definition 1.1. *A cryptographic algorithm is symmetric if it is computationally easy to determine from the encryption key k_e the decryption key k_d, and vice versa.*[86, p. 15]

For the majority of such algorithms, the encryption and decryption keys are the same. The basic principle of symmetric crypto is illustrated in Figure 1.1. The sender, Alice, encrypts a plaintext P, i.e. the message, using a shared key k. The encrypted message C, i.e. the ciphertext, is then sent to Bob. He can decrypt C and recover P, because he knows the same key k. An eavesdropper, Eve, who does not know k, learns nothing about the plaintext P from the ciphertext C, apart from perhaps its length.

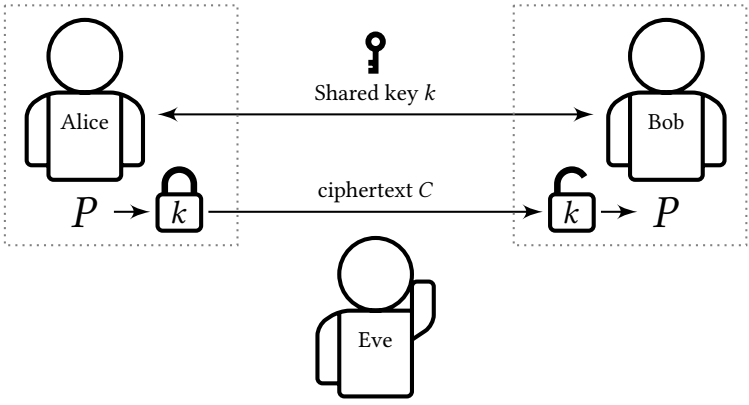


Figure 1.1: The working principle of symmetric key cryptography.

[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

Symmetric key crypto has been in use since ancient times, first by the Babylonians starting around 500 B.C.,[61] later on famously by Julius Caesar[33] to keep military messages secret. These ancient ciphers, and a myriad of more recent ones, have not been able to withstand the test of time, and have been broken by cryptanalytic methods. Nowadays, the two most used symmetric ciphers are 3DES,[9] mostly in the financial industry, and AES,[30] the de facto standard used in the majority of protocols where a symmetric cryptography algorithm is required.

Symmetric crypto is very fast, which is one of the major reasons why it is the workhorse of modern crypto. The biggest drawback is the necessity for all parties to have knowledge of the same secret key. Naturally, such a situation leads to a chicken and egg problem: how does one securely transfer a secret key to other parties if there is no established secret key yet?

In general, symmetric crypto algorithms are divided into two groups, block ciphers and stream ciphers. We will discuss both very concisely in the next few paragraphs.

Definition 1.2. *A stream cipher is an encryption scheme that generates one symbol (i.e. a bit) of the keystream at a time, based on an internal state. This symbol is then combined with the plaintext, usually with a XOR operation.*[86, p. 20]

The main advantage of stream ciphers is that they are generally fast and compact. This is because they only have to generate a single symbol of the keystream at a time, and hence the mathematical operations which they have to execute at each time step are limited. Furthermore, since ciphertext bits are encrypted independent from one another, losing a ciphertext bit during transmission only leads to that bit being lost, it has no impact on the decryption of other ciphertext bits.

However, care should be taken to never generate a keystream from the same initial value, since this makes the generated ciphertexts extremely vulnerable to attack.

Due to their properties, stream ciphers are often used in communication systems, where high throughput is important. For example, the RC4[105] cipher, which is now considered broken, has been used for a very long time to encrypt transmissions to and from secure websites on the internet. Certain companies, such as Google, are now using another stream cipher, Salsa20,[13] to encrypt such transmissions.

[9] W. C. Barker, "Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher" (2004).
[13] D. J. Bernstein, "The Salsa20 Family of Stream Ciphers" (2008).
[30] J. Daemen and V. Rijmen, "The Design of Rijndael: AES – The Advanced Encryption Standard" (2002).
[33] C. A. Deavours *et al.*, "Cryptology: Yesterday, Today, and Tomorrow" (1987).
[61] C. D. Isbell, "Some Cryptograms in the Aramaic Incantation Bowls" (1974).
[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).
[105] R. L. Rivest and J. C. N. Schuldt, "Spritz—A spongy RC4-like stream cipher and hash function" (2014).

Another example of a famous stream cipher is A5/1,[48] which was one of the original ciphers used to encrypt transmissions over the cellular GSM network. Unfortunately, this cipher has been broken as well.

Definition 1.3. *A block cipher is an encryption scheme that breaks up the plaintext into strings (called blocks) of a fixed length l and encrypts these one at a time![86, p. 16]*

Block ciphers are very flexible building blocks, not only can they be used to encrypt data, but also to construct e.g. message authentication codes, hash functions, or pseudo-random number generators. These concepts will be explained further on. They can even be used to construct stream ciphers. Hence, block ciphers are not limited to encryption of fixed length data blocks.

The most famous block cipher nowadays is undoubtedly AES,[30] which is used in the majority of protocols that utilize a block cipher. As stated before, many financial institutions still use 3DES[9] for encryption of data.

Block ciphers are always used in so-called modes of operation.[86, p. 228] The most straightforward mode to use a block cipher in is by inputting data block by block and then concatenating the generated result. This is called electronic codebook (ECB) mode, and is illustrated in Figure 1.2.

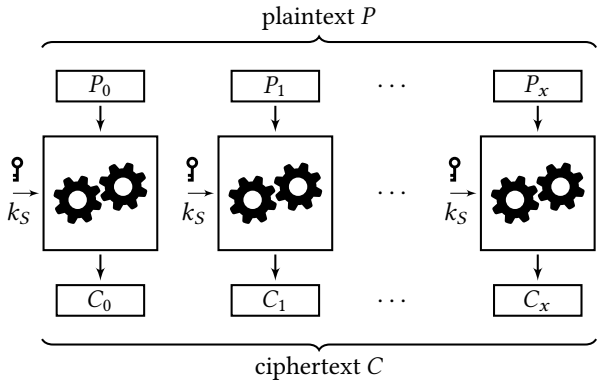


Figure 1.2: Block cipher encrypting data in ECB mode.

[9] W. C. Barker, "Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher" (2004).
[30] J. Daemen and V. Rijmen, "The Design of Rijndael: AES – The Advanced Encryption Standard" (2002).
[48] J. D. Golic, "Cryptanalysis of Alleged A5 Stream Cipher" (1997).
[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

However, ECB mode is notorious for its insecurity, because identical blocks of plaintext all get encrypted to the same value. This reveals patterns in the plaintext and is thus clearly not secure. In order to solve this problem, there are many other modes that one can use, for example the cipher-block chaining (CBC) mode, shown in Figure 1.3. One important difference between ECB and CBC mode is that CBC mode requires an extra input, a so-called initialization vector (IV). An IV can be made public, although it is recommended to keep it a secret. IVs should always be selected at random and used only once, otherwise the security of the algorithm implementation in which it is used is (severely) weakened.

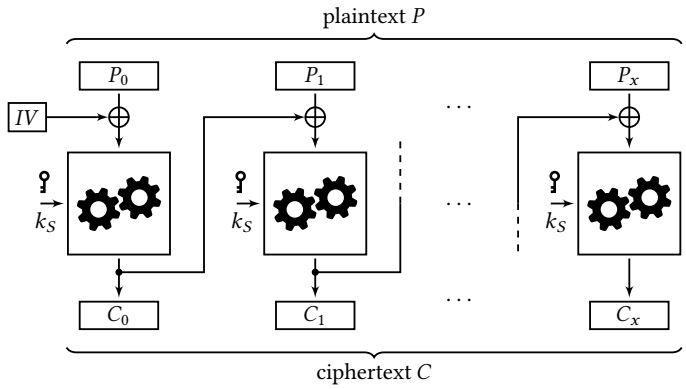
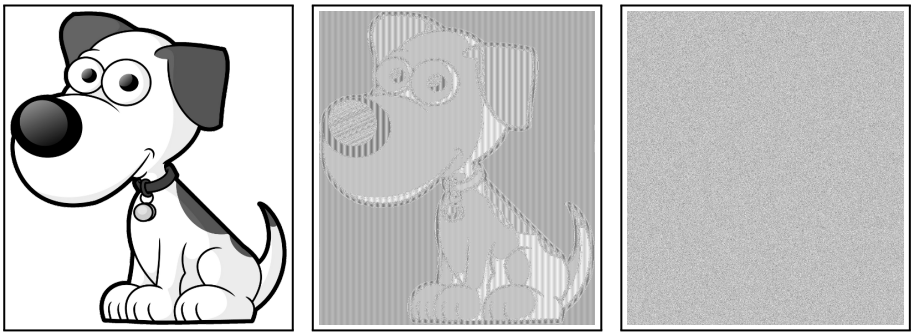


Figure 1.3: Block cipher encrypting data in CBC mode.

Example An illustration of the difference between an image encrypted with AES in ECB and CBC mode is shown in Figure 1.4. It is clear that even though the correct colors cannot be seen in the ECB encrypted image, it is still trivial to identify the shape of the unencrypted image. Thus, a lot of information about the plaintext, the unencrypted image, is leaked. The CBC encrypted image, on the other hand, looks like random data: it is impossible to make out the original image without knowledge of the key used to encrypt it.

There are various other secure modes in which a block cipher can be used, all with their specific pros and cons. The difference between ECB mode and secure modes is that all of the latter ones require as an extra input a value used only once (nonce). The generation of an good nonce/IV often requires the availability of a strong random number generator (RNG), although in some cases using a simple counter is sufficient.



(a) Unencrypted (b) ECB mode encryption (c) CBC mode encryption

Figure 1.4: Example of encrypting an image with AES in different modes.

1.1.2 Asymmetric cryptography

It wouldn't be until 1976 that a solution to symmetric crypto's chicken and egg problem was invented in the form of the Diffie-Hellman key exchange algorithm.[39] A year later, RSA[104] made its entrance as the first asymmetric encryption algorithm.

Definition 1.4. *An asymmetric cryptographic algorithm has the encryption key k_P publicly available, while the decryption key k_S remains a secret. For such an algorithm to be secure, it should be computationally infeasible to compute k_S from k_P.*[86, p. 25]

Asymmetric cryptography is often called public key cryptography, the encryption key called the public key, and the decryption key the private key. The general principle is illustrated in Figure 1.5. The sender, Alice, gets Bob's public key k_P from a publicly available key storage database. She then encrypts a plaintext P using k_P to generate the ciphertext C. The only way to decrypt C is by using Bob's secret key k_S. Thus, even Alice cannot decrypt the message she just encrypted, because she does not know k_s. Eve, an eavesdropper who has no knowledge of k_S either, the ciphertext C reveals nothing about the plaintext P.

Since the publication of RSA, many more public key algorithms have been invented. Examples of these are McEliece,[85] lattice-based,[2] and elliptic curves.[66,89] Despite its

[2] M. Ajtai and C. Dwork, "A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence" (1996).
[39] W. Diffie and M. E. Hellman, "New Directions in Cryptography" (1976).
[66] N. Koblitz, "Elliptic Curve Cryptosystems" (1987).
[85] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory" (1978).
[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).
[89] V. S. Miller, "Use of Elliptic Curves in Cryptography" (1985).
[104] R. L. Rivest *et al.*, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems" (1978).

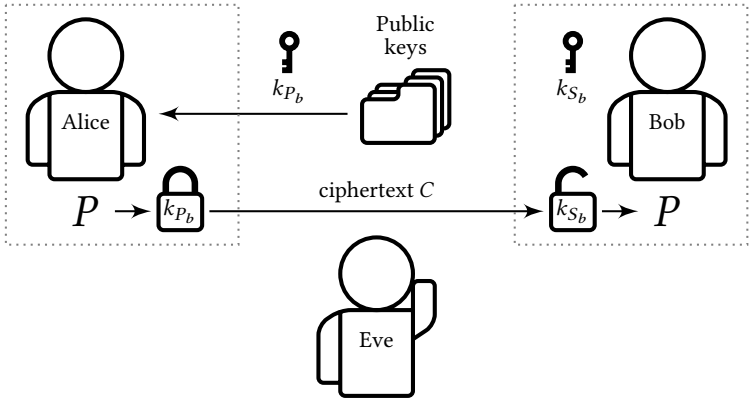


Figure 1.5: The working principle of asymmetric key cryptography.

age, RSA is still being used for the majority of public key protocols. However, recently elliptic curve crypto has been becoming more popular, since it requires much shorter keys and can, in many cases, be made to run much faster.

Compared to symmetric crypto the speed of asymmetric crypto is orders of magnitude slower, which is why it is generally only used to set up a shared secret between parties, after which the switch to symmetric crypto can be made.

Example Key establishment is one of the most important uses of asymmetric cryptography. Using a key exchange algorithm, it is possible for two parties to establish a secret shared key over an insecure channel, while only having access to the other party's public key.

One way to do this is with the Diffie-Hellman algorithm.[39] Figure 1.6 gives an overview of how it works. Both parties, Alice and Bob, generate a random number which will function as their private key. They then calculate their public key with the use of their private one. Because it is extremely difficult to invert this calculation, i.e. to find a private key given the matching public one, Alice and Bob can send each other their public key over a network that requires only authentication, no encryption. They then combine the received public key their own private key to generate a shared key, which can then be used for symmetric cryptography.

Both parties generally generate a different random private key every time they execute the Diffie-Hellman algorithm. Thus, a strong random number generator is required, or an attacker will be able to guess the private key.

[39] W. Diffie and M. E. Hellman, "New Directions in Cryptography" (1976).

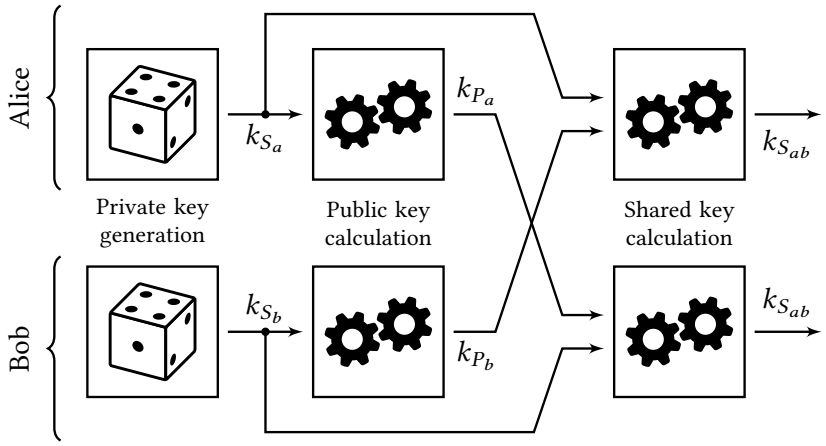


Figure 1.6: Overview of Diffie-Hellman key exchange.

1.1.3 Hash functions

Definition 1.5. *A hash function is a computationally efficient mapping of inputs of arbitrary length to a fixed length output.*[86, p. 33]

Apart from the length reduction, a second important difference between hash functions and (a)symmetric crypto algorithms is that the former do not need a (secret) key as one of its inputs. The working principle of a hash function is shown in Figure 1.7. A plaintext P is divided into fixed length sections, which are fed into the hash function. Once all sections have been processed, a so-called hash digest D is generated. The length of the digest D is independent of the length of P, and instead depends solely on the hash function.

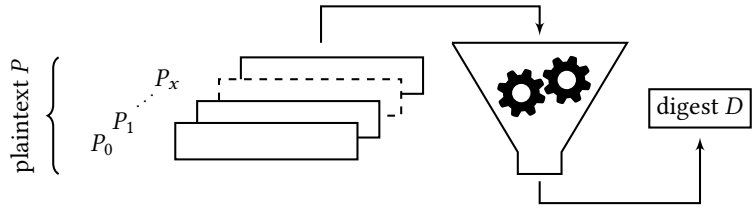


Figure 1.7: The working principle of a hash function.

[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

Different hash function exist for a variety of use-cases. For cryptography, one mostly uses cryptographic hash functions. Although a large number of such functions are available, nowadays SHA-2[92] is often used. Recently, a competition was held to choose its successor, SHA-3, which was won by the Keccak[16] algorithm.

Definition 1.6. *A cryptographic hash function is a hash function for which it is computationally infeasible to find* i*) two distinct inputs which hash to the same value (*collision resistance*); and* ii*) find an x which hashes to the same value as a given hash value, i.e. $h(x) = y$ (*pre-image resistance*).*[86, p. 323]

Another group of hash functions which often finds use in cryptography are the universal hash functions, which are defined as follows.

Definition 1.7. *Given a set $\mathcal{H}$ of hash functions h mapping inputs from set A to outputs of set B. This set is universal if, given inputs $x \neq y \in A$, and h chosen uniformly at random from $\mathcal{H}$, then the probability for a collision is* $\Pr(h(x) = h(y)) \leq \frac{1}{|B|}$.*[26]*

Example An important use of hash functions is password verification. One of the properties of cryptographic hash functions is that it is extremely difficult to find a plaintext which hashes to a given digest. Therefore servers should store digests of user's passwords instead of the passwords themselves. Checking a password is then done by first hashing the password a user input, and then comparing this with the stored digest. An attacker getting hold of these digests will not be able to invert digests to find the passwords.

However, if the attacker does not care which person's password he finds, he can use a prepared database of plaintext–digest pairs, with the plaintexts being often used passwords. Due to the birthday paradox, changes are high that the attacker will be able to match digests which also appear in his prepared database.

In order to stop such attacks, a system called salting is used, shown in Figure 1.8. With salting, one concatenates each password and some random data, called the salt, and then hashes the result. The digest is stored together with the salt value. To verify a login attempt, one first concatenates the stored salt and the to-be-verified password. An attacker who wants to use the previously mentioned attack against such a system needs to have a separate plaintext–digest pair database for every different salt value. However, doing so is computationally infeasible due to the required calculation time.

[16] G. Bertoni *et al.*, "The Keccak reference" (2011).
[26] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions" (1977).
[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).
[92] National Institute of Standards and Technology, "FIPS 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4" (2002).

Another important use case of hash functions is the creation of a message authentication code (MAC). MACs are created by combining a secret key k and a message m using a hash function. If a message and MAC pair (m, M) is transmitted, the receiver, who knows the secret key, can verify whether the received pair (m', M') has been tampered with. He does this by calculating $M_r = \mathsf{MAC}(k, m')$ and comparing it to the received MAC M'. Without knowledge of the secret key, an attacker cannot calculate the MAC, and thus cannot tamper with either the message or MAC without creating a mismatch between them. Thus, if M_r matches M', the receiver knows that m' matches m.

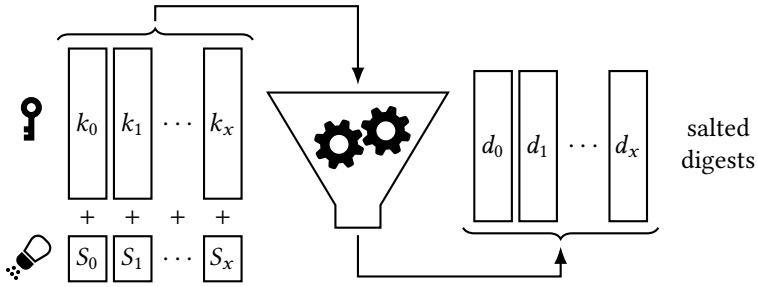


Figure 1.8: Principle of password salting

1.1.4 Keys and random numbers

As the previous examples clearly show, any implementation of a secure cryptographic protocol typically requires a secure RNG, whether to generate a key, a nonce, or both.

Definition 1.8. *A random number generator (RNG) is a device or algorithm that outputs a sequence of statistically independent and unbiased numbers.*[86, p. 170]

In an ideal world all RNGs would be based on unpredictable natural phenomena (e.g. thermal noise). Such systems, called true random number generators (TRNGs), unfortunately suffer from one or more disadvantages which makes them often unfeasibly for practical use, e.g. large size, slow number generation, or high cost.

The major advantage of TRNGs is that they generate data with very high entropy, a measure for information content (see Section 2.6.7). The higher the entropy, the more information the data contains, and thus the harder it is to estimate its contents.

Because of the drawbacks, most implementations are relegated to the use of pseudo-random number generators (PRNGs), deterministic algorithms which produce

[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

seemingly random data. Many PRNGs exist, a few of which are suitable for use in cryptographic systems, e.g. Yarrow,[64] Blum Blum Shub[18] and various hash- and block cipher-based designs.[8]

Definition 1.9. *A pseudo-random number generator (PRNG) is an algorithm that given a truly random value of length n, the seed, outputs a sequence of length $l \gg n$.*[86, p. 170]

Since the PRNG algorithm itself is deterministic, the output only looks random to someone without knowledge of the seed value. In order to test the quality of a PRNG several statistical tests are used, which are discussed in Section 2.6.6. However, passing these tests is not sufficient for a PRNG to be cryptographically secure. Such a PRNG should pass the next-bit test. Note that this requires the seed value to be of sufficient length, such that an adversary can not simply iterate over each possible seed value and check whether it generates the same values as the given output sequence.

Definition 1.10. *A PRNG passes the next-bit test if there exists no polynomial-time algorithm that can predict the $(l+1)^{th}$ bit of output sequence s with a probability significantly greater than $\frac{1}{2}$, given the first l bits of s.*[86, p. 171]

There are a few extra properties which a cryptographically secure PRNG should have. These are grouped under the term robustness,[7] which entails three properties: forward security, backward security, and resiliency.

Forward security means that an adversary who knows the internal state of the PRNG at time t can not predict past outputs generated at time $t - i$. In other words, the algorithm should function as a one-way function. Bellare and Yee[11] present methods that make any PRNG forward secure. One simple one consists of hashing the output of the PRNG before disclosing it. Of course, it is more efficient if such functionality is already built into the algorithm itself.

Backward security is similarly defined. Assuming a compromise of the PRNG's internal state at time t, then we say the PRNG is backward secure if knowledge of this state does not allow an adversary to predict future outputs at time $t + i$. Since a PRNG algorithm is deterministic, this property can only be met if new entropy is introduced into the state, a process which is called reseeding. Note that in most

[7] B. Barak and S. Halevi, "A Model and Architecture for Pseudo-Random Generation with Applications to /dev/random" (2005).
[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).
[11] M. Bellare and B. Yee, "Forward-Security in Private-Key Cryptography" (2001).
[18] L. Blum *et al.*, "A Simple Unpredictable Pseudo-Random Number Generator" (1986).
[64] J. Kelsey *et al.*, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" (1999).
[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

practical implementations, one can not constantly reseed a PRNG, since that would require a TRNG with throughput at least equal to the PRNG. Hence there would be no reason to use a PRNG to start with. Thus, a PRNG that is backward secure will in reality gradually recover its security after its state has been compromised![43]

A PRNG is resilient if an adversary can not predict future PRNG outputs, even if he can influence the entropy pool that is used to (re)seed the internal state of the PRNG![42] Note that this does not mean that the adversary knows the internal state.

Note the distinction between a cryptographically secure PRNG and the keystream generator of a stream cipher. These two building blocks, while similar, have a few distinct differences. The most obvious one is that a keystream generator block can not be backward secure, since this would require both parties to have access to the same source of true randomness. If that were the case, then this source of true randomness must be deterministic, which is clearly contradictory to its definition.

In modern operating systems, the difficult task of generating random numbers is often left to a central PRNG module. Thus, if the PRNG used in such a system is not robust, then an adversary will be able to predict both future and past states of the "centralized" PRNG all from within a single application. This will then allow him to predict and calculate keys and nonces used by cryptographic algorithms in other applications that also make use of the "centralized" PRNG.

Thus, although cryptographically secure PRNGs solve many drawbacks of TRNGs, they are certainly not without problems. The major problems is that their output depends on an internal secret state. Guaranteeing unpredictable output requires that the state is initialized, i.e. seeded, and later on reseeded, with high entropic data.

On desktop and server computers a few such seeding sources are available. For example, there is the time between network packet arrival, delay between keyboard key presses, and mouse cursor movement patterns. Furthermore, manufacturers have been picking up on the problem of seeding PRNGs. As a result Intel has added a hardware PRNG with a TRNG at its core[60] to their newest line of processors.

Finding entropy sources on embedded devices is much more problematic. Unlike general purpose computers, many embedded devices do not have user input. Most of them do not have a network connection either, and even if they do, not much data is sent in order to conserve power. It is thus no surprise that the security of embedded systems suffers worse due to this problem, further exemplified later on in this section.

[42] Y. Dodis *et al.*, "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust" (2013).

[43] Y. Dodis *et al.*, "How to Eat Your Entropy and Have it Too – Optimal Recovery Strategies for Compromised RNGs" (2014).

[60] Intel Corporation, "Intel Digital Random Number Generator (DRNG) Software Implementation Guide" (2012).

Physically unclonable functions (PUFs), a recent development in cryptography which functions as an electronic fingerprint, have the potential to solve some of these problems. PUFs allow for secure generation of device-dependent keys without external input, thus removing the need for companies to program high quality keys into their systems. For more background information on PUFs, see Chapter 2.

Unfortunately, the requirement for a strong (P)RNG is a fact often overlooked during the implementation of cryptographic systems. There are many examples of systems being broken because of the use of either a non-secure random number generator, or because of incorrectly seeding a cryptographically secure PRNG. The next section contains a multitude of examples of this.

1.2 Problem sketch

The general problem which we want to solve is the lack of secure keys, and of good TRNGs and PRNGs, i.e. we want to provide solutions which provide highly entropic keys and pseudo-random numbers. Since secure keys are generated from random data, these two building blocks are related. Thus, what we are looking for are sources of entropy that can be used to create these important building blocks. More specifically, we are looking for such sources in embedded devices, since those are much scarcer, as discussed earlier, and there is an urgent need for them, given the ubiquity of embedded devices in our society. To further drive home the point that many cryptographic implementations lack sources of adequate entropy, we will use the remainder of this section to discuss several high-profile cases of cryptographic systems failing due to exactly this problem.

Problems with PRNG implementations have plagued cryptography implementations for many years. A famous example from the 90's is the problematic implementation of the PRNG in the Netscape browsers. Goldberg and Wagner[47] discovered that the seed for the browser's PRNG was constructed from the time of day, the browser's process ID, and the ID of the process that launched the browser. All of these values are simple to guess, and thus an attacker could reconstruct the PRNG seed. Due to this weakness, any data protected with the Secure Sockets Layer (SSL) protocol could be easily decrypted.

A more recent high profile example of a buggy implementation is the seeding of the PRNG used in Debian's OpenSSL library between 2006 and 2008.[34,40] Due to a software optimization the PRNG algorithm was not seeded with high quality random

[34] Debian Security, "DSA-1571-1 OpenSSL – Predictable Random Number Generator" (2008).
[40] R. Dingledine, "Tor Security Advisory: Debian Flaw Causes Weak Identity Keys" (2008).
[47] I. Goldberg and D. Wagner, "Randomness and the Netscape Browser. How secure is the World Wide Web?" (1996).

data. Its seeding data was of very bad quality, due to which only 32 768 different output streams could be generated by the PRNG. This lead to all keys generated by those versions of the widely used library to be insecure, and millions of keys had to be regenerated after the discovery of this bug.

Plenty of other examples are shown in research papers by both Lenstra *et al.*[71] and Heninger *et al.*[53] The authors investigate the security of public keys freely available on the internet. Most of the investigated keys are for use with the RSA algorithm. Such keys are constructed using two randomly generated prime numbers. In an ideal world, all available keys would be constructed with different prime numbers. However, it turns out that many publicly available keys share prime factors, thereby severely weakening the security they provide. Heninger *et al.* managed to trace many of these weak keys back to certain embedded networking devices which seed their PRNG algorithm with very extremely low quality seeds.

More key related problems were discovered by Bernstein *et al.*,[14] who researched the security of keys used on Taiwanese ID cards. The TRNG in certain models of these cards is severely flawed, due to which the security of many generated keys is weakened. Because the Taiwanese ID card is used for all sorts of activities in everyday life, having breakable keys can have a large impact on people. Furthermore, the only proper solution to this problem is replacing the cards in question and updating various databases, a costly endeavor.

The MIFARE Classic RFID card by NXP is another high-profile example of a broken design, in this case due to weaknesses in the PRNG algorithm itself[32] This RFID card, of which billions were in circulation, was used for e.g. entry into Dutch military installations, and payment on the Dutch and London public transport systems. Naturally, due to its widespread use, updating all affected system to use another RFID card was an extremely expensive operation.

Finally, research by Gutterman *et al.*[51] and Dodis *et al.*[42] exposes flaws in the system used to seed the PRNG used in Linux distributions. Given the widespread use of Linux operating systems, such weaknesses are problematic.

All of these examples should make it clear that weak PRNG implementations (and associated seeding algorithms) are rife, and are not limited to low-profile cases. Instead, they have been resulted in flaws in multiple economically important infrastructures.

[14] D. J. Bernstein *et al.*, "Factoring RSA keys from certified smart cards: Coppersmith in the wild" (2013).
[32] G. de Koning Gans *et al.*, "A Practical Attack on the MIFARE Classic" (2008).
[42] Y. Dodis *et al.*, "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust" (2013).
[51] Z. Gutterman *et al.*, "Analysis of the Linux Random Number Generator" (2006).
[53] N. Heninger *et al.*, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices" (2012).
[71] A. K. Lenstra *et al.*, "Ron was wrong, Whit is right" (2012).

Clearly, one cannot treat them as an afterthought and expect to end up with a cryptographically secure system.

1.3 Thesis outline

This thesis contains several contributions to the field of embedded security, and more precisely to that of key generation and random number generation. We have developed an area-efficient building block for key generation, based on a novel PUF design and an extremely tiny, custom-built error correction microcontroller. Furthermore, we present the first large study of PUFs inherent to commercial-off-the-shelf (COTS) microcontrollers. Finally, combining the results of this study with the world's smallest implementation of Keccak to date, we implement an extremely compact software-only securely seeded PRNG.

1.3.1 Chapter summaries

A summary of this thesis' chapters, and, if applicable, a listing of the publications on which they are based, follows.

Chapter 1 – Introduction The first chapter gives an introduction to selected cryptographic building blocks. For each of them, we give examples which highlight the importance of both secure keys and random number generators. We discuss recent high-profile cases of broken cryptographic systems due to a lack of security in their key generation and/or random number generation modules. We summarize the contents and contributions of the remaining chapters. Finally, we give a brief overview of our papers whose content is not discussed in this thesis.

Chapter 2 – PUF and RNG Background This chapter is firstly a primer on PUFs: what are physically unclonable functions, how does one use them, and how can they be constructed? A major section deals with various metrics used to qualify PUFs. This chapter furthermore includes an overview of the mathematical notation used in the remainder of the thesis. We also discuss the way one can harness PUF behavior for random number generation. Most of the content in this chapter can be considered background knowledge, a few select sections are based on as yet unpublished material.

Chapter 3 – PUFKY: An Area-Efficient Key Generation Module The third chapter deals with the design and implementation of a custom PUF-based hardware

key generation device. We focus on the design and implementation of the error correction microcontroller required for the design. Our contribution is the design of an area efficient, PUF-based black box module for cryptographic key generation. This design is the first ever published completely functional one of its kind. Furthermore, our microcontroller is the world's smallest design for decoding BCH codes.

This chapter's contents are derived from *"Tiny, Application-Specific, Programmable Processor for BCH Decoding"*,[123] by Van Herrewege and Verbauwhede, presented at the International Symposium on System on Chip (2012), and *"PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator"*,[79] by Maes, Van Herrewege, and Verbauwhede, presented at the workshop on Cryptographic Hardware and Embedded Systems (CHES 2012).

Chapter 4 – Analysis of SRAM in COTS Microcontrollers In the fourth chapter, we discuss the evaluation SRAM as a PUF for four important microcontroller families. These families are respectively the Microchip PIC16, Atmel ATmega, Texas Instruments MSP430, and ARM Cortex-M. This is followed by a discussion on the impact of these results on the possibility of software-based PUF and PRNG implementations on these microcontrollers. Our contribution is the first thorough experimental verification whether embedded SRAM in COTS microcontrollers is fit for software-based PUF and RNG designs.

This chapter's content is based both on as yet unpublished work, and on *"Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers"*,[121] by Van Herrewege, van der Leest, Schaller, Katzenbeisser, and Verbauwhede, presented at the workshop on Trustworthy Embedded Devices (TrustED 2013).

Chapter 5 – Software-based Secure PRNG Design The fifth chapter focuses on the design of a securely seeded PRNG, using nothing but software on COTS microcontrollers. We discuss a possible attacker model for such a system and its impact on real-world implementations. Our contribution is the presented PRNG system, as well as a highly optimized implementation of this system on various ARM Cortex-M microcontrollers. Part of our design is the, at time of publication, world's smallest implementation of Keccak.

The content of this chapter is based on *"Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers"*,[121] by Van Herrewege, van der Leest, Schaller, Katzenbeisser, and Verbauwhede, presented at the workshop on Trustworthy Embedded Devices (TrustED 2013), on *"DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-the-Shelf Microcontrollers"*,[118] by Van Herrewege, Schaller, Katzenbeisser, and Verbauwhede, presented at the conference on Computer and Communications Security (CCS 2013), and on *"Software Only, Extremely Compact, Keccak-based Secure PRNG on*

ARM Cortex-M",[122] by Van Herrewege and Verbauwhede, presented at the Design Automation Conference (DAC 2014).

Chapter 6 – Conclusions In this final chapter we look back on the presented results and their implications. We also discuss which uncharted terrain might prove interesting for future research on key generation and RNGs for embedded devices.

1.3.2 Other publications

We conclude this chapter with a mention of publications whose content is not included. The publications are listed in reverse chronological order, unless multiple publications deal with the same topic, in which case all are mentioned together with the newest publication.

"Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers",[91] by Mouha, Mennink, Van Herrewege, Watanabe, Preneel, and Verbauwhede, presented at the conference on Selected Areas in Cryptography (SAC 2014), contains a proposal for a new, extremely fast and small message authentication code (MAC) for 32-bit processors. The MAC algorithm is based on a extremely efficient round function. We present implementation results on various ARM Cortex-M platforms, and show that our design is about 7 to 15 times faster than AES-CMAC, and approximately 10 times smaller.

The work in *"Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+"*,[31] by de Clercq, Uhsadel, Van Herrewege, and Verbauwhede, presented at the Design Automation Conference (DAC 2014), describes efficient methods of implementing fast elliptic curve cryptography (ECC) on the ultra-low power ARM Cortex-M0+ platform. By improving upon the commonly used Lopez-Dahab field multiplication method, the fastest implementation to date of an ECC design on any ARM Cortex-M platform is obtained. Furthermore, the implementation has the lowest energy requirements of any published microcontroller implementation with similar ECC security parameters.

"Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base",[96] by Noorman, Agten, Daniels, Huygens, Piessens, Preneel, Strackx, Van Herrewege, and Verbauwhede, the USENIX Security Symposium (USENIX 2013) investigates methods by which one can provide hardware-based trusted computing on embedded platforms. The theoretical work is supported by an implementation of the proposed design, based on a Texas Instrument MSP430 microcontroller. Our contribution consists of an efficient hardware implementation of a cryptographic hash function.

In *"Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs"*,[76] by Maes, Peeters, Van Herrewege, Wachsmann, Katzenbeisser, Sadeghi, and Verbauwhede, presented at the workshop on Financial Cryptography (FC 2012), we present a lightweight authentication protocol for PUFs which greatly reduces computation and storage requirements compared to traditional PUF authentication protocols.

Both *"LiBrA-CAN: A Lightweight Broadcast Authentication Protocol for Controller Area Networks"*,[49] by Groza, Murvay, Van Herrewege, and Verbauwhede, presented at the conference on Cryptology and Network Security (CANS 2012), and *"CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus"*,[119, 120] by Van Herrewege, Singelée, and Verbauwhede, presented at the workshop Embedded Security in Cars (escar 2011), present protocols for authenticating messages on the controller area network (CAN) bus. The protocol in the former paper allows for flexible master-slave and fail-safe signature generation. In the latter paper, the problems associated with MACs on the CAN bus are investigated, and a backward compatible protocol which allows real-time transmission of MACs is proposed.

In *"Compact Implementations of Pairings"*,[117] by Van Herrewege, Batina, Knežević, Verbauwhede, and Preneel, presented at the Benelux Workshop Information and System Security (WiSSeC 2009), and *"Compacte implementaties van paringen"*,[116] the Master's thesis on which the previous work is based, the design and implementation of a low-area hardware design for the calculation of binary pairings is presented.

1.4 Conclusion

Using nothing but basic cryptographic constructions, we have shown in this chapter that no secure cryptographic implementation can exist without secure key and random number generation. Unfortunately, as our many real-world examples show, it has been demonstrated time and again that many devices lack these basic building blocks, leading to broken cryptography.

Generating secure keys and random numbers can be particularly problematic on embedded devices. Yet, such devices are taking on an increasingly prominent role in our daily lives, e.g. smart phones, RFID cards, wireless gadgets, … It is therefore of critical importance that adequate cryptographic solutions are designed with these devices in mind.

In the remainder of this thesis, we will therefore focus on secure key and random number generation implementations fit for embedded designs. We have summarized the content of each of the remaining chapters, and presented an overview of our publications in the last section of the current chapter.

2 PUF and RNG Background

This time it had been magic.
And it didn't stop being magic just because you found out how it was done.
— Terry Pratchet, The Wee Free Men (2003)

As the discussions and examples in the previous chapter have shown, a good key generator and RNG method are essential for secure cryptographic systems. In the last decade a new cryptographic primitive, the physically unclonable function (PUF), has been introduced which shows great promise as, amongst other things, a secure key generator. Our work focuses on the use of such PUFs for both key and random number generation. Before we go into the details, an introduction on PUFs is in order.

This chapter presents a concise background on PUFs, followed by a short section on RNGs. We start out by defining PUFs, and give an overview of some of the cryptographic constructions they help make possible. The physical effects leading to a PUF's behavior are discussed and, to give the reader a better idea of PUF designs, some of the most studied constructions are presented. Next, a threat model for PUFs is presented. We then discuss which measurements are used in order to assess the quality of PUF constructions. Next, a method for error correction for PUFs is discussed. Finally, we discuss how RNGs tie in with PUFs. For an extended treatise on what exactly constitutes a PUF, its designs, and applications, we refer to Maes.[75]

2.1 Physically Unclonable Function

Describing a PUF is most easily done by regarding it as a fingerprint for CMOS integrated circuits. Just like a human's fingerprints, an ideal PUF is:

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

Physical Both fingerprints and a PUFs behavior are (due to) physical phenomena. They are not learned or programmed, but are rather inseparable from the object that they are part of.

Unclonable Control over the processes resulting in a specific fingerprint or PUF instance are beyond the grasp of today's technology, making them physically unclonable.

Function Taking the image of different fingers on a person's hand results in different fingerprints. In much the same way, a PUF can be queried with 'challenges' resulting in different 'responses', most often these are both bit strings.

A PUF's unclonability only applies to the physical aspect: just as it is possible to make an artificial copy of a person's fingerprints, it is in some cases possible to model PUFs. This effectively creates a mathematical clone of the PUF, an algorithm for the challenge-response mapping.

Perhaps the most important parallel between fingerprints and PUFs is their uniqueness. No matter how large a group of people, or how big an amount of PUF circuits, the chance to have two of them generate the same response to a challenge is negligible. Even though a group of PUF circuits is manufactured using the same process, chaotic behavior on a (sub-)microscopic scale results in each PUF instance responding differently to being queried with the same challenge.

Another similarity between fingerprints and PUFs is non-perfect reliability. Each time one gets his fingerprints scanned, the result is slightly different. This is due to the angle at which the finger is scanned, dust or cuts on the finger, … Specialized algorithms are used to help match the fingerprint to the one stored in a database. Whereas an ideal PUF always returns exactly the same response to a given challenge, so far no manufactured PUFs have managed this feat. Due to noisy processes, a PUF instance will with very high probability generate a slightly different response to the same challenge each time it is queried. Just like fingerprints, all PUF constructions published so far require some form of post-processing or error-correction in order to guarantee a given challenge always produces the same response.

Maes[75, Section 3.2] defines several PUF-related properties. A summary overview of these properties is listed in Table 2.1. Many of the listed properties imply the existence of others, e.g. none of the listed properties can exist in a PUF if it is not constructible.

In order to qualify as a PUF, a system needs at least both identifiability and physical unclonability. Besides these two minimally required properties, there are several nice-to-have ones. These are not required for a system to be classified as a PUF, but increase its usability as one. For example, a PUF which has true unclonability obviously has a greatly increased security margin over a PUF which only boasts physical unclonability.

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

Table 2.1: Summary of important PUF properties, based on work by Maes![75]

Property	Meaning
Constructibility	Practical production of PUF is possible.
Evaluability	Generation of response from challenge is easy.
Reproducibility	Similar responses to same challenge for a PUF instance.
(Inter-device) uniqueness	Different PUF instances generate largely different responses to same challenge.
Intra-device uniqueness	The same PUF instance generates largely different responses to different challenges.
Identifiability	Combination of reproducibility and uniqueness.
Physical Unclonability	Technical impossibility to create a physical copy of a PUF instance.
Unpredictability	Infeasibility to predict a PUF's response to a challenge given a limited set of other challenge-response pairs.
Mathematical Unclonability	Unpredictability assuming unlimited challenge-response pairs, i.e. infeasibility of creating a mathematical model.
True Unclonability	Combination of both physical and mathematical unclonability.
One-wayness	Predicting challenge from random given response is infeasible.
Tamper Evidence	Physical changes to PUF modify its challenge-response behavior.
Strong	Combination of true unclonability and a *large* challenge-response set.

Note that inter-device uniqueness is called uniqueness in PUF literature, whereas intra-device uniqueness is generally never mentioned. Therefore, if we don't further specify uniqueness, we are talking about the inter-device uniqueness property.

An ideal PUF is *strong*, meaning it has a response set that is exponentially large in the size of the challenge, and true unclonability![75] Unfortunately, it turns out that designing such a PUF is exceptionally difficult. All practical PUF designs published so far have been unable to meet these requirements and are thus *weak*. Most often, this is due to not having an exponentially large challenge-response set. On top of this,

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

many types of PUFs can be modeled mathematically. Designing a cheap, practical and *strong* PUF remains one of the biggest challenges in the field.

However, we would like to point out that that the established terminology *strong* and *weak* is rather poorly chosen. Using a *weak* PUF in a design does not imply an inherent lack of security in that design. One can almost always use a variety of protocols and algorithms in order to work around the problems posed by *weak* PUFs.

Note that at least two different definitions for *strong* exist. The first one, we have already explained. The second common definition of a *strong* PUF is a PUF that has a exponentially large challenge-response set, yet is not required to be unclonable.[35,38] In this thesis, we use the first definition.

2.2 Applications

Due to their rather unique properties as cryptographic primitives, PUFs allow for applications that would be impossible to construct otherwise. In the next paragraphs, we list some of these applications and point out how they are made possible or improved by the use of PUFs.

2.2.1 Identification & entity authentication

Identification and entity authentication are closely related cryptographic notions, and are often regarded as being one and the same. We make a slight distinction between the two, as discussed by Menezes *et al.*[86, p. 385]

By identification, we mean being able to produce some sort of identifying information, without any proof whether this information is valid, or even belongs to the entity presenting it. Entity authentication goes one step further. It requires proving the validity of the identifying information, and that the entity presenting the proof did so at the time the proof was requested, i.e. that the proof was not prepared beforehand.

A concept related to entity authentication is data origin authentication or message authentication, which deals with proving the validity of a message's content, i.e. that the message originates from the correct party and hasn't been tampered with.[86]

[35] J. Delvaux *et al.*, "Secure Lightweight Entity Authentication with Strong PUFs: Mission Impossible?" (2014).

[38] J. Delvaux and I. Verbauwhede, "Side Channel Modeling Attacks on 65nm Arbiter PUFs Exploiting CMOS Device Noise" (2013).

[86] A. J. Menezes *et al.*, "Handbook of Applied Cryptography" (1996).

In the remainder of this work, when we mention authentication, we mean entity authentication.

An example of identification would be presenting the serial number of a passport, whereas authentication would be presenting the passport itself and having the picture in it compared to the face of the person presenting the passport.

Traditional methods of electronic identification and authentication come down to presenting some form of serial number. Such identifiers, no matter how well protected they are, are inherently clonable. Thus, they only provide reasonable authentication as long as one is guaranteed that the identifying information has not been copied or stolen.

Due to its unclonability, a PUF-based authentication system does not have this problem. For each PUF instance, a set of challenge-response pairs are stored. Each PUF can then later on be authenticated by comparing its response to the stored response for the same challenge. If both match, one has a strong assurance that the correct PUF generated that response.

2.2.2 Anti-counterfeiting

From a technological standpoint anti-counterfeiting is the same as authentication. We mentioned it here separately, because it was one of the first suggested use-cases for PUFs, and because it is not an everyday application of cryptographic technology.

Traditional non-electronic anti-counterfeiting methods rely on incorporating difficult to manufacture artifacts in a device. Take for example banknotes, which contain watermarks, microtext, etc![29] Naturally, such methods are rather expensive.

In many devices, a cheaper, electronic method makes much more sense, and thus anti-counterfeiting was one of the first suggested applications for PUFs![98] A manufacturer stores a set of challenge-response pairs for each device containing the PUF. Verifying whether a device is genuine is accomplished by querying the PUF with one of the stored challenges and checking if the returned response matches the stored one.

In many ways, this is similar to an electronic serial number. Serial numbers, however, have one crucial disadvantage: they are trivial to copy. PUFs, on the other hand, cannot be cloned, which makes it impossible to create exact physical replicas of a device containing PUF anti-counterfeiting. Furthermore, with a serial number, there is no way to guarantee that each device has a different number, whereas PUFs provide such a feature by design.

[29] R. D. Wagner, “Introduction to Security Printing” (2005).
[98] R. S. Pappu *et al.*, “Physical One-Way Functions” (2002).

2.2.3 Key generation

There are two methods by which today's electronic devices are provided with cryptographic keys. The fist method consists of the manufacturer generating a key and programming this into the device, repeating this for every device. Assuming the manufacturer does not skip corners, one can be reasonably assured that the key will be of high quality. However, having to program a different key into every device is a costly, potentially time-consuming, process.

Therefore, manufacturers often opt for the second method, which has each device generate its own secret key. While this is a lot simpler, it does bring with it the problems associated with generating high quality random numbers. Especially on embedded devices this can lead to very weak practical security![53]

Instead a PUF can be used to generate the key, which guarantees that each device will have a secure key. In this case, the PUF is always queried with the same challenge, so that it always generates the same response, i.e. the key. This combines the advantages of the security afforded by the first method with the cost reduction and ease of use of the second method. An additional advantage is that the key is inherently guaranteed to be strong, of course assuming the PUF design itself has not been tampered with, whereas with the first method, one has to take the manufacturer's word for it.

Making the challenge programmable changes this system from a fixed to a programmable key generator, albeit one where there is no control over exactly which key is generated. An example of such a system, called a logically reprogrammable PUF, is presented by Katzenbeisser *et al.*[63]

Dodis *et al.*[41] introduced the concept of a fuzzy extractor to generate reliable keys from sources such as PUFs. This is a concatenation of a secure sketch, as described in Section 2.7, with a strong extractor[95] and is able to generate information-theoretically secure keys.

To obtain such a very high security level, one has to make a strong assumption about the min-entropy of the randomness source, which is often impossible. Furthermore, strong extractors generally induce a large loss in entropy (see Section 2.6.7), i.e. the output length is much smaller than the entropy of the input, which is undesirable or impractical since high-entropy randomness is scarce in most implementations. Finally, a random seed is required, which, as illustrated before, is not easy to obtain in a lot of cases.

[41] Y. Dodis *et al.*, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data" (2008).

[53] N. Heninger *et al.*, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices" (2012).

[63] S. Katzenbeisser *et al.*, "Recyclable PUFs: Logically Reconfigurable PUFs" (2011).

[95] N. Nisan and D. Zuckerman, "Randomness is Linear in Space" (1996).

In order to work around these problems, multiple works[8,44,64] instead recommend the use of cryptographic hash functions as entropy accumulators, a method which is heavily used in practice.

2.2.4 Key storage

Although the previously discussed application is an elegant solution to the key generation problem, in some cases it is required that keys can be set to a chosen value. In a traditional securely-designed system, the key is stored in specialized secure storage, designed to protect against attacks attempting to read out the storage. Such a system allows the key to be reprogrammed at will, but is expensive to manufacture.

By combining a PUF with traditional non-secure storage, a secure and cheap reprogrammable key storage unit can be created.[113] The PUF is used in the same way as for key generation (see Section 2.2.3): it is queried with a fixed challenge, and thus generates a fixed response. Instead of using this response as the system key, it is used to encrypt the actual key. This encrypted key can then be stored in unsecured storage and whenever it is needed, it is decrypted with the help of the PUF.

2.2.5 Hardware-software binding

One last application made possible by the advent of PUFs is hardware-software binding. A hardware-software binding system couples code running on a device to that device's hardware. The same code will not run on any other device, even if it is of the same type. Such a system allows manufacturers to lock down their devices, preventing any but their own approved software to run on it.[77]

In a basic hardware-software coupling system, the PUF is used as a key generator. This key is used to encrypt the software. Attempting to run this encrypted software on another device does not work, since the other device's PUF will generate a different key.

One can make the hardware-software binding even stricter by using a logically reconfigurable PUF that is updated together with every software update. This way,

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).

[44] D. Eastlake *et al.*, "Randomness Requirements for Security" (2005).

[64] J. Kelsey *et al.*, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" (1999).

[77] R. Maes *et al.*, "A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM based FPGAs" (2012).

[113] V. van der Leest *et al.*, "Soft Decision Error Correction for Compact Memory-Based PUFs Using a Single Enrollment" (2012).

every software version that has ever run on the device will be encrypted with a different key. It is therefore impossible to downgrade the software without knowledge of the current key, i.e. only the manufacturer can do so.

2.3 Design

The previous sections have dealt with properties and applications of PUFs. This section discusses the physical processes causing a PUF's behavior and give a description of some well-studied PUF designs.

This thesis text concerns itself only with digital, *intrinsic* PUFs. PUFs which are not completely digital, i.e. which require some form of analog electronics, are not discussed. The general class of *intrinsic* PUFs is defined by Maes[75, Section 2.3.3] as *i*) having the circuitry required to generate responses embedded within the PUF itself; and *ii*) displaying PUF behavior due to factors intrinsic to the manufacturing process.

2.3.1 CMOS process variation

We now discuss the (sub-)microscopic processes that lead to PUF behavior in certain digital constructions. For a more thorough discussion, we refer the reader to Böhm and Hofer![20, Chapter 6]

The majority of modern digital devices are constructed using complementary metal-oxide-semiconductor (CMOS) technology. Digital circuits built with this technology are designed as complementary pairs of P- and N-channel MOSFETs. MOSFETs require power only when (dis)charging capacitance on their gate and drain, and the wiring connected to it. This makes CMOS technology rather power efficient, since the majority of power consumption is due to MOSFET pairs changing state. During switching, power is consumed due to (dis)charging of capacitance and temporary short circuits. When the MOSFETs are not switching, only a small amount of power is consumed due to leakage currents. Furthermore, CMOS technology allows for very dense implementations. It requires less silicon area than rival technologies, and is thus cheaper. These are two important factors why it has become the main technology for modern designs.

In order to increase their yield rate, semiconductor manufacturers attempt to control process variation to the best of their abilities. Doing so also reduces the difference in physical characteristics of the MOSFETs in a circuit. This in turn improves the match

[20] C. Böhm and M. Hofer, "Physical Unclonable Functions in Theory and Practice" (2013).
[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

between simulated and actual circuit operation. In other words, it is in everyone's best interest to control process variations as well as possible. However, due to variations at a microscopic level, and even at an atomic one, all MOSFETs will still have minuscule differences between them.

The operating characteristics of a MOSFET are determined by its physical layout, of which the size of all its individual layers is a part. For example, a thinner gate layer will have a higher capacitance than a thick one, leading to increased switching times. Thus, one of the effects of process variations on a microscopic scale is discrepancies between operational characteristics of the MOSFETs in the circuit.

Even if manufacturers would be able to eliminate these microscopic variations, differences would still exist at a sub-microscopic (atomic) level. This is because so-called doped semiconducting materials are used. These materials consist of a semiconductor material into which atoms of a different material have been inserted.

Process variation at an atomic level influences local doping concentration, also called random dopant fluctuation (RDF).[5] This leads different sections of material to have slightly different characteristics. The smaller (i.e. newer) the technology, the bigger a problem RDF is, because the smaller size of newer technologies allows for less dopant atoms to begin with, and so a small (absolute) difference in the amount of dopant atoms have a much larger impact on characteristics than in older, larger technologies. For MOSFETs, such sub-microscopic variations can, for example, influence the output resistance $R_{DS_{\text{on}}}$. This in turn has an effect on the speed at which a capacitance can be (dis)charged by the MOSFET, effectively altering the speed at which it can switch on and off other MOSFETs.

Furthermore, these static process variation effects lead to each MOSFET having a slightly different response to external effects, e.g. localized temperature differences will have a higher impact on thinner semiconducting layers due to their reduced thermal mass compared to thicker layers.

Process variations can be classified into two groups. First of all there are global variations, which are variations between wafers or lots of ICs. Such variations affect all circuit elements on the wafer or IC lot by the same amount. For example, oscillators constructed from one silicon wafer might run 0.5% faster than those constructed from another wafer. Secondly, there are local variations, which lead to the various elements in a single circuit instance all having slightly different characteristics. Such variations are the major reason for a PUF's identifiability property.

Although every operating characteristic is influenced up to a certain point by process variations, one of the major influences on PUF behavior is the MOSFET gate threshold voltage V_{th}. This value is determined as the point at which a MOSFET starts to

[5] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub-0.1 µm MOSFET's: A 3-D "Atomistic" Simulation Study" (1998).

conduct current across its drain-source interface. MOSFETs with a lower V_{th} will start to conduct earlier, effectively making them faster than those with a higher V_{th}. On the other hand, the lower V_{th}, the higher the leakage current.

Thoroughly discussing all the factors that determine V_{th} would probably require a book all by itself![62] However, to give an idea of how many variables have an influence on this single parameter, the general formula for V_{th} for an N-channel MOSFET[62,90] is

$$V_{\text{th}} = V_{\text{th}_0} + \gamma \cdot \left(\sqrt{|-2\phi_F + V_{\text{sb}}|} - \sqrt{|2\phi_F|} \right),$$

where

$$V_{\text{th}_0} = \Phi_{\text{gc}} - 2\phi_F - \frac{Q_{B0} + Q_{\text{ox}}}{C_{\text{ox}}} \qquad \gamma = \frac{\sqrt{2q \cdot N_A \cdot \epsilon_{\text{Si}}}}{C_{\text{ox}}}$$

$$Q_{B0} = -\sqrt{2q \cdot N_A \cdot \epsilon_{\text{Si}} \cdot |-2\phi_F|} \qquad Q_{\text{ox}} = q \cdot N_{\text{ox}}$$

$$C_{\text{ox}} = \frac{\epsilon_{\text{ox}}}{t_{\text{ox}}} \qquad \phi_F = \frac{kT}{q} \ln\left(\frac{n_i}{N_A}\right)$$

$$n_i = 2.70 \cdot 10^{13} \cdot T^{2.54} \cdot \exp\left(\frac{-6726}{T}\right).$$

Suffice to say that perfectly controlling each and every one of these parameters is an impossible task, and thus, no matter the technological advances, V_{th} will most probably always remain one of the first parameters to look at when considering PUF behavior in CMOS technologies.

Differences in V_{th} lead to two types of behavior in PUFs. First of all, when large V_{th} differences exist within a circuit's implementation, the circuit will behave in a deterministic way, e.g. out of two oscillators, one might always run slightly faster. Of course, for every implementation, which of the two oscillators runs fastest might differ. This is the type of behavior that gives rise to a PUF's identifiability property.

A second type of behavior occurs when these is very little to no difference in V_{th}, in which case a circuit will behave randomly, e.g. which of the oscillators runs fastest will depend on seemingly random external factors. Because of these effects error correction is required for PUFs.

Looking at the formulas determining V_{th} one can clearly distinguish between three different groups of parameters. First of all there are the physical constants, e.g. the

[62] S.-M. Kang and Y. Leblebici, "CMOS Digital Integrated Circuits Analysis & Design" (2003).

[90] K. Misiakos and D. Tsamakis, "Accurate measurements of the silicon intrinsic carrier density from 78 to 340 K" (1993).

unit charge q, the Boltzmann constant k, and the dielectric constants of silicon ϵ_{Si} and of silicon dioxide ϵ_{ox}. Since these parameters are unchanging, they have no variable influence on any of the MOSFET's characteristics.

A second group of parameters are those governed by the production properties of the MOSFET, i.e. those that are influenced by process variation. For example, the gate oxide thickness t_{ox}, the oxide-interface charge density N_{ox}, the doping concentration N_A, and in part the work function difference Φ_{gc} (the exact formula of which depends on the MOSFET construction). All of these parameters are influenced by process variation at various stages in the production process, i.e. by the thickness of a semiconductor layer, by the amount of dopant atoms, ...It is this kind of variation which is responsible for the PUF-like behavior in circuits, i.e. the identifiability property. Thus, it is clear that PUF-like behavior originates from physical phenomena and production effects.

Finally, there are parameters influenced by the environment, such as the Fermi potential ϕ_F, the intrinsic carrier concentration of silicon n_i, and in part the work function difference Φ_{gc}. All of these are influenced by the temperature T. Due to the complexity of accurately modelling the tiny localized temperature variations within an operating circuit, current technology cannot predict what will happen exactly within that circuit. As such, parameters like these have a randomizing effect on a circuit's operation. Hence, this type of variation is one of the reasons error correction is required on a PUF's response, the other being circuit noise.

2.3.2 Example designs

Knowing how (un)wanted variations appear in CMOS circuits, we will now present a brief overview of some PUF designs. The presented designs are rather simple, and don't always behave as an ideal PUF would. For an overview of improved, more complex designs we once again refer to Maes.[75, Chapter 2]

2.3.2.1 Ring oscillator PUF

The basis of a ring oscillator PUF is, as the name implies, a group of ring oscillators. Due to process variation, these identically-designed oscillators will all run at slightly different frequencies. The frequencies are measured, and based on these measurements a response is generated. Exactly how the measurements are used depends on the PUF design.

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

One simple example, shown in Figure 2.1, uses a large amount of oscillators pairs. The challenge consists of which pairs to measure. For each pair, the two frequencies are measured and then compared to generate a single response bit. These response bits are then all concatenated to generate the complete response bitstring.

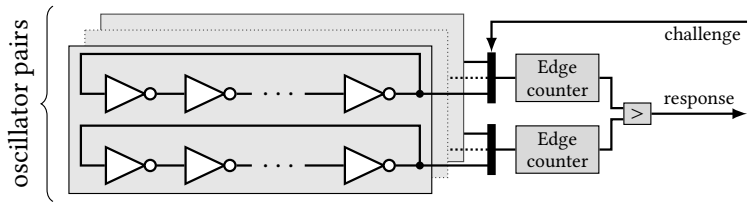


Figure 2.1: Oscillator PUF

The biggest drawback of this simple response generation scheme is that many of oscillator pairs are required in order to have an acceptably large challenge-response space. E.g. assume each oscillator pair has an ID, pairs are always queried in order of increasing ID, the response should be 32 bit, and at least 2^{32} possible challenge-response pairs are required. Then the PUF needs a total of at least 86 oscillators, since

$$\binom{\frac{x}{2}}{32} \geq 2^{32} \Leftrightarrow x \geq 86\,.$$

However, in this scheme all response bits are independent of each other, thus once an attacker figures out the response of a specific pair, he knows that bit of the response string every time the pair appears in the challenge. When employing such simple PUF designs in a system, it is of crucial importance that the raw response is never made public, and instead e.g. a hash of it is transmitted.

Another common problem happens when oscillator pairs are not fixed. For example, given the frequency of three oscillators A, B and C, then if

$$\left.\begin{array}{l} f_A < f_C \\ f_B < f_A \end{array}\right\} \Rightarrow f_B < f_C\,.$$

Thus, an adversary can infer the response to the challenge requesting the pairing of A and C, if he knows the responses to the challenges pairing A and B, and B and C.

Ring oscillator PUFs making use of a simple pairwise comparison have very good reproducibility, meaning they require little in the way of error correction. However, designs which rely on this property often require many oscillators, due to the drawbacks mentioned earlier. Finding a design that makes an optimal trade-off between the number of oscillators, which influences size and thus cost of the final circuit, and the provided security remains an active area of research.

2.3.2.2 Arbiter PUF

Arbiter PUFs are a class of PUFs relying on variable delays within a circuit to generate response bits. Figure 2.2 shows the simplest version of such a PUF.

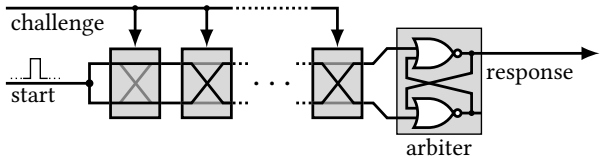


Figure 2.2: Arbiter PUF

The arbiter PUF consists of a collection of configurable crossover switches with two in- and outputs. Signals can be routed either straight through or to the opposite output. Internally, these switches are constructed out of two multiplexers (MUXes). Due to process variations, the V_{th} of the MOSFETs making up these MUXes, as well as wiring capacitance and length, will differ, effectively making their response time all different, which leads to different propagation speed for both signals. In order to generate a response bit, a signal is sent down both paths simultaneously, and at the output a so-called arbiter circuit (often a simple SR flip-flop) detects which of the two signals arrives first.

Each challenge bit controls whether a switch crosses over or not. By storing the challenge in a linear feedback shift register (LFSR) and clocking this after every generated response bit, multiple bits can be generated.

As each switch influences the remainder of the signal's path, the number of challenges is exponential in the number of switches. However, even though an exponential number of challenges can be queried, the total time required for each path to reach the arbiter is linear in the number of switches used. It is therefore fairly easy to create a mathematical model an arbiter PUF, even when only a few challenge-response pairs are known.[58] Such PUFs should thus never be used in conjunction with a protocol that leaks information about responses.

Because the size of the challenge-response set grows exponentially with the number of crossover switches, arbiter-based designs are an important candidate for a strong PUF. Whether it is possible to design a strong arbiter PUF in a way that cannot easily be modeled remains an open question.[35]

[35] J. Delvaux *et al.*, "Secure Lightweight Entity Authentication with Strong PUFs: Mission Impossible?" (2014).

[58] G. Hospodar *et al.*, "Machine Learning Attacks on 65nm Arbiter PUFs: Accurate Modeling poses strict Bounds on Usability" (2012).

2.3.2.3 SRAM PUFs

SRAM PUFs are part of the family of memory-based PUF designs. Some other members of this family are the latch, flip-flop, and buskeeper PUFs. Whereas the previous two PUF types require a (time-based) measurement to generate a response, memory-based PUFs do not: they can simply be read out. These PUFs rely for their response bits on the fact that certain types of memory start up in a state which cannot be controlled during manufacturing.

Take for example the basic SRAM PUF shown in Figure 2.3. In its simplest form, an SRAM PUF consists of nothing more than an SRAM chip. SRAM cells consist of two cross-coupled inverters, whose outputs are both low on power-up. Due to this, they both drive the input of the other inverter low, and thus will start to raise their output. Again due to V_{th} differences it is likely that one of the inverters is able to do so before the other, thereby stabilizing the state of the cell and determining its value.

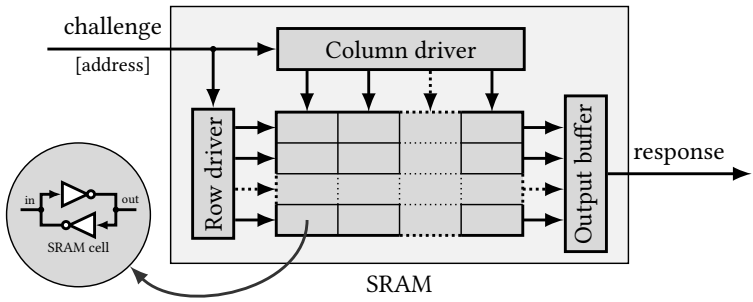


Figure 2.3: Basic SRAM PUF

For the majority of SRAM cells the difference in V_{th} is large enough to lead to the same, deterministic, start-up state every time![74] In cases where the V_{th} differences are very small, power-up effectively starts a race condition between the inverters, with a random start-up state as the result. Due to external factors, start-up state is not a black and white affair, i.e. a cell could power up in the one state 80% of the time and in the zero state 20%.

Challenging a memory-based PUF is done by reading out certain sections of memory; its challenge-response space is thus linear. The big drawback of this is that model generation of a section is trivial should a response ever become known to an attacker. The largest advantage of memory-based PUFs, especially SRAM ones, is that their behavior closely matches that of an ideal PUF.

[74] R. Maes, "An Accurate Probabilistic Reliability Model for Silicon PUFs" (2013).

2.4 Threat model

A PUF instance is considered broken if an attacker manages to extract the entropy contained within the PUF. This means that an attacker is able to predict what the response to any given challenge will be for that particular PUF instance. In some designs, such as a PUF-based fixed key generator, being able to determine just one response, i.e. the one to the fixed challenge, is enough to warrant it broken. There are three main categories of threats which could allow one to accomplish such a feat. We will discuss each of them succinctly in this section.

2.4.1 Remote threats

The first category of threats are those which can be launched remotely, i.e. these are software-based attacks. In general, such attacks don't require specialized hardware, and are thus cheap. For example, one can attack the protocol that is used when interfacing with the PUF. Thus, care should be taken that the protocol does not inadvertently discloses information about secret data,[35] such as the PUF response. Although this is a difficult problem, it is not specific to PUFs. Thus, established cryptographic protocol analysis techniques are the recommended prevention method against protocol attacks.

On the other hand, attacks on helper data, which we discuss in Section 2.7, are specific to PUFs. Helper data, which is required to correct errors in a PUF's response, is assumed to be public and stored in non-secure storage. Helper data manipulation attacks[36,37] modify this data to learn some information about the PUF response. By iterating these manipulations one can eventually calculate the complete response.

One possible prevention technique is storing not only the helper data, but also a hash of the helper data and the PUF response. After response correction, the circuit then verifies whether the received helper data and generated PUF response hash to the same value. The drawback of this technique is that it requires extra hardware, i.e. at least a hash function block, a comparator, and extra storage.

Another way of preventing helper data manipulation attacks, is by not having helper data in the first place. However, this requires a PUF design that never generates any errors in its responses, a feat that, so far, no one has been able to manage.

[35] J. Delvaux *et al.*, "Secure Lightweight Entity Authentication with Strong PUFs: Mission Impossible?" (2014).

[36] J. Delvaux and I. Verbauwhede, "Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation" (2014).

[37] ——, "Key-recovery Attacks on Various RO PUF Constructions via Helper Data Manipulation" (2014).

Modeling attacks are yet another type of attacks which can be executed remotely. If an attacker gets his hands on a set of challenge-response pairs of a PUF instance, he might be able to create a model of that instance. The viability of such attacks strongly depends on the design of the PUF and the number of known challenge-response pairs. For example, arbiter PUFs are well known for being very susceptible to this type of attack![58,106] Since these attacks require a certain number of challenge-response pairs for adequate modeling precision, restricting availability of such pairs is a good way to slow them down. Ultimately, however, the only surefire prevention method is using a PUF design which is not susceptible to modeling.

If an attacker can query the PUF, which is sometimes assumed for strong PUFs, still more attacks might be possible. For example, information about the error rate in an arbiter PUF can be used to generate a model of that PUF![38] Delvaux and Verbauwhede warn that such attacks are likely to be viable for all strong PUF designs.

2.4.2 Physical access threats

A second category of threats are those that require physical access to the PUF circuitry. In general, these are more expensive than remote threats, because specialized hardware is required.

Side-channel attacks are the best known attacks of this type. Measuring data which is seemingly unrelated to the PUF response, e.g. power consumption of the PUF, allows one to determine exactly what the response is![87] Side-channel analysis[46] is a large subfield of cryptanalysis, and should be expected that many sophisticated attacks, which have so far only been attempted on classical cryptographic building blocks, are viable on PUFs as well. Prevention techniques against such attacks often involve modifications to the physical layout of a circuit, which can be very expensive.

Invasive attacks also fall under this category. By using microscopes, probes, and lasers, it is possible to observe and modify individual elements that make up the PUF circuit. The techniques required to do so have long been established for failure analysis, an engineering domain that concerns itself with bug fixing of hardware designs. Using these techniques, one can read out PUF responses[93] and even create physical

[38] J. Delvaux and I. Verbauwhede, "Side Channel Modeling Attacks on 65nm Arbiter PUFs Exploiting CMOS Device Noise" (2013).
[46] B. Gierlichs, "Statistical and Information-Theoretic Methods for Power Analysis on Embedded Cryptography" (2011).
[58] G. Hospodar *et al.*, "Machine Learning Attacks on 65nm Arbiter PUFs: Accurate Modeling poses strict Bounds on Usability" (2012).
[87] D. Merli *et al.*, "Side-Channel Analysis of PUFs and Fuzzy Extractors" (2011).
[93] D. Nedospasov *et al.*, "Invasive PUF Analysis" (2013).
[106] U. Rührmair *et al.*, "Modeling Attacks on Physical Unclonable Functions" (2010).

clones of a PUF.[52] Prevention is aimed at making these attacks as time consuming as possible, e.g. by randomizing circuit layout or adding intrusion detection layers. However, given a sufficient amount of time and money, it is unlikely that any of these prevention methods will stop a determined attacker.

2.4.3 "Inside job" threats

Finally, one should consider "inside job"-type of threats. In this case the manufacturer tampers with the design of the PUF, such that he, or a third party, can recreate or read out the PUF's responses.

There a two ways a manufacturer can go about this. In the first case, a so-called trojan circuit is inserted in the design. Such a circuit allows an adversary who is aware of it to read out secret data from a chip. A lot of research is available on such hardware trojans and on methods to detect them.[1] Note that the adversary, the manufacturer, is essentially almighty in this case, so there is no realistic way of preventing the insertion of the trojan circuit.

In the second case, the manufacturer replaces the PUF design with something that mimics a PUF. For example, a programmed seed can be expanded with the help of a block cipher to generate responses. Since the programmed seed is known to the manufacturer, he can simply calculate the PUF responses to any challenge. In order to make the PUF behavior more realistic, some source of noise should be added, so that at least a few errors are present in the response. In order to detect such tampering, one could verify the PUF's behavior. Given a well-thought-out and executed design, it is unlike the attack would be detected though. Even worse, for some designs the PUF response is not accessible and thus no such verification is possible, e.g. a PUF-based key generator. A guaranteed way of detecting this attack is to use invasive techniques and compare the actual circuit layout to the design files. However, this is a time-consuming and costly method.

These attacks are particularly devastating, since for the victim they are difficult and expensive to detect, whereas for the manufacturer they are cheap and relatively easy to put in place.

[1] D. Agrawal *et al.*, "Trojan Detection using IC Fingerprinting" (2007).
[52] C. Helfmeier *et al.*, "Cloning Physically Unclonable Functions" (2013).

2.5 Mathematical notation

In this short section, we briefly discuss the mathematical notation used in the remainder of this thesis. A summary can be found in the List of Symbols (p. xxi).

Multiple physical implementations, called instances or samples, of the same PUF design all generate different responses, hence mathematical notations which clearly denote all variables involved in measurements can be rather elaborate. For brevity, we use the compact mathematical notation introduced by Maes.[75, Section 2.2]

The design of a PUF is denoted as $\mathcal{P}$, and the ith physical instance of $\mathcal{P}$ by puf_i. Challenges to a PUF are denoted by x, and the set of all challenges for $\mathcal{P}$ by $\mathcal{X}_\mathcal{P}$. When a PUF is queried with a challenge, we write this as $\mathsf{puf}_i(x)$, the resulting response is $Y_i(x)$, i.e.

$$Y_i(x) \leftarrow \mathsf{puf}_i(x)\,.$$

If multiple responses are measured for the same challenge, we denote the response index in superscript: $Y_i^{(j)}(x)$. In case it is not important to designate exactly which PUF instance is queried, we write $Y(x) \leftarrow \mathsf{PUF}(x)$. The set of all responses which $\mathcal{P}$ can produce is $\mathcal{Y}_\mathcal{P}$.

Groups of measurements are written in bold, e.g. $\mathbf{Y}_i(x)$. If for particular measurements the environmental conditions α, e.g. temperature, are of importance, then these are designated as $Y_{i;\alpha}(x)$.

Finally, many of the metrics presented are often written fractionally, i.e. as a percentage of the response length. Doing so makes comparisons between different PUF designs significantly easier.

Additional notation is required for various mathematical concepts later on. A binary Galois field is written as $\mathbb{F}_{2^x}$. The symbol $\oplus$ is an addition in such a field, i.e. a XOR operation, and $\otimes$ a multiplication. An element of $\mathbb{F}_{2^x}$ is written in capitals, e.g. A.

The notation $C(n,k,t)$ stands for a linear error correcting code with code length n, dimension k and number of correctable errors t. Such a code can be defined by a generator matrix G. If the code is polynomial, it can also be defined by its generator polynomial $\mathcal{G}$.

By $\mathrm{B}_{n,p}(t)$ we denote the binomial cumulative distribution function with as parameters the number of trials n and the success probability of each trial p evaluated in t, and $\mathrm{B}^{-1}_{n,p}(q)$ is its inverse.

When discussing arrays of data, $A[i]$ is used to indicate the ith element of array A. Arrays start at index position 0, unless specified otherwise.

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

2.6 Quality metrics

Qualifying the degree to which a physical PUF behaves as a theoretically ideal PUF is a complex task. Over the years a multitude of metrics have been suggested to assist in this process. In this section we discuss those which have become the de facto way of qualifying PUF designs and, in passing, mention a few less used ones.

2.6.1 Operating conditions

As noted before, semiconductor behavior can change with varying operating conditions. One obvious example is the dependence of the MOSFET threshold voltage V_{th} on the operating temperature T, as discussed in Section 2.3.1.

Due to its impact on many semiconductor-defining parameters, it is important to measure challenge-response sets under various temperature conditions. Next to room temperature (25 °C), both the lower (−25 °C) and upper (90 °C) commercial semiconductor operating limits are of particular importance. Measurements at these conditions help to establish whether a PUF will react the same under all expected operating conditions.

The second factor that has a large impact on semiconductor behavior is the voltage that circuits are supplied with. Depending on semiconductor technology and implementation, the required voltage can range anywhere from 0.2 V, for recent technology, to 5 V, for very old designs. As such the regular supply voltage as well as its lower and upper limits are different for most PUFs.

Other commonly variable environmental condition, such as pressure, are not important in this context, since encapsulated digital circuits are not expected to be influenced by them under realistic conditions.

Aging is another type of operating condition which can drastically influence circuits. In order to artificially induce aging in a circuit, one can operate it at increased temperature and supply voltage.

If not explicitly mentioned, metrics are calculated on measurements taken at reference conditions $\alpha_{\text{ref}} = (T_{\text{env}} = 25\ ^\circ\text{C}, V_{\text{CC}} = V_{\text{req}})$, where V_{req} is the recommended, required supply voltage for the device under test.

2.6.2 Hamming weight

Definition 2.1. *The Hamming weight (HW) of a string is the number of non-zero elements in that string, i.e. for a binary string the Hamming weight is equal to the*

number of '1' bits. The fractional HW of a string is equal to its Hamming weight divided by its length, and is written as a percentage.

For an ideal PUF response bits behave as independent random processes, i.e. an n-bit response can be modeled by a binomial distribution with parameters n and $p = 0.5$. Therefore the average expected Hamming weight in the response is $0.5n$, with a standard deviation of $0.5\sqrt{n}$. Values either below or above this value can indicate a correlation between the response bits, and thus might indicate non-ideal PUF behavior.

The Hamming weight is related to the uniqueness property, since if all of a PUF's responses are heavily biased, i.e. have a very high or low HW, then they must be less unique than those of a PUF generating responses close to the ideal HW. For example, consider a PUF design that generates 4-bit responses all with 75% HW. This means there are only $\binom{4}{3} = 4$ possible responses, which differ at most at two positions. On the other hand, if all responses have a 50% HW, then there are $\binom{4}{2} = 6$ possible outputs, for some of which all four bits are different. Thus, for the PUF design with 75% HW responses, the responses will be less different from one another than those of the PUF with 50% HW responses are.

Another property about which Hamming weight gives an indication is unpredictability. It is easy to see that in the previous example, an adversary has an easier time guessing a response for the PUF with 75% HW responses than for the one that generates 50% HW responses, since there are only four possible responses in the first case, versus six in the second one.

Assessing Hamming weight is often a first step in qualifying a PUF's responses. However, the fact that a PUF's responses have close to a 50% Hamming weight does not in any way guarantee anything about the uniqueness or unpredictability of its outputs. E.g. a truly random process outputting 'unique' 32 bits generates bitstrings with a 50% Hamming weight only 14% of the time:

$$\binom{32}{16} \cdot \frac{1}{2^{32}} \approx 0.140\,.$$

On the other hand, a process that repeatedly outputs '01' generates a bitstring with 50% Hamming weight, yet clearly has neither the uniqueness nor the unpredictability property.

Even though Hamming weight is a very simple metric, and will thus miss properties such as bit dependence, it helps to reveal potential problems in a PUF's response set, such as bias.

2.6.3 Inter-device distance

Definition 2.2. *The inter-device distance is the distance between two responses from different PUF instances to the same challenge:*

$$D_{\mathcal{P}}^{\text{inter}}(x) \triangleq \mathbf{dist}[Y(x), Y'(x)] \,.$$

The fractional inter-device distance is defined as the inter-device distance divided by the length of the PUF response, and is written as a percentage.

The distance function can be any metric over $\mathcal{Y}_{\mathcal{P}}$, although in practice almost all scientific works use Hamming distance with a XOR function, i.e. the Hamming weight of the bitstring $\big(Y(x) \oplus Y'(x)\big)$.

Inter-device distance is used to qualify the inter-device uniqueness of PUF instances. If the challenge-response behavior of a PUF is indeed a random process, then the fractional inter-device distance is expected to be 50%. Deviations from this expected value demonstrate correlation between instances.

Given an experiment collecting challenge-response measurements $\boldsymbol{Y}_{\text{Exp}(\mathcal{P})}$ made on a number of PUF instances N_{puf}, and a set of challenges N_{chal}, with each challenge queried N_{meas} times, an array $\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}}$ can be calculated:

$$\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}} = \left[\mathbf{dist}\left[Y_{i_1}^{(j)}(x_k); Y_{i_2}^{(j)}(x_k)\right]\right]_{\forall 1 \le i_1 \ne i_2 \le N_{\text{puf}}; \forall 1 \le k \le N_{\text{chal}}; \forall 1 \le j \le N_{\text{meas}}} \,.$$

We get an estimate for the expected value $\mu_{\mathcal{P}}^{\text{inter}}$ with the sample mean:

$$\overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}}} = \frac{2}{N_{\text{puf}} \cdot (N_{\text{puf}} - 1) \cdot N_{\text{chal}} \cdot N_{\text{meas}}} \cdot \sum \boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}} \,.$$

The sample standard deviation is used to estimate the standard deviation $\sigma_{\mathcal{P}}^{\text{inter}}$:

$$s_{\text{Exp}(\mathcal{P})}^{\text{inter}} = \sqrt{\frac{2}{N_{\text{puf}} \cdot (N_{\text{puf}} - 1) \cdot N_{\text{chal}} \cdot N_{\text{meas}} - 2} \cdot \sum \left(\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}} - \overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{inter}}}\right)^2} \,.$$

These variables are used to asses the quality of PUFs in an experiment, e.g. if the sample mean is around 50%, yet the standard deviation is large (15%), then that would indicate that certain groups of tested PUF instances have correlated responses. In their fractional form, the variables allow convenient quality comparissons between various PUF designs.

2.6.4 Intra-device distance

Definition 2.3. *The intra-device distance is the distance between two responses to the same challenge on the same PUF instance:*

$$D_{\mathcal{P}}^{\text{intra}}(x) \triangleq \mathbf{dist}[Y_i^{(j_1)}(x), Y_i^{(j_2)}(x)] .$$

The fractional intra-device distance is equal to the intra-device distance divided by the length of the PUF response, and is written as a percentage.

Whereas inter-device distance qualifies the inter-device uniqueness between different PUF instances, the intra-device distance is a measure of reproducability. For an ideal PUF, the intra-device distance is zero, i.e. it always returns the same response for a given challenge. For actual PUF implementations, there is bound to be some intra-device distance, and in that case this metric is used to determine the required error correction method and parameters.

PUF designs with high intra-device distance are not usable, since, first of all, they require excessive error correction, which is costly in terms of storage, computation time, and energy consumption. More importantly, high intra-device distance can lead to error correction methods correcting a response of instance puf_i to one of instance puf_j, leading to misidentification.

Estimations for the expected value and standard deviation are calculated using the same method as for inter-device distance, except the array of distances $\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}}$ is defined as

$$\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}} = \left[\mathbf{dist}\left[Y_i^{(j_1)}(x_k); Y_i^{(j_2)}(x_k)\right]\right]_{\forall 1 \le i \le N_{\text{puf}}; \forall 1 \le k \le N_{\text{chal}}; \forall 1 \le j_1 \ne j_2 \le N_{\text{meas}}} .$$

Thus, the estimate for the expected value, i.e. the sample mean, becomes

$$\overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}}} = \frac{2}{N_{\text{puf}} \cdot N_{\text{chal}} \cdot N_{\text{meas}} \cdot (N_{\text{meas}} - 1)} \cdot \sum \boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}} .$$

And the standard deviation estimate, i.e. the sample standard deviation, is now

$$s_{\text{Exp}(\mathcal{P})}^{\text{intra}} = \sqrt{\frac{2}{N_{\text{puf}} \cdot N_{\text{chal}} \cdot N_{\text{meas}} \cdot (N_{\text{meas}} - 1) - 2} \cdot \sum \left(\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}} - \overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{intra}}}\right)^2} .$$

Metrics for measurements taken at non-reference conditions are calculated with respect to to the reference condition α_{ref}. E.g. the intra-device distance for a PUF design measured at $\alpha = (T_{\text{env}} = 90\ ^\circ\text{C})$ is calculated as

$$D_{\mathcal{P};\alpha}^{\text{intra}}(x) \triangleq \mathbf{dist}[Y_{i;\alpha_{\text{ref}}}(x), Y_{i;\alpha}(x)] .$$

2.6.5 Self-similarity

Definition 2.4. *Self-similarity is the distance between responses to different challenges on the same PUF instance:*

$$D_{\mathcal{P}}^{\text{self}}(x_1, x_2) \triangleq \mathbf{dist}[Y_i(x_1), Y_i(x_2)] .$$

Self-similarity measures the intra-device uniqueness, i.e. uniqueness within responses to different challenges on the same PUF instance. Self-similarity is thus a measure of intra-device uniqueness, whereas inter-device distance is a measure of inter-device uniqueness. Much like inter-device distance, in an ideal case a PUF's self-similarity will be an average of 50%, i.e. responses to different challenges will appear to be generated by a different PUF instance.

Depending on the use-case self-similarity may or may not be important. E.g. for identification purposes it does not matter whether puf_i generates similar responses to different challenges. However, for reconfigurable key generators it does, since in such a case PUFs with high self-similarity might generate similar keys for different challenges, thereby weakening the security of the generated keys. It should be noted that one can often reduce the impact of low self-similarity by use of appropriate protocol and algorithm design, e.g. only transmitting hashed responses and keeping the responses themselves secret.

The formula for the distance array $\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}}$ is in this case given by

$$\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}} = \left[\mathbf{dist}\left[Y_i^{(j)}(x_{k_1}); Y_i^{(j)}(x_{k_2})\right]\right]_{\forall 1 \le i \le N_{\text{puf}}; \forall 1 \le k_1 \ne k_2 \le N_{\text{chal}}; \forall 1 \le j \le N_{\text{meas}}} .$$

Similarly, the sample mean as an estimate for the expected value becomes

$$\overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}}} = \frac{2}{N_{\text{puf}} \cdot N_{\text{chal}} \cdot (N_{\text{chal}} - 1) \cdot N_{\text{meas}}} \cdot \sum \boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}} .$$

And the standard deviation estimate, i.e. the sample standard deviation, is now

$$s_{\text{Exp}(\mathcal{P})}^{\text{self}} = \sqrt{\frac{2}{N_{\text{puf}} \cdot N_{\text{chal}} \cdot (N_{\text{chal}} - 1) \cdot N_{\text{meas}} - 2} \cdot \sum \left(\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}} - \overline{\boldsymbol{D}_{\text{Exp}(\mathcal{P})}^{\text{self}}}\right)^2} .$$

Note that in certain cases, self-similarity has little meaning. Take for example an SRAM PUF for which one assumes the complete content of the SRAM to be a single response, i.e. a PUF with only a single challenge. Although it is possible to arbitrarily divide such a response in smaller parts in order to calculate the self-similarity, the practical relevance of this is not immediately clear. Thus, calculating self-similarity only makes sense if one has a specific PUF design in mind for which the response length is fixed, and for which, of course, multiple challenges exist.

This metric has, to the best of our knowledge, not been defined in any publications. We define it here for the sake of completeness and due to its usefulness for e.g. the applications discussed before.

2.6.6 Randomness tests

Because an ideal PUF behaves essentially as a "fixed" random number generator, it makes sense to run RNG test algorithms on the output of several PUF responses to the same challenge:

$$\boldsymbol{Y}(x) = [Y_1(x), Y_2(x), \ldots, Y_n(x)] .$$

Alternatively, one can compose a dataset consisting of responses from multiple PUF instances to multiple challenges:

$$\boldsymbol{Y'}(x_1, \ldots, x_n) = [Y_1(x_1), \ldots, Y_1(x_n), \ldots, Y_n(x_1), \ldots, Y_n(x_n)] .$$

Note that in doing so both inter- and intra-device uniqueness will be qualified at the same time, which is not necessarily meaningful, depending on the intended PUF application.

RNG quality evaluation is a whole field by itself; typically (one of) three de facto test suites are used: the NIST randomness test,[107] Diehard,[81] and finally Dieharder.[24] Of these, the Dieharder test suite is the most stringent, and thus provides the best assurance regarding RNG quality. An additional test suite, partially based on the NIST tests, is provided by the German BSI agency.[65]

The problem with these test suites for the purpose of PUF evaluation is that they require very large datasets, i.e. preferably in the order of several (hundred) GiB. This is problematic, because PUFs are generally many orders of magnitudes slower in generating output than RNGs are. Hence, generating a dataset acceptably large for e.g. the Dieharder test suite would take multiple years.

An additional problem is that we want to use these RNG test suites to assess whether PUF responses are unique, thus no repeated measurements for the same challenge should be included in the dataset. This means that if dataset $\boldsymbol{Y}(x)$ is used, an huge number of PUF instances would be required. Using a dataset $\boldsymbol{Y'}(x_1, \ldots, x_n)$ would somewhat ameliorate the problem, but only to a small extent, since the majority of PUF designs are *weak*, and can thus only produce a linear number of challenge-response pairs.

[24] R. G. Brown *et al.*, "Dieharder: A Random Number Test Suite" (2013).
[65] W. Killmann and W. Schindler, "AIS20/AIS31: A proposal for Functionality classes for random number generators (version 2.0)" (2011).
[81] G. Marsaglia, "Diehard Battery of Tests of Randomness" (1995).
[107] A. Rukhin *et al.*, "A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications" (2010).

These two facts prevent us from using any of these test suites for the purpose of evaluating the uniqueness of a PUF design.

2.6.7 Entropy

Consider a random variable X that can take on values $x_1, x_2, \ldots, x_n$ with probability $P(X = x_i) = p_i$, where $\sum_{i=1}^{n} p_i = 1$ and $\nexists\, p_i < 0$.

Definition 2.5. *The (Shannon) entropy of X is defined as:*

$$H_1(X) \triangleq -\sum_{i=1}^{n} p_i \log_2 p_i \,,$$

where

$$p_i \cdot \log_2 p_i = 0 \text{ if } p_i \leq 0 \,.$$

Entropy is a measure for the amount information one gets by an observation of X. It is thus also a measure for the uncertainty about the outcome of an observation. I.e. if one knows beforehand what the outcome of an observation will be, the observation in itself contains no new information. Entropy is expressed in bits, e.g. a truly random binary process A produces an entropy of one bit:

$$H_1(A) = -\sum \left(0.5 \log_2 0.5 + 0.5 \log_2 0.5\right) = -\sum \left(2 \cdot -0.5\right) = 1 \,.$$

Closely related to Shannon entropy is min-entropy H_∞. Min-entropy provides a lower bound on the amount of entropy contained within a random process by only taking the most likely outcome into account. Thus, if a process has a min-entropy of a bits of information, one is guaranteed that every observation of that process will contain at least a bits of information. Therefore $H_\infty(X) \leq H_1(X)$. Clearly, for cryptographic applications, where one wants to guarantee a certain minimum level of security, this is a more relevant metric.

Definition 2.6. *The min-entropy of a random process X is*

$$H_\infty(X) \triangleq -\log_2 \left(\max p_i\right) \,.$$

Consider a binary random process B which generates outputs with probabilities $p_0 = 0.4, p_1 = 0.6$. Where the Shannon entropy is a weighted average for the entropy

one can expect from this process, the min-entropy measures how much entropy the process will generate in the worst case. For the given example, we have

$$H_1(B) = -\sum \left(0.4 \log_2 0.4 + 0.6 \log_2 0.6\right) \approx 0.97 .$$

$$H_\infty(B) = -\log_2 \left(\max(0.4, 0.6)\right) \approx 0.74 .$$

In addition to these two metrics, we also use their fractional versions, which we call entropy density. These are defined as the entropy divided by the output length, e.g. given a random process X with an output of length n, that process' min-entropy density is

$$\rho_\infty(X) \triangleq \frac{H_\infty(X)}{n} .$$

A process that outputs data with a certain amount of entropy is called an entropy pool.

In the context of PUFs, there are two ways one can use measurement data to calculate results. The first way generates a metric which we call inter-device entropy $H^{\text{inter}}(\mathcal{P})$; it is calculated over one or more responses for multiple PUF instances. E.g. assume that the PUF output can be modeled as a binary random process which generates n independent (but not identically distributed) bits. Each of the output bits y_i has a bias $p_1^{(i)}$ towards one, i.e. $P(y_i = 1) = p_1^{(i)}$.

Definition 2.7. *The inter-device min-entropy of design $\mathcal{P}$ can be calculated as*

$$H_\infty^{inter}(\mathcal{P}) \triangleq \sum_{i=1}^{n} -\log_2 \left(\max\left(p_1^{(i)}, 1 - p_1^{(i)}\right)\right) .$$

For a given set of measurements $Y_{\text{Exp}(\mathcal{P})}$, each $p_1^{(i)}$ can be estimated as

$$\overline{p_1^{(i)}} = \frac{1}{N_{\text{puf}} \cdot N_{\text{chal}}} \cdot \sum_{\substack{1 \le j \le N_{\text{puf}} \\ 1 \le k \le N_{\text{chal}}}} \left(Y_j^{(\text{bit } i)}(x_k)\right), \ \forall\, 1 \le i \le n .$$

Note that a large number challenge-response sets (and thus PUFs) are required in order to get a good estimate for this parameter, which this calculation infeasible for many practical experiments, which have a limited number of PUFs to measure.

Furthermore, note that the formula to estimate the $p_1^{(i)}$ values is only valid in case the response bits are independent. If they are not, then one has to take this into account and modify the formula accordingly (see e.g. Maes[75, Section 4.4]). As an example of

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

the influence a certain model can have on entropy calculations, consider a binary process C which outputs five bits, the first one being random and each subsequent bit taking on the inverted value of the previous one, i.e. "10101". If one is unaware of this pattern, and treats the bits as being independent, then the calculated min-entropy equals 5 bits, since each bit in the output has a bias of exactly 0.5. However, when one takes the interdependence of the bits into account, only the first bit contains randomness, and the calculated min-entropy will equal only 1 bit.

Inter-device entropy allows one to assess uniqueness of a PUF design and the difficulty an adversary will have guessing the response to a challenge. For an ideal PUF the inter-device entropy is equal to the length of the PUF's response, i.e. if every bit in the response is independent and identically distributed (i.i.d.) with an equal probability of being one or zero, then each bit contains one bit of entropy. This is in line with the expected inter-device distance for an ideal PUF, i.e. 50%.

The inter-device entropy is thus crucial to establish the minimum response length required in order to meet a device's security requirements. E.g. if a PUF generates 200-bit responses, but those responses only contain 50 bits of min-entropy, then the maximum security level one can obtain from algorithms and protocols using those responses is 50 bits of security.

A second method of using PUF measurement data for entropy calculations is intra-device entropy $H^{\text{intra}}(\mathcal{P})$ is not particularly useful for PUF applications. However, it is for RNGs, so we will discuss it further on, in Section 2.8.

2.6.8 Compressibility

Using a theoretically perfect encoding scheme, input data can be compressed such that the output has full entropy per bit. By comparing the input to the output length, the entropy in the input can be calculated. This principle is taken advantage of by Maurer[83] in order to test the entropy in the output of an RNG. More recently, Ignatenko *et al.*[59] suggest the use of the Context-Tree Weighting compression scheme to estimate the entropy, i.e. the uniqueness, in a PUF's responses.

While this does give an indication of problematic PUF behavior, this approach suffers from similar problems as the RNG test suites: large bitstrings are required in order to get meaningful results. It is possible to create such bitstrings by concatenating the responses to multiple challenges, as is effectively done in an arbiter PUF. However, it is not clear to what extent the resulting metrics would then apply to the PUF as it is used in a realistic setting, i.e. with much shorter responses.

[59] T. Ignatenko *et al.*, "Estimating the Secrecy-Rate of Physical Unclonable Functions with the Context-Tree Weighting Method" (2006).

[83] U. M. Maurer, "A Universal Statistical Test for Random Bit Generators" (1990).

Because of these problems, we have chosen not to use this metric in the remainder of this thesis.

2.7 Error correction

As discussed, whenever a PUF instance is queried with a challenge, it will return a slightly different response, due to errors. In order to be able to use a PUF in applications, a method is required to correct erroneous bits in the response. Furthermore, PUF responses should never be transmitted as cleartext, since that will allow an attacker to attempt replay attacks, or create a model for the PUF. In this section, we will first explain the workings of cyclic error-correcting codes, and then discuss an algorithm that can be used to correct PUF responses without revealing the response.

2.7.1 Cyclic error-correcting codes

For a thorough explanation of error-correcting codes, we refer to MacWilliams and Sloane.[73] In this section, we only touch upon the bare necessities required to understand the remainder of this thesis. Furthermore, we only discuss cyclic codes, a subset of polynomial codes, which are themselves a subset of linear codes.

The process of generating a codeword for a bit string D of length k, given the generator polynomial $\mathcal{G}$ for a cyclic error-correcting code of length n, dimension k, and with number of correctable errors t, i.e. $C(n,k,t)$, is straightforward. One concatenates the data D with the remainder of dividing D, interpreted as a polynomial, by $\mathcal{G}$, i.e.

$$C = \left(D \otimes 2^{n-k}\right) \oplus \left(\left(D \otimes 2^{n-k}\right) \bmod \mathcal{G}\right).$$

In order to verify whether a received codeword C' contains any errors E, one checks if $\mathcal{G}$ divides C':

$$\begin{aligned} C' \bmod \mathcal{G} &= (C \oplus E) \bmod \mathcal{G} \\ &= \left(\left(D \otimes 2^{n-k}\right) \oplus \left(\left(D \otimes 2^{n-k}\right) \bmod \mathcal{G}\right) \oplus E\right) \bmod \mathcal{G} \\ &= \left(\left(D \otimes 2^{n-k}\right) \bmod \mathcal{G}\right) \oplus \left(\left(D \otimes 2^{n-k}\right) \bmod \mathcal{G}\right) \oplus (E \bmod \mathcal{G}) \\ &= E \bmod \mathcal{G}\,. \end{aligned}$$

[73] F. J. MacWilliams and N. J. A. Sloane, “The Theory of Error-Correcting Codes” (1978).

Thus, if C' contains no errors, the result of this calculation is zero. If it is not, the resulting value can be used to correct the errors in C', assuming that $\mathrm{HW}(E) \leq t$. The method required to correct these errors depends on the type of error-correcting code that is used.

Codes that generate codewords this way are called cyclic because circular shifts of a codeword result in another valid codeword. By shortening the dimension by m bits, the code $C(n,k,t)$ is also reduced in length by m bits to $C(n-m,k-m,t)$. E.g. from $C(255,21,55)$, one can create $C(235,1,55)$, which has the same generator polynomial and error correction capabilities, at the expense of less data one can correct. Note that such a code is no longer cyclic.

2.7.2 Secure sketching

The notion of a secure sketch was proposed by Dodis *et al.*[41] 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, i.e. to remove errors from a PUF's response data. In this work, we focus on the *syndrome construction* for binary vectors. In order to prevent confusion with similar terms later on, in the remainder of the text we refer to this construction as the *reconstruction data* secure sketch, or simply secure sketch.

We describe the operation of the *reconstruction data* secure sketch which uses a cyclic code $C(n,k,t)$ with generator polynomial $\mathcal{G}$. The secure sketch method is not restricted to the use of such codes however, any lineair code will work. The *enrollment* procedure takes as input a response $Y_i(x)$, of length n, and produces a *helper data* $h_i(x)$ bit string of length $n-k$:

$$h_i(x) = Y_i(x) \bmod \mathcal{G}\,.$$

The *recovery* procedure takes as input a different (possibly noisy) response $Y_i'(x)$ ($= Y_i(x) \oplus E$, with E a bit error vector) and the previously generated helper data $h_i(x)$, and calculates the reproduction data $s_i'(x)$:

$$\begin{aligned} s_i'(x) &= (Y_i'(x) \bmod \mathcal{G}) \oplus h_i(x) \\ &= \left(Y_i'(x) \bmod \mathcal{G}\right) \oplus (Y_i(x) \bmod \mathcal{G}) \\ &= (Y_i(x) \oplus E \oplus Y_i(x)) \bmod \mathcal{G} \\ &= E \bmod \mathcal{G}\,. \end{aligned}$$

[41] Y. Dodis *et al.*, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data" (2008).

Thus, because of the linearity of the code, the value of the reproduction data $s'_i(x)$ depends only on the errors in $Y'_i(x)$ and not on the actual responses. If $\mathrm{HW}(E) \leq t$, then E can be decoded from $s'_i(x)$, and $Y_i(x)$ recovered as $Y_i(x) = Y'_i(x) \oplus E$.

Note that error-correcting codes are normally used in a setting where one does not know beforehand what is being corrected. However, in the setting of the helper data construction, we know that the output of the error-correcting algorithm should be the response $Y_i(x)$, which was generated during the sketch generation. Whereas one would normally input a received codeword to the correction algorithm, we thus instead input the reproduction data $s'_i(x)$.

In a sense, we are supplying the error correction algorithm with the zero codeword C_0 plus errors E. Indeed, we know that in the case without errors, i.e. $s'_i(x) = C_0 = 0$, every bit of $s'_i(x)$ the correction algorithm marks as erroneous must be due to the error vector E, and not due to C_0.

Thus, the complete length of the codeword can be used for data, i.e. $n = |Y_i(x)|$. This allows for a much more efficient use of the error correcting codes, e.g. with $C(318, 174, 17)$, we can check 318 PUF response bits for errors, instead of only 174.

2.7.3 Helper data entropy loss

Only the helper data $h_i(x)$ needs to be stored in between sketching and recovering. Thus, if error correction happens within the design, then at no point in time does an adversary gain knowledge of the actual PUF response $Y_i(x)$. However, due to the linear nature of the error-correcting codes used, one has to assume that $h_i(x)$ leaks information about $Y_i(x)$.

More specifically, $h_i(x)$ discloses at most $n - k$ bits of entropy of $Y_i(x)$. Thus, one has to take into account that $Y_i(x)$ only contains a remaining $H(Y_i(x)) - (n - k)$ bits of entropy in the worst case. If the entropy in $Y_i(x)$ is less than $n - k$ bits, the remaining entropy in $Y_i(x)$ is zero.

Designs such as PUF-based cryptographic key generators do not reveal the reproduction data $s'_i(x)$ and thus one only has to take the entropy loss due to the helper data $h_i(x)$ into account. However, other designs, such as those making use of a reverse fuzzy extractor protocol,[76] transmit $s'_i(x)$, and thus partly the error vector E, over a public channel. In that case, there is an extra loss of entropy, unless $s'_i(x)$ is independent of $Y_i(x)$. This is only true if the error vector E is independent of $Y_i(x)$. Schaller *et al.*[109] show that this is not necessarily true, and present a modified

[76] R. Maes *et al.*, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs" (2012).

[109] A. Schaller *et al.*, "Eliminating Leakage in Reverse Fuzzy Extractors" (2014).

protocol which solves this problem.

Unfortunately, a large number of publications do not take this entropy loss into account,[21,22,50,78,113] which leads to the designs in all of those works being weaker than advertised. The importance of taking this loss of entropy into account was recently stressed once more by Koeberl *et al.*[67] Note that this entropy reduction not only applies to the secure sketch construction, but should be kept in mind for any PUF application where data derived from a response is made publicly available.

2.7.4 Dealing with biased responses

Biased PUF output is problematic when used in conjunction with an error correction method such as a secure sketch. Due to helper data entropy loss, it is often the case that one cannot assume any remaining entropy in the corrected PUF response.

As a simple example to illustrate this, consider a 318-bit PUF response with a 75% bias towards one, and a maximum error rate no more than 5%. This allows us to use a $C_{BCH}(318, 174, 17)$ code for error correction. Because of the bias, the min-entropy of the PUF response is only $-\log_2 0.75 \cdot 318 = 132$ bits of entropy, assuming no further correlation between the bits. Because of information loss due to the helper data, a total of $318 - 174 = 144$ bits of entropy have to be considered lost. Therefore, one has to assume only $\max(0, 132 - 144) = \max(0, -12) = 0$ bits of entropy remain in the corrected PUF output. Thus, due to the high bias, the corrected response has no entropy remaining at all.

In these next paragraphs, we present a simple, lightweight method which makes it possible to use such biased PUF responses and still have them retain entropy after error correction. Our method requires only a single readout of the PUF response, and thus does not significantly increase response generation times. To the best of our knowledge, this idea has not been published before.

[21] C. Böhm *et al.*, "A Microcontroller SRAM-PUF" (2011).
[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).
[50] J. Guajardo *et al.*, "FPGA Intrinsic PUFs and Their Use for IP Protection" (2007).
[67] P. Koeberl *et al.*, "Entropy Loss in PUF-based Key Generation Schemes: The Repetition Code Pitfall" (2014).
[78] R. Maes *et al.*, "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs" (2009).
[113] V. van der Leest *et al.*, "Soft Decision Error Correction for Compact Memory-Based PUFs Using a Single Enrollment" (2012).

2.7.4.1 Enrollment step

Our method requires two extra steps. The first one takes place during enrollment, and consists of applying the von Neumann debiasing algorithm[94] to the PUF response. This algorithm works as follows: the input bitstream is split up into pairs B_i of two bits. The output b_i of the algorithm is

$$b_i = \begin{cases} \varnothing & \text{if } B_i = 00 \vee 11 \\ 0 & \text{if } B_i = 01 \\ 1 & \text{if } B_i = 10\,, \end{cases}$$

where $\varnothing$ means the input is discarded and no output is generated. As long as there is no correlation between the bits, except for bias, then the result of this processing step is an unbiased bit stream. For an input bitstream of length n with bias p towards one, and bias $q = 1 - p$ towards zero, the expected number of output bits l is

$$l = npq\,.$$

For each output bit, not only the bit itself, but also the index of the input pair which generated this bit is recorded. The resulting list of indexes is helper data h_{vn} which will be used in subsequent readouts of the PUF. The helper data h_{vn} contains only information on the location of a bit toggle in the PUF response, but no information whatsoever on the value of the response bits themselves. Thus, h_{vn} does not require any loss of entropy to be taken into account.

The output of this step is a bit string without bias which can be used as the input for a "regular" PUF error correction method, such as a secure sketch.

2.7.4.2 Reconstruction step

The second step of the method is executed during reconstruction, each time the PUF is read out. Using h_{vn}, bit pairs B'_i are extracted from the PUF response at locations which generated an output bit at enrollment. On these pairs a modified version of the von Neumann debiasing algorithm is run:

$$b'_i = \begin{cases} 0 & \text{if } B'_i = 0x \\ 1 & \text{if } B'_i = 1x\,, \end{cases}$$

where $x \in \{0, 1\}$.

[94] J. von Neumann, "Various Techniques Used in Connection With Random Digits" (1951).

Since the output of the modified van Neumann algorithm only depends on the value of the first bit, errors in the output must be due to an erroneous first bit. Therefore, assuming the error rate for both zero and one bits is the same, i.e. $p_{e,0} = p_{e,1}$, it is trivial to see that the bit error rate p'_e of the processed data is $p'_e = p_e$. Thus, this method does not increase the error rate of the original PUF response.

Unfortunately, the above assumption will generally not hold in a real life scenario. The stronger the bias of a PUF's responses, the higher the bit error rate will be for the bit value that appears the least in the responses. Therefore, the output of the above method will have a higher bit error rate than the unprocessed PUF response. The bit error rate is upper bounded by

$$p'_e \leq \max(p_{e,0}, p_{e,1}),$$

and will on average equal $p'_e = \frac{1}{2}(p_{e,0} + p_{e,1})$.

After processing, the output of the second step can then be used the same way as unbiased PUF data is handled. Of course, quite a few bits will be lost due to the processing. Continuing with the previous example, if we require 318 response bits to be error corrected, the expected required input length for the debiasing algorithm is $318/(0.75 \cdot 0.25) = 1696$ bits. After error correction, we would then be left with an expected minimum of 174 bits of entropy.

The crucial idea, which makes our method both lightweight and efficient, is that the von Neumann algorithm is not executed directly on the PUF response during reconstruction, but instead on extracted bit pairs. If instead the whole response is used, the output of the debiasing would be a different length each time, which would require advanced solutions in order to be able to use standard error correction methods on the output.

2.7.4.3 Increasing the number of extracted bits

The biggest drawback to our method is that a quite a few bits are lost to the debiasing algorithm. Methods exist to extract a number of bits arbitrarily close to the amount of Shannon entropy in the bit stream.

Here we discuss how to adapt a method published by Peres[101] for PUF-based applications. Peres' method iterates the von Neumann algorithm on data extracted from discarded pairs and extracts information both from the value of pairs as well as from their position.

Extracting information from discarded pairs works by considering only one bit in those pairs and feeding those bits back into the von Neumann algorithm. E.g. given

[101] Y. Peres, "Iterating Von Neumann's Procedure for Extracting Random Bits" (1992).

a response "0011", we get pairs "00 → 0" and "11 → 1". From these, we generate an output "01 → 0". One can keep iterating this by considering pairs discarded in the previous step. Helper data $h_{vn'}$ for this step would consist of the locations of the first bit of all sets of bit pairs which generate an output value.

In order to extract even more information from the data, Peres also takes into account the order in which values appear. First bits are generated depending on whether a pair has two identical bits. E.g. given a response "0100", we get "01 → 1" and "00 → 0". These generated bits are then fed back into the von Neumann algorithm to generate output, i.e. in the example "10 → 1". The helper data $h_{vn''}$ for this step would consist of the locations of the first bit of a sequence of four bits which will generate an output.

By using this iterated algorithm approximately $H_1(0.75) = 258$ bits can be extracted from our example 318 bit biased PUF response.

Unfortunately, the helper data necessary to extract bits from the order of values leaks data on the PUF response. By checking whether a certain location in $h_{vn''}$ exists either in h_{vn} or $h_{vn'}$, one can find the value which will be generated using $h_{vn''}$. For clarity, we will illustrate this with an example.

Assume a location i_0 in $h_{vn''}$. From this location two bits, b_0'' and b_1'', are generated which will get fed back into the von Neumann algorithm. If i_0 also exists in h_{vn}, then the two bits which generate b_0'' must be different, and thus $b_0'' = 1$. Furthermore, since i_0 is in $h_{vn''}$, we know that the von Neumann algorithm generates an output from the combination $b_0''b_1''$, and thus $b_0'' \neq b_1''$. Therefore, the bit generated from location i_0 due to $h_{vn''}$ will be "1". Similarly, if i_0 had been in $h_{vn'}$ and $h_{vn''}$ instead, the generated bit due to $h_{vn''}$ would have been "0".

Note that knowledge of $h_{vn''}$ and only one of h_{vn} or $h_{vn'}$ is enough to compute all of the data which $h_{vn''}$ will generate. Clearly, the $h_{vn''}$ helper data cannot be used in a cryptographic setting. However, the $h_{vn'}$ data can still be used, since it doesn't leak information on the response.

The chance for a bit pair having two identical bits is $p^2 + q^2$, and the chance for a pair to be "11" from the set of all such pairs is $\frac{p^2}{p^2+q^2}$. Therefore, the expected number of bits l' generated by using both helper data h_{vn} and $h_{vn'}$ is:

$$\begin{aligned} l' &= \frac{n}{2} \cdot 2pq + \frac{n}{4} \cdot (p^2 + q^2) \cdot \left(2 \cdot \frac{p^2}{p^2 + q^2} \cdot \frac{q^2}{p^2 + q^2} \right) \\ &= n \cdot \left(pq + \frac{1}{2} \cdot \frac{p^2 q^2}{p^2 + q^2} \right). \end{aligned}$$

Similarly to our earlier statement, if the modified von Neumann algorithm listed above is applied, then the output value generated using $h_{vn'}$ is equal to the first of

those four bits. Therefore, the output will only be incorrect if this first bit is erroneous, what happens to the other three bits does not matter. If the bit error rate for both bit values is equal, then once again the bit error rate p'_e for the data generated using $h_{vn'}$ is thus $p'_e = p_e$. Since data generated using h_{vn} and $h_{vn'}$ is simply concatenated, the bit error rate for the completely processed response is also equal to p_e. The bit error rate for cases where $p_{e,0}$ and $p_{e,1}$ are not equal is the same as for data generated using h_{vn}.

Using this improved method on our example PUF, the output length will be approximately 21.56% of the input length. I.e. the response length n needs to be approximately 1475 bits for $l' = 318$. Compared to using only h_{vn}, this is an improvement of approximately 13%, at the cost of 13% more helper data. Iteratively processing dropped pairs can further improve the output length.

2.7.4.4 Potential security issues

Even though the presented method does not directly leak any information about the value of the response through the helper data, it does leak some information. Any pair whose location which is not present in h_{vn} must consist of two equal values. Using this information, one can easily calculate the bias of the response. For PUFs which are easily modeled, i.e. arbiter PUFs, such information could greatly ease modelling. Thus, for such PUFs, it is likely a bad idea to use the proposed debiasing method.

Another potential weakness is the susceptibility of the helper data to manipulation attacks. If an attacker is able to systematically modify parts of the helper data, he will eventually be able to reduce the PUF's response to two possible values. One method of preventing such attacks makes use of a hash function: both the helper data and the final corrected PUF response are hashed and the digest is stored together with the helper data. During reconstruction the hash is recalculated using the available helper data and corrected PUF response, and is compared to the stored hash. For more information on manipulation attacks and prevention schemes, we refer to e.g. Delvaux and Verbauwhede.[36,37]

It is clear that the presented solution is not yet complete and will not work for all types of PUFs. We therefore consider it a starting point for future research.

[36] J. Delvaux and I. Verbauwhede, "Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation" (2014).

[37] ——, "Key-recovery Attacks on Various RO PUF Constructions via Helper Data Manipulation" (2014).

2.8 Random number generation

Before finishing this chapter, it is important to mention the possibility of using a PUF not only for its "fixed" response generation behavior, but for generating random numbers as well.[57,114,115]

Process variation, as explained in Section 2.3.1, leads to two types of random behavior. The "static", fingerprint-like response behavior originates from circuit elements which are rather strongly mismatched due to process variations. However, when comprised of well-matched elements, certain designs, e.g. an SRAM cell, will function as a TRNG, due to very small differences in operating conditions and circuit noise. This is a "variable" random behavior, which leads to errors in a PUF's response.

Whereas PUF designs most often assume that errors in the response are fixed, i.e. thrown away, one can instead use the errors as a source of random numbers. These two ideas can be combined to create a circuit that behaves both as a PUF and an RNG. Such a design can serve as a building block providing two critical requirements for cryptographic systems: key generation/storage and random number/nonce generation. One example of such a circuit is the design by Varchola *et al.*[124]

The intra-device entropy metric mentioned earlier is a good way to qualify the usability of errors in a PUF's response for an RNG design. It gives an indication of the amount of randomness one can expect in a PUF's response, and thus the difficulty for an adversary to guess the erroneous bits, even if he knows the expected response to the challenge (i.e. the response retrieved at enrollment).

Definition 2.8. *The intra-device min-entropy of instance* puf_i *can be calculated as*

$$H_\infty^{intra}(\mathrm{puf}_i) \triangleq \sum_{i=1}^{n} -\log_2\left(\max\left(p_1^{(\mathrm{puf}_i;\, bit\ j)}, 1 - p_1^{(\mathrm{puf}_i;\, bit\ j)}\right)\right) .$$

This formula is almost the same as for the inter-device entropy (see Definition 2.7), however the crucial difference is in the bias values $p_1^{(\mathrm{puf}_i;\,\mathrm{bit}\ j)}$. For a dataset $Y_{\mathrm{Exp}(\mathcal{P})}$, using the same assumptions about $\mathcal{P}$ as in Section 2.6.7 (i.e. independent bit bias), each bias value can in this case be estimated as

$$\overline{p_1^{(\mathrm{puf}_i;\,\mathrm{bit}\ j)}} = \max_{k=1}^{N_{\mathrm{chal}}}\left(\frac{1}{N_{\mathrm{meas}}} \cdot \sum_{l=1}^{N_{\mathrm{meas}}}\left(Y_i^{(l;\,\mathrm{bit}\ j)}(x_k)\right)\right),\ \forall\, 1 \le i \le N_{\mathrm{puf}};\ \forall\, 1 \le j \le n\,.$$

[57] D. E. Holcomb *et al.*, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers" (2009).

[114] V. van der Leest *et al.*, "Efficient Implementation of True Random Number Generator Based on SRAM PUFs" (2012).

[115] E. van der Sluis *et al.*, "Random Number Generating System Based on Memory Start-Up Noise" (2012).

[124] M. Varchola *et al.*, "New Universal Element with Integrated PUF and TRNG Capability" (2013).

Note that for each puf_i, we are taking as bias value for bit j the maximum bias encountered over all challenges. This in order to guarantee that a worst-case bias value is used when calculating the min-entropy. Calculating the intra-device min-entropy for the design $\mathcal{P}$ is done by selecting the worst case (i.e. lowest) value from all PUF instances. This is again done to ensure that the calculated value gives a lower bound to the amount of intra-device entropy that one can expect.

Definition 2.9. *The intra-device min-entropy for design $\mathcal{P}$ is calculated as*

$$H_\infty^{intra}(\mathcal{P}) \triangleq \min_{i=1}^{N_{puf}} \left(H_\infty^{intra}(\mathsf{puf}_i) \right) .$$

Note that calculating an accurate estimate of $H^{\text{intra}}(\mathcal{P})$ requires many PUF instances, i.e. it is hard to calculate, for the same reason accurately estimating $H^{\text{inter}}(\mathcal{P})$ is hard. However, it is easy to get a very good estimate for $H^{\text{intra}}(\mathsf{puf}_i)$, since one can just keep querying puf_i until an appropriate number of responses have been gathered.

If one is instead interested in the intra-device Shannon entropy $H_1^{\text{intra}}(\mathcal{P})$, instead of using minimum and maximum values, one will most likely want to average out the calculations over multiple challenges and multiple PUF instances. This matches better with Shannon entropy being a measure of average expected entropy.

2.9 Summary

We started this chapter with a description of what constitutes a PUF and gave an overview of the most important PUF properties. Next, we gave an overview of applications which are made possible, or at the very least greatly improved, by PUFs. The underlying processes leading to PUF behavior were described, and example designs for three important types of PUFs were discussed. Next, we discussed several metrics by which the quality of PUFs and RNGs are assessed. Then, we described an algorithm used to securely correct erroneous bits in PUF responses and the impact this has on available response entropy. As part of this, we have also presented an as yet unpublished method which makes it possible to securely use heavily biased PUFs with error correction methods. The presented method is only a starting point, and requires more research before it can be considered a full-fledged solution. Finally, we discussed how PUFs can be used to generate random numbers.

3 PUFKY: An Area-Efficient Key Generation Module

That's not magic, that's just engineering!
— Terry Pratchett, Interesting Times (1994)

With the theorethical background on PUFs out of the way, we will now discuss a practical, efficient implementation of a PUF-based key generation module, named PUFKY. PUFKY is designed to be used as a flexible black box module, which can easily be configured at design time to meet various constraints, such as required key length or acceptable failure rate. Our design was at the time of publication the first practical PUF-based key generation building block for FPGAs.

Content sources

R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, E. Prouff and P. Schaumont, Eds., ser. Lecture Notes in Computer Science, vol. 7428, Leuven, Belgium: Springer, 2012, pp. 302–319
Contribution: Main author, responsible for all hardware designs, except PUF & repetition decoder.

A. Van Herrewege and I. Verbauwhede, "Tiny, Application-Specific, Programmable Processor for BCH Decoding", in *International Symposium on System on Chip (SoC)*, Tampere, Finland: IEEE, 2012, 4 pp.
Contribution: Main author.

3.1 Introduction

Our aim is to construct a practical, efficient PUF-based key generator. Furthermore, the design should easily permit changes to both the length of the generated key, and the maximum allowable error rate. Since PUF responses are generally noisy and of relatively 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[41,76] perform exactly these two functions and can be immediately applied for this purpose, as suggested in a number of earlier PUF key generator proposals. Guajardo *et al.*[50] 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.*[22] who propose a concatenated block code configuration, and Maes *et al.*[78] who propose to use a soft-decision decoder. Yu *et al.*[127] propose a configuration based on ring oscillator PUFs and apply an alternative error-correction method.

Our main contribution is a highly practical PUF-based cryptographic key generator design, 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,[22,67] 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.

The two core building blocks of PUFKY are the novel ring oscillator-based PUF and the extremely lightweight microcontroller for BCH decoding. In this chapter, we

[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).
[41] Y. Dodis *et al.*, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data" (2008).
[50] J. Guajardo *et al.*, "FPGA Intrinsic PUFs and Their Use for IP Protection" (2007).
[67] P. Koeberl *et al.*, "Entropy Loss in PUF-based Key Generation Schemes: The Repetition Code Pitfall" (2014).
[76] R. Maes *et al.*, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs" (2012).
[78] R. Maes *et al.*, "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs" (2009).
[127] M.-D. M. Yu *et al.*, "Lightweight and Secure PUF Key Storage Using Limits of Machine Learning" (2011).

will only briefly touch upon the PUF design, and instead focus on the supporting error-correction microcontroller.

The design of the controller can easily be adapted to other types of PUFs, with minimal required manual changes to its firmware. For our chosen "default" PUFKY configuration (which generates a 128-bit key with a 10^{-9} failure rate), the microcontroller, which implements a BCH(318, 174, 17) decoder, only requires 68 slices of a Xilinx Spartan-6 FPGA, or 1 130 GE in 130 µm UMC ASIC technology.

3.2 Background

The secure sketch algorithm as presented in Section 2.7 uses a single linear error-correcting code to repair errors in PUF responses. However, Bösch *et al.*[22] 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 secure sketch based on the concatenation of a repetition code $C_1(n_1, 1, t_1 = \lfloor \frac{n_1-1}{2} \rfloor)$ as an inner code and a second linear block code $C_2(n_2, k_2, t_2)$ as an outer code, are given in the second column of Table 3.1.

Table 3.1: Parameter constraints for secure sketches, for a single linear code and for a repetition code concatenated with a linear code, for inputs with an entropy density ρ and error rate p_e. The maximum failure rate after error correction is p_{fail}, and the output has m bits remaining entropy.

	$C(n,k,t)$	$C_2(n_2,k_2,t_2) \circ C_1(n_1,1,t_1 = \lfloor \frac{n_1-1}{2} \rfloor)$
Rate	$\frac{k}{n} > 1 - \rho$	$\frac{k_2}{n_1 n_2} > 1 - \rho$
Correction	$t \geq \mathrm{B}^{-1}_{n,p_e}\left((1-p_{\text{fail}})^{\frac{1}{r}}\right)$, with $r = \left\lceil \frac{m}{k-n(1-\rho)} \right\rceil$	$t_2 \geq \mathrm{B}^{-1}_{n_2,p'_e}\left((1-p_{\text{fail}})^{\frac{1}{r}}\right)$, with $p'_e = 1 - \mathrm{B}_{n_1,p_e}(t_1)$, $r = \left\lceil \frac{m}{k_2 - n_1 n_2 (1-\rho)} \right\rceil$

The design parameters of the secure sketch construction are mainly determined by the selection of an appropriate linear block code $C(n, k, t)$. In order to yield a meaningful secure sketch, $C(n, k, t)$ needs to meet some constraints determined by the available PUF, with error rate p_e and entropy density ρ, and by the required remaining entropy

[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).

m and reliability $1 - p_{\text{fail}}$ of the output of the secure sketch. These constraints are listed in the first column of Table 3.1. The *rate* and *correction* constraints 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. Finally, a *practicality* constraint, not listed in the table, is defined simply as "efficient decoding algorithms must exist for the chosen code".

For example, assume a PUF with $n = 127$, $\rho = 95\%$, and $p_e = 5\%$. Let's say that we require $m \geq 50$ bits of entropy after error correction and a failure rate $p_{\text{fail}} \leq 10^{-6}$. We will use the formulas for one code, i.e. those in the left column of Table 3.1. The rate formula gives us $k/n > 1 - \rho$. By rearranging this formula, we get $n \cdot \rho > n - k$, i.e. the entropy in the PUF response needs to be larger than the amount of entropy lost due to helper data. For this example, we find $k > 7$.

The formula for r rounds up the required entropy after correction divided by the actual entropy after correction, i.e. it tells us how many blocks of n bits need to be corrected, and afterwards concatenated, in order to obtain at least m bits of entropy in the corrected response. Since the formula of r depends on k, we can not calculate a fixed value for it, since we only have an inequality for k. However, if we pick a code $C(n, k, t)$, then both k and t are fixed, and we can verify whether the inequality for t holds, i.e. the inequality in the second row of Table 3.1. This demonstrates the iterative procedure required for finding an adequate code. Note that there is no guarantee that such a code can be found for the given PUF parameters, so in some cases a redesign of the PUF will be necessary in order to obtain the required performance.

For this example, multiple codes are possible. The best BCH code in this case, i.e. the one that minimizes the entropy loss due to helper data while still meeting all requirements, is $C_{BCH}(127, 64, 21)$. This leaves us with $m = 64 - 127 \cdot (1 - 0.95) = 57.65$ bits of entropy in the corrected response. To obtain the required failure rate, we need $t \geq 21$, thus the inequality is satisfied.

3.2.1 Repetition code construction

A repetition code is an exceptionally simple error-correcting code, based on the simple premise that if one transmits the same bit multiple times, the majority of bits will be received without errors.

Constructing an n-bit codeword consist of simply repeating the 1-bit data word n times. In order to decode the word, a majority vote is taken and the received data bit is decoded to whichever value appears most often in the received codeword. The generator polynomial for a repetition code has a coefficient equal to one at each

position, e.g. for $C_{REP}(3,1,1)$ we have $\mathcal{G} = x^2 + x + 1$. For a channel with error rate p_{err}, a repetition code $C_{REP}(n,1,t = \lfloor \frac{n-1}{2} \rfloor)$ has a decoding error rate p_{fail}:

$$p_{\text{fail}} = 1 - \mathrm{B}_{n,p_{\text{err}}}(t) \ .$$

I.e. the failure probability for the decoding step is equal to one minus the probability that t or less bits are erroneous. A repetition code for data consisting of multiple bits can be created by simply concatenating single data bit repetition codes.

The following simple example illustrates the use of a repetition code in a secure sketch construction. Assuming the PUF outputs 5 bits, a $C_{REP}(5,1,2)$ code will be used. Let's say the response at enrollment was $Y = 11010$, therefore

$$\begin{aligned} h &= Y \bmod \mathcal{G} \\ &= (x^4 + x^3 + x) \bmod (x^4 + x^3 + x^2 + x + 1) \\ &= x^2 + 1 \\ &= 0101 \, . \end{aligned}$$

Note that the length of h is equal to $n - k = 4$. A subsequent query of the PUF returns $Y' = 10011$. This response contains 2 errors:

$$E = Y \oplus Y' = 11010 \oplus 10011 = 01001 \, ,$$

which is within the correction limit of the code. Calculating the reproduction data s' gives us

$$\begin{aligned} s' &= (Y' \bmod \mathcal{G}) \oplus h \\ &= \big((x^4 + x + 1) \bmod (x^4 + x^3 + x^2 + x + 1)\big) \oplus (x^2 + 1) \\ &= (x^3 + x^2) \oplus (x^2 + 1) \\ &= x^3 + 1 \\ &= 1001 \, . \end{aligned}$$

Remember from Section 2.7 that we are effectively calculating $s' = C_0 \oplus E$. Thus, there are two ways s' can take on this value, either $E = x^3 + 1$ or $E = x^4 + x^2 + x$. Note that these two options are each other's binary inverse. Out of the two options, one must have $\mathrm{HW}(E) > t \ (= 2)$, and thus we would not be able to correct the response if that were the correct error vector. Thus, one chooses the option which has a Hamming weight $\leq t$. A simple way to calculate the value of the data error bit e_1, i.e. the most

significant error bit, which determines whether the bits in the reproduction data s' need to be inverted, is

$$e_1 = (\mathrm{HW}(s') > t) \ .$$

The full error vector can then be reconstructed as the concatenation of e_1 and s' XORed with e_1 at each position. In our example, we thus have

$$\begin{aligned} Y &= Y' \oplus E \\ &= Y' \oplus (e_1 \mid (s' \oplus e_1 e_1 \ldots e_1)) \\ &= 10011 \oplus 01001 \\ &= 11010 \ . \end{aligned}$$

3.2.2 BCH code construction

As shown by Bösch *et al.*,[22] BCH codes[23,55] are very efficient when used as the outer code in a PUF error correction design. Despite the belief[22] that a BCH decoder implementation is large, and thus not suited for a small PUF-based design, we nevertheless choose to use this code for PUFKY.

A BCH code $C_{BCH}(n, k, t)$, defined by its generator polynomial $\mathcal{G}$, is constructed as follows. First, one selects the size u of the underlying field $\mathbb{F}_{2^u}$. Let $A \in \mathbb{F}_{2^u}$ be of order $\mathrm{ord}(A)$. For each $A^i, i = b, \ldots, b + 2t - 1$, define $\mathcal{M}_i$ as the minimal polynomial of A^i. $\mathcal{G}$ is defined as the least common multiple of all $\mathcal{M}_i$. This gives a code of length $n = \mathrm{ord}(A)$, with $k = n - \mathrm{ord}(\mathcal{G})$, and $\mathrm{ord}(\mathcal{G}) = t$. In this chapter, we only consider codes for which $A = \alpha$, a primitive element of $\mathbb{F}_{2^u}$, and $b = 1$, so-called primitive narrow-sense BCH codes.

BCH decoding consists of a three step process: syndrome calculation, error polynomial calculation and error position calculation. Each of these steps is explained in more detail in the next paragraphs.

3.2.2.1 Syndrome calculation

The first decoding step is calculating the so-called *syndromes*. In order to calculate the *syndrome*, one takes a received codeword C', i.e. the sum of an error-free codeword

[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).
[23] R. C. Bose and D. K. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes" (1960).
[55] A. Hocquenghem, "Codes Correcteurs d'Erreurs" (1959).

C and an error vector E, and evaluates it as a polynomial. The syndromes are the evaluations of $C'(x)$ for $x = \alpha^i$, with $i = 1, \ldots, 2t$. Thus, syndrome S_i is defined as

$$\begin{aligned} S_i &= C'(\alpha^i) = C(\alpha^i) \oplus E(\alpha^i) \\ &= E(\alpha^i)\,. \end{aligned}$$

The evaluation of the C' polynomial can be calculated a lot faster than straightforward polynomial evaluation, since every coefficient must be either 0 or 1. This allows one to replace all multiply-accumulate operations in an implementation with conditional additions.

3.2.2.2 Error location polynomial calculation

Suppose we have an error vector $E = x^{l_1} + x^{l_2} + \ldots + x^{l_y}$. Then the value of the first three syndromes is

$$\begin{aligned} S_1 &= \alpha^{l_1} + \alpha^{l_2} + \ldots + \alpha^{l_y} \\ S_2 &= \alpha^{2l_1} + \alpha^{2l_2} + \ldots + \alpha^{2l_y} \\ S_3 &= \alpha^{3l_1} + \alpha^{3l_2} + \ldots + \alpha^{3l_y}\,. \end{aligned}$$

The Berlekamp-Massey (BM) algorithm,[12,82] when given a list of syndromes S_i, returns an error location polynomial

$$\begin{aligned} \Lambda(x) &= (\alpha^{l_1}x + 1)\cdot(\alpha^{l_2}x + 1)\cdot \ldots \cdot (\alpha^{l_y}x + 1) \\ &= (x - \alpha^{-l_1})\cdot(x - \alpha^{-l_2})\cdot \ldots \cdot (x - \alpha^{-l_y})\,. \end{aligned}$$

One of the problems with the original BM algorithm is that it requires an inversion of an element $A \in \mathbb{F}_{2^u}$ in each of the $2t$ iterations it executes internally. To eliminate this costly operation, Burton[25] devised an inversionless version of the algorithm. Multiple authors have suggested improvements to this algorithm in the form of space-time tradeoffs.[100,103,108]

[12] E. Berlekamp, "On Decoding Binary Bose-Chadhuri-Hocquenghem Codes" (1965).
[25] H. Burton, "Inversionless Decoding of Binary BCH codes" (1971).
[82] J. Massey, "Shift-Register Synthesis and BCH Decoding" (1969).
[100] J.-I. Park *et al.*, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm" (2009).
[103] I. Reed and M. Shih, "VLSI Design of Inverse-Free Berlekamp-Massey Algorithm" (1991).
[108] D. Sarwate and N. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders" (2001).

3.2.2.3 Error location calculation

Finding the roots of $\Lambda(x)$ gives the location of the errors in R. The Chien search algorithm[28] is an efficient way of evaluating all possible values of α^i, thus allowing to check for every bit position if it is erroneous. It does this by improving multiplications in the evaluation formula to constant factor multiplications by noting that intermediate results for $\Lambda(\alpha^{i+1})$ differ by a constant factor from intermediate results for $\Lambda(\alpha^i)$:

$$\begin{aligned}
\Lambda(\alpha^i) &= \lambda_y \cdot \alpha^{it} &&+ \ldots + \lambda_1 \cdot \alpha^i &&+ \lambda_0 \\
&\equiv \lambda_{y,i} &&+ \ldots + \lambda_{1,i} &&+ \lambda_{0,i} \\
\Lambda(\alpha^{i+1}) &= \lambda_y \cdot \alpha^{(i+1)t} &&+ \ldots + \lambda_1 \cdot \alpha^{i+1} &&+ \lambda_0 \\
&= \lambda_{y,i} \cdot \alpha^t &&+ \ldots + \lambda_{1,i} \cdot \alpha &&+ \lambda_{0,i} \\
&\equiv \lambda_{y,i+1} &&+ \ldots + \lambda_{1,i+1} &&+ \lambda_{0,i+1} \,.
\end{aligned}$$

3.2.3 Cryptographic key generation

As discussed in Section 2.2.3, the fuzzy extractor by Dodis *et al.*[41] can be used to generate reliable keys using PUFs. Their proposed method of accumulating entropy with a strong extractor brings with it heavy losses in entropy and requires a random seed. We therefore follow the recommendations described in multiple reference works,[8,44,64] and use a cryptographic hash function as entropy accumulator instead.

The amount of data to be accumulated to reach a sufficient entropy level depends on the (estimated) entropy rate of the considered source. 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.

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).

[28] R. Chien, "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes" (1964).

[41] Y. Dodis *et al.*, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data" (2008).

[44] D. Eastlake *et al.*, "Randomness Requirements for Security" (2005).

[64] J. Kelsey *et al.*, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator" (1999).

3.3 Design

The top-level architecture of our PUFKY PUF-based key generator is shown in Figure 3.1. As a PUF, we use a ring oscillator PUF (ROPUF) which produces high-entropy outputs based on the frequency ordering of a selection of ring oscillators, as described in Section 3.3.1. 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 $C_{REP}(n_{REP}, 1, \lfloor \frac{n_{REP}-1}{2} \rfloor)$ with n_{REP} odd and a BCH code $C_{BCH}(n_{BCH}, k_{BCH}, t_{BCH})$. Our reference implementation requires $C_{REP}(7,1,3)$ and $C_{BCH}(318,174,17)$ in order to guarantee that the 128-bit key has an error rate $\leq 10^{-9}$. The design of the helper data generation and repetition error decoder blocks used in the secure sketching is described in Section 3.3.2, the design of the BCH decoder is described in more detail in Section 3.4. To accumulate the remaining entropy after secure sketching, we apply the recently proposed light-weight cryptographic hash function SPONGENT.[19]

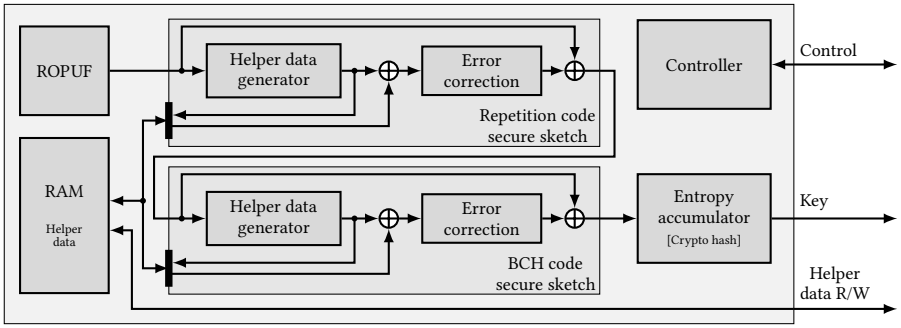


Figure 3.1: PUFKY high-level architecture.

3.3.1 ROPUF design

Our ROPUF design is inspired by the design of Yin and Qu[126] which generates a response based on the frequency ordering of a set of oscillators. An oscillator's frequency is measured by counting the number of oscillations in a fixed time interval. To reduce the area taken up by frequency counters, oscillators are ordered in b batches of a oscillators sharing a counter. In total, our ROPUF design contains $a \times b$ oscillators of which sets of b can be measured in parallel. The measurement time is equal to a fixed number of cycles of an independent on-chip ring oscillator and, in our case,

[19] A. Bogdanov *et al.*, "SPONGENT: A Lightweight Hash Function" (2011).
[126] C.-E. D. Yin and G. Qu, "LISA: Maximizing RO PUF's Secret Extraction" (2010).

is equal to approximately 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 Figure 3.2.

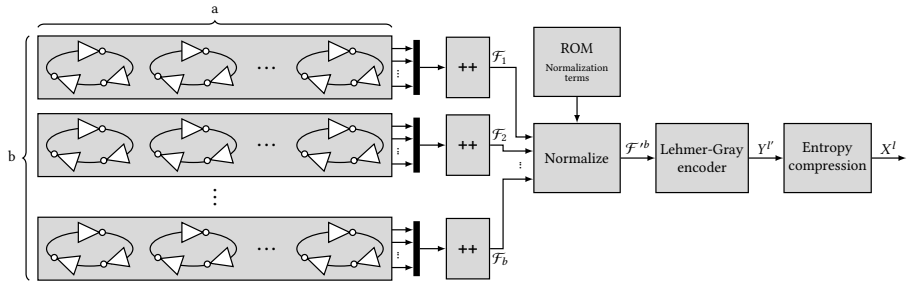


Figure 3.2: ROPUF architecture.

As discussed in Section 3.2, 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 process to convert b frequency measurements $\boldsymbol{\mathcal{F}}^b = (\mathcal{F}_0, \ldots, \mathcal{F}_{b-1})$ in an ℓ-bit response $X = (X_0, \ldots, X_{\ell-1})$:

1. *Frequency normalization:* remove structural bias from the measurements.
2. *Order encoding:* encode the normalized frequency ordering to a stable bit vector, while preserving all ordering entropy.
3. *Entropy compression:* compress the order encoding to maximize the entropy density without significantly increasing the bit error probability.

3.3.1.1 Frequency normalization

Only a portion of a measured frequency $\mathcal{F}_i$ is random, and only a portion of that randomness is caused by the effects of process variations on the considered oscillator. Analysis by Maiti *et al.*[80] demonstrates that $\mathcal{F}_i$ 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 has a potentially severe impact on the randomness of the frequency ordering. It is reasonable to assume for the frequencies

[80] A. Maiti *et al.*, "A Large Scale Characterization of RO-PUF" (2010).

$\mathcal{F}_i$ to be independent, but due to the oscillator-dependent structural bias we cannot consider them to be identically distributed, since each $\mathcal{F}_i$ has a different expected value $\mu_{\mathcal{F}_i}$.

The ordering of $\mathcal{F}_i$ is largely determined by the deterministic ordering of $\mu_{\mathcal{F}_i}$ and not by the effect of random process variations on $\mathcal{F}_i$. We prevent this by subtracting the sample average $\overline{\mathcal{F}_i}$, calculated over many $\mathcal{F}_i$ on many devices, from $\mathcal{F}_i$, thereby obtaining the normalized frequency $\mathcal{F}_i' = \mathcal{F}_i - \overline{\mathcal{F}_i}$. Assuming $\overline{\mathcal{F}_i} \approx \mu_{\mathcal{F}_i}$, the normalized frequencies $\mathcal{F}_i'$ are i.i.d.

Calculating $\overline{\mathcal{F}_i}$ needs to be performed only once for a single design.When these normalization terms are known with high accuracy, they can be included in the design, e.g. using a ROM.

Order encoding Sorting a vector $\boldsymbol{\mathcal{F}}'^b$ of normalized frequencies 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 $\boldsymbol{\mathcal{F}}'^b$.

Since the elements of $\boldsymbol{\mathcal{F}}'^b$ are i.i.d., each of the $b!$ possible orderings is equally likely to occur,[125] leading to $H_1(Y^{\ell'}) = \log_2 b! = \sum_{i=2}^{b} \log_2 i$. An optimal order encoding has a high entropy density but a minimal sensitivity to noise on the $\mathcal{F}_i'$ values. Since deviations from the "standard" frequency measurement will most likely manifest themselves as order swaps between two neighboring values, it would be prefered that such an error only influences a single response bit. 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 efficient to obtain since it does not require explicit value sorting. It represents the sorted ordering of $\boldsymbol{\mathcal{F}}'^b$ as a coefficient vector $\boldsymbol{L}^{b-1} = (L_0, \ldots, L_{b-2})$ with $L_i \in \{0, 1, \ldots, (i+1)\}$. It is clear that $\boldsymbol{L}^{b-1}$ can take $2 \cdot 3 \cdot \ldots \cdot b = b!$ possible values which is exactly the number of possible orderings.

The Lehmer coefficients are calculated from $\boldsymbol{\mathcal{F}}'^b$ as $L_j = \sum_{i=0}^{j} (\mathcal{F}_{j+1}' > \mathcal{F}_i')$, with $(x > y) = 1$ if true and 0 if not. The Lehmer encoding has the nice property that two neighboring values swapping places only changes a single coefficient by ± 1. Using a binary Gray encoding for the Lehmer coefficients, this translates to only a single bit difference in the resulting output. The length of the binary representation becomes $\ell' = \sum_{i=2}^{b} \lceil \log_2 i \rceil$, close to the theoretical optimum $H_1(Y^{\ell'})$.

[125] K. Wong and S. Chen, "The Entropy of Ordered Sequences and Order Statistics" (1990).

Consider the following example of Lehmer encoding and decoding. Assume an array X with a permutation of the first four letters in the alphabet:

$$X = [B, C, D, A] .$$

In order to generate an encoding for this permutation, we start by finding how many elements in X are "smaller" than the last one. Using the formula given above, we get

$$L_0 = \sum_{i=0}^{0} (X[1] > X[i]) = 1 \qquad L_1 = \sum_{i=0}^{1} (X[2] > X[i]) = 1 + 1 = 2$$

$$L_2 = \sum_{i=0}^{2} (X[3] > X[i]) = 0 + 0 + 0 = 0 .$$

Thus, the encoding is $L = [1, 2, 0]$. Note that at no point did we have to do any sorting on the array.

In order to reconstruct the permutation from $Y = [A, B, C, D]$, we work from right to left. First, one takes from Y the element at position L_2 and inserts it on the left of the result array, i.e. $X' = [Y[L_2]] = [A]$. Next, this element is removed from Y. This process is then repeated, i.e. in the next step $X' = [Y[L_1], A] = [D, A]$ and $Y = [B, C]$. After all elements in L have been processed, one element will remain in Y, which is then inserted again on the left of X'. Thus, the result in this case is $X' = [B, C, D, A]$, which is indeed equal to X.

Entropy compression Ideally, we want each bit in the order encoding to have full entropy. The entropy density of the order encoding is already quite high:

$$\rho_1\left(Y^{\ell'}\right) = \sum_{i=2}^{b} \frac{\log_2 i}{\lceil \log_2 i \rceil} .$$

It can be increased further by compressing it to Y^{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.

It is easy to see this in the example presented above. The value L_1 can take on either 0 or 1, and can thus be optimally encoded with one bit. However, $L_1 \in \{0, 1, 2\}$ and thus two bits are required to encode this value, even though, in the ideal case, the entropy content is only $H_1(L_1) = -\log_2(\frac{1}{3}) \approx 1.58$ bits, because L_1 will never encode the value 3.

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 $\frac{\ell'}{\ell}$.[75]

In the example above, if one compresses L_1 to a single bit, then this bit $L_{1'}$ will be 0 one third of the time, and 1 the remaining two thirds. Thus, the entropy $H_1(L_{1'})$ has been increased to 0.92 bits of entropy, whereas L_1 only has 0.79 bits of entropy per bit.

The bound on the increase in bit error probability can be proven as follows. Take a string A of length n bits, which are XORed together to create a string B of length 1 bit. We will assume that each bit in A has the same error probability p_{err}. An error will manifest itself in B whenever there is an odd number of errors in A, otherwise the errors will cancel out due to the XOR operation. The error rate of B is thus

$$p_B = \sum_{\substack{i=1 \\ 2 \nmid i}}^{n} \binom{n}{i} \cdot p_{\text{e}}^{i} \cdot (1 - p_{\text{e}})^{(n-i)} .$$

Dividing this by the original error rate p_{e} of an individual bit, we get:

$$\begin{aligned}
\frac{p_B}{p_{\text{e}}} &= \frac{1}{p_{\text{e}}} \cdot \sum_{\substack{i=1 \\ 2 \nmid i}}^{n} \binom{n}{i} \cdot p_{\text{e}}^{i} \cdot (1 - p_{\text{e}})^{(n-i)} \\
&\leq \frac{1}{p_{\text{e}}} \cdot \left(\sum_{i=0}^{n} \binom{n}{i} \cdot p_{\text{e}}^{i} \cdot (1 - p_{\text{e}})^{(n-i)} - \binom{n}{0} \cdot p_{\text{e}}^{0} \cdot (1 - p_{\text{e}})^{n} \right) \\
&\leq \frac{1}{p_{\text{e}}} \cdot (1 - (1 - p_{\text{e}})^{n}) \\
&\leq n .
\end{aligned}$$

To prove this last step, consider the function on the third line:

$$\begin{aligned}
&\frac{1}{p_{\text{e}}} \cdot (1 - (1 - p_{\text{e}})^{n}) \leq n \\
\Leftrightarrow\ &1 - n \cdot p_{\text{e}} \leq (1 - p_{\text{e}})^{n} .
\end{aligned}$$

[75] R. Maes, "Physically Unclonable Functions: Constructions, Properties and Applications" (2012).

This is easily seen to hold for $n = 1$. We can prove the general case by induction:

$$\begin{aligned}(1-p_e)^{n+1} &= (1-p_e)^n \cdot (1-p_e)\\ &\geq (1-n\cdot p_e)\cdot(1-p_e)\\ &= 1-(n+1)\cdot p_e + n\cdot p_e^2\\ &\geq 1-(n+1)\cdot p_e\,.\end{aligned}$$

3.3.2 Helper data generation and error decoding

We consecutively describe our design of the helper data generator and error decoder blocks for the used repetition code, as well as the helper data generation for the BCH code. The design of the BCH decoder block is described in greater detail in Section 3.4.

3.3.2.1 Repetition code C_{REP}

As shown in Section 3.2.1, the generator polynomial of a repetition code of length n is $\mathcal{G} = \sum_{i=0}^{n-1} x^i$. That means that in hardware, one only has to XOR $Y_{\text{bit }1}$ together with each of the other bits $Y_{\text{bit }2}, \ldots, Y_{\text{bit }n_{REP}}$, i.e. $h_i = Y_{\text{bit }1} \oplus Y_{\text{bit }i+1}$.

The design of the REP decoder, shown in Figure 3.3, is very straightforward. Our block takes for a $C_{REP}(n,1,t)$ code n bits of PUF data Y'' and $(n-1)$ bits of helper data h_{rep}. A total of $2\cdot(n-1)$ XOR gates calculate the reproduction data $s'' = (Y'' \bmod \mathcal{G}) \oplus h_{\text{rep}}$ from these inputs.

Remember that the value for the first error bit is $e_1 = (\text{HW}(s'') > t)$. The remaining error bits are obtained by a pairwise XOR of e_1 with the other bits of s''. However, as pointed out in Section 2.7, we have to assume that $n-1$ bits of entropy are lost because the helper data is public. Thus, we only output the corrected bit Y'_1. In order to generate the helper data h_{rep}, an all-zero vector is input as the h_{rep} value, s'' is then equal to h_{rep}.

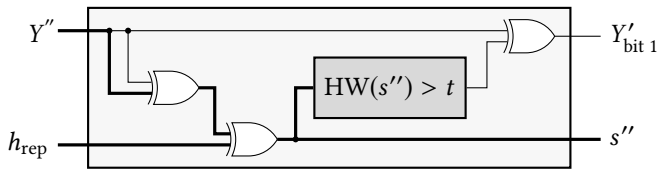


Figure 3.3: Design of our REP decoder with single bit corrected output.

3.3.2.2 BCH code C_{BCH}

Just as for the repetition code, the BCH code requires us to calculate a modular division of an input with the code's generator polynomial. However, unlike the repetition code, not all of the generator polynomial's coefficients are one at each position. One can efficiently implement the required modular division operation in hardware as an LFSR of length $(n_{BCH} - k_{BCH})$, with feedback taps determined by the generator polynomial. Thus, the size of this block, for $C_{BCH}(n, k, t)$, is equal to $(n - k)$ registers and a number of XOR gates equal to the sum of the generator polynomial's coefficients.

Calculating the reproduction data $s' = Y' \oplus h_{\text{bch}}$ is done by first sequentially clocking in each bit of Y' in the LFSR, and then XOR-ing this with the helper data h_{bch}. Generating helper data is done the same way as in the repetition secure sketch block, by simply taking the output of the LFSR XORed with zero.

3.4 BCH decoding microcontroller

In the following sections, we will take a closer look at the design of the microcontroller used to correct the PUF responses. Particular attention is paid to the various design techniques used to reduce the controller's footprint.

In order to better understand the design choices we make, one must look at the design requirements. Our application requires very long BCH codes. For example, as discussed in Section 3.5, the default PUFKY configuration (128-bit key with 10^{-9} failure rate) requires a BCH(318, 174, 17) code. The optimal BCH parameters depend on many factors, amongst them the PUF's error rate, its response length and required key length. Thus, our design has to be flexible and adaptable to any set of BCH code parameters. Another peculiarity is the fact that execution time is not of major importance in our design, since key generation is typically not a time-critical operation. Our primary goal is area reduction, in order to make the PUFKY design as small as possible.

Most, if not all, of these design goals are in stark contrast to what is generally required of BCH decoders. A review of published BCH and closely related Reed-Solomon decoder designs[72,99,100,103,108] shows that they are mostly constructed as systolic

[72] W. Liu *et al.*, "Low-Power High-Throughput BCH Error Correction VLSI Design for Multi-Level Cell NAND Flash Memories" (2006).
[99] J.-I. Park *et al.*, "An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders" (2011).
[100] J.-I. Park *et al.*, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm" (2009).
[103] I. Reed and M. Shih, "VLSI Design of Inverse-Free Berlekamp-Massey Algorithm" (1991).
[108] D. Sarwate and N. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders" (2001).

array designs, which, roughly speaking, comes down to creating a large calculation array out of identical building blocks.

Such designs are inherently highly pipelined, which leads to a very high throughput. However, such designs also require a large area. Furthermore, published designs are almost without exception targeted towards communication systems, in which different code parameters are used than the ones required for PUFKY in various configurations. In particular, the number of correctable bits for the PUFKY design needs to be larger than in conventional designs, which can have a rather big impact on decoding time and storage requirements.

Due to our primary design goals being a small area footprint and a high flexibility with regards to code parameters, the best choice in this case is to forgo the de facto systolic array design and opt for a microcontroller-based approach instead.

3.4.1 Hardware design

Our design is heavily influenced by the requirements of the BCH decoding algorithms, as explained in Section 3.2.2, and hence our microcontroller is very application-specific. In this section we describe the individual components of the microcontroller, as well as the communication interface.

3.4.1.1 Components

The BCH decoding controller design consists of three main components, which are shown in Figure 3.4. Each of them is described separately in the next paragraphs. The critical path of the controller is highlighted in orange in the figure. This path originates in the ROM, goes through the control unit, address RAM, data RAM, and $\mathbb{F}_{2^u}$ ALU, to finally end back in the control unit.

Data block The data block contains, first, a data RAM block that stores all variables necessary for the decoding as well as the error word, and, second, an arithmetic unit (ALU). The size, as well as the width, of the data RAM depends on the parameters of the BCH code. Because calculations done on the RAM content only return a single result, the dual port RAM only has one data input port. At the start of execution, the data RAM should contain reconstruction data s', after execution has finished, it contains the error word e.

As will be shown further on, virtually all calculations for BCH decoding are over elements in $\mathbb{F}_{2^u}$. This lead us to develop a minimal ALU supporting only a single complex operation: single-cycle multiply-accumulate in $\mathbb{F}_{2^u}$. The ALU contains a

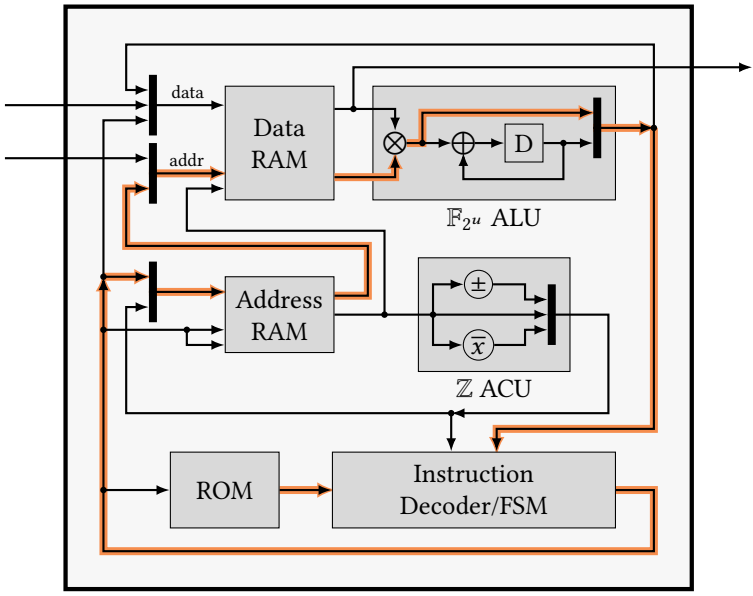


Figure 3.4: High-level architecture of the BCH decoding microcontroller. The highlighted line indicates the critical path.

single register for the accumulator and has a dual port input from the RAM, both of the ports are fed into the multiplier. The operation is implemented in a flexible way, such that one can chose between executing either a multiply-accumulate, a multiplication or an addition.

Not shown in Figure 3.4, but also present in the $\mathbb{F}_{2^u}$ ALU, is support for fixed length rotation and shifting, however this only requires wires, no active circuitry. Naturally, some extra multiplexer are needed in order to select between the outputs of these various operations.

Address block Part of the novelty of our design is the use of an extremely tiny, dedicated address block. This block consists of a tiny address RAM and an attached address calculation unit (ACU), reminiscent of digital signal processor (DSP) designs.[70]

The size of the address RAM is only 5 words, independent of the BCH code parameters, while its width does depend on them, since larger codes require more data RAM, and thus a larger address space. The reason for including a dedicated address block is due to the high amount of array processing required by the algorithms.

[70] P. Lapsley *et al.*, "DSP Processor Fundamentals: Architectures and Features" (1996).

The attached ACU (see Figure 3.4) works over elements in $\mathbb{Z}$ and supports increase and decrease by one, passthrough and binary inversion. By inputting a destination address on the first address port and a source address on the second one, the passthrough can be used to copy values to different locations in RAM. This is the reason why the RAM block has a dual port address input and a dual port value output, yet only a single data input. The binary inversion operation is equal to negate and decrease by one in two's complement notation, an operation required by our Berlekamp-Massey algorithm. The combination of these operations allow the use of the address block for address pointer storage, for array pointer arithmetic and for keeping track of counter values.

Controller The controller consists of a firmware ROM, which contains the machine code for BCH decoding. The firmware is fixed, and generated at synthesis time. Since our controller contains no pipelines, the design used to decode the firmware and control the datapath can be implemented as a single finite state machine.

Because the size of the address RAM is always five words, including the source and/or destination addresses of an operation into an instruction word takes only three bits. This allows for much smaller machine code than if the full addresses would have to be included with each instruction.

3.4.1.2 Communication

Both the address RAM output and the data block ALU register output are wired back to the controller in order to enable comparisons. Write enable signals to both RAMs as well as to the data block ALU register can be blocked by the controller. This allows predicated execution for almost every instruction, i.e. an instruction can be executed completely, except its result will not be stored, so it effectively acts as a `nop` operation.

The controller reset line is not only coupled to the controller block, but also to multiplexers in front of the data RAM. By keeping the controller in reset, one can write data, i.e. the reproduction data s', to the data RAM address and data inputs. As soon as reset is released, the controller starts executing the firmware, i.e. decoding. The last instruction in the firmware tells the controller to assert a ready output and halt. The error word can then be retrieved by asserting the reset line again and reading out the result from the data RAM. This type of design allows the controller to be easily integrated in larger designs with little to no overhead, since there is no communication protocol one has to implement in order to interface with the controller.

Analysis of the required algorithms for BCH decoding shows that the majority of operations takes place in loops over arrays. This is the reason the controller contains a separate address block. The output of the address RAM is wired straight into the data RAM address input. Thus, all addressing in the controller is indirect. However,

since virtually all loops operate over arrays, this construction allows for very intuitive and efficient firmware development. For example, implementing an array summation takes only three assembly instructions: accumulate, increase address pointer and conditional branch, note that at no point explicit loads or stores are necessary.

All RAMs are non-buffered, meaning there are no cycles between querying an address and the value at that address appearing at the output. The drawback of this design choice is a reduced maximum frequency, since the critical path runs through both RAMs. If buffered RAM were used instead, the critical path would most likely be one third of what it currently is. Such an improvement would require a much more advanced controller though. First of all, assuming approximately one instruction is handled per cycle, an instruction pipeline would be required. Furthermore, due to predicated execution, our controller would then need to be able to flush the pipeline, as well as insert stall/NOP states. We have therefore opted not to implement this, since execution time is rather low on the list of design goals, and the complex controller would take up a larger area.

3.4.2 Software design

In this next section, we describe the software aspects of our BCH decoder. First of all, we list the microcontroller instruction set architecture (ISA), next we go over the algorithm implementations and finally we describe the optimization techniques used during firmware implementation.

3.4.2.1 Instruction set

An overview of the instruction set architecture of the microcontroller is given in Table 3.2. The instructions have a minimum width of 10 bits; the actual required width depends on the parameters of the chosen BCH code and the firmware. If the size u of the underlying field $\mathbb{F}_{2^u}$ is larger than 10, the instruction width needs to increase in order to be able to initialize RAM to any value, unless all initialization values have their most significant bit equal to zero. For the PUFKY designs we tested, the field size was always 9 or less, and thus the instruction width is fixed at 10 bits. However, increasing this would require no more than modifying a constant in our HDL code. Encoded in an instruction word are the type of operation to execute, predicated execution flags and (if applicate) source and/or destination address pointers.

The instructions are chosen specifically in function of the algorithms required for BCH decoding. The most important considerations in choosing an instruction to implement are *i*) the size of the hardware required for it; *ii*) whether adding it will allow for

significant speed-up of the execution time; and *iii*) whether it can be used for multiple algorithms. For an architecture as highly optimized for one function as this one, instruction selection is a trail-and-error process that goes hand in hand with firmware development. With the 10-bit instruction width of our example implementation, every single combination of bits is reserved for some function, i.e. not a single extra instruction can be added without increasing the instruction width.

One good example of the instruction selection process is the `gf2_add_mult` instruction combined with predicated execution. Initially, the processor had two separate instructions for $\mathbb{F}_{2^u}$ operations: `gf2_add` and `gf2_mult`. Each of the three algorithms requires at least one of these operations, thus, combining them was a logical choice, since the hardware was already available. Furthermore, replacing two functions by one frees up a machine word which can then be used for another instruction. It also speeds up those instances where a multiply-accumulate operation is required, e.g. the Berlekamp-Massey algorithm (Algorithm 3.2). The problem is now how to select whether an addition, a multiplication or a multiply-accumulate is required. By adding support for predicated execution, not only can the `gf2_add_mult` instruction execute those three operations, other instructions benefit from it as well, e.g. branches can be removed in favor of non-executed instructions, thereby reducing execution time. Naturally, adding predicated execution comes at a price, in this case an extra 2 bits in the instruction word and a few extra gates in the instruction decoder.

Another example of a highly optimized instruction is `rotr`, which not only rotates a word to the right, but can also set a predicated execution flag to the value of the LSB of the rotated word. This specialized instruction allows the implementation of the inner loop of the *syndrome calculation* algorithm (Algorithm 3.1) with just two instructions: `rotr` on $R[j]$ followed by a conditional `gf2_add_mult` of $S[i]$ and evalarg. Another example is the `shiftl` instruction, which has two variants, one which sets the newly shifted-in LSB to zero (`shiftl_clr`), another which sets it to one (`shiftl_set`). This instruction is used in the *error location calculation* algorithm (Algorithm 3.3) during reconstruction of the error vector.

It is possible to increase the execution speed of some of the following algorithms by including a squaring instruction, in particular Algorithms 3.1 and 3.3. However, doing so would greatly increase the size of the $\mathbb{F}_{2^u}$ ALU. Thus, since low area is a more important design goal than fast execution, this is an example of an instruction we choose not to implement.

Jump instructions encode the jump length relative to the current instruction address. Due to the machine code representation of instructions, the jump offset cannot take up more than 7 bits, and thus no more than 127 instructions at a time can be jumped over. Should longer jumps be required, then this can be accomplished by chaining jumps, i.e. jumping to another jump instruction.

Comparisons can have as their left hand side the content of either any of the five address RAM locations, or the $\mathbb{F}_{2^u}$ ALU register. The microcontroller can check for $\neq, >, or \leq$ with an 8-bit fixed value encoded into the instruction.

Table 3.2: Instruction set architecture of the microcontroller.

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`	$\text{data}[\text{addr}_i] \leftarrow \text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j]$ reg $\leftarrow$ reg $\oplus$ ($\text{data}[\text{addr}_i] \otimes \text{data}[\text{addr}_j]$)	1

3.4.2.2 Algorithm implementation

Next, we briefly discuss our implementations of the algorithms for BCH decoding. The firmware was manually written and optimized in a custom assembly language and processed with a machine code translator specifically written for our microcontroller. As stated before, a total of three algorithms are required. Before any calculations are done, an outside source should load reproduction data s' into the microcontroller's data RAM, as discussed in Section 3.4.1.2.

Syndrome calculation The first algorithm in the decoding process is the *syndrome calculation* algorithm, a listing of which can be found in Algorithm 3.1. As discussed

in Section 3.2.2.1, this algorithm evaluates the received codeword as a polynomial for successive powers of a primitive element $\alpha^i, i = 1, \ldots, 2t$. The outer loop starting at line number 2 iterates over every power of α. Due to the specialized instruction set, the inner loop at lines 7 to 10 can be implemented with just three assembly instructions: shift right, conditional multiply-accumulate and conditional jump.

Algorithm 3.1: Syndrome calculation.

Input: $R[n] \in \mathbb{F}_2$
Output: $S[2t] \in \mathbb{F}_{2^u}$
Data: curArg, evalArg $\in \mathbb{F}_{2^u}$; $i, j \in \mathbb{N}$

```
1  curArg ← α
2  for i ← 0 to 2t − 1 do                          /* Loop over powers of α. */
3      S[i] ← 0
4      evalArg ← 1
5      if R[0] = 1 then                            /* Check whether LSB of R is set. */
6          S[i] ← S[i] ⊕ evalArg                   /* Add 1 (= α^0) to syndrome element. */
7      for j ← 1 to n − 1 do                       /* Loop over each term of the R polynomial. */
8          evalArg ← evalArg ⊗ curArg              /* Multiply evalArg by α^(i+1). */
9          if R[j] = 1 then
10             S[i] ← S[i] ⊕ evalArg               /* Add x (= α^((i+1)·j)) to syndrome element. */
11     curArg ← curArg ⊗ α                         /* Set curArg to α^(i+2). */
```

Berlekamp-Massey algorithm Once *syndrome calculation* is finished, an inversionless form of the *Berlekamp-Massey* algorithm is run. Inversionless means that, unlike the first published version of this algorithm, no time-consuming inversions over $\mathbb{F}_{2^u}$ are required. This algorithm calculates the error location polynomial, which is discussed in Section 3.2.2.2. The listing for the algorithm can be found in Algorithm 3.2. Due to the relatively large amount of arithmetic operations and loops, this is the slowest of the three required algorithms.

At no time can the order of the polynomial Λ be larger than i or t, i.e. $\mathrm{ord}(\Lambda) = \min(i, t)$. Therefore the total number of times that line 12 gets executed, can be improved to $(3t^2 + t)/2$, compared to $2(t^2 + t)$ which we found in the literature.

The variable k requires integer arithmetic, thus we store it in the address RAM, since the rest of the algorithm only requires four address pointers. The operation on line 19 can be calculated by bitwise inversion if k is treated as a two's complement number.

Error location calculation Finally, the error positions are calculated using a polynomial evaluation algorithm based on the Chien search method, as discussed

Algorithm 3.2: Inversionless Berlekamp-Massey algorithm, based on work by Burton.[25]

Input: $S[2t] \in \mathbb{F}_{2^u}$
Output: $\Lambda[t+1] \in \mathbb{F}_{2^u}$
Data: $\mathrm{b}[t+2], \delta, \gamma \in \mathbb{F}_{2^u}; \mathrm{flag} \in \mathbb{F}_2; \mathrm{k} \in \mathbb{Z}; i, j \in \mathbb{N}$

```
1  b[−1] ← 0                          /* Initialize first element of b. Starts at index −1! */
2  b[0] ← 1
3  Λ[0] ← 1
4  for i ← 1 to t do                                  /* Initialize b and Λ polynomials. */
5      b[i] ← 0
6      Λ[i] ← 0
7  γ ← 1
8  k ← 0
9  for i ← 0 to 2t − 1 do                             /* Loop over every element of S. */
10     δ ← 0
11     for j ← 0 to min(i, t) do                      /* Calculate discrepancy due to Λ coeff. */
12         δ ← δ ⊕ (S[i − j] ⊗ Λ[j])
13     flag ← (δ ≠ 0) & (k ≥ 0)                       /* Check whether b and γ need to be updated. */
14     if flag = 1 then
15         for j ← t to 0 do
16             b[j] ← Λ[j]          /* Store Λ coeff. which generated current discrepancy. */
17             Λ[j] ← (Λ[j] ⊗ γ) ⊕ (b[j − 1] ⊗ δ)          /* Apply correction to fix Λ. */
18         γ ← δ                                  /* Store value of current discrepancy. */
19         k ← −k − 1
20     else
21         for j ← t to 0 do
22             b[j] ← b[j − 1]                                  /* Shift coeff. of b. */
23             Λ[j] ← (Λ[j] ⊗ γ) ⊕ (b[j − 1] ⊗ δ)          /* Apply correction to fix Λ. */
24         k ← k + 1
```

in Section 3.2.2.3. A listing of this algorithm is given in Algorithm 3.3. The error location polynomial evaluates to zero for α^{-l_1}, if there is an error at location l_1. Thus, normally one would use powers of α^{-1} in the Chien search algorithm. However, due to Fermat's little theorem, the following holds:

$$\begin{aligned}\alpha^i &= \alpha^{n-(n-i)} \\ &= \alpha^{-(n-i)}.\end{aligned}$$

Therefore, we can evaluate the error location polynomial for positive powers of α and if an error location is found at step i set an error bit at location $n-i$. This is the reason the loop on line 1 in the algorithm listing runs backwards, otherwise the error bit would be stored at the wrong location.

[25] H. Burton, "Inversionless Decoding of Binary BCH codes" (1971).

In the firmware the error location vector is created by left shifting an array and setting the newly inserted bit to either one or zero, using one of two specially implemented instructions. Due to the combined multiply-accumulate and predicated execution, which can be turned on or off per assembly instruction, the inner loop at lines 4–7 can be implemented very efficiently: one unrolled loop requires only three assembly instructions.

Algorithm 3.3: Error location calculation algorithm, based on work by Chien.[28]

Input: $\Lambda[t+1] \in \mathbb{F}_{2^u}$
Output: errorLoc$[n] \in \mathbb{F}_2$
Data: curAlpha, curEval $\in \mathbb{F}_{2^u}$; $i, j \in \mathbb{N}$

```
1  for i ← n − 1 to 0 do                    /* Run backwards so error locations are correct. */
2      curEval ← Λ[0]
3      curAlpha ← α
4      for j ← 1 to t do                          /* Iterate over each coeff. of Λ. */
5          Λ[j] ← Λ[j] ⊗ curAlpha                          /* Update Λ[j] coeff. */
6          curEval ← curEval ⊕ Λ[j]            /* Add Λ[j] · α^{j·(n−i)} to result. */
7          curAlpha ← curAlpha ⊗ α                  /* Set curAlpha to α^{j+1}. */
8      if curEval = 0 then                  /* If a root is found, set error bit. */
9          errorLoc[i] ← 1
10     else
11         errorLoc[i] ← 0
```

3.4.2.3 Software and Hardware Optimization Techniques

Although low execution time is only the third of our design goals, it is the only one which can significantly be impacted by software design. Thus, we focus on fast execution speed as much as possible during programming of the firmware. As a bonus, faster running firmware will decrease total energy consumption.

In these next paragraphs, we give an overview of the optimization techniques used. Many of these are software-based, yet a few can be seen as hardware optimizations. Due to the tight coupling between the microcontroller's hardware design and the firmware, it is difficult to separate these two types of optimizations and thus we present them here together.

The effect of various optimization steps on $C_{BCH}(318, 174, 17)$, used in the reference PUFKY implementation, is shown in Figure 3.5.

[28] R. Chien, "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes" (1964).

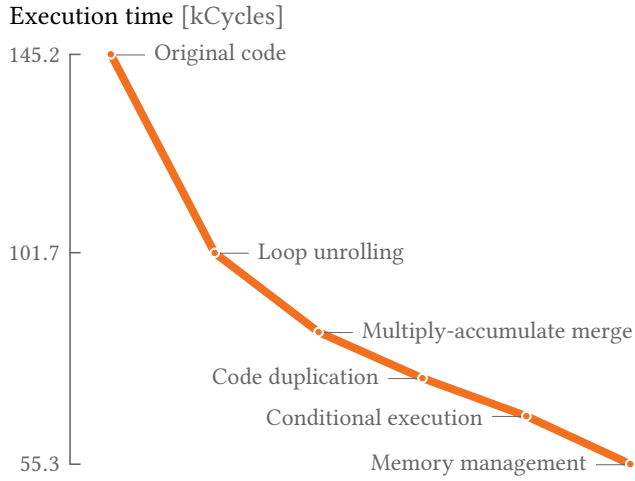


Figure 3.5: Effect of optimizations on $C_{BCH}(318, 174, 17)$ decoding runtime.

Our initial software design requires approximately 145 000 cycles to decode one 318-bit codeword. First we apply loop unrolling to the inner loops of all three algorithms. For the chosen BCH parameters, the loops can be unrolled in such a way that no extra pre- and post-loop code needs to be added. Loop unrolling significantly improves runtime, due to the elimination of a large fraction of conditional branch operations which are, at three cycles, the slowest instruction in our microcontroller. Naturally, this comes at the cost some extra ROM. This first optimization improves runtime by more than 30%: one decoding now requires approximately 100 000 cycles.

The next optimization is the merge of the multiply and accumulate instruction, which has an impact on both hard- and software. On the hardware side, modifications to both datapath and controller are required. This leads to improvements on both runtime and code size. All three algorithms benefit from this instruction merge, more specifically Algorithm 3.1 lines 8–10, Algorithm 3.2 lines 12, 17 and 19, and Algorithm 3.3 lines 5–6. This optimization decreases total runtime to approximately 84 000 cycles, an improvement of 16%.

In order to eliminate even more conditional branches, the second inner loop in the *Berlekamp-Massey* algorithm is duplicated, once for each possible value of the `flag` variable. A simple optimization which leads to another 12% runtime decrease, at the cost of a few extra words of ROM.

Reducing the number of conditional branches even more is done by introducing predicated execution. All instructions can be programmed to either always execute, execute when the condition flag is false or when it is true. For the multiply-accumulate

instruction, two condition flags are used, which makes it possible to use the instruction as a multiplication or addition if required. The introduction of this feature has a large impact on the inner loop of the *syndrome calculation* algorithm, which can now be implemented with only two instructions: rotate right and set condition bit, and conditional multiply-accumulate. After this optimization, the total runtime is reduced to approximately 66 000 cycles, another 11% decrease.

Finally, by improving the memory management of some variables, i.e. moving them to more appropriate locations, we are able to reduce the runtime by another 17%, to 55 000 cycles. The memory layout, together with size requirements dependent on code parameters, is shown in Figure 3.6.

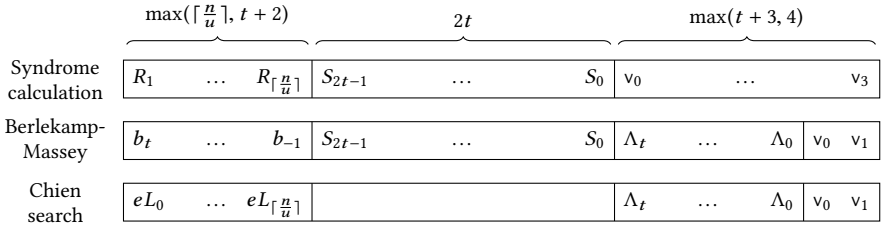


Figure 3.6: Memory layout for each algorithm in the BCH decoding process.

The combination of all these optimizations reduces the total runtime by 62%, compared to the original implementation, from 145 000 to 55 000 cycles. Obviously this not only benefits runtime, but energy requirements as well.

3.4.2.4 Framework for Flexible Code Generation

Our design is implemented completely in the HDL language Verilog. Parts of the Verilog code can be dynamically generated by a framework we designed. The framework takes the three BCH code parameters as input, verifies that such a parameter combination is valid, generates appropriately sized RAMs and calculates parameters such as datapath and address bus widths. Next an assembly listing for the firmware is created and from this a custom ROM design in Verilog is generated. Due to this tool, only a minimum amount of manual modification to the firmware code is required, in the worst case, and the effect of many BCH code parameters on different design metrics can be quickly evaluated.

3.4.3 Implementation

In the following paragraphs, we give an overview of our implementation results for the BCH decoding microcontroller. First we give results for FPGA and ASIC synthesis of the hardware. Next, the impact of BCH code parameters on runtime is shown.

3.4.3.1 Synthesis

ASIC Synthesis for ASIC was done with Synopsys Design Compiler D-2010.03-SP4, using the UMC's 130 nm *fsc0l_d_sc_tc* low leakage library. Table 3.3 shows that the controller core requires only 1 130 GE. This excludes the two dual-port RAMs (for data and address storage) and the ROM containing the firmware. The amount of storage required is listed in the same table. Finally, we also give the total area required including the size of RAMs and ROM, synthesized for the same UMC technology with the Faraday Technology memory compiler.

It is clear that the chosen set of code parameters have no influence on the size of the core. The area required by the RAM mostly depends on the error correcting capability, i.e. t, of the code, which can also be seen in Figure 3.6. For the ROM, the change in size mostly depends on what the most efficient loop unrolling stride is for the chosen code parameters.

Table 3.3: Synthesis results for ASIC implementation using UMC's 130 nm *fsc0l_d_sc_tc* library. "Area [GE]" does not include RAM/ROM area, "Total area [μm^2]" does.

BCH(n, k, t)	Area [GE]	F_{max} [MHz]	RAMs [bit]	ROM [bit]	Total Area [μm^2]
(413, 296, 13)	1130	119.5	88×9 5×7	249×10	56×10^3
(380, 308, 8)	1130	119.9	72×9 5×7	237×10	55×10^3
(318, 174, 17)	1130	119.3	96×9 5×7	246×10	57×10^3

FPGA Synthesis for FPGA was done for the Xilinx Virtex-6 family using Xilinx ISE 12.2 M.63c with design strategy "Area reduction with Physical synthesis." Table 3.4 shows that the total size of our design is very small: it requires less than 1% of the size of the smallest Virtex-6 FPGA, the 11 640-slice XC6VLX75T.

Different BCH code parameters have little influence on the design size. No separate BlockRAMs are used, since our design uses RAM and ROM blocks which are implemented within LUTs. Thus, the listed slice count is the total size that the design requires.

Table 3.4: Synthesis results for implementation on a Xilinx Virtex-6 FPGA.

BCH(n, k, t)	**Area**			**F_{max}**
	[slice]	**[FF]**	**[LUT]**	**[MHz]**
(413, 296, 13)	65	33	244	94.4
(380, 308, 8)	66	33	244	97.8
(318, 174, 17)	68	33	251	93.6

3.4.3.2 Runtime

Although the size of the hardware is not significantly influenced by the BCH code parameters, the runtime of our firmware is. Table 3.5 clearly shows the large effect of the parameter t, the number of correctable errors, on the total runtime. The same table also gives formulas for *ideal* runtime. We define this as: the number of cycles required if each loop iteration executes in a single cycle, without executing multiple loop iterations in parallel; e.g. a loop of x iterations would require x cycles.

Table 3.5: High-order approximations for runtime of algorithms in the BCH decoding process. *Ideal* assumes single cycle inner loops.

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

A comparison between *ideal* and actual, high-order runtime formulas show that our software implementation is very efficient. For example, the *syndrome calculation* and *error location* algorithms requires only 2–4 times more cycles than in the *ideal* case, assuming a realistic value for u, i.e. $8 \leq u \leq 10$. This includes delays introduced by setup code for variables and conditional branches, which are non-existent in the *ideal* case. The Berlekamp-Massey implementation requires an approximately ten times

longer runtime than the *ideal* case, which is very efficient as well, considering the amount of arithmetic operations which have to be computed in the inner loops.

Table 3.6 lists the runtime of our controllers for some example BCH codes. Once more, it is clear that the t parameter has the largest influence on total runtime.

Table 3.6: Total runtime for BCH decoding.

BCH(n,k,t)	**Runtime @ 90 MHz**	
	[cycles]	**[µs]**
(413, 296, 13)	55 379	615
(380, 308, 8)	26 165	291
(318, 174, 17)	50 320	559

3.4.3.3 Comparison with existing designs

Although we would like to compare our design with existing BCH decoders, this is extremely difficult, mostly because of the unique goals of our design. Whereas our primary goal is compactness, existing published results are aimed at communication systems and are designed with high throughput in mind.[27,72,99,100,103,108]

This problem is compounded by the type of code used. In our design we use a BCH code, since erasures are not a problem in our target application. Most published designs are for Reed-Solomon (RS) decoders, which are related, yet require slightly different algorithms. This leads to differences in both the arithmetic hardware and storage requirements of the decoders.

Finally, the target application of PUF-based key generation forces us to use BCH code parameters unlike any we have been able to find in the literature.

[27] H.-C. Chang and C. Shung, "New Serial Architecture for the Berlekamp-Massey Algorithm" (1999).
[72] W. Liu *et al.*, "Low-Power High-Throughput BCH Error Correction VLSI Design for Multi-Level Cell NAND Flash Memories" (2006).
[99] J.-I. Park *et al.*, "An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders" (2011).
[100] J.-I. Park *et al.*, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm" (2009).
[103] I. Reed and M. Shih, "VLSI Design of Inverse-Free Berlekamp-Massey Algorithm" (1991).
[108] D. Sarwate and N. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders" (2001).

The reason no other BCH decoders for PUF applications have been published, is probably that at least two papers[22,67] mention that such decoders would be large and complex. As our implementation results show, the opposite is true.

Despite these difficulties, Table 3.7 attempts to compare published designs with ours. In order to make the already difficult comparison as meaningful as possible, we have selected from our designs the one for which the BCH parameters most closely match those of the compared designs. Our implementation has smaller throughput than those designs, however, its area is between an order of magnitude two and three times smaller. Given the area of the other designs, as well as the fact that they consist of many simultaneously clocked registers, it is likely that the power consumption of our design is at least three to four orders of magnitude better. If the time-area product is the main optimization goal, then clearly our design is not the best, due to its very slow throughput compared to other designs. It is difficult to make meaningful statements about energy consumption, since this requires knowledge of both the throughput, which we know, and the power consumption, which we can only broadly estimate.

Table 3.7: Comparison of BCH decoder microcontroller with published designs. Area does not include RAM/ROM/FIFOs. See Section 3.4.3.3 for interpretation.

Design	Code	Throughput [Gbit/s]	Area [GE]	Time × area [s/Gbit · GE]
Park *et al.*[100]	RS(255, 239, 8)	5.6	43 600	7 786
Park *et al.*[99]	RS(255, 239, 8)	3.2	19 723	6 163
Liu *et al.*[72]	BCH(4148, 4096, 4)	0.4	25 000	62 500
Our design	BCH(380, 308, 8)	0.0013	1 130	869 231

3.5 Full generator implementation

We present the results for a reference implementation of PUFKY in this section. The aim of this implementation is to generate a 128-bit key, with a maximum failure rate

[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).

[67] P. Koeberl *et al.*, "Entropy Loss in PUF-based Key Generation Schemes: The Repetition Code Pitfall" (2014).

[72] W. Liu *et al.*, "Low-Power High-Throughput BCH Error Correction VLSI Design for Multi-Level Cell NAND Flash Memories" (2006).

[99] J.-I. Park *et al.*, "An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders" (2011).

[100] J.-I. Park *et al.*, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm" (2009).

$p_{\text{fail}} \leq 10^{-9}$. Such a failure rate would be adequate for for military-grade devices, whereas a failure rate of 10^{-6} could be considered sufficient for consumer-grade devices.

In order to determine the required code parameters and number of ring oscillators in the PUF, the ROPUF response was measured on 10 identical FPGAs at various temperature conditions (10 °C, 25 °C, and 80 °C), using a design with $b = 16$ batches of $a = 64$ oscillators each (see Section 3.3.1). Such a design generates $\sum_{i=2}^{b} \lceil \log_2 i \rceil = 49$ bits of output at a time. Post-processing of these results show a maximum expected error rate of 13% and an inter-device entropy density $\rho = 97.95\%$ after compressing the output down to $l = 42$ bits, taking into account a safety margin of 2% for the error rate.

An exhaustive search of possible code parameters and ROPUF size meeting our reference goals, and taking into account the above ROPUF measurement results, returns $b = 16$ batches of $a = 53$ oscillators for the ROPUF, $C_{REP}(7,1,3)$, and $C_{BCH}(318,174,17)$. For every 42 bits output by the ROPUF, $|h_{\text{rep}}| = 36$ bits of helper data are generated, leaving 6 bits to be forwarded to the BCH decoding block. The BCH decoder requires $|h_{\text{bch}}| = 144$ bits of helper data, and supplies the SPONGENT-128 entropy accumulator with 318 bits of corrected response data, which compresses this to a 128-bit key.

The ROPUF generates $a \times l = 2226$ bits in total, which contain $a \times l \times \rho = 2180.40$ bits of inter-device entropy. Due to the helper data being public, $a \cdot |h_{\text{rep}}| + |h_{\text{bch}}| = 53 \cdot 36 + 144 = 2052$ of those bits need to be considered lost, which leaves a total of $2180.40 - 2082 = 128.40$ bits of entropy, just enough to meet our goal.

Our PUFKY reference implementation requires 1162 slices on a Xilinx Spartan-6 FPGA, of which 82% is taken up by the ROPUF. Table 3.8a lists the size of each submodule used in the design. The total time required to generate the 128-bit key is approximately 5.62 ms (at 54 MHz). The ROPUF requires 82% of that time to generate output; the remainder of the time is taken up for 90% by the BCH decoding. Table 3.8b lists the number of cycles required for each step of the key generation.

3.6 Conclusion

In this chapter we discussed the design of a PUF-based key generation device. Generic design constraints were identified and used to construct a practical design. We presented a complete implementation of this design, based on a ring-oscillator PUF, a BCH decoding microcontroller and a hash function as entropy accumulator.

Using novel response encoding methods, our ROPUF can produce high-entropy responses (up to 99% with a sufficient amount of XOR compression). An in-depth

Table 3.8: Area requirements 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, it is determined by the time required for an internal free-running oscillator to complete a fixed number of oscillations.

(a) Area requirements

Module	**Size [slices]**
ROPUF	952
REP decoder	37
BCH syndrome calc.	72
BCH decoder	112
SPONGENT-128	22
helper data RAM	38
Total	1 162

(b) Runtimes

Step	**Time [cycles]**
PUF output	4.59 ms
REP decoding	0
BCH syndrome calc.	511
BCH decoding	50 320
SPONGENT hashing	3 990
control overhead	489
Total @ 54 MHz	5.62 ms

treatise showed which design strategies and techniques allow our BCH decoding microcontroller to be very efficient and flexible, yet fit in less than 1% of the area (< 70 slices) of even a small FPGA. The proposed BCH decoder design proves that the established perceived area overhead of BCH decoders for PUF error correction is grossly exaggerated. Finally, motivated by their wide-spread use in PRNG-based key generators, we use a hash function as entropy accumulator, which offers a considerable efficiency gain compared to the much more stringent design constraints for information-theoretically secure key extraction.

A reference implementation was presented which generates a 128-bit key in 5.62 ms and requires 1162 slices in a Xilinx Spartan-6 FPGA. Due to its completeness and efficiency, our PUFKY reference implementation is the first PUF-based key generator to be immediately deployable in an embedded system.

4 Analysis of SRAM in COTS Microcontrollers

Seeing, contrary to popular wisdom, isn't believing.
It's where belief stops, because it isn't needed any more.
— Terry Pratchet, Pyramids (1989)

Although a great many custom hardware designs for PUFs and RNGs have been published, it is not always economically feasible to use such designs in a system. We therefore focus our attention on solutions requiring only commercial off-the-shelf microcontrollers. More specifically, in this chapter we will present an overview of important metrics concerning PUFs and RNGs for embedded SRAM power-up content. We focus our efforts on the most important families of microcontrollers.

Our measurements and calculations show that many COTS microcontrollers have rather well-behaved SRAM, so to speak. However, there are exceptions, and one should thus not blindly assume that SRAM power-up values can always be used for the purpose of constructing PUFs and RNGs. Of the chips we tested especially the STMicroelectronics STM32 ARM Cortex-M microprocessors, more specifically the STM32F100R8, exhibits desirable characteristics. On the other hand, the Microchip PIC16 family, more specifically the PIC16F1825, is unsuitable for the applications we have in mind.

Content sources

A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers", in *International Workshop on Trustworthy Embedded Devices (TrustED)*, A.-R. Sadeghi, F. Armknecht, and J.-P. Seifert, Eds., ser. TrustED '13, Berlin, Germany: ACM, 2013, pp. 55–64
Contribution: Main author.

A. Van Herrewege, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-the-Shelf Microcontrollers", in *ACM Conference on Computer and Communications Security (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., Berlin, Germany: ACM, 2013, pp. 1333–1336
Contribution: Main author, responsible for PRNG section.

4.1 Introduction

Our goal is to make PUFs and strong RNGs economically viable by presenting designs that require no more than a commercial off-the-shelf microcontroller. This means designs that do not need any additional electronics or custom hardware apart from a microcontroller. The problem with the many custom hardware PUF and RNG designs presented over the years, is that for many devices they are not economically feasible. As evidenced by Section 1.1.4, there is a big market for cheap PUFs and RNGs. Thus, finding resources in COTS microcontrollers that allow us to create such designs is critical in improving the security of everyday electronic devices.

One microcontroller-contained resource that immediately comes to mind, and that has been shown repeatedly to be excellent for PUF constructions, is SRAM. Holcomb *et al.*[56] and Bösch *et al.*[22] both researched the use of SRAM power-up values as a PUF. Holcomb *et al.*[56] also present a design whereby entropy is extracted from the SRAM power-up values using a universal hash function. Although many other papers on SRAM PUFs have since then been published, virtually all of them require either custom hardware or external SRAM chips. Two recent exceptions to this are by Platonov *et al.*[102] and Böhm *et al.*;[21] who both describe the creation of

[21] C. Böhm *et al.*, "A Microcontroller SRAM-PUF" (2011).
[22] C. Bösch *et al.*, "Efficient Helper Data Key Extractor on FPGAs" (2008).
[56] D. E. Holcomb *et al.*, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags" (2007).
[102] M. Platonov *et al.*, "Using Power-up SRAM State of Atmel ATmega1284P Microcontrollers as Physical Unclonable Function for Key Generation and Chip Identification" (2013).

a PUF purely from software in respectively an Atmel ATmega1284P and an NXP LPC1768 microcontroller.

We focus our efforts on SRAM for a few reasons. First of all, it is to the best of our knowledge the only resource in COTS microcontrollers that can conceivably be used to construct a self-contained PUF. Furthermore, every microcontroller contains embedded SRAM, and thus we are not restricted in our choice of microcontrollers to specialized or expensive high-level types. Next, SRAM power-up values are, by their very nature, instantly available the moment the microcontroller powers-up. This in contrast to, e.g., oscillator-based systems, that need to be measured. And finally, there is a certain flexibility with SRAM. E.g. assume that due to application requirements one needs 8 KiB of SRAM in order to create an adequately strong 256-bit PUF-generated key. If the microcontroller for which one is developing does not have that much SRAM, then chances are very good that a very similar microcontroller from the same family will.

We qualify the SRAM power-up content using the metrics presented in Section 2.6. For PUFs, in particular the intra- and inter-device distances are of importance. For RNGs on the other hand, we are interested in the min-entropy calculated as explained in Section 2.8. Perhaps counter-intuitively, we do not want a too high fractional min-entropy, since that would mean that the related intra-device distance would be high as well, which would make error correction impossible for PUF-based designs.

In order to create an as encompassing overview as possible, we attempt to do this qualification for the most important microcontroller families. Deciding which families these are, is a difficult exercise, because of the myriad of different microcontroller families available, and, as a result, a very fractured market. We have based our selection on data from the Gartner Group, and filings by various microcontroller manufacturers to the U.S. Securities and Exchange Commission (SEC). This data is summarized in a recent presentation for investors by MicrochipMicrochip.[88] Our final selection includes microcontrollers from, what we believe to be, four big and important families. These are Atmel AVR, Microchip PIC16, Texas Instruments MSP430, and ARM Cortex-M (using STMicroelectronics STM32 ICs).

4.2 Measurement setup

Before presenting the results of our measurements, we first have to discuss the measurement setup used. Our setup uses a central controller board that sequentially power-cycles microcontrollers under measurement. Although this is a fairly straightforward setup, there are a few important requirements on both the controller

[88] Microchip, “Stephens Inc. Spring Investment Conference” (2013).

hardware and the firmware in the microcontrollers being measured. We will first present the central measurement controller design, and next discuss the few requirements on firmware in the microcontrollers. We then discuss some important aspects of the measured microcontrollers. Finally, we briefly explain the conditions under which measurements are taken.

4.2.1 Hardware setup

Our goal is to have a controller board that is as universal as possible, meaning it has to be able to assist in taking SRAM measurements from any COTS microcontroller. Measurements are to be stored on a computer, thus storage capacity is not an issue. There is however the issue of how measurement data finds its way from the microcontrollers to the computer. We choose to have the microcontroller interface transmit at the same rate as the controller to computer interface. This makes it possible to bypass the controller while a microcontroller is transmitting data, reducing the chance that bugs in the controller firmware influence measurements.

One problem with SRAM is that it it exhibits data remanence when powered off, i.e. the SRAM cells retain their last values. Furthermore, the lower the temperature, the worse this effect becomes, i.e. the longer data remains stored. This is clearly shown by Skorobogatov,[111] who describes the effect for many types of external SRAM ICs. Our own tests indicate that for some types of microcontrollers data remanence can last for extended periods of time, e.g. various types of Microchip PIC16 chips retain stored data for over 10 minutes when disconnected from a power supply, even at room temperature. Clearly this effect heavily interferes with the ability to take independent measurements quickly and effectively. It is thus crucial for our measurement setup that the data remanence effect is prevented. This can be accomplished by forcefully discharging the ICs under test by tying their V_+ lines to ground.

In order to simulate a realistic conditions, the voltage curve at power-up on the microcontroller's supply pins should behave as in a regular circuit. It should thus follow a curve similar to

$$V(t) = V_+ \cdot (1 - \exp^{-t/RC}),$$

where R is the output resistance of the power supply, and C the capacitance on the supply line. We assume a reference circuit running at 3.3 V in which the power supply can deliver 1 A, thus $R = 3.3\ \Omega$. We further assume a decoupling capacitance of $C = 100\ \mu\text{F}$. Thus, the voltage on the microcontroller's supply pins should reach 99% of the supply voltage after

$$\ln(1 - 0.99) \cdot RC \approx 1.52\ \text{ms}.$$

[111] S. Skorobogatov, "Low Temperature Data Remanence in Static RAM" (2002).

The voltage curve should be monotonically increasing, i.e. no glitches or spikes should be present. Note that as long as the supply power-up curve is realistic, slight deviations of the calculated time are not much of a problem, especially if the actual time is shorter.

Summarized, the following properties are required of the controller:

1. Support any arbitrary SRAM size.
2. Support any realistic baud rate.
3. Allow connection of many microcontrollers at once.
4. Be extensible in regard to the number of attached microcontrollers.
5. Make automated, unsupervised measurements possible.
6. Support remote setup.
7. Actively discharge microcontrollers that are not being measured.
8. Supply voltage curve of microcontrollers should be realistic.

A simplified schematic of our design is shown in Figure 4.1. A set of multiplexers allow the controller to either connect itself or the microcontroller under test to the external serial communication interface. The controller continuously monitors the output of the currently powered microcontroller in order to detect when the communication has finished, after which a measurement of the next microcontroller is started. Using this system it is also possible to detect when communication has stalled, or no microcontroller is present. This allows the controller to continue taking unattended measurements should some of the devices under test fail, since it will dynamically adapt to such situations.

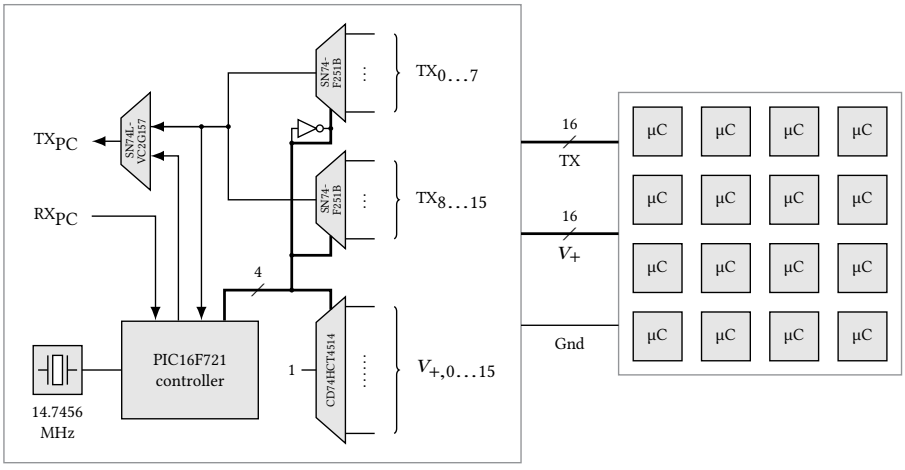


Figure 4.1: High-level schematic of the measurement controller board (*left*) with a board of microcontrollers to be measured attached (*right*).

The two 1-to-8 multiplexers used, SN74F251B, have tri-state outputs. This allows us to simply tie the outputs of several ICs together in order to combine the outputs, as long as we ensure only a single multiplexer is enabled at a time. Such a design is especially convenient for future extensions, whereby more microcontrollers are connected to the central controller.

Our current setup can be connected to up to sixteen microcontrollers. However, the system is easily extensible to at least 1024 devices, should that prove necessary. Note however that each microcontroller is measured sequentially, and thus for measuring a large number of microcontrollers multiple controllers operating in parallel might be preferable in order to reduce measurement time.

The microcontrollers are powered by a 4-to-16 demultiplexer, more specifically a CD74HCT4514 IC. This IC features push-pull outputs, and thus microcontrollers are automatically discharged when they are not powered. Using an oscilloscope we verified that this demultiplexer supplies voltage with a realistic curve to the microcontrollers.

The brain of the measurement controller itself is a Microchip PIC16F721. A basic interface allows the selection of different baud rates, and taking both automatic and manual measurements. The IC is clocked with a crystal oscillator specifically targeted at asynchronous serial communication, running at 14.7456 MHz. That way, we can guarantee compatibility with all common baud rates, and thus virtually all microcontrollers can be interfaced with the controller.

Finally, additional components that were used, but not shown in the schematic of Figure 4.1, are a voltage converter, decoupling capacitors, and LEDs to allow visual identification of the currently powered microcontroller.

4.2.2 Firmware setup

Firmware for the microcontrollers under test is simple. The required behavior at power-up can be summarized as:

1. Initialize the serial port (UART).
2. Transmit the value of every SRAM byte over the UART.
3. Idle or execute code that doesn't use the UART.

Care should be taken not to use any of the SRAM storage while doing this. Most microcontrollers have several working registers to store variables, such as a pointer to the current SRAM byte, which makes this is easy to achieve. However, some microcontrollers, such as the Microchip PIC16 family, only have a single working register and therefore, in order not to use any SRAM bytes, some variables used by the measurement firmware have to be stored in unused configuration registers instead.

Note that this more or less requires the firmware to be written in assembly. Using a C (or other language) compiler to generate code will almost certainly guarantee that part of the SRAM is used, and thus initialized.

In the interest of making our measurements as repeatable as possible, assembly listings of the firmware for each measured microcontroller are included in Appendix A.

4.2.3 Microcontroller overview

Despite all being part of prominent microcontroller families, there are quite some differences between the measured microcontroller ICs.

Two of the selected microcontrollers are 8-bit architectures. The first of these is the Atmel ATmega328P, part of Atmel's AVR family. This chip contains a total of 2 KiB SRAM. Pictured in Figure 4.2a is a PCB populated with the sixteen ICs used for our measurements. A firmware assembly listing can be found in Listing A.1.

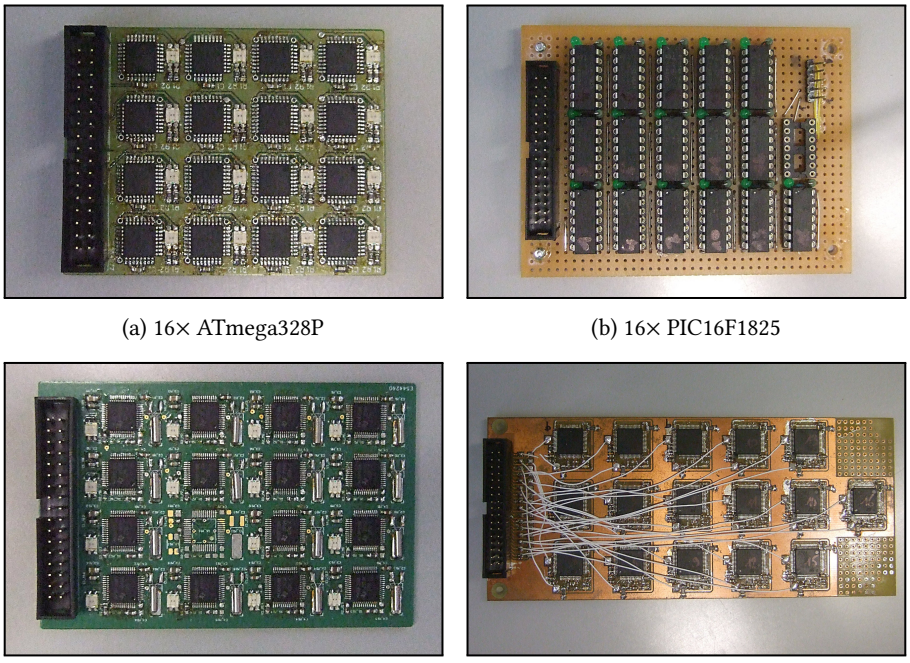
The second 8-bit microcontroller is the Microchip PIC16F1825, part of Microchips' PIC10/12/16 family. The PIC16F1825 contains 1 KiB of SRAM. The protoboard PCB with 16 ICs which we used for our measurements is shown in Figure 4.2b. The firmware assembly is shown in Listing A.2.

Next up is the Texas Instruments MPS430 family. These are 16-bit microcontrollers targeted to devices requiring ultra low-power consumption. The IC we measured, the MSP430F5308, contains 6 KiB SRAM. Figure 4.2c shows the PCB we used for measurements, populated with fifteen ICs. The assembly for this microcontroller's firmware can be found in Listing A.4.

The fourth, and final, chip is the 32-bit STMicroelectronics STM32F100R8. This is an ARM Cortex-M chip, more specifically a Cortex-M3. It contains 8 KiB SRAM in total. Shown in Figure 4.2d is the PCB used for our measurements. Due to lifted traces underneath some ICs, only eleven of the sixteen microcontrollers can be measured. Listing A.3 shows the assembly code used to generate the measurement firmware.

4.2.4 Operating conditions

While operating conditions such as supply voltage and power-up voltage curve, have an influence on SRAM power-up behavior, they are generally well controlled, assuming no physical tampering takes place. Temperature, on the other hand, is an operating condition which can, and does, significantly change depending on where a device is deployed, and which can significantly alter SRAM behavior.



(a) 16× ATmega328P

(b) 16× PIC16F1825

(c) 15× MSP430F5308

(d) 11× STM32F100R8

Figure 4.2: Measurement PCBs for the various microcontrollers.

Therefore all measurements are repeated three times, once at room temperature (±20 °C), and furthermore at both ends of the commercial temperature spectrum, respectively at −30 °C and 85–90 °C. In order to guarantee controlled and repeatable conditions, the measurements at both temperature extremes were taken in a temperature chamber (TestEquity 105 Half Cube).

4.3 Measurements & evaluations

With the preliminaries out of the way, we now discuss the measurements for the four microcontroller families. The results are presented per metric, making it easy to compare results for the various microchips. It will furthermore allow us to clearly show that all SRAM is, in fact, not created equally, at least for microcontrollers.

As the astute reader will notice, some of the plots further on in this section, and some of the tables in Appendix B, contain data only for a subset of the measured ICs. This is because partial failure of the interconnection between the PCB containing the

microcontrollers under test and the central controller board due to both freezing and elevated temperatures, which lead to disturbances in the communication between the measurement infrastructure and the PC storing the measurement data.

The number of measurements taken for each microcontroller IC are listed in Table 4.1. These include faulty measurements due to the aforementioned interconnection failure. The software used to calculate metrics automatically discards outliers. A measurement is considered an outlier when its fractional Hamming weight is either below 10% or above 90%, unless this is the case for the majority of measured ICs of a particular type. We have also discarded measurements that do not contain the correct number of bits due to communication issues.

Table 4.1: Number of measurements taken for each microcontroller IC.

Micro-controller	Temperature [°C]		
	−30	±20	85–90
ATmega328P	2 916	9 695	2 989
PIC16F1825	3 662	3 700	3 671
MSP430F5308	3 339	3 174	3 359
STM32F100R8	7 745	3 419	9 003

4.3.1 Plot conventions

This section contains a lot of plots, the conventions of which we will now explain. Visual representations of SRAM contain a single block per bit of data, black for those with value one, white for those with value zero. The blocks are plotted MSB first, from left to right, and from top to bottom. Each line contains an integer multiple of eight bits. If the last row does not have the same length as other rows, it is padded with zero blocks. Plots for the same device at different temperatures are always generated from data from the same physical IC. This allows one to check for obvious problems with intra-device distance, i.e. if the visual patterns at different temperature are very dissimilar. The plots also allow one to look for obvious patterns in the data, which are not necessarily discovered by the other metrics.

For box plots, the central box extends from the lower (25^{th} percentile) to the upper quartile (75^{th} percentile), i.e. 50% of the data points fall inside this range. The median of all measurements is drawn as a white line inside the box. The top and bottom whiskers extend to the 2^{nd} and 98^{th} percentile respectively. Finally, the minimum and maximum are marked with a • symbol.

We present all results as fractions, i.e. a Hamming weight of 1024 for an IC with 2048 bits of SRAM is written as 50%. This greatly simplifies making comparisons between the different microcontrollers. Expansive tables containing values for each metric for each microcontroller instance we measured, can be found in Appendix B.

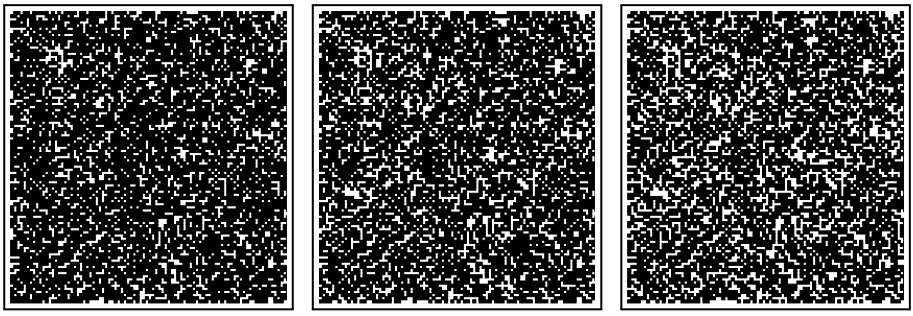
4.3.2 Visual inspection

We first investigate whether a visual representation of the SRAM power-up values reveals any obvious problems with the data, such as the presence regular patterns within it. In the ideal case, each pattern should look random, but patterns for the same device at different temperature should look very similar.

In order to improve visibility, we crop the SRAM visualization so they display maximum 1 KiB. If this is not done, identifying details in the SRAM plots is extremely difficult, if not impossible, since they tend to look like greyish blobs. Because of this cropping, all visualizations have the same scale, further easing comparability.

4.3.2.1 Atmel ATmega328P

The plots in Figure 4.3 do not reveal any obvious patterns in the ATmega328P data. There is, however, a pronounced darkening of the visualizations as the temperature decreases. This indicates a strong bias towards one-valued bits, which wil show up in the Hamming weight metric. Because the density of the plots differs across the temperature range, intra-device density over temperature is likely to be quite high.

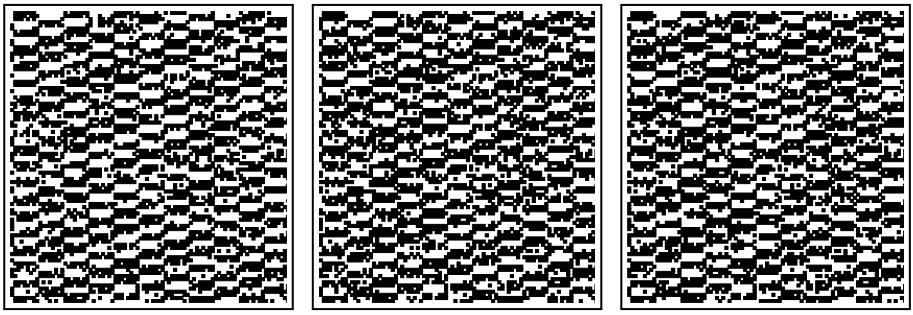


(a) −30 °C (b) ±20 °C (c) 90 °C

Figure 4.3: 1 KiB slice SRAM power-up data visualization for ATmega328P.

4.3.2.2 Microchip PIC16F1825

Figure 4.4 immediately show that the PIC16F1825 SRAM power-up data is highly problematic for PUF exploitation. There is a very obvious, regular, byte-wise pattern, whereby each byte consists of either all ones or all zeros, with the value being determined as the opposite of the previous byte's bits. Because all measured ICs show a similar pattern, inter-device distance is greatly reduced. The intra-device distance does seem to be reasonable over the temperature range, although the regular pattern makes it rather hard to estimate the plot's density.



(a) −30 °C (b) ±20 °C (c) 85 °C

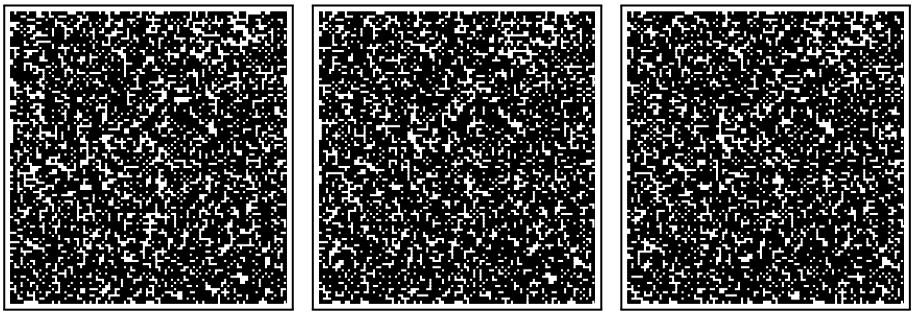
Figure 4.4: SRAM power-up data visualization for PIC16F1825.

4.3.2.3 Texas Instruments MSP430F5308

Similar random-looking patterns in all three plots in Figure 4.5 indicate a good intra-device distance. The MSP430F5308 SRAM power-up values seem markedly less influenced by temperature variations. The SRAM power-up values do seem to be rather strongly biased towards one, as evidenced by the high density of the plots.

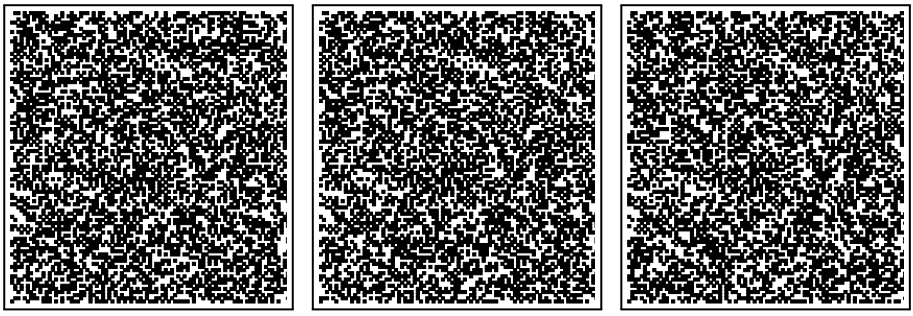
4.3.2.4 STMicroelectronics STM32F100R8

Like the MSP430F5308, the STM32F100R8 SRAM power-up values seem to be hardly influenced by temperature variation, as shown in Figure 4.6. The visualization looks quite similar at the three measured temperatures, so intra-device distance should be good. There are no obvious regular patterns visible, and the distribution of one and zero bits seems equal, indicating a Hamming weight close to the ideal 50%.



(a) −30 °C (b) ±20 °C (c) 90 °C

Figure 4.5: 1 KiB slice SRAM power-up data visualization for MSP430F5308.



(a) −30 °C (b) ±20 °C (c) 85 °C

Figure 4.6: 1 KiB slice SRAM power-up data visualization for STM32F100R8.

4.3.3 Hamming weight

For both PUF and RNG purposes the ideal SRAM should have an average Hamming weight of 50%. In Figure 4.7 boxplots are drawn for each of the measured microcontroller families. Statistics for the Hamming weight are calculated using a maximum of 3000 randomly selected measurements per IC. The values displayed by the boxplots are calculated using the sample mean of each of the measured ICs.

4.3.3.1 Atmel ATmega328P

One can clearly see the large effect temperature variations have on the Hamming weight of the ATmega328P, which is also visible as the change in density in its SRAM visualizations (see Figure 4.3). As already evidenced by the SRAM visualizations, there

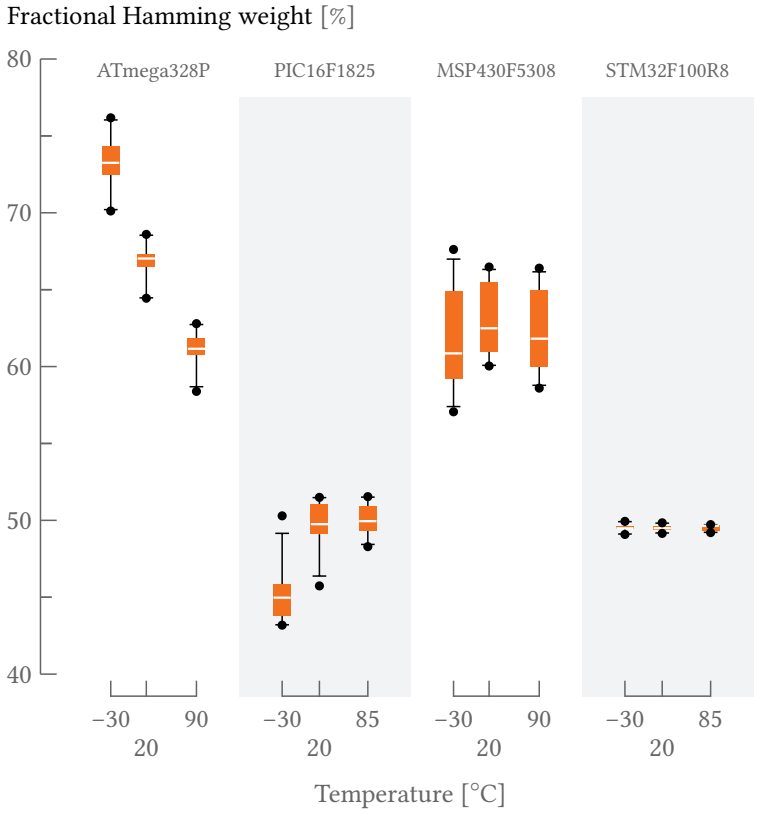


Figure 4.7: Hamming weight for each microcontroller family.

is a large deviation from the expected 50% HW: at −30 °C almost 75% of the bits power up with value one. At increasing temperatures the HW improves in fairly big steps, yet even at 90 °C the sample median is still above 60%. There is also a rather large difference in HW between the measured ICs, up to 7% at −30 °C.

Such a strong bias to one must necessarily reduce the inter-device distance, as shown in the example in Section 2.6.2, and thus reduces the entropy of the SRAM power-up values if used as a PUF response. However, this does not necessarily mean that the ATmega328P cannot be used as a PUF, only that for a given security level a larger SRAM will be required compared to an ideal SRAM with 50% HW. One can expect the intra-device distance, and thus the available entropy for RNGs, to be reduced as well, although the HW metric does not give definite proof of that.

4.3.3.2 Microchip PIC16F1825

The PIC16F1825 has a well-behaved HW at both room and elevated temperatures, with most ICs having close to the ideal 50% HW. At the high end of the commercial temperature range, the average HW for all measured ICs is clustered within 48 to 52%, almost ideal. There is a fairly long tail at room temperature though, with a few ICs having a bias of almost 5% to zero.

However, at −30 °C the HW has a bias of approximately 5% on average to zero. The HW distribution is long-tailed, with the HW of a few of the measured ICs being hardly influenced at all, while for the majority the SRAM power-up values show a strong bias towards zero (down to 43% HW). The PIC16F1825 is the only one of the measured ICs that shows a linear correlation between HW and temperature; for all other measured microcontrollers the relationship is inverted.

Note that the very obvious pattern displayed in Figure 4.4 has no influence on the HW metric, because it has a HW of exactly 50%.

4.3.3.3 Texas Instruments MSP430F5308

A large fraction of approximately 62% of bits of the MSP430F5308 power up as one; this represents a rather large bias, albeit smaller than for the ATmega328P. The bias remains fairly constant across the measured temperature range, which hints at a low intra-device distance at non-room temperatures. For the measured ICs, the HW distribution is skewed towards 50%, the distance between the lower quartile and the median is a lot shorter than that from the median to the upper quartile.

Of all the measured microcontrollers, the MSP430F5308 displays the largest deviation in HW between the measured ICs, the distance between the lower and upper quartile at −30 °C is 11%, larger than the deviation of the median from the ideal 50% HW.

4.3.3.4 STMicroelectronics STM32F100R8

Finally, we have the STM32F100R8, which produces very good results. The Hamming weight is almost exactly the same at all temperatures, and centered just shy of 50%. Furthermore, there is no significant deviation from the median to speak of, all measured ICs behave the same way. Judging from this metric intra-device distance will likely be very small, and inter-device distance large.

4.3.4 Inter-device distance

For PUFs, it is important for the inter-device distance to be as large as possible, since it is a measure for the uniqueness of each PUF response. For RNG use cases, this metric has no importance, since it is based on the static content of the SRAM, which contains no entropy for an RNG to exploit. Ideally, the inter-device distance should be 50%, which is the maximum possible.

The plotted statistics are calculated over all pairwise combinations of measurements over all measured ICs, using 500 randomly selected measurements per IC. Thus, for each microcontroller family $500^2 \cdot \frac{n(n-1)}{2}$ pairs are considered, with n the number of measured ICs (i.e. between $13.75 \cdot 10^6$ and $30 \cdot 10^6$ pairs). Boxplots of the inter-device distance for each of the measured microcontroller families are shown in Figure 4.8.

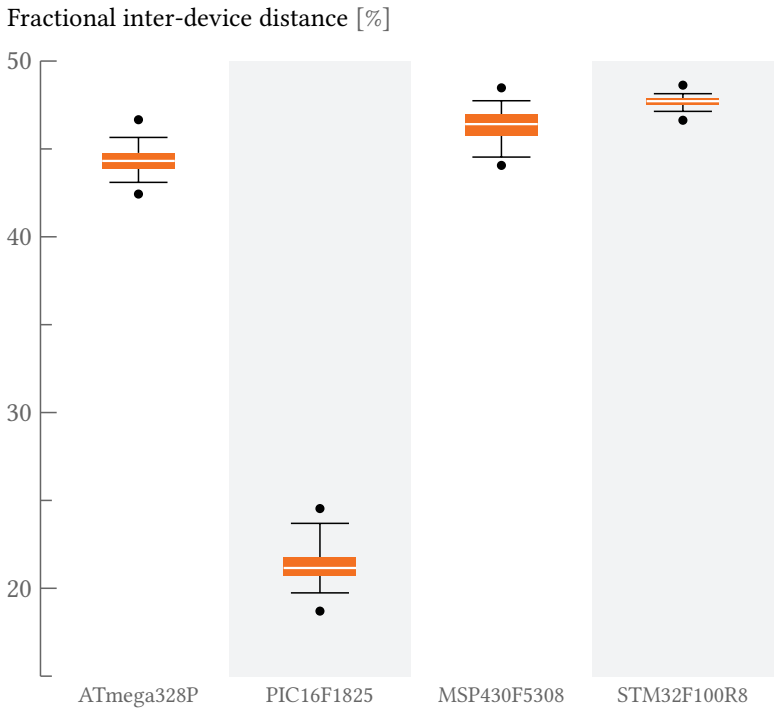


Figure 4.8: Inter-device distance for each microcontroller family.

4.3.4.1 Atmel ATmega328P

Despite its very strong bias towards one, and due to the lack of regular pattern within its SRAM power-up values, the ATmega328P has a reasonable good inter-device distance centered around 44%.

4.3.4.2 Microchip PIC16F1825

As could be expected due to the regular pattern shown in Figure 4.4, the PIC16F1825 has a very low median inter-device distance of 22%. The worst case pair even has a distance as low as 18%. Even ignoring the obvious security issues arising from the regular pattern, this microcontroller is likely to be highly problematic for PUF purposes, since, with such a low inter-device distance, error correcting codes are likely to transform puf_i's response into that of another puf_j. Just how likely this is to happen also depends on the intra-device distance.

4.3.4.3 Texas Instruments MSP430F5308

Similarly to the ATmega328P, the MSP430F5308 has a reasonable inter-device distance of approximately 46.5%.

4.3.4.4 STMicroelectronics STM32F100R8

Again the STM32F100R8 shows the best results of all the measured microcontrollers, with very little deviation from the median over all $13.75 \cdot 10^6$ pairs used for calculating the inter-device distance. Note that there seems to be a slight correlation between the measured devices, due to which the median sample mean, at 48%, is slightly deviated from the ideal 50%. This is to be expected, since all the HW statistics are also situated slightly below the 50% mark.

4.3.5 Intra-device distance

The intra-device distance plays an important role in parameter selection for both PUF and RNG designs. For PUFs, the lower the intra-device distance, the better, since that means one can use smaller and less complex error correction algorithms. Thus, for PUFs the ideal fractional intra-device distance is 0%. On the other hand, for RNG designs the intra-device distance is an indirect measure of the amount of entropy available. Thus, in that case, the ideal fractional intra-device distance is 50%.

Based on published results for external SRAM chips,[110] we can expect the intra-device distances across temperature to be higher than the intra-device distance at room temperature. This effect is also visible in the SRAM visualizations where plots at reduced and elevated temperature are either denser or sparser than the plot at room temperature. Note that intra-device distance at non-reference temperatures (i.e. non-room temperatures) are always calculated versus reference temperature measurements. This is because one always wants to correct the errors in a PUF response relative to a response measured at enrollment time, which happens under reference conditions.

We have calculated the intra-device distance statistics at reference temperature using all pairwise combinations of measurements per IC, using 1000 randomly selected measurements per IC. Thus, for each microcontroller family $n \cdot \frac{1000 \cdot 999}{2}$ pairs are considered, with n the number of measured ICs (i.e. between approximately $5.5 \cdot 10^6$ and $8 \cdot 10^6$ pairs). For intra-device distance at non-reference temperatures each possible pair of a non-reference and a reference temperature measurement was considered, i.e. $n \cdot 1000^2$ pairs per IC. The plotted data for each microcontroller family is calculated from the sample mean intra-device distance for each measured IC in that family. The results are plotted in Figure 4.9.

4.3.5.1 Atmel ATmega328P

At room temperature, the ATmega328P produces remarkably consistent responses, with a maximum calculated intra-device distance of only 2.5%. As expected, this maximum rises at 90 °C, to approximately 6.5%. At both temperatures the deviation from the median is very limited, approximately 0.5% and 1.25%.

However, at −30 °C a very different result emerges. The median intra-devices distance at this temperature is nearly 10%, with certain pairs showing up to 13%, the largest of any of the measured ICs. Furthermore, the range between minimum and maximum calculated distance is over 5%, i.e. the maximum is almost double the minimum. This means that in order to use the ATmega328P's SRAM power-up values as a PUF, one has to use error correction which for the majority of ICs will be able to correct way more errors than required, and thus use more energy and space than what would be required for the average IC.

[110] G.-J. Schrijen and V. van der Leest, "Comparative Analysis of SRAM Memories Used as PUF Primitives" (2012).

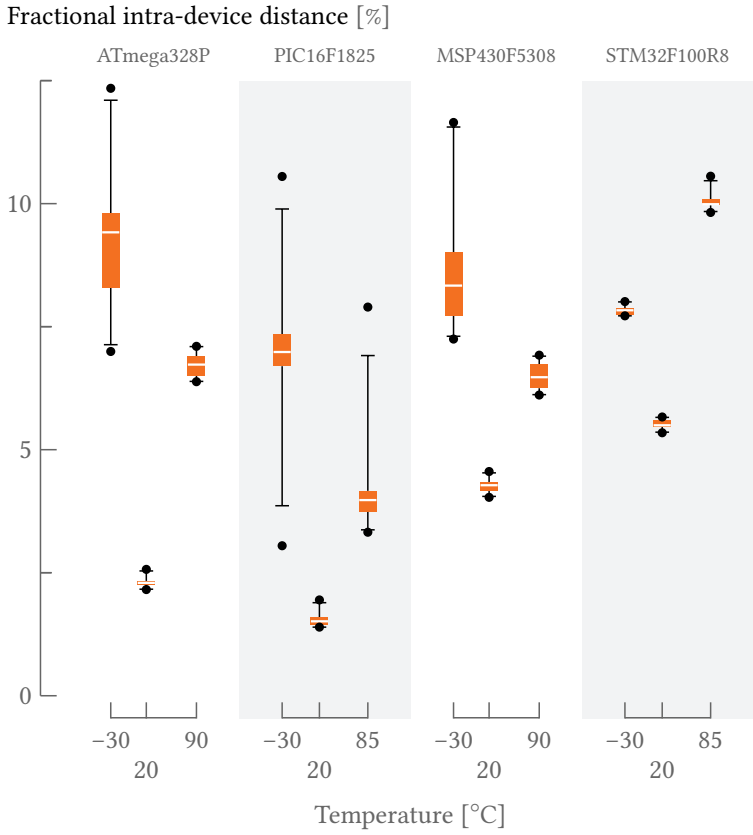


Figure 4.9: Intra-device distance for each microcontroller family.

4.3.5.2 Microchip PIC16F1825

The PIC16F1825 has an even smaller intra-device distance at room temperature than the ATmega328P, in fact, it boasts the smallest distance of all measured ICs. The calculated maximum is only 2%. Furthermore, the deviation from the median is very small, all calculated distances fall within approximately 0.75% of each other.

Unfortunately, at both −30 and 85 °C no trace remains of this behavior. The deviation increases to approximately 7.5% and 4.5% respectively. The maximum calculated intra-device distance at these temperatures is 10.6% and 8%, a multiple of the maximum at room temperature.

4.3.5.3 Texas Instruments MSP430F5308

The intra-device distance statistics of the MSP430F5308 are very similar to those of the ATmega328P, even if it has a larger intra-device distance at room temperatures. Just as for the ATmega328P, for real world use, the required parameters for the error correction algorithm will be such that for the majority of cases, a lot of unrequired energy and space is wasted, due to the 11.5% intra-device distance at −30 °C.

4.3.5.4 STMicroelectronics STM32F100R8

Compared to the other three microcontrollers, the STM32F100R8 has a much higher intra-device distance at room temperature, at approximately 5.5% it is even higher than the MSP430F5308's one. Contrary to the other ICs, freezing temperatures do not have that large an effect on the intra-device distance. At 90 °C, however, the distance does increase remarkably, to around 10%. At all three temperature conditions, the behavior of all measured ICs is fairly consistent, due to which the interquartile range is fairly limited. Due to this small range, out of all four measured microcontrollers, the STM32F100R8 will require the most lightweight error correction algorithm in practical applications.

4.3.6 Intra-device min-entropy

Lastly, we present the results for the min-entropy content of the SRAM power-up values of each microcontroller. As explained in Sections 2.6.7 and 2.8, we are interested in the worst case entropy, and will thus calculate min-entropy. Because we only have a rather limited number of microcontroller instances, it makes little sense to calculate the inter-device entropy. Calculating intra-device entropy for each of the microcontroller ICs poses no problem however, and it is thus this type of entropy metric we present here.

As discussed before, intra-device entropy is crucial for RNG applications. It gives a strong indication about the quality of the SRAM power-up values for use as an entropy pool. For RNG applications, the ideal case is if each bit in the SRAM has an entropy content of 1 bit of information.

The min-entropy for each microcontroller is calculated using 500 randomly selected measurements per IC, i.e. $500n$ measurements in total per microcontroller family, with n the number of measured ICs. We assume a model where the noise on the SRAM bits is independent, but not identically distributed. The results are shown in Figure 4.10.

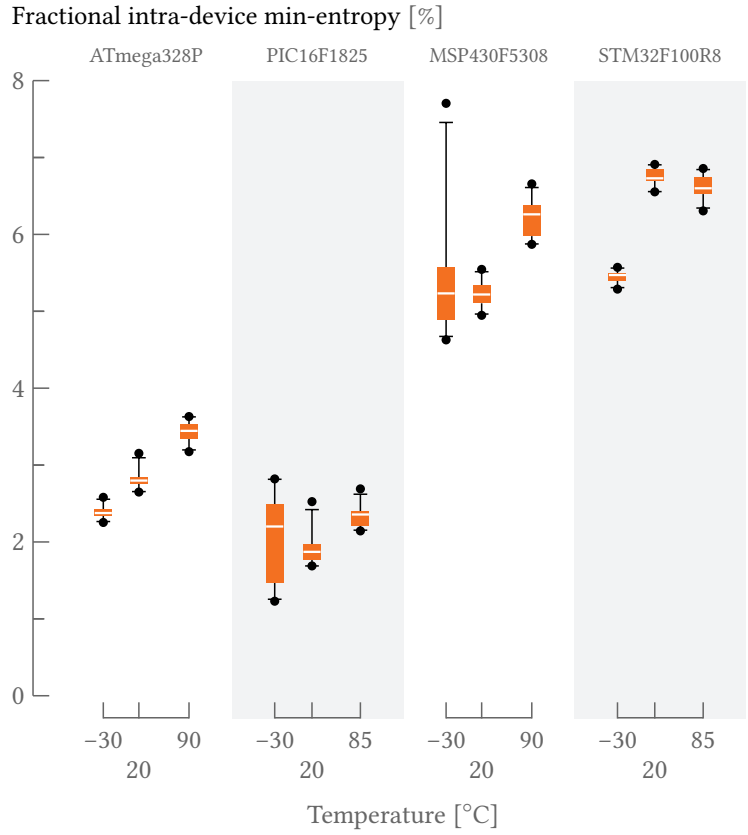


Figure 4.10: Intra-device min-entropy of SRAM power-up values for each microcontroller family.

4.3.6.1 Atmel ATmega328P

The min-entropy for the ATmega328P rises the higher the temperature becomes, with the results for each temperature point being closely grouped together. As can be expected, the min-entropy at room temperature is similar to the intra-device distance.

4.3.6.2 Microchip PIC16F1825

Calculations show that the PIC16F1825 min-entropy is rather low, compared to the other microcontrollers. The median rises with temperature, although it should be

noted that at −30 °C the range between minimum and maximum is very large, with the maximum being higher than the maximum calculated for 20 and 85 °C.

An inspection of the raw SRAM data reveals that while a few ICs do not seem affected much by the freezing temperature, for others a large percentage of SRAM cells powers up in the zero state, which explains the extremely low minimum of 1.2%.

4.3.6.3 Texas Instruments MSP430F5308

At −30 °C two of the measured ICs contain a large amount of min-entropy, even higher than at 90 °C. Although these two ICs are definitely outliers compared to the others, it is hard to ignore them, given that they represent one eight of the measured ICs.

For the majority of ICs the change in min-entropy at different temperatures follows a pattern similar to the ATmega328P. However, in the case of the MSP430F5308 the min-entropy is almost twice as high. The minimum measured is 4.6% at −30 °C.

4.3.6.4 STMicroelectronics STM32F100R8

The results for the STM32F100R8 are, as for all other metrics, tightly grouped. At −30 °C there is a slight decrease in min-entropy, due to an increased number of cells powering up in the zero state. Increased temperature seems to have little effect, the min-entropy at both 20 and 90 °C is almost identical. Ignoring the two outliers in the MSP430F5308 results, the STM32F100R8 has the highest amount of min-entropy in its SRAM power-up state.

4.4 Discussion

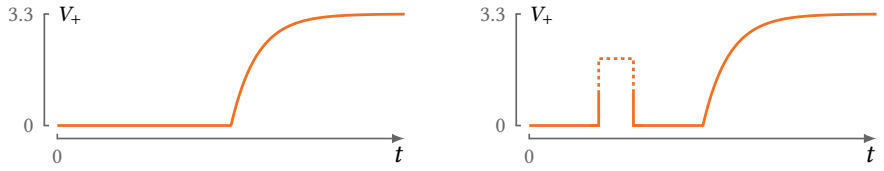
It is clear from the results that SRAMs are definitely not all created equal. As such, a short discussion on our findings and a qualitative comparison between the measured microcontrollers for the purpose of SRAM-based PUF and PRNG applications is in order.

4.4.1 PIC16F1825

Of all the measured ICs, particularly the PIC16F1825 is unsuited for either PUF or RNG applications. For PUF applications, the biggest problem is that all of the measured PIC16F1825 ICs contain a repetitive, highly structured pattern. The existence of this

pattern heavily undermines the security of whatever PUF-based application one might implement on this microcontroller.

In the process of trying to find the cause of this pattern, we have found that the voltage power-up curve has a strong influence on the SRAM power-up state. While an IC is normally powered up with a curve such as that in Figure 4.11a, we have found that using a "glitched" pattern, such as in Figure 4.11b, greatly reduced the regularity of the pattern.



(a) Regular (b) Glitched

Figure 4.11: Voltage power-up curves tested on PIC16F1825.

From a practical point of view, the "glitch" solution solves little. First of all, one would need to add extra components to the circuit in order to reproduce the glitch at every power-up. Besides working against our goal of using no external components, this would increase the cost of the circuit. Even more troublesome however, is the fact that the voltage level to which the glitch needs to rise in order to reduce the pattern is different for all sixteen ICs we tested it on. The required glitch voltage was in the range of 2.3 to 3.3 V for the tested ICs. Thus, the glitch-producing circuitry would need to be calibrated for every device. Due to these two reasons, we did not investigate this phenomenon any further.

Unfortunately, the internal layout of the PIC16F1825 is a closely guarded secret by Microchip. As such, our best guess as to the origin of the pattern is an analog circuit inside the microcontroller that has an influence on the power-up voltage curve to the SRAM.

An extra problem is that the maximum calculated intra-device distance is 10.6%, and the minimum inter-device distance approximately 18%. Assume that the intra-device distance follows a Gaussian distribution, with as parameters the calculated sample mean and standard deviation of the worst case measured IC ($\mu = 10.6, \sigma = 1.5$). In that case, approximately 1 in $10^{6.4}$ PUF responses will be further than 18% from the non-erroneous response, which is borderline acceptable. Note that this example does not use any safety margin whatsoever.

If a small 3% safety margin is added, i.e. $\mu = 13.6$, then in order to guarantee an error

correction failure rate $\leq 10^{-6}$ (standard for consumer grade devices), one needs to be able to correct at least

$$1 - 0.5 \cdot \left(1 + \operatorname{erf}\left(\frac{x - 13.6}{\sqrt{2} \cdot 1.5}\right)\right) = 10^{-6} \Leftrightarrow x \approx 20.76\% \text{ errors.}$$

These are more errors than the minimum expected inter-device distance, i.e. the erroneous response of puf_i will often be closer to the corrected response of puf_j than to puf_i itself. Thus, even ignoring the problematic pattern, it seem infeasible to implement a reliably working PUF using the PIC16F1825's SRAM power-up state.

For the purpose of PRNG applications, the PIC16F1825 is problematic as well. The very low min-entropy combined with the small available amount of SRAM, leads to an expected minimum min-entropy of approximately 100 bits of information. One generally wants to seed a PRNG with entropy at least as high as the security level![8] Thus, that means that no PRNG with a security level higher than 2^{100} can be constructed with the PIC16F1825.

4.4.2 ATmega328P and MSP430F5308

The ATmega328P and the MSP430F5308 are rather similar, for the purpose of PUF applications. They both have a strong bias towards one, especially so the ATmega328P. Due to this bias, one needs to use longer PUF responses in order to obtain the security level afforded by the response. For our purspose, we will only consider the bias at reference temperature, because that is the temperature at which enrollment is executed. In the following example, we assuming a fractional Hamming weight of 70% for both ICs, and require a min-entropy of at least 128 bits of information. In order to achieve, one requires a PUF response of at least

$$\left\lceil \frac{128}{-\log_2(0.70)} \right\rceil = 249 \text{ bit.}$$

Note that in this case, we assume no loss of information due to public helper data. I.e. we assume that, apart from the bias, the SRAM in both microcontrollers behaves as a perfect PUF with 0% intra-device distance, and thus no error correction is required.

In reality, for the ATmega328P one would need to take into account an error rate of at least 15%; for the MSP430F5308 13% might be sufficient. As it turns out, this is the same as in the PUFKY design. Assume the same error correction algorithms and parameters are used as in the PUFKY design (i.e. 2 226 bits of PUF response data and 2 052 bits of helper data, see Section 3.5). It is clear that using only error

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).

correction methods, it is in this case impossible to construct a viable PUF design, since the assumed loss of information due to helper data is greater than the worst case inter-device entropy in the SRAM:

$$-\log_2(0.70) \cdot 2226 \approx 1145 < 2052\,.$$

Due to the bias, a PUF construction using only error correction is thus infeasible.

However, it is feasible to construct a PUF if one of the techniques for dealing with bias proposed in Section 2.7.4 is used. We will once again assume the same error correction scheme as used for PUFKY. Note that we simplify our example design by assuming the error rate for both zero and one bits are the same. Therefore, a total of $l' = 2226$ bits of PUF response are required. We further assume the two-step debiasing algorithm and that the bias p towards one is the same as before, i.e. $p = 0.7$. Therefore, the number of biased PUF response bits n required is:

$$\begin{aligned} n &= l' \cdot \left(pq + \frac{1}{2} \cdot \frac{p^2q^2}{p^2 + q^2}\right)^{-1} \\ &= \frac{2226}{0.2480} \approx 8976\,. \end{aligned}$$

A closer look at our measurements reveals that these parameters should also work in an actual implementation, i.e. one where the error rate for zeros and ones differs.

In total, approximately 1 KiB of SRAM is required to extract enough unbiased data for the error correction algorithms to work with. Both the ATmega328P and the MSP430F5308 have more SRAM than that available, so this is not a problem. The remaining amount of entropy after error correction is expected to be approximately

$$-\log_2(0.5) \cdot 2226 - 2052 = 174 \text{ bit.}$$

PRNG constructions are also feasible on both microcontrollers. For the ATmega328P, the minimum observed 2% intra-device min-entropy corresponds to approximately 327 bits of information available in the whole SRAM. Whereas for the MSP430F5308, the minimum 4.6% equates to 2.2 Kibit of information. In both cases this is more than the suggested minimum for seeding a PRNG with 128-bit security level.

4.4.3 STM32F100R8

Again, lastly we have the STM32F100R8, which, as can be expected from the preceding sections, seems to be an excellent platform for both PUF and PRNG applications.

In the case of PUFs, assuming a worst case Hamming weight of 48%, and an error rate of 13%, and the same PUFKY parameters as before, after correcting a block of 2226 bits,

a total entropy of $-\log_2 0.52 \cdot 2226 - 2052 = 48$ bits of information remains. Applying a correction on three such blocks gives a total of 144 bits of information. Of course, in an actual implementation, one would choose error correcting code parameters better suited to the 32-bit ARM Cortex-M architecture.

As for PRNG applications, we expect a min-entropy of at least 5.25%, corresponding to 3.36 Kibit. Thus, even factoring in a generous safety margin should pose no problem to obtaining sufficient entropy for PRNG designs with very high security levels.

4.5 Conclusion

We conclude this chapter by summarizing the discussion on practical PUF and PRNG applications using the measured microcontrollers. The feasibility of using these microcontrollers for either purpose is shown in Table 4.2.

Table 4.2: Practical feasibility of PUF and PRNG applications on measured microcontrollers. An ✗ indicates infeasible, an ✓ indicates feasible, and an ~ indicates feasible if certain conditions are met (e.g. the use of a debiasing algorithm).

	ATmega328P	**PIC16F1825**	**MSP430F5308**	**STM32F100R8**
PUF	~	✗	~	✓
PRNG	✓	✗	✓	✓

It seems that especially for PUF implementations, one should not pick just any microcontroller. Of the four microcontrollers we tested, only the STMicroelectronics STM32F100R8 is suitable for secure, pure software-based PUF implementations using only error correction codes. Implementations on the ATmega328P and MSP430F5308 are possible, but require a debiasing step first, due to the strong bias of their SRAM cells towards one. The PIC16F1825 has two factors working against it: a regular patterns in the SRAM and a low inter-device distance, both of which make a secure PUF implementation impossible.

On the PRNG side, all tested microcontrollers, except for the Microchip PIC16F1825, can be used. In this case the problem with the PIC16F1825 is its low min-entropy, coupled with its small amount of SRAM. The ATmega328P, MSP430F5308 and STM32F100R8 are all adequate choices for a PRNG application. Out of these three, the ATmega328P is the least suited, due to the relatively low min-entropy in its SRAM power-up state and the rather small amount of SRAM available on the microcontroller.

In closing, it is clear that there is a large difference between embedded SRAMs. Therefore, the importance of conducting adequate measurements before deciding on a microcontroller for software-based PUF and/or PRNG designs cannot be overstressed. Of all the chips we tested, the STM32F100R8 proved to be exemplary in nearly every aspect, and thus seems like an ideal testing platform for future work.

5 Software-based Secure PRNG Design

Chaos is found in greatest abundance wherever order is being sought.
It always defeats order, because it is better organized.
— Terry Pratchett, Interesting Times (1994)

Section 1.1.4 has pointed out that embedded devices often lack methods to generate strong random numbers. As the measurements in Chapter 4 reveal, many microcontrollers contain relatively large amounts of entropy in their SRAM start-up values, both in the inter- and intra-device distances. Thus, there are no technological restrictions that prevent software-based implementations of both PUFs and strong PRNGs on those chips.

In this chapter, we present an implementation of a purely software-based, strongly seeded, cryptographically secure PRNG for COTS microcontrollers. Initial seeding material is extracted from the SRAM power-up data. Our design can function as a drop-in, black box module that provides strong random numbers to the rest of the system.

We build an implementation of this system for the ARM Cortex-M family of microcontrollers. The implementation is built around a version of Keccak with small parameters, and generates one byte of pseudo-random data every 3 337 cycles on the ARM Cortex-M3/4. The primary optimization goal is minimal code size. At 496 bytes of ROM and 52 bytes of RAM, our Keccak-f[200] implementation, and supporting functions, is the smallest Keccak implementation published so far.

Content sources

A. Van Herrewege and I. Verbauwhede, "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M", in *Design Automation Conference (DAC)*, San Francisco, CA, USA: ACM, 2014, 6 pp.
Contribution: Main author.

A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers", in *International Workshop on Trustworthy Embedded Devices (TrustED)*, A.-R. Sadeghi, F. Armknecht, and J.-P. Seifert, Eds., ser. TrustED '13, Berlin, Germany: ACM, 2013, pp. 55–64
Contribution: Main author.

A. Van Herrewege, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-the-Shelf Microcontrollers", in *ACM Conference on Computer and Communications Security (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., Berlin, Germany: ACM, 2013, pp. 1333–1336
Contribution: Main author, responsible for PRNG section.

5.1 Introduction

Our goal is the design and implementation of a completely self-sufficient, software-based cryptographically secure PRNG for an embedded microcontroller. It should be possible to run the PRNG as a black box in a larger design, without it compromising any functionality of the microprocessor. As our target platform, we have chosen the ARM Cortex-M microcontroller family, one of the most popular embedded platforms currently available. Code for this processor can be used on many powerful ARM Cortex-A microcontrollers as well, which are used in virtually every smartphone today.

Since a PRNG fulfills a support function, the primary design goal is to reduce both ROM and RAM requirements to a minimum, leaving more space available for the core functionality of the system the PRNG will be used in. Speed is far less important in this case, since random number generation occurs far less often in cryptographic protocols than, e.g., encryption.

The past years have seen a few publications related to generating random numbers from SRAM, or on embedded devices. Holcomb *et al.*[56] presents a method of using the power-up value as a source of randomness. The method's main drawback is having to power cycle the SRAM chip every time new randomness is required. Exactly due to this requirement, it is not possible to use the suggested method in a COTS microcontroller, since the controller would have to be reset each time random data is required. O'Donnell *et al.*[97] present a design whereby a PUF is repeatedly queried until an unbiased response is generated, which is then further post-processed in order to generate a random number. van der Leest *et al.*[114] present a custom hardware PRNG designed according to NIST recommendations,[8] which makes use of SRAM power-up values for seed generation. Francillon and Castelluccia[45] present a PRNG solution for embedded systems equipped with a radio module. In their design bit transmission errors are used as a source of randomness.

To the best of our knowledge, there are two other publications which discuss self-sufficient PRNG designs, requiring only a microcontroller, and no external components. The first of these, by Hlaváč *et al.*,[54] discusses the use of jitter between an external crystal and the internal RC oscillator on an Atmel ATmega169 as a source of randomness. They also suggest the use of both the internal RC oscillator and watchdog timer, to make the system self-sufficient, but do not verify whether such a construction has good properties as an entropy source. One possible drawback of their suggested design is the loss of the watchdog timer functionality, which might be critical in certain embedded systems.

A second publication, by Kristinsson,[69] investigates the use of analog-to-digital (ADC) measurements on floating analog inputs as a source of entropy. Such a design is suggested as a good source of "fairly random numbers" by the documentation of the `randomSeed()` function in the official Arduino Language Reference.[3] The paper's conclusion however is that this design in fact generates quite predictable data, not fit for PRNG seed generation. This is rather unfortunate, since the Arduino system is extremely popular, and thus the suggestion in its manual might inspire millions of people worldwide to make use of this insecure method.

---

[3] Arduino SA, "Arduino Language Reference" (2014).

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).

[45] A. Francillon and C. Castelluccia, "TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes" (2007).

[54] J. Hlaváč *et al.*, "True Random Number Generation on an Atmel AVR Microcontroller" (2010).

[56] D. E. Holcomb *et al.*, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags" (2007).

[69] B. Kristinsson, "Ardrand: The Arduino as a Hardware Random-Number Generator" (2012).

[97] C. W. O'Donnell *et al.*, "PUF-Based Random Number Generation" (2004).

[114] V. van der Leest *et al.*, "Efficient Implementation of True Random Number Generator Based on SRAM PUFs" (2012).

5.2 Design

In this section we will first discuss the attacker model we assume when later on deciding on parameters for our design. Next is a high-level overview of our PRNG design, followed by a discussion on parameter selection for the design.

5.2.1 Attacker model

In order to improve the practicality of our design, we choose an attacker model which most closely matches realistic expectations. We distinguish two main types of attackers: passive and active. For both of these, the goal is to break the PRNG's security by determining its internal state.

The passive attacker is equivalent to a regular user of the system, in that he only has access to parts of the system which are designed to be public. Thus, in the case of a PRNG, the passive attacker has access to the output of the PRNG and the publicly accessible interface required to make the system generate an output. The passive attacker has no physical access of any kind to the system, and thus his attacks are of a mathematical nature. Protection against such attacks requires adequate choices of security-related parameters, and can thus be done entirely in software.

Active attackers can further be divided into different categories, depending on the amount of funds they have access to. The basic active attacker is able to influence the state of the PRNG, e.g. through a restricted (re)seeding interface. Such attacks are software-based, and thus come for free. Defending against privilege escalation attacks can preclude these attacks, and requires either software-based solutions or additional hardware functionality, such as e.g. Sancus[96] or ARM TrustZone.[4] Note, however, that such attacks are not specific to our implementation, but rather a problem of the underlying operating system and/or hardware. Hence, preventing these attacks is outside the scope of our design.

A second class of active attackers are attackers that have physical access to the system. For low-budget adversaries this opens up at least two attack vectors. First of all, there are side-channel attacks,[46] which use e.g. power measurements in order to infer a secret used in software. Increasing the difficulty of such attacks can be done in software by making the PRNG run in constant time, and not having any conditional jumps based on secret data (i.e. the state), although this does not completely prevent such

[4] ARM Ltd, "ARM Security Technology: Building A Secure System Using TrustZone Technology" (2005).

[46] B. Gierlichs, "Statistical and Information-Theoretic Methods for Power Analysis on Embedded Cryptography" (2011).

[96] J. Noorman *et al.*, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base" (2013).

attacks. Stronger defenses against side-channel attacks include the use specialized hardware logic styles or adding noise sources to the circuit. Such constructs are outside the scope of this implementation; the best we can do is implement software-based solutions.

A second attack vector for low-budget physical access adversaries is the use of debugging interfaces to read out and/or reprogram (part of) the microcontroller on which the PRNG is running. This attack is rather problematic, since there is no way to defend against it from software. Furthermore, many microcontrollers are not built for security purposes, hence they do not allow one to completely disable all debugging interfaces. For example, the STM32L1xx Cortex-M3 microcontrollers by STMicroelectronics allow one to completely disable readout of both ROM and RAM storage through debugging interfaces.[112] Other microcontrollers in related families, e.g. the STM32F1xx, do not offer such functionality, and are always vulnerable to readout through the JTAG interface. However, even with (irreversible) protection enabled, the system might still be reprogrammed, and thus read out, through a bootloader. Therefore, even with appropriate fuses set, a system might still be vulnerable. There are no steps that can be taken in software against these kinds of attacks, they are something that should be dealt with through a combination of hardware protection and thorough code auditing of outward facing communication interfaces (e.g. a bootloader).

The final class of physical access attackers consists of those with a large to virtually unlimited amount of funds. Such attackers can, amongst other feats, physically probe buses and registers inside the microcontroller, and reverse the state of "permanently" blown fuses. The only realistic defense against such attacks is in trying to make them economically infeasible. However, there exist no mechanisms which can completely defend against a determined, skilled attacker of this class. Note that COTS microcontrollers exist which offer such kinds of deterrents, e.g. Maxim DeepCover Secure Microcontrollers,[84] naturally these cost significantly more than unprotected ones. Protecting against such an attacker is, again, clearly outside of the scope of our design.

Taking into account the above, the only adversary which we can realistically defend against by adapting our software implementation is the passive one. Defending against the basic active attacker should be feasible as well, but doing so is matter to be handled by the operating system (if any is used) or supporting hardware; it has no influence on our implementation. However, adversaries with physical access will most likely be able to break into the system through a debugging interface, unless specialized high-security microcontrollers are used. Thus, from a realistic point of

[84] Maxim Integrated, "DeepCover ® Secure Microcontrollers" (2014).

[112] STMicroelectronics, "AN4246 - Proprietary Code Read Out Protection on STM32L1 microcontrollers" (2013).

view, if the attacker manages to gain physical access to the system, he will most likely be able to break it. Defenses against the strongest class of attackers all fall far outside the scope of our lightweight implementation, and thus such attackers can be ignored in our this implementation.

5.2.2 High-level PRNG overview

As is common practice, we base the design of our PRNG on the recommendations outlined in the NIST SP800-90 document.[8] A high level overview of the strongly seeded PRNG architecture is given in Figure 5.1, and consists of three building blocks.

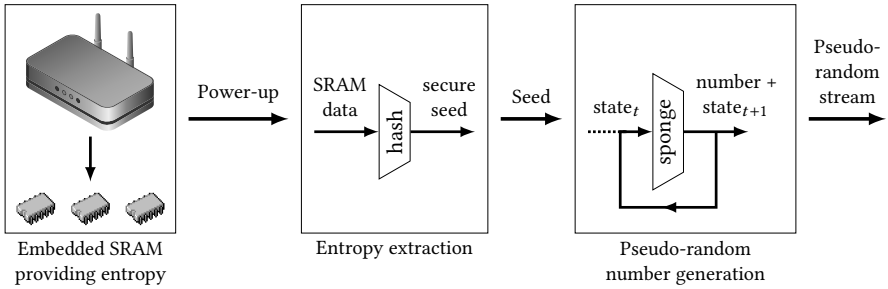


Figure 5.1: High-level construction of PRNG strongly seeded with SRAM power-up data entropy.

The first of these building blocks is the entropy pool, for which we have chosen the power-up values of the SRAM built into the microcontroller. The main reason for selecting SRAM have been outlined in Chapter 4: instant availability and a sufficiently high entropy content. The SRAM should contain an amount of entropy at least equal to the required security level of the PRNG.[8] Taking for example the entropy density of the STMicroelectronics STM32F100R8's SRAM power-up values, which is minimum 3% under realistic conditions, and assuming that the entropy is distributed uniformly, one would need at least $\lceil 128/0.03 \rceil \text{bit} \approx 0.5$ KiB of SRAM to seed a PRNG with a security level of 128 bits. Since even the smallest microcontroller in the STM32F1 family has 4 KiB RAM, meeting this requirement poses no problem.

Accumulating the entropy available in the SRAM power-up data can be accomplished with a universal hash function (see Definition 1.7) or a cryptographic hash function.

[8] E. Barker and J. Kelsey, "Recommendation for Random Number Generation Using Deterministic Random Bit Generators" (2012).

In order to accumulate the entropy, the SRAM data is hashed, with the final hash serving as the seed for the PRNG. The hash function's state does not need to be initialized for this, that way a little time is saved and the state might already contain some entropy due to the SRAM power-up values it contains.

The final block in the design is the PRNG itself. For this step, NIST recommends using either a block cipher in hash mode (e.g. AES in CBC-MAC) or HMAC[10,68] with an approved hash function.

Since both the second and third step can be implemented using a cryptographic hash function, we choose to implement such function just once and use it for both steps, for efficiency reasons. The disadvantage of using an HMAC construction in the third step is that two calls to the hash algorithm are required for each call to the HMAC algorithm. Newer, sponge-based hash functions do not require the HMAC construction to be able to securely generate random numbers and don't have this disadvantage. We therefore deviate from the NIST recommendations and follow those outlined by Bertoni *et al.*,[15] who present a lightweight PRNG construction using a sponge-based hash function. In our implementation, we choose to use Keccak[16] as the sponge-based hash function. Keccak has recently been accepted as the new SHA-3 standard. The NIST recommendations have not yet been updated to include this new standard.

Keccak is shown in Figure 5.2 as it is used in building blocks two and three. Its internal state is divided into two section, a public part of length r, and a secret part of length c. The size of data input and output is equal to r, and is called the rate. The length c is a measure for the security of the hash function, and is called the capacity. The total size of the internal state is thus $r + c$.

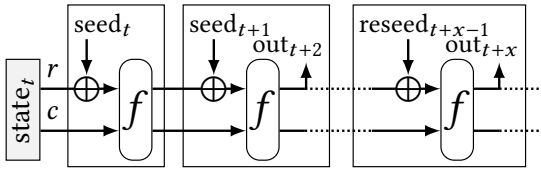


Figure 5.2: A sponge-based hash function used as PRNG. First the state is initialized with seeding data, after which pseudo-random numbers can be generated. Reeseding can be done at any time by XORing in new data. The rate of the sponge is r, its capacity c.

[10] M. Bellare *et al.*, "Keying Hash Functions for Message Authentication" (1996).
[15] G. Bertoni *et al.*, "Sponge-Based Pseudo-Random Number Generators" (2010).
[16] G. Bertoni *et al.*, "The Keccak reference" (2011).
[68] H. Krawczyk *et al.*, "HMAC: Keyed-Hashing for Message Authentication" (1997).

To accumulate entropy in the state, data seed_t is fed into the hash. This process of inputting data is called absorbing in sponge function terminology. Once enough entropy is accumulated, the output of the hash function can be used as a PRNG. Generating output, i.e. running the sponge function without feeding in data, is called squeezing. Multiple random numbers out_t can be generated by repeatedly squeezing the sponge function.

Concerning the robustness of the PRNG, as discussed in Section 1.1.4, we need to take into account three properties. This design is resilient as long as an attacker doesn't request more than a certain number of output blocks from the PRNG, as discussed in Section 5.2.3. Backward security can be achieved by reseeding, which is done by simply absorbing more data into the state. Due to the hash function's entropy accumulating nature, one does not need to know which bits of (re)seeding material contain the actual entropy, which enables the use of entropy sources with low entropy density. Finally, Bertoni *et al.*[15] discuss a technique to make the design forward secure by reseeding the PRNG with part of its own output.

5.2.3 Keccak parameter selection

From Bertoni *et al.*,[15] we learn the resistance against both passive and active state recovery attacks for the above construction. For passive state recovery attacks, i.e. where the adversary cannot influence the seeding data being absorbed into the sponge, the construction provides a security level of $2^c/m(A)$. The value $m(A)$ is the multiplicity as defined by Bertoni *et al.*,[15] and is approximately 1 if $l \leq 2^{r/2}$, with l being the number of blocks that have been output. For active state recovery attacks, whereby the adversary can influence the value of the state, e.g. by modifying seeding material, the construction offers a security level equal to $2^c/l$.

We assume that a security level of at least 128 bits is required and that pseudo-random numbers of 64 bits in length are convenient in most applications. Furthermore, the primary optimization goal of our implementation is to be as compact as possible, both in ROM and RAM. With that in mind, the ideal Keccak permutation to use is Keccak-f[200] with a rate $r = 64$ bits and a capacity $c = 136$ bits. This provides the required security as long as reseeding happens at least every $r \cdot 2^{r/2} = 32$ GiB of output in passive state recovery attack scenarios, or every $2^8 \cdot r = 2$ KiB of output in active state recovery attack scenarios.

Note that the allowed amount of output in the active state recovery attack is rather limited. We nevertheless choose these parameters for the following reasons. If an attacker can influence the value of the state, this must happen either directly through software, or indirectly through reducing the content of the entropy pools by

[15] G. Bertoni *et al.*, "Sponge-Based Pseudo-Random Number Generators" (2010).

influencing the operating conditions (e.g. reducing the operating temperature to a very low level). For a software attack, this means that the protections offered by either the operating system or hardware have failed against a privilege escalation attack. Preventing such attacks is, as discussed in Section 5.2.1, outside the scope of our implementation, and picking different parameters would not improve matters. The second option, in which operating conditions are influenced directly by an attacker, requires physical access to the system. As discussed in Section 5.2.1 as well, such access almost certainly guarantees that the system will be broken.

We therefore decide that the chosen parameters offer adequate protection under the assumption that no privilege escalation attacks are possible, and that an attacker does not have physical access to the system. Different parameters would not be able to offer increased security anyway, should either of those conditions be violated.

5.3 Implementation

We will now discuss the reasoning behind our target platform selection, present an overview of the actual implementation and discuss some techniques used to obtain our final result. This result, and a comparison of our implementation of Keccak with other published ones, is given next. Finally, we discuss some tests used to verify the correct working of the PRNG design.

5.3.1 Platform selection

Our implementation platform of choice is the ARM Cortex-M. This is a popular family of embedded 32-bit microcontrollers. The three major types of chips available in this family are the M0(+), M3, and M4(F). The M0(+) is an ARMv6-M architecture, the M3 is an ARMv7-M, and the M4(F) is an ARMv7E-M. Chips of the less powerful variety (e.g. M0) use a subset of the instruction set of the more powerful controllers (e.g. M4).

Furthermore, most of the chipsets used in modern mobile devices, such as tablets and cellphones, are of the ARM Cortex-A family. Because many of these chipsets additionally contain one or multiple Cortex-M controllers, our implementation can run on most of them as well.

Therefore we choose to first implement and optimize our code for the Cortex-M0, since the same code can then run on any Cortex-M chip available. This Cortex-M0 implementation is then further optimized for the Cortex-M3 and -M4 devices, because those are often incorporated into more powerful Cortex-A chips. The chips on which we tested our implementations are the STMicroelectronics STM32F051R8, STM32F100RB, and STM32F407VGT6.

5.3.2 High-level implementation overview

An execution of Keccak works by running the internal state, into which input data can be XORed, through a permutation function for a number of rounds. There are seven possible variants of this function, which is called Keccak-$f[x]$, with $x \in \{25, 50, 100, 200, 400, 800, 1600\}$. The state has a size in bits equal to the x parameter. The major difference between the functions is the word size with which they work. The state consists of 25 words of word size $x/25$. Thus, for Keccak-$f[200]$ the function works with 8-bit words, i.e. bytes. The second difference is the number of rounds per execution, which is equal to $12 + 2 \cdot \log_2(x/25)$, i.e. 18 in our case.

A straightforward implementation of the Keccak-f permutation, as shown in Algorithm 5.1, requires temporary variables larger than the size of the state while executing. Especially in constrained embedded devices, one would prefer to reduce this temporary storage as much as possible, which is why small space requirements are our primary optimization goal. It is possible to implement Keccak using a so-called in-place technique,[17] which reduces the amount of stack space required. Speed is a second optimization goal: it is not crucial in a cryptographic PRNG design, since in real-world systems most time will be spent executing other cryptographic algorithms.

Algorithm 5.1: Straightforward Keccak-f permutation algorithm for round i.[17]

Data: state A[5,5], temp. arrays B[5,5], C[5], D[5]

1 **for** $x \leftarrow 0$ **to** 4 **do**
2 $C[x] \leftarrow A[x, 0] \oplus A[x, 1] \oplus A[x, 2] \oplus A[x, 3] \oplus A[x, 4]$
3 **for** $x \leftarrow 0$ **to** 4 **do**
4 $D[x] \leftarrow C[x - 1] \oplus (C[x + 1] \lll 1)$
5 **for** $x \leftarrow 0$ **to** 4 **do**
6 **for** $y \leftarrow 0$ **to** 4 **do**
7 $A[x, y] \leftarrow A[x, y] \oplus D[x]$
8 **for** $x \leftarrow 0$ **to** 4 **do**
9 **for** $y \leftarrow 0$ **to** 4 **do**
10 $B[x, 2x + 3y] \leftarrow A[x, y] \lll p[x, y]$
11 **for** $x \leftarrow 0$ **to** 4 **do**
12 **for** $y \leftarrow 0$ **to** 4 **do**
13 $A[x, y] \leftarrow B[x, y] \oplus (\sim B[x + 1, y] \otimes B[x + 2, y])$
14 $A[0, 0] \leftarrow A[0, 0] \oplus RC[i]$

[17] G. Bertoni *et al.*, "Keccak implementation overview" (2011).

High-level pseudo-code for the in-place KECCAK-f permutation algorithm is given in Algorithm 5.2, which is based on Alg. 4 in the work of Bertoni *et al.*[17] Note that all indices are calculated mod 5. The arrays p and RC contain constants. The in-place technique saves on storage space by overwriting those parts of the state which are not required anymore by future calculations. This can be seen in the listing as the indexes into the state array A are first permuted by means of a matrix multiplication with $N = \left(\begin{smallmatrix} 1 & 0 \\ 1 & 2 \end{smallmatrix}\right)$. Because $\text{rank}(N) = 4$, bytes in the state array occupy their original position every four rounds. However, because for KECCAK-$f[200]$ the number of rounds is not a multiple of four, $18 \bmod 4 = 2$, the state array elements need to be swapped around after the last round execution in order to put them back in their correct position.

Whereas Alg. 4 in the work of Bertoni *et al.*[17] uses two separate arrays B and C, we combine these into array BC, since the variable C is not used after the calculation of D finishes. This reduces the required storage by 5 words (i.e. 5 bytes in this case), around 10% of the total required RAM space. Compared to the straightforward implementation the RAM space is reduced by 25 words, approximately 40% less.

Algorithm 5.2: "Looped" in-place KECCAK-f permutation algorithm for round i. Based on Alg. 4 in the work of Bertoni *et al.*[17]

Data: state A[5,5], temp. arrays BC[5], D[5]

for $x \leftarrow 0$ **to** 4 **do**
  $BC[x] \leftarrow 0$
  for $j \leftarrow 0$ **to** 4 **do**
    $BC[x] \leftarrow BC[x] \oplus A[N^i(x, j)^T]$
for $x \leftarrow 0$ **to** 4 **do**
  $D[x] \leftarrow BC[x-1] \oplus (BC[x+1] \lll 1)$
for $y \leftarrow 0$ **to** 4 **do**
  for $x \leftarrow 0$ **to** 4 **do**
    $BC[x+2y] \leftarrow (A[N^{i+1}(x, y)^T] \oplus D[x]) \lll p[N(x, y)^T]$
  for $x \leftarrow 0$ **to** 4 **do**
    $A[N^{i+1}(x, y)^T] \leftarrow BC[x] \oplus (\sim BC[x+1] \otimes BC[x+2])$
$A[0, 0] \leftarrow A[0, 0] \oplus RC[i]$

5.3.3 Implementation details

Our first implementation is done for a Cortex-M0, since that guarantees compatibility with other Cortex-M microcontrollers. Since minimal code and RAM size are the

[17] G. Bertoni *et al.*, "KECCAK implementation overview" (2011).

primary optimization goal, everything is implemented as loops (i.e. no loop unrolling). To further reduce code size, all indices are calculated on the fly, expect for those into the state register A. Those indices are bit packed into a byte together with the content of p (the number of positions to rotate). Because $\text{rank}(N) = 4$, the same bytes values repeat every four rounds. Thus, $4 \cdot 25 = 100$ bytes are required for this lookup table (LUT). Another 18 byte LUT is required for the round constants RC. It is possible to eliminate the 100 bytes lookup table by calculating the index into A on the fly. This would cost a few cycles extra each time, and, in the case of our implementation, comes at the expense of more RAM, a trade-off we choose not to make.

Further reductions in code size are made possible by making the LUTs word-aligned. This allows the use of certain efficient instructions for address calculation (i.e. `adr`) which would crash the microcontroller if the LUTs were byte-aligned.

Some more code space is reclaimed by reversing as many loop counters as possible, i.e. running the loops from 4 to 0. This is more efficient because comparisons with 0 can be done simultaneously with decrementing the loop counter, and thus an instruction can be saved when checking if the loop should be ended. Although such an optimization only allows the removal of a single instruction in most cases, it can significantly improve speed for the innermost loops, due to loop overhead reduction.

We obtain a speed increase of around 25% by optimizing our code for the Cortex-M3 and -M4 cores. This is first of all made possible by the more powerful instruction set, which allows operations such as addition with on-the-fly shifted operands or boolean operations with immediate values, whereas those values have to be loaded into working registers first on the ARMv6-M architecture (i.e. the M0).

The code can be further reduced and sped up by using the conditional execution capabilities of these more powerful cores. Conditional execution allows the microcontroller to ignore instructions if certain flags are not set, thereby removing the need for costly conditional jump instructions. This effect is most noticeable in inner loops, where the removal of jumps leads to a significant speed increase.

Finally, the order of load-modify-write instructions matters on these cores, whereas it doesn't on the Cortex-M0 that we used. On the M0 data is available for calculation as soon as its load instruction finishes, which is not the case on the M3 and M4 devices we used, due to a more complex pipeline. Thus, by carefully moving around code, speed gains can be obtained without any changes to the actual instructions used.

Each of these three techniques accounts for around 30% of the speed-up gained versus the Cortex-M0 implementation.

5.3.4 Results and comparison

These techniques reduce our initial Cortex-M0 code size from 736 bytes of ROM to 500 bytes, a 32% reduction. This includes not only the KECCAK-f[200] function (404 bytes), but initialization, hashing, padding and output generation code as well. During hashing 52 bytes of RAM are required, of which 28 are used to store the PRNG state and which thus need to be kept afterwards. Hashing takes 4 205 cycles per byte. On the Cortex-M3 and -M4, we manage to very slightly reduce code size to 496 bytes (of which 400 bytes are required for the KECCAK-f permutation), and the time required to hash is reduced to 3 337 cycles per byte. On an STMicroelectronics STM32F4DISCOVERY development board running at 168 MHz, that equates to approximately 20 µs/byte or 50 KiB/s.

To the best of our knowledge, only one other implementation of KECCAK for an embedded device has been published.[6] A comparative listing is given in Table 5.1. Our implementation requires less ROM, making it the smallest published KECCAK implementation so far. However, the one by Van Assche and Van Keer[6] is faster.

The main reason for the speed difference is that the Atmel ATtiny45 microcontroller allows addressed access to the working registers. I.e. one can load data from `r3` by using address `r1[2]`. This is not possible in ARM Cortex-M microcontrollers, thus a lot of time in our implementation is spent loading and storing data in memory, which is addressable in that fashion, but is two to three times as slow as working registers.

Table 5.1: Comparison of our KECCAK implementations with other implementations of KECCAK-f[200] for embedded platforms. The RAM usage for our implementations is given as *static + while running KECCAK*. All listed implementations include required utility functions for absorbing, squeezing, padding, …

Implementation	Platform	ROM [bytes]	RAM [bytes]	Speed [cycles/byte]
Van Assche *et al.*[6]	ATtiny45	752	48	2 412
Ours	Cortex-M0	500	28 + 24	4 205
Ours	Cortex-M3/4	496	28 + 24	3 337

[6] J. Balasch *et al.*, "Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices" (2012).

5.3.5 PRNG verification

In order to verify our PRNG implementation, multiple experiments were conducted. For the first round of tests, we used three development boards, a Cortex-M0, -M3, and -M4F one. More specifically, the STMicroelectronics STM32F0-, STM32VL-, and STM32F4DISCOVERY boards, which contain respectively an STM32F051R8T6, STM32F100RB, and STM32F407VGT6 microcontroller. In all of these tests, 8 KiB RAM content was used to generate the PRNG seed. A picture of our test and demo setup is shown in Figure 5.3. A Texas Instruments Stellaris LaunchPad (Cortex-M4F) asynchronously controls the three boards, guaranteeing each board is power-cycled as fast as possible, in order to maximize the number of measurement outputs.

The first experiment was running the NIST, Diehard and Dieharder tests on the output of the PRNG. As can be expected from a cryptographic hash function, all boards passed these tests. These tests however, say very little about the quality of the PRNG design. Even if the initial seeding material contains zero entropy, then the tests would still succeed. This is because the output of Keccak, and that of other strong cryptographic hash functions, is not distinguishable from a random permutation.

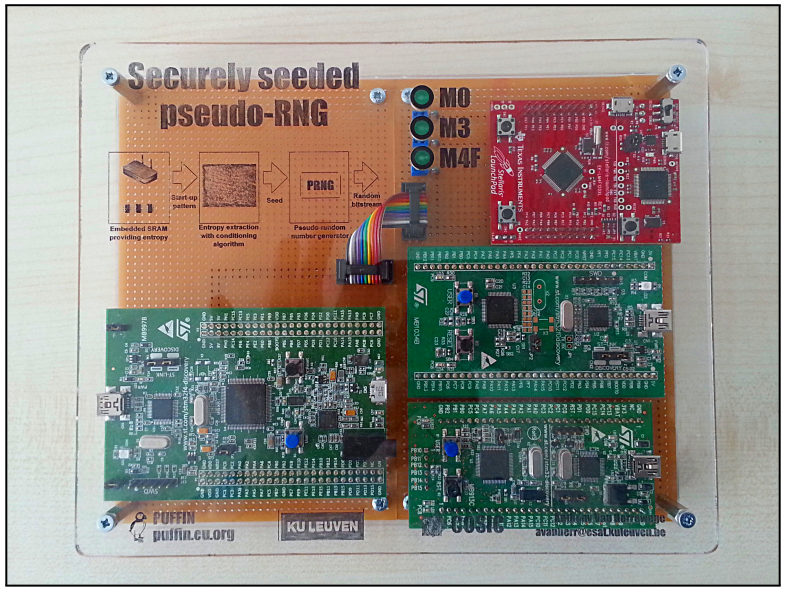


Figure 5.3: PRNG test and demo setup, containing STMicroelectronics STM32F0-, STM32VL-, and STM32F4DISCOVERY boards, asynchronously controlled by a Texas Instruments Stellaris LaunchPad.

The second set of experiments was conducted on our STM32F100R8 measurement board (see Chapter 4). As a first test, we checked whether the PRNG generates different numbers on every power-up. The microcontrollers were repeatedly power-cycled and the first generated 64-bit pseudo-random number was logged each time. The NIST test was run on the logged data sets, all of which passed the test.

As a second test, we used the thousands of SRAM measurements we had for each of the STM32F100R8 chips to generate a 64-bit number from the first 4 KiB of each measurement, using Keccak. This test is similar to the previous one, except that we use less SRAM data to generate the numbers, and that the processing was done on a PC. Just as for the previous test, this dataset also passed the NIST test.

For the two tests in the last experiment, we could not run the Diehard(er) tests, because generating sufficient data would take over 8 years, due to the time required for power cycling the microcontrollers.

5.3.6 Alternative implementations

Unfortunately, Keccak works over loops of size 5, which makes it impossible to store all bytes required in a sub-loop into a single 32-bit register. Also, because registers in an ARM Cortex-M are not addressable (meaning that there is no instruction to e.g. load data offset 7 bytes from register `r0` as `r0[7]`), we cannot make efficient use of the full 32-bit width of the registers.

A possible solution to this would be to implement Keccak-f[800], which uses 32-bit words. This would allow number generation approximately four times faster, at the trade-off of requiring approximately four times as much RAM. Code size could probably be reduced even further, because operations such as e.g. rotating a 32-bit word can be executed in a single instruction, whereas rotating an 8-bit word in a 32-bit register requires at least two instructions.

5.4 Conclusion

In this chapter we have presented a design for a securely seeded, software-based, strong PRNG with a security level of at least 128 bits. We use the entropy in a microcontroller's SRAM power-up data to generate a strong seed value for the PRNG. Such a design allows the implementation of strong PRNGs on commercial off-the-shelf microcontrollers, without requiring any extra external components. This type of design is valuable in those situations where the increased expense of a secure microcontroller is not economially feasible, yet strongly random data is required, e.g. set-top boxes and routers.

We implemented an extremely space optimized version of our PRNG design for the ARM Cortex-M0 and ARM Cortex-M3/4. The implementation is based on KECCAK, and has a security level of at least 128 bits. Only 496 bytes of ROM and 52 bytes of RAM are required. The ROM size includes initialization, hashing, padding and output generation functions, the KECCAK-f[200] permutation implementation itself requires only 400 bytes. The total time required to hash a byte, or generate a pseudo-random byte of output, is 3 337 cycles on the ARM Cortex-M3/4. To the best of our knowledge, our implementation of KECCAK is the smallest one available for any platform.

6 Conclusions

As the examples in Chapter 1 made clear, a myriad of embedded systems are severely lacking with respect to security. This problem is compounded by the fact that network-enabled devices are making an ever-increasing presence in our daily lives. Now, more than ever, cryptography is required in order to protect our data and privacy. Furthermore, companies are depending on it to implement digital rights management (DRM) schemes and secure updating of softwares. Physically unclonable functions (PUFs) are a recently discovered class of cryptographic primitives that can improve many low-level cryptographic building blocks required by all of these applications. In particular, they are very promising for secure key generation modules and random number generators (RNGs).

Chapter 2 is reserved mainly for background information on PUFs, as well as some information on random number generators. We discuss the various properties of PUFs which makes them unique as a cryptographic building block, and show how applications can benefit from the use of PUFs. An overview of some of the most important PUF types are presented, along with a discussion of what exactly lies at the origin of PUF-like behavior. Metrics by which the quality of a PUF can be ascertained are discussed. Next is the description of a basic protocol for error correction, which PUFs require, and its impact on the security afforded by PUFs. We also present a promising method for debiasing of PUF data. Finally, we show how one can harness PUFs for use in RNGs.

In Chapter 3 we present a practical design for an area efficient, PUF-based key generation module. By outfitting an electronic device with such a module, one can guarantee that each device will have a personal, secure key. Our design makes use of a novel ring-oscillator PUF and an efficient error correction scheme. It is in particular the design of a custom BCH decoding microcontroller which we study. By tailoring every part of the microcontroller specifically to the algorithms required for BCH decoding, we end up with the smallest implementation which has been published so far for such a decoder. This result disproves the prevalent belief that BCH decoders are unwieldy and complex. Our reference key generation module, which generates an 128-bit key with an failure rate of 10^{-9}, fits in less than 1200 Xilinx Spartan-6 FPGA slices. With this design, which is the first fully functional PUF-based key generator,

we prove that PUFs are a viable building block, even for small embedded devices.

Even though our design in Chapter 3 proves that PUF-based designs can be made very small and efficient, custom hardware is often not an option due to economic reasons. Therefore, we turn our attention to PUFs in commercial off-the-shelf (COTS) components in Chapter 4. More specifically, we investigate the SRAM power-up behavior in four important microcontroller families. These families, which have been selected due to their large market share, are the Atmel ATmega, the Microchip PIC16, the Texas Instruments MSP430, and the ARM Cortex-M. We discuss in detail various metrics, and the impact these have on practical usability of the embedded SRAM for either PUF or RNG designs. Our experiments show that there are large differences between the various microcontroller families: for some the SRAM behaves almost ideally for the intended designs, while in others very obvious start-up patterns make the SRAM all but unusable.

In Chapter 5, we continue our quest for secure PUF and RNG constructions which require nothing but a COTS microcontroller and some software. An efficient and simple design is presented for a securely seeded PRNG on a microcontroller. Generating a secure PRNG seed, while critical for strong cryptography, is often troublesome on embedded devices, which has lead to weak security in many devices. By exploiting the errors in SRAM on power-up as a source of entropy, our design generates strong randomness every time. We develop a proof of concept implementation on multiple ARM Cortex-M microcontrollers. At the core of this lies a tiny optimized Keccak implementation, at time of publication, the smallest in the world. Our proof of concept passes the NIST randomness test, which proves its merit.

Future work Although a significant amount of research has been conducted on PUFs in the past few years, several open problems still exist. For one, there have been no practical designs for a strong PUF, and indeed, it is not certain whether such a design exists. The invention of silicon-based strong PUF would allow the simplification of many protocols, and would also permit for more lightweight implementations.

If PUFs are ever to be used in high security systems, then they should resist side-channel attacks. Very little research concerning this has been published so far. Clearly, this is an area in which a lot of progress can still be made. Luckily, a lot of research has been done on these types of attacks in general, so it should be possible if not to completely copy, then at least to build upon existing counter-measure techniques. One of the main questions will be whether it is possible to design a side-channel resistant PUF with good properties that is not so big so as to make secure memory an attractive alternative.

A second important area of PUF research lies in improving error correcting techniques. Due to the construction of many PUF-based protocols, a certain amount of security is

lost due to the use of error correcting codes. By improving upon the existing methods, the required error correcting codes can be shortened. This will reduce the "security leakage", which will in turn lead to a reduction in required PUF real estate. Methods to deal with heavily biased PUFs are another part of this area of research. We presented an initial idea for a lightweight method which allows one to extract unbiased data efficiently from such PUFs, but more research is required in order to turn this into a full-fledged solution.

Almost all of the research directions mentioned above will invariably lead to new, custom hardware designs. Unfortunately, cryptography, as any other engineering discipline, is often a game of compromise. It is not always feasible to add custom hardware to a device, often due to economic reasons. We therefore believe it to be of utmost importance that research is done into PUFs in COTS semi-conductors. Even though such PUFs might not function as well as custom designed ones, their use could significantly improve the current, often non-existing, security measures in electronic devices. Interesting directions for research are the behavior of several on-chip oscillators or of analog-to-digital converters.

Discovery of such "COTS PUFs" will open the door to cheap, software-based PUF and PRNG designs. It is therefore also important that designs and algorithms are developed that make these PUFs practical for everyday use. Efficient implementations are necessary in order to convince manufacturers that improving the security of their devices is possible at a very low cost. Very often, it is unclear just how efficient one can implement a design, and which trade-offs will need to be made in order to reach certain goals. The importance of high-quality, practical work on cryptographic implementations can therefore not be overstated. Such applied research will help in bringing the benefits of PUFs to the masses, and not relegate them to an object of purely academic research, which, unfortunately, happens all too often with promising cryptographic designs.

Although it is beneficial that multiple manufacturers are adding (T)RNGs to their microcontrollers nowadays, it is often unclear from the microcontrollers' datasheet exactly how random numbers are generated and which, if any, kind of post-processing is applied to the output of the hardware RNG module. An independent review of the quality of these RNGs is required in order to establish how secure they are. Note that this not only applies to RNG modules in embedded microcontrollers, but also to such modules in desktop and server CPUs, e.g. Intel's DRNG module.

In light of this, it would be very beneficial if guidelines were be drawn up on how to practically implement secure software-based PUF and PRNG implementations for embedded devices. Such guidelines could help prevent many common mistakes which are still being made. A framework implementing such designs for the most commonly used microcontroller families would be very welcome as well. One particularly useful implementation that comes to mind is a PRNG for COTS microcontrollers that is

seeded using true randomness (e.g. such as our implementation in Chapter 5), and that continuously accumulates additional entropy from various sources on the device (e.g. available hardware RNG modules and/or clock jitter). Such an implementation could greatly improve the security of millions of embedded devices worldwide, many, if not most, of which are currently lacking in this department.

A Microcontroller Firmware

In this appendix chapter, we list the assembly code used for the firmware of each of the measured microcontrollers in Chapter 4. This in order to encourage a repeat of our tests.

Listing A.1: Assembly code for Atmel ATmega328P.

```
; Atmel ATmega328P firmware for SRAM readout on startup.
; Transmission in binary over UART at 38400 baud.

; Register name definitions
.nolist
#define _SFR_ASM_COMPAT 1
#include  <avr/io.h>
.list

; Define register names for readability
#define curAddr  X
#define curAddrH R27
#define curAddrL R26
#define curVal   R16
#define cmpAddrH R29
#define cmpAddrL R28

.section .text ; Start code segment
.org 0x00

Init:
  ; Setup clock for 8 Mhz
  ldi curVal, 0
  sts CLKPR, curVal ; Set prescaler to 1

  ; Disable watchdog timer
  ldi curVal, 0x00
  sts MCUSR, curVal  ; Clear all "powerup" reset flags
  sbr curVal, WDCE   ; Enable watchdog change enable bit
  sts WDTCSR, curVal ; Disable watchdog related settings
```

```
31
32    ; Setup UART
33    ldi curVal, 0x00
34    sts UBRR0H, curVal        ; Set baud rate
35    ldi curVal, 0x0c          ; 38 400 Hz
36    sts UBRR0L, curVal
37    ldi curVal, (1 << TXEN0)  ; Enable transmitter
38    sts UCSR0B, curVal
39    ldi curVal, (3 << UCSZ00) ; Set 8 data bits
40    sts UCSR0C, curVal
41
42    ; Setup IO
43    cbi _SFR_IO_ADDR(PORTB), PORTB0 ; Set B0 low (LED on)
44    sbi _SFR_IO_ADDR(DDRB), DDB0    ; B0 as output
45
46    ; Setup variables
47    ldi curAddrL, 0x00
48    ldi curAddrH, 0x01
49    ldi cmpAddrL, 0x00
50    ldi cmpAddrH, 0x09
51
52    ; Start transmission of SRAM content
53  DoTransmission:
54    ld  curVal, curAddr+  ; Indirect load and post-inc pointer
55    sts UDR0, curVal      ; Store value in transmit register
56
57  TransmissionWait:
58    lds  curVal, UCSR0A   ; Load UART status register
59    sbrs curVal, TXC0     ; Skip next instr if TX clear is set
60    rjmp TransmissionWait ; Jump back for some more idle waiting
61
62    ; Done with TX, clear flag by writing 1 to it (docs p. 194)
63    ori   curVal, (1 << TXC0)
64    sts   UCSR0A, curVal
65
66    ; Check if the last SRAM byte has been transmitted yet
67    cp   curAddrH, cmpAddrH ; Compare values
68    brlo DoTransmission     ; Branch if lower
69    cp   curAddrL, cmpAddrL ; Compare values
70    brlo DoTransmission     ; Branch if lower
71
72  LEDOff:
73   sbi _SFR_IO_ADDR(PORTB), PORTB0 ; Set B0 high (LED off)
74
75  BusyWait:
76    ; If we end up here, then everything has been transmitted
77    rjmp BusyWait
78
79  .end ; End of file
```

Listing A.2: Assembly code for Microchip PIC16F1825.

```
; Microchip PIC16F1825 firmware for SRAM reaout on startup.
; Transmission in binary over UART at 38400 baud, 8N/1.

#include <p16f1825.inc>
  processor 16f1825

  __CONFIG _CONFIG1, _FOSC_INTOSC & _WDTE_OFF & _PWRTE_ON & _MCLRE_OFF &
       _CP_OFF & _CPD_OFF & _BOREN_ON & _CLKOUTEN_OFF & _IESO_OFF &
       _FCMEN_OFF
  __CONFIG _CONFIG2, _WRT_OFF & _PLLEN_ON & _STVREN_ON & _BORV_HI &
       _LVP_OFF

; Helper macro
CMP_NEQ macro  CMP_REG, CMP_VALUE  ; 3/4 cycles
  movlw   CMP_VALUE
  subwf   CMP_REG, W
  btfss   STATUS, Z
  endm

; Use FSR1x registers as temporary storage
#define tempStore FSR1L
#define bankStore FSR1H

#define MaxAddrL  0xEF
#define MaxAddrH  0x23

RESET_VECTOR: org 0x00
  goto  MAIN

MAIN: org 0x05
  ; Setup INTOSC - 32 MHz
  banksel OSCCON
  movlw   b'11110000'
  movwf   OSCCON

  ; Setup ports
  banksel PORTA
  movlw   0xFF
  movwf   PORTA
  movwf   PORTC

  banksel ANSELA
  clrf    ANSELA
  clrf    ANSELC

  banksel TRISA
  clrf    TRISA
  clrf    TRISC

  ; Setup UART
  banksel SPBRGL
```

```
49    movlw   .51       ; Baud rate 38400 bps (0.2% error)
50    movwf   SPBRGL
51    clrf    SPBRGH
52
53    banksel TXSTA
54    movlw   b'00100100'  ; Set up UART
55    movwf   TXSTA
56
57    banksel RCSTA
58    movlw   b'10000000' ; Enable UART
59    movwf   RCSTA
60
61    ; Wait for oscillator to become stable
62    banksel OSCSTAT
63  WAIT_FOR_OSC:
64    btfsc   OSCSTAT, PLLR     ; Wait for PPL to become ready
65    btfss   OSCSTAT, HFIOFS   ; Wait for stable HF int. osc
66    goto    WAIT_FOR_OSC
67
68    ;; Start reading out SRAM ;;
69    ; Setup FSR for readout of bank 0 (incl. common RAM)
70    banksel FSR0L
71    movlw   0x20
72    movwf   FSR0L
73    clrf    FSR0H
74
75    moviw   --FSR0  ; Decrement pointer by 1
76
77  READOUT_BANK_0:
78    moviw   ++FSR0          ; Inc pointer & move value to W
79    call    TRANSMIT_HEX    ; Transmit value
80    clrf    INDF0           ; Clear value
81
82    CMP_NEQ FSR0L, 0x7F     ; Skip if at end of bank 0
83    goto    READOUT_BANK_0
84
85    ; Setup FSR for readout of bank 1 & up
86    banksel FSR0L   ; Setup FSR0 to point to linear memory
87    movlw   0x50
88    movwf   FSR0L
89    movlw   0x20
90    movwf   FSR0H
91
92    moviw   --FSR0    ; Decrement pointer by 1
93
94  READOUT_RAM:
95    moviw   ++FSR0   ; Inc pointer & move value to W
96    call    TRANSMIT ; Transmit value
97    clrf    INDF0    ; Clear value
98
99    ; Check if maximum value is reached (without using RAM!)
100   CMP_NEQ FSR0L, MaxAddrL ; Check LSB
101   goto    READOUT_RAM
```

```
102
103   CMP_NEQ FSR0H, MaxAddrH ; Check MSB
104   goto    READOUT_RAM
105
106 DONE:
107   goto  DONE
108
109 TRANSMIT:
110   banksel TXREG
111   movwf   TXREG
112   banksel TXSTA         ; Go to correct bank
113 TRANSMIT_TEST:
114   btfss   TXSTA, TRMT   ; Skip if TX register is empty
115   goto    TRANSMIT_TEST ; Busy wait until transmitted
116   return                ; Return to caller function
117
118   end
```

Listing A.3: Assembly code for STMicroelectronics STM32F100R8.

```
1  /* STMicro STM32F100 firmware for SRAM readout on startup. */
2  /* Transmission in binary over UART at 115200 baud, 8N/1.  */
3    .syntax unified
4    .cpu cortex-m3
5    .fpu softvfp
6    .thumb
7
8  .global g_pfnVectors
9  .global SystemInit_ExtMemCtl_Dummy
10 .global Default_Handler
11
12 /* Memory locations for linker script. */
13 .word _sidata
14 .word _sdata
15 .word _edata
16 .word _sbss
17 .word _ebss
18
19 .equ  BootRAM, 0xF108F85F
20   .section  .text.Reset_Handler
21   .weak Reset_Handler
22   .type Reset_Handler, %function
23 Reset_Handler:
24   ldr  r0, =0x40021000 /* RCC_BASE */
25   ldr  r1, =0x00004009 /* PB + AF + USART1 enable */
26   str  r1, [r0, #0x18] /* Store r1 in RCC_APB2ENR */
27
28   ldr  r0, =0x40010C00 /* GPIOB_BASE */
29   ldr  r1, =0x4B444444 /* PB6 = AF @ 50MHz, rest floats */
30   str  r1, [r0, #0]    /* Store r1 in GPIOB_CRL */
31
32   ldr  r0, =0x40010000 /* AFIO_BASE = remapping register */
```

```
  ldr  r1, =0x00000004 /* USART1 on PB6/7 */
  str  r1, [r0, #4]    /* Store in AFIO_MAPR */

  ldr  r0, =0x40013800 /* UART1 */
  movs r1, #0
  strh r1, [r0, #4]    /* +4 USART_DR */
  strh r1, [r0, #16]   /* +16 USART_CR2 = 0 */
  strh r1, [r0, #24]   /* +24 USART_GTPR = 0 - Prescaler */
  movs r1, #69         /* 8MHz / 69 == 115200 */
  strh r1, [r0, #8]    /* +8 USART_BR */
  movw r1, #0x200C     /* 8-bit, no parity, enable TX */
  strh r1, [r0, #12]   /* +12 USART_CR1 */

  ldr  r2, =0x2000     /* Size = Length (8K) */
  ldr  r3, =0x20000000 /* Mem = RAM Address */
TXLoop:
  ldrh r1, [r0, #0]    /* USART->SR */
  ands r1, #0x80       /* TXE */
  beq  TXLoop
  ldrb r1, [r3], #1    /* [Mem++] */
  strh r1, [r0, #4]    /* USART->DR */
  subs r2, r2, #1      /* Size-- */
  bne  TXLoop

main:  b main          /* Infinite loop */

.size Reset_Handler, .-Reset_Handler
```

Listing A.4: Assembly code for Texas Instruments MSP430F5308.

```
; TI MSP430 firmware for SRAM reaout on startup.
; Transmission in binary over UART at 115200 baud, 8N/1.

; Load processor definitions in C-style
#include  <msp430.h>

; Variable name definition
#define Status    R2
#define ptrSRAM   R4

  .text           ; Program start
ISR_Reset:
  mov.w #(WDTPW + WDTHOLD), &WDTCTL   ; Stop watchdog timer

Port_Setup:
  bis.b #(BIT0 + BIT6), &P1OUT ; Set P1.0 & 1.6 high (LEDs on)
  bis.b #(BIT0 + BIT6), &P1DIR ; Set P1.0 & 1.6 as output

  bic.b #BIT6, &P1OUT ; Disable green LED

  bis.b #(BIT1 + BIT2), &P1SEL  ; Set P1.1 & 1.2 to RX/TX
  bis.b #(BIT1 + BIT2), &P1SEL2 ; Set P1.1 & 1.2 to RX/TX
```

```
23
24    bis.b #BIT4, &P1DIR ; P1.4 output
25    bis.b #BIT4, &P1SEL ; P1.4 output SMCLK
26
27  Setup_DCO:
28    clr.b &DCOCTL                 ; Set DCO to lowest setting
29    mov.b &CALBC1_12MHZ, &BCSCTL1 ; Set range
30    mov.b &CALDCO_12MHZ, &DCOCTL  ; Set DCO step + modulation
31
32  UART_Setup:
33    ; Keep UART in reset and select SMCLK
34    bis.b #(UCSSEL1 + UCSWRST), &UCA0CTL1
35
36    ; 115.2 kbps @ 11.0592 MHz
37    mov.b #0x60, &UCA0BR0
38    mov.b #0x00, &UCA0BR1
39    mov.b #0x00, &UCA0MCTL
40
41    bic.b #UCSWRST, &UCA0CTL1 ; Enable UART
42
43  SRAM_Prepare:
44    mov.w #0x0200, ptrSRAM ; Load SRAM base addres
45
46  SRAM_Transmit:
47    mov.b @ptrSRAM+, &UCA0TXBUF ; Transmit value & inc ptr
48
49  1:
50    bit.b #UCA0TXIFG, &IFG2 ; Check if TX int flag is set
51    jz    1b                ; If not ready yet, wait some more
52
53    cmp.w #0x0400, ptrSRAM ; Calculate diff(ptrSRAM, endSRAM + 1)
54    jnz   SRAM_Transmit    ; Not finished, so continue
55
56  LEDs_End:
57    bis.b #BIT6, &P1OUT ; Turn on green LED
58    bic.b #BIT0, &P1OUT ; Turn off red LED
59
60  Endless_Loop:
61    jmp   Endless_Loop ; Endless loop
```

B Microcontroller Metrics

In this chapter, we list various metrics for the SRAM power-up values of each of the individual microcontrollers which were read out as part of the study presented in Chapter 4. The metrics are listed here in the same order as they are discussed in that chapter.

Each table contains the following columns (in order): IC identifier (ID), sample mean ($\overline{X}$), sample standard deviation (s), median (Md), first quartile (Q_1), third quartile (Q_3), minimum (X_{min}), and maximum (X_{max}).

For metrics where data is available for multiple ICs, the last row in each table, identified by Σ, is an aggregated result, calculated using the mean for each IC. This row is used to generate the plots in Chapter 4. A missing IDs indicates that the IC with that particular identifier did not function correctly (e.g. due to lifted traces on the measurement PCB), and is thus not included in our measurements.

Note that all metrics are given as fractional values in order to improve comparability between the metrics of different microcontrollers. Thus, to get the actual result in bits for a metric, has to multiply by the total number of bits in SRAM for that particular microcontroller. As a refresher, those amounts are listed in Table B.1.

Table B.1: Number of SRAM bits for each measured microcontroller.

Microcontroller	SRAM bits
Atmel ATmega328P	16 384
Microchip PIC16F1825	8 192
Texas Instruments MSP430F5308	49 152
STMicroelectronics STM32F100R8	65 536

B.1 Hamming weight

Table B.2: Frac. HW for ATmega328P ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	64.52	0.11	64.53	64.45	64.60	64.11	64.85
2	67.68	0.11	67.69	67.60	67.77	67.30	68.00
3	67.21	0.17	67.21	67.14	67.29	59.69	67.52
4	66.62	0.11	66.62	66.55	66.69	66.23	66.97
5	66.14	0.11	66.14	66.06	66.21	65.74	66.44
6	66.96	0.12	66.96	66.88	67.04	66.52	67.34
7	67.42	0.11	67.42	67.34	67.49	67.05	67.76
8	66.84	0.12	66.84	66.75	66.92	66.44	67.18
9	67.29	0.13	67.30	67.21	67.38	66.87	67.66
10	67.10	0.11	67.10	67.02	67.18	66.74	67.45
11	65.33	0.11	65.34	65.26	65.41	64.94	65.75
12	64.44	0.12	64.45	64.37	64.53	64.06	64.81
13	66.70	0.12	66.71	66.61	66.79	66.30	67.06
14	67.29	0.12	67.29	67.21	67.37	66.90	67.63
15	68.42	0.12	68.43	68.34	68.51	67.93	68.77
16	68.60	0.12	68.60	68.51	68.68	68.20	68.94
Σ	66.79	1.19	67.03	66.50	67.32	64.44	68.60

Table B.3: Frac. HW for ATmega328P ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	70.12	0.08	70.12	70.06	70.17	69.84	70.45
2	74.77	0.09	74.77	74.71	74.83	74.47	75.06
3	73.08	0.08	73.08	73.03	73.14	72.78	73.34
4	73.44	0.09	73.44	73.38	73.50	72.64	73.74
5	73.03	0.09	73.04	72.97	73.09	72.73	73.41
6	72.47	0.10	72.47	72.41	72.53	72.17	72.80

Continued on next page

Table B.3 - Cont.: Frac. HW for ATmega328P ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
7	72.49	0.08	72.49	72.44	72.55	71.97	72.77
8	73.17	0.09	73.17	73.11	73.23	72.85	73.54
9	74.66	0.09	74.66	74.60	74.73	74.35	74.96
10	73.92	0.09	73.92	73.87	73.98	73.57	74.26
11	70.41	0.08	70.41	70.36	70.47	70.04	70.68
12	70.87	0.09	70.87	70.81	70.92	69.89	71.14
13	73.33	0.09	73.33	73.27	73.39	73.00	73.65
14	74.22	0.09	74.22	74.16	74.27	73.94	74.60
15	75.72	0.09	75.72	75.66	75.78	75.40	76.09
16	76.18	0.09	76.18	76.12	76.23	75.85	76.48
Σ	73.24	1.74	73.25	72.49	74.33	70.12	76.18

Table B.4: Frac. HW for ATmega328P ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
1	59.40	0.09	59.40	59.34	59.47	59.07	59.74
2	62.25	0.09	62.25	62.19	62.31	61.96	62.54
3	61.87	0.09	61.87	61.80	61.93	61.50	62.16
4	60.95	0.09	60.95	60.88	61.01	60.62	61.26
5	60.69	0.10	60.69	60.63	60.75	60.36	61.02
6	61.24	0.10	61.24	61.18	61.30	60.90	61.54
7	61.50	0.09	61.50	61.44	61.57	61.13	61.81
8	60.87	0.10	60.88	60.81	60.94	60.54	61.16
9	61.07	0.09	61.07	61.01	61.13	60.65	61.37
10	61.76	0.10	61.76	61.69	61.82	61.44	62.09
11	59.81	0.10	59.81	59.75	59.88	59.47	60.18
12	58.39	0.10	58.39	58.32	58.45	58.06	58.77
13	60.77	0.09	60.77	60.71	60.83	60.46	61.04

Continued on next page

Table B.4 - Cont.: Frac. HW for ATmega328P ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
14	61.82	0.09	61.82	61.76	61.88	61.48	62.14
15	62.60	0.09	62.60	62.54	62.66	62.27	62.93
16	62.79	0.09	62.79	62.72	62.85	62.43	63.09
Σ	61.11	1.17	61.16	60.75	61.83	58.39	62.79

Table B.5: Frac. HW for PIC16F1825 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	47.88	0.33	47.89	47.82	47.95	30.71	48.32
2	45.74	0.44	45.57	45.36	46.18	44.85	46.77
3	50.28	0.09	50.28	50.21	50.34	49.93	50.66
4	50.04	0.10	50.04	49.98	50.11	49.72	50.35
5	51.00	0.10	51.00	50.94	51.06	50.68	51.31
6	50.31	0.10	50.32	50.24	50.38	50.01	50.67
7	48.59	0.10	48.60	48.52	48.66	48.27	48.95
8	49.06	0.10	49.06	48.99	49.12	48.72	49.39
9	51.08	0.09	51.07	51.01	51.15	50.71	51.38
10	51.44	0.09	51.44	51.37	51.50	51.12	51.79
11	49.11	0.10	49.11	49.05	49.17	48.77	49.41
12	49.33	0.11	49.33	49.26	49.40	49.01	49.70
13	51.31	0.09	51.31	51.25	51.38	50.98	51.61
14	51.50	0.10	51.50	51.43	51.56	51.16	51.87
15	49.28	0.09	49.28	49.22	49.34	48.99	49.59
16	49.46	0.09	49.46	49.40	49.52	49.13	49.76
Σ	49.71	1.52	49.75	49.10	51.02	45.74	51.50

Table B.6: Frac. HW for PIC16F1825 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	44.03	2.14	44.85	44.75	44.91	37.78	47.57
2	43.96	2.11	44.75	44.65	44.82	37.74	50.04
3	43.27	2.78	44.32	44.23	44.40	35.14	50.15
4	45.82	1.92	46.52	46.44	46.58	40.05	50.05
5	45.08	2.16	45.90	45.81	45.97	38.76	46.28
6	43.18	2.58	44.17	44.08	44.24	35.62	44.52
7	43.56	2.18	44.39	44.30	44.46	37.11	48.54
8	44.86	1.97	45.61	45.52	45.68	39.16	48.68
9	45.31	1.10	45.52	45.45	45.58	38.40	51.03
10	46.09	0.28	46.11	46.05	46.17	43.47	51.73
11	43.58	0.49	43.63	43.57	43.69	38.75	49.68
12	46.50	0.15	46.51	46.45	46.57	45.37	49.15
13	45.62	0.36	45.65	45.58	45.72	42.60	51.61
14	45.96	0.20	45.96	45.89	46.03	45.14	51.81
15	43.89	0.26	43.91	43.82	43.99	42.43	48.99
16	50.29	0.10	50.29	50.26	50.33	50.22	50.37
Σ	45.06	1.77	44.97	43.81	45.85	43.18	50.29

Table B.7: Frac. HW for PIC16F1825 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	48.29	0.13	48.29	48.21	48.38	47.83	48.76
2	49.80	0.12	49.81	49.72	49.88	49.43	50.20
3	50.14	0.11	50.15	50.06	50.22	49.81	50.49
4	49.81	0.11	49.82	49.74	49.88	49.41	50.18
5	51.33	0.12	51.33	51.25	51.40	50.94	51.69

Continued on next page

Table B.7 - Cont.: Frac. HW for PIC16F1825 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
6	50.51	0.11	50.51	50.43	50.59	50.16	50.90
7	48.77	0.12	48.77	48.68	48.85	48.35	49.18
8	49.33	0.14	49.33	49.23	49.43	48.89	49.76
9	50.87	0.13	50.87	50.78	50.96	50.49	51.27
10	51.44	0.11	51.44	51.37	51.51	51.11	51.87
11	49.25	0.11	49.24	49.17	49.33	48.84	49.61
12	49.68	0.12	49.68	49.61	49.77	49.26	50.13
13	51.14	0.11	51.14	51.06	51.21	50.73	51.49
14	51.54	0.12	51.54	51.47	51.62	51.12	51.90
15	49.35	0.13	49.35	49.26	49.44	48.94	49.72
16	50.09	0.13	50.07	50.00	50.16	49.68	50.89
Σ	50.08	0.98	49.95	49.34	50.94	48.29	51.54

Table B.8: Frac. HW for MSP430F5308 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	64.84	0.06	64.84	64.80	64.89	64.61	65.09
2	62.49	0.06	62.49	62.45	62.53	62.27	62.69
3	66.48	0.06	66.48	66.43	66.52	66.25	66.68
4	61.80	0.06	61.80	61.76	61.85	61.58	62.03
5	60.80	0.06	60.80	60.75	60.84	60.60	61.03
6	62.36	0.06	62.36	62.32	62.40	62.11	62.59
7	60.20	0.07	60.19	60.15	60.24	59.92	60.42
8	60.60	0.07	60.60	60.56	60.65	60.36	60.82
9	65.78	0.06	65.79	65.74	65.83	65.54	65.97
10	60.04	0.06	60.04	59.99	60.08	59.81	60.29

Continued on next page

Table B.8 - Cont.: Frac. HW for MSP430F5308 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
12	65.91	0.06	65.91	65.87	65.95	65.72	66.17
13	61.11	0.06	61.11	61.06	61.15	60.89	61.37
14	65.81	0.06	65.81	65.77	65.86	65.61	66.05
15	63.05	0.39	63.05	63.01	63.10	42.05	63.28
16	65.16	0.06	65.16	65.12	65.21	65.00	65.35
Σ	63.10	2.35	62.49	60.95	65.47	60.04	66.48

Table B.9: Frac. HW for MSP430F5308 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	63.50	0.32	63.56	63.42	63.65	61.88	73.63
2	60.86	1.00	60.72	60.64	60.81	60.44	77.33
3	65.25	0.32	65.26	65.17	65.32	64.83	75.86
4	60.56	0.48	60.63	60.40	60.95	58.11	62.53
5	59.56	0.44	59.70	59.47	59.81	56.77	61.73
6	60.02	1.37	59.58	59.53	59.75	59.29	74.03
7	58.36	0.65	58.58	58.25	58.74	54.23	60.99
8	58.82	0.61	59.03	58.70	59.18	55.22	61.46
9	67.62	3.09	66.24	65.71	68.18	65.27	79.93
10	57.06	1.62	56.67	56.55	56.81	56.33	73.16
12	64.98	1.22	64.72	64.66	64.79	64.51	79.65
13	58.27	0.83	58.33	57.94	58.55	57.15	70.11
14	65.37	1.14	65.14	65.06	65.26	63.13	79.71
15	61.52	0.47	61.59	61.40	61.69	60.44	70.48
16	64.75	1.61	64.23	64.14	64.60	63.95	79.16
Σ	61.77	3.23	60.86	59.19	64.87	57.06	67.62

Table B.10: Frac. HW for MSP430F5308 ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	64.77	0.07	64.77	64.73	64.82	64.54	65.01
2	61.77	0.07	61.77	61.72	61.81	61.54	62.10
3	66.40	0.07	66.40	66.35	66.45	66.09	66.66
4	60.85	0.07	60.85	60.81	60.90	60.63	61.14
5	59.80	0.07	59.80	59.75	59.85	59.55	60.03
6	61.81	0.07	61.81	61.77	61.86	61.56	62.05
7	59.25	0.07	59.25	59.21	59.30	58.96	59.51
8	59.65	0.07	59.65	59.60	59.70	59.43	59.93
9	65.19	0.07	65.19	65.15	65.24	64.95	65.40
10	58.60	0.07	58.60	58.55	58.64	58.36	58.85
12	65.36	0.07	65.36	65.31	65.41	65.11	65.58
13	60.16	0.07	60.16	60.11	60.21	59.90	60.40
14	65.55	0.07	65.55	65.50	65.60	65.31	65.79
15	62.37	0.07	62.37	62.32	62.42	62.10	62.59
16	64.74	0.40	64.75	64.70	64.79	43.37	64.95
Σ	62.42	2.68	61.81	59.98	64.98	58.60	66.40

Table B.11: Frac. HW for STM32F100R8 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	49.49	0.07	49.49	49.44	49.54	49.23	49.70
6	49.15	0.07	49.15	49.11	49.20	48.90	49.39
8	49.84	0.07	49.84	49.79	49.89	49.59	50.11
9	49.64	0.07	49.64	49.59	49.68	49.41	49.89
10	49.49	0.07	49.49	49.45	49.53	49.22	49.73
11	49.36	0.07	49.36	49.31	49.40	49.10	49.58
12	49.39	0.07	49.39	49.34	49.43	49.16	49.66
13	49.29	0.07	49.29	49.24	49.33	49.01	49.50
14	49.73	0.07	49.73	49.69	49.78	49.50	49.99

Continued on next page

Table B.11 - Cont.: Frac. HW for STM32F100R8 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
15	49.58	0.07	49.59	49.54	49.63	49.37	49.82
16	49.52	0.07	49.52	49.48	49.57	49.29	49.80
Σ	49.50	0.20	49.49	49.37	49.61	49.15	49.84

Table B.12: Frac. HW for STM32F100R8 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	49.44	0.06	49.44	49.40	49.49	49.12	49.72
6	49.08	0.06	49.09	49.04	49.13	48.85	49.31
8	49.93	0.06	49.93	49.89	49.98	49.69	50.17
9	49.64	0.06	49.64	49.60	49.69	49.40	49.86
10	49.42	0.06	49.42	49.38	49.46	49.18	49.65
11	49.42	0.06	49.42	49.38	49.46	49.21	49.64
12	49.48	0.06	49.48	49.44	49.53	49.27	49.71
13	49.22	0.06	49.22	49.18	49.26	49.02	49.45
14	49.83	0.06	49.83	49.79	49.88	49.63	50.05
15	49.58	0.06	49.58	49.54	49.63	49.37	49.80
16	49.57	0.06	49.57	49.52	49.61	49.36	49.81
Σ	49.51	0.25	49.48	49.42	49.61	49.08	49.93

Table B.13: Frac. HW for STM32F100R8 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	49.69	0.07	49.69	49.65	49.74	49.46	49.90
6	49.20	0.07	49.21	49.16	49.25	48.97	49.43
8	49.70	0.07	49.70	49.66	49.75	49.45	49.93
9	49.67	0.07	49.67	49.62	49.72	49.45	49.90
10	49.50	0.07	49.50	49.45	49.55	49.30	49.71

Continued on next page

Table B.13 - Cont.: Frac. HW for STM32F100R8 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
11	49.34	0.07	49.34	49.30	49.39	49.08	49.58
12	49.33	0.07	49.33	49.28	49.38	49.10	49.64
13	49.29	0.07	49.29	49.24	49.33	49.04	49.53
14	49.67	0.07	49.67	49.62	49.71	49.44	49.91
15	49.72	0.07	49.72	49.68	49.77	49.52	49.92
16	49.65	0.07	49.65	49.60	49.70	49.46	49.89
Σ	49.52	0.20	49.65	49.34	49.68	49.20	49.72

B.2 Inter-device distance

Table B.14: Frac. inter-device distance for measured microcontrollers.

Microcontroller	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
ATmega328P	44.31	0.63	44.32	43.88	44.78	42.43	46.66
PIC16F1825	21.29	0.93	21.16	20.72	21.75	18.70	24.54
MSP430F5308	46.35	0.84	46.41	45.78	46.99	44.06	48.48
STM32F100R8	47.69	0.25	47.72	47.53	47.87	46.63	48.63

B.3 Intra-device distance

Table B.15: Frac. intra-device distance for ATmega328P ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	2.37	0.10	2.37	2.30	2.44	1.90	2.84
2	2.22	0.10	2.22	2.16	2.29	1.78	2.64
3	2.29	0.10	2.30	2.23	2.36	1.85	2.73
4	2.30	0.10	2.30	2.23	2.36	1.85	2.71

Continued on next page

Table B.15 - Cont.: Frac. intra-device distance for ATmega328P ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
5	2.30	0.10	2.30	2.24	2.37	1.86	2.72
6	2.34	0.10	2.34	2.27	2.40	1.88	2.76
7	2.31	0.10	2.31	2.24	2.37	1.90	2.75
8	2.33	0.10	2.33	2.26	2.40	1.89	2.79
9	2.29	0.10	2.28	2.22	2.35	1.83	2.76
10	2.25	0.09	2.25	2.19	2.31	1.76	2.75
11	2.46	0.10	2.46	2.39	2.53	2.01	2.97
12	2.57	0.10	2.57	2.50	2.64	2.13	3.05
13	2.28	0.10	2.28	2.21	2.34	1.85	2.70
14	2.16	0.09	2.16	2.09	2.22	1.73	2.58
15	2.18	0.09	2.18	2.12	2.25	1.75	2.64
16	2.26	0.10	2.26	2.19	2.32	1.83	2.68
Σ	2.31	0.10	2.29	2.26	2.33	2.16	2.57

Table B.16: Frac. intra-device distance for ATmega328P ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
1	7.46	0.13	7.46	7.37	7.54	6.89	8.04
2	9.82	0.16	9.83	9.72	9.92	9.17	10.47
3	7.97	0.13	7.97	7.89	8.06	7.31	8.67
4	9.52	0.17	9.53	9.41	9.63	7.76	10.16
5	9.24	0.15	9.24	9.14	9.35	8.55	9.94
6	7.96	0.23	7.92	7.80	8.10	7.20	8.96
7	8.41	0.16	8.42	8.31	8.51	6.03	9.02
8	9.67	0.21	9.68	9.53	9.81	8.71	10.59
9	11.54	0.20	11.55	11.41	11.67	10.71	12.35
10	9.32	0.15	9.32	9.22	9.42	8.61	9.92
11	7.00	0.13	7.00	6.91	7.09	6.36	7.56

Continued on next page

Table B.16 - Cont.: Frac. intra-device distance for ATmega328P ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
12	9.81	0.13	9.80	9.72	9.89	9.23	10.43
13	9.11	0.16	9.11	9.00	9.22	8.37	9.81
14	9.72	0.16	9.71	9.61	9.83	9.11	10.63
15	10.92	0.19	10.93	10.79	11.05	10.13	11.90
16	12.34	0.17	12.35	12.24	12.46	11.52	13.14
Σ	9.36	1.43	9.42	8.30	9.81	7.00	12.34

Table B.17: Frac. intra-device distance for ATmega328P ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	6.40	0.14	6.40	6.31	6.49	5.77	6.95
2	6.47	0.13	6.47	6.38	6.56	5.85	7.09
3	6.43	0.14	6.43	6.34	6.52	5.71	7.04
4	6.88	0.14	6.88	6.78	6.97	6.13	7.51
5	6.66	0.14	6.66	6.57	6.75	5.94	7.26
6	6.74	0.14	6.74	6.65	6.84	6.10	7.33
7	6.87	0.14	6.87	6.78	6.96	6.27	7.51
8	7.01	0.14	7.01	6.91	7.10	6.36	7.65
9	7.08	0.15	7.09	6.98	7.18	6.38	7.76
10	6.52	0.14	6.52	6.42	6.61	5.91	7.07
11	6.66	0.14	6.67	6.57	6.76	6.02	7.29
12	7.10	0.14	7.10	7.01	7.20	6.40	7.72
13	6.95	0.14	6.95	6.85	7.04	6.39	7.61
14	6.38	0.14	6.38	6.29	6.48	5.78	6.95
15	6.72	0.14	6.72	6.62	6.81	6.07	7.29
16	6.76	0.15	6.76	6.67	6.87	6.16	7.52
Σ	6.73	0.24	6.73	6.50	6.89	6.38	7.10

Table B.18: Frac. intra-device distance for PIC16F1825 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
1	1.63	0.12	1.64	1.55	1.71	1.05	2.20
2	1.95	0.39	1.92	1.59	2.30	0.99	3.20
3	1.49	0.11	1.49	1.42	1.56	0.98	2.04
4	1.53	0.11	1.53	1.45	1.60	0.99	2.03
5	1.55	0.11	1.55	1.48	1.62	1.04	2.08
6	1.56	0.12	1.56	1.48	1.64	1.07	2.11
7	1.58	0.11	1.58	1.50	1.66	1.09	2.21
8	1.75	0.12	1.75	1.66	1.83	1.23	2.32
9	1.39	0.11	1.39	1.32	1.47	0.92	1.93
10	1.44	0.11	1.44	1.37	1.51	0.96	1.97
11	1.49	0.11	1.49	1.42	1.56	0.99	2.00
12	1.76	0.12	1.76	1.69	1.84	1.20	2.38
13	1.50	0.11	1.50	1.42	1.58	0.98	2.01
14	1.45	0.11	1.45	1.38	1.53	0.96	1.94
15	1.39	0.11	1.39	1.32	1.47	0.95	1.97
16	1.44	0.11	1.44	1.37	1.51	0.89	1.92
Σ	1.56	0.15	1.51	1.45	1.59	1.39	1.95

Table B.19: Frac. intra-device distance for PIC16F1825 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
1	6.77	1.42	6.27	6.18	6.40	5.64	11.55
2	10.55	1.49	10.18	9.67	10.41	6.27	15.25
3	7.95	2.32	7.13	7.03	7.24	6.51	15.64
4	6.27	1.47	5.75	5.65	5.86	3.03	11.05
5	7.10	1.75	6.48	6.38	6.59	5.85	12.71

Continued on next page

Table B.19 - Cont.: Frac. intra-device distance for PIC16F1825 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
6	8.35	2.26	7.50	7.40	7.62	2.58	15.21
7	7.20	1.64	6.62	6.52	6.74	5.96	12.42
8	6.80	1.46	6.25	6.15	6.38	5.62	11.35
9	7.07	0.90	6.91	6.82	7.00	6.32	13.17
10	6.48	0.26	6.46	6.37	6.54	2.45	8.75
11	6.90	0.28	6.87	6.79	6.96	6.31	10.91
12	5.76	0.16	5.75	5.66	5.85	2.93	6.70
13	7.30	0.25	7.28	7.18	7.36	6.62	9.88
14	6.88	0.19	6.89	6.80	6.97	2.37	7.67
15	7.43	0.23	7.41	7.31	7.53	3.06	8.78
16	3.05	0.10	3.05	2.99	3.11	2.65	3.33
Σ	6.99	1.50	6.99	6.70	7.34	3.05	10.55

Table B.20: Frac. intra-device distance for PIC16F1825 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	$X_{\min}$	$X_{\max}$
1	4.18	0.16	4.18	4.07	4.29	3.46	4.92
2	7.90	0.45	8.03	7.47	8.28	6.60	9.05
3	3.32	0.14	3.32	3.22	3.42	2.60	4.04
4	4.12	0.16	4.11	4.00	4.22	3.43	4.85
5	3.73	0.15	3.74	3.64	3.83	3.00	4.54
6	3.96	0.14	3.97	3.87	4.05	3.22	4.79
7	4.16	0.15	4.16	4.07	4.26	3.47	4.88
8	4.61	0.15	4.61	4.50	4.71	3.91	5.36
9	3.76	0.14	3.76	3.66	3.86	3.04	4.43
10	3.48	0.15	3.48	3.38	3.58	2.84	4.18
11	3.72	0.14	3.72	3.63	3.82	3.09	4.40

Continued on next page

Table B.20 - Cont.: Frac. intra-device distance for PIC16F1825 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
12	4.39	0.14	4.38	4.29	4.48	3.69	5.04
13	3.78	0.14	3.77	3.69	3.87	3.15	4.41
14	3.75	0.14	3.75	3.65	3.85	3.13	4.46
15	4.00	0.14	4.00	3.91	4.09	3.31	4.64
16	3.99	0.13	3.99	3.89	4.08	3.35	4.57
Σ	4.18	1.04	3.98	3.74	4.16	3.32	7.90

Table B.21: Frac. intra-device distance for MSP430F5308 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	4.20	0.08	4.20	4.15	4.25	3.85	4.55
2	4.12	0.08	4.12	4.07	4.17	3.72	4.49
3	4.25	0.08	4.25	4.20	4.30	3.86	4.60
4	4.35	0.08	4.35	4.30	4.40	3.98	4.69
5	4.34	0.08	4.34	4.29	4.40	3.98	4.70
6	4.28	0.08	4.28	4.23	4.33	3.93	4.60
7	4.46	0.08	4.46	4.41	4.51	4.10	4.82
8	4.56	0.08	4.56	4.50	4.61	4.20	4.95
9	4.03	0.07	4.03	3.98	4.08	3.68	4.37
10	4.30	0.08	4.30	4.24	4.35	3.95	4.66
12	4.17	0.07	4.17	4.11	4.22	3.82	4.49
13	4.40	0.08	4.40	4.35	4.45	4.07	4.76
14	4.10	0.08	4.10	4.05	4.15	3.76	4.47
15	4.32	0.08	4.32	4.27	4.38	3.93	4.68
16	4.16	0.08	4.16	4.11	4.21	3.80	4.54
Σ	4.27	0.14	4.28	4.16	4.34	4.03	4.56

Table B.22: Frac. intra-device distance for MSP430F5308 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	7.55	0.91	7.41	7.30	7.59	6.88	27.25
2	9.01	1.94	8.34	8.09	9.04	7.54	29.83
3	7.25	0.38	7.17	7.08	7.31	6.66	14.61
4	7.45	0.55	7.36	7.26	7.50	5.86	17.66
5	7.64	0.76	7.48	7.38	7.64	6.96	16.68
6	10.89	1.98	10.23	9.64	11.42	8.89	25.35
7	8.12	0.89	7.90	7.78	8.14	6.20	22.42
8	8.20	0.94	7.99	7.86	8.22	7.40	20.37
9	11.32	4.12	9.56	8.57	12.54	7.75	30.42
10	11.65	2.84	10.58	9.92	12.20	9.13	32.60
12	8.49	2.11	7.83	7.62	8.37	7.17	26.76
13	9.01	1.75	8.40	8.15	9.09	6.14	27.05
14	8.33	1.39	7.88	7.67	8.34	6.93	22.51
15	7.81	1.13	7.56	7.44	7.78	5.96	36.49
16	9.00	1.69	8.39	8.02	9.17	7.48	23.00
Σ	8.78	1.42	8.33	7.73	9.01	7.25	11.65

Table B.23: Frac. intra-device distance for MSP430F5308 ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
1	6.47	0.09	6.47	6.41	6.54	6.05	7.07
2	6.20	0.09	6.20	6.14	6.26	5.82	6.63
3	6.39	0.09	6.40	6.33	6.45	6.00	6.82
4	6.78	0.09	6.78	6.72	6.84	6.37	7.22
5	6.75	0.09	6.75	6.69	6.81	6.26	7.21
6	6.35	0.09	6.35	6.29	6.41	5.91	6.74
7	6.85	0.09	6.85	6.79	6.91	6.41	7.29

Continued on next page

Table B.23 - Cont.: Frac. intra-device distance for MSP430F5308 ICs at 90 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
8	6.92	0.09	6.92	6.86	6.98	6.48	7.35
9	6.22	0.09	6.22	6.16	6.27	5.82	6.68
10	6.70	0.09	6.70	6.64	6.76	6.30	7.10
12	6.28	0.09	6.28	6.22	6.34	5.83	6.74
13	6.67	0.09	6.67	6.60	6.73	6.23	7.14
14	6.14	0.09	6.14	6.08	6.20	5.68	6.59
15	6.54	0.60	6.52	6.46	6.58	6.07	25.41
16	6.11	0.09	6.11	6.05	6.17	5.74	6.53
Σ	6.49	0.27	6.47	6.25	6.73	6.11	6.92

Table B.24: Frac. intra-device distance for STM32F100R8 ICs at 20 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	5.46	0.08	5.46	5.41	5.51	5.15	5.79
6	5.50	0.07	5.50	5.45	5.55	5.15	5.82
8	5.61	0.08	5.61	5.56	5.66	5.28	5.99
9	5.63	0.08	5.62	5.57	5.68	5.24	5.99
10	5.50	0.08	5.50	5.45	5.55	5.17	5.84
11	5.47	0.08	5.47	5.42	5.53	5.07	5.80
12	5.40	0.07	5.41	5.35	5.45	5.07	5.76
13	5.58	0.08	5.58	5.53	5.63	5.21	5.97
14	5.34	0.07	5.34	5.30	5.39	4.98	5.72
15	5.49	0.08	5.49	5.44	5.54	5.15	5.81
16	5.67	0.08	5.66	5.61	5.72	5.30	6.04
Σ	5.51	0.10	5.50	5.47	5.59	5.34	5.67

Table B.25: Frac. intra-device distance for STM32F100R8 ICs at −30 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	7.98	0.08	7.98	7.92	8.03	7.60	8.35
6	7.86	0.08	7.86	7.81	7.91	7.44	8.24
8	7.86	0.08	7.86	7.81	7.92	7.48	8.23
9	7.83	0.08	7.83	7.78	7.89	7.44	8.25
10	7.79	0.08	7.79	7.74	7.85	7.42	8.16
11	7.72	0.08	7.72	7.67	7.78	7.32	8.13
12	7.72	0.08	7.72	7.67	7.78	7.34	8.14
13	7.72	0.08	7.72	7.67	7.77	7.35	8.09
14	7.76	0.08	7.76	7.71	7.82	7.36	8.10
15	7.90	0.08	7.90	7.84	7.95	7.51	8.33
16	8.01	0.08	8.01	7.95	8.06	7.64	8.41
Σ	7.83	0.10	7.83	7.74	7.88	7.72	8.01

Table B.26: Frac. intra-device distance for STM32F100R8 ICs at 85 °C.

ID	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
2	9.97	0.09	9.97	9.91	10.03	9.44	10.38
6	10.10	0.09	10.10	10.04	10.16	9.62	10.51
8	10.10	0.09	10.10	10.04	10.16	9.47	10.52
9	9.99	0.09	9.99	9.93	10.05	9.46	10.42
10	9.97	0.09	9.97	9.92	10.03	9.20	10.40
11	10.08	0.09	10.08	10.02	10.14	9.44	10.50
12	9.97	0.09	9.98	9.92	10.03	9.35	10.40
13	10.03	0.09	10.03	9.97	10.09	9.36	10.47
14	9.82	0.09	9.82	9.76	9.88	9.22	10.24
15	9.92	0.09	9.92	9.86	9.98	9.14	10.38
16	10.56	0.09	10.56	10.50	10.62	9.85	10.99
Σ	10.05	0.19	9.99	9.97	10.09	9.82	10.56

B.4 Intra-device min-entropy

Table B.27: Frac. intra-device min-entropy for ATmega328P.

Temp. [°C]	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
−30	2.39	0.08	2.38	2.35	2.43	2.25	2.58
20	2.81	0.12	2.80	2.76	2.84	2.65	3.15
90	3.44	0.13	3.45	3.35	3.53	3.17	3.63

Table B.28: Frac. intra-device min-entropy for PIC16F1825.

Temp. [°C]	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
−30	2.00	0.57	2.20	1.47	2.48	1.23	2.82
20	1.92	0.22	1.87	1.77	1.97	1.69	2.52
85	2.34	0.14	2.36	2.21	2.40	2.14	2.69

Table B.29: Frac. intra-device min-entropy for MSP430F5308.

Temp. [°C]	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
−30	5.49	0.85	5.23	4.89	5.57	4.63	7.71
20	5.22	0.17	5.22	5.10	5.33	4.95	5.55
90	6.22	0.25	6.26	5.98	6.38	5.87	6.66

Table B.30: Frac. intra-device min-entropy for STM32F100R8.

Temp. [°C]	$\overline{X}$	s	Md	Q_1	Q_3	X_{min}	X_{max}
−30	5.45	0.08	5.47	5.40	5.50	5.29	5.57
20	6.75	0.12	6.73	6.70	6.85	6.55	6.91
85	6.62	0.16	6.60	6.52	6.74	6.31	6.86

§ Bibliography

[1] D. Agrawal, S. Baktir, D. Karakoyunlu, P. Rohatgi, and B. Sunar, "Trojan Detection using IC Fingerprinting", in *IEEE Symposium on Security and Privacy (SP)*, Berkeley, CA, USA, 2007, pp. 296–310. « Cited on p. 35. »

[2] M. Ajtai and C. Dwork, "A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence", *Electronic Colloquium on Computational Complexity (ECCC)*, vol. 3, no. 65, 1996. « Cited on p. 6. »

[3] Arduino SA, *Arduino Language Reference*, 2014. URL: `http://arduino.cc/en/reference`. « Cited on p. 117. »

[4] ARM Ltd, *ARM Security Technology: Building A Secure System Using TrustZone Technology*, 2005. URL: `http://www.arm.com/products/security/trustzone`. « Cited on p. 118. »

[5] A. Asenov, "Random Dopant Induced Threshold Voltage Lowering and Fluctuations in Sub-0.1 µm MOSFET's: A 3-D "Atomistic" Simulation Study", *IEEE Transactions on Electron Devices*, vol. 45, no. 12, pp. 2505–2513, 1998. « Cited on p. 27. »

[6] J. Balasch, B. Ege, T. Eisenbarth, B. Gérard, Z. Gong, T. Güneysu, S. Heyse, S. Kerckhof, F. Koeune, T. Plos, T. Poppelmann, F. Regazzoni, F.-X. Standaert, G. Van Assche, R. Van Keer, L. Van Oldeneel Tot Oldenzeel, and I. Von Maurich, "Compact Implementation and Performance Evaluation of Hash Functions in ATtiny Devices", in *Smart Card Research and Advanced Applications International Conference (CARDIS)*, S. Mangard, Ed., ser. Lecture Notes in Computer Science, *Implementation of Keccak by G. Van Assche and R. Van Keer*, vol. 7771, Springer-Verlag, 2012, pp. 158–172. « Cited on p. 127. »

[7] B. Barak and S. Halevi, "A Model and Architecture for Pseudo-Random Generation with Applications to /dev/random", in *ACM Conference on Computer and Communications Security (CCS)*, V. Atluri, C. Meadows, and A. Juels, Eds., Alexandria, VA, USA: ACM, 2005, pp. 203–212. « Cited on p. 11. »

[8] E. Barker and J. Kelsey, *Recommendation for Random Number Generation Using Deterministic Random Bit Generators*, NIST Special Publication 800-90A, National Institute of Standards and Technology, 2012.
« Cited on pp. 11, 25, 64, 111, 117, 120. »

[9] W. C. Barker, *Recommendation for the Triple Data Encryption Algorithm (TDEA) block cipher*. National Institute for Standards and Technology, 2004.
« Cited on pp. 3, 4. »

[10] M. Bellare, R. Canetti, and H. Krawczyk, "Keying Hash Functions for Message Authentication", in *International Conference on Advances in Cryptology (CRYPTO)*, N. Koblitz, Ed., ser. Lecture Notes in Computer Science, vol. 1109, Santa Barbara, CA, USA: Springer, 1996, 15 pp. « Cited on p. 121. »

[11] M. Bellare and B. Yee, *Forward-Security in Private-Key Cryptography*, Cryptology ePrint Archive, Report 2001/035, 2001. « Cited on p. 11. »

[12] E. Berlekamp, "On Decoding Binary Bose-Chadhuri-Hocquenghem Codes", *IEEE Transactions on Information Theory*, vol. 11, no. 4, pp. 577–579, 1965.
« Cited on p. 63. »

[13] D. J. Bernstein, "The Salsa20 Family of Stream Ciphers", in *New Stream Cipher Designs - The eSTREAM Finalists*, ser. Lecture Notes in Computer Science, M. J. B. Robshaw and O. Billet, Eds., vol. 4986, Springer, 2008, pp. 84–97. URL: http://dx.doi.org/10.1007/978-3-540-68351-3_8. « Cited on p. 3. »

[14] D. J. Bernstein, Y.-A. Chang, C.-M. Cheng, L.-P. Chou, N. Heninger, T. Lange, and N. van Someren, *Factoring RSA keys from certified smart cards: Coppersmith in the wild*, Cryptology ePrint Archive, Report 2013/599, 2013. « Cited on p. 14. »

[15] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, "Sponge-Based Pseudo-Random Number Generators", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, S. Mangard and F.-X. Standaert, Eds., ser. Lecture Notes in Computer Science, vol. 6225, Santa Barbara, CA, USA: Springer, 2010, pp. 33–47. « Cited on pp. 121, 122. »

[16] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche, *The Keccak reference*, Round 3 submission to NIST SHA-3, 2011. « Cited on pp. 9, 121. »

[17] G. Bertoni, J. Daemen, M. Peeters, G. Van Assche, and R. Van Keer, *Keccak implementation overview*, Round 3 submission to NIST SHA-3, 2011.
« Cited on pp. 124, 125. »

[18] L. Blum, M. Blum, and M. Shub, "A Simple Unpredictable Pseudo-Random Number Generator", *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, 1986. « Cited on p. 11. »

[19] A. Bogdanov, M. Knežević, G. Leander, D. Toz, K. Varici, and I. Verbauwhede, "SPONGENT: A Lightweight Hash Function", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, B. Preneel and T. Takagi, Eds., ser. Lecture Notes in Computer Science, vol. 6917, Naga, Japan: Springer, 2011, pp. 312–325. « Cited on p. 65. »

[20] C. Böhm and M. Hofer, *Physical Unclonable Functions in Theory and Practice*. New York, NY, USA: Springer, 2013. « Cited on p. 26. »

[21] C. Böhm, M. Hofer, and W. Pribyl, "A Microcontroller SRAM-PUF", in *International Conference on Network and System Security (NSS)*, P. Samarati, S. Foresti, J. Hu, and G. Livraga, Eds., Milan, Italy: IEEE, 2011, pp. 269–273. « Cited on pp. 49, 90. »

[22] C. Bösch, J. Guajardo, A.-R. Sadeghi, J. Shokrollahi, and P. Tuyls, "Efficient Helper Data Key Extractor on FPGAs", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 5154, Washington, DC, USA: Springer-Verlag, 2008, pp. 181–197. « Cited on pp. 49, 58, 59, 62, 86, 90. »

[23] R. C. Bose and D. K. Ray-Chaudhuri, "On a Class of Error Correcting Binary Group Codes", *Information and Control*, vol. 3, no. 1, pp. 68–79, 1960. « Cited on p. 62. »

[24] R. G. Brown, D. Eddelbuettel, and D. Bauer, *Dieharder: A Random Number Test Suite*, 2013. URL: `http://www.phy.duke.edu/~rgb/General/dieharder.php`. « Cited on p. 42. »

[25] H. Burton, "Inversionless Decoding of Binary BCH codes", *IEEE Transactions on Information Theory*, vol. 17, no. 4, pp. 464–466, Jul. 1971. « Cited on pp. 63, 79. »

[26] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions", in *ACM Symposium on Theory of Computing (STOC)*, Boulder, Colorado, USA: ACM, 1977, pp. 106–112. « Cited on p. 9. »

[27] H.-C. Chang and C. Shung, "New Serial Architecture for the Berlekamp-Massey Algorithm", *IEEE Transactions on Communications*, vol. 47, no. 4, pp. 481–483, 1999. « Cited on p. 85. »

[28] R. Chien, "Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes", *IEEE Transactions on Information Theory*, vol. 10, no. 4, pp. 357–363, 1964. « Cited on pp. 64, 80. »

[29] R. D. Wagner, *Introduction to Security Printing*. Graphic Arts Center Publishing Company, 2005. « Cited on p. 23. »

[30] J. Daemen and V. Rijmen, *The Design of Rijndael: AES – The Advanced Encryption Standard*. Springer-Verlag, 2002, 238 pp. « Cited on pp. 3, 4. »

[31] R. de Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+", in *Design Automation Conference (DAC)*, San Francisco, CA, USA: ACM, 2014, 6 pp. « Cited on p. 17. »

[32] G. de Koning Gans, J.-H. Hoepman, and F. D. Garcia, "A Practical Attack on the MIFARE Classic", in *Smart Card Research and Advanced Applications*, ser. Lecture Notes in Computer Science, G. Grimaud and F.-X. Standaert, Eds., vol. 5189, Springer, 2008, pp. 267–282. « Cited on p. 14. »

[33] C. A. Deavours, D. Kahn, L. Kruh, G. Mellen, and B. Winkel, *Cryptology: Yesterday, Today, and Tomorrow*. Norwood, MA, USA: Artech House, Inc., 1987. « Cited on p. 3. »

[34] Debian Security, "DSA-1571-1 OpenSSL – Predictable Random Number Generator", Tech. Rep., 2008. URL: `http://www.debian.org/security/2008/dsa-1571.en.html`. « Cited on p. 13. »

[35] J. Delvaux, D. Gu, D. Schellekens, and I. Verbauwhede, "Secure Lightweight Entity Authentication with Strong PUFs: Mission Impossible?", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, Busan, South Korea: Springer-Verlag, 2014, 18 pp. « Cited on pp. 22, 31, 33. »

[36] J. Delvaux and I. Verbauwhede, "Attacking PUF-Based Pattern Matching Key Generators via Helper Data Manipulation", in *Topics in Cryptology - CT-RSA 2014, The Cryptographers' Track at the RSA Conference*, J. Benaloh, Ed., ser. Lecture Notes in Computer Science, vol. 8366, San Francisco, CA, USA: Springer-Verlag, 2014, pp. 106–131. « Cited on pp. 33, 53. »

[37] ——, "Key-recovery Attacks on Various RO PUF Constructions via Helper Data Manipulation", in *Conference on Design, Automation and Test in Europe (DATE)*, Dresden, Germany: IEEE, 2014, 6 pp. « Cited on pp. 33, 53. »

[38] ——, "Side Channel Modeling Attacks on 65nm Arbiter PUFs Exploiting CMOS Device Noise", in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, Austin, TX, USA: IEEE, 2013, pp. 137–142. « Cited on pp. 22, 34. »

[39] W. Diffie and M. E. Hellman, "New Directions in Cryptography", *IEEE Transactions on Information Theory*, vol. 22, no. 6, pp. 644–654, 1976. « Cited on pp. 6, 7. »

[40] R. Dingledine, *Tor Security Advisory: Debian Flaw Causes Weak Identity Keys*, 2008. URL: `https://lists.torproject.org/pipermail/tor-announce/2008-May/000069.html`. « Cited on p. 13. »

[41] Y. Dodis, R. Ostrovsky, L. Reyzin, and A. Smith, "Fuzzy Extractors: How to Generate Strong Keys from Biometrics and Other Noisy Data", *SIAM Journal on Computing*, vol. 38, no. 1, pp. 97–139, 2008. « Cited on pp. 24, 47, 58, 64. »

[42] Y. Dodis, D. Pointcheval, S. Ruhault, D. Vergnaud, and D. Wichs, "Security Analysis of Pseudo-Random Number Generators with Input: /dev/random is not Robust", in *ACM Conference on Computer and Communications Security (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., Berlin, Germany: ACM, 2013, pp. 647–658. « Cited on pp. 12, 14. »

[43] Y. Dodis, A. Shamir, N. Stephens-Davidowitz, and D. Wichs, *How to Eat Your Entropy and Have it Too – Optimal Recovery Strategies for Compromised RNGs*, Cryptology ePrint Archive, Report 2014/167, 2014. « Cited on p. 12. »

[44] D. Eastlake, J. Schiller, and S. Crocker, *Randomness Requirements for Security*, RFC 4086 (Best Current Practice), Internet Engineering Task Force, 2005. URL: `http://www.ietf.org/rfc/rfc4086.txt`. « Cited on pp. 25, 64. »

[45] A. Francillon and C. Castelluccia, "TinyRNG: A Cryptographic Random Number Generator for Wireless Sensors Network Nodes", in *International Symposium on Modeling and Optimization in Mobile, Ad Hoc and Wireless Networks and Workshops (WiOpt)*, 2007, 7 pp. « Cited on p. 117. »

[46] B. Gierlichs, "Statistical and Information-Theoretic Methods for Power Analysis on Embedded Cryptography", B. Preneel and I. Verbauwhede (promotors), PhD thesis, KU Leuven, 2011. « Cited on pp. 34, 118. »

[47] I. Goldberg and D. Wagner, "Randomness and the Netscape Browser. How secure is the World Wide Web?", in *Dr. Dobb's Journal*, Jan. 1996. « Cited on p. 13. »

[48] J. D. Golic, "Cryptanalysis of Alleged A5 Stream Cipher", in *International Conference on the Theory and Application of Cryptographic Techniques (EUROCRYPT)*, W. Fumy, Ed., ser. Lecture Notes in Computer Science, vol. 1233, Konstanz, Germany: Springer, 1997, pp. 239–255. URL: `http://dx.doi.org/10.1007/3-540-69053-0_17`. « Cited on p. 4. »

[49] B. Groza, P.-S. Murvay, A. Van Herrewege, and I. Verbauwhede, "LiBrA-CAN: A Lightweight Broadcast Authentication Protocol for Controller Area Networks", in *International Conference on Cryptology and Network Security (CANS)*, J. Pieprzyk, A.-R. Sadeghi, and M. Manulis, Eds., vol. 7712, Darmstadt, Germany: Springer, 2012, pp. 185–200. « Cited on p. 18. »

[50] J. Guajardo, S. S. Kumar, G.-J. Schrijen, and P. Tuyls, "FPGA Intrinsic PUFs and Their Use for IP Protection", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, P. Paillier and I. Verbauwhede, Eds., ser. Lecture Notes in Computer Science, vol. 4727, Vienna, Austria: Springer, 2007, pp. 63–80. « Cited on pp. 49, 58. »

[51] Z. Gutterman, B. Pinkas, and T. Reinman, "Analysis of the Linux Random Number Generator", in *IEEE Symposium on Security and Privacy (SP)*, ser. SP, Washington, DC, USA: IEEE Computer Society, 2006, pp. 371–385. « Cited on p. 14. »

[52] C. Helfmeier, C. Boit, D. Nedospasov, and J.-P. Seifert, "Cloning Physically Unclonable Functions", in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, Austin, TX, USA: IEEE, 2013, 6 pp. « Cited on p. 35. »

[53] N. Heninger, Z. Durumeric, E. Wustrow, and J. A. Halderman, "Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices", in *USENIX Security Symposium*, Bellevue, WA, USA, 2012. « Cited on pp. 14, 24. »

[54] J. Hlaváč, R. Lórencz, and M. Hadáček, "True Random Number Generation on an Atmel AVR Microcontroller", in *International Conference on Computer Engineering and Technology (ICCET)*, vol. 2, 2010, pp. 493–495. « Cited on p. 117. »

[55] A. Hocquenghem, "Codes Correcteurs d'Erreurs", *Chiffres*, vol. 2, pp. 147–156, 1959. « Cited on p. 62. »

[56] D. E. Holcomb, W. P. Burleson, and K. Fu, "Initial SRAM State as a Fingerprint and Source of True Random Numbers for RFID Tags", in *Conference on RFID Security*, 2007. « Cited on pp. 90, 117. »

[57] ——, "Power-Up SRAM State as an Identifying Fingerprint and Source of True Random Numbers", *IEEE Transactions on Computers*, vol. 58, no. 9, pp. 1198–1210, 2009. « Cited on p. 54. »

[58] G. Hospodar, R. Maes, and I. Verbauwhede, "Machine Learning Attacks on 65nm Arbiter PUFs: Accurate Modeling poses strict Bounds on Usability", in *IEEE International Workshop on Information Forensics and Security (WIFS)*, Tenerife, Spain: IEEE, 2012, pp. 37–42. « Cited on pp. 31, 34. »

[59] T. Ignatenko, G.-J. Schrijen, B. Škorić, P. Tuyls, and F. Willems, "Estimating the Secrecy-Rate of Physical Unclonable Functions with the Context-Tree Weighting Method", in *IEEE International Symposium on Information Theory*, 2006, pp. 499–503. « Cited on p. 45. »

[60] Intel Corporation, *Intel Digital Random Number Generator (DRNG) Software Implementation Guide*, `http://software.intel.com/en-us/articles/intel-digital-random-number-generator-drng-software-implementation-guide`, 2012. « Cited on p. 12. »

[61] C. D. Isbell, "Some Cryptograms in the Aramaic Incantation Bowls", *Journal of Near Eastern Studies*, vol. 33, no. 4, pp. 405–407, 1974. « Cited on p. 3. »

[62] S.-M. Kang and Y. Leblebici, *CMOS Digital Integrated Circuits Analysis & Design*, 3rd ed. New York, NY, USA: McGraw-Hill, Inc., 2003. « Cited on p. 28. »

[63] S. Katzenbeisser, Ü. Koçabas, V. van der Leest, A.-R. Sadeghi, G.-J. Schrijen, H. Schröder, and C. Wachsmann, "Recyclable PUFs: Logically Reconfigurable PUFs", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, B. Preneel and T. Takagi, Eds., ser. Lecture Notes in Computer Science, vol. 6917, Nara, Japan: Springer, 2011, pp. 374–389. « Cited on p. 24. »

[64] J. Kelsey, B. Schneier, and N. Ferguson, "Yarrow-160: Notes on the Design and Analysis of the Yarrow Cryptographic Pseudorandom Number Generator", in *International Workshop on Selected Areas in Cryptography (SAC)*, H. M. Heys and C. M. Adams, Eds., ser. Lecture Notes in Computer Science, vol. 1758, Kingston, Ontario, Canada: Springer, 1999, pp. 13–33. « Cited on pp. 11, 25, 64. »

[65] W. Killmann and W. Schindler, "AIS20/AIS31: A proposal for Functionality classes for random number generators (version 2.0)", Bundesamt für Sicherheit in der Informationstechnik (BSI), Germany, Tech. Rep., 2011, 133 pp. « Cited on p. 42. »

[66] N. Koblitz, "Elliptic Curve Cryptosystems", *Mathematics of Computation*, vol. 48, no. 177, pp. 203–209, 1987. « Cited on p. 6. »

[67] P. Koeberl, J. Li, A. Rajan, and W. Wu, "Entropy Loss in PUF-based Key Generation Schemes: The Repetition Code Pitfall", in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, Arlington, VA, USA: IEEE, 2014, 6 pp. « Cited on pp. 49, 58, 86. »

[68] H. Krawczyk, M. Bellare, and R. Canetti, *HMAC: Keyed-Hashing for Message Authentication*, RFC 2104, Internet Engineering Task Force, 1997. URL: `http://www.ietf.org/rfc/rfc2104.txt`. « Cited on p. 121. »

[69] B. Kristinsson, "Ardrand: The Arduino as a Hardware Random-Number Generator", *Computing Research Repository (CoRR)*, 2012. URL: `http://arxiv.org/abs/1212.3777`. « Cited on p. 117. »

[70] P. Lapsley, J. Bier, E. A. Lee, and A. Shoham, *DSP Processor Fundamentals: Architectures and Features*, 1st ed. Wiley-IEEE Press, 1996. « Cited on p. 73. »

[71] A. K. Lenstra, J. P. Hughes, M. Augier, J. W. Bos, T. Kleinjung, and C. Wachter, *Ron was wrong, Whit is right*, Cryptology ePrint Archive, Report 2012/064, 2012. « Cited on p. 14. »

[72] W. Liu, J. Rho, and W. Sung, "Low-Power High-Throughput BCH Error Correction VLSI Design for Multi-Level Cell NAND Flash Memories", in *IEEE Workshop on Signal Processing Systems (SiPS)*, Banff, Alberta, Canada: IEEE, 2006, pp. 303–308. « Cited on pp. 71, 85, 86. »

[73] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error-Correcting Codes*, 2nd ed. North-Holland Publishing Company, 1978, 762 pp. « Cited on p. 46. »

[74] R. Maes, "An Accurate Probabilistic Reliability Model for Silicon PUFs", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, G. Bertoni and J.-S. Coron, Eds., ser. Lecture Notes in Computer Science, vol. 8086, Santa Barbara, CA, USA: Springer, 2013, pp. 73–89. « Cited on p. 32. »

[75] ——, "Physically Unclonable Functions: Constructions, Properties and Applications", I. Verbauwhede (promotor), PhD thesis, KU Leuven, 2012. « Cited on pp. 19–21, 26, 29, 36, 44, 69. »

[76] R. Maes, R. Peeters, A. Van Herrewege, C. Wachsmann, S. Katzenbeisser, A.-R. Sadeghi, and I. Verbauwhede, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs", in *International Conference on Financial Cryptography and Data Security (FC)*, A. D. Keromytis, Ed., ser. Lecture Notes in Computer Science, vol. 7397, Kralendijk, Bonaire: Springer, 2012, pp. 374–389. « Cited on pp. 18, 48, 58. »

[77] R. Maes, D. Schellekens, and I. Verbauwhede, "A Pay-per-Use Licensing Scheme for Hardware IP Cores in Recent SRAM based FPGAs", *IEEE Transactions on Information Forensics and Security*, vol. 7, no. 1, pp. 98–108, 2012. « Cited on p. 25. »

[78] R. Maes, P. Tuyls, and I. Verbauwhede, "Low-Overhead Implementation of a Soft Decision Helper Data Algorithm for SRAM PUFs", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 5747, Lausanne, Switzerland: Springer-Verlag, 2009, pp. 332–347. « Cited on pp. 49, 58. »

[79] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, E. Prouff and P. Schaumont, Eds., ser. Lecture Notes in Computer Science, vol. 7428, Leuven, Belgium: Springer, 2012, pp. 302–319. « Cited on pp. 16, 57. »

[80] A. Maiti, J. Casarona, L. McHale, and P. Schaumont, "A Large Scale Characterization of RO-PUF", in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, USA, 2010, pp. 94–99. « Cited on p. 66. »

[81] G. Marsaglia, *Diehard Battery of Tests of Randomness*, 1995. URL: `http://www.stat.fsu.edu/pub/diehard`. « Cited on p. 42. »

[82] J. Massey, "Shift-Register Synthesis and BCH Decoding", *IEEE Transactions on Information Theory*, vol. 15, no. 1, pp. 122–127, 1969. « Cited on p. 63. »

[83] U. M. Maurer, "A Universal Statistical Test for Random Bit Generators", in *International Conference on Advances in Cryptology (CRYPTO)*, A. Menezes and S. A. Vanstone, Eds., ser. Lecture Notes in Computer Science, vol. 537, Santa Barbara, CA, USA: Springer, 1990, pp. 409–420. « Cited on p. 45. »

[84] Maxim Integrated, *DeepCover® Secure Microcontrollers*, 2014. URL: `http://www.maximintegrated.com`. « Cited on p. 119. »

[85] R. J. McEliece, "A Public-Key Cryptosystem Based On Algebraic Coding Theory", *Deep Space Network Progress Report*, vol. 44, pp. 114–116, 1978. « Cited on p. 6. »

[86] A. J. Menezes, S. A. Vanstone, and P. C. V. Oorschot, *Handbook of Applied Cryptography*, 1st ed. Boca Raton, FL, USA: CRC Press, Inc., 1996. « Cited on pp. 2–4, 6, 8–11, 22. »

[87] D. Merli, D. Schuster, F. Stumpf, and G. Sigl, "Side-Channel Analysis of PUFs and Fuzzy Extractors", in *Trust and Trustworthy Computing*, ser. Lecture Notes in Computer Science, J. McCune, B. Balacheff, A. Perrig, A.-R. Sadeghi, A. Sasse, and Y. Beres, Eds., vol. 6740, Springer, 2011, pp. 33–47. « Cited on p. 34. »

[88] Microchip, *Stephens Inc. Spring Investment Conference*, 2013. URL: `http://www.microchip.com/investor`. « Cited on p. 91. »

[89] V. S. Miller, "Use of Elliptic Curves in Cryptography", in *International Conference on Advances in Cryptology (CRYPTO)*, H. C. Williams, Ed., ser. Lecture Notes in Computer Science, vol. 218, Santa Barbara, CA, USA: Springer, 1985, pp. 417–426. « Cited on p. 6. »

[90] K. Misiakos and D. Tsamakis, "Accurate measurements of the silicon intrinsic carrier density from 78 to 340 K", *Journal of Applied Physics*, vol. 74, pp. 3293–3297, 1993. « Cited on p. 28. »

[91] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers", in *International Conference on Selected Areas in Cryptography (SAC)*, A. Joux and A. Youssef, Eds., ser. Lecture Notes in Computer Science, Montreal, Quebec, Canada: Springer, 2014, 18 pp. « Cited on p. 17. »

[92] National Institute of Standards and Technology, "FIPS 180-4, Secure Hash Standard, Federal Information Processing Standard (FIPS), Publication 180-4", Department of Commerce, Tech. Rep., 2002, 35 pp. « Cited on p. 9. »

[93] D. Nedospasov, J. Seifert, C. Helfmeier, and C. Boit, "Invasive PUF Analysis", in *Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, W. Fischer and J. Schmidt, Eds., Los Alamitos, CA, USA: IEEE, 2013, pp. 30–38. « Cited on p. 34. »

[94] J. von Neumann, "Various Techniques Used in Connection With Random Digits", *Journal of Research of the National Bureau of Standards, Appl. Math. Series*, vol. 3, pp. 36–38, 1951. « Cited on p. 50. »

[95] N. Nisan and D. Zuckerman, "Randomness is Linear in Space", *Journal of Computer and System Sciences*, vol. 52, no. 1, pp. 43–52, 1996. « Cited on p. 24. »

[96] J. Noorman, P. Agten, W. Daniels, C. Huygens, F. Piessens, B. Preneel, R. Strackx, A. Van Herrewege, and I. Verbauwhede, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base", in *USENIX Security Symposium*, Washington, DC, USA: USENIX, 2013, pp. 479–498. « Cited on pp. 17, 118. »

[97] C. W. O'Donnell, G. E. Suh, and S. Devadas, "PUF-Based Random Number Generation", in *Massachusetts Institute of Technology CSAIL CSG Technical Memo 481*, 2004, 4 pp. « Cited on p. 117. »

[98] R. S. Pappu, B. Recht, J. Taylor, and N. Gershenfeld, "Physical one-way functions", *Science*, vol. 297, pp. 2026–2030, 2002. « Cited on p. 23. »

[99] J.-I. Park, H. Lee, and S. Lee, "An Area-Efficient Truncated Inversionless Berlekamp-Massey Architecture for Reed-Solomon Decoders", in *IEEE International Symposium on Circuits and Systems (ISCAS)*, 2011, pp. 2693–2696. « Cited on pp. 71, 85, 86. »

[100] J.-I. Park, K. Lee, C.-S. Choi, and H. Lee, "High-Speed Low-Complexity Reed-Solomon Decoder using Pipelined Berlekamp-Massey Algorithm", in *International SoC Design Conference (ISOCC)*, 2009, pp. 452–455. « Cited on pp. 63, 71, 85, 86. »

[101] Y. Peres, "Iterating Von Neumann's Procedure for Extracting Random Bits", *The Annals of Statistics*, vol. 20, no. 1, pp. 590–597, 1992. « Cited on p. 51. »

[102] M. Platonov, J. Hlaváč, and R. Lórencz, "Using Power-up SRAM State of Atmel ATmega1284P Microcontrollers as Physical Unclonable Function for Key Generation and Chip Identification", in *Workshop on Trustworthy Manufacturing and Utilization of Secure Devices*, 2013. « Cited on p. 90. »

[103] I. Reed and M. Shih, "VLSI Design of Inverse-Free Berlekamp-Massey Algorithm", *IEEE Proceedings on Computers and Digital Techniques*, vol. 138, no. 5, pp. 295–298, 1991. « Cited on pp. 63, 71, 85. »

[104] R. L. Rivest, A. Shamir, and L. Adleman, "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978. « Cited on p. 6. »

[105] R. L. Rivest and J. C. N. Schuldt, *Spritz—a spongy RC4-like stream cipher and hash function*, Presented at Charles River Crypto Day (2014-10-24)., 2014. URL: `http://people.csail.mit.edu/rivest/pubs/RS14.pdf`. « Cited on p. 3. »

[106] U. Rührmair, F. Sehnke, J. Sölter, G. Dror, S. Devadas, and J. Schmidhuber, "Modeling Attacks on Physical Unclonable Functions", in *ACM Conference on Computer and Communications Security (CCS)*, ser. CCS, New York, NY, USA: ACM, 2010, pp. 237–249. « Cited on p. 34. »

[107] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo, *A Statistical Test Suite for Random and Pseudorandom Number Generators for Cryptographic Applications*, NIST Special Publication 800-22, 2010. URL: `http://csrc.nist.gov/groups/ST/toolkit/rng`. « Cited on p. 42. »

[108] D. Sarwate and N. Shanbhag, "High-Speed Architectures for Reed-Solomon Decoders", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 9, no. 5, pp. 641–655, Oct. 2001. « Cited on pp. 63, 71, 85. »

[109] A. Schaller, B. Škorić, and S. Katzenbeisser, *Eliminating Leakage in Reverse Fuzzy Extractors*, Cryptology ePrint Archive, Report 2014/741, 2014. « Cited on p. 48. »

[110] G.-J. Schrijen and V. van der Leest, "Comparative Analysis of SRAM Memories Used as PUF Primitives", in *DATE*, W. Rosenstiel and L. Thiele, Eds., Dresden, Germany: IEEE, 2012, pp. 1319–1324. « Cited on p. 105. »

[111] S. Skorobogatov, "Low Temperature Data Remanence in Static RAM", University of Cambridge, Tech. Rep. UCAM-CL-TR-536, 2002, 9 pp. URL: `http://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-536.pdf`. « Cited on p. 92. »

[112] STMicroelectronics, *AN4246 - Proprietary Code Read Out Protection on STM32L1 microcontrollers*, 2013. URL: `http://www.st.com`. « Cited on p. 119. »

[113] V. van der Leest, B. Preneel, and E. van der Sluis, "Soft Decision Error Correction for Compact Memory-Based PUFs Using a Single Enrollment", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, E. Prouff and P. Schaumont, Eds., ser. Lecture Notes in Computer Science, vol. 7428, Leuven, Belgium: Springer, 2012, pp. 268–282. « Cited on pp. 25, 49. »

[114] V. van der Leest, E. van der Sluis, G.-J. Schrijen, P. Tuyls, and H. Handschuh, "Efficient Implementation of True Random Number Generator Based on SRAM PUFs", in *Cryptography and Security*, D. Naccache, Ed., ser. Lecture Notes in Computer Science, vol. 6805, Springer, 2012, pp. 300–318. « Cited on pp. 54, 117. »

[115] E. van der Sluis, G.-J. Schrijen, and H. Handschuh, "Random Number Generating System Based on Memory Start-Up Noise", pat. EP 2695052 A2, Oct. 11, 2012. URL: `http://www.google.com/patents/EP2695052A2`. « Cited on p. 54. »

[116] A. Van Herrewege, "Compacte implementaties van paringen", B. Preneel and I. Verbauwhede (promotors), Master's thesis, KU Leuven, 2009, 74 pp. « Cited on p. 18. »

[117] A. Van Herrewege, L. Batina, M. Knežević, I. Verbauwhede, and B. Preneel, "Compact Implementations of Pairings", in *Benelux Workshop on Information and System Security (WiSSeC)*, Louvain-La-Neuve, Belgium, 2009, 4 pp. « Cited on p. 18. »

[118] A. Van Herrewege, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-the-Shelf Microcontrollers", in *ACM Conference on Computer and Communications Security (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., Berlin, Germany: ACM, 2013, pp. 1333–1336. « Cited on pp. 16, 90, 116. »

[119] A. Van Herrewege, D. Singelée, and I. Verbauwhede, "CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus", in *Conference on Embedded Security in Cars (escar)*, Dresden, Germany, 2011, 7 pp. « Cited on p. 18. »

[120] ——, "CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus", in *ECRYPT Workshop on Lightweight Cryptography*, Louvain-La-Neuve, Belgium, 2011, pp. 229–239. « Cited on p. 18. »

[121] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers", in *International Workshop on Trustworthy Embedded Devices (TrustED)*, A.-R. Sadeghi, F. Armknecht, and J.-P. Seifert, Eds., ser. TrustED '13, Berlin, Germany: ACM, 2013, pp. 55–64. « Cited on pp. 16, 90, 116. »

[122] A. Van Herrewege and I. Verbauwhede, "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M", in *Design Automation Conference (DAC)*, San Francisco, CA, USA: ACM, 2014, 6 pp. « Cited on pp. 16, 17, 116. »

[123] ——, "Tiny, Application-Specific, Programmable Processor for BCH Decoding", in *International Symposium on System on Chip (SoC)*, Tampere, Finland: IEEE, 2012, 4 pp. « Cited on pp. 16, 57. »

[124] M. Varchola, M. Drutarovsky, and V. Fischer, "New Universal Element with Integrated PUF and TRNG Capability", in *International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, 2013, 6 pp. « Cited on p. 54. »

[125] K. Wong and S. Chen, "The Entropy of Ordered Sequences and Order Statistics", *IEEE Transactions on Information Theory*, vol. 36, no. 2, pp. 276–284, 1990. « Cited on p. 67. »

[126] C.-E. D. Yin and G. Qu, "LISA: Maximizing RO PUF's Secret Extraction", in *IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, Anaheim, CA, USA, 2010, pp. 100–105. « Cited on p. 65. »

[127] M.-D. M. Yu, D. M'Raihi, R. Sowell, and S. Devadas, "Lightweight and Secure PUF Key Storage Using Limits of Machine Learning", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, ser. Lecture Notes in Computer Science, vol. 6917, Nara, Japan, 2011, pp. 358–373. « Cited on p. 58. »

Anthony Van Herrewege

PhD in cryptography with a love for robotics

+32 477 25 89 05
anthony@anthonyvh.com
www.anthonyvh.com
anthonyvh

Professional Experience

Aug '09 – Jan '15 **Research assistant**, KU Leuven – ESAT/COSIC, Leuven (Belgium)
- PhD research on efficient implementations and algorithms for embedded cryptography.
- Active involvement in multiple FPGA and embedded microcontroller design projects.
- Supervision of multiple Master theses and of robotic design projects.
- Organization of the CHES 2012 conference (500+ attendees).

Jul '08 – Jun '09 **Board member**, Flemish Engineering Student Union (VTK), Leuven (Belgium)
- Responsible for support and improvement of an IT infrastructure for 2000+ students.
- Planning and execution of a varied set of extra-curricular activities for the students.

Education

Aug '09 – Jan '15 **PhD in Electrical Engineering**, KU Leuven – ESAT/COSIC, Leuven (Belgium)
- Topic: "Lightweight PUF-based Key and Random Number Generation"
- Supervisor: prof. dr. ir. Ingrid Verbauwhede

Sep '04 – Jul '09 **Master of Engineering (Electrical Engineering)**, KU Leuven, Leuven (Belgium)
- Cum laude – Specialization: Multimedia and Signal Processing
- Thesis: "Compact Implementation of Pairings"

Skills

Communication
Dutch Native speaker
English Professional proficiency (PhD in an international research environment)
French Average (eight years of lessons at high school level)
German Basic (two years of lessons at high school level)

Programming
General: C, C++, Java, LaTeX, Matlab, PHP, Python, …
Assembly: Atmel AVR, Microchip PIC, TI MSP430, ARM Cortex-M

Electronics
General: basic analog design, PCB layout (Altium, DipTrace, Eagle), HDL (Verilog)
Digital design: FPGA development (Xilinx ISE & System Generator), ASIC synthesis (Synopsys), simulation (ModelSim, Icarus Verilog)

Miscellaneous

Fall '11 Finished within top 1% (53rd of 7897) in University of Waterloo AI Challenge.

Hobbies

All kinds of DIY projects (e.g. electronics, mechanics, sewing), climbing, and skiing.

§ List of Publications

Journals

[1] B. Groza, S. Murvay, A. Van Herrewege, and I. Verbauwhede, "LiBrA-CAN: Lightweight Broadcast Authentication for Controller Area Networks", *Transactions on Embedded Computing Systems*, 23 pp., 2014, *In submission.*

[2] A. Schaller, A. Van Herrewege, V. van der Leest, T. Arul, S. Katzenbeisser, and I. Verbauwhede, "Empirical Analysis of Intrinsic Physically Unclonable Functions Found in Commodity Hardware", *Journal of Cryptographic Engineering*, 11 pp., 2014, *In submission.*

International Conferences

[3] I. Verbauwhede, J. Balasch, S. Sinha Roy, and A. Van Herrewege, "Circuit Challenges from Cryptography", in *International Solid-State Circuits Conference (ISSCC), Accepted*, San Francisco, CA, USA: IEEE, 2015, 2 pp.

[4] N. Mouha, B. Mennink, A. Van Herrewege, D. Watanabe, B. Preneel, and I. Verbauwhede, "Chaskey: An Efficient MAC Algorithm for 32-bit Microcontrollers", in *International Conference on Selected Areas in Cryptography (SAC)*, A. Joux and A. Youssef, Eds., ser. Lecture Notes in Computer Science, Montreal, Quebec, Canada: Springer, 2014, 18 pp.

[5] A. Van Herrewege and I. Verbauwhede, "Software Only, Extremely Compact, Keccak-based Secure PRNG on ARM Cortex-M", in *Design Automation Conference (DAC)*, San Francisco, CA, USA: ACM, 2014, 6 pp.

[6] R. de Clercq, L. Uhsadel, A. Van Herrewege, and I. Verbauwhede, "Ultra Low-Power Implementation of ECC on the ARM Cortex-M0+", in *Design Automation Conference (DAC)*, San Francisco, CA, USA: ACM, 2014, 6 pp.

[7] J. Noorman, P. Agten, W. Daniels, C. Huygens, F. Piessens, B. Preneel, R. Strackx, A. Van Herrewege, and I. Verbauwhede, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base", in *USENIX Security Symposium*, Washington, DC, USA: USENIX, 2013, pp. 479–498.

[8] A. Van Herrewege and I. Verbauwhede, "Tiny, Application-Specific, Programmable Processor for BCH Decoding", in *International Symposium on System on Chip (SoC)*, Tampere, Finland: IEEE, 2012, 4 pp.

[9] R. Maes, A. Van Herrewege, and I. Verbauwhede, "PUFKY: A Fully Functional PUF-Based Cryptographic Key Generator", in *International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, E. Prouff and P. Schaumont, Eds., ser. Lecture Notes in Computer Science, vol. 7428, Leuven, Belgium: Springer, 2012, pp. 302–319.

[10] R. Maes, R. Peeters, A. Van Herrewege, C. Wachsmann, S. Katzenbeisser, A.-R. Sadeghi, and I. Verbauwhede, "Reverse Fuzzy Extractors: Enabling Lightweight Mutual Authentication for PUF-enabled RFIDs", in *International Conference on Financial Cryptography and Data Security (FC)*, A. D. Keromytis, Ed., ser. Lecture Notes in Computer Science, vol. 7397, Kralendijk, Bonaire: Springer, 2012, pp. 374–389.

[11] B. Groza, P.-S. Murvay, A. Van Herrewege, and I. Verbauwhede, "LiBrA-CAN: A Lightweight Broadcast Authentication Protocol for Controller Area Networks", in *International Conference on Cryptology and Network Security (CANS)*, J. Pieprzyk, A.-R. Sadeghi, and M. Manulis, Eds., vol. 7712, Darmstadt, Germany: Springer, 2012, pp. 185–200.

Peer-Reviewed Workshop Presentations and Posters

[12] A. Van Herrewege, V. van der Leest, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "Secure PRNG Seeding on Commercial Off-the-Shelf Microcontrollers", in *International Workshop on Trustworthy Embedded Devices (TrustED)*, A.-R. Sadeghi, F. Armknecht, and J.-P. Seifert, Eds., ser. TrustED '13, Berlin, Germany: ACM, 2013, pp. 55–64.

[13] A. Van Herrewege, A. Schaller, S. Katzenbeisser, and I. Verbauwhede, "DEMO: Inherent PUFs and Secure PRNGs on Commercial Off-the-Shelf Microcontrollers", in *ACM Conference on Computer and Communications Security (CCS)*, A.-R. Sadeghi, V. D. Gligor, and M. Yung, Eds., Berlin, Germany: ACM, 2013, pp. 1333–1336.

[14] A. Van Herrewege, D. Singelée, and I. Verbauwhede, "CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus", in *Conference on Embedded Security in Cars (escar)*, Dresden, Germany, 2011, 7 pp.

[15] ——, "CANAuth - A Simple, Backward Compatible Broadcast Authentication Protocol for CAN bus", in *ECRYPT Workshop on Lightweight Cryptography*, Louvain-La-Neuve, Belgium, 2011, pp. 229–239.

[16] A. Van Herrewege, L. Batina, M. Knežević, I. Verbauwhede, and B. Preneel, "Compact Implementations of Pairings", in *Benelux Workshop on Information and System Security (WiSSeC)*, Louvain-La-Neuve, Belgium, 2009, 4 pp.

Miscellaneous

[17] A. Van Herrewege, "Enabling Hardware Performance Counters on the ARM Cortex-A8", KU Leuven - ESAT/COSIC, Belgium, COSIC internal report, 2010, 11 pp.

[18] D. V. Bailey, L. Batina, D. J. Bernstein, P. Birkner, J. W. Bos, H.-C. Chen, C.-M. Cheng, G. De Meulenaer, L. J. Dominguez Perez, J. Fan, T. Guneysu, F. Gurkaynak, T. Kleinjung, T. Lange, N. Mentens, R. Niederhagen, C. Paar, F. Regazzoni, P. Schwabe, L. Uhsadel, G. Van Damme, A. Van Herrewege, and B.-Y. Yang, *Breaking ECC2K-130*, Cryptology ePrint Archive, Report 2009/541, 2009.

[19] A. Van Herrewege, "Compacte implementaties van paringen", B. Preneel and I. Verbauwhede (promotors), Master's thesis, KU Leuven, 2009, 74 pp.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COMPUTER SECURITY AND INDUSTRIAL CRYPTOGRAPHY
Kasteelpark Arenberg 10 bus 2452
3001 Heverlee
firstname.lastname@esat.kuleuven.be
http://www.esat.kuleuven.be