HECTOR

D3.1

Report on the Efficient Implementations of Crypto Algorithms & Building Blocks

and on Cost and Benefits of Countermeasures Against Physical Attacks

Project number:	ICT-644052
Project acronym:	**HECTOR**
Project title:	Hardware Enabled Crypto and Randomness
Project Start Date:	1 March, 2015
Duration:	36 months
Programme:	H2020-ICT-2014-1
Deliverable Type:	Report
Reference Number:	ICT-644052-D3.1-1.0
Workpackage:	WP 3
Due Date:	Feb 2017 - M24
Actual Submission Date:	28 February, 2017
Responsible Organisation:	ST Italy
Editor:	Filippo Melzani
Dissemination Level:	Public
Revision:	1.0
Abstract:	This report represents the final version of Deliverable 3.1 of the HECTOR work package WP3. It covers two main activities. First, the definition of cryptographic primitives, with a special focus on authenticated encryption and their efficient implementations in hardware. Second, the study of side-channel attacks and countermeasure for those cryptographic primitives. In this context our contribution is twofold. We analyze the attacks and propose countermeasure from the system-level viewpoint. Then we introduce a methodology for the evaluation at design-time of the side-channel robustness of hardware implementations.
Keywords:	Authenticated Encryption, Side-channel attacks, Side-channel countermeasures, Side-channel evaluation

This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement no. 644052.

Editor

Filippo Melzani (ST Italy)

Contributors (ordered according to beneficiary numbers)

Josep Balasch, Danilo Šijačić (KUL)
Guido Bertoni, Filippo Melzani, Ruggero Susella (STI)
Maria Eichelseder, Thomas Korak (TUG)

Disclaimer

The information in this document is provided as is, and no guarantee or warranty is given that the information is fit for any particular purpose. The content of this document reflects only the author's view the European Commission is not responsible for any use that may be made of the information it contains. The users use the information at their sole risk and liability.

Executive Summary

This deliverable concerns the activities carried out within tasks T3.3 and T3.4 of the HECTOR project. It covers two main topics: efficient hardware implementations of cryptographic primitives and side-channel protection of such primitives. These two activities are tightly connected. Considering side-channel protection from the beginning when defining new cryptographic primitives, results in more efficient implementations, and helps making side-channel protection more affordable.

Our focus has been mostly on algorithms based on the sponge construction because of its efficiency and security gains potential and because HECTOR partners are among its initial and very active contributors. Knowledge of the state of the art of side-channel attacks and research on countermeasures is fundamental in order to achieve effective designs. Finally, in order to improve efficiency in the process of implementing side-channel-protected cryptographic functions, we propose a methodology allowing to help designers to guarantee and verify at design-time that the final implementation will meet the expected security claims.
One of the goals of the HECTOR project is to stimulate research on these topics, which can have an important impact on the industrial adoption of solutions embedding the state of the art protections.

This deliverable is organized as follows. The overall topic of the report is introduced in Chapter 1.
The important role of Authenticated Encryption (AE) algorithms is explained in Chapter 2, together with the description of the algorithms designed by HECTOR partners. The chapter covers both considerations on the algorithms' specification and strategies towards implementation efficiency. This supports a reasoned selection of the cryptographic blocks used in the applicative use cases targeted in the HECTOR project.
Chapter 3, provides an overview of attacks that can be carried out against a device and describes countermeasures, with a particular focus on those implementable at system level. Such analysis is instrumental for the evaluation of the overall security of a device and it lays-down options to efficiently protect cryptographic algorithms, including those described in Chapter 2.
In Chapter 4, we propose a methodology to effectively realize hardware implementations of cryptographic algorithms with side-channel countermeasures. Such methodology includes a first part that focuses on the specification of the hardware IP through functional languages, and a second part that investigates how to model some important properties of the hardware instantiations (i.e. glitches) that are critical in order to achieve protection.
Finally, Chapter 5 concludes this deliverable.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Cryptographic primitives are the basic building blocks that allow developing and offering security services for all kinds of applications. The range of applications that cryptographic primitives can serve is wide and the corresponding priorities on requirements vary extensively. In the context of the HECTOR project, we decided to focus primarily on Authenticated Encryption (AE), based on Sponges or Permutation-based constructions, with the objective to optimize the overall efficiency.
In the past, cryptographic algorithms have been developed targeting specific security properties, such as either confidentiality or authenticity of data (one at a time). However, real-life applications usually require a set of properties. In particular, a common requirement is to enforce both the confidentiality and authenticity of a message. Authenticated Encryption refers to the class of cryptographic algorithms providing the means to protect both the confidentiality and authenticity of data. This is especially useful in use cases where the attacker can eavesdrop and actively manipulate data. These scenarios include several real-world applications such as secure networking (SSL/TLS, IPSEC, SSH), or data-storage encryption. In most applications, there is not much value in keeping data secret if they are not also authenticated. Data authentication is often more important than confidentiality.

An important motivation behind the focus on AE is the fact that it can serve a wide range of applications. There is currently a lot of interest and intense research in the field looking for solutions that are better than classical approaches. The worldwide cryptographic community organized workshops such as Direction In Authenticated Ciphers (DIAC), and the Competition for Authenticated Encryption (CAESAR), with the specific goal of selecting few algorithms providing state-of-the-art protection, efficiency, and ease of use for applications requiring data authentication and privacy. This is of particular interest within HECTOR since some partners are participating to the CAESAR competition with their own candidate algorithms.

Among the available alternatives, we will show in Chapter 2 how a recent cryptographic construction (i.e. the cryptographic sponge construction) allows to re-use the same primitive for different security services, including authentication, confidentiality and deterministic random number generation, covering the applicative use cases envisioned in the HECTOR project. The fact that a single primitive embedded in a device can be used for different tasks, implicitly leads towards efficient use of resources, and this is the rationale behind our focus on this class of primitives within the HECTOR project. In Chapter 2 we will provide an overview of the authenticated encryption algorithms designed by the HECTOR partners. Besides their original specifications, the focus of this document is to explain the improvements made to those algorithms and their implementations towards efficiency in the context of the HECTOR project.

The specification of cryptographic primitives with strong security properties from the mathematical point of view is a fundamental first step, but it is not enough for achieving robust devices in practice. During the last twenty years, the main threats against security devices have been brought by the so-called *side-channel attacks*. Side-channel attacks exploit unintended information leakage of computing devices or implementations to infer sensitive information.
Many actual attacks against cryptographic implementations have been presented. They allow to extract key material by means of timing information, power consumption, or electromagnetic emanation. Although these attacks require the attacker to be in physical possession of the device, they assume different types of attackers and levels of invasiveness. In Chapter 3 we provide an overview of such attacks, together with some contributions that are defining the cutting edge of research in this direction. This knowledge about attack techniques is essential in order to develop and refine effective and often very sophisticated countermeasures.

Designing and manufacturing devices that are practically robust against side-channel attacks remains difficult. Even with strong cryptographic primitives and theoretically sound countermeasures, achieving good practical robustness against side-channel attacks in physical implementations remains challenging.
The information exploited by side-channel attacks are physical quantities that are not necessarily easy to predict at design-time. For example, methodologies and tools for low power designs that have been developed for a different purpose do not always provide the proper level of detail required for a rigorous side-channel evaluation. More generally, standard design flows do not take into account design-time side-channel evaluation.

We believe that there is the need from the industry to achieve a *side-channel-aware design methodology* in a similar way to methodologies for verification or low-power designs. Such a methodology should help in the design choices and should help increase confidence about the robustness of the resulting physical implementations. In Chapter 4 we describe our attempts in this direction. We address the problem from two perspectives.
First, a top-down methodology based on Functional Languages is introduced, with the goal of closing the gap between high-level specifications and hardware implementations.
Second, as bottom-up approach, we propose a way to model one of the most critical aspects of side-channel-aware hardware design, i.e. the leakages resulting from transitional glitches induced by combinational logic in synchronous designs. These two approaches are complementary to each other and are providing encouraging results. We hope this will stimulate further research on this fundamental topic.

Chapter 2

Cryptographic Algorithms for Authenticated Encryption

In the context of information security, the protection of the information relies on some high-level security services, such as confidentiality or authenticity of data. In order to realize those service requirements, several cryptographic tools have been introduced. Cryptography by itself provides a set of basic tools needed to implement security services.
There exist different families of cryptographic primitives, such as:

- **symmetric key ciphers**, which use the same key to both encrypt and decrypt;
- **public key systems**, which instead have a key pair: one private and one public;
- **cryptographic hash functions**, which are able to generate a fixed size digest from any message, in order to check that the message has not been modified;
- **generators of random values**.

More than one primitive can be used in order to realize a task. For instance, a digital signature can require a public key function, a hash function as well as a random generator.

The HECTOR project mainly addresses:

- random generators, which are discussed in the context of the WP2;
- symmetric key ciphers and hash functions, in particular when they are used to achieve at the same time both authenticity and confidentiality of the data.

The latter point is the content of this chapter, which focuses on a new class of cryptographic algorithms for Authenticated Encryption, specifically introduced to securely and efficiently address applications requiring both authentication and confidentiality.

2.1 Authenticated Encryption

When communicating over an insecure channel, there are two main security properties that must be guaranteed: confidentiality and authenticity of data.
It has long been known how to ensure both of them independently. For instance, starting from a secure block cipher, confidentiality can be achieved by using a suitable encryption mode [17] and authenticity can be achieved by using a block cipher-based MAC [18].

To ensure jointly the two properties, the two constructions can be combined. For instance, one might encrypt a string, prepend a header and then MAC the resulting string. However, it might not be the most efficient solution, especially when the two schemes rely on two different primitives. Moreover, the combination is left to the practitioners and if not properly done, it can have dramatic consequences for security [84, 52, 5].
For these reasons, interest has shifted in recent years towards the definition of so-called AE schemes, specialized constructions that simultaneously provide confidentiality and authenticity of data. The advantage of such ad-hoc constructions is that they can handle both properties with a single primitive, usually with a single key, resulting in better efficiency and less prone to incorrect usage.

A number of AE designs started to appear around 2000, such as IAPM [80, 81], XCBC [65] and OCB [115, 114]. Six authenticated encryption modes (i.e. OCB 2.0, Key Wrap, CCM, EAX, Encrypt-then-MAC (EtM) and GCM) have been standardized in ISO/IEC 19772:2009. Three of them (Key Wrap, CCM and GCM), based on the AES block cipher, have also been standardized by the NIST and are the most commonly adopted.

- **Key Wrap** is an algorithm specifically designed for encrypting key material when it is transmitted over untrusted channels or stored in untrusted places. It uses the AES to securely encrypt key message and an IV for the integrity check. When the key is unwrapped, the recovered IV value is compared with the expected one and if there is a match the key is accepted as valid. Otherwise not. The properties achieved by this integrity check depend on the definition of the IV. For instance, an application might require to check the integrity of a key throughout its lifecycle or just when it is unwrapped.

- The **CCM** (Counter with CBC-MAC), as the name suggests, combines the CBC-MAC mode for authentication with the counter mode for encryption. Specifically, at first CBC-MAC is computed on the message to obtain a tag and then message and tag are encrypted using counter mode. Thus, the overall computation requires two AES encryption operations per message block and the same encryption key can be used for both of them. A designer can thus choose between using two blocks and execute the two passes in parallel or using a single block to perform the two passes in sequence. The first approach will lead to a bigger but faster design, while the second approach will lead to a smaller but slower design. However, the CBC-MAC is not parallelizable. Thus, the maximum achievable performances depend on the throughput of the underlying block cipher, i.e. the AES.

- The **GCM** (Galois Counter Mode) can take full advantage of parallel processing, which makes it widely adopted. It combines counter mode to encrypt the message and Galois field multiplication to combine the ciphertext with an authentication code, obtaining an authentication tag that can be used to check integrity of data. Thus, the scheme requires one AES encryption operation and one 128-bit field multiplication per message block. Both can be easily parallelized and this allows GCM to reach higher throughput with respect to other modes like CCM. GCM is adopted in the IEEE 802.1AE (MACsec) Ethernet security, ANSI (INCITS) Fibre Channel Security Protocols (FC-SP), IETF IPsec standards, SSH and TLS 1.2.

In 2010, Bertoni et al. presented an AE scheme based on the duplex construction [30], showing that confidentiality and authenticity can be achieved together with a single call to the underlying function. It is also the first AE mode based on a permutation instead of a block cipher and that supports intermediate tags.
The ECRYPT network organized the DIAC workshop in 2012 [1], with the aim of evaluating the state of the art in AE and collect input from the community regarding desired future directions. The workshop has shaped the CAESAR competition [126], which started in 2014 and whose goal is to define (by the end of 2017) a portfolio of AE schemes suitable for widespread adoption. Several schemes have been proposed and all of them have been made public for evaluation. The designs submitted to the competition follow different constructions: from purely ad-hoc designs to compression function-based and from sponge constructions to block cipher operating modes.

AE schemes are usually required to handle so-called associated data (AD), namely pieces of information bound to the ciphertext (such as an IP address) that are authenticated but not encrypted. This feature is so important that all modern AE schemes support it.
In order to achieve strong security goals, AE schemes must employ a random IV or a nonce. This follows from the fact that AE schemes are usually built from deterministic primitives. Thus, it is fundamental to ensure high entropy for the IV and non-repetition for the nonce, otherwise the lack of these guarantees can lead to major security breaches. For instance, the repetition of a nonce in OCB allows the attacker to detect repeated message blocks since the corresponding ciphertext blocks will be equal, thus breaking confidentiality.
High-entropy for the IV and non-repetition for the nonce can be hard to achieve in some contexts, for instance when the randomness source is not good or the device where encryption is implemented is stateless. For this reason, interest has grown in designing schemes where the repetition of a nonce has a limited impact on security. Such schemes are called nonce-misuse resistant AE (MRAE) [116] and ensure that authenticity is not affected by nonce repetition and that confidentiality is harmed only if the adversary can detect that a triplet (nonce,AD,message) has been repeated. Example of nonce-misuse resistant schemes are EAX [19], SIV [116], AEZ [74] and GCM-SIV [72].
The notion of MRAE requires that each bit of the ciphertext depends on all bits of the plaintext, and then it cannot be achieved by online schemes. Thus a relaxed definition, called online AE (OAE), was introduced in [58], which can be achieved with a single pass on the plaintext. Examples of OAE are McOE [58], COPA [7] and POET [2]. However, some recent attacks reduced the interest in OAE, especially the chosen-prefix/secret-suffix (CPSS) attack [75].

Most AE schemes provide birthday-bound security with respect to the length of the block of the underlying primitive. Thus, if the primitive is for example AES (with block length of 128 bits) security is lost after 2^{64} calls at best. Moreover, attacks matching this bound are known. For instance, OCB authenticity can be broken by a collision-based attack with 2^{64} message blocks [57].
Doubling the block length would improve the security. However, this approach penalizes performances (as can be seen in generic constructions of block ciphers with double block length) and would be problematic in hardware due to the area cost of the internal state. Algorithms based on the sponge construction can easily overcome this limitation, by properly sizeing the capacity vs the rate portions of the internal state.

2.2 Permutation-based Cryptography

We introduce here the concept of cryptographic sponge, and more generally of permutation-based cryptography, with the aim of showing why such construction is perfectly suited to securely and efficiently address the need for AE, which is a relevant objective of the HECTOR project.
The sponge construction is a mode of operation, based on a fixed-length permutation, which builds a function mapping variable-length input data to variable-length output data. It has been originally introduced by Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche in [23]. Such a function is called *sponge function*.
A sponge function is a generalization of both hash functions, which have a fixed output length, and stream ciphers, which have a fixed input length. It operates on a finite state by iteratively applying the inner permutation to it. Depending on the specific usage, the permutation is interleaved with stages where input data are combined into the internal state or output data are extracted from the state.

One very interesting practical property of the sponge construction is that it can be used to implement a wide variety of cryptographic functions. This includes hashing, reseedable pseudo random number generation, key derivation, encryption, message authentication code computation and authenticated encryption. The fundamental cryptographic primitive underlying all this is a fixed-length permutation. These permutation-based modes are efficient alternatives for all the applications usually addressed with the classical symmetric ciphers. Compared with such symmetric ciphers, from the implementation point of view, a permutation has the advantages that it does not have a key schedule and that its inverse does not need to be implemented or efficient.

2.2.1 The sponge construction

The sponge construction is a simple iterated construction for building a function F with variable-length input M and arbitrary output length Z based on a fixed-length permutation (or transformation) f operating on a fixed number b of bits. The sponge construction operates on a state of $b = r + c$ bits. The value r is called the bitrate and the value c the capacity.
As depicted in Figure 2.1, first the input string M is split into blocks of r bits. Then the b bits of the state are initialized to zero and the sponge construction proceeds in two phases:

- In the **absorbing phase**, the r-bit input blocks are XORed into the first r bits of the state, interleaved with applications of the function f. When all input blocks are processed, the sponge construction switches to the squeezing phase.
- In the **squeezing phase**, the first r bits of the state are returned as output blocks, interleaved with applications of the function f. The number of output blocks is chosen at will by the user, in order to reach the desired number of output bits ℓ.

The last c bits of the state are never directly affected by the input blocks and are never output during the squeezing phase.
In the definition of an algorithm based on a cryptographic sponge, the fundamental aspect is that the primitive to be designed is a fixed-length permutation rather than harder-to-build structures such as block ciphers or dedicated compression functions. This means that all

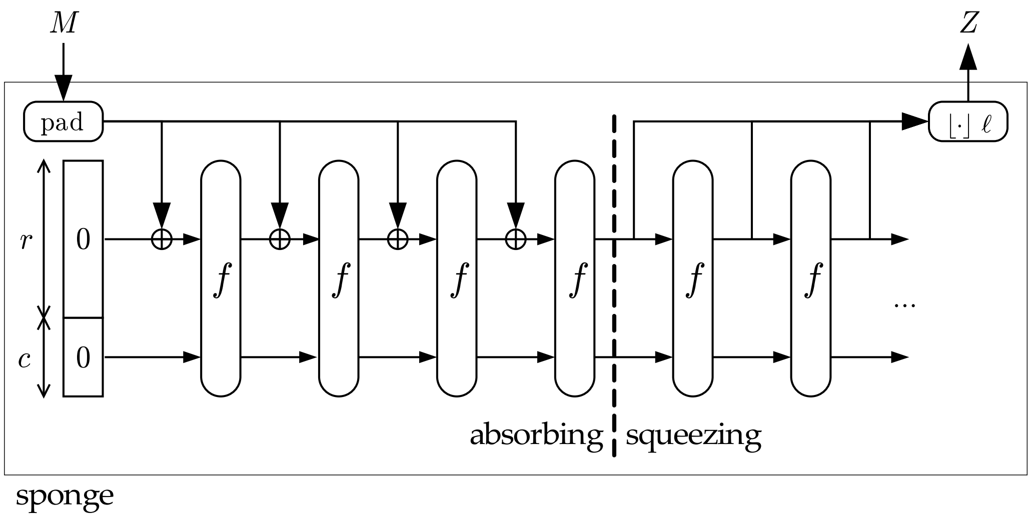

Figure 2.1: The Sponge construction.

the supported cryptographic services can be realized using only a single primitive: a fixed-length permutation. Differently from classical block ciphers, the sponge construction does not require different processing for the message and for the key, resulting then in a simpler design.

The only property required to the permutation in order to be used for a secure cryptographic primitive is that it cannot be distinguished from a typical randomly-chosen permutation. Concretely this means that the permutation must not exhibit any special property that makes easier for an attacker to predict its output compared to finding it by random guesses.
Ultimately, it is necessary to design a permutation f on $b = r + c$ bits that cannot be distinguished from a random permutation, and then the sponge construction can be used to build the sponge function F. Among the defined bits of the state b, the size of the capacity c sets the level of security claimed by the resulting cryptographic primitive. Using the same state size b and the same permutation defined on b bits, it is possible to trade security for speed by increasing the size of the capacity c and decreasing the bitrate r accordingly, or vice-versa.
The sponge construction allows to build various cryptographic primitives (e.g. hash functions, stream ciphers, MAC) since it supports both arbitrarily long input and output sizes. In fact some cases require to use a short input to generate a long output (e.g. a key stream). Some others, instead, have to process a long input in order to generate a short output (e.g. a hash digest or a MAC).
Besides the pure sponge construction just described, the duplex construction has also been introduced by the same authors (see [30]). It is represented in Figure 2.2. The duplex construction is closely related to the sponge construction and their security can be shown to be equivalent.
Compared to the original sponge construction, the duplex construction allows:

- To interleave stages of data absorbing with stages of data squeezing;
- To process input data and generate output data at the same stage.

This allows for instance to implement an efficient reseedable pseudo random bit generator, or an authenticated encryption scheme requiring only one call to f per input (or output) block.

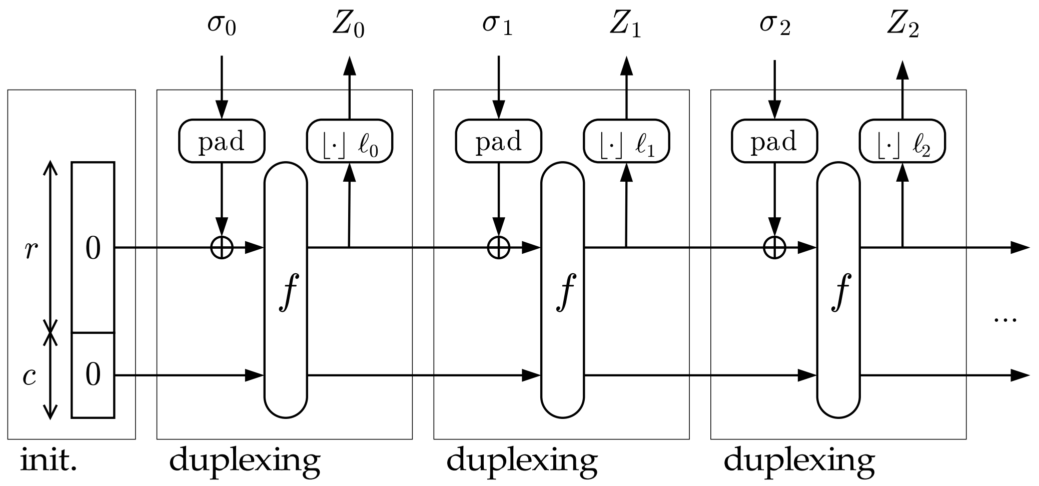

Figure 2.2: The Duplex construction.

2.2.2 SHA-3

The most prominent example of cryptographic primitive built on the sponge construction is KECCAK. KECCAK is a cryptographic sponge function family that has become the FIPS-202 (SHA-3 [45]) standard.
As we have explained, the main building block required in the sponge construction is the permutation. The KECCAK instances are built upon one of seven permutations named KECCAK-$f[b]$, with $b = 25, 50, 100, 200, 400, 800$ or 1600. In the scope of the SHA-3 contest, only the largest one has been proposed, but smaller (or more lightweight) permutations can still be effectively used in constrained environments. Each permutation consists of the iteration of a simple round function, similar to a block cipher without a key schedule. The choice of operations is limited to bitwise XOR, AND and NOT and rotations. Any kind of table-lookups, arithmetic operations, or data-dependent rotations are avoided.
One of the main interesting features of KECCAK is the flexibility it inherits from the sponge construction:

- KECCAK has **arbitrary output length**. This allows to simplify modes of use where dedicated constructions would be needed for fixed-output-length hash functions. In particular, it can be natively used for hashing, stream encryption, and MAC computation.

- As a duplex object, KECCAK can be used in **several clean and efficient modes** as a reseedable pseudo-random generator and for authenticated encryption. Efficiency of duplexing comes from the absence of the output transformation.

- KECCAK has a **simple security claim**. A given security strength level can be targeted by means of choosing the appropriate capacity. This means that for a given capacity c, KECCAK is claimed to stand any attack up to complexity $2^c/2$ (unless easier generically). This is similar to the approach of security strength used in NIST's SP 800-57 [14].

- The **security claim is disentangled from the output length**. There is a minimum output length as a consequence of the chosen security strength level (to avoid generic birthday attacks), but it is not the other way around. Namely, it is not the output length that determines the security strength level.

- The instances proposed for SHA-3 make use of a **single permutation** for all security strengths. This cuts down implementation costs compared to hash function families making use of two (or more) primitives, such as the SHA-2 family. Moreover, with the same permutation the desired performance-security trade-off can be realized by simply choosing the appropriate capacity-rate pair.

From the implementation point of view, KECCAK excels in hardware performance, with speed/area trade-offs, especially when compared to classical SHA algorithms (e.g. it outperforms SHA-2 by an order of magnitude). For instance, Grkaynak et al. in [73] report a maximum throughput of 4.235 Gbit/s for SHA-2 and 27.162 Gbit/s for KECCAK implemented on the same technology.

2.3 Algorithms for Authenticated Encryption

The sponge construction (and in general permutation-based constructions) can be used to implement authenticated encryption by defining a proper mode of operation on top of it. A mode of operation specifies how the input and output data must be formatted and possibly combined when provided to and collected from the underlying cryptographic primitive. AES-GCM and AES-CCM are notable examples of modes of operations built on top of the AES cryptographic primitive.

2.3.1 KEYAK

KEYAK is a family of algorithms for authenticated encryption, all sharing the same structure based on the duplex construction, as defined in section 2.2.1.
The original first version of the KEYAK algorithm was submitted as candidate to the CAESAR competition in 2014 and it was a plain instantiation of the duplex construction. KEYAK has currently been accepted to the third round of the CAESAR competition.

KEYAK has been designed in order to get the most from KECCAK when applied to the specific case of AE. Following this idea, the underlying permutation is the same, with a reduced number of rounds compared to the general case that also serves non-keyed states (e.g. hashing in the SHA-3 context). This means that the security properties of KEYAK directly derive from KECCAK and the benefit from all the cryptanalysis on KECCAK.
After the selection for the second round, KEYAK's authors introduced an improved mode of operation for the algorithm, which extended the plain duplex construction for the specific context of AE. This new construction is named *Motorist mode* and since it is now the basis for the KEYAK algorithm and it brings several benefits, it is described below. Besides being the foundation of KEYAK, in general the Motorist is a mode that can be built on top of any generic cryptographic sponge primitive and it improves the overall efficiency, which is one of the targets of the HECTOR project.

Motorist authenticated encryption mode

The Motorist mode has been introduced in [29] in order to specifically target AE built on top of sponge-based cryptographic primitives. The Motorist mode allows to encrypt and to guarantee the authenticity of sequences of messages in sessions (rather than only a single message). A message consists of a plaintext and possibly some associated data.

At high level (see Figure 2.3), each original message is processed by enciphering the plaintext into a ciphertext and computing a tag over the full sequence of messages. The result consists of a ciphertext, possible associated data (in clear) and a tag. Each encrypted packet, can be unwrapped by deciphering back the ciphertext into the original plaintext, verifying the tag (related to both the plaintext and the associated data), and returning the plaintext only if the tag is valid. A message can also consist of additional data alone, which means that there will be no corresponding ciphertext. Within the same session, the tag of a message authenticates the full sequence of messages sent/received since the start of the session, and not only the specific message as commonly done by classical algorithms (e.g. AES-CCM, AES-GCM). The start of a session requires a secret key and possibly a nonce, if the secret key is not unique for the session. The Motorist mode is sponge-based and supports one or more duplex instances operating in parallel. It makes duplexing calls with input containing key, nonce, plaintext and associated data and uses its output as tag or as key stream bits for encryption (or decryption).

As mentioned before, the duplex instances in the Motorist mode differ from the original duplex construction as described in 2.2.1. The main difference is that the new instances use the full size of the permutation to process data, rather than only the rate part. This variant, initialized with a secret key and denoted fullstate keyed duplex (FSKD), was introduced by Mennink, Reyhanitabar and Vizr [8]. They proved a strong result on the generic security of the FSKD. More precisely, they give an upper bound on the advantage of distinguishing a FSKD calling a random permutation from a random oracle, which is quite close to that of the original keyed duplex construction. This means that increasing the input block length from the rate (r bits) to the width of the permutation (b bits) has no noticeable impact on the generic security, while allowing the injection of more bits per call to the underlying permutation, thus improving performance. At high level, the new Motorist mode offers the same functionality as the original duplex construction of the initial candidates, and it is still built on the security of the sponge construction.

In addition, the Motorist mode supports a parameterized degree of parallelism, in order to sustain very high throughputs. The message is properly distributed over the different duplex instances. It also performs some dedicated processing at the end of each message (called a knot), in order to produce a tag that depends on the full message and not only on the data injected in a single duplex instance. Furthermore, chaining values are extracted from each duplex instance, concatenated, and injected into all duplex instances. This makes the state of all duplex instances depend on the full sequence of messages. At the end a tag is extracted from a single duplex object. To start a session, Motorist takes as input a string that must be secret and (globally) unique, called SUV (Secret and Unique Value). The SUV plays the classical role of secret key and initialization vector.

Figure 2.3 shows a session in MOTORIST. First, the session is started with a given secret and unique value (SUV). Optionally, a tag $T^{(0)}$ on SUV can be produced or verified. Then,

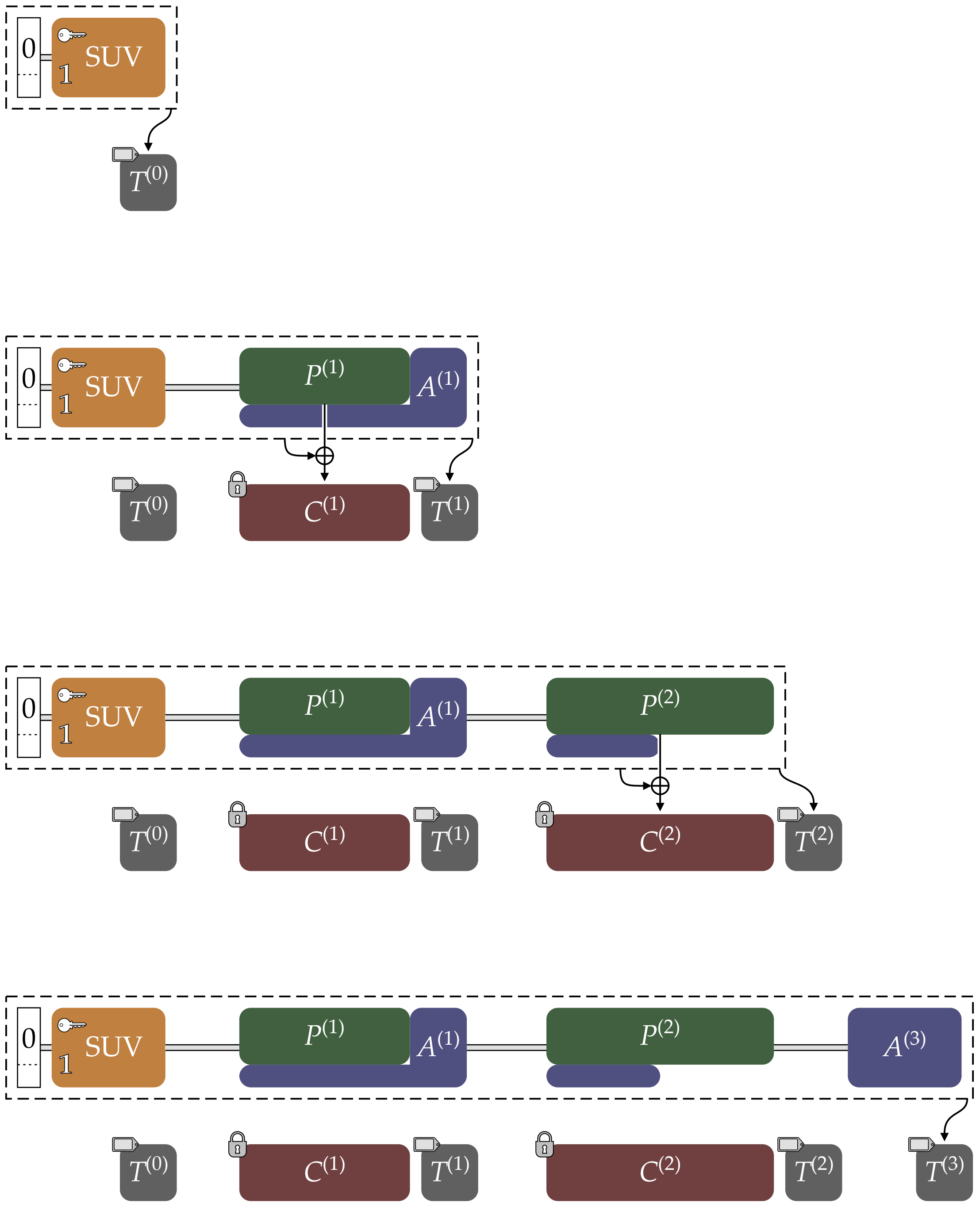

Figure 2.3: A session in MOTORIST.

MOTORIST processes both the plaintext $P^{(1)}$ and additional data $A^{(1)}$ in parallel. The plaintext $P^{(1)}$ is encrypted into ciphertext $C^{(1)}$ and $T^{(1)}$ authenticates $(\mathrm{SUV}, P^{(1)}, A^{(1)})$. After processing the second message, $T^{(2)}$ authenticates $(\mathrm{SUV}, P^{(1)}, A^{(1)}, P^{(2)}, A^{(2)})$, and after the third message, $T^{(3)}$ authenticates the full session $(\mathrm{SUV}, P^{(1)}, A^{(1)}, P^{(2)}, A^{(2)}, P^{(3)}, A^{(3)})$, where $P^{(3)}$ is the empty string.
More formally, the Motorist mode is defined starting from its basic components as:

$$\textsc{Motorist}[f, \Pi, W, c, \tau]$$

where:

- f represents the permutation of the underlying sponge primitive;
- Π is the number of parallel instances, with $1 \leq \Pi \leq 255$ (see *Pistons* in the description below);
- W represents the alignment in bits and it defines the basic atomic size of data, with W a strictly positive multiple of 8;
- c is the required capacity in bits and it sets the trade-off between security and performance;
- τ is the tag length in bits, and it must be a multiple of W.

Securing two-way communication between two parties is a prominent use case for AE where the management of session results being very valuable. In that case, for each message it is necessary to clearly indicate who is the sender and who is the receiver. This can be done by including its identifier in the associated data of the message, or by relying on a strict convention, such as messages alternating in the two directions.
The Motorist mode, being still based on the duplex construction inherits some interesting features, which were already present in the original duplex construction:

- **In-place encryption and decryption**; the encryption/decryption operation coincides with the absorbing operation of the sponge construction. This means that no buffer is necessary and plaintext or ciphertext bits can be processed as they arrive. To preserve this feature in the Motorist mode, the plaintext fragment is limited to the outer part of the input blocks.
- **Sessions**; During a session, a tag of a message authenticates the full sequence of messages since the start of the session and only a single nonce (if any) is required per session.
- **Authentication-only**; the Motorist mode supports the (efficient) generation of tags over messages consisting of additional data only (only authenticated but not encrypted).
- **Stream-compatible**; the Motorist mode does not require prior knowledge of the length of plaintext, ciphertext or additional data.
- **Word-alignment**; the Motorist mode can be instantiated such that it processes data in 64-bit or 32-bit units, without the need for additional bit or byte shuffling.
- **Universal**; the Motorist mode can be applied to any fixed-length permutation with sufficient width.

Besides all the features listed above, the introduction of the Motorist mode allows to further improve the algorithms, especially in term of efficiency. Specifically the new mode provides additional features that increase the efficiency:

- It reduces the computational cost for **short messages**. The original duplex construction required at least two calls to the permutation f, for any (whatever short) message composed of some plaintext: one call for absorbing the (possibly empty) additional data and producing the key stream, and one call for absorbing the plaintext and producing the tag. By supporting output blocks to be used partially as tag and partially as key stream and supporting the combination of metadata and plaintext in a single input block, this can be reduced to one call to f.

- It reduces the computational cost for **long messages**. Thanks to the result described in [8], increasing the length of input blocks from r to b bits has no impact on the generic security bounds that can be proven for the keyed sponge and duplex construction. This allows absorbing up to $c = b - r$ additional bits per call to f.

The new Motorist mode also provides additional features available at application level, which make its use easier and more secure:

- **Tag on session setup**; the setup of a session can return a tag, or can be subject to a tag. This means that when two communicating entities both start a Motorist session, one of them can send the tag to the other party that can then set up the same session on the condition that the tag it receives is valid (for the common SUV). The benefit is that no unwrapping process can start unless a legitimate session is setup.

- **Integrated forgetting**; the mechanism that Motorist uses for making the tag depend on the state of all duplex instances has as side effect that knowledge of the full state does not allow the reconstruction of the state prior to the wrapping (unwrapping) of the current message. It is also supported in the setup of a session and hence a key that is loaded during session setup cannot be recovered from the state.

The Motorist mode, as specified in [29], consists of a layered structure, which, together with the SUV needed for setup, ultimately explain the name of the mode. Three layers have been defined, each one in charge of a specific aspect of the final security service. The layers are, from bottom to top:

- **Piston**. This layer keeps an internal state defined over b bits and applies the permutation f to it. It performs the basic functions such as absorbing data, encrypting or decrypting, and extracting tags. It has a squeezing rate equal to the classical sponge rate, and an absorbing rate that has the same width of the state.

- **Engine**. This layer controls $\Pi \geq 1$ Piston objects that operate in parallel. It serves as a dispatcher keeping its Piston objects busy, imposing that they are all treating the same kind of requests. The Engine also ensures that the SUV and the message sequence can be reconstructed from the sponge input to each Piston object and that each output bit of its Piston objects is used at most once.

- **Motorist**. This layer implements the user interface. It supports the starting of a session and subsequent wrapping and unwrapping of messages by driving the Engine.

The layered structure just explained allows to derive security properties from the underlying basic cryptographic primitives up to the applicative level. The effort of specifying the interface up to the point of being directly and easily usable by the final application specifically aims at preventing all the kind of issues and misuses the characterized many classical solutions for both authenticity and confidentiality of data.

The KEYAK algorithm

There exist five instances of KEYAK, with different trade-offs from lightweight to very high throughput. Each one is defined by specific fixed parameters, the underlying permutation, and has corresponding security goals.
The permutations used in the KEYAK instances are named KECCAK-p permutations and are directly derived from the original KECCAK-f permutations specified in the SHA-3 algorithm from NIST [45]. They are iterated permutations, consisting of a sequence of rounds, which is tunable. A KECCAK-p permutation is defined by its width $b = 25 \times 2^{\ell}$, with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, and its number of rounds n_{r}. Namely, KECCAK-$p[b, n_{\mathrm{r}}]$ consists in the application of the last n_{r} rounds of KECCAK-$f[b]$. When $n_{\mathrm{r}} = 12+2\ell$, KECCAK-$p[b, n_{\mathrm{r}}] =$ KECCAK-$f[b]$.
In order to have a unique way to arrange the bits of the key into the SUV, the *key pack* has been defined within the KEYAK specification. The key pack embeds the key and the other unique values to be used for initialization, with a specific format and encoding. The key pack consists of:

- a first byte indicating the full length of the key pack in bytes;
- the key itself;
- the required padding.

The key pack makes use of *simple padding*: $\mathrm{pad10}^*[r](|M|)$. This padding rule returns a bit string 10^q with $q = (-|M| - 1) \bmod r$. When r is divisible by 8 and M is a sequence of bytes, then $\mathrm{pad10}^*[r](|M|)$ returns the byte string $\texttt{0x01 0x00}^{(q-7)/8}$.
More formally, for a key K, a key pack of ℓ bytes is defined as

$$\mathrm{keypack}(K, \ell) = \mathrm{enc}_8(\ell)||K||\mathrm{pad10}^*[8\ell - 8](|K|),$$

where the length of the key K is limited to $8(\ell - 1) - 1$ bits and with $\ell < 256$.
In general KEYAK makes use of MOTORIST$[f, \Pi, W, c, \tau]$, with f an instance of KECCAK-p. Namely:

$$\text{KEYAK}[b, n_{\mathrm{r}}, \Pi, c, \tau] = \text{MOTORIST}[f, \Pi, W, c, \tau],$$

with $f =$ KECCAK-$p[b, n_{\mathrm{r}}]$ and $W = \max(\frac{b}{25}, 8)$.
The SUV consists of $\mathrm{keypack}(K, \ell_{\mathrm{k}})||N$ with $\ell_{\mathrm{k}} = \frac{W}{8}\left\lceil\frac{c+9}{W}\right\rceil$ and $N \in \mathbb{Z}_2^*$ with no limitation on its length.
Five different instances of KEYAK, have been specified in the submission to the CAESAR competition. Each one is identified by the specific parameter values. Some parameter are the same for all the five instances: the number of rounds (n_{r}) is set to 12, the size of the capacity (c) and then the associated level of security is 256 bits, while the size of the tag (t) is 128 bits. Instead, the KEYAK variants differ for the size of the internal state (b) and the number of multiple instances running in parallel, which both affect the final throughput of the cryptographic primitives.

Name	b	Π
RIVER KEYAK	800	1
LAKE KEYAK	1600	1
SEA KEYAK	1600	2
OCEAN KEYAK	1600	4
LUNAR KEYAK	1600	8

RIVER KEYAK can absorb up to 96 bytes of metadata per permutation call, or up to 68 bytes of plaintext, with additionally 28 bytes of metadata. LAKE KEYAK can absorb up to 192 bytes of metadata per permutation call, or up to 168 bytes of plaintext, with additionally 24 bytes of metadata. For SEA, OCEAN and LUNAR KEYAK, these sizes are multiplied by Π for every Π parallel calls to the permutation.
The security analysis of KEYAK is based on two main pillars:

- Generic security of the Motorist mode: see [29] for a detailed analysis;
- Security assurance from cryptanalysis of KECCAK: thanks to the Matryoshka property, most analysis performed on KECCAK-$f[1600]$ transfers to KECCAK-$f[800]$.

KEYAK provides some features that advantageously compare with classical cryptographic primitives such as AES-GCM. Namely:

- KEYAK supports sessions, differently from most authenticated ciphers, which work on single messages. This means that besides single messages, KEYAK can ensure the authenticity of a sequence of several messages within the same session. A new SUV (key and IV pair) is required per session rather than per message.
- The simple structure of KEYAK results in improved hardware efficiency, when compared with AES-GCM (which requires 2 separate cryptographic primitives) or AES-CCM (which requires 2 calls to the same cryptographic primitive). Furthermore, KEYAK is based on the same primitive as that of SHA-3, therefore allowing to re-use resources when hashing is also needed.
- KEYAK offers robustness against side-channel attacks due to the Motorist construction itself. In addition, the round function can be easily protected against different types of side-channel attacks.

Further details about the underlying permutations, the encodings and the security goals of the KEYAK algorithms and the Motorist mode can be found in the original submission to the CAESAR competition [29].

2.3.2 KETJE

KETJE is a set of four AE functions with support for message-associated data. They aim at memory-constrained devices and the nonce uniqueness is at the base of the security.
The original first version of the KETJE algorithm was submitted as candidate to the CAESAR competition in 2014. KETJE has currently been accepted to the third round of the CAESAR competition.
KETJE mainly targets lightweight use cases, using constrained devices. With this goal, all the possible parameters of the algorithm has been scaled down as much as possible, while still guaranteeing a good level of security.

MonkeyDuplex and MonkeyWrap

The MonkeyDuplex construction [27] is a toolbox aimed at building stream ciphers and AE schemes. It uses a permutation f with a tunable number of rounds. The instance of f with n_r rounds is represented by $f[n_r]$. Similar to the Duplex construction [30], the MonkeyDuplex is stateful and accepts calls taking a string as input and returning a string as output. This output string depends on all inputs received so far. Unlike duplex, MonkeyDuplex supports two types of calls that are different in the number of rounds of f executed between input and output.
The MonkeyDuplex$[f, r, n_{\text{start}}, n_{\text{step}}, n_{\text{stride}}]$ construction works as follows:

- A MonkeyDuplex instance has a state of b bits, where b is the width of the underlying permutation. It starts by initializing the state to the input string I, extended to b bits with padding. Subsequently, it applies $f[n_{Start}]$ to it.
- A bit string σ of up to $r - 2$ bits can be processed for each call. After the bits are injected, either $f[n_{Step}]$ or $f[n_{Stride}]$ is applied to the state and the first ℓ bits of the state are extracted, with $\ell \leq r$.

The MonkeyDuplex construction is illustrated in Figure 2.4.

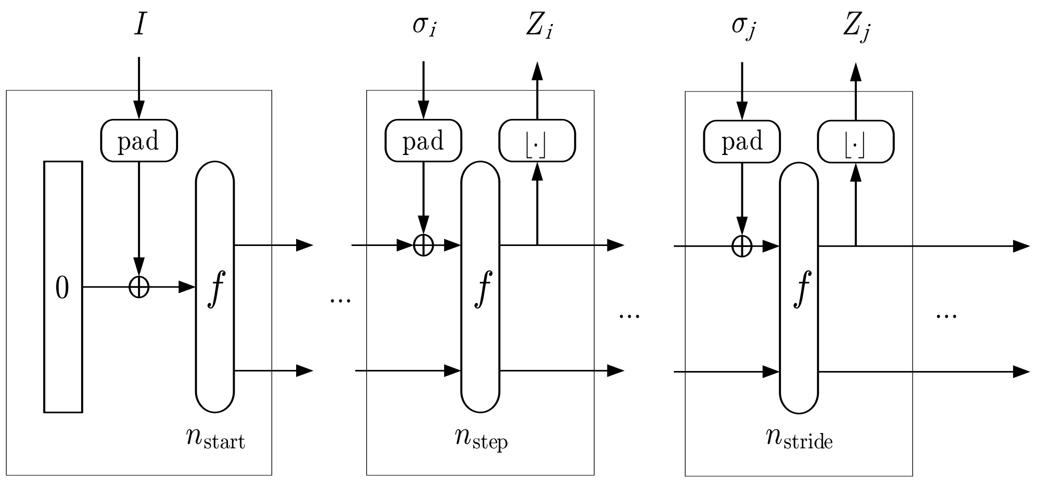

Figure 2.4: The MonkeyDuplex construction.

MonkeyDuplex is meant to be used in a keyed mode. During its start-up it shall be loaded with I containing a secret key and a nonce and during operation an attacker shall not have access to the inner state.
The MonkeyDuplex construction can be used in several use cases; the most relevant in the context of the HECTOR project is authenticated encryption. Similarly to the Motorist mode of operation described in 2.3.1, authenticated encryption can be realized by specifying how to arrange input and output data on top of the MonkeyDuplex. This mode of operation is called MonkeyWrap.
The authenticated encryption process is initialized by loading a key K and a nonce N. The key and nonce are concatenated to form I and a new instance can start. From then on, messages with associated data can be processed. Encryption is done by bitwise addition with the output of the MonkeyDuplex. The inner state depends on all the messages and

associated data presented to the MonkeyDuplex instance. The tag is the output of a MonkeyDuplex call. Note that MonkeyDuplex uses different operations when transitioning to tag generation than in other cases.
Similar to Motorist, MonkeyWrap supports *sessions*, allowing the processing of several messages (each with associated data), where the tag for each message authenticates the full sequence of messages rather than only the message to which it was appended. The requirement of nonce uniqueness plays at the level of the session. Within a session, different messages have no explicit message number or nonce. However, they must be processed in order for the tags to verify. An alternative way to see this concept of session is that the mode supports intermediate tags.

The Ketje algorithm

Ketje builds on Keccak-p permutations, round-reduced versions of Keccak-f, specified in the SHA-3 algorithm from NIST [26]. Keccak-p permutation has a tunable number of rounds, and it is defined by its width $b = 25 \times 2^{\ell}$, with $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, and its number of rounds n_{r}. In particular, for Ketje the twisted permutation Keccak-p^* is used, defined as

$$\text{Keccak-}p^*[b, n_r] = \pi \circ \text{Keccak-}p[b, n_r] \circ \pi^{-1},$$

where π is a step of one round of Keccak-p (for more details on steps of the permutation, see [26]). The purpose of this twist is to effectively re-order the bits in the Keccak state, at each round.
In Ketje specification [28], two different padding rules are considered:

- The *simple padding*, denoted pad10*[r](|M|), returns a bit string 10^q with $q = (-|M|-1) \bmod r$.
- The *multi-rate padding*, denoted pad10*1[r](|M|) returns a bitstring 10^q1 with $q = (-|M|-2) \bmod r$.

Moreover, for a key K, in [28] a *key pack* of ℓ bits is defined as

$$\mathrm{keypack}(K, \ell) = \mathrm{enc}_8(\ell/8)||K||\mathrm{pad10}^*[\ell - 8](|K|),$$

where the key K is at most ($\ell - 9$)-bit long and where ℓ is a multiple of 8 not greater than 255×8. Then, the key pack consists of

- a first byte indicating its whole length in bytes;
- the key itself;
- simple padding.

In general Ketje makes use of MonkeyWrap, calling Keccak-p*. For all four instances, the parameters of the MonkeyDuplex are fixed to $n_{\text{start}} = 12$, $n_{\text{step}} = 1$ and $n_{\text{stride}} = 6$. In order to increasing state sizes, the instances are:

Name	f
Ketje Jr	Keccak-$p^*[200]$
Ketje Sr	Keccak-$p^*[400]$
Ketje Minor	Keccak-$p^*[800]$
Ketje Major	Keccak-$p^*[1600]$

- **KETJE SR** is the more recommended by specification [28]. KETJE SR supports keys K of variable length up to 382 bits and a nonce of length up to $382 - |K|$. It is recommended a key length of 128 bits, but higher key lengths can be adopted as possible countermeasure against multi-target attacks.
- **KETJE JR** supports keys K of length up to 182 bits and a nonce of length up to $182 - |K|$. It is recommended a key length of 96 bits, but higher key lengths can be adopted as a possible countermeasure against multi-target attacks.
- The two last instances, **KETJE MINOR** and **KETJE MAJOR**, exploit the twisted permutation KECCAK-p^* to absorb more lanes per round. Both instances support keys K of length up to $b-18$ bits and a nonce of length up to $b-|K|-18$. It is recommended a key length of 128 bits, but higher key lengths can be adopted as a possible countermeasure against multi-target attacks.

The basic philosophy behind the four KETJE proposals is to maximize the capacity by taking a small rate and compensate the loss of performance by reducing the number of KECCAK-p rounds in the steps calls to a single one, namely $n_{step} = 1$. It is important to observe that the uniqueness of the nonce N is as critical for security as the secrecy of K. Indeed, users are required to use the public message number N as a nonce, i.e., the cipher may lose all integrity and confidentiality if the legitimate key holder uses the same public message number N to encrypt the plaintext and the data under the same key K. More generally, here the indications target the SUV. This means that if the size of the key does not reserve room for the IV, it must be used only once.
KETJE has the following security assurance features:

- Generic security of the mode MONKEYWRAP.
- Security assurance from cryptanalysis of KECCAK.

KETJE provides some features that advantageously compare with classical cryptographic primitive such as AES-GCM.

- KETJE JR, SR and MINOR have a small code and working memory footprint and they require a relatively small amount of computation. KETJE MAJOR require similar amount of operations per bit than KETJE MINOR and it is better adapted to exploiting 64-bit CPUs.
- The implementation of the round function can be used also for the other symmetric cryptographic primitives.
- KETJE offers robustness against side-channel attacks, both in hardware and software. This is of particular importance for constrained devices and smartcards.
- Differently from most authenticated ciphers, KETJE supports sessions, and then sequences of messages can be authenticated rather than a single message.

2.3.3 ASCON

ASCON [49] is a family of authenticated encryption designs ASCON-128$_{a,b}$-k-r designed by Dobraunig et al. In August 2016, ASCON was selected as one of the candidates participating in round 3 of the CAESAR competition [126]. Like KEYAK and KETJE, ASCON is a sponge-based design. However, ASCON targets slightly different use cases, which also motivates a different permutation-based mode of operation. The ASCON family was designed to be lightweight and easy to implement, even with added countermeasures against side-channel attacks. It offers a good trade-off that is efficient in both hardware and software:

- ASCON's small state and simple round function are well-suited for small implementations, without compromising on the full security of 128 bits. Existing lightweight implementations are as small as 2.6 kGE [70]. Comparison of implementation results in [60] show that throughput per area of both ASCON variants is very good compared to many other CAESAR candidates. ASCON is also among the fastest CAESAR candidates for short messages according to current software benchmarking results[9, 20].

- The design of the permutation is well-suited for protected implementations to prevent side-channel attacks. ASCON's SBOX has a low algebraic degree of 2 and a low number of Boolean multiplications, which is well-suited for threshold implementations and similar protection approaches. The bitsliced design means that straightforward software implementations require no data-dependent table look-ups or other cache accesses.

- Compared to other sponge-based constructions, ASCON provides better robustness in case of a potential state recovery, since both initialization and finalization are keyed additionally. Furthermore, ASCON's mode is compatible with alternative decryption interfaces for secure implementations in memory-constrained settings [4].

Tunable parameters include the key size k, the rate r, as well as the number of rounds a for the initialization and finalization permutation p^a, and the number of rounds b for the intermediate permutation p^b processing the associated data and plaintext. The recommended key, tag and nonce length is 128 bits. The sponge state and permutation size is fixed to $320 = 5 \times 64$ bits. The designers recommend two variants, which inject message blocks of 64 or 128 bits, and vary in their number of rounds, as summarized in Table 2.1.

Table 2.1: Recommended parameter configurations for ASCON.

Name	Algorithm	Bit size of				Rounds	
		key	nonce	tag	data block	p^a	p^b
ASCON-128	ASCON-128$_{12,6}$-128-64	128	128	128	64	12	6
ASCON-128a	ASCON-128$_{12,8}$-128-128	128	128	128	128	12	8

ASCON's mode, illustrated in Figure 2.5 is inspired from duplex sponges, but uses a stronger keyed initialization and finalization, which has advantages both from a security perspective and for secure implementation in a low-memory crypto unit. The padding rule for plaintext P and associated data A is similar to KECCAK:

$$P_1, \ldots, P_t \leftarrow \mathrm{pad}_r(P) = r\text{-bit blocks of } P\|1\|0^{r-1-(|P| \bmod r)}$$

$$A_1, \ldots, A_s \leftarrow \mathrm{pad}^*_r(A) = \begin{cases} r\text{-bit blocks of } A\|1\|0^{r-1-(|A| \bmod r)} & \text{if } |A| > 0 \\ \varnothing & \text{if } |A| = 0 \end{cases}$$

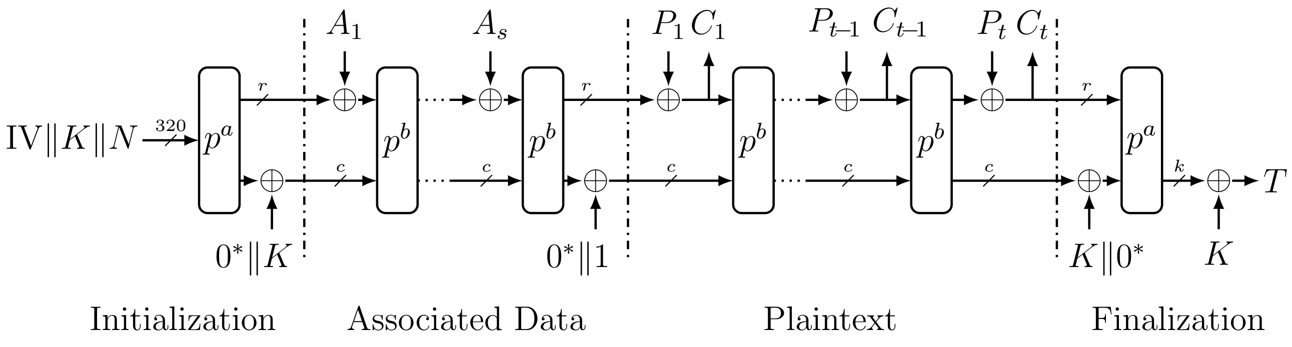

Figure 2.5: ASCON's mode of operation.

The core permutation iteratively applies an SPN-based round transformation with a 5-bit SBOX and a lightweight linear layer. All operations work in a bit-sliced manner on the 5×64-bit words $x_0, \ldots, x_4$:

- A round-constant addition adds a round-dependent value to parts of register word x_2.
- The nonlinear SBOX layer applies a 5-bit SBOX $\mathcal{S}$ 64 times in parallel in a bit-sliced fashion (vertically, across words). This SBOX $\mathcal{S}$ applies the same 5-bit χ function as KECCAK, but preceded and followed by an affine linear operation to improve its differential and linear branch number.
- The linear layer uses an XOR of rotated copies of each 64-bit word for horizontal diffusion within each word, with different rotation values $r_i^{(1)}, r_i^{(2)}$ for each word x_i:

$$x_i := x_i \oplus (x_i \ggg r_i^{(1)}) \oplus (x_i \ggg r_i^{(2)}).$$

For a detailed specification, we refer to the design document [50].
ASCON is designed to provide 128-bit security (confidentiality and integrity of plaintext, associated data, and nonce), as long as the nonce N is never reused to encrypt two messages under the same key. The decryption algorithm may only release the decrypted plaintext after verification of the final tag. The number of processed plaintext and associated data blocks protected by the encryption algorithm is limited to 2^{64} blocks per key. For an overview of the third-party analysis conducted for ASCON, we refer to the designers' website [49].

2.3.4 PRIMATEs

PRIMATEs [6] is a family of single-pass nonce-based algorithms for AE, and a second-round candidate in the CAESAR competition. Members of PRIMATEs are designed for constrained hardware and they differ slightly to achieve various trade-offs between security and performance.
The PRIMATEs family is defined by two parameters:

1. The *security level* denoted by $s \in \{80, 120\}$ bits, which determines the sizes of the binary state b, the rate r and the capacity c, as well as the permutation family PRIMATE-s: $\{0,1\}^b \to \{0,1\}^b$. This is summarized in Table 2.2

2. The *mode of operation* $\in$ APE, HANUMAN, GIBBON that selects between generic constructs designed by following principles from the sponge methodology and MonkeyDuplex. The mode of operation determines the remaining parameters of the algorithm (key length k, tag length t, nonce length n) and the subset of permutations from PRIMATE-s (p_1, p_2, p_3, p_4), as shown in Table 2.3.

Table 2.2: Security levels of PRIMATEs family.

	$s = 80$ bits	$s = 120$ bits
b state size	200 bits	280 bits
c capacity size	160 bits	240 bits
r rate size	40 bits	40 bits
permutation	PRIMATE-80	PRIMATE-120

Table 2.3: PRIMATEs modes of operation.

	APE-s	HANUMAN-s	GIBBON-s
k key size	2s	s	s
t tag size	2s	s	s
n nonce size	s	s	s
PRIMATE	p_1	p_1,p_4	p_1,p_2,p_3

The PRIMATEs's authors recommend the use of HANUMAN for lightweight authenticated encryption, GIBBON for lightweight applications where speed is critical, and APE for lightweight environments where additional security requirements are needed or security is critical. The primary recommended security level is $s = 120$ bits, whereas $s = 80$ bits is suggested for extremely lightweight applications.
The underlying permutation of PRIMATEs is called PRIMATE. It is designed according to the wide trail strategy and its structure resembles the data transform part of the Rijndael block cipher. The PRIMATE permutation operates on a 5×8 (resp. 7×8) state matrix composed by 5-bit elements for PRIMATE-80 (resp. PRIMATE-120). The element in the i^{th} row and j^{th} column of the state matrix is denoted by $a_{i,j}$, where $i \in \{0, \dots, 4\}$ (resp. $i \in \{0, \dots, 6\}$) and $j \in \{0, \dots, 7\}$. The first row $a_{0,*}$ in the state matrix contains the rate of the state, and is henceforth referred to as the rate row. PRIMATE updates the state by using a sequence of four transformations described as follows:

1. *SubElements* (SE) is the only non-linear transformation. It consists of an element-wise permutation $X \to S(X) : \{0,1\}^5 \to \{0,1\}^5$ (SBOX) applied to each element of a state.

2. *ShiftRows* (SR) performs cyclical shifts of each row for a different number of elements. Row i is shifted left by $s_i = \{0,1,2,4,7\}$ in PRIMATE-80, or by $s_i = \{0,1,2,3,4,5,7\}$ in PRIMATE-120.

3. *MixColumns* (MC) operates on a state column at a time. It is a left-hand multiplication by a 5×5 (7×7) Maximum Distance Separable (MDS) matrix. The matrices are chosen in a way that allows recursive calculation of a smaller matrix five (resp. seven) times.

4. *ConstantAddition* (CA) modifies a single state element $a_{1,1}$ by bitwise XOR-ing a 5-bit constant in each round.

Round constants are generated by a 5-bit Fibonacci LFSR. Varying on the sequence of values sampled from this LFSR and the number of rounds, four permutations p_1, p_2, p_3, and p_4 are derived from the core permutation (either PRIMATE-80 or PRIMATE-120), as shown in Table 2.4.

Table 2.4: PRIMATE permutations.

PRIMATE	p_1	p_2	p_3	p_4
Number of rounds	12	6	6	12
LFSR initial value	1	24	30	24

For more information about other slight differences between the different modes of operation, we refer the reader to the official specification submitted to the CAESAR competition [6], which details also the security claims and security analysis of PRIMATEs.

2.4 AE implementations in Hardware

2.4.1 Hardware interface

In the context of CAESAR competition professor Gaj from George Mason University (GMU) has proposed a standard interface in order to benchmark the hardware implementations of the different algorithms. This proposal was first called GMU hardware API and after some discussions, in the CAESAR mailing list, it was officially adopted by the CAESAR competition and named CAESAR HW API. The purpose of this API is to perform a fair and easy benchmark of the different proposals. For this reason, the interface is generic in such a way that any submission can be measured in this framework. The consequence is that a primitive based on a block cipher needs two separate inputs for data and key. On the other hand, most of the primitives based on permutations use the key only at the beginning of the computation. In order of being compliant to the API a dedicated port for the key is needed in any instances and the wrapper would store the key in dedicated register. This implies a major amount of resources and primitives based on permutation are somehow more negatively affected in this context. In short, if the generic interface is used there is an overhead of resources. Recently an improvement of the API for tackling lightweight use cases has been proposed (DIA2016), the improvement is very good but still a dedicated API for specific algorithm is preferable compared to a benchmark driven interface

2.4.2 KEYAK

In this subsection, we discuss about KEYAK implementations. The VHDL code of KEYAK implementations is publicly available on github [22]. Since the target of KEYAK is high speed and not implicitly low area, the overhead of the CAESAR HW API is not so problematic and thus the implementation is compliant with it. In Table 2.5 we report the result for the RIVER and LAKE variants of KEYAK. The other variants targeting parallelism are more interesting for CPUs with multi core or SIMD instructions and less suitable for hardware implementations; i.e. hardware implementations would have difficulty in sustaining the input throughout of the 2, 4 or 8 parallel cores. For this reason, we are not investigating these variants. Looking at the results it is interesting to note that area for registers and for round logic are scaling linearly between the RIVER and the LAKE proposal since the size of the permutation doubles

Table 2.5: Implementation figures of different KEYAK ASIC hardware implementations using the CAESAR hardware API. (Note that the core area includes overhead to be compliant with the CAESAR hardware API.)

Version	Area (full) [kGE]	Area (registersl) [kGE]	Area (round) [kGE]	Throughput [Mbps]
KEYAK RIVER	23.7	4.5	6	4500
KEYAK LAKE	52.5	7.5	12	11200

from 800 to 1600 bits. The overhead in the management of the API takes some area but it remains reasonable. It is still possible to increase the throughput via round duplication or triplication. Since the round represents 15 to 20 percent of the cost, we will have an interesting improvement. These area figures are obtained using a 90 nm ST technology with a clock speed of 100MHz, and in case of need the cores allow to increase the frequency.

2.4.3 KETJE

The Ketje algorithm is a proposal targeting lightweight systems. For this reason, it is interesting to explore the implementations compliant with GMU API but also an optimized with a custom interface. The custom interface is not only interesting for reducing footprint but also for introducing additional functionalities. The mode presented with Ketje allows to adopt session oriented streaming, thus the key and nonce are absorbed at the beginning of the session and a sequence of packets can be elaborated inside a session without any need of reusing the key. This session approach is not supported natively in the GMU API. Additionally the underlying permutation can be used also for implementing a hash function and thus basically have a complete proposal of symmetric cipher suite.

Table 2.6: Implementation figures of different KETJE ASIC hardware implementations using the CAESAR hardware API. (Note that the core area includes overhead to be compliant with the CAESAR hardware API.)

Version	Area (full) [kGE]	Area (registersl) [kGE]	Area (round) [kGE]	Throughput [Mbps]
KETJE JR.	5.7	1.7	1.5	1600
KETJE SR.	9.7	3.3	2.7	3200

Table 2.7: Implementation figures of different KETJE ASIC hardware implementations using a custom interface.

Version	Area (full) [kGE]	Area (registersl) [kGE]	Area (round) [kGE]	Throughput [Mbps]
KETJE JR. v2	4.3	1.0	1.5	1600
KETJE SR. v2	8.3	2.0	2.7	3200
KETJE MINOR	17.4	1.5	6.0	12800
KETJE MAJOR	26.8	8.7	12.1	25600

The implementation compliant to the GMU API are related to the first version of the KETJE proposal, thus related to the SR. and JR. versions only, while the implementations with

a custom API are in line with the version 2 of the proposal with the two new members KETJE MINOR and KETJE MAJOR. Synthesis results for ASIC with or without GMU API are reported in Table 2.6 and Table 2.7 The KETJE SR. represent a very interesting trade-off in terms of speed and limited requirement of resources. In order to increase the performances, similarly to KEYAK, it is possible to instantiate more rounds executed in one clock cycle. The interesting figure of KETJE is also the capability of processing one 32 bit word per clock cycle, making it a perfect fit for embedded systems that are largely using a 32 bit architecture. The instantiation of more rounds would increase the size of the data with the consequence of requiring some sort of buffering. When comparing hardware with GMU or without GMU interface it is possible to note that round logic costs are constant, as expected, while there is a small increase in registers and a non-negligible increase for additional logic. From the values in the tables, it is possible to see a cost increase of about 1.5 kGE, that is particular annoying for the low-end version since it represents about 30% area increase.
Although the implementations are for an ASIC and not for a Microsemi FPGA, the implementation figures can serve as a first indicator which version is the best candidate for integration on the target FPGA platform. HECTOR Deliverable D3.2 provides area figures for KETJE in Microsemi FPGA.

2.4.4 ASCON

A number of hardware implementations can be found on the Ascon website[49]. All hardware implementations are publicly available from github [51]. As in the case of KETJE, area figures for ASCON in Microsemi FPGA are available in the HECTOR Deliverable D3.2. In the following Table 2.8 we list the implementation figures of five versions of Ascon. Ascon-fast 1 round is the reference with a rate of 128 bit and calculates 1 round per clock cycle. Ascon-fast 2 rounds (Ascon-fast 4 rounds) calculate 2 (4) rounds per clock cycle to increase the throughput. Ascon-64bit and Ascon-x-low-area have a rate of 64bit. Ascon-x-low-area is designed for a small area footprint.

Table 2.8: Implementation figures of different Ascon ASIC hardware implementations using a custom, lightweight interface. (^{1}...Ascon-128a: 128bit rate; ^{2}...Ascon-128: 64bit rate).

Version	Area [GE]	Throughput [Mbps]	Power@1MHz [μW]
Ascon-fast 1 round^1	7320	7597	47
Ascon-fast 2 rounds^1	10857	11232	74
Ascon-fast 4 rounds^1	17991	16097	180
Ascon-64bit^2	4990	72	32
Ascon-x-low-area^2	2570	17	15

All the implementations listed in Table 2.8 use a custom and lightweight interface in order to minimize the required resources. In order to compete in the CAESAR competition, it is also required to use a specific interface, the CAESAR HW API. As already mentioned, this interface ensures a high degree of flexibility, but it also adds a significant overhead in terms of area. Comparing the area numbers from Table 2.8 and Table 2.9 shows that the CAESAR hardware API adds additional 6kGE-8kGE in area. Therefore we conclude that a custom interface is the better choice for the integration of the AE algorithm into the demonstrator. Comparing the numbers from Table 2.8 (Area) and Table 2.9 (Area (core)) reveals that the Ascon core for the CAESAR hardware API versions already requires more area compared to

Table 2.9: Implementation figures of different Ascon ASIC hardware implementations using the CAESAR hardware API. (Note that the core area includes overhead to be compliant with the CAESAR hardware API.)

Version	Area (core) [GE]	Area (full) [GE]	Throughput [Mbps]
Ascon-fast 1 round	9680	13351	7326
Ascon-fast 2 rounds	13249	16920	11743
Ascon-fast 4 rounds	20380	24051	16675

the Ascon version including the custom interface. This is due to the additional features (e.g. error handling) that have to be added to the core to be compliant to the CAESAR hardware API.

2.4.5 PRIMATEs

As in all Sponge-based designs, the majority of implementation cost of PRIMATEs comes from its core permutations. We have implemented round-based and serialized versions of both PRIMATE-80 and PRIMATE-120. In the following, we denote each implementation by P80-x and P120-x, where x indicates the number of cycles per round.
A brief description of each implementation follows:

- *P80-1 and P120-1* correspond to the round-based architectures where all transformations (SBOXes, MDS matrix multiplications and constant addition) are implemented as combinational networks, while the SR transformation is simply a rewiring of rows.
- *P80-9 and P120-9* correspond to the 9 clock cycle serial implementation, where the SRF has only two modes of operation: MC and SR. When MC is active SRF is configured as a 25-bit FIFO register which feeds the data into the combinational network at its output. This mode is used for data input, as well. SR mode is always active during the first cycle of computation, during which it rewires the SRF to perform the SR transformation.
- *P80-41 and P120-57* serializes the MC step by performing the 5 (resp. 7) matrix multiplications in the MC transformation in 5 (resp. 7) clock cycles for PRIMATE-80 (resp. PRIMATE-120).
- *P80-16 and P120-16* correspond to the 16 clock cycle serial implementation where the SR transformation is serialized instead of the MC transformation.

Table 2.10 shows the synthesis results of our PRIMATE implementations using Synopsys Design Compiler v2015.06 and 2 different standard-cell libraries: Faraday UMC 90 nm (`UMC 90`) and NangateOpenCellLibrary 45 nm (`NAN 45`)). We have used Synopsys PrimeTime v2015.06 with PX add-on to perform more accurate static timing analysis and switching activity based power estimation. We also provide some figures at the operating frequency of 100 kHz.
Lastly, we have developed a co-processor architecture, which can be used for all PRIMATEs. It is designed to be compatible with 8- and 16-bit processors and features a generic interface (although different than the CAESAR HW API). We have used this interface to implement a

Design	Library	Max. Freq. [MHz]	Area [kGE]	T'put [$\frac{\text{Mbit}}{\text{s}}$]	Impl. Eff'cy [$\frac{\text{Mbit}}{\text{kGE}\cdot\text{s}}$]	@ 100 kHz D. Pwr. [μW]	S. Pwr. [nW]	E. Eff'cy [$\frac{\text{pJ}}{\text{bit}}$]
P80-1	UMC90	179.60	3.68	20.00	5.43	2.32	74.00	0.12
	NAN45	341.53	4.72		4.24	1.63	83.30	0.09
P80-9	UMC90	256.74	1.43	2.22	1.56	0.74	29.80	0.35
	NAN45	439.77	2.05		1.08	0.78	32.80	0.37
P80-16	UMC90	509.50	1.20	1.25	1.04	0.68	25.20	0.57
	NAN45	896.38	1.78		0.70	0.42	27.60	0.36
P80-41	UMC90	204.18	1.32	0.49	0.37	0.46	26.70	0.99
	NAN45	267.61	1.98		0.25	0.30	31.80	0.68
P120-1	UMC90	142.27	6.32	28.00	4.42	4.61	137.00	0.17
	NAN45	281.31	8.23		3.51	3.65	159.00	0.14
P120-9	UMC90	183.69	2.17	3.11	1.43	1.26	46.00	0.42
	NAN45	490.17	3.10		1.00	1.17	165.00	0.43
P120-16	UMC90	447.21	1.82	1.75	0.96	1.13	38.60	0.67
	NAN45	722.33	2.69		0.65	0.80	42.60	0.48
P120-57	UMC90	114.32	1.87	0.49	0.26	0.63	36.80	1.37
	NAN45	239.24	2.79		0.18	0.40	44.80	0.91

Table 2.10: Post-synthesis hardware implementation results of PRIMATE permutations.

complete design of HANUMAN-80 with the P80-9 core, which is the most efficient of all serialized versions (at the cost however of increased area and power).

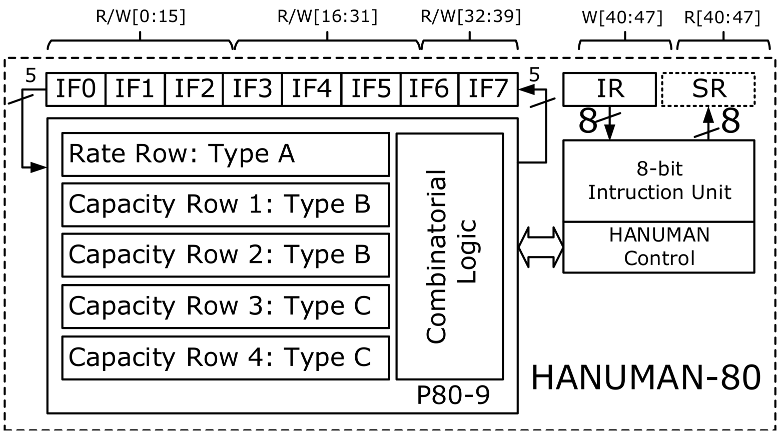

Figure 2.6: HANUMAN-80 coprocessor architecture.

The architecture of our design is depicted in Figure 2.6. We have used a Spartan-6 FPGA (XC6SLX45-3CSG324) to implement and test this design, using an OpenMSP430 implementation from the popular MSP430 microcontroller family. On this platform the whole coprocessor fits in a total of 72 (1.06%) slices (206 FFs and 278 LUTs). In ASIC, using UMC 90 standard-cell library, the entire coprocessor requires 2 kGE. Note that HANUMAN-80 compliant P80-9 requires 1.69 kGE. The overhead of 0.26 kGE (18.68% larger than the raw P80-9 core of 1.43 kGE) includes all the glue logic; and entire control logic, including the FSM of the coprocessor for fetching, decoding, and executing micro-instructions. An extra overhead of

0.31 kGE is introduced for the 8-bit instruction unit and the 40-bit IF register, which enables circular access to SRF in a block-pipeline manner, allowing to almost negligible interface overhead.

2.5 Improvements towards Efficiency

We described in section 2.2 the fact that the same cryptographic primitive (i.e. the cryptographic sponge construction) can serve different purposes and that it can be used for different security services required by applications. We also showed that in order to instantiate such a cryptographic sponge, only one main building block is necessary: a suitable permutation (as defined in 2.2.1). This overall translates in the possibility to significantly improve the efficiency of the security part of an application. This becomes clear when compared with the set of previously available cryptographic tools, which needed to be combined together when used in most of the current practical applications.

Efficiency benefits from the cryptographic sponge constructions in several tangible aspects. First of all, of course, having a single primitive rather than several ones for different tasks allows to save hardware resources. For instance, there is no more the need to put together a symmetric cipher and a hash function in the same device. In essence, a single sponge-based hardware IP can cover any security need, except for asymmetric cryptographic schemes and for non-digital ones (e.g. TRNG, PUF).
In addition to that, since there is no more need to compose together different primitives to accomplish a task, the possibility to insert vulnerabilities in the application is reduced. As a warning, we explained in 2.1 for instance how the combination of encryption and MAC generation led to catastrophically undermine the security of some largely used communication protocols. Besides pure security-related considerations, less building blocks translates in simplified integration of these blocks together. Moreover, reduced complexity helps preventing inefficiencies in the system.

Such simplification positively affects different aspects of the system on two different dimensions:

- from the high-level definition of a security scheme to its actual implementation;
- from the cryptanalysis of the underlying permutation up to the security analysis of the whole system built on top of it.

For instance, in case of side-channel protections, security researchers can focus on refining the protection of a single permutation, since it will then be used for a multiplicity of services. In contrast, it would be much harder to build a system composed of completely different primitives (e.g. AES and SHA-2-256) that is also balanced from the side-channel protection viewpoint. The following chapters of this document will show how it can be possible to build a side-channel protected primitive and how such protection can be evaluated at design-time. One of the goals of the HECTOR project is to showcase the benefits in term of efficiency and simplicity that can be brought to some practical use cases, such as the ones addressed by the demonstrators 2 and 3 in the WP4.

Chapter 3

System-level Vulnerabilities and Countermeasures

In this chapter we discuss vulnerabilities and corresponding countermeasures for devices working with confidential data. These confidential data can e.g. be memory content only accessible to authorized parties or communication data encrypted with some secret key to avoid eavesdropping. The goals of an attacker in the aforementioned scenarios are typically reading the confidential memory content or revealing the secret key that secures a communication.

In Section 3.1 we start with some general considerations on side-channel attacks targeting embedded devices and corresponding countermeasures.

Section 3.2 provides a state-of-the-art side-channel attack classification. In the first place, this topic is discussed from a top-level view and then converges down to physical attacks, which can be mounted if the attacker is in physical possession of the targeted device. This subtopic is then further split into passive and active physical attacks. For active physical attacks, we discuss one recently published attack, a so-called statistical fault attack targeting authenticated encryption algorithms, in detail.

Section 3.3 discusses relevant countermeasures which allow to increase the security of a device against side-channel attacks. This section is split up into two parts. First, implementation-level countermeasures are discussed in detail. This type directly modifies the implementation of cryptographic primitives for minimizing side-channel leakage. We also present a recently published masking scheme. Second, the focus is put on protocol-level countermeasures. Here, the protocol itself is modified in a way to minimize the exploitable side-channel leakage available to an attacker. In this context, we discuss an authenticated encryption scheme inherently secure against passive side-channel attacks.

3.1 Rationale

A huge amount of electronic devices nowadays store and process confidential data. To ensure the confidentiality and also the integrity of this data, well-proven and mathematically secure encryption algorithms are applied. These algorithms allow to encrypt and decrypt the data with a secret key. In order to get unauthorized access to the confidential data, the key poses an interesting target for attackers. Especially in scenarios, where the electronic device might be in physical possession of an attacker for a short period of time, special care has to be taken when implementing the cryptographic algorithm.

During the execution of the cryptographic algorithm the attacker can measure side-channel information like power consumption or electromagnetic emanation. This side-channel leakage can then be used to reveal the secret key. In order to limit the exploitable side-channel leakage, the implementation of the cryptographic algorithm can be equipped with countermeasures. However, the increased security level does not come for free. Countermeasures introduce additional overhead in terms of chip area, power consumption, runtime, and the requirement for random numbers. Therefore, the integration of countermeasures is a trade-off between intended security level and implementation overhead.

3.2 Attacks

In this section, we discuss important aspects of side-channel attacks, which have become a busy research field during the last twenty years. For the industry, it is important to keep pace with the evolution of always new attack strategies to guarantee the security of their products. For the remainder of this section we give a high-level classification of state-of-the-art side-channel attacks. This high-level view is then followed by a more-detailed discussion of passive and active physical attacks, which pose the most-serious threat within the HECTOR project context.

3.2.1 Classification of Side-Channel-Attacks

Side-channel attacks exploit (unintended) information leakage of computing devices or implementations to infer sensitive information. Starting with the seminal works of Kocher [82], Kocher et al. [83], Quisquater and Samyde [109], as well as Mangard et al. [88], many follow-up papers considered attacks against cryptographic implementations to exfiltrate key material from smartcards by means of timing information, power consumption, or electromagnetic (EM) emanation. Although "traditional" side-channel attacks required the attacker to be in physical possession of the device, different attacks assumed different types of attackers and different levels of invasiveness. To systematically analyse side-channel attacks, they have been categorized along the following two orthogonal axes:

1. *Active* vs *passive*: depending on whether the attacker actively influences the behaviour of the device or only passively observes leaking information.
2. *Invasive* vs *semi-invasive* vs *non-invasive*: depending on whether the attacker removes the passivation layer of the chip, depackages the chip, or does not manipulate the packaging at all.

With the era of cloud computing, the scope and the scale of side-channel attacks have changed significantly in the early 2000s. While early attacks required attackers to be in physical possession of the device, newer side-channel attacks, for example, cache-timing attacks [129, 134, 71, 61] or DRAM row buffer attacks [106], are conducted remotely by executing malicious software in the targeted cloud environment. In fact, the majority of recently published side-channel attacks rely on passive attackers and are strictly non-invasive.
With the advent of mobile devices, and in particular the plethora of embedded features and sensors, even more sophisticated side-channel attacks targeting smartphones have been proposed since around the year 2010. For example, attacks allow to infer keyboard input on touchscreens via sensor readings from native apps [41, 11, 119] and websites [92],

to deduce a user's location via the power consumption available from the proc filesystem (procfs) [93], and to infer a user's identity, location, and diseases [137] via the procfs.
Although side-channel attacks and platform security are already well-studied topics, it must be noted that smartphone security and associated privacy aspects differ from platform security in the context of smartcards, desktop computers, and cloud computing. Especially the following *key enablers* allow for more devastating attacks on mobile devices.

1. *Always-on and portability*: First and foremost, mobile devices are always turned on and due to their mobility they are carried around at all times. Thus, they are tightly integrated into our everyday life.

2. *Bring your own device (BYOD)*: To decrease the number of devices carried around, employees are encouraged to use private devices to process corporate data and to access corporate infrastructure, which clearly indicates the importance of secure mobile devices.

3. *Ease of software installation*: Due to the appification [3] of mobile devices, *i.e.*, where there is an app for almost everything, additional software can be installed easily by means of established app markets. Hence, malware can also be spread at a fast pace.

4. *OS based on Linux kernel*: Modern mobile operating systems (OS), for example, Android, are based on the Linux kernel. The Linux kernel, however, has initially been designed for desktop machines and information or features that are considered harmless on these platforms turn out to be an immense security and/or privacy threat on mobile devices (cf. [136]).

5. *Features and sensors*: Last but not least, these devices include many features and sensors, which are not present on traditional platforms. Due to the inherent nature of mobile devices (always-on and carried around, inherent input methods, etc.), such features often allow for devastating side-channel attacks [41, 11, 119, 92, 93, 137]. Besides, these sensors have also been used to attack external hardware, such as keyboards [89, 138], and computer hard drives [32], to infer videos played on TVs [118], and even to attack 3D printers [122, 76], which clearly demonstrates the immense power of mobile devices.

Today's smartphones are vulnerable to (all or most of the) existing side-channel attacks against smartcards and cloud computing infrastructures. However, due to the above mentioned *key enablers*, a new area of side-channel attacks has evolved. The appification [3] of mobile platforms—*i.e.*, where there is an app for anything—allows to easily target devices and users at an unprecedented scale compared to the smartcard and the cloud setting. Yet again, the majority of these attacks are passive and non-invasive, which means that the existing side-channel classification system is not appropriate anymore as it is too coarse grained for a systematic categorization of modern side-channel attacks against mobile devices.
We refer to [110] for an in-depth discussion of a new categorization system for modern side-channel attacks on mobile devices. For the remainder of this document we keep the focus of the discussion on traditional side-channel attacks (passive and active physical attacks) which can be identified as the main threat within the HECTOR project context.

3.2.2 Basic Concept of Side-Channel Attacks

Passive Side-Channel Attacks. The general notion of a passive side-channel attack can be described by means of three main components, *i.e.*, *target*, *side-channel vector*, and *attacker*. A *target* represents anything of interest to possible attackers. During the computation or operation of the target, it influences a *side-channel vector* and thereby emits potential sensitive information. An *attacker* who is able to observe these side-channel vectors potentially learns useful information related to the actual computations or operations performed by the target.

Active Side-Channel Attacks. An active attacker tries to tamper with the device or to modify/influence the targeted device via a side-channel vector, e.g., via an external interface or environmental conditions. Thereby, the attacker aims to influence the computation/operation performed by the device in a way that leads to malfunctioning, which in turn allows for possible attacks either indirectly via the leaking side-channel information or directly via the (erroneous) output of the targeted device.

Figure 3.1 illustrates the general notion of side-channel attacks. A target emits specific side-channel information as it influences specific side-channel vectors. This can be the physical movement captured by a sensor of a device during an input on a touchscreen (e.g. password input) or more traditional the power consumption or EM emanation of a device while executing some cryptographic algorithm. The relations defined via the solid arrows, *i.e.*, *target* $\rightarrow$ *side-channel vector* $\rightarrow$ *attacker*, represent passive side-channel attacks. The relations defined via the dashed arrows, *i.e.*, *target* $\leftarrow$ *side-channel vector* $\leftarrow$ *attacker*, represent active side-channel attacks where the attacker tries to actively influence/manipulate the target via a side-channel vector. By tampering with the supply voltage, the clock signal or by injecting EM pulses, an active manipulation of the device can be achieved.

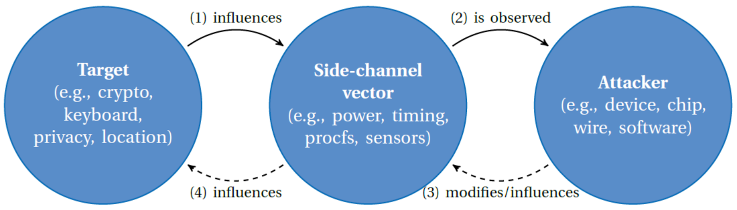
(1) influences
(2) is observed
Target (e.g., crypto, keyboard, privacy, location)
Side-channel vector (e.g., power, timing, procfs, sensors)
Attacker (e.g., device, chip, wire, software)
(4) influences
(3) modifies/influences

Figure 3.1: General notion of active and passive side-channel attacks. A passive side-channel attack consists of steps (1) and (2), whereas an active side-channel attack also includes steps (3) and (4).

3.2.3 Types of Side-Channel Information Leaks

Considering existing side-channel attacks on mobile devices, we identify two categories of side-channel information leaks, namely *unintended information leaks* and *information published on purpose*. Figure 3.2 depicts these two types of information leaks. Informally, side-channel attacks exploiting unintended information leaks of computing devices can also be considered as “traditional” side-channel attacks since this category has already been extensively analysed and exploited during the smartcard era [88]. For example, unintended information leaks include the execution time, the power consumption, or the electromagnetic emanation of a computing device. This type of information leak is considered as unintended

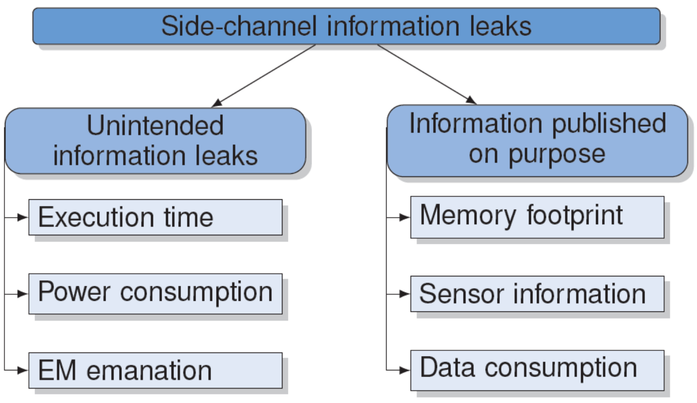

Figure 3.2: Categorization of side-channel information leaks.

because smartcard designers and developers did not plan to leak the timing information or power consumption of computing devices on purpose.
The second category of information leaks (referred to as *information published on purpose*) is mainly a result of the ever-increasing number of features provided by today's smartphones. In contrast to unintended information leaks, the exploited information is published on purpose and for benign reasons. For instance, specific features require the device to share (seemingly harmless) information and resources with all applications running in parallel on the system. This information as well as specific resources are either shared by the OS directly (via the procfs) or through the official Android API.[1] Although this information is extensively used by many legitimate applications for benign purposes[2], it sometimes turns out to leak sensitive information and, thus, leads to devastating side-channel attacks. The fundamental design weakness of assuming information as being innocuous in the first place also means that it is not protected by dedicated permissions. Many investigations have impressively demonstrated that such seemingly harmless information can be used to infer sensitive information that is otherwise protected by dedicated security mechanisms, such as permissions. Examples include the memory footprint [79] and the data-usage statistics [123] that have been shown to leak a user's browsing behaviour and, hence, bypass the `READ_HISTORY_BOOKMARKS` permission.
Furthermore, the second category seems to be more dangerous in the context of smartphones as new features are added frequently and new software interfaces allow to access an unlimited number of unprotected resources. Even developers taking care of secure implementations in the sense of unintended information leaks, e.g., by providing constant-time crypto implementations, and taking care of possible software vulnerabilities like buffer overflow attacks, inevitably leak sensitive information due to shared resources, the OS, or the Android API. Additionally, the provided software interfaces to access information and shared resources allow for so-called *software-only attacks*, *i.e.*, side-channel attacks that only require the execution of malicious software. This clearly represents an immense threat as these attacks (1) do not exploit any obvious software vulnerabilities, (2) do not rely on specific privileges or permissions, and (3) can be conducted remotely via malicious apps or even websites.

[1] In the literature some of the information leaks through the procfs are also denoted as *storage side-channels* [132].

[2] For example, the data-usage statistics, *i.e.*, the amount of incoming and outgoing network traffic, is publicly available for all applications.

The HECTOR USB device is on the one hand also a mobile device with input (keypad) and output (display) functionality, but the use is more restricted compared to smartphones. It is e.g. not possible to launch user apps on the HECTOR USB device nor it is possible to read sensor data. Therefore the most important attack scenarios are local side-channel attacks, both passive and active, relying on unintended information leaks. These kind of attacks will be discussed in the following sections in more detail[3].

3.2.4 Passive Physical Attacks

Passive attacks only observe leaking information without actively influencing or manipulating the target. In the context of passive physical attacks one has to distinguish *Simple Power Analysis (SPA)* from *Differential Power Analysis (DPA)*. In an SPA scenario the attack succeeds with a small number of measurements (maybe even with a single measurement) while a DPA scenario requires a high amount of measurements (depending on the target this number ranges from a few thousand to several billion measurements). Hence, DPA attacks require the observation of a high number of executions using the same secret to succeed. Below we survey some passive side-channel attacks that require a local adversary.

Power Analysis Attacks. The actual power consumption of a device or implementation depends on the processed data and executed instructions. Power analysis attacks exploit this information leak to infer sensitive information.

Traditional side-channel attacks exploiting the power consumption of smartcards [88] have also been applied on mobile devices. For instance, attacks targeting symmetric cryptographic primitives [13] as well as asymmetric primitives [63, 67, 16] have been successfully demonstrated. Such attacks have even been conducted with low-cost equipment, as has been impressively demonstrated by Genkin et al. [63]. Furthermore, the power consumption of smartphones allows to identify running applications [133].

Electromagnetic Analysis Attacks. Another way to attack the leaking power consumption of computing devices is to exploit electromagnetic emanations. Gebotys et al. [62] demonstrated attacks on software implementations of AES and ECC on Java-based PDAs. Later on, Nakano et al. [97] attacked ECC and RSA implementations of the default crypto provider (JCE) on Android smartphones and Belgarric et al. [16] attacked the ECDSA implementation of Android's Bouncy Castle.

3.2.5 Active Physical Attacks

Besides passively observing leaking information, an active attacker can also manipulate the target, its input, or its environment in order to subsequently observe leaking information via abnormal behaviour of the target (cf. [88]). Most active attacks that require the attacker to be physically present with the attacked device have been investigated in the smartcard setting. Only few of these attacks are investigated on larger systems like smartphones.

Clock/Power Glitching. Variations of the clock signal, e.g., overclocking, have been shown to be an effective method for fault injection on embedded devices in the past. One prerequisite for this attack is an external clock source. Microcontrollers applied in smartphones typically have an internal clock generator making clock tampering impossible. Besides clock tampering, intended variations of the power supply represent an additional method for fault

[3]For the interested reader, all the other side-channel attacks which pose a serious threat for mobile devices like smartphones, are discussed in detail in the work ”Systematic Clssification of Side-Channel Attacks on Mobile Devices”, `https://arxiv.org/pdf/1611.03748.pdf`

injection. With minor hardware modifications, power-supply tampering can be applied on most microcontroller platforms.
In [98] it is shown how to disturb the program execution of an ARM CPU on a *Raspberry PI* by underpowering, *i.e.*, the supply voltage is set to GND for a short time. Tobich et al. [127] take advantage of the so-called *forward body bias injection* for inducing a fault during a RSA-CRT calculation. Due to the relatively easy application on modern microcontrollers, voltage-glitching attacks pose a serious threat for smartphones if attackers have physical access to the device. This has been demonstrated by O'Flynn [101] for an off-the-shelf Android smartphone.
Electromagnetic Fault Injection (EMFI). Transistors placed on microchips can be influenced by electromagnetic radiation. EMFI attacks take advantage of this fact. These attacks use short (in the range of nanoseconds), high-energy EM pulses to, e.g., change the state of memory cells resulting in erroneous calculations. In contrast to voltage glitching, where the injected fault is typically global, EMFI allows to target specific regions of a microchip by precisely placing the EM probe, e.g., on the instruction memory, the data memory, or CPU registers. Compared to optical fault injection, EMFI attacks do not necessarily require a decapsulation of the chip, making them less invasive and thus more practical.
Ordas et al. [102] report successful EMFI attacks targeting the AES hardware module of a 32 bit ARM processor. Rivière et al. [113] use EMFI attacks to force instruction skips and instruction replacements on modern ARM microcontollers. Considering the fact that ARM processors are applied in modern smartphones, EMFI attacks represent a serious threat for such devices.
Laser/Optical Faults. Optical fault attacks using a laser beam are among the most-effective fault-injection techniques. These attacks take advantage of the fact that a focused laser beam can change the state of a transistor on a microcontroller resulting in, e.g., bit flips in memory cells. Compared to other fault-injection techniques (voltage glitching, EMFI), the effort for optical fault injection is high. (1) Decapsulation of the chip is a prerequisite in order to access the silicone with the laser beam. Besides, (2) finding the correct location for the laser beam to produce exploitable faults is also not a trivial task.
First optical fault-injection attacks targeting an 8-bit microcontroller have been published by Skorobogatov and Anderson [121] in 2002. Inspired by their work, several optical fault-injection attacks have been published in the following years, most of them targeting smart-cards or low-resource embedded devices (e.g. [130], [117]). The increasing number of metal layers on top of the silicone, decreasing feature size (small process technology), and the high decapsulation effort make optical fault injection difficult to apply on modern microprocessors used in smartphones.
NAND Mirroring. Data mirroring refers to the replication of data storage between different locations. Such techniques are used to recover critical data after disasters but also allow to restore a previous system state.
The Apple iPhone protects user's privacy by encrypting the data. Therefore, a passcode and a hardware-based key are used to derive various keys that can be used to protect the data on the device. As a dedicated hardware-based key is used to derive these keys, brute-force attempts must be done on the attacked device. Furthermore, brute-force attempts are discouraged by gradually increasing the waiting time between wrongly entered passcodes up to the point where the phone is wiped. In response to the Apple vs FBI case, Skorobogatov [120] demonstrated that NAND mirroring can be used to reset the phone state and, thus, can be used to brute-force the passcode. Clearly, this approach also represents an active attack as the attacker actively influences (resets) the state of the device.

Temperature Variation. Operating a device outside of its specified temperature range allows to cause faulty behaviour. Heating up a device above the maximum specified temperature can cause faults in memory cells. Cooling down the device has an effect on the speed RAM content fades away after power off (*remanence effect* of RAM).
Hutter and Schmidt [77] present heating fault attacks targeting an AVR microcontroller. They prove the practicability of this approach by successfully attacking an RSA implementation on named microcontroller. FROST [96], on the other hand, is a tool for recovering disc encryption keys from RAM on Android devices by means of cold-boot attacks. Here the authors take advantage of the increased time data in RAM remains valid after power off due to low temperature.

The Application of Fault Attacks Targeting Authenticated Encryption Algorithms

In this section, we want to summarize the outcome of the work "Statistical Fault Attacks on Nonce-Based Authenticated Encryption Schemes"[4]. The results presented there are closely related to the applied encryption mechanisms discussed in the previous chapter and are therefore highly relevant for the HECTOR project.
In this work, first practical fault attacks on several nonce-based authenticated encryption modes for AES are presented. This includes attacks on the ISO/IEC standards GCM, CCM, EAX, and OCB, as well as several second-round candidates of the ongoing CAESAR competition. All attacks are based on the Statistical Fault Attacks by Fuhr et al. [59], which use a biased fault model and analyse collections of faulty ciphertexts. Hereby, we put effort in reducing the assumptions made regarding the capabilities of an attacker as much as possible. In the attacks, we only assume that we are able to influence some byte (or a larger structure) of the internal AES state before the last application of MixColumns, so that the value of this byte is afterwards non-uniformly distributed.
In order to show the practical relevance of Statistical Fault Attacks and for evaluating our assumptions on the capabilities of an attacker, we perform several fault-injection experiments targeting real hardware. For instance, laser fault injections targeting an AES co-processor of a smartcard microcontroller, which is used to implement modes like GCM or CCM, show that 4 bytes (resp. all 16 bytes) of the last round key can be revealed with a small number of faulty ciphertexts.

Preliminaries. In 2013, Fuhr et al. proposed a new type of fault attack, called Statistical Fault Attack (SFA) [59]. In contrast to most previous attacks, the adversary only requires a collection of faulty ciphertexts encrypted with the same key. Hence, SFA works with random and unknown plaintexts. The main requirement for this attack to succeed is that one or several bytes of the state follow a non-uniform distribution after the fault injection. As the practical results presented later show, this requirement is met on all our evaluated targets with different fault-injection techniques.
For evaluating the non-uniformity of specific state bytes we apply the Squared Euclidian Imbalance (SEI) distinguisher: Let s be the bitsize of our biased intermediate value $S_i = f^{-1}(\hat{K}, \tilde{C}_i)$, computed from the faulty ciphertext $\tilde{C}_i$ under the key hypothesis $\hat{K}$. Assuming that we have N faulty ciphertexts, the SEI d is calculated as:

$$d(\hat{K}) = \sum_{\delta=0}^{2^s-1} \left(\frac{\#\{i \mid f^{-1}(\hat{K}, \tilde{C}_i) = \delta\}}{N} - \frac{1}{2^s} \right)^2 .$$

[4]The full work can be accessed via `https://doi.org/10.5281/zenodo.154485`

This distinguisher assigns high values to key hypotheses $\hat{K}$ that lead to distributions of intermediate values S_i with many collisions. For instance, consider a sample size of $N = 2^s$ samples. Then, the SEI is essentially counting collisions, since only events that occur exactly once do not increase d. Moreover, since the deviation from uniform is squared, a greater deviation, or in our sense a multi-collision, contributes more to d.

Results of the Practical Fault Attacks. In order to demonstrate the practical relevance of Statistical Fault Attacks, three fault-injection experiments targeting real hardware were performed.
An AES-GCM implementation executed on an off-the-shelf microcontroller served as target for the first experiment. In this context, we used the ASM AES version from [104] to realize the block cipher. Due to the lack of embedded platforms implementing GCM or CCM completely in hardware, we put the focus of the following analysis on hardware AES co-processors available on a smartcard microcontroller and on a general-purpose microcontroller, respectively. The remaining parts for realizing the authenticated encryption modes are then implemented in software.
In all settings, the fault injections aim to induce a bias on at least one byte of the AES state before the last MixColumns transformation, and allow to reveal 32 bits of the last AES round key. For full key recovery, the attack has to be repeated three more times. The following list provides an overview of the fault-injection methods and the attack results for the three settings:

1. Clock tampering has been used to disturb the execution of the AES software implementation running on an ATxmega 256A3 general-purpose microcontroller. This setting allowed to reveal 4 bytes of the last round key with less than 30 faulted ciphertexts.

2. Laser fault injections on an AES co-processor on a smartcard microcontroller. Our experiments show that less than 16 faulty ciphertexts are sufficient to reveal 4 bytes of the last round key.

3. Clock tampering on a hardware AES co-processor implemented on a general-purpose microcontroller. In this setting, we need approximately 1 200 faulted ciphertexts for recovering 4 bytes of the last round key.

For all attacks, 4 bytes of the last round key can be recovered out of the faulted ciphertexts in less than one hour using an Intel Core i7 3770K. In the following, we give a detailed description and summary of the practical fault-injection attacks.

(1) AES Software Implementation on an 8-bit Microcontroller
In the following setting, we used clock glitches to provoke faults during an AES computation implemented in software on an 8-bit microcontroller. In particular, we used the ASM AES version from [104] for realizing the GCM AE mode.
For the clock-glitch experiments, we used a nominal clock frequency of 24 MHz ($T_{clk} = 41.7ns$). According to [104], one 128-bit encryption requires 2 555 clock cycles. For simplicity, we used one general-purpose I/O pin of the microcontroller for indicating the start of the AES encryption. This trigger pin together with the knowledge of the length of the AES encryption procedure allows to find the correct time interval for inserting the clock glitch. Next to that, our results show that faults in consecutive clock cycles also lead to successful

key recovery. As a consequence, this behaviour allows to relax the precision prerequisite of the trigger information.
With the found parameters, we collected two sets, each containing 80 faulty ciphertexts. For the first set, a single clock glitch was inserted. For the second set, clock glitches in 50 consecutive clock cycles were inserted. Next, we performed SFA attacks using an increasing number of faulty ciphertexts on both sets individually. The results containing the set size N, the SEI value for the correct subkey (SEI_c), and the maximum SEI value of the wrong subkey guesses (SEI_w) were stored in two separate lists (one list for each set) in the format $[N, \mathrm{SEI}_c, \max(\mathrm{SEI}_w)]$. For this attack scenario, we started with $N = 4$ and increased N in every iteration by 4.
Figure 3.3 displays the evolution of the SEI values for increasing number of ciphertexts in the single clock glitch setting. Values corresponding to the correct subkey are plotted in red, the maximum SEI values of the wrong subkey guesses are plotted in blue. With 30 faulty ciphertexts, SEI_c exceeds $\max(\mathrm{SEI}_w)$, which allows to reveal the correct subkey value.

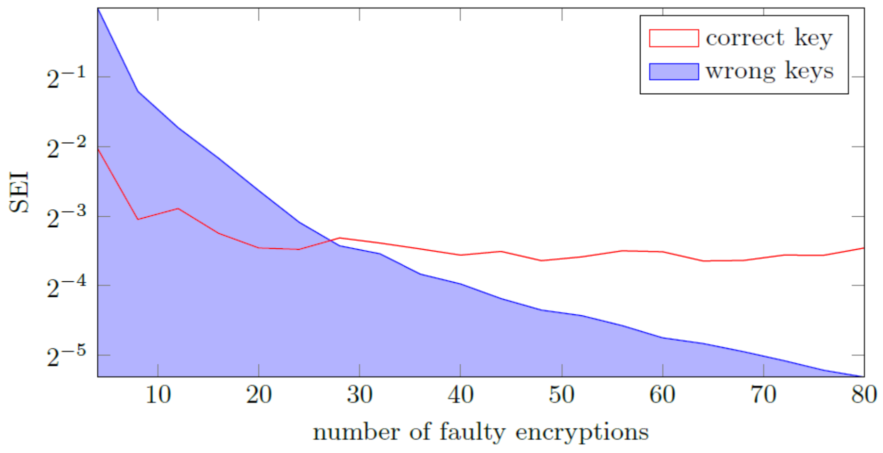

Figure 3.3: SEI values for correct key (SEI_c) plotted against best SEI for a wrong key ($\max(\mathrm{SEI}_w)$) for increasing number of faulty encryptions. Setup: AES software implementation, single clock glitch.

Figure 3.4 displays the evolution of the SEI values for an increasing number of ciphertexts for the setting with 50 consecutive clock glitches. In this setting, 24 ciphertexts are sufficient for SEI_c to exceed $\max(\mathrm{SEI}_w)$, which allows to reveal the correct subkey value.
Results of the fault attacks targeting the AES software implementations using clock glitches show that with 30 faulty ciphertexts, it is possible to reveal the 32-bit subkey if a single clock glitch is inserted. Furthermore, if the clock glitch is inserted in 50 consecutive clock cycles, approximately 25 faulty ciphertexts are sufficient for subkey recovery. We did not further investigate the approach of inserting the clock glitch in consecutive clock cycles because this is out of scope of the current work. Nevertheless, by carefully trimming the fault injection parameters, the number of faulty ciphertexts for successful subkey recovery could probably be further decreased.

(2) AES Hardware Co-Processor of a Smartcard Microcontroller
In this experiment, we used a laser fault injection system to induce faults during encryptions of an AES Hardware co-processor of a smartcard microcontroller. This co-processor can easily be used as building block for realizing authenticated encryption modes like GCM or CCM on the smartcard.

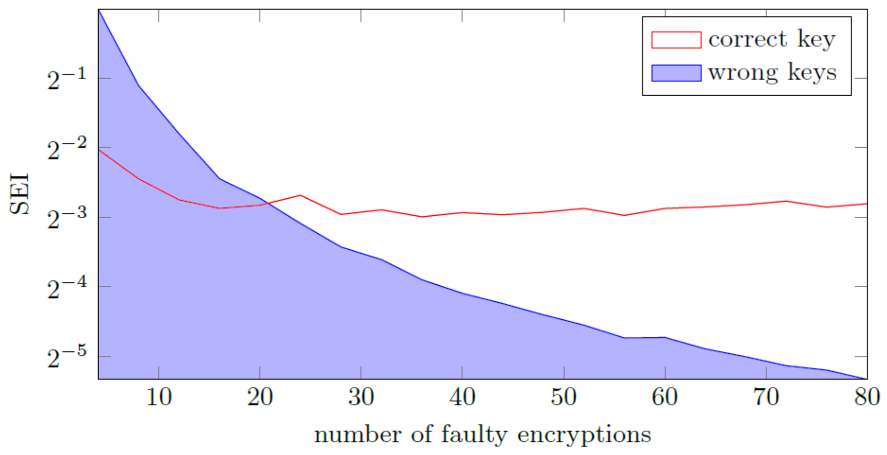

Figure 3.4: Evolution of the SEI values with increasing number of faulty encryptions. Setup: AES software implementation, multiple clock glitches.

The laser fault injection system consists of an infrared laser diode module and a microscope allowing to focus the laser spot depending on the microscope objective used. Here an objective with a $10\times$ magnification is used. The whole system is mounted on a motorized X-Y-Z stage.

As the smartcard microcontroller runs its own operating system, the only signal available for triggering the laser injection system is the sending of the encryption command through APDU command. Therefore, a temporal delay is added to postpone the laser injection during the AES encryption thanks to a remotely controllable pulse generator. Furthermore, as the smartcard microcontroller runs on its own internal clock network, an inherent temporal jitter is present due to the asynchronism between the laser injection system and the smartcard microcontroller clock network. These experimental conditions are very close to the ones present in real world scenarios.

By applying a spatial fault injection cartography, we have been able to find a spatial position where only one byte of the AES state is faulted. Furthermore, by trying different delays, we found a spatio-temporal setting where only 4 bytes of the ciphertext were faulted with a high reliability. By studying the indices of the faulted ciphertext bytes, we concluded that we successfully induced a fault on one byte of the AES state just before the last MixColumns . The fact that the hardware AES module can also be used outside of the context of authenticated encryption, i.e., for encrypting single plaintext blocks, simplified this profiling. However, if the stand-alone usage of the AES co-processor is not possible on the attacked platform, the search for the right fault injection parameters becomes more complicated, but is still feasible. With the found parameters, we collected again 80 faulty ciphertexts. With the collected faulty ciphertexts, the same evaluation as in the previous section was conducted. We started again with an initial attack set size $N = 4$ and increased the size of the attack set by 4 in every iteration. The evolution of the SEI values with increasing set size is depicted in Figure 3.5. Values corresponding to the correct subkey are plotted in red, the maximum SEI values of the wrong subkey guesses are plotted in blue.

As depicted on Figure 3.5, SEI_c already exceeds $\max(\mathrm{SEI}_w)$ with only $N = 16$ ciphertexts. Therefore, this number of ciphertexts allows to retrieve 4 bytes of the correct last round key. This result validates the practicability of the fault model and even shows that laser-based fault injection systems are well suitable for this kind of attacks.

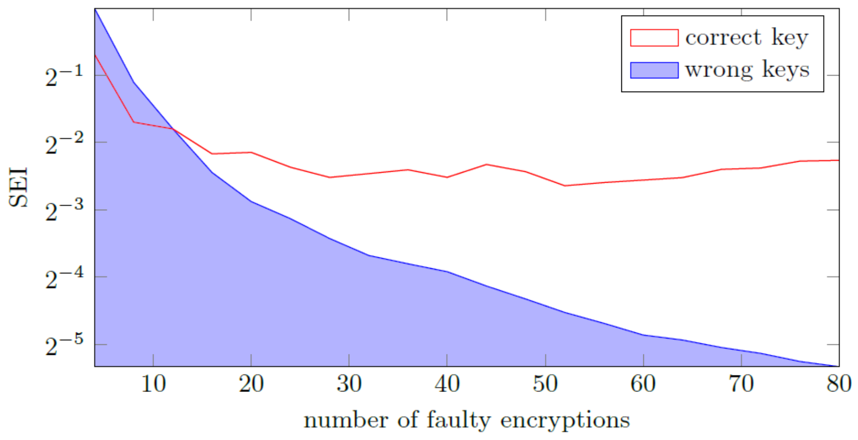

Figure 3.5: Evolution of the SEI values with increasing number of faulty encryptions. Setup: AES hardware co-processor of a smartcard microcontroller, laser.

(3) AES Co-Processor on a General-Purpose Microcontroller

In this setting, we use clock glitches to inject faults during the encryption procedure of an AES co-processor integrated on a general-purpose microcontroller. This co-processor can on the one hand be used as stand-alone block cipher to encrypt plaintext blocks, on the other hand it can be used in the context of AE for realizing a mode of operation like GCM or CCM. The co-processor in stand-alone mode allows profiling the hardware in order to find suitable fault-injection parameters. The target of the fault injection is the output of the byte substitution (SubBytes) in the 9th AES round. The AES co-processor implements the SubBytes function with pure combinational logic. Since one column of the state is processed in a single clock cycle, this allows to create faults in 4 bytes of the state with a single clock glitch.

We define with T_{glitch} the time interval between two subsequent positive clock edges in case of a clock glitch. This value is smaller compared to the nominal clock period T_{clk}, as illustrated in Figure 3.6. If T_{glitch} is smaller than the path delay of the combinational SubBytes block, the output value of this block has not settled to its correct, stable value. As a result, a wrong value is sampled by the registers at the output of the block, which leads to faults in the ciphertext.

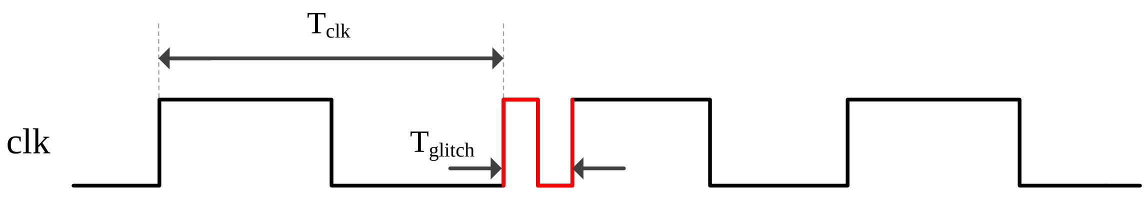

Figure 3.6: Clock signal with intentionally inserted additional positive clock edge.

For the clock glitch experiments, we used a nominal clock frequency of 10 MHz ($T_{clk} = 100ns$). Preliminary fault experiments allowed to find the correct clock cycle (i.e., the delay between the start of the encryption and the targeted instruction) to disturb the SubBytes operation in the 9th round before the MixColumns step. With $T_{\text{glitch}} = 10.2ns$, we achieved a fault probability of 99.5 %.

With these parameters, we executed the AES encryption to receive 2 000 faulty ciphertexts. The increased number of ciphertexts was required because preliminary experiments re-

vealed that the bias introduced with the clock glitch was significantly smaller compared to the bias introduced by the laser attack. With the collected faulty ciphertexts, the same evaluation as in the previous section was conducted. Due to a smaller bias, we started with an initial attack set size $N = 32$ and increased the size of the attack set by 32 in every iteration. The evolution of the SEI values with increasing set size is depicted in Figure 3.7. Values corresponding to the correct subkey are again plotted in red, the maximum SEI values of the wrong subkey guesses are plotted in blue.
As depicted on Figure 3.7, starting at 1 200 ciphertexts, SEI_c exceeds $\max(\mathrm{SEI}_w)$. This allows to reveal the correct subkey in an attack setting. Compared to the results presented in the previous section, the number of required ciphertexts is nearly 100 times higher, but the number is still practical and this amount of ciphertexts can be collected within minutes. However, the effort for performing clock-glitch attacks compared to laser fault attacks (e.g., preparing the fault-injection environment, finding good fault-injection parameters) is significantly smaller, which has to be taken into account.

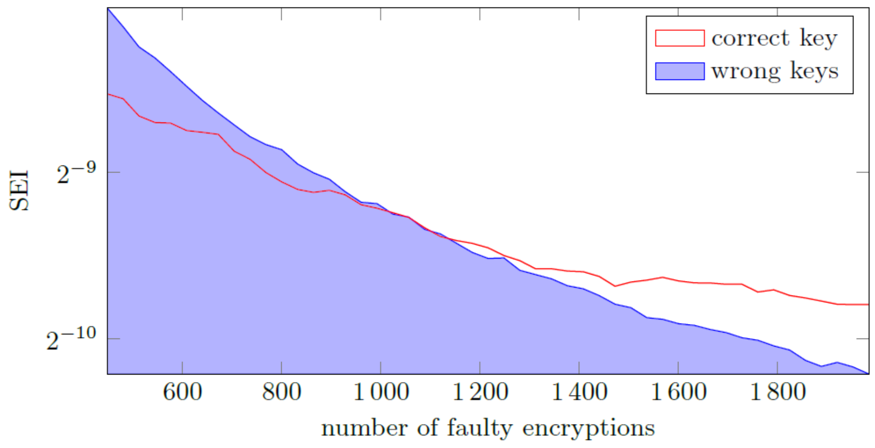

Figure 3.7: Evolution of the SEI values with increasing number of faulty encryptions. Setup: AES co-processor on a general-purpose microcontroller, clock glitch.

3.3 Countermeasures

In order to protect cryptographic implementations against side-channel attacks like the ones discussed in the previous section, a variety of countermeasures have been proposed during the last years. This section provides an overview of some of the most-popular countermeasures at the moment. We split this section in two main parts. In the first part, implementation level countermeasures are discussed. This approach directly integrates the countermeasure into the cryptographic implementation that needs to be protected. The second part deals with protocol-level countermeasures. Here the protocol is modified in order to minimize the amount of exploitable leakage available to an attacker.

3.3.1 Implementation-Level Countermeasures

One approach to achieve resistance against passive physical attacks is hiding in the time domain [88]. This approach randomizes the order of sensitive operations from one execution to the next one. In addition, so-called dummy operations, which cannot be distinguished from the sensitive operations, can be randomly inserted. In a DPA scenario, these techniques make it hard for an attacker to find the point in time where the sensitive data is processed in every measurement. Depending on the intended security level the degree of randomization can be scaled. These countermeasures introduce a significant overhead in terms of runtime (due to the insertion of dummy operations) and demand random numbers.

A second approach to achieve resistance against passive physical attacks is to make sensitive computations independent from the processed data by using so-called masking schemes. There exist many masking schemes, the scheme of Goubin et al. [69], or Ishai et al.'s private circuits [78], and the Trichina gate [128]. However, the aforementioned schemes have been shown to be vulnerable against glitches and thus rigorous care has to be taken during the implementation to avoid this issue.

Basically, sensitive values are combined with masks at the beginning of the algorithm, and the algorithm acts on these masked values. This means that any leakage observed by the attacker will be related to the masked values, rather than the original sensitive values. Of course the algorithm itself must be adapted in order to properly reconstruct the correct (i.e. unmasked) result at the end. When a single sensitive value is protected only by a single mask value, it is still possible for the attacker to overcome the countermeasure by combining two leakages: one related to the masked value, and the other related to the mask. This can be extended even when more than a single mask is involved, by exploiting several leakages together. Because the attacker must consider more than a leakage, such attacks are called high-order attacks. However, the more leakages must be evaluated, the more complex and hard to apply in practice becomes the attack. For that reason the most sophisticated countermeasures make use of several mask values at the same time.

There exist masking schemes that are inherently immune against glitches. The most popular scheme is the threshold implementation (TI) masking scheme introduced by Nikova et al. [99]. It has been extensively researched and extended by Bilgin et al. [33, 35, 37] during the last years. There exist many protected hardware implementations that are based on TI [34, 36, 95].

Recently, Reparaz et al. [112] introduced the Consolidated Masking Scheme (CMS). One interesting aspect of the CMS scheme is the possibility to reduce the number of required input shares of TI from $td + 1$ to $d + 1$, where d corresponds to the attack order and t is the algebraic degree of the function that should be protected. At CHES 2016, De Cnudde et al. [44] demonstrated the suitability of using only $d + 1$ shares on an AES hardware design. The design requires less chip area than related work, but at the cost of an increased randomness demand compared to $td + 1$ TI. More specifically, the CMS scheme requires $(d + 1)^2$ random bits for protecting one $GF(2^n)$ multiplication as required multiple times for the AES SBOX.

Availability of random values is crucial for the efficiency of masked implementations and for the effectiveness of the countermeasure. As shown in other Work Packages of the HECTOR project, producing a high amount of random numbers in hardware, however, is not trivial.

In the remainder of this section we present the results of the work "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order"[5].

[5]The full version of the work "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order" can be found online at `https://eprint.iacr.org/2016/486`

Table 3.1: First-order secure AES-128 implementation results.

Design/Module	Chip Area		Randomness	Cycles	Throughput @0.1 MHz
	[%]	*[kGE]*	*[Bits/SBOX]*		*[Kbps.]*
Our Implementation (90 nm)					
This work	100.0	**6.0**	**18**	**246**	**52**
SBOX	37.3	2.2			
State registers	34.0	2.0			
Key registers	21.0	1.3			
Control, et cetera	7.7	0.5			
td+1 Threshold Implementations (180 nm)					
Moradi et al. [95]		11.0 / 10.8[a]	48	266	48
Bilgin et al. [34]		9.1 / 8.2[a]	44	246	52
Bilgin et al. [36]		8.1 / 7.3[a]	32	246	52
d+1 Threshold Implementations (45 nm)					
De Cnudde et al. [44]		6.7 / 6.3[a]	54	276	46

[a] This variant uses the *compile_ultra* flag which is not available in our tool chain.

An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order

In this work, an efficient side-channel protected AES hardware design is introduced. We demonstrate how to achieve resistance against multivariate higher-order attacks in the presence of glitches for the same randomness cost as the private circuits scheme of Ishai et al. [78]. Although our AES design is scalable, it is smaller, faster, and less randomness demanding than other side-channel protected AES implementations. Our first-order secure AES design, for example, requires only 18 bits of randomness per SBOX operation and 6 kGE of chip area. We demonstrate the flexibility of our AES implementation by synthesizing it up to the 15th protection order.

Implementation Results. All stated numbers are post-synthesis results for a 90 nm UMC Low-K process with 1.0 V power supply and 0.1 MHz clock frequency (in accordance with related work). Our designs are compiled with the Cadence Encounter RTL compiler version v08.10-s28_1 and routed with Cadence NanoRoute v08.10-s155. Please note that in general hardware results for different technologies, compiled and synthesized with different tool chains are difficult to compare. Furthermore, the functionality implemented by different modules is not always consistent with other implementations. The comparison of chip area results with related work should therefore be seen under this premise. To make comparison with our generic AES design easier for future work, we therefore decided on publishing the source code online[6].
Besides the area of the design itself, for a masked hardware design the required number of fresh random bits is also crucial for the overall efficiency of an implementation.

First-order secure AES.
Table 3.1 compares our first-order secure AES hardware implementation with related work. The $d+1$ share designs of [44] with 6.7 kGE and our design with 6 kGE are smaller than the $td+1$ TI designs. The size difference mainly comes from the fact that $td+1$ TI requires at least three shares for securely calculating non-linear functions while the first-order $d+1$

[6]DOM Protected Hardware Implementation of AES can be found at https://github.com/hgrosz/aes-dom

Table 3.2: Second-order secure AES-128 implementation results.

Design/Module	Chip Area		Randomness	Cycles	Throughput @0.1 MHz
	[%]	*[kGE]*	*[Bits/SBOX]*		*[Kbps.]*
Our Implementation (90 nm)					
This work	100.0	**10.0**	**54**	**246**	**52**
SBOX	45.1	4.5			
State registers	30.3	3.0			
Key registers	18.7	1.9			
Control, et cetera	5.9	0.6			
td+1 Threshold Implementation (estimated [44] , 45 nm)					
De Cnudde et al. [43]		*18.6 / 14.9*[a]	126	276	46
d+1 Threshold Implementation (45 nm)					
De Cnudde et al. [44]		10.5 / 10.3[a]	162	276	46

[a] This variant uses the *compile_ultra* flag which is not available in our tool chain.

share designs require only two shares.
In comparison with $d+1$ TI design [44] which requires 54 random bits per SBOX calculation, our design requires with 18 bits only a third of its random bits. Nevertheless, our design achieves the same throughput as the $td+1$ TI design of Bilgin et al.with 52 Kbps for a 100 kHz clock and requires 14 bits less fresh randomness.

Second-order secure AES.
In Table 3.2, a comparison of our second-order AES design with other second-order secure designs is given. In case of the $td+1$ TI design the chip area was estimated by De Cnudde et al. [44]. Again, there is a noticeable gap between the $td+1$ share design with about 14.9 kGE and the $d+1$ share designs with about 10 kGE in terms of chip area resulting from the increased amount of shares (five shares versus three shares). Considering the randomness demand of the designs, our design requires 54 bits which is more than two times less than the $td+1$ design with 126 fresh random bits, and three times less than the $d+1$ TI design with 162 bits. In terms of throughput, our AES design requires 246 cycles instead of 276 cycles per encryption.

d^{th}-Order AES.
The generic construction of our AES implementation not only allows the calculation of the number of required fresh random bits of $9d(d+1)$, but furthermore it is possible to synthesize the AES implementation for arbitrary protection orders by just changing one input parameter of our hardware design.
Figure 3.8 shows the post-synthesis area results for the different components in relation to the protection order. It can be observed that the state key and control logic requirements grow linearly with the protection order. The SBOX and the contained GF gates grow quadratically. For the SBOX, the size increases from 37.4% for the first-order implementation to about 78.5% for the 15th-order. The relative size of the state and key register decrease from 34% and 21% to around 12.2% and 7.5%, respectively. The smallest amount of chip area is spent on the control logic, which stays almost constant.

Side-Channel Evaluation. We analyse the resistance of our AES designs against side-channel analysis attacks by applying leakage detection tests based on the methodology proposed by Goodwill et al. [68]. In particular, we use a *fix vs. random* test in order to assess whether the means of two sets of power measurements are different or not. To

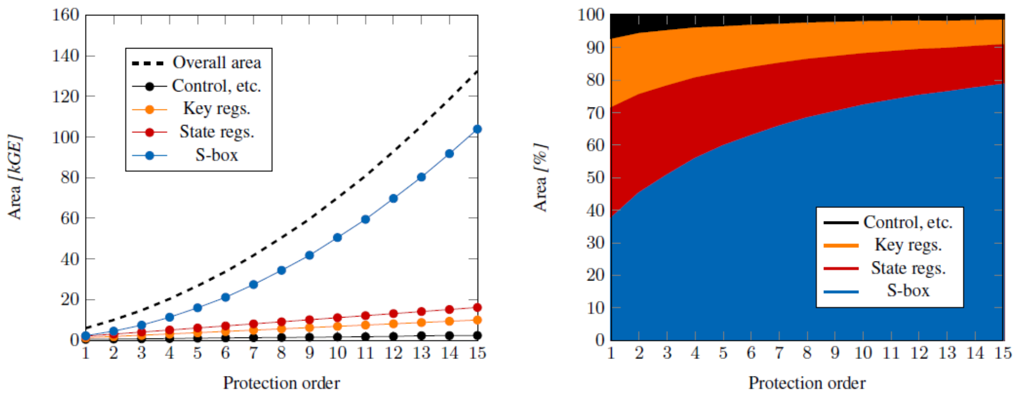

Figure 3.8: Area requirements absolute (left) and in percent (right) per protection order.

this end we collect a set *A* containing traces with a constant (unshared) input, and a set *B* containing traces with randomly picked inputs. The so-called t value is calculated by applying the Welch's t-test according to Equation 3.1, where X denotes the sample mean, S^2 the sample variance, and N the number of samples in each set.

$$t = \frac{X_A - X_B}{\sqrt{\frac{S_A^2}{N_A} + \frac{S_B^2}{N_B}}} \tag{3.1}$$

If the t value is outside the confidence interval of ± 4.5 the null-hypothesis is rejected with confidence greater than 99.999% for large sizes of N, i.e. indicating that the two sets are distinguishable and thus highlighting the existence of side-channel leakage.
Our evaluation approach is quite similar to what is checked in the d-probing model. Instead of using power trace values of, e.g., an FPGA implementation of our design, the t values of each individual signal are recorded for a post-synthesis netlist of our AES design during simulation. In comparison to an FPGA based validation this approach has three advantages: (1) the signals are completely noise free, meaning that the distributions of each signal are compared for both sets under perfect attacking conditions, (2) if any statistical differences are found, the signals can be directly pin-pointed that fail the t-test, (3) if ASIC implementations are targeted, the synthesized netlist is closer to the final ASIC implementation than an FPGA implementation.

First-order AES design.
The results of the first-order t-test for our first-order secure design are shown in Figure 3.9 (left) for up to one million traces. The t-value stays below the ± 4.5 border as required by the t-test to succeed. To demonstrate the soundness of our evaluation setup we also performed a second-order t-test. However, for the second-order t-test in a bivariate attack setting, performing individual t-tests for each signal separately is no longer feasible. The evaluation of each signal combined with every other signal for different points in time would take too long. Therefore, one single trace is calculated that sums up all signal transitions together. We then combine in each case two trace points over centered product pre-processing for all points in time within an eight clock cycles period (the delay of the SBOX). As expected the t-tests fail with great confidence with t values clearly above the ± 4.5 border even for just a hundred traces.

Second-order AES design.

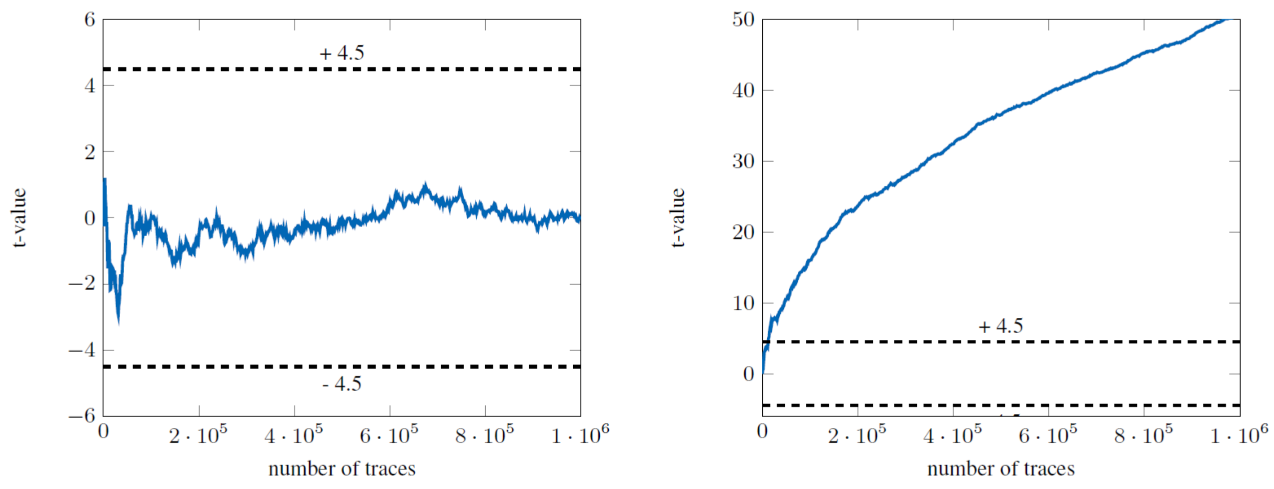

Figure 3.9: First-order t-test (left) and second-order t-test (right) for first-order secure AES design.

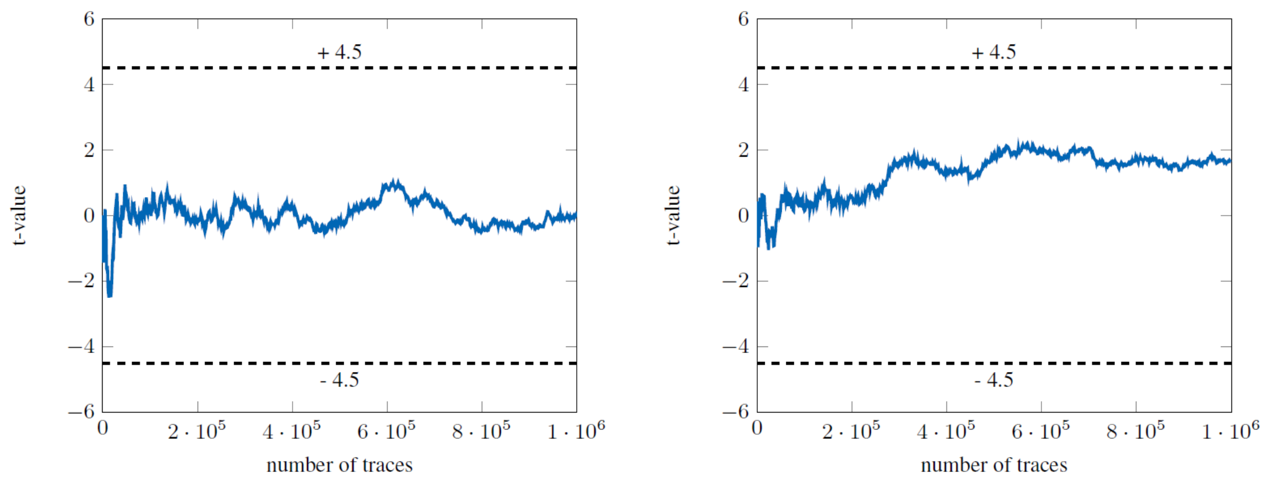

Figure 3.10: First-order t-test (left) and second-order t-test (right) for second-order secure AES design.

The t-test for the second-order AES design are illustrated in Figure 3.10. The result for a first-order t-test are on the left side and for the second-order bivariate t-test on the right. In both cases the t-tests do not indicate any leakage. We thus conclude that our implementation seems to be correct and secure in a bivariate second-order attack scenario.

3.3.2 Protocol-Level Countermeasures

Next to implementation level countermeasures, another approach to counteract side-channel attacks is to change cryptographic protocols in such a way that certain types of side-channel attacks cannot be performed at all on the underlying cryptographic primitive. Most protocol designs aim at inherently preventing DPA attacks, which is the strongest class of passive side-channel attacks. DPA attacks accumulate information about a cryptographic key by observing multiple encryptions/decryptions of different inputs. The fact that different inputs are used allows to extract keys very efficiently via statistical techniques like Bayesian dis-

tinguishers [42] or correlation [39]. In case DPA attacks are prevented by the design of the protocol, the basic approach thus is to limit the exploitable data complexity of the underlying cryptographic primitive for each key by a certain number q (q-limiting [124]). The corner case are 1-limiting constructions which are *inherently secure against DPA attacks* as they allow attackers to just observe one input per secret key. Hence, attackers are restricted to techniques like SPA which eventually leads to significantly lower overheads for the implementation of the cryptographic primitive.
Examples of the approach of inherently preventing DPA attacks are fresh re-keying [91, 90] and leakage-resilient cryptography, which brought forth encryption schemes [107, 56] and message authentication codes (MACs) [105].
While the schemes as such are quite different, the security of all the published schemes with inherent protection against DPA attacks relies on two basic properties. First, the schemes require a side-channel secure initialization with a fresh session key on every invocation. Second, the schemes require that the information an attacker can learn by collecting side-channel information about the session key is bounded [53]. These two basic properties do not just guarantee side-channel security, but also result in designs that turn out to be quite efficient for processing bulk data, since—besides the side-channel secure initialization—the cryptographic primitive does not need to be protected using implementation-level countermeasures.
In the work “ISAP – Authenticated Encryption Inherently Secure Against Passive Side-Channel Attacks” [47] a novel symmetric authenticated encryption scheme that also relies on these two basic properties is presented. The presented scheme provides significant improvements with respect to both properties. First, the authenticated encryption scheme can be applied to settings where it is highly beneficial, or even required, to allow multiple decryptions and verifications of the same ciphertext. Current schemes have not been designed to be used in such settings. Second, it is shown that the parameters of permutation-based cryptographic primitives provide a flexible tool to cope with the maximum tolerable leakage of cryptographic schemes on an algorithmic level. For the remainder of this section we summarize the two main contributions and results of this work. For the full version including all details we refer to [47].

ISAP – Authenticated Encryption Inherently Secure Against Passive Side-Channel Attacks

Contributions. Schemes with inherent protection against DPA require a side-channel secure initialization in order to obtain a fresh session key for every cryptographic operation. Such session key k_0 is typically derived from a pre-shared master key K using a nonce n by means of a re-keying function $g : (K, n) \mapsto k_0$ that is carefully designed to prevent both DPA and SPA attacks. The purpose of this secure initialization is to ensure that cryptographic operations for different data inputs are always done using different keys. Hence, whenever a party encrypts or authenticates data, a new nonce is generated to derive a new session key. While this approach successfully prevents DPA on the party performing the encryption or authentication (sender of a message), the situation is more challenging for the party performing decryptions or verifications (receiver of a message). The reason for this is that the decrypting party has no control of the nonce n. Therefore, an attacker might send arbitrary messages to the decryption device using the same nonce n for all sent messages. This behaviour results in different messages being decrypted using the same session key k_0. As a result, decryption is vulnerable to DPA, and more concretely, it is the multiple decryption with the same session key k_0 that causes this DPA vulnerability. In order to prevent this kind of DPA attacks, the receiver either needs to be protected by other means [91], or all communicating parties are required to contribute to the nonce that is used to derive the session key from a pre-shared master key [90].

In our first contribution, we overcome this problematic situation and present a symmetric authenticated encryption scheme that does not have any special requirements on the initialization and the nonces. In fact, with respect to both usage and requirements, it is a standard nonce-based symmetric authenticated encryption scheme that fulfils all functional requirements of the CAESAR call [126] and at the same time provides inherent protection against DPA attacks for all involved parties, i.e., also the decrypting party. This is achieved by making the initialization of the authenticated encryption scheme depend on the processed message itself. This implicitly prevents DPA attacks and enables several new use cases. In particular, the scheme remains secure in settings that allow multiple decryptions and verifications of the same ciphertext. Such settings cannot be realized with existing schemes, as they would require fresh nonces for every encryption and decryption.

Our second contribution is that we show how permutation-based designs can be used in order to scale implementations for different leakage bounds. Essentially, we model the side-channel leakage as part of the public output of the permutation. This allows us to adjust the maximum tolerable leakage by varying the permutation parameters. Using this flexible tool as a basis, we propose an efficient sponge-based variant of our authenticated encryption scheme and two novel permutation-based re-keying functions inherently secure against DPA attacks. We instantiate the sponge-based authenticated encryption scheme ISAP using KECCAK and present the results of its hardware implementation. This instance maintains 128-bit security in the presence of up to 16 bits leakage per permutation call, can be used in settings of multiple decryptions and verifications, and yet has approximately the same runtime and area requirements as state-of-the-art schemes. Put into numbers, the hardware implementation using an UMC 130 nm technology consumes 14 kGE, takes $22.35\mu s$ for secure initialization, and performs authenticated encryption in roughly $0.15\mu s$ per 128-bit block.

All these properties make ISAP suitable for a set of highly relevant settings in practice. One prominent example is the decryption and verification of firmware images or FPGA bitfiles, which requires that it is possible to do multiple decryptions and verifications with the same session key by different parties. There is one party that encrypts and authenticates the image or bitfile once and there are many devices that decrypt and verify it. Another example is the bulk storage of data. In such scenario, the goal is to encrypt and authenticate once and to allow multiple decryptions and verifications without the need to re-encrypt the data upon every read operation. These scenarios again highlight the main benefit of ISAP over existing schemes: ISAP remains secure in all these settings allowing for multiple decryption and verification and simultaneously has practical implementation cost.

As described in Chapter 2, the versatile sponge construction has been adopted in the design of various cryptographic primitives, e.g., in authenticated encryption (AE) designs for the ongoing CAESAR competition. Besides their flexibility, permutation-based constructions also offer a convenient way to deal with bounded SPA leakage.

In the following, ISAP – an authenticated encryption scheme inherently secure against passive side-channel attacks that solely relies on permutation-based primitives is introduced. ISAP consists of both a sponge-based encryption scheme ISAPENC and a sponge-based authentication part ISAPMAC . ISAPENC was designed as a streaming mode since previous work [107, 56] already suggests the high suitability of streaming modes to obtain encryption

schemes secure against side-channel attacks.
On the other hand, ISAPMAC combines a re-keying function and a sponge-based MAC in a novel way to obtain a MAC inherently secure against DPA with only one pass over the input data. For the secure re-keying function, either a generic permutation-based construction, ISAPRK1 , or a sponge construction, ISAPRK2 , can be used. ISAPRK1 is a design inherently secure against DPA, whereas ISAPRK2 is a more efficient design based on a stronger side-channel assumption.
Throughout this section, the side-channel discussion of the four ISAP primitives assumes SPA secure implementations of the single components and focuses on DPA only.

Authenticated Encryption Mode. The sponge-based instances of the encryption part ISAPENC and the authentication part ISAPMAC are now presented consecutively.

Encryption/Decryption.
The sponge mode to encrypt plaintexts, ISAPENC , is shown in Figure 3.11. It is an adaptation of the streaming mode in [25], which is proven cryptographically secure in [8]. In contrast to the "standard" sponge-based streaming mode in [25], ISAPENC uses a different session key k_1 for each new nonce n. This session key k_1 is provided via the secure re-keying function g_1.

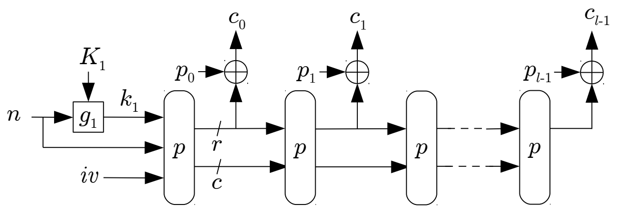

Figure 3.11: Encryption part: ISAPENC .

Whenever a cryptographic primitive is frequently re-keyed, care has to be taken to preclude generic time-memory trade-off (TMTO) attacks to recover the secret master key K_1 [46]. To avoid such attacks, ISAPENC uses both the session key k_1 and the nonce n as inputs to the first permutation call p. Hence, the design principle is similar to the fresh re-keying schemes recently presented in [48].
The initialization with a fresh nonce n and a new session key k_1 for each encryption ensures that the key stream is unpredictable and unique for different encryptions. Multiple decryption of different ciphertexts with the same nonce n and session key k_1 is inherently prevented by the authentication part. DPA on the master key K_1 is prevented by the use of the (DPA and SPA) secure re-keying function g_1 to initialize the streaming mode in ISAPENC .

Authentication/Verification.
The authentication part of the authenticated encryption mode consists of the following three steps:

1. Hash the data to get y,
2. Use y to derive the data-dependent MAC session key k_2, and
3. Compute the MAC with k_2 to authenticate the data.

Following this description, two cryptographic primitives, a hash function and a MAC, are required. However, a suffix-MAC allows to virtually combine the hash function and the MAC in one primitive. The result is ISAPMAC in Figure 3.12, a sponge-based suffix-MAC that is inherently secure against DPA.

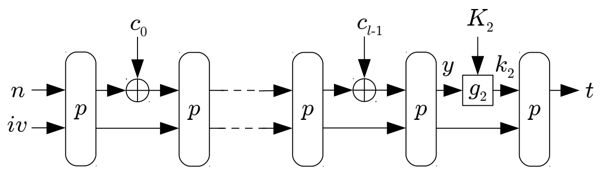

Figure 3.12: Authentication part: ISAPMAC (not showing authenticated data).

Bertoni et al. [25] showed that one can always turn a sponge into a MAC by either putting the key before (prefix-MAC), or after the message (suffix-MAC), as this always gives a pseudo-random function as long as the sponge itself behaves like a random oracle. Compared to a “standard” sponge-based suffix-MAC, ISAPMAC uses a secure re-keying function g_2 to absorb the secret key K_2. Note however, that whenever a suffix-MAC is used, care has to be taken with the choice of the parameters and the padding rule to preclude some generic attacks [108].
ISAPMAC prevents DPA on the tag computation in two ways. First, and as shown in Figure 3.12, the MAC session key k_2 is derived from the hash value y and the MAC master key K_2 via a secure re-keying function g_2, thus prohibiting DPA on K_2. Second, the design inherently prevents DPA on the MAC session key k_2 by binding it to the data being processed, thus leading to different MAC session keys k_2 for different data.
A collision in the hash value y allows for two side-channel measurements of the MAC using different data but the same MAC session key k_2. This holds true for ISAPMAC as well. Yet, to perform a successful DPA, usually more than two traces will be needed to recover one fixed session key k_2. Such a setting occurs with hash multi-collisions. The generic complexity for finding a v-collision is $\sqrt[v]{v! \cdot 2^{m(v-1)}}$. Luckily, the complexity is quite high already for small values of v as shown in Table 3.3 for a 128-bit key. Furthermore, we want to stress that even though a DPA attack exploiting multi-collisions might be able to recover the MAC session key k_2, this does not imply a key recovery attack on the master key K_2 if a non-invertible re-keying function g_2 is used.

Table 3.3: Complexity for receiving a v-collision for a 128-bit session key k_2.

v	2	3	4	5	...	34
complexity	$2^{64.5}$	$2^{86.2}$	$2^{97.1}$	$2^{103.8}$	...	2^{128}

Side-channel Secure Re-keying. Our authenticated encryption scheme requires two re-keying functions $g_1, g_2 : (K, n) \mapsto k$ that are secure against passive side-channel attacks (DPA and SPA). These two functions g_1, g_2 must not necessarily be distinct, but can be the same. We now present two possible options to design such secure re-keying function allowing to reuse the permutation p from our sponge instances ISAPENC and ISAPMAC . While the first design is inherently secure against DPA attacks, the second design is 2-limiting.

Variant 1.

In our first design, we use a variation of the classical GGM construction [66]. The respective re-keying function ISAPRK1 is shown in Figure 3.13 and works as follows. The state is first initialized with the padded master key K, followed by an application of the permutation p. In each iteration, one bit of the nonce n is processed by either choosing the left or right half of the permutation output, padding it to the permutation size, and again applying the permutation p. Hereby, the padding incorporates information on which half was chosen and on the index of the nonce bit being processed. After all nonce bits have been processed, the session key k is generated from the last permutation output.

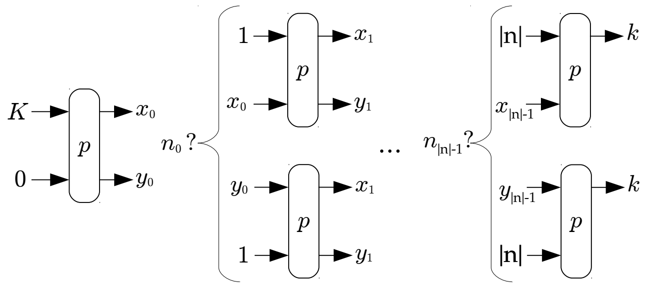

Figure 3.13: Re-keying inherently secure against DPA attacks: ISAPRK1 .

The approach to re-keying used in ISAPRK1 inherently protects against DPA attacks, since the same secret (i.e., right or left part of the permutation output) is never combined with more than one public input. In this respect, ISAPRK1 has a lower data complexity bound than present GGM-based re-keying functions [56, 124] which are 2-limiting when instantiated using common block ciphers [107].

Variant 2.
A more efficient re-keying function than ISAPRK1 can be obtained from sponges directly similar to [125], potentially reducing the required state and permutation size. However, the presented re-keying function uses a stronger security assumption than ISAPRK1 , namely, that DPA is impossible on a 2-limiting primitive, i.e., given the leakages from two different public inputs.
The basic idea is to make DPA infeasible by reducing the input data complexity accordingly. For this purpose, a secret state is constantly updated with small portions of public data by repeating two phases, (1) modifying the secret state according to the public data, and (2) updating the state such that predictions on the future state based on the absorbed public data become infeasible.
Sponges are an ideal choice to implement this basic idea as the rate directly influences the input data complexity for each permutation. Choosing the smallest possible rate ($r = 1$) results in the design ISAPRK2 shown in Figure 3.14. ISAPRK2 first initializes the sponge state by applying the initial permutation p to the padded master key K. Then, ISAPRK2 repeatedly injects single nonce bits into the state, each separated by a permutation call. After full absorption of the nonce and a final permutation call, the session key k is output. This working principle is similar to sponge instances of a prefix-MAC. While for general MAC computations the absorption rate can be as big as the state size [27], ISAPRK2 uses a small absorption rate in order to limit the data complexity exploitable in a DPA.
In terms of DPA security, ISAPRK2 uses a different assumption than the rest of this paper. For each secret state in ISAPRK2 , a permutation p will produce the leakages for two different public inputs. Thus, ISAPRK2 is not inherently secure to DPA attacks, but 2-limiting. This

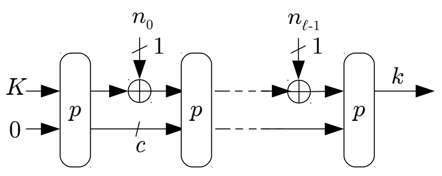

Figure 3.14: Sponge construction for re-keying: ISAPRK2 .

results in ISAPRK2 being a secure re-keying function under the assumption that the combined leakage resulting from the processing of two different public inputs is bounded such that DPA on the secret state is infeasible.
However, note that this construction for a secure re-keying function is again related to the classical GGM construction [66] and can be seen as their sponge equivalent. ISAPRK2 is similar to it in the sense that the exploitable data complexity is equal for ISAPRK2 and the block-cipher based instantiations of both [56] and the 2PRG primitive used in [124].

Instantiation and Implementation
For the practical use of ISAP we propose an instance based on the KECCAK permutation. It provides 128-bit security in the presence of up to 16 bits leakage per permutation call. Our parameter choices (permutation size, capacity, rate, number of rounds, etc.) are based on state-of-the-art cryptanalysis results.
In terms of implementation cost, an UMC-130 nm implementation of ISAP with ISAPRK2 as the re-keying function consumes merely 14 kGE, takes $22.35\mu s$ for all kind of initialization, and performs authenticated encryption in roughly $0.15\mu s$ per 128-bit block. These hardware results show that ISAP extends DPA resistance to settings allowing multiple decryption, resists up to 16 bits leakage per permutation call, and yet yields performance and area figures comparable to state-of-the-art schemes.

Table 3.4: Implementation results for secure re-keying functions (130 nm).

Function	Area	f	Cycles	Runtime
	[kGE]	[MHz]		[μs]
ISAPRK1	8.5	172	2 709	15.8
ISAPRK2	7.7	212	1 677	7.9
AES-GGM [124]	11.2	101	1 536	15.2
PolyMult [91]	10.2	–	1 160	–

Table 3.5: Implementation of the AE modes (130 nm).

Function	Area	f	Cycles	Runtime
	[kGE]	[MHz]		[μs]
ISAPAE-RK1	15.8	169	6 171	36.5
ISAPAE-RK2	14.0	171	3 853	22.5

These results suggest that ISAP is particularly suitable for the encryption and authentication of bulk data, because the scheme does not pose any DPA requirements on the implementation of the cryptographic primitive.
An advantage of this approach is its flexibility. If at some later point the leakage of an existing implementation turns out to be larger, one could simply reduce the rate r to retain the same security level.

In terms of our proposed schemes ISAPENC , ISAPMAC , and ISAPRK2 , our assumptions can be straightforwardly applied. With respect to ISAPRK1 , the modelling works analogously: the 2λ bits learned about the intermediate state account to the known part of the state that without leakage consists of the padding bits. Thus, the size of the permutation p used in ISAPRK1 has to be chosen accordingly to obtain a sufficiently large secret part to maintain the desired security level in the presence λ-bit leakage of the permutation.

Chapter 4

Side-channel-aware HW Designs

It should be clear from Chapter 3 that specifying cryptographic primitives with strong security properties from the mathematical point of view is a fundamental first step, but it is not enough for achieving robust devices in practice. Designing devices that demonstrate to be robust against side-channel attacks once in the field is a challenging task. The information exploited by these attacks are physical leakages that are not easy to predict at design-time. Design methodologies do not always provide the level of detail required for a rigorous design-time side-channel evaluation. Traditional design flows do not take into account side-channel evaluation requirements.

We believe that there is the need from the industry to establish a *side-channel-aware design methodology* in a similar way to methodologies for verification or low-power designs. Such a methodology should help making design choices and it should allow to increase the design-time confidence about the robustness of the resulting manufactured silicon devices. In this Chapter we describe our attempt in this direction.
Specifically we addressed the problem from two perspectives.

- We introduce a **top-down methodology** based on **Functional Languages**. The main goal here is closing the gap between the high-level specifications and the actual hardware implementation. We aim at a reliable way to move from specifications to actual implementations that allows to analyse countermeasures at high level and prevents the insertion of unwanted vulnerabilities in the final designs.

- We propose a **bottom-up approach** able to **model glitches on combinational logic**, which is one of the most critical sources of side-channel leakage in hardware implementations.

These two approaches are complementary to each other.

4.1 Functional Specifications of Cryptographic Circuits

4.1.1 Motivations

This part of work is motivated by a genuine concern about the functional properties exposed by a cryptographic primitive once it has been implemented in hardware, especially considering side-channel and fault attacks, which are among the most effective practical threats.

Nowadays, it is common to equip these primitives with additional circuitry to avoid side-channel leakage of information and prevent fault attacks; however, the path from the specification of such countermeasures down to the hardware implementation is far from being completely automatic. We know that manual refinement steps can be error prone and the sheer potential of these errors can be devastating in a sensitive scenario. We thus set out to identify formal tools whose purpose are to help us fill this gap.
In a broad sense, we are interested in automatic approaches that allow an automated path from the initial specification down to the Register-Transfer Level (RTL) of the cryptographic primitive under scrutiny.
This goal imposes requirements both to the expressiveness of our specification and to the tools available to convert it into a hardware implementation. In our quest, we are thus particularly concerned about the following properties of the specification language:

- Rigorousness (*internal consistency*, *non-ambiguity* and *completeness*)
- Fluency and conciseness (*construct-ability*, *manageability*, *communicability* and *evolv-ability*)
- Ability to prove or validate assertions about the system
- Ability to provide an automatic refinement framework down to the hardware level

Apart from the intrinsic scientific interest, we are looking for something that is acceptable from an industrial perspective. Several other questions thus arise:

- Is the specification system flexible enough to incorporate changes, without disrupting the development time-frame?
- Does the system require not-widely-available skills that would compromise a broader industrialization?
- Is the specification of the system cost effective?

Although the term ‘specification’ might imply a declarative artefact describing *what* the system should do, we are really looking for something that does not completely abstract away from architectural details. Our ideal specification is rooted into the solution space rather than the problem space. Moreover, it should provide a basis for further implementation and guarantee that the produced artefacts present the same properties as the source.
Perhaps, the last point asks for a clarification; what do we mean with ‘same properties’? First of all, we are interested into a bit accurate correspondence between important observation points at both abstraction levels. These may include input and output signals of the primitive, as well as signals that connect important internal blocks of the primitive. To completely define our picture we also require a cycle-by-cycle correspondence between the values observed at both levels of abstraction. Given the above definitions, it becomes increasingly clear that we are looking for an architectural specification language.
Functional programming means writing programs (and hardware specifications) that are *side-effects free* to the highest extent possible. An alternative way to define functional programming is that programs written in this way become referentially transparent, i.e., function evaluation can be substituted with the value of the computation itself (if known).
Why should we care in the design of cryptographic circuits? There are several observations that can bring us to an affirmative answer; first of all, RTL descriptions lend themselves to be

described by *pure* functions applied to a state and this is just what functional programming languages do at their best. A by-product of this is that codes are **easier to test**, reason about and compose, thus fewer bugs and more robustness.
Given their nature, functional languages lend themselves to be extremely effective in modelling mathematical concepts (by using *abstract data types*). This allows us to reason at a high level of abstractions without ambiguity, somewhat a holy grail of all hardware specification languages. Besides, they stress and enforce strong type checking, as a means to guarantee the correctness of function composition. They can go as far as providing a whole arithmetic on types (in *dependently typed languages*), guaranteeing that certain assumptions are invariant throughout the code, otherwise the program will not even be parse-able.

4.1.2 The current state of cryptographic algorithm design

In the context of current, industrial high-level specifications, we typically deal with a decoupling of the specs with respect to the implementation. In fact, RTL design is done in a separate step with respect to architectural specification design. This fact has several consequences

- Manual translation can introduce subtle bugs and unanticipated behaviour
- It makes exploration of alternatives error-prone, time consuming and tedious
- Views generated at this level (e.g., test vectors, test benches, etc.) may not be in-sync with the final implementation.
- It is usually done in weakly typed languages, additional source of unexpected behaviour
- It is difficult to test and refactor

We point out that, while the spec is written by a domain expert, RTL design might be done by a different expert, who might not be experienced with domain-critic concepts. If anything, this introduces another level of complexity in the interaction between members of the team.

Why functional programming

If a functional language were provided with an RTL back-end, part of these problems would be easily solved; no more manual translation into RTL, efficient and productive spec design, provably correct RTL correct code generation and robust spec composition.
Today there exist several options to produce RTL from functional programming languages. In the rest of this document we will explore the alternatives and choose a solution to be used in the rest of the work.

State of the art tools

In this section, we will analyse in detail which of the functional programming solutions available are suitable for specifying algorithms that are protected against side-channel attacks.

A) Cryptol

- Sponsor/Developer: Galois, Inc.
- License Type: BSD
- Years active: 2013 - 2017 (now)
- Main website: Cryptol

Introduction. The Cryptol specification language was designed by Galois for the NSA's Trusted Systems Research Group as a public standard for specifying cryptographic algorithms [55]. A Cryptol reference specification can serve as the formal documentation for a cryptographic module.
Galois provides a comprehensive documentation of Cryptol on which this section is loosely based. The original goals of the project were to (Galois docs):

> ... reduce the gap that currently exists between the specification of a cryptographic algorithm and its executable implementation. As a result, a well-written Cryptol program will look very much like the specification of the algorithm it implements, and is also executable.

At first sight, it seems that Cryptol's focus is on implementation correctness; in fact a great deal of effort has been put into automatically proving thus formally verifying Cryptol programs through the use of an SMT solver.
However, this is not the only way where a particular algorithm property can be checked. Cryptol provides a way to check a property by random testing, just as Haskell's QuickCheck does. In fact, we have reason to believe that the underlying random testing tool is effectively QuickCheck.

Basic Cryptol properties. Cryptol is a functional language closely related to Haskell (in fact, it has been developed using Haskell libraries and modules). As such, there is a great deal of focus on referential transparency and strong typing.

Bit level types. Being oriented towards specifying bit-level computations, Cryptol provides one basic data type (bits) and three type constructors (words and sequences, tuples, and records). For example:

```
12 : [8]
```

means the value 12 has type [8], i.e., it is an 8-bit word.

Strong typing. The language has fairly advanced type inference similar to other functional languages and can catch most common mistakes in programming during the type-checking phase, before run-time.
It is also *size-polymorphic* and *dependently-typed*. This means that we can describe type level computations, in addition to value level computations, and that those can be checked at compile time. For example, the bit-precise type system makes sure that we could never pass an argument that is a-bits wide in a buffer that can only fit $a/2$ bits or take the first element from a provably zero-length list.

Lazy computation, modular arithmetic. Cryptol is also a lazily evaluated language, where it is possible to create and reason about potentially infinite sequences without incurring into infinite loops. An additional interesting property of data-types is that all arithmetic is modular with respect to the underlying word size. For example, we could define an infinite list of numbers where the word length is 2, starting from one:

```
           specifies
          word length
Cryptol> [(1:[2])...]
[1, 2, 3, 0, 1 ...
```

This would effectively give a sequence of numbers modulo-2.

Stream equations. This feature of the language is akin to the execution of Synchronous Data Flow Graphs. It is useful for generating bit-streams.
The following diagram depicts a bit stream named d whose values at times $t = 0$ and $t = 1$ are `0x0F` and `0x01`.

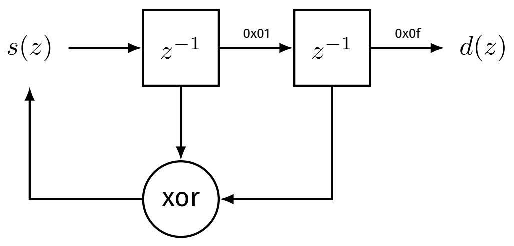

Figure 4.1: A simple synchronous dataflow graph. We use the Z-transform notation to indicate time delayed signals.

The remaining d's are given by s delayed by two cycles; s is given by:

$$s(z) = d(z) \oplus d(z)z^{-1}$$

and a possible description in Cryptol language could be the following:

```
             concatenation
d  = [0x0F, 0x01] # s where
     s = [ a ^ b | a <- d
                 | b <- drop {1} d ]
```

A Cryptol description of an AES implementation. We will give a brief introduction to the AES implementation as provided by the Cryptol documentation. As said at the beginning, we are mainly interested into how Cryptol fares with respect to the following properties:

- Rigorousness (or *internal consistency*)
- Fluency and conciseness
- Ability to prove or validate assertions about the system
- Ability to provide an automatic refinement framework down to the hardware level

We are pretty much convinced that the framework provides a reasonable rigorousness dictated by its own type system. We will see in the next few paragraphs if it meets also the other requirements.
The AES implementation is less than 250 lines long and is well founded above a mathematical description of the algorithm given directly in terms of Polynomials in $GF(2^8)$. For example we can describe a number either as its $GF(2^8)$ representation or as a bit-level representation:

```
Cryptol> <| x^^4 + x^^3 + x |>
26
Cryptol> 0b11010
26
```

Both polynomial addition and multiplication in $GF(2^8)$ are defined and can be used directly to describe the AES algorithm. Given this representation power, it is of no surprise the conciseness with which the whole encryption is described at the top level. Here, for example, we show how each round is described:

```
AESRound : (RoundKey, State) -> State
AESRound (rk, s) = AddRoundKey (rk, MixColumns (ShiftRows (SubBytes s)))
```

As can be seen, it actually exploits pure function composition to create the whole round from each of the smaller stages.

Refinement towards hardware. Although it is mentioned that Cryptol can generate VHDL (at least in its original version), it was not possible at the time of this writing to access that particular part of the tool chain.
It has also been mentioned that the language has been used for co-verification of third party VHDL IPs. Apart from the reference we did not find any other artefact to examine.

Side-channel countermeasures. There is no evidence that Cryptol has been used to prove non-functional properties such as robustness with respect to side-channel leakage of information.

Metric	*Evaluation (0-5)*	*Notes*
Rigorousness	+++++	Given by the type system
Fluency	+++++	
Validation	+++++	SMT/Quick check
Refinement towards hardware	NA	
Side-channel countermeasures	NA	
Flexibility	+++	
Widely available Skills	+	

B) Haskell - CλaSH

- Sponsor/Developer: Christiaan Baaij, University of Twente - NL
- License Type: BSD2
- Years active: 2013 - 2017 (now)
- Main website: CλaSH

Introduction. CλaSH is a functional hardware description language [12]. It is based on Haskell, a widely used functional language. According to the documentation, it provides:

> ... a familiar structural design approach to both combination and synchronous sequential circuits. The CλaSH compiler transforms these high-level descriptions to low-level synthesizable VHDL, Verilog, or SystemVerilog.

CλaSH is provided with a comprehensive guide and tutorial but requires some strong hardware development skills. However, one of the most important characteristics of CλaSH is that it is based on Haskell, and thus it comes with all the bells and whistles of a mature programming language. Besides, it is fairly reasonable to find developers who have already used Haskell, so the probability of a shortage of engineers with such skills is reduced.

Basic CλaSH properties. We can sum up the basic features of CλaSH in the following list; we will go in a deeper detail in some of them to clarify how these could be useful when developing cryptographic circuits.

- **Synchronous signals**. All signals described in a basic CλaSH circuit are implicitly synchronous with respect to a system clock. A signal is just an *applicative functor*, i.e., a particular subset of standard Haskell types, which are equipped with some additional operators. These operators specify how these types should be treated together when transformed through a pure function. So, if we have a pure addition function:

```
add :: Int -> Int -> Int
add a b = a + b
```

  and two signals, `s1` and `s2`, we can create a dataflow version of an adder by applying the function through its applicative functor interface:

```
adder :: Signal Int -> Signal Int -> Signal Int
adder s1 s2 = (pure add) <*> s1 <*> s2
```

- **Base types**. CλaSH allows to define types derived from some basic constituents, such as Bit and Vectors but it also allows to use more trivial types such as integers to describe the functionality of a circuit.

- **Type-level natural numbers and type-level functions**. One of the most useful characteristics of Haskell is the use of a strong type system equipped with type inference. These two features allow to put into the language additional constraints on the types allowed by a particular function; it is in fact possible, to some extent, to mimic dependently typed languages. Just as in the Cryptol case, this allows to detect at compile time some erroneous operation such as taking the first element from a provably zero-length list.

Describing synchronous circuits in CλaSH. Registers in CλaSH are typically inferred by delaying signals. A signal can be delayed for 1 clock cycle by construction, applying a `register` function to an already existing signal. The function has an additional parameter that should be used to describe the value of the signal at time 0.

```
s1 :: Signal Int
s1 = fromList [1, 2, 3, 4]

s2 :: Signal Int
s2 = register 8 s1
```

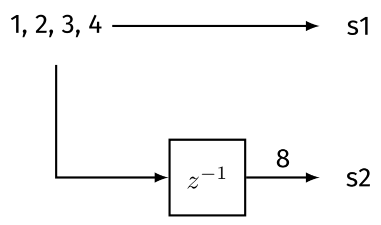

Figure 4.2: Using the CλaSH `register` function to create a simple synchronous circuit.

More complex finite state machines can be defined by using suitable constructors. For example, the `mealy` function produces a finite state machine. A simple counter can be defined by

- Defining the state as the current counter value
- Defining the next state function:

```
--     current        next state
          V              V
next      s   ()    =   (s + 1, s   )
                                ^
--                         output value
```

- Construct the state machine by invoking the `mealy` function (which requires the initial value of the state):

```
counter :: Signal () -> Signal Int
counter = mealy next 0
```

We should point out that Haskell provides abstractions to deal easily with stateful computation. One of these abstractions is the State Monad.

Simulation. The Haskell interpreter (GHCi) can be used for high-level simulation of circuits described in CλaSH. In our case, if we wanted to simulate our counter we could use the `simulate` function:

```
myCircuitSim:: [Int]
myCircuitSim = Data.list.take 4 $ simulate counter [ (), (), (), () ]
```

and invoke it in the GHCi command line:

```
>> myCircuitSim
[0,1,2,3]
```

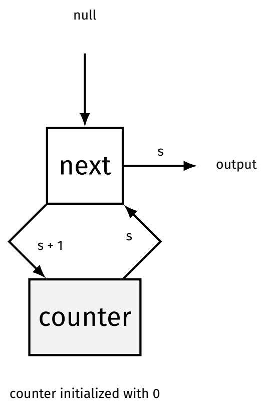

Figure 4.3: A finite state machine created by using CλaSH's mealy function.

Generating Verilog and VHDL. CλaSH comes with backends for generating both Verilog and VHDL. The high-level synthesis is triggered by a specific flag on the command line. The generated hardware blocks are isomorphic to function invocations on `Signal` types derived from a `topEntity` symbol in the Haskell program; however, the synthesis step seems to take care of avoiding resource duplication when possible.
Let us take back the hardware counter we saw in the previous section. To trigger a synthesis, one has to specify a `topEntity` symbol:

```
topEntity :: Signal ()      Signal Int
topEntity = counter
```

In this case, the topEntity is just the counter. We then invoke the verilog (or alternatively vhdl) synthesis on the command line:

```
> clash --verilog clash-test.hs
```

Verilog is then generated in a subfolder.

Test-bench generation. CλaSH can generate an HDL test-bench program as well. To do this, we specify test vectors through two additional functions defined in the Haskell program, called `testInput` and `expectedOutput`. Here, we could find the functions defined for our counter, by using two helpers (`stimuliGenerator` and `outputVerifier`):

```
testInput :: Signal ()
testInput = stimuliGenerator $(v [ (), (), (), () ] )

expectedOutput :: Signal Int -> Signal Bool
expectedOutput = outputVerifier $(v [ 0, 1, 2, 3 ] );
```

AES Implementations in CλaSH. There is no evidence in our search about an AES version described in CλaSH (with or without side-channel countermeasures). However, CλaSH looks to be very useful to be used for this kind of task.

Metric	*Evaluation (0-5)*	*Notes*
Rigorousness	+++++	Given by the typesystem
Fluency	+++++	
Validation	+++++	Quick check
Refinement towards hardware	+++	
Side-channel countermeasures	NA	
Flexibility	++++	Lots of reusable modules
Widely available Skills	+++	Haskell is very well known

C) BlueSpec

- Sponsor/Developer: Bluespec, Inc.
- License Type: Commercial
- Years active: 2003 - 2017 (now)
- Main website: Bluespec

Introduction. The Bluespec language is part of a broader set of tools sold by Bluespec Inc. The language (Bluespec's website):

> ... simplifies hardware design expression and enables automatic generation of control logic, accelerating development and eliminating many errors. The BSV language and BSC compiler can be used to generate synthesizable Verilog RTL from high-level models, verification IP, transactors and production IP.

Let's dig a bit deeper into the claims.

Inner workings. Bluespec is based on Haskell. In fact, it incorporates Haskell-style polymorphism and overloading (typeclasses) into SystemVerilog's type system. BSV also treats modules, interfaces, rules, etc. as first-class objects, permitting very powerful static elaboration (including recursion).

Static typing. Bluespec has static typing and can be used to define polymorphic data types, just like Cryptol and Haskell. The compiler takes care of type usage consistency.

Synchronous circuits. A very peculiar feature of Bluespec is describing how state evolves with time. This is done by using rules which describe when an action must be performed (a *guard*) and what are the steps to be done. The latter describes an *atomic action*. Atomic actions are the main tools with which Bluespec solves race conditions.
To make a more pragmatic decision of which actions should be scheduled in a certain clock cycle, the scheduler employs some heuristics concerning rule ordering (one rule's state update is read by another rule in the same clock cycle) and access to resources. The authors of the Bluespec language affirm that these constraints on rule firing are able to address most questions about functional correctness.

Existing AES Implementations. There exist several AES implementations in Bluespec but none of them seems to implement any kind of countermeasures against side-channel leakage of information.

Metric	*Evaluation (0-5)*	*Notes*
Rigorousness	+++++	Given by the typesystem
Fluency	+++	
Validation	+++++	
Refinement towards hardware	+++++	
Side-channel countermeasures	NA	
Flexibility	++++	Lots of reusable modules
Widely available Skills	+++	Not very well known

D) Other similar approaches A number of alternative approaches have been developed in the past. Most of them are theme variations of Haskell.

D.1) Forsyde Haskell Forsyde stands for “Formal System Design” and it is implemented as an Domain Specific Language implemented in Haskell [87][10]. ForSyDe systems are modelled as networks of processes interconnected by signals and can be synthesized to VHDL.
Forsyde makes explicit use of Template Haskell. The architecture of the circuit is in fact modelled with Template Haskell expressions, which resemble Haskell expressions but are translated at compile time into an intermediate form that is then used for the synthesis.
For example, the following declaration specifies a block that adds one to its input:

```
addOnef :: ProcFun (Int32 -> Int32)
addOnef = $(newProcFun [d|addOnef :: Int32 -> Int32
addOnef n = n + 1      |])
```

The `[d| .. |]` brackets enclosing the function declaration produce an AST (Abstract Syntax Tree). Then, the AST is used by `newProcFun` to produce a representation of the actual circuit description. It is important to note that everything happens at compile-time.
CλaSH seems much less verbose, since the same thing can be done by simply define a pure Haskell adder function.
By comparison, here is a description of a counter described in Forsyde:

```
{-# LANGUAGE TemplateHaskell #-}
module Counter where

import ForSyDe
import Data.Int (Int32)
import Plus1 (addOnef)

counterProc :: Signal Int32
counterProc = out'
where out  = mapSY "addOneProc" addOnef out'
out' = delaySY "delayOne" 1 out

counterSysDef :: SysDef (Signal Int32)
counterSysDef = newSysDef counterProc "counter" [] ["count"]
```

where the sequential circuit is implemented architecturally by using an explicit delay (`delaySY`) and a feedback loop. The code seems more verbose than CλaSH but it provides the same functionality.

D.2) (Kansas) Lava Lava [38] was an experimental system design to aid the digital design of circuits by providing a library for composing structural circuit descriptions. Kansas Lava [64] is an evolution of Lava based on Haskell. Kansas Lava has the same basic building blocks for sequential circuits as CλaSH. For example, a register has the following type signature:

```
register :: (Clock c, sig ~ CSeq c) => a -> sig a -> sig a
```

The syntax for the constraint on the type of register (preceding `=>`) states there is a clock called `c`, and there is a signal called `sig`, which is interpreted using this clock. It is just a way to guarantee that both input and output signals are synchronized using the same clock.
Defining a register shows a tendency of Kansas Lava to describe architecturally synchronous circuits:

```
counter :: (Rep a, Num a, Clock clk, CSeq clk ~ sig) =>
           sig Bool -> sig Bool -> sig a
counter restart inc = loop where
            reg = register 0 loop
            reg' = mux2 restart (0,reg)
            loop = mux2 inc (reg' + 1, reg')
```

Final considerations about functional languages

There is a great deal of recent work targeting the formal verification of countermeasures against side-channel attacks. The major concern is to provide ways to assess whether an existing specification does not leak sensitive data.
Concerning our view of the problem, we can articulate this macro-goal into at least three goals to be achieved by using functional languages:

1. **High assurance verification of the countermeasure**. We would like to guarantee that masking schemes devised in the specification are effectively implemented correctly in the RTL. We could imagine a “constructive” flow where an implementation is constructed from the spec with automatic tools, just as it is possible with CλaSH.

2. **Statistical verification of the countermeasure**. We would like to guarantee that the statistical properties of a particular primitive do not present side-channel information leakage, even when we can assure that implementation followed correctly the spec. This is a somewhat more important goal that can drive the exploration of *new countermeasures*.

3. **Specification supported by formal tools**. In this case one could specify an algorithm and have automatic tools to prove that the algorithm is safe from side-channel attacks. As seen with Cryptol, formal languages can be coupled with an SMT solver to decide, at development time, whether some formal property holds [54].

One thing to note is that the most advanced approaches seem oriented toward formal verification of software with few, if any, applications to hardware. However, as noted in [15], type

systems such as those provided in the above languages, are increasingly used to tag each variable with its level of secrecy; for example, one could use security types to represent each variable as either public or secret. Therefore, it seems plausible to pursue a similar approach also for hardware.
Some very recent approaches towards information analysis at the hardware level have been proposed (SecVerilog, [135]). While not specifically oriented towards protection against DPA with SecVerilog, hardware designers specify hardware-level information flow policies by annotating wires and registers with security labels. Labels are expressed using a lattice of security levels such that higher elements in the lattice correspond to information with more restricted flow. An example of such a lattice could be: `Unclassified` $<$ `Secret` $<$ `Top Secret`.
Other approaches targeting a formalization of hardware information flow are Caisson [86] and Sapper [85].

4.1.3 Implementing AES with CλaSH

In this section of the document, we will describe how some of the major features of the Haskell programming language can be useful to write and test a cryptographic primitive to be synthesized in hardware. Most of the presented techniques exploit both Haskell's referential transparency and extremely mature type system to develop concise yet comprehensive modules for designing hardware. In the following, we will target the construction of an AES 128 primitive by using Haskell and the CλaSH library [12].
For comprehensiveness, we restate below the goals of our research:

1. **High assurance verification of the countermeasure**. Guaranteeing that masking schemes devised in the specification are effectively implemented correctly in the RTL.
2. **Statistical verification of the countermeasure**. Guaranteeing that a particular primitive does not present side-channel information leakage.
3. **Specification supported by formal tools**. Formally proving that the algorithm is safe from side-channel attacks.

Fundamental types

Types represent the domain of the discourse of every program and play an essential role in our specification. For what concerns the cryptographic domain, the ability to create fixed size vectors of bytes is fundamental to enable the construction of more complex primitives.
To achieve this goal, we first define an `AESByte` as the datatype that will be used to represent unsigned, 8 bit (`Unsigned 8`) integers:

```
data AESByte = B { unByte::(Unsigned 8) }
               deriving (Show, Eq, Bounded, Ord)
```

The previous code defines and states several properties about `AESByte`:

- A type constructor `B`, i.e. a function that given an unsigned integer (e.g. a literal) will create a value of type `AESByte`. Type constructors are important because they can be used to define functions by pattern matching.
- An accessor function `unByte` that returns the corresponding unsigned integer when invoked on a `AESByte`.

- `AESByte` inherits a number of properties of an `Unsigned` type, namely, it is an instance of a number of basic type classes of the Haskell language.

To allow testing of blocks with arbitrary byte values (using QuickCheck) to automatically produce random values of type `AESByte`, we must create an instance of the Arbitrary typeclass for `AESByte`:

```
instance Arbitrary AESByte where
         shrink = shrinkIntegral
         arbitrary = arbitraryBoundedIntegral
```

AES state and SBOX. The AES state is a vector of 16 AESBytes:

```
type AESState = Vec 16 AESByte
```

The type `Vec` is a type that encodes (i.e., is *indexed*) by an integer value that specifies (in this case) the quantity of elements contained. It is a basic type of CλaSH that allows to put constraints on the size of the data elaborated by the primitive. In this way, it is statically possible to check for a consistent use of signals.
The AES SBOX type is built with the same principle:

```
type AESSbox = Vec 256 AESByte
```

AES input and output. Input data is fed to the AES primitive by with a type `AESInput` isomorphic to a pair:

```
data AESInput = I { unIText::AESState, unIEn:: Bool }
                          deriving (Show, Eq)
```

where the first data is the plaintext (accessible with the `unIText` function), while the second data is the value of a 1 bit signal (`unIEn`) which is high whenever a new plaintext should be encrypted.
Analogously, the output type `AESOutput` of the primitive is composed of the encrypted text and a boolean signal indicating when the output is ready.

```
data AESOutput = O { unOText::AESState, unOEn:: Bool }
                           deriving (Show, Eq)
```

Derived types. As one can imagine, we could define the type of the AES primitive as a function from an input to the output:

```
topEntity:: AESInput -> AESOutput
```

however, this signature is missing some important information about the actual implementation i.e. both input and output are not just point values but represent a stream of corresponding values one for each clock cycle of the circuit.
To express this additional information, we use the `Signal` CλaSH type level function (see also Figure 4.4):

```
topEntity:: Signal AESInput -> Signal AESOutput
```

The `Signal a` function acts by constructing a recursive type which is a stream of data of type `a`:

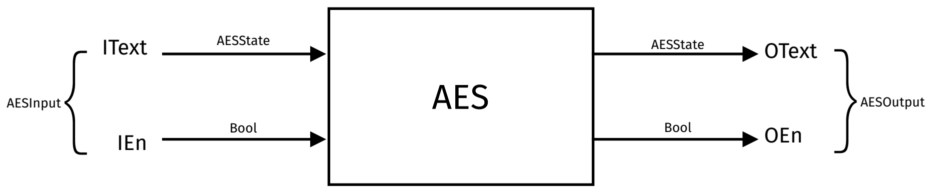

Figure 4.4: Top entity of the AES Primitive .

```
data Signal a = Cons a (Signal a)
```

i.e., a stream is a value followed by a stream of the same type.

Top level modular architecture

The `topEntity` function is the top level block that is going to be synthesized.

The top entity. We define `topEntity` as a partial application of a function, called `roundFsm` that returns another function; `roundFsm` in fact receives the internal secret key of the AES and returns a fully instantiated `topEntity`; the key can be effectively seen as a compile time constant:

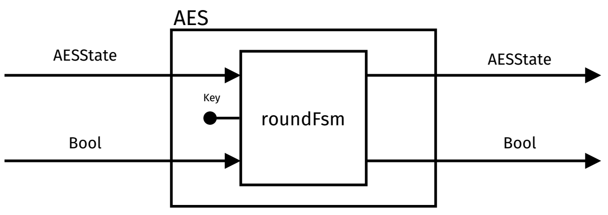

Figure 4.5: `topEntity` inner modules. The key is wired in at compile time.

Architecturally, `roundFsm` is a finite state machine that computes AES encoding in a multi-cycle fashion. We decomposed the rounds of the FSM by considering an ideal case with 16 SBOX that can be used in parallel.

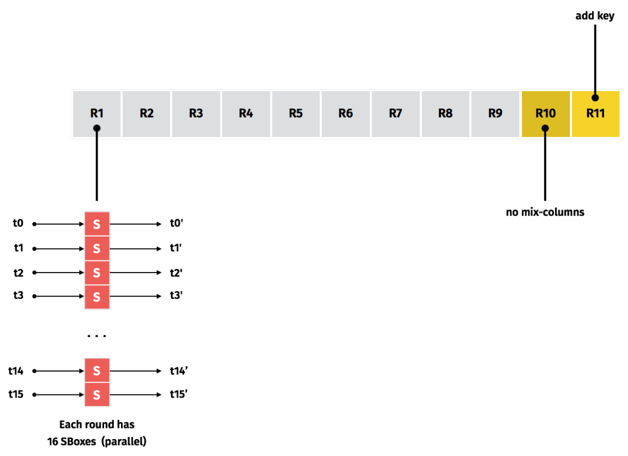

Figure 4.6: Multicycle computation of AES cipher encoding.

Multicycle round execution. Besides, we consider the round execution as being composed of 9 equivalent rounds and two non-common ones. The 10th round in fact does not apply mix-columns at the end, while the 11th round adds the final key.
The execution schedule of the multicycle AES can be seen in Figure 4.7.

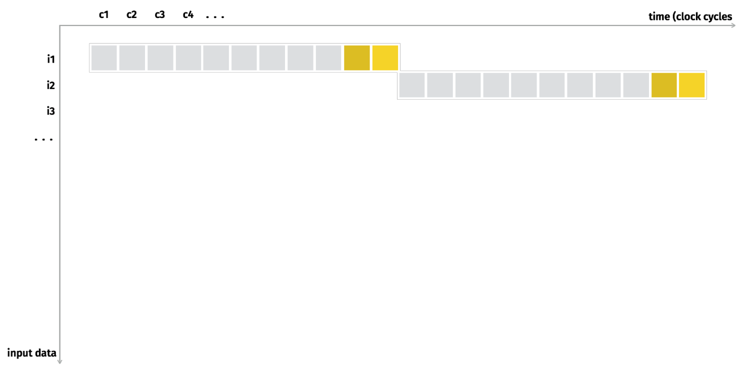

Figure 4.7: Multicycle execution for each input data/plain text.

AES finite state machine. The registers of the finite state machine represent three sub states:

- The round number: `roundnum`
- The current AES state (16 bytes): `curtext`
- The current key: `curkey`

During the first cycle (when `IEn` is true) the `AESState` is initialized to the plain text and the key is memorized in the corresponding register. The round number is initialized to 1 as well. In the remaining cycles, the next value of the three sub states (i.e., `roundnum'`, `curtext'`, `curkey`) is computed by the next state function `fan`.

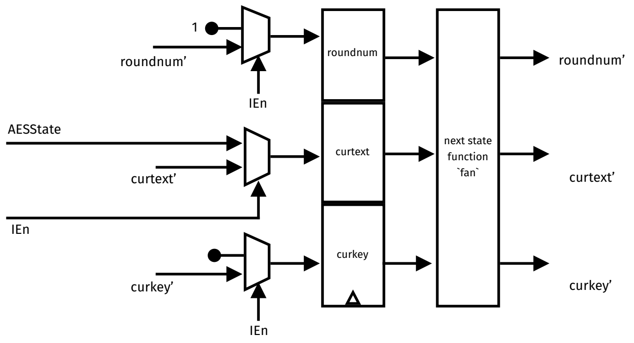

Figure 4.8: Architecture of the finite state machine.

Definition of the state. Formally, the three substates are composed into a single `record`-like state that we will call `AESControl`:

```
data AESControl = AC {
  _roundnum :: AESRoundNum,
  _curtext  :: AESState,
  _curkey   :: AESKey,
}
```

In the next sections, we will detail how the state is transformed throughout the pipeline.

The next state function. The next state function feeds the current key to both the `keyschedule` and the `round` block, where the processing of the current AES state is done. The round block is currently responsible for computing the next value of the AES State (and round number) while the `keyschedule` generates the key to be used in the next round.

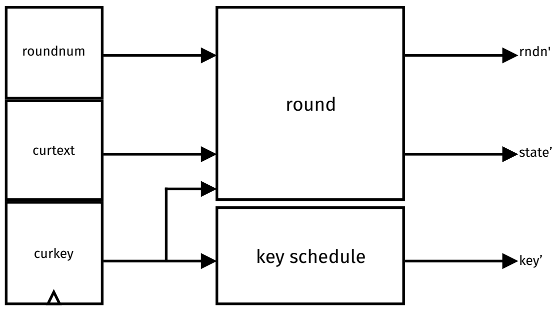

Figure 4.9: Architecture of the finite state machine.

Modules for round computation

To provide the designer with a concise Domain-Specific Language for describing the round function, we decided to use Haskell's `State` monad.
The Haskell type `State` is a type function that describes functions that consume a state and produce both a result and an updated state:

```
newtype State s a = State { runState :: s -> (a, s) }
```

where `s` is the type of the state, and `a` the type of the produced result.

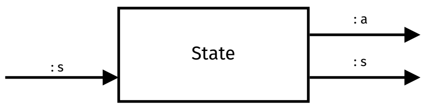

Figure 4.10: The `State` type.

In our case, we are mostly interested in the state that is carried over the round. We thus will use the following derived type throughout the `round`:

```
type AESControlProcessor =  State AESControl ()
```

where `()` is the *unit* type (i.e., a type that has only one habitant). Graphically, we can depict the generic `AESControlProcessor` type as in the following picture:

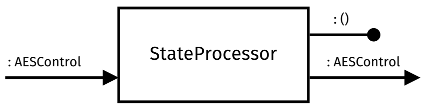

Figure 4.11: The `AESControlProcessor` type.

Monadic description of the computation. To build the round processing, we use the DSL provided by the State Monad abstraction. This is one of the major reasons we opted for a monadic implementation, instead of a simpler one. In particular we will use the State monad operator `>>` to combine state processing functions. The operator allows to sequence actions, where an action, in our case, is the effect that each block produces on the state. For example, to compose two blocks in the `AESControlProcessor` monad we use the following notation:

```
example:: AESControlProcessor
example = b0 >> b1
```

the net effect of the `>>` sequencing operator is a single `AESControlProcessor` which can be thought of as the concatenation of two simpler state processors, as shown in the following picture.

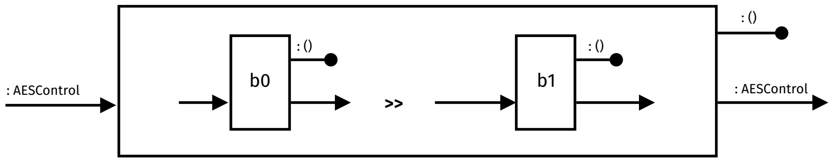

Figure 4.12: Combining AESControlProcessors.

Note that, implicitly the blocks access the current state. The major benefit of using the monad abstraction for `AESControlProcessor` is that the state is passed implicitly between one block and another, making the language more concise.

Lenses. To simplify the construction of blocks that access `AESControl`, we will use some of the combinators from the lenses library. In particular we will use the `%=` combinator which takes a function changing a particular part of the state and transforms it into a fully operational `AESControlProcessor`. In our case, the state is composed of three fields:

```
data AESControl = AC {
  _roundnum :: AESRoundNum,
  _curtext  :: AESState,
  _curkey   :: AESKey,
}
```

however, rounds are only concerned in modifying the `_curtext` which is the portion of state that contains the actual `AEState`. We thus define a `lift` function as:

```
lift :: (AESState -> AESState) -> AESControl
lift f = curtext %= f
```

The `%=` operator:

- extracts `curtext` from the state

- applies the function `f`
- “saves” the new result into `curtext` substate.

Beware that both the function `curtext` and the operator `%=` are defined by the lens library through Template Haskell. In fact, the lens library generates automatically a number of getters and setters for `AESControl`.
Finally, `lift` acts as a `wrapper` that allows to *upgrade* a function `AESState -> AESState` into a fully fledged `AESControlProcessor`, as can be seen in the following diagram:

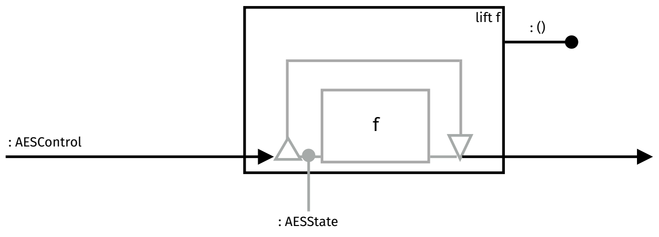

Figure 4.13: The `lift` function.

Pipelined round execution. The `round` function is indeed the most important block of the circuit. As we have seen previously, we can see it as a composition of sub-round blocks that are lifted into a corresponding `AESControlProcessor`:

```
round :: AESControlProcessor
round = let
     genround 11 k = ak k
     genround 10 k = ak k >> sb >> sr
     genround _  k = ak k >> sb >> sr >> mc in
    get >>= \s -> genround (_roundnum s) (_curkey s)
```

where `ak`, `sb`, `sr` and `mc` are lifted version of *add key*, *sub bytes*, *shift rows* and *mix columns* blocks. `round` is defined as a function that transforms the state based on the current round and the current key. In particular:

- It gets the current state (`get`) to extract the current round number (`_roundnum s`) and the current key (`_curkey s`).
- It invokes the `genround` function by passing the current round number and current key.
- `genround` actually returns a `AESControlProcessor`.

4.1.4 Comparison of AES HW designs

The goal of our research is producing an artefact that is suitable for industrialization. Therefore, a lot of effort went into an experimental process aimed at evaluating the actual feasibility of the circuits that are delivered by our new design methodology.
In the previous section, we have outlined the methodology of employing the Haskell functional language, alongside the CλaSH compiler, as a means of producing correct specifications of hardware cryptographic primitives, meeting the desired functional specifications.

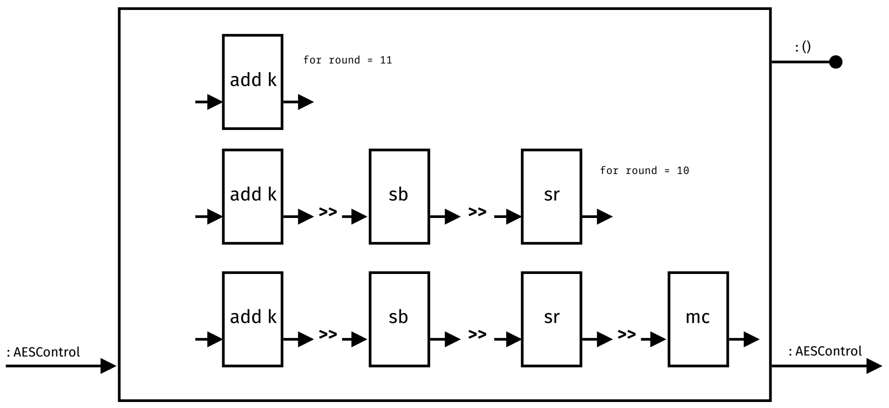

Figure 4.14: The round function.

The output of our new design methodology is a circuit description, at the RTL level, of a cryptographic hardware primitive. The CλaSH compiler is used to generate these descriptions in a standard HDL, such as Verilog or VHDL, starting from the "functional" specifications.
This section aims at demonstrating that the introduced methodology can match the results of hand-written (either in VHDL or Verilog) RTL implementations, in terms of the classic non-functional figures of merit, when looking at the outcome of the logic synthesis process. The target of our observations are the corresponding optimized *gate-level netlists* mapped to a target technology library, proprietary of STMicroelectronics, implementing 90nm CMOS standard cells. The expectation is ultimately to obtain synthesized netlists with *comparable* performance and efficiency characteristics with respect to what can be achieved by traditional design methods.
For this reasons, we tested the new design methodology against a specific *"reference implementation"*, written by hand directly in structural VHDL. For this evaluation, we designed a slightly optimized version of an (unmasked) AES-128 encryption primitive. The architectures of the two designs have been kept as close as possible to each other, in order to support this evaluation. The objective is to show the validity of the design flow from the point of view of hardware designers.

The AES-128 Encryption design

We now describe the architecture used for our evaluation. In the previous section, we used a basic cryptographic primitive, for the sake of illustrating the major features of Haskell/CλaSH for circuits' specifications. Nevertheless, now we will consider a slightly different architecture, specifically tailored to our evaluation. Everything detailed below is valid for both the VHDL as well as the Haskell/CλaSH versions. Notably, we enforced the same block interface, that is, its input and output wires, along with an identical cycle-by-cycle protocol specification.
The circuit's design is the implementation of an *unmasked AES-128 Encryption* primitive. That is, we left out the decryption part of the algorithm in order to keep the design cleaner. Since the AES decryption procedure is very similar to the encryption one, no special design principle has been neglected in the Haskell/CλaSH specification.
With respect to the design described in the previous chapter, several details are different. In particular, we introduced the input wires for the 128-bit key and we slightly improved the block encryption latency. Hence, the block's interface and the handshaking protocol have now changed.
The design's main features are:

- AES-128 Encryption-only algorithm.
- Not pipelined, but rather employing just a simple block encryption. Each block encryption is performed in **10 clock cycles**, one cycle for each AES round.
- Input interface: `IText` *(128-bit plaintext)*, `IKey` *(128-bit secret key)*, `IEn` *(1-bit "start" signal)*
- Output interface: `OText` *(128-bit ciphertext)*, `OEn` *(1-bit "done" signal)*
- **16 SBOX** instances for the AES *Round* function, implemented as a simple lookup-table (LUT).
- **1 SBOX** instance for the `KeySchedule` function, also implemented as a lookup-table.

Other relevant implementation details, with respect with the previous design, are:

- The `MixColumn` block is implemented without the help of any lookup table, instead an optimized combinatorial circuit is used for the required constant multiplications in $GF(2^8)$.
- The `RCon` block, used by the Key Schedule, is realized with a sequential logic circuit synchronized by the main FSM — instead of employing a 10-elements LUT.

From an implementation point of view, we did not identify any particular difficulty when translating the design into a Haskell/CλaSH circuit specification. On the contrary, we found that the Haskell/CλaSH approach allows to reliably describe a cryptographic primitive in a very similar fashion with respect to traditional hardware description languages, while ensuring a number of additional advantages, some of which were described in the previous section, and which derive from the higher-order level of the specification.
In addition to the reference design, we also evaluated *two* modifications of the original architecture (which have been realized with Haskell/CλaSH as well), in order to demonstrate the flexibility of the proposed methodology:

1. **Composite field SBOX**: an unmasked AES-128 Encryption primitive where all 17 SBOXes are realized by a $GF((2^4)^2)$ *composite field* implementation (also known as *tower field*, for more details see [131]). This should lead to a reduced area occupation, whereas a penalty in terms of gate propagation latency is expected. In this case, the Haskell/CλaSH specifications have been devised to allow the choice between the two different SBOX types, through a simple compile-time flag.
2. **32-bit architecture**: an unmasked AES-128 Encryption primitive performing each round function only on 32-bit words per cycle. This means that such architecture needs **5 SBOX** instances in total, one of which being reserved for the Key Schedule. Therefore, a considerable area and power consumption reduction is to be expected. In this case, we also wanted to test the refinement development cycle on the Haskell/CλaSH platform, and evaluate how easy really is to perform a significant modification to an existing design.

Experimental validation

We explored the typical design trade-offs involving the synthesis of ASIC digital circuits. The main aspects that we considered in our analysis are *area occupation* and *power consumption*. Such figures of merit have been evaluated along a reasonable range of clock frequency values for the 128-bit block encryption.
For our analysis, we set up a logic synthesis process, which involves the conversion of a design high-level RTL description (either the one produced by Haskell/CλaSH or the hand-written reference) into an optimized gate-level netlist, mapped to a target "technology library". In particular, we used a specific library of *low power 90nm CMOS* standard cells, designed by STMicroelectronics. For the logical synthesis process we used Synopsys Design Compiler alongside a number of proprietary tools in place within STMicroelectronics, in order to demonstrate the feasibility of the new methodology in an industrial setting with respect to an established reference flow and, ultimately, to ensure the manufacturability of the resulting netlist, when working under realistic constraints and design rules.

Comparison against the reference design. First, we compared the Haskell/CλaSH circuit specification against the VHDL reference design. Please recall that both designs share the very same architectural choices. We report here some of the curves obtained, related to the area occupations and the power consumptions of both designs, for the different clock frequency values.

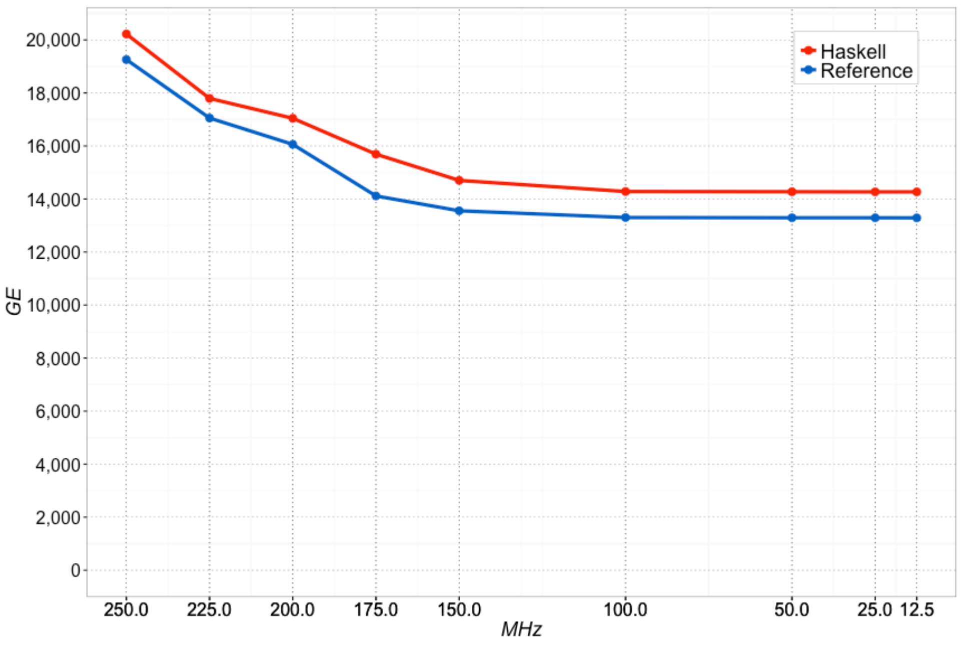

Figure 4.15: Area occupation diagram, VHDL reference vs. Haskell/CλaSH.

From the results in Figures 4.15 and 4.16, we could observe that:

- With respect the area occupation, the Haskell/CλaSH suffers from a slight but evident overhead, which stays more or less constant along the frequency values range. The measured *mean value* of such overhead is `1031.0` GE.
- The power consumption profiles of the two designs are, on the other hand, very close to each other. The measured *mean power overhead* is `0.09118` mW. At the high-

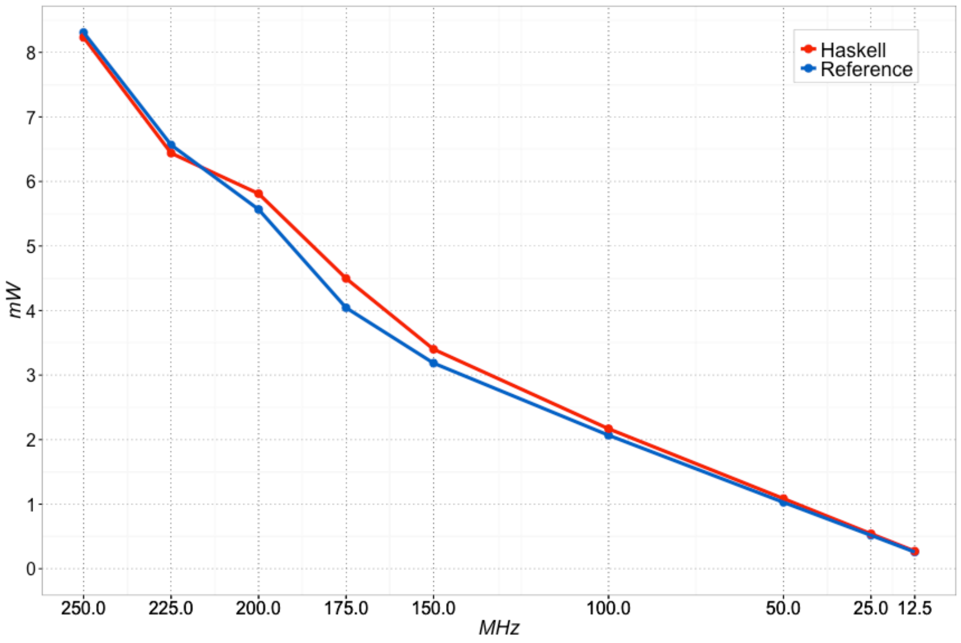

Figure 4.16: Power consumption diagram, VHDL reference vs. Haskell/CλaSH.

est frequency point, the Haskell/CλaSH version is even slightly better than the VHDL reference implementation, consuming `0.0763` mW less.

Overall, we can see that the CλaSH compiler introduced a slight overhead, especially noticeable in the area occupation measurements. Nevertheless, the results are quite encouraging as the two netlists are not too far and the technology underlying CλaSH still has plenty of room for improvement.

Inspecting the composite field SBOX implementation. After the first evaluation, we proceeded by modifying the architecture and by substituting the large 17 SBOX look-up tables with an optimized combinatorial circuit that implements the AES inversion in a $GF((2^4)^2)$ composite field, as described in [131]. We then inspected the effect of the optimization on the synthesized netlists, comparing the same figures of merit than before. Please note that, since the combinatorial circuits that implement the composite field SBOXes have a much higher propagation latency than LUTs, inevitably this architecture could not be synthesized for all the clock frequency values as the original design. This is reflected in the missing points in each of the two diagrams of Figures 4.17 and 4.18.
Here we observe a clear improvement in area occupation at 50 MHz, while there is no real power consumption reduction. However, as the synthesis constraints become tighter with the clock frequency increase, both area occupation and power consumption turn out to be far worse than the original design. As expected, it is clear that the benefits of a solution with composite-field SBOXes are appreciable only for lower clock frequency values (50MHz and below in the used technology). Nevertheless, they are very important for the implementation of protected primitives with masked SBOXes, allowing for a better robustness against side-channel attacks.

Evaluating the 32-bit AES-128 architecture. Finally, we considered the 32-bit architecture, which has been designed for area and power minimization, as it employs far less SBOX

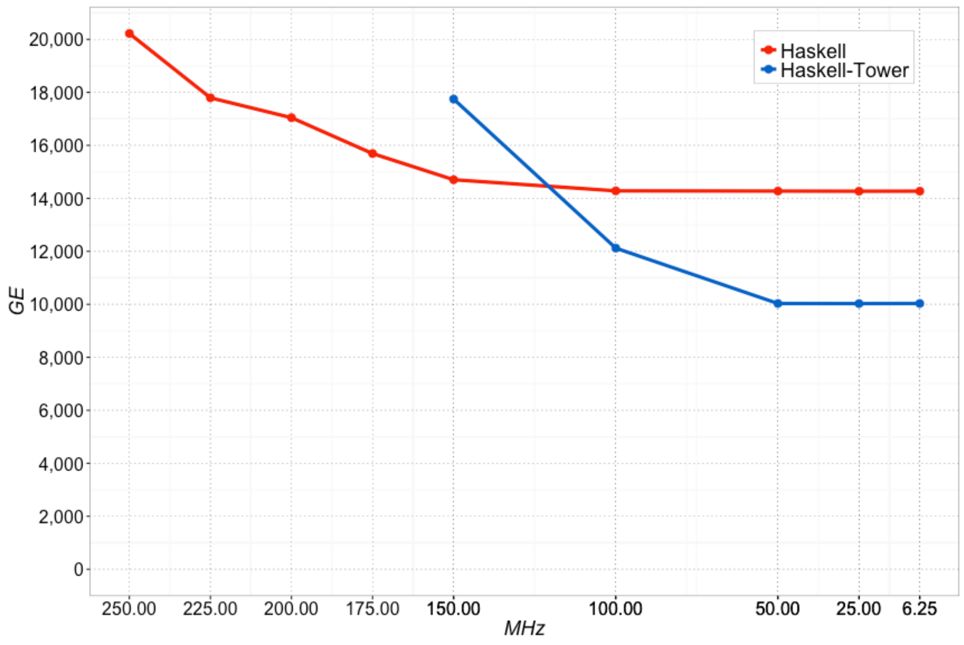

Figure 4.17: Area occupation diagram, original design vs. tower-field version (both in Haskell/CλaSH).

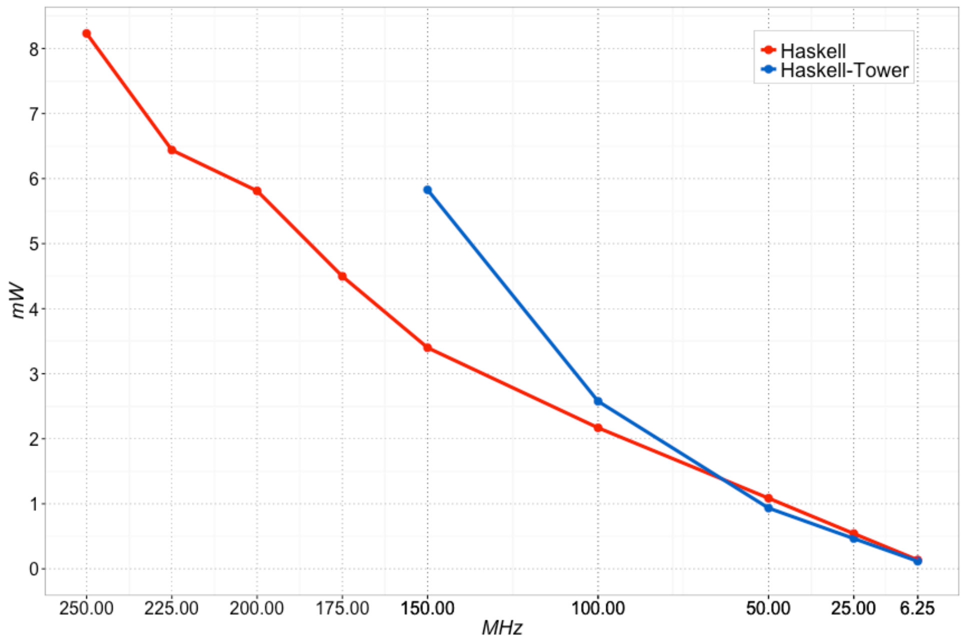

Figure 4.18: Power consumption diagram, original design vs. tower-field version (both in Haskell/CλaSH).

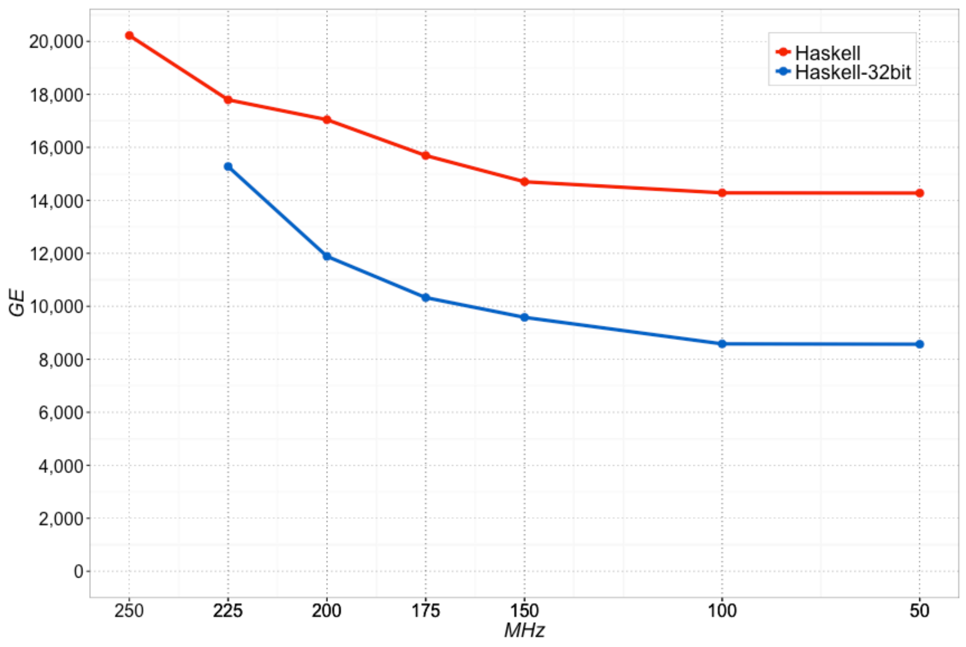

Figure 4.19: Area occupation diagram, original design vs. 32-bit version (both in Haskell/CλaSH).

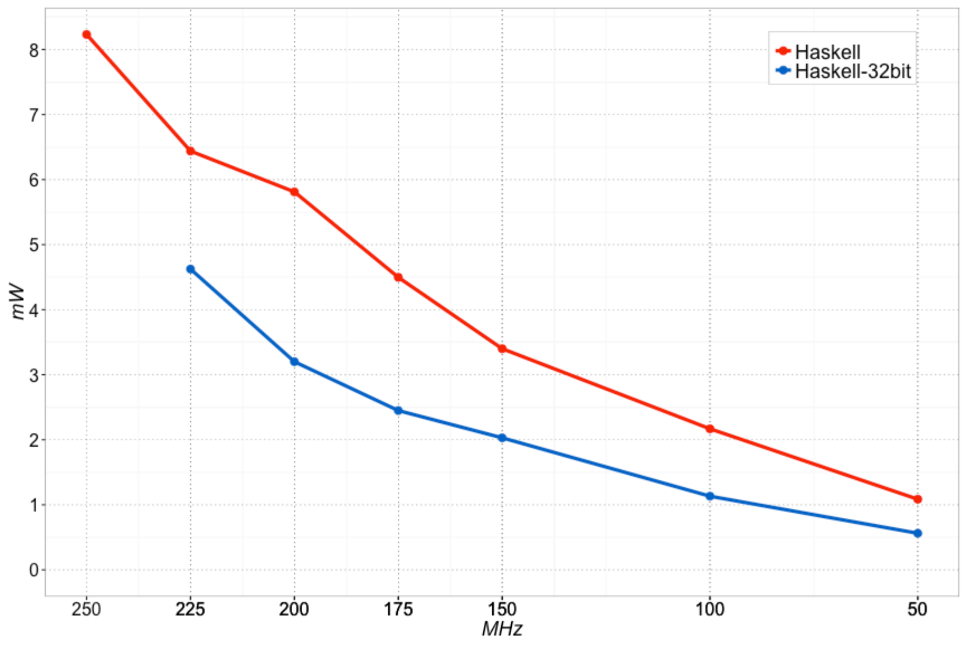

Figure 4.20: Power consumption diagram, original design vs. 32-bit version (both in Haskell/CλaSH).

instances (5 versus 17). It has been implemented with the new Haskell/CλaSH methodology as well. We compared such design against the original Haskell/CλaSH implementation. Since the 32-bit design has a slightly higher propagation latency, it cannot reach the 250 MHz clock frequency mark, hence the corresponding point is missing in the two diagrams of figures 4.19 and 4.20. Moreover, it should be noted that the following comparisons are performed along a range of fixed clock frequencies, rather than throughput values: the specific 32-bit AES-128 implementation requires 5 times the number of cycles required by the original 128-bit design to process 16 bytes.
In this case, the measurements confirm an overall gain in both area occupation and power consumption by the 32-bit architecture with respect to the main design. On average, the 32-bit architecture is `5186` GE smaller and consumes `1.0950` mW less. At 225 MHz, which is the maximum clock frequency reached by the 32-bit version, the area consumption gain is however considerably reduced. In fact, such result was predictable: the logic synthesis, when reaching a design's latency limits, usually has much more constraints.

4.1.5 Implementing 1st order countermeasures in CλaSH

In this section, we will describe the implementation of a first order masking countermeasure with the CλaSH library [12] as an extension of the AES 128 primitive presented in the previous sections. We will also provide an overview of the statistical verification of the countermeasure by validating it at the Haskell source code level and some preliminary results associated with the synthesis of the hardware primitive.

Description of the first order countermeasure

The first order countermeasure is characterized by a one byte mask (`m0`) that is replicated and summed across all 16 bytes of the state. To avoid correlation attacks exploiting the non-linearity of the SBOX, there is an additional byte mask `m1` that is used to mask the output of the SBOX.

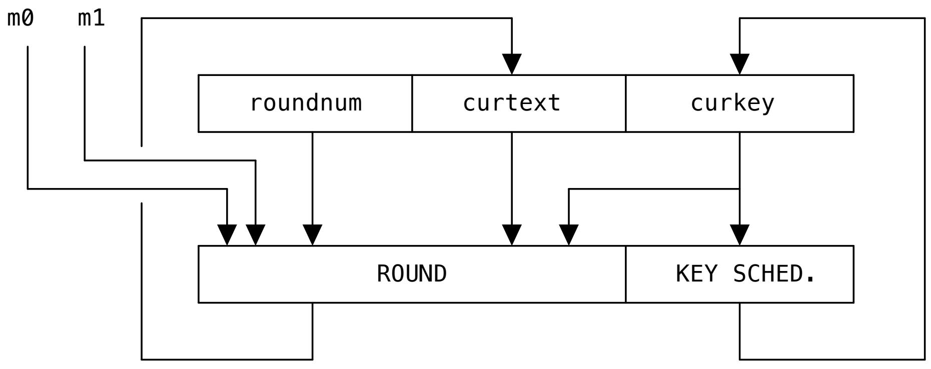

Figure 4.21: Overview of the masked computation.

Formally speaking, the input to the SBOX is the signal

$$m_0 \oplus k_i \oplus s_i, \forall i \in [1..16]$$

where k_i is the i-th byte of the key while s_i is the i-th byte of the state. The corresponding output is

$$m_1 \oplus \text{SBOX}(k_i \oplus s_i), \forall i \in [1..16]$$

where the signals k_i and s_i have the same meaning as before. We will see later that the design of the SBOX cannot expose any intermediate result while switching masks, so it must be designed carefully.
A particular characteristic of our implementation is that mask m_0 is an invariant of the state at the output of the round; i.e., it is restored into the state in the latter stages after the SBOX.

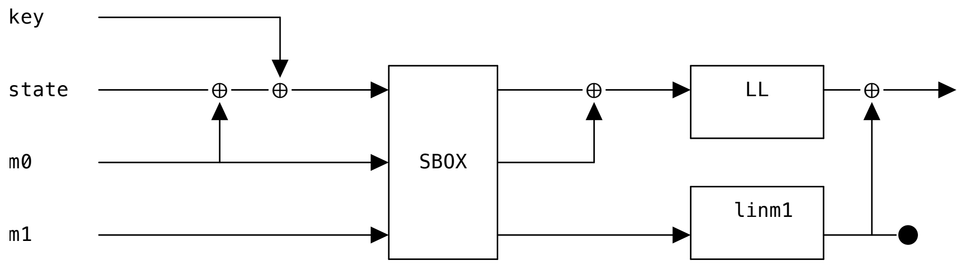

Figure 4.22: Overview of the masked round.

Design of the state machine

With respect to the solution presented in the previous sections, the state representation has been changed in order to include the current value of the masks and a number of probes:

```
data AESMasked = AMS {
#ifdef PROBE
  _prbd      :: AESProbed,
#endif
  _roundnum :: AESRoundNum,
  _curtext  :: AESState,
  _curkey   :: AESKey,
  _box      :: AESSbox,
  _ms0      :: AESMask,
  _ms1      :: AESMask
  }

type AESMaskedState =  State AESMasked ()
```

In particular, the accessor methods `_ms0` and `_ms1` allow to access the actual value of the masks whose type is a synonym for `AESByte`:

```
type AESMask            = AESByte
```

The `prbd` field of the state is used to keep track of the information that flows within the state, in fact its type is just a list of tagged texts (`AESIProbed`):

```
type AESProbed = [ AESIProbed ]

data AESIProbed = AIP {
  __probedInnerText  :: AESState,
  __probedMsg        :: String
}
```

The `prbd` field of the state is updated by a function in the AESMasked monad:

```
prm :: String -> AESMaskedState
prm m = get >>= addProbe where
    addProbe s = (prbd .= prbd') where
                         prbd' = (AIP (_curtext s) m):(_prbd s)
```

which adds the current text into the list of texts stored in the `_prbd` field. A `probe` function will output these to stdout at the end of each round and eventually they will be used for the statistical verification of the primitive.

FSM instantiation and round computation

The finite state machine now accepts a new parameter (of type `AESMaskInfo`) which is used to pass in both masks and the SBOX:

```
roundFsm' ::     AESKey ->
                 Signal AESMaskInfo ->
                 Signal AESInput -> Signal AESOutput
roundFsm' k mi t = formatOut <$> si where

   si = register controlInit (choose <$> t <*> fan si mi)
   controlInit = (_zero, (_zero, 0))

   choose (I txt True) _  = (txt, (k, 1))
   choose (I _ False)  cs = cs

   formatOut (s, (_, 12)) = O s True
   formatOut (s, (_, _))  = O _zero False

data AESMaskInfo = AMI {
  _mi_box :: AESSbox,
  _mi_ms0 :: AESMask,
  _mi_ms1 :: AESMask
  }
```

Note that the SBOX passed into the finite state machine has been already preprocessed by shuffling the original SBOX according to `m0` and XORing all the elements with `m1`. The round computation is still done through the `fan` function, which now receives also information about masks to be applied. The result of the round computation is stored into the register associated with variable `si`.
The `fan` function is simple:

```
fan :: Signal AESControl -> Signal AESMaskInfo -> Signal AESControl
fan si mi = let      (s, ks)  = unbundle si
                     (k, n)   = unbundle ks
                     s'       = round <$> n <*> s <*> k <*> mi
                     ks'      = nextKey <$> ks
                     si'      = bundle (s', ks') in
           si'
```

In fact, it unbundles the key `k` and the current state `s` from the input signal (`si`) and invokes the round by using the applicative notation, because `round` is pure with respect to `Signal`:

```
round <$> n <*> s <*> k <*> mi
```

where `n` is the round number, `s` is the current state, `k` is the key and `mi` is the mask data.

Pipelined round execution. The `round` function is indeed the most important block of the circuit. As we have seen previously, we can see it as a composition of sub-round blocks that are lifted into the `AESMaskedState` monad:

```
round' :: AESMaskedState
round' =
  let
   genround 11 k =       ak k >> m0
   genround 10 k =       ak k >> sb          >> lin' >> m0    >> linm1'
   genround 2  k =       ak k >> sb          >> lin  >> m0    >> linm1 >> _po2
   genround 1  k = m0 >> ak k >> sb   >> _psb >> lin  >> m0    >> linm1 >> _po1
   genround _  k =       ak k >> sb          >> lin  >> m0    >> linm1 in
  get >>= \s -> genround (_roundnum s) (_curkey s)
```

where `ak`, `sb`, are, as in the previous version, lifted version of *add key* and *sub bytes*. Blocks `m0`, `lin`, `lin'`, `linm1` and `linm1'` are added to functionally mask the circuit; they are defined as:

```
m0 :: AESMaskedState
m0 = get >>= \s -> xorWith (replicate d16 $ _ms0 s)

lin :: AESMaskedState
lin = sr >> mc

lin' :: AESMaskedState
lin' = sr

linm1 :: AESMaskedState
linm1 = get >>= \s -> xorWith $ (mixColumns . shiftRows . replicate d16 . _ms1)
    s

linm1' :: AESMaskedState
linm1' = get >>= \s -> xorWith $ (shiftRows . replicate d16 . _ms1) s
```

In practice `lin`, `lin'` apply usual computations (`shiftRows` and `mixColumns`) while `linm1` and `linm1'` remove effectively mask `m1` from the computation, bringing back the state masked only by `m0`.

Experimental validation - First order analysis

In this section, we are going to validate experimentally whether the first order countermeasure protects the circuit against attacks. The countermeasure is based on a byte mask that is generated randomly.

Preliminaries. First of all, to guarantee the robustness of the following tests, we are interested into checking whether masks and keys used are effectively random. We thus instrumented the Haskell code to insert two probes in the first round (p_m and p_k) to profile both masks and keys (see Figure 4.23).
Actually, the goal is twofold:

1. When observing probe p_m we want to assess the uniform distribution of the actual random generation of masks.
2. When using p_k we want to make sure that the algorithm is stressed using arbitrary keys.

The following diagrams are generated from data taken from Haskell simulations of the circuit. This information should be compared in all cases with the expected behaviour. Every

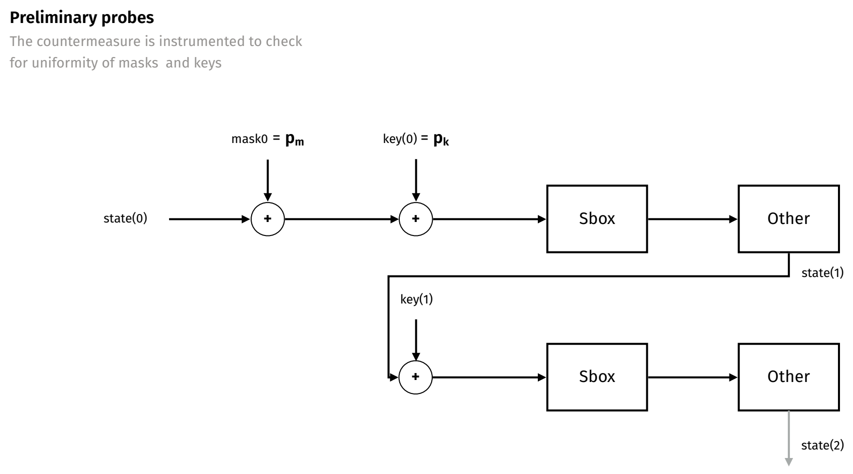

Figure 4.23: Probes in the first round.

deviation from it should be interpreted as a warning to be investigated. The following picture is an histogram of mask values used during the experiments. The histogram should cover uniformly all the possible values of a byte. As data samples go up, this should converge to the same amount for all byte values (Figure 4.24).

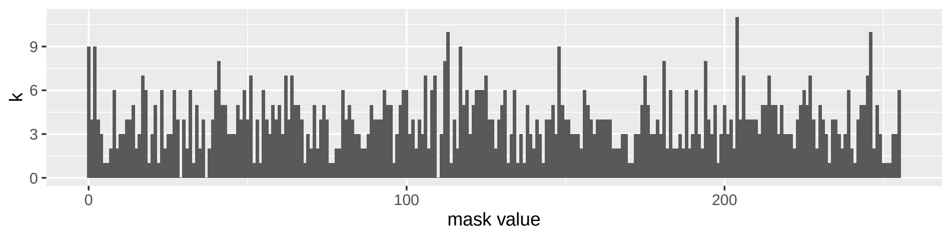

Figure 4.24: Histogram of mask values.

What follows is the histogram of byte values used for the key. Again, the histogram should cover uniformly all the possible values of a byte. As data samples go up, this should converge to the same amount for all byte values (Figure 4.25).

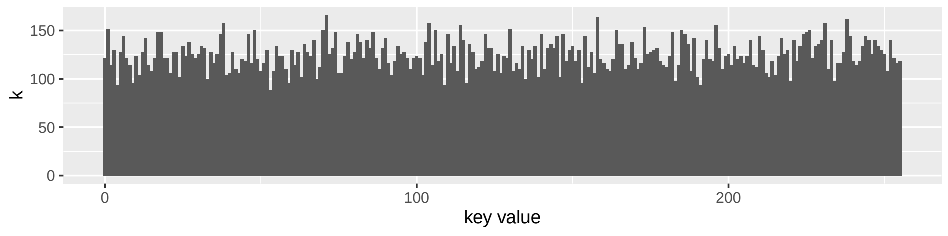

Figure 4.25: Histogram of key values.

Correlation between masked and non-masked state after the SBOX. In this first analysis we are interested in correlating the state after the SBOX with countermeasure against the same state without countermeasure. We thus require that keys and input text are the same (Figure 4.26).
We thus introduced two probes p_{sbm} and p_{sb} in the two implementations described above. Both probes produce a stream of 16 bytes each. We are interested in the cross-correlation

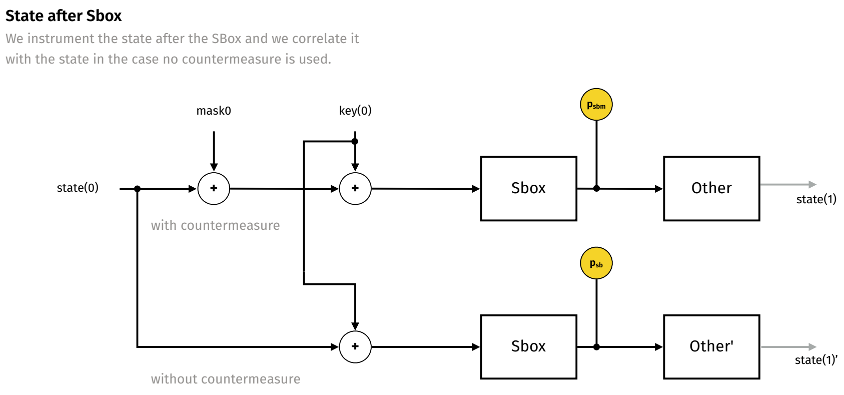

Figure 4.26: Architecture with the probes for first-order analysis.

of the 32 byte stream. If the countermeasure works, we should not see any significant cross-correlation in the upper-right and lower-left parts of the matrix itself (see Figure 4.27). As a reference, the heatmap in Figure 4.28 shows what happens when we correlate p_{sb} with itself.

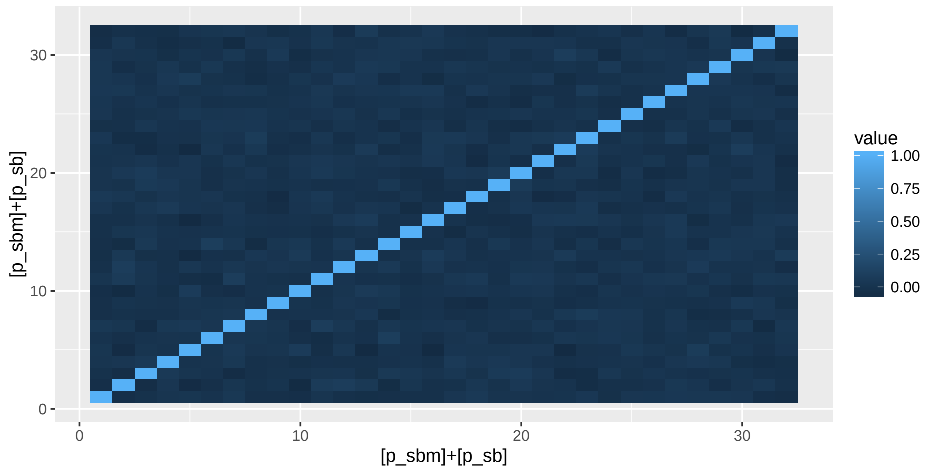

Figure 4.27: Correlation unmasked vs masked.

Per-byte statistical analysis. For each byte taken from probe p_{sb} we are now interested in the distribution of the corresponding values taken by p_{sbm}. This is actually a conditional probability distribution.
In Figure 4.29 we plot, for each of the possible byte values (0-255) in the unmasked state, the average value that has been assumed in the corresponding masked situation. Each byte value has 16 samples due to the different positions in the state of the AES.
For comparison, here is the distribution when we turn off masking:
In Figure 4.31, we look at the distribution of the raw data for each byte value. Beware that here we lose the distinction between different bytes of the state.

Experimental validation - Second order analysis

In this analysis we are interested in correlating the XOR of round 1 and 2 exit state with countermeasure ($p_{mr} = p_{mr1} \oplus p_{mr2}$), with the equivalent we would have without counter-

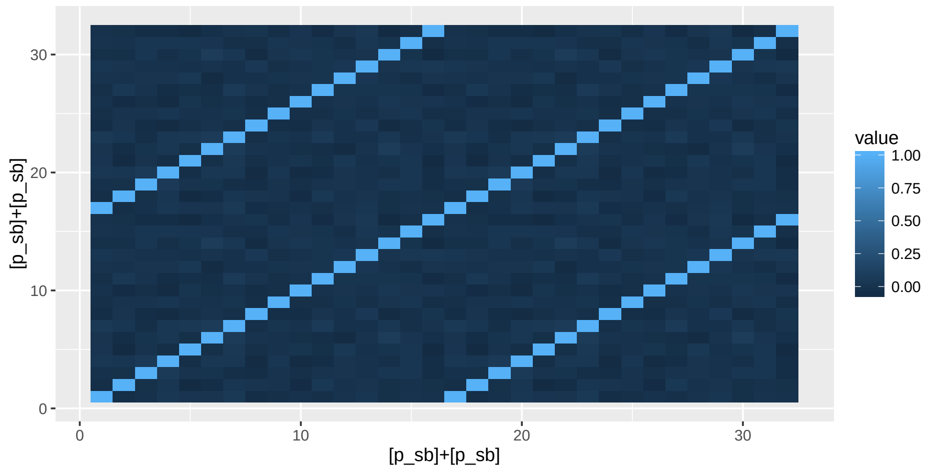

Figure 4.28: Correlation unmasked vs unmasked.

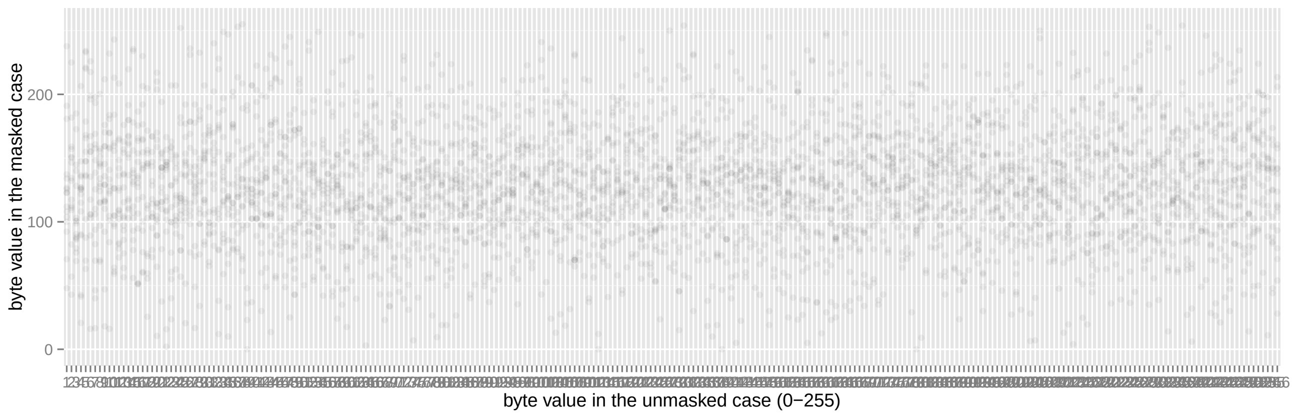

Figure 4.29: Distribution unmasked vs masked.

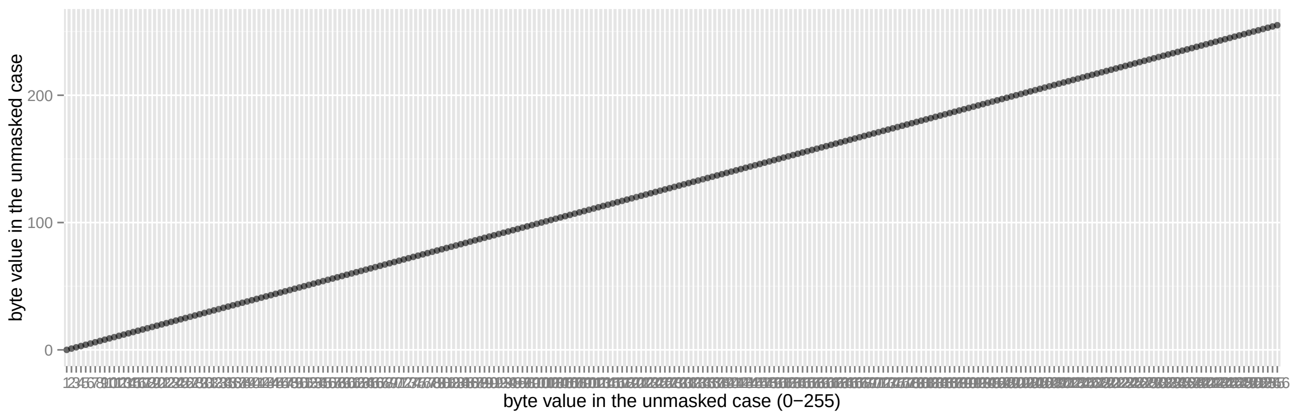

Figure 4.30: Distribution unmasked vs unmasked.

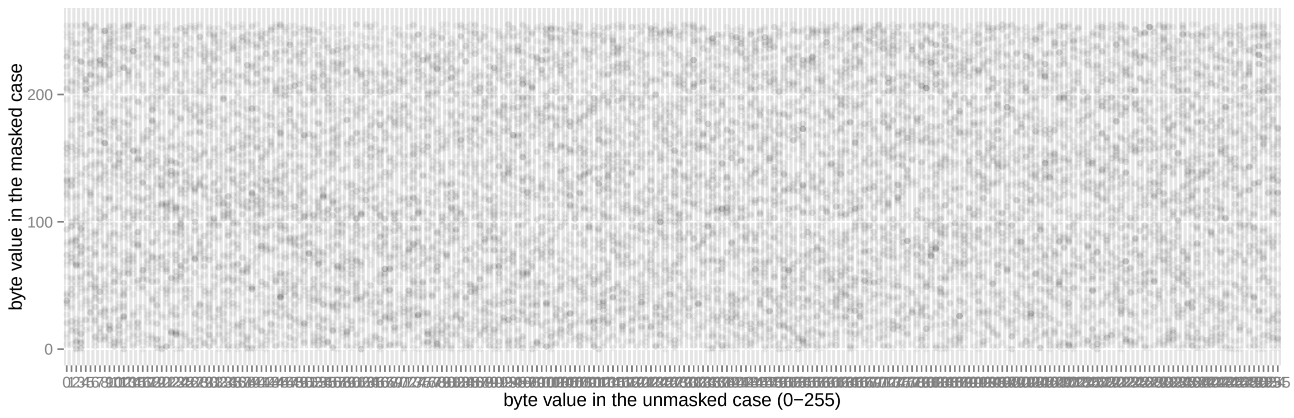

Figure 4.31: Per-byte distribution unmasked vs masked.

measure ($p_r = p_{r1} \oplus p_{r2}$). This actually corresponds to evaluating how the switching activity on the finite state machine registers is correlated.

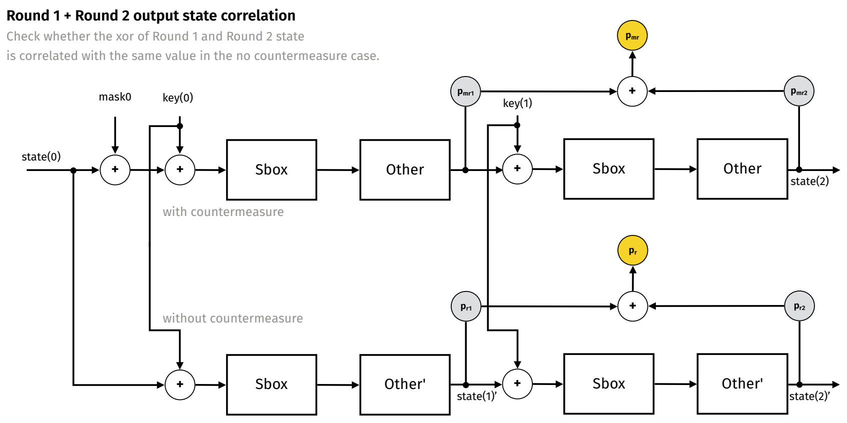
Round 1 + Round 2 output state correlation
Check whether the xor of Round 1 and Round 2 state
is correlated with the same value in the no countermeasure case.
pmr
pmr1
pmr2
mask0
key(0)
key(1)
state(0)
Sbox
Other
Sbox
Other
state(2)
with countermeasure
pr
pr1
pr2
without countermeasure
Sbox
Other'
state(1)'
Sbox
Other'
state(2)'

Figure 4.32: Architecture with the probes for second-order analysis between mask and masked value.

Since one mask protects only against a first order attack, there should be evident correlation between p_{mr} and p_r, as demonstrated in Figure 4.33.

XOR between bytes after SBOX. We now introduce two probes p_{sbm} and p_{sb} that produce a single byte, which is the result of a XOR between each pair of bytes in the state. We then correlate this value with the case with no countermeasure.
A first order countermeasure cannot hide this correlation. The diagram in Figure 4.35 shows an evident correlation.

4.1.6 Conclusions

We described the major Haskell features that can be used to describe hardware cryptographic blocks. Any design flow that does not guarantee acceptable results, both cost-wise and performance-wise, would be completely useless in an industrial setting. Is our design methodology ultimately valuable? Our experimental evaluations yield encouraging results and show that a hardware design workflow that includes Haskell/CλaSH is indeed feasible

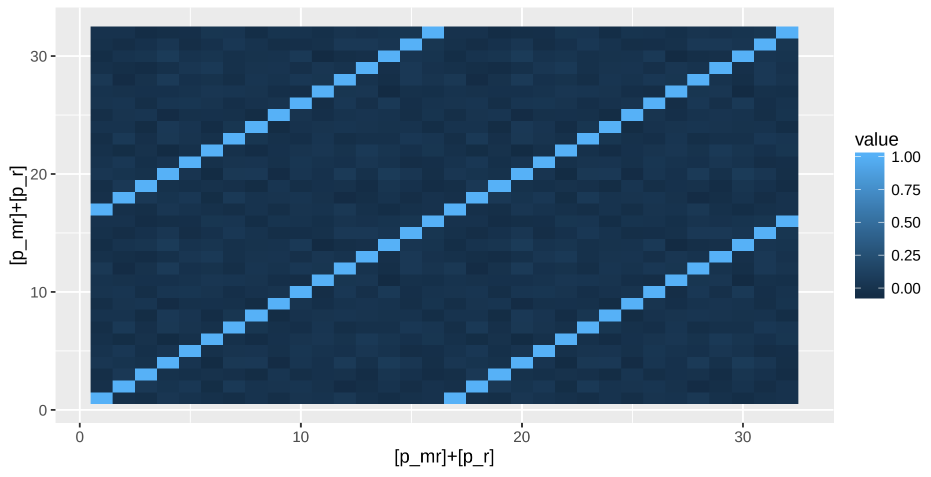

Figure 4.33: Second-order Correlation between mask and masked value.

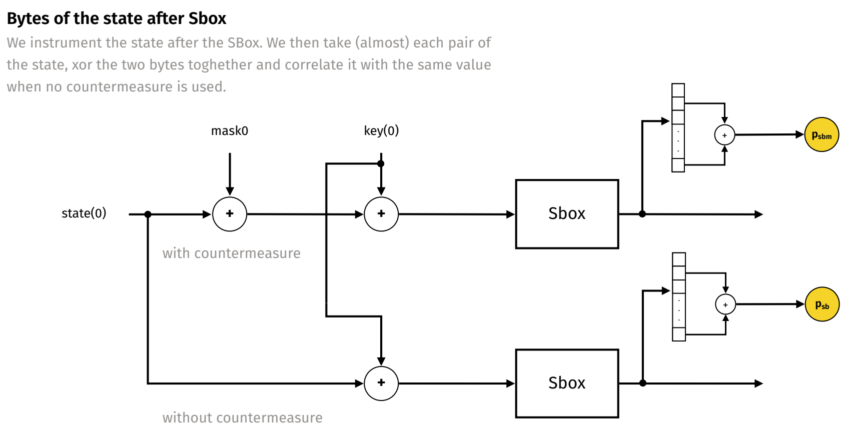
Bytes of the state after Sbox
We instrument the state after the SBox. We then take (almost) each pair of the state, xor the two bytes toghether and correlate it with the same value when no countermeasure is used.
mask0
key(0)
state(0)
+
+
+
Sbox
Sbox
with countermeasure
without countermeasure
p_{sbm}
p_{sb}

Figure 4.34: Architecture with the probes for second-order analysis between two masked values.

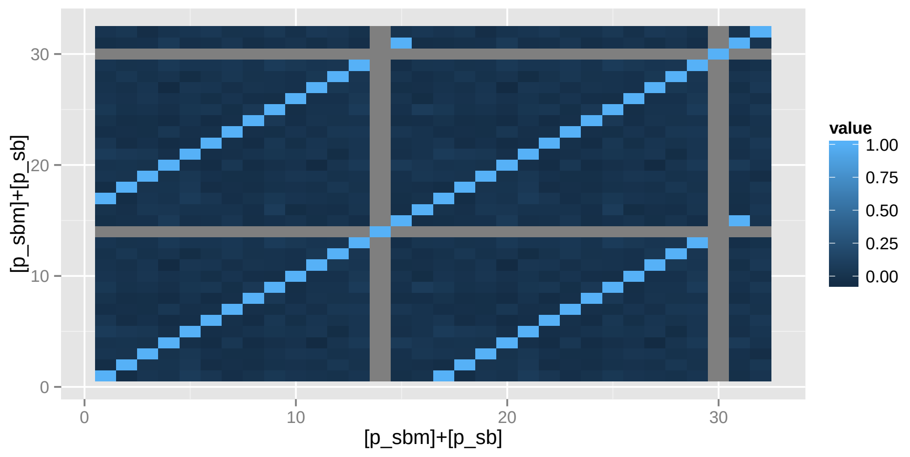

Figure 4.35: Second-order Correlation between two masked values.

and should be further explored in future efforts.

Such a framework already provides benefits in the design of security-critical hardware IPs, thanks to the possibility to strictly bind an actual implementation to its original high-level specifications. When side-channel analysis comes into the picture, we successfully showed some examples about how the framework can support statistical analysis and checking of high level side-channel properties.
One very concrete use case is the one described in the recent work [111]. It describes a method to verify the soundness of a masking scheme before implementing it on a device, based on instrumenting a high-level implementation of the masking scheme and by applying leakage detection techniques on it. In this way, a system designer can quickly assess at design-time whether the masking scheme is flawed or not, and to what extent. It is a technique extensively applied in practical contexts. It is evident how such a method can benefit when it is realized in practice by using a high-level description in Haskell/CλaSH. In this case, the framework described in the previous sections guarantees the coherence between the model and the HDL instance, which would be otherwise impossible to obtain.

4.2 Hazard Algebra

4.2.1 Motivations

The power consumption leaked by a device and exploited for a side-channel attack can be produced by **glitches**, side-channel information caused by the different propagation of signals in the circuits that are difficult to predict. Recently, a large study about glitches in circuits has been developed, since they represent a new side-channel information previously not highly regarded, but that defines a threat for secrets concealed in the devices. In [40], published at the beginning of the 21^{th} century by Brzozowski and Esik, an accurate description of propagation of glitches in a circuit is reported, through the definition of an algebra called **Hazard Algebra**. Hazard Algebra describes the propagation of glitches in the worst-case scenario, namely when there is the higher number of glitches in a specific circuit.
A further study about glitches has been implemented starting from the Hazard Algebra and developed to create a methodology for the characterization of leakages in combinatorial logic, called **LP model** [31]. It works in the worst-case of propagation of glitches, i.e. in the same situation examined by Hazard Algebra. LP model is a method that allows to recover some fatal leakages caused by glitches, specifying the combinatorial part of the circuit that produces these leakages. This model can be interpreted as a tool to assess the security of cryptographic circuits against side-channel attacks.
Then, both Hazard Algebra and LP Model describe the propagation of glitches in the worst-case, but we realized that this situation described not always is a real situation; we think that is necessary a study about some structures that can represent a more realistic situation for circuits and better quantify the amount of critical leakages.
Another relevant point of our study is the **order of an attack** (and in particular of a side-channel attack). There are some articles in literature that give some different definitions of the order of an attack, and in particular we focus our attention on definitions in two articles [78] [94]. In [78], the order of an attack is the number of probes (metal needles that read information through wires) placed in the circuit; instead, in [94] the order of the attack is the statistical moment used to implement it. Also in article [100], in which some features

about security of hardware implementations against side-channel attack of the first order are exposed, it is not clear what they mean with “first order”. For some attacks, these two definitions of the order of an attack are not in contrast, and then the order of them can be simply given; but very often these definitions are discordant and create some misunderstandings, and for this we think that it is necessary a new redefinition of this concept.
The main cryptographic algorithm that we use as example is the KECCAK algorithm [26] [25]. In particular, we often analyse only the nonlinear part of this algorithm, since generally this is the target of side-channel attacks [24] [21].

4.2.2 Hazard Algebra

Definition 4.1. A *glitch or hazard* is a fast and unwanted change at the output of a gate in a circuit due to different time propagation of the inputs.

Also in the case of a synchronous circuit, an unwanted change in a signal increases the energy consumption, which can be exploited for a side-channel attack, where glitches are considered the side-channel information.
The study of glitches in circuits has been a widespread problem in early years of design of circuits. A very interesting description of glitches through bitstrings and an Algebra based on them has been described in [40] by J. Brzozowski and Z. Ésik. In that work, a tool to evaluate the presence of glitches under worst-case conditions of propagation was introduced.

Definition 4.2. In a circuit, the *Worst-Case Scenario for glitches propagation* describes what happens when the highest number of glitches occurs at each gate in the circuit. This situation implies the maximum power consumption in the circuit, and this is the reason why it is called *worst-case*.

In our work, we are not interested in the specific value of signals in all time instances, but only in their changes. For this reason, we introduce the definition of transient.

Definition 4.3. A *transient* is a bitstring t without any repetition of zeros and ones:

- $t = e$ is the *empty transient*, corresponding to the empty bitstring;
- $t = b_1 b_2 ... b_k$ with $b_i \in \mathbb{Z}_2$ and $b_j \neq b_{j+1}, \forall j \in \{1, ..., k-1\}$.

The set of all non-empty transients is defined as the set

$$T = 0(10)^i \bigcup 1(01)^j \bigcup 0(10)^h 1 \bigcup 1(01)^r 0$$

with $i, j, h, r \in \mathbb{N}_0$.
For any $t \in T$, with $t = a_1 a_2 ... a_n$, some features are defined: $z(t)$ is the number of zeros of t; $u(t)$ is the number of ones of t; $\alpha(t)$ is the first element of t, i.e. a_1; $\omega(t)$ is the last element of t, i.e. a_n; $l(t)$ is the length of t, i.e. n. Given these specifications, it is possible to notice that every transient t is uniquely determined by $\alpha(t)$ and $l(t)$, or by $\omega(t)$ and $l(t)$, or by $\alpha(t)$, $\omega(t)$ and $z(t)$, or by $\alpha(t)$, $\omega(t)$ and $u(t)$.
We can define three operations on transients: $\boxplus$ is the sum in T, $\boxtimes$ the product and $\overline{\cdot}$ the complement.

- For any transient $t \in T$, $t \boxplus 0 = 0 \boxplus t = t$ and $t \boxplus 1 = 1 \boxplus t = 1$. If w and w' are two elements of T such that both their lengths are >1, then their sum is $w \boxplus w' = t$, where $\alpha(t) = \alpha(w) \vee \alpha(w')$, $\omega(t) = \omega(w) \vee \omega(w')$ and $z(t) = z(w) + z(w') - 1$.

- For any transient $t \in T$, $t \boxtimes 0 = 0 \boxtimes t = 0$ and $t \boxtimes 1 = 1 \boxtimes t = t$. If w and w' are two elements of T such that both their lengths are >1, then their product is $w \boxtimes w' = t$, where $\alpha(t) = \alpha(w) \wedge \alpha(w')$, $\omega(t) = \omega(w) \wedge \omega(w')$ and $u(t) = u(w) + u(w') - 1$.
- The complement $\bar{t}$ of a transient $t \in T$ is obtained by complementing each element in t.

Give these three operations, a new algebra is defined: the *Change-counting Algebra* or *Hazard Algebra* $C = (T, \boxplus, \boxtimes, \bar{\cdot}, 0, 1)$, where 0 and 1 are the only transients with length 1 of C.

Theorem 4.2.1. *The Hazard Algebra $C = (T, \boxplus, \boxtimes, \bar{\cdot}, 0, 1)$ is a commutative de Morgan bimonoid, i.e, given $x, y, z \in T$, it satisfies the equations in Table 4.4.*

Table 4.4: Laws of Hazard Algebra.

L1	$x \boxplus y = y \boxplus x$	L1'	$x \boxtimes y = y \boxtimes x$
L2	$x \boxplus (y \boxplus z) = (x \boxplus y) \boxplus z$	L2'	$x \boxtimes (y \boxtimes z) = (x \boxtimes y) \boxtimes z$
L3	$x \boxplus 1 = 1$	L3'	$x \boxtimes 0 = 0$
L4	$x \boxplus 0 = x$	L4'	$x \boxtimes 1 = x$
L5	$\bar{\bar{x}} = x$		
L6	$\overline{x \boxplus y} = \bar{x} \boxtimes \bar{y}$	L6'	$\overline{x \boxtimes y} = \bar{x} \boxplus \bar{y}$

On the set of transients T it is also possible to define a XOR function among n transients, in such a way that it always represents the worst-case scenario of glitches propagation. Given $t_1, ..., t_n \in T$, $w = XOR(t_1, ..., t_n)$ is such that:

$$\alpha(w) = XOR(\alpha(t_1), ..., \alpha(t_n))$$
$$\omega(w) = XOR(\omega(t_1), ..., \omega(t_n))$$
$$l(w) = 1 + \sum_{i=1}^{n}(l(t_i) - 1) = 1 - n + \sum_{i=1}^{n} l(t_i)$$

Glitch-counting algorithm

Given a circuit with m inputs and k gates, we denote by $\underline{X} = (X_1, X_2, ...X_m)$ the *vector of input variables* and by $\underline{s} = (s_1, s_2, ..., s_k)$ the *vector of state variables*, which are the gates' outputs. We define with $\underline{s}^h$ vector $\underline{s}$ without $h - th$ component, $1 \leq h \leq k$. The Boolean function $S_j : \mathbb{Z}_2^m \times \mathbb{Z}_2^{k-1} \longrightarrow \mathbb{Z}_2$ is called *excitation* and it is the function that allows to compute the state variable $s_j = S_j(\underline{X}, \underline{s}^j) = S_j(X_1, ..., X_m, s_1, ..., s_{j-1}, s_{j+1}, ..., s_k)$. The vector $S(\underline{X}, \underline{s}) = (S_1(\underline{X}, \underline{s}^1), ..., S_k(\underline{X}, \underline{s}^k))$ is called the *vector of excitations of the circuit*.
Initially, the circuit is in a stable state $(\underline{a}', \underline{b})$; then the input bits change from $\underline{a}'$ to $\underline{a}$ and the input states vector is set to **a** $= \underline{a}' \circ \underline{a} = (a_1 \circ a_1', a_2 \circ a_2', \ldots a_m \circ a_m')$ where $a_i \circ a_i'$ is $a_i a_i'$ if $a_i \neq a_i'$ or a_i if $a_i = a_i'$. After the input bits change, some state variables become unstable. Indeed now they do not represent the correct logic output of their corresponding gates. All unstable variables are changed at the same time to their excitations. We obtain a new internal state s^{*0}, which is a vector of transients from the set T. The process is then repeated, computing at each round h the new internal state s^{*h}, until the last state variable vector computed is equal to the second-last vector. A pseudo-code of the Glitch-counting algorithm is reported in Algorithm 1.

Algorithm 1 Glitch-counting algorithm

Input: The initial stable state $(\underline{a}', \underline{b})$, the new input $\underline{a}$ and the vector of excitations among transients $S(\underline{\mathbf{X}}, \underline{\mathbf{s}})$ of a circuit.

Output: A list of k transients, one for each gate's output, describing the worst possible switching activity during the transition $\underline{\mathbf{a}} = \underline{a}' \circ \underline{a}$.

```
1: $h \leftarrow 0$;
2: $\underline{\mathbf{a}} \leftarrow \underline{a}' \circ \underline{a}$;
3: $\underline{\mathbf{s}}^{*0} \leftarrow \underline{b}$;
4: repeat
5:     $h \leftarrow h + 1$;
6:     $\underline{\mathbf{s}}^{*h} \leftarrow S(\underline{\mathbf{a}}, \underline{\mathbf{s}}^{*(h-1)})$;
7: until $\underline{\mathbf{s}}^{*h} = \underline{\mathbf{s}}^{*(h-1)}$;
8: return $\underline{\mathbf{s}}^{*h}$;
```

4.2.3 LP Model

The *LP (Leakage Path) model* [31] is a mathematical abstraction that expands the functionalities of the glitch-counting algorithm; it is a tool that allows to evaluate whether a circuit has a critical leakage from the security point of view. This model is based on three entities: *Input Variables*, *Literal Transients* and *Literifiers*.

Definition 4.4. *Input Variables* are the circuit's inputs, and they are the only part of the circuit that can trigger a signal propagation. Generally, if we consider only one gate, Input Variables of this gate are variables that are directly given as input to a gate. We denote Input Variables as X_j, or $\mathbf{X}_j$ if they are seen as transients.

Given a circuit with m inputs, we can enumerate all the Input Variables, calling each of them i instead of X_i; $I = \{1, ..., m\} \subset \mathbb{N}$ is the *set of Input Variables*.

Definition 4.5. Given a gate with m inputs, namely $\underline{X} = \{X_1, ..., X_m\}$, we call *Literal Transient* any subset of $I = \{1, ..., m\}$. The set of Literal Transients is denoted by $\mathcal{P}(\{1, ..., m\})$, where $\mathcal{P}$ is the power set. Each of this Literal Transient is a list of Input Variables, which are responsible for the switching activity (and then power consumption) and could then be leaked according to our power model.

Literal Transients play a central role in the LP Model, because they define which Input Variables can be leaked in a circuit at gate level, caused by combinatorial logic.

Definition 4.6. *Literifier* is the function

$$\mathcal{L}_f : (T \times I)^r \longrightarrow I$$

i.e., given a gate that implements a boolean function f, and that depends on r gates and/or Inputs Variables labelled $((t_1, l_1), ..., (t_r, l_r)) \subset (T \times I)^r$, the Literifier gives the corresponding Literal Transient. This gate is then labelled itself with a couple composed by a transient, calculated by glitch-counting algorithm, and a Literal Transient, computed by the Literifier.

For some applications of LP Model on a circuit, see section 4.2.7.

4.2.4 Propagation Sequences

The first step, fundamental for the description of propagation sequences, is to define a new time unit. Such time unit, which is typical of a circuit, refers to an ideal case with a null switching time (time necessary to the signal's stabilization in a transition from 0 to 1 or from 1 to 0).

Definition 4.7. A *Time Slot* is a time unit that refers to a fraction of the time of the signal propagation in a circuit, in which at most one change of the output bit of one gate occurs.

Given a circuit, we are interested in finding the gates that can be affected by different signal propagations. In particular, these gates are AND, OR and XOR gates. We start to study combinations on gates.

Definition 4.8. Given a gate with n inputs, we call *combination* the string $a_1a_2...a_n$, such that $a_i \in \{1,...,n\}$ and $a_i \neq a_j$ $\forall i \neq j$ and $1 \leq i,j \leq n$.

Definition 4.9. Given a gate with n inputs, Cb is the *set of all possible combinations* on n inputs; its cardinality is $n!$.

Given the previous definitions, it is easy to see a link between the set of combinations of n inputs and the symmetric group S_n.

Proposition 4.2.2. *Let n be the number of inputs, Cb the set of all combinations on n inputs and S_n the symmetric group on $\{1,...,n\}$. Then there is a bijective map ι between S_n and Cb.*

Now, starting from what has been exposed for gates, we can give the same definitions and framework for a generic circuit, with n inputs and m gates.
In order to make this study, we have to follow some steps.

1. At first, it is necessary to define the set of all inputs of the circuit, that can be seen as a set $I \subset \mathbb{N}$.

2. At each circuit's input X_j is linked the set $\{j\}$, with $1 \leq j \leq n$, called *Circuit Input Variable*.

3. At each $i-th$ gate ($1 \leq i \leq m$) is linked a set $Ig_i \subset I$, called *Gate Inputs Variables*, that corresponds to inputs that can have some effects on the gate, namely those inputs on which the gate depends on. It can be also seen as the union of Gate Inputs Variables and Circuit Inputs Variables before the $i-th$ gate. Namely, given $X_{j_1},...,X_{j_v}$ circuit's inputs that are also input of the $i-th$ gate and $g_{j_1},...,g_{j_w}$ gates of circuit which outputs are inputs of the $i-th$ gate:

$$Ig_i = (\bigcup_{s\in\{1,...,v\}} \{j_s\}) \bigcup (\bigcup_{s\in\{1,...,w\}} Ig_{j_s})$$

   If a gate depends on a circuit's input X_j ($1 \leq j \leq n$) in two different ways, then we have to add one input circuit $n+1$ to set $I = \{1,...,n\}$, creating an extended set $\bar{I} = \{1,...,n+1\}$, and the subset of $\bar{I}$ linked to the gate will contain both j and $n+1$. The same happens also if a gate depends on a circuit's input X_j in r ways, with $r > 2$. This procedure is repeated for each gate that depends on one or more inputs in two or more ways, adding every time elements to the extended set $\bar{I}$.

4. The number of time slots of the circuit is the sum of the cardinalities of all the Gate Inputs Variables defined in the previous point:

$$n_{ts} = \sum_{i=1}^{m} |Ig_i|$$

5. The number of combinations for this circuit is given by $l!$, where l is the cardinality of $\bar{I}$ (which is the set linked to the gate that produces the circuit's output bit).

Given a circuit with n inputs and m gates, and a vector of time slots of a combination c, we can associate to this vector two other vectors of the same length:

- TSi is composed of cells that define which switch of circuit's input have triggered the change in the corresponding time slot;
- TSg reports which gate is considered in the corresponding time slot.

Glitch-counting algorithm for combinations

Initially, the circuit C is in a stable state $(\underline{a}', \underline{b})$. Then the input bits change from $\underline{a}'$ to $\underline{a}$. Now we have an additive information with respect to the worst-case, i.e. the vector of time slots of a combination $c \in Cb$, and then we know how the signal propagates in the circuit. A pseudo-code of the Glitch-counting algorithm for combinations is reported in Algorithm 2.

Algorithm 2 Glitch-counting algorithm for combinations

Input: The initial stable state $(\underline{a}', \underline{b})$, the new input $\underline{a}$, the vector of excitations among transients $S(\underline{\mathbf{X}}, \underline{\mathbf{s}})$ of a circuit, n_{ts} the number of time slots, two vectors TSi and TSg (that correspond to time slots vector of combination c).

Output: A list of m transients, one for each gate's output, describing switching activity during the transition $\underline{\mathbf{a}} = \underline{a}' \circ \underline{a}$ for combination c.

```
1: $h \leftarrow TSi(1)$;
2: $\underline{\mathbf{a}}(h) \leftarrow \underline{a}'(h) \circ \underline{a}(h)$;
3: for $j = 1 \rightarrow n_{ts}$ do
4:     if $TSi(j) \neq h$ then
5:         $h \leftarrow TSi(j)$
6:         $\underline{\mathbf{a}}(h) \leftarrow \underline{a}'(h) \circ \underline{a}(h)$;
7:     end if
8:     $\mathbf{s}_{TSg(j)} \leftarrow S_{TSg(j)}(\underline{\mathbf{a}}, \underline{\mathbf{s}})$
9: end for
10: return $\underline{\mathbf{s}}$;
```

Propagation Sequences as quotient set

The gate that produces the circuit's output bit is affected by switches happening all over the circuit (and then by glitches) and their propagation in it. Then, we can give the following equivalence, in which we use transients computed by the Glitch-counting algorithm for a given combination.

Definition 4.10. Given a circuit, let $\{X_1^{t_0}, ..., X_n^{t_0}\}$ be the set of inputs at time t_0 and $\{X_1^{t_1}, ..., X_n^{t_1}\}$ the set of inputs at time t_1. Let Cb be the set of all combinations for the circuit. We define an equivalent relation $\sim_{Cb}$ in Cb such that, for all $c_1, c_2 \in Cb$, $c_1 \sim_{Cb} c_2$ if and only if, for each possible set of inputs $\{\{X_1^{t_0}, ..., X_n^{t_0}\}, \{X_1^{t_1}, ..., X_n^{t_1}\}\}$, the transient at the gate producing the circuit's output bit (computed with the glitch-counting algorithm on combination c_1 and combination c_2) is always the same.

Definition 4.11. Let n be the number of inputs of a circuit, Cb the set of its combination and $\sim_{Cb}$ the equivalence on it defined above. The quotient set

$$\frac{Cb}{\sim_{Cb}}$$

is the set of *Propagation Sequences*, and *Propagation Sequences* are the equivalent classes of Cb on the equivalence $\sim_{Cb}$.

Definition 4.12. The *number of Propagation Sequences* for a particular circuit is the cardinality of the set of Propagation Sequences, namely

$$n_{PS} = \left| \frac{S_n}{\sim_{S_n}} \right|$$

An example of the use of combinations and propagation sequences is presented in section 4.2.7.

Comparison between Worst-Case Scenario and Propagation Sequences

The study of propagation sequences is justified starting from what has been stated at the beginning of this section: given a circuit and some specific inputs for it at two different consecutive times t_0 and t_1, the Worst-Case Scenario of Glitches Propagation represents what can happen in the worst possible case. Instead, propagation sequences on a circuit are all the possible signal propagations that can happen, and some of them can be different from the worst-case.

Example 4.2.1. *In Figure 4.36 is given an example circuit. In this circuit the following six time slots can be identified:*

- $TS_1 \longrightarrow$ *a switch of* X_1 *causes a (possible) change of* s_3 *output bit*
- $TS_2 \longrightarrow$ *a switch of* X_2 *causes a (possible) change of* s_1 *output bit*
- $TS_3 \longrightarrow$ *a switch of* s_1 *output bit causes a (possible) change of* s_2 *output bit*
- $TS_4 \longrightarrow$ *a switch of* s_2 *output bit causes a (possible) change of* s_3 *output bit*
- $TS_5 \longrightarrow$ *a switch of* X_3 *causes a (possible) change of* s_2 *output bit*
- $TS_6 \longrightarrow$ *a switch of* s_2 *output bit causes a (possible) change of* s_3 *output bit*

So, in this circuit there are two possible propagation sequences:

PS_1:	PS_2:
TS_1	TS_1
TS_2	TS_5
TS_3	TS_6
TS_4	TS_2
TS_5	TS_3
TS_6	TS_4

All the other $(3! - 2)$ *combinations define a propagation sequence equivalent to one of the two above.*
On this circuit with these two propagation sequences, two examples are described.

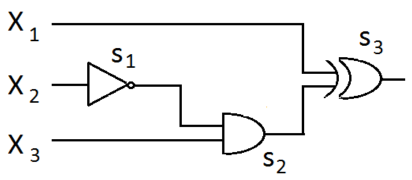

Figure 4.36: Circuit with transients.

1. *In the first case, represented in Figure 4.37 on the left, we consider inputs at time t_0 $\{X_1, X_2, X_3\} = \{0, 1, 1\}$, and at time t_1 $\{X_1, X_2, X_3\} = \{0, 0, 0\}$. What happens in the worst-case can be computed with the glitch-counting algorithm (depicted at the top). What happens in the circuit if the signal propagation is described by first propagation sequence is in the middle, with the second propagation sequence in the bottom. In this case, with these inputs, the worst-case matches with the first propagation sequence.*

2. *In the second case, represented in Figure 4.37 on the right, we consider inputs at time t_0 $\{X_1, X_2, X_3\} = \{0, 0, 0\}$, and at t_1 $\{X_1, X_2, X_3\} = \{0, 1, 1\}$. Again, what happens in the worst-case can be computed with the glitch-counting algorithm (depicted at the top). What happens in the circuit if the signal propagation is described by the first propagation sequence is in the middle, with the second propagation sequence at the bottom. Differently than the previous case, the propagation sequence that describes the worst-case is the second one.*

Then, sometimes the worst-case coincides with one propagation sequence, and sometimes with the other one.

Observation 1. In a real circuit, if there are no changes of physical conditions (like temperature of environment) the propagation of signal is always the same, and then the propagation of the signal respects only one specific propagation sequence. A propagation sequence in a circuit is dictated by intrinsic properties of the circuit, as the length of the wires, or the material of which is realized. In our work, given a circuit, every time we consider all possible propagation sequences, to give a complete analyses of all possible cases that can happen in real circuits, with the same gates and connections of the ideal one, but with different features.

4.2.5 Order of an Attack

Probes that are considered in [78] are metal needles placed on wires of interest, and their task is to read off the value carried along the wires during the computations. We can revise this concept, introducing a new instrument similar to the probe but with a different task.

Definition 4.13. An *ideal probe* is a probe that can reveal the value carried along a specific wire, discerning all possible switches in every time slot.

In general, a measuring instrument has a proper spatial resolution and time resolution.

- *Spatial resolution* is inherent to the precision whereby it is possible to measure the consumption: if an instrument is very sophisticated, it can recover consumptions in each wire of a circuit, while a less sophisticated instrument records a sum of these

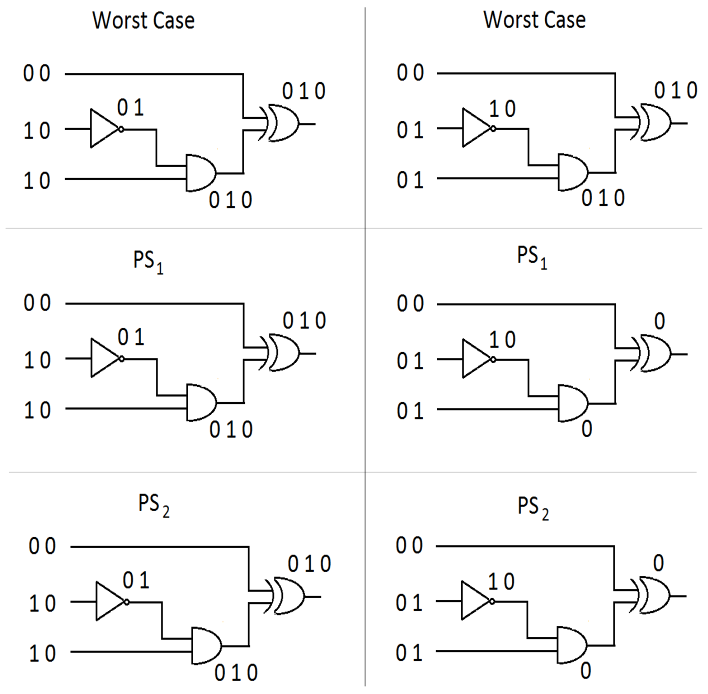

Figure 4.37: Circuit with transients.

consumptions. Ideal probes can be placed on every wire in a circuit, and then they have maximal spatial resolution.

- *Time resolution* is defined by the bandwidth of the instrument: the wider the bandwidth is, more switches can be identified in time. We can say that ideal probes have infinite bandwidth, since they can identify any switch in any time slot considered, and therefore with maximal time resolution.

It follows from this that ideal probes are instruments that can recover all switches that happen in the circuit (i.e, they can read activity in all time slots) and at level of all gates. In contrast, a real measuring instrument cannot recover all switches that happen (less time resolution) and/or cannot monitor all gates (less spatial resolution).
As mentioned also in 4.2.1, some authors refer to the number of probes placed on the wires of the circuit [78], while some others refer to the statistical moment used to implement the attack [94].
Often these two definitions are discordant and create some misunderstandings. For example, in [99] and [100] the authors describe threshold implementations, a countermeasure against side-channel attacks in presence of glitches; specifically in [100], they enumerate three properties that this countermeasure must have to offer protection against an attack of the first order, without specifying what they accept as order of an attack.
Starting from these concepts, we think that a new definition of order of an attack is needed, collecting together all notions exposed until now.

- The first concept that is necessary for the definition is the *statistical moment* used to implement the side-channel attack. The order of the statistical moment is a relevant notion about the attack, and it must not be omitted; this point conciliates our definition with what is exposed in [94].

- Another relevant concept is the *number of points* examined in the circuit, namely the parts of the circuit that an attacker has to monitor to implement an attack. In this way, the concept exposed in [78] is incorporated in our definition.
- The last concept is how many points in time have to be considered, or, in other words, the *time instances* that an attacker has to consider to attack the circuit. This concept reflects definition of $v-$variate attack in [94], but where v indicates different time instances in the same clock cycle.

These three dimensions contain all the fundamental features of an attack; then, starting from them, definition 4.14 follows.

Definition 4.14. The order of an attack is an element (sm, sp, ti) of the tridimensional space $\mathbb{R}^3$ such that:

- sm is the *order of the statistical moment* used to distinguish two distributions;
- sp id the number of *spatial points* monitored in the circuit;
- ti is the number of *time instances* analysed.

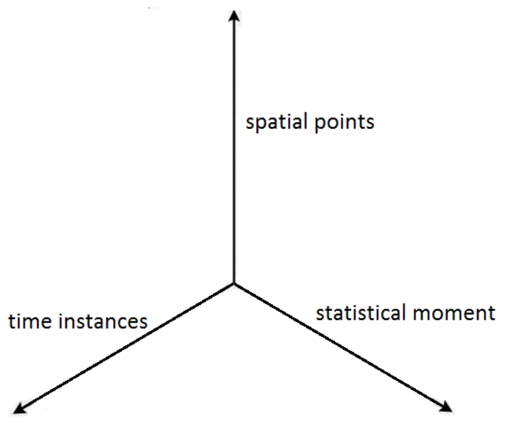

Figure 4.38: Order of an attack, in three dimensions: statistical moment, spatial points and time instances.

Each time slot scans a time instance and it refers to one gate only; gates in the circuit are the spatial points considered. Since we are interested in the gate considered for each time slot, also TSg is important (section 4.2.4), because this vector defines our possible monitored points in the circuit. Therefore, the features of an attack that we have to recover in order to define its order are:

- the order of the statistical moment used;
- the number of spatial points considered, that are output wire of gates in the circuit;
- the number of time instances, that are time slots kept in consideration for the attack.

The space of the order of an attack is a tridimensional space, namely a set S such that $S \subset \mathbb{R}^3$. For each circuit that we can consider, this set is contained but not equal to $\mathbb{R}^3$ since there are some points in $\mathbb{R}^3$ that cannot represent any attack. For example, given a circuit C, the number of points in space is limited by the number of gates in C. Furthermore, if the time slots corresponding to each gate in C are at most n, in S there cannot be a point $(sm, 1, n+r)$ with $r \geq 1$, because it does not exist any gate with $n+r$ instances.
The number of time slots for each gate can be read in the vector TSg, because it is the length of TSg.

Example 4.2.2. *If we consider the TSg vector*

3
1
3
2
3
2
3

we have the following information:

- *the time slots are 7, and then the highest number of time instances that we can consider is 7;*
- *the number of gates in the circuit is 3;*
- *it is possible to consider one time instance for the gate 1, two instances for the gate 2 and four for the gate 3.*

An attack unlikely involves higher values for each dimension, since an attack of higher order is more expensive. Furthermore, there are limitations in circuit's conformation when recovering traces for a side-channel attack and the noise becomes more relevant with an increase of the statistical order. Then, we can resume what explained until now in the following points:

- if the order of statistical moment used for the attack grows, then the noise in recovering traces becomes more relevant and the attack more difficult to implement;
- if the number of points considered in the circuit is high, it means that the instrument used for the measurements has to be more sophisticated (and then more expensive);
- if the number of time instances necessary to implement the attack is high, the measuring instrument has to recover all the switches that happen in time slots, thus requiring wider bandwidth (once again resulting more expensive).

Observation 2. If an instrument reads a consumption equivalent to the sum of consumptions recovered by n probes placed on n different points of the circuit, then the spatial points used for an attack that involves this instrument are n; time instances are all m time slots referring to the spatial points considered ($m \geq n$). The attack described is a $(sm, \mathbf{n}, \mathbf{m})$ attack.

Examples of attacks on AES

To validate what has been exposed until now in this section, some examples of attacks are exposed below, focusing on how the order of these attacks can be reformulated as a tridimensional point.
The first two examples are two different possible attacks on **AES without countermeasures**. The part of the algorithm that generally is subjected to attacks is the non-linear one: in this part of AES, plaintext of 128 bits is split into blocks of 8 bits, and SBOX function is applied on each block independently, producing an output of 8 bits.
In a device that implements AES, the nonlinear function can be realized in two different ways:

- with one circuit that executes SBOX, and then the function is applied on the blocks of the plaintext serially, one block at time, in different clock cycles;
- with 16 circuits, each one executing SBOX, and then the function is applied on the blocks of the plaintext in parallel executions.

In the first case, not much physical space is employed, but it demands more time of execution; instead, the second realization allows more speed of execution, but there is more physical space in the device.
If there is only a circuit that implement the SBOX function, an attack using the first statistical moment (mean) can be implemented by monitoring the output of the SBOX, each time it processes a new block: thus it is necessary only one ideal probe placed at the output wire of the SBOX circuit, but one time instance for each block. To recover 8 bits of the key in each clock cycle, the attack is a (1,1,1) attack, in accordance with definition 4.14. To recover the complete key, this attack procedure must be repeated sixteen times, one for each byte.
In the second case, it is possible to carry out an attack (again using the first statistical moment) by placing 16 ideal probes at the output wires of each SBOX. In this case we are considering, in a clock cycle, 16 spatial points, and 16 time instances (time slots); then this is a (1,16,16) attack that allows to recover all blocks of 8 bits of the key.
In the second case, if we have not an instrument with a wide bandwidth, but we can recover only power consumption as summation of consumptions at the output gates of all SBOXes, it is possible a (1,16,1) attack, that allows to recover 8 bits of the key, and in which consumptions of the other SBOXes create noise. To recover all block of the key, this attack procedure must be repeated sixteen times, one for each byte.
This exposition does not consider all variables that can make an attack more difficult to implement; for example, we do not consider that to compare some attacks' implementations it is important also the number of traces that are necessary to recover the secret.
Another example on AES can be the situation described in [103]: in this article, a second order DPA attack on **AES with mask** is presented. This attack targets the XOR operation of a byte of the key and a byte of the masked data (two spatial points), in two different moments (two time instances); finally, first statistical moment (mean) is used for this attack. Then, through the new definition of the order of an attack, the attack in [103], that allows to recover 8 bits of the key, is a $(1, 2, 2)$ attack (if the measuring instrument is sophisticated enough and it has a wide bandwidth).

4.2.6 Power Consumption Models

Glitches described by transients in a circuit are generated by switches of input variables, from bits in the original stable state at time t_0 to new bits at time t_1. Power Consumption models are methods to recover consumption in a circuit, exploiting propagation sequences, and based on the model of power consumption in definition 4.15.

Definition 4.15. The *Model of power consumption* used in this section provides that at each time slot the consumption is 1 if the output of the corresponding gate has a switch; otherwise if the output signal of the gate is constant in the considered time slot, then the consumption is 0.

Observation 3. The model defined in 4.15 is directly connected with glitches, since a glitch in a gate defines a consumption p such that $p \geq 2$; vice versa, if $p = 0$ there is not any change in the gate, while if $p = 1$ there is a rightful change, that is present also without considering the glitches.

Considering any propagation sequence, it is possible to collect the consumptions caused by glitches in different ways, depending on the particular time unit and the method of collection.

- *Different time units*: the consumption can be computed at the end of a clock cycle, as the summation of consumptions in all the gates of the circuit, or at intermediate levels; these intermediate levels are *time slots*.
- *Different methods of collection*: it is possible to collect consumptions for each input at time t_1 and each previous input at time t_0; otherwise, consumptions can be collected computing, for all inputs at time t_1, a mean over all inputs at time t_0.

Starting from these features and combining them with each other, we define four Power Consumption Models.

Power Consumption Model 1

The first Power Consumption Model consists in the collection of every consumption of each gate in the circuit (in the meaning of definition 4.15), for each input at time t_1 and considering each possible input at time t_0. This is the case of an attacker with a measuring instrument with both very high spatial and time resolutions.
Given a circuit with n inputs, let $\underline{X}$ be the Input Variable set (see section 4.2.3); for each $x \in \underline{X}$, x is an element in $\mathbb{Z}_2$, and there are 2^n different Input Variable sets depending on different values that elements of $\underline{X}$ can take. If we consider all values that inputs can take at time t_0 and all at time t_1, in total there can be 2^{2n} possibilities.
Now, given a specific propagation sequence in the circuit, we consider all possible inputs at time t_0 (that are 2^n), all possible inputs at time t_1 (that are 2^n), and for each of these we assign at all time slots 1 or 0, if in this particular time slot there is a switch or not.
Power Consumption Model 1 is the most detailed case, since with this an attacker can distinguish all gates and all time instances. Also for this reason, it is the most difficult case to achieve and the most unrealistic.

Power Consumption Model 2

Power Consumption Model 2 consists in the collection of every change of each gate in the circuit, for each input at time t_1, computing the consumption mean over all possible input at time t_0.
Power Consumption Model 2 is such that an attacker, as for Power Consumption Model 1, can monitor all gates of the circuit and for all time instances, but she knows only inputs at

time t_1 and not previous inputs at time t_0. Then we have only 2^n different vectors of time slots, corresponding to 2^n possible values at time t_1.
This case is more likely than the first one, since very often an attacker can guess input at time t_1, but she does not know what was stored in the register, at time t_0.

Power Consumption Model 3

In Power Consumption Model 3 the sum of all consumptions in the circuit is collected, for each input at time t_1 and considering each possible input at time t_0. As Power Consumption Model 2, also this model describes a situation in which the attacker knows less information with respect to Power Consumption Model 1: in this case, the attacker knows all inputs at time t_0 and all inputs at time t_1, but she cannot monitor all gates in the circuit and she can only recover a summation of consumptions in them.
Like Power Consumption Model 1, in this model we have 2^{2n} possibilities of Inputs Variable, 2^n at time t_0 and 2^n at time t_1.

Power Consumption Model 4

The last Power Consumption Model that we present consists in a collection of the sums of every consumption in the circuit, for each input at time t_1, and computing the consumption mean over all possible inputs at time t_0. This is the case in which an attacker has a measuring instrument with low spatial and time resolutions.
If an attacker decides to collect consumptions in a circuit with this model, this means that she has a restricted amount of information: she knows only inputs at time t_1 and not inputs at time t_0; moreover, she cannot recover consumptions for each particular time slot and gate, and then the registered consumption is the sum of all consumptions in the circuit.
Similar to Power Consumption Model 2, in Power Consumption Model 4 only 2^n possible inputs values are considered, since the attacker knows only inputs at time t_1.

Comparison among Power Consumption Models

Every Power Consumption Model has some specific features and, in accordance with them, it is possible to create the following scheme:

$$\begin{tikzcd}
\text{Power Consumption Model 4} \arrow[d] & \begin{array}{c}\underline{\text{Worst-case}}\text{ for an attack:}\\ \text{if there is an attack}\\ \text{exploiting this model,}\\ \text{then it is possible to attack also}\\ \text{other models}\end{array} \arrow[dd] \\
\begin{array}{c}\text{Power Consumption Model 3}\\ \text{Power Consumption Model 2}\end{array} \arrow[d] & \\
\text{Power Consumption Model 1} & \underline{\text{Best case}}\text{ for an attack}
\end{tikzcd}$$

The worst-case for an attack is represented by Power Consumption Model 4: this model is used by an attacker when she does not know inputs at time t_0 but only inputs at time t_1, and also she cannot monitor all gates in the circuit.
Instead, the best case for an attack is Power Consumption Model 1, because in this case the attacker can recover all helpful information that she wants, since she has access to all gates in the circuit and she knows all inputs (at time t_0 and at time t_1). However, this is also the situation hardest to realize, because generally an attacker does not know inputs at time t_0 and she cannot monitor all gates.
Both Power Consumption Model 2 and 3 have some features of Power Consumption Model 1 and Power Consumption Model 4, and then they are between the worst-case and the best one.

Observation 4. The worst-case for an attack is not the same as the worst-case of glitches propagation: they are defined in two different frameworks, the first one is in terms of power consumptions' collection and the second one is referred to the signal propagation in the circuit (and thus to glitches).

4.2.7 Application to KECCAK

As we said in section 4.2.1, our main example is KECCAK, and in particular its nonlinear part (the χ function) is the main object of our study, since generally the target of a side-channel attack is the nonlinear part of cryptographic algorithms. In this section, we apply LP model and Power Consumption Model to define some possible fatal leakage of the circuits and to know if there is some attack, the order of which we express in three dimensions.

Unshared χ

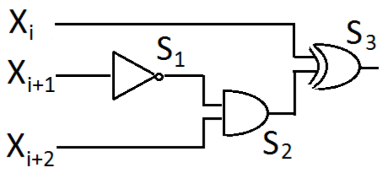

Figure 4.39: Circuit of χ function.

A first example is χ with a single-share. Any leakage at gate level is a leakage of a native variable, so every transition consuming power produces a leakage of a native variable. χ has three input bits, and then all the possible transitions are $2^3 \cdot 2^3 = 64$ (2^3 inputs at time t_0 and 2^3 inputs at time t_1). Trivial transitions, i.e. transitions where inputs at time t_0 are the same as inputs at time t_1, do not leak anything, and they are $2^3 = 8$. The LP model shows that there also exist non-trivial transitions that do not lead to any switch of gates' outputs, and hence to any leakage. Referring to Figure 4.39, for instance, if X_3 changes, but X_1 is fixed (to 0 or 1) and X_2 is fixed to 1, no switch is produced. This happens in four cases, namely when the input transitions are $(1,1,1) \rightarrow (1,1,0)$, $(0,1,1) \rightarrow (0,1,0)$, $(1,1,0) \rightarrow (1,1,1)$ and $(1,1,0) \rightarrow (1,1,1)$. In conclusion, there are four non-trivial transitions out of $2^6 - 2^3 = 56$ that do not leak anything, which means that the remaining roughly 81% of transitions are vulnerable. In Figure 4.40, on the left there is an example of transition that produces some leakages (for X_1 at gate s_3, for X_2 at each gate and for X_3 at gate s_2 and s_3), and on the right there is an example of non-trivial transition that does not produce any leakage.

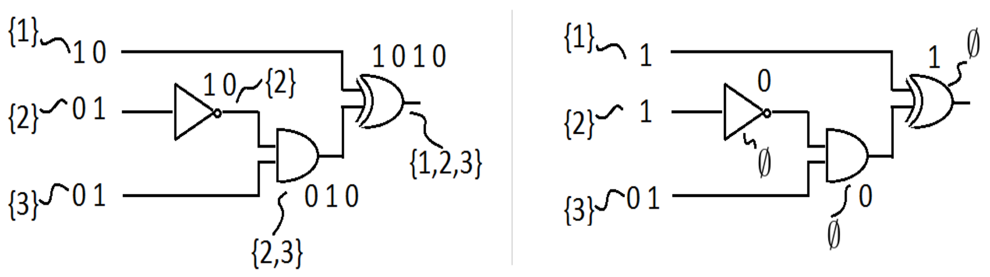

Figure 4.40: Examples of LP model applied to the unshared χ, in two different cases: on the left it is a case in which there are same leakages, and on the right it is a case where there is no leakage.

χ is a function applied five times to each KECCAK state's row, each time on three different bits. A KECCAK row is composed by five bits, and thus there are 2^5 possible values, denoted $\underline{x}^0, ..., \underline{x}^{31}$, such that $\underline{x}^j = [x_1^j, x_2^j, x_3^j, x_4^j, x_5^j]$ with $0 \leq j \leq 31$.
Referring another time to Figure 4.39, we can note that this circuit has three inputs, x_i, x_{i+1} and x_{i+2}, and three gates, s_1, s_2 and s_3. Time slots are:

TS_1 $\rightarrow$ change of output bit of s_3 caused by the switch of x_i

TS_2 $\rightarrow$ change of output bit of s_1 caused by the switch of x_{i+1}

TS_3 $\rightarrow$ change of output bit of s_2 caused by the switch of output of s_1

TS_4 $\rightarrow$ change of output bit of s_3 caused by the switch of output of s_2 (in TS_3)

TS_5 $\rightarrow$ change of output bit of s_2 caused by the switch of x_{i+2}

TS_6 $\rightarrow$ change of output bit of s_3 caused by the switch of output of s_2 (in TS_5)

At first, we define how many propagation sequences there can be in this circuit. Input variables are three, then there are six combinations. Among them, combinations 123, 213 and 231 represent the same propagation sequence (because transients of output gate are always the same), and 132, 312 and 321 are the same propagation sequence. Therefore, we have two propagation sequences, with representatives:

- 123 for the first propagation sequence,
- 132 for the second propagation sequence.

Since with LP Model some leakages of native variables have been detected, we can analyse if these leakages are exploitable with a Power Consumption Model. The first model studied is the fourth, since if an attack with this **Power Consumption Model 4** is found, then there is an attack with all the other models (since it represents the worst-case for an attack).
We have computed all consumptions with both propagation sequences and they are reported in Table 4.5 .

Table 4.5: Consumptions of unshared χ in accordance with Power Consumption Model 4, considering both propagation sequences.

x_i	x_{i+1}	x_{i+2}	*consumption mean*	
			with propagation sequence 123	*with propagation sequence 132*
0	0	0	1.5	2.5
0	0	1	2.5	2.5
0	1	0	1.5	1.5
0	1	1	2.5	1.5
1	0	0	1.5	2.5
1	0	1	2.5	2.5
1	1	0	1.5	1.5
1	1	1	2.5	1.5

Only the first propagation sequence is considered, since the discussion for the second one is very similar.
We can notice that the consumption for this propagation sequence is 1.5 if x_{i+2} is 0 and 2.5 if x_{i+2} is 1, thus there is a relation between the power consumed and a native value, and this is a fatal leakage. These power consumptions recovered with Power Consumption Model 4 can be exploited for a CPA attack using the mean as statistical moment. Then we exploit information recovered with power consumption model 4 to find one bit of the key with a $(1, 3, 6)$ attack, since we use the mean (first statical moment) and spatial points are all gates in the circuit (observation 2), considering all six time slots.

χ with two shares

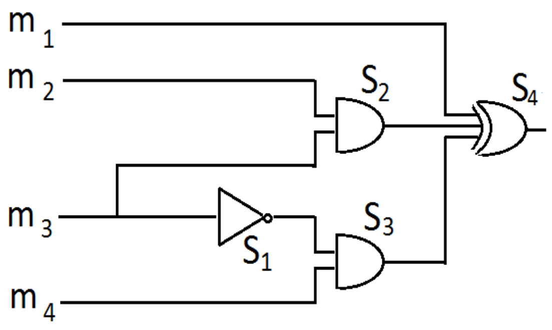

Figure 4.41: Atomic circuit of χ with two shares.

Secret sharing with two shares on χ can be a secure scheme in software (namely, it does not leak any native variable), if the order of operations is kept fixed from left to right. The same is not true in hardware, for instance because of glitches.
Since the two branches are symmetric, we can consider the circuit in Figure 4.41 when it produces a_i, then $m_1 = a_i$, $m_2 = b_{i+2}$, $m_3 = a_{i+1}$ and $m_4 = a_{i+2}$. Since there are four input variables, all possible transitions are 2^8, and 2^4 of them are trivial. As analysed in [31], in this circuit a natural vulnerability arises when the two variables a_{i+2} and b_{i+2} are processed at the same moment by the last XOR gate, as this could leak the value $a_{i+2} \oplus b_{i+2}$, that is

an unshared native value. If a particular transition generates this leakage, it can be read through the LP model: it is present when literal transient for the last XOR gate contains $\{2\}$ and $\{4\}$. An example of what has been explained is represented in Figure 4.42: the transition is $(1,1,0,0) \rightarrow (1,0,1,1)$, and there is a leakage of both m_2 and m_4, namely b_{i+2} and a_{i+2}. By running the model for all the $2^8 - 2^4$ non-trivial possible input transitions, it is found that 32 of them produce the defined vulnerability, i.e. 12.5% of all 2^8 transitions.

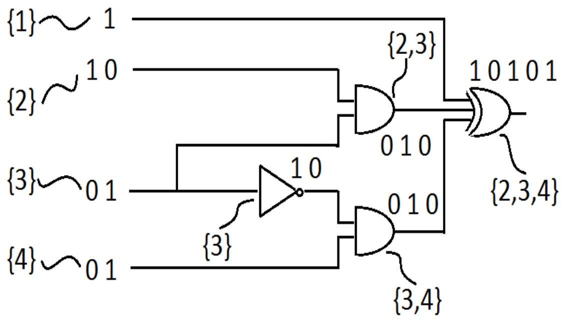

Figure 4.42: Example of LP model applied to a branch of χ with two shares: at XOR level there can be some leakages of m_2, m_3 and m_4.

Now we explain how these leakages can be exploited to compute an attack using power consumption models. First step is to define propagation sequences in the atomic circuit of χ with two shares (Figure 4.41); it has four inputs m_1, m_2, m_3, m_4 and four gates s_1, s_2, s_3, s_4. Time slots are:

TS_1 $\rightarrow$ change of output bit of s_4 caused by the switch of m_1

TS_2 $\rightarrow$ change of output bit of s_2 caused by the switch of m_2

TS_3 $\rightarrow$ change of output bit of s_4 caused by the switch of output of s_2 (in TS_2)

TS_4 $\rightarrow$ change of output bit of s_2 caused by the switch of m_3

TS_5 $\rightarrow$ change of output bit of s_4 caused by the switch of output of s_2 (in TS_4)

TS_6 $\rightarrow$ change of output bit of s_1 caused by the switch of m_3

TS_7 $\rightarrow$ change of output bit of s_3 caused by the switch of output of s_1

TS_8 $\rightarrow$ change of output bit of s_4 caused by the switch of output of s_3 (in TS_7)

TS_9 $\rightarrow$ change of output bit of s_3 caused by the switch of m_4

TS_{10} $\rightarrow$ change of output bit of s_4 caused by the switch of output of s_3 (in TS_9)

We can notice that a possible change of circuit's input m_3 can have effect on the last gate s_4 following two ways (TS_4/TS_5 and TS_6/TS_7/TS_8). The set corresponding to the last gate is $I = \{1,2,3,4,5\}$, hence the number of combinations is $5! = 120$. The number of propagation sequences is 4, since the quotient of the set of combination Cb on the equivalent relation $\sim_{Cb}$ has four equivalent classes, composed by thirty elements each one, and the chosen representative elements are:

- 12345 for propagation sequence 1
- 12354 for propagation sequence 2
- 13245 for propagation sequence 3
- 13254 for propagation sequence 4

For some inputs, there can be some leakage of the native variable $x_{i+2} = m_2 \oplus m_4$ at XOR gate. We start to analyse if it is possible to use **Power Consumption Model 4** to exploit these leakages, otherwise we will consider the other Power Consumption Models. Consumptions with all the propagation sequences are given in Table 4.6.

Table 4.6: Consumptions of an atomic circuit of χ with two shares in accordance with Power Consumption Model 4, considering all propagation sequences.

m_1	m_2	m_3	m_4	$m_2 \oplus m_4$	*consumption mean*			
					PS 12345	PS 12354	PS 13245	PS 13254
0	0	0	0	0	3	2	3	2
0	0	0	1	1	3	3	3	3
0	0	1	0	0	2	2	3	3
0	0	1	1	1	2	3	3	4
0	1	0	0	1	4	3	3	2
0	1	0	1	0	4	4	3	3
0	1	1	0	1	3	3	3	3
0	1	1	1	0	3	4	3	4
1	0	0	0	0	3	2	3	2
1	0	0	1	1	3	3	3	3
1	0	1	0	0	2	2	3	3
1	0	1	1	1	2	3	3	4
1	1	0	0	1	4	3	3	2
1	1	0	1	0	4	4	3	3
1	1	1	0	1	3	3	3	3
1	1	1	1	0	3	4	3	4

At first, we can notice that with propagation sequence 13245, the consumption mean is the same for all inputs at time t_1, therefore with this propagation sequence no information can be recovered, if consumptions are collected with Power Model Consumption 4. For the other propagation sequences, we can analyse the distributions of the consumption means.

- With propagation sequence 12345, the distribution of consumptions (third column in Table 4.6) is represented in Figure 4.43. It is possible to notice that the distribution of consumptions when $m_2 \oplus m_4$ is 1 and the distribution of consumptions when $m_2 \oplus m_4$ is 0 are the same, thus it is impossible to distinguish them.
- With propagation sequence 12354, the distribution of consumptions (fourth column in Table 4.6) is represented in Figure 4.44. Now distribution relative to 1 and distribution relative to 0 are different: they have the same mean, but different variance; this feature can be exploited for an attack with the second statistical order.

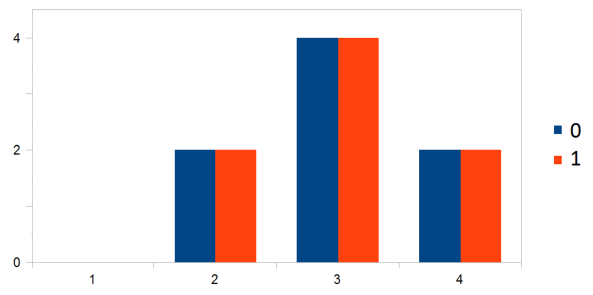

Figure 4.43: Distribution of consumptions collected with Power Consumption Model 4 for propagation sequence 12345.

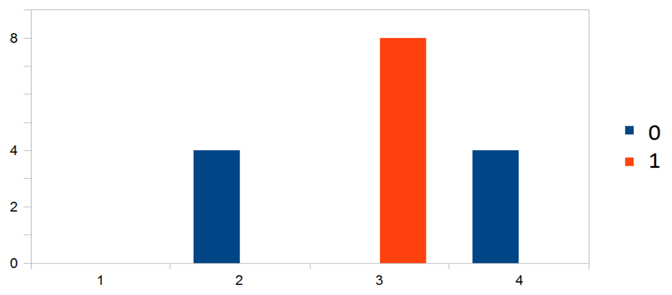

Figure 4.44: Distribution of consumptions collected with Power Consumption Model 4, propagation sequence 12354.

- With propagation sequence 13254, the distribution of consumptions (last column in Table 4.6) is the same of propagation sequence 1, and then the conclusions are equal.

In conclusion, with this Power Consumption Model it is possible to implement an attack to recover one bit of the key with the second statistical moment (considering propagation sequence 2), but not with the first (all the distributions have equal mean). With notation in three dimensions, to recover one bit of the key an attacker implements a (2,4,10) attack, since she uses the second statistical moment and the consumptions are recovered in ten time slots, referring to four gates.
Since for any propagation sequence there is no attack that uses the first statistical moment and Power Consumption Model 4, we can try to analyse **Power Consumption Model 2**. We recover consumptions for each time slot and for each input at time t_1, computing the mean on all inputs at time t_0. Analysing all the propagation sequences for χ with two shares, it is possible to notice that only propagation sequence 12354 can be interesting for our purpose, since it is the only one with some relations between consumptions and $x_{i+2} = m_2 \oplus m_4$; consumptions are reported in Table 4.7.
We are interested in columns TS_5 and TS_{10}, since, for each row, mean consumption in time slot TS_5 is proportional to m_2 and TS_{10} is proportional to m_4. Then, an attacker can implement a CPA considering consumption c, for each set of inputs of χ with 2 shares circuit, defined as:

$$c = (p_5 + p_{10})\text{mod}_2$$

Table 4.7: Consumptions of an atomic circuit of χ with two shares and propagation sequence 12354, in accordance with Power Consumption Model 2

m_1	m_2	m_3	m_4	$m_2 \oplus m_4$	*consumption in each time slot*									
					TS_1	TS_2	TS_3	TS_4	TS_5	TS_6	TS_7	TS_8	TS_9	TS_{10}
0	0	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0	0
0	0	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5
0	0	1	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0	0
0	0	1	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5
0	1	0	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0
0	1	0	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0.5	0.5
0	1	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0
0	1	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0.5	0.5
1	0	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0	0
1	0	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5
1	0	1	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0	0
1	0	1	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5
1	1	0	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0
1	1	0	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0.5	0.5
1	1	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0
1	1	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0.5	0.5

where p_5 is the power consumption recovered at time slot TS_5 and p_{10} the power consumption at time slot TS_{10}.
This attack, implemented to recover a bit of the key, in notation with three dimensions is a (1,1,2). Indeed:

- $sm = 1$, because to implement this attack an attacker uses the first statistical moment;
- $sp = 1$, because only the consumptions recovered at level of the XOR gate is considered, as if there was an ideal probe placed in that point of the circuit;
- $ti = 2$, because the useful consumptions are recovered in two time slots, i.e. two different time instances at level of the XOR gate (TS_5 and TS_{10}).

χ with three shares

The last case of study is χ with three shares. As it can be seen in Figure 4.45, in each isolated branch no single native variable can be leaked, since every function is independent of at least one share of each native variable (first function is independent from a_j, second function from b_j and third one from c_j). We can then consider two branches together: without loss of generality, we consider the first function (that produces a_i), and the second one (that produces b_i) (Figure 4.46).
In this case, if we monitor branches in at least two different points, for some transitions it is possible to observe leakages that, combined together in some way, can give information about a native variable. At first, there are eight input variables: a_{i+1}, a_{i+2}, b_i, b_{i+1}, b_{i+2}, c_i, c_{i+1} and c_{i+2}; then native input variables that can be leaked are $x_{i+1} = a_{i+1} \oplus b_{i+1} \oplus c_{i+1}$ and $x_{i+2} = a_{i+2} \oplus b_{i+2} \oplus c_{i+2}$. The total number of transitions is $2^8 \cdot 2^8 = 2^{16}$, and 2^8 of them are trivial. With only two probes, positioned at XOR gates, and running the LP model, we get:

- 6016 transitions produce a leak of information about x_{i+1}, namely roughly $9,18\%$ of all transitions;

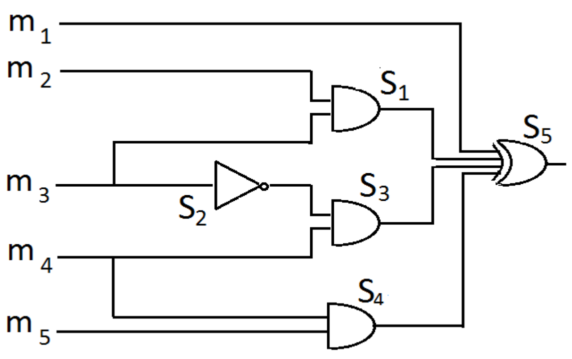

Figure 4.45: Atomic circuit of χ with 3 shares.

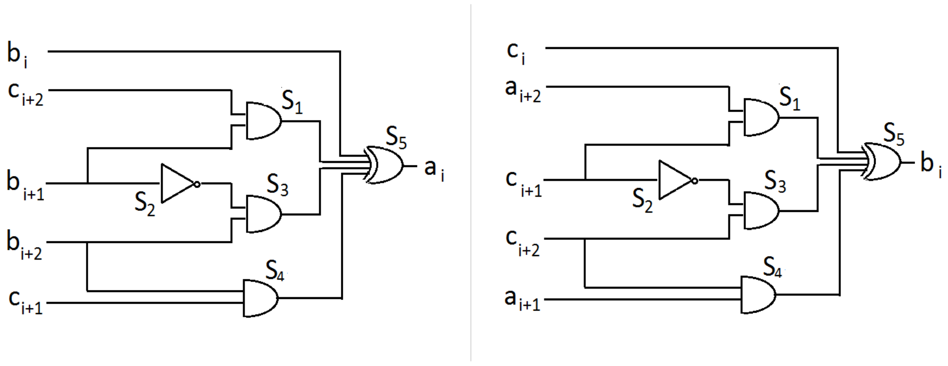

Figure 4.46: Two atomic circuits of χ with 3 shares: on the left the circuit that produces output a_i is depicted, on the right there is the circuit that produces output b_i.

- 6144 transitions produce a leak of information about x_{i+2}, roughly $9,38\%$ of all transitions;
- among them, 1024 transitions produce a leak of both x_{i+1} and x_{i+2}, $1,56\%$ of all transitions.

Example 4.2.3. *In Figure 4.47 there is an example of LP model application on two branches of χ with three shares. Probes are placed on the two XOR gates. Input literals for input variables are computed by literifiers in this way: $\mathcal{L}(b_i) = \{1\}$, $\mathcal{L}(c_{i+2}) = \{2\}$, $\mathcal{L}(b_{i+1}) = \{3\}$, $\mathcal{L}(b_{i+2}) = \{4\}$, $\mathcal{L}(c_{i+1}) = \{5\}$, $\mathcal{L}(c_i) = \{6\}$, $\mathcal{L}(a_{i+2}) = \{7\}$,*
$\mathcal{L}(a_{i+1}) = \{8\}$.
Inputs at time t_0 are $(b_i, c_{i+2}, b_{i+1}, b_{i+2}, c_{i+1}, c_i, a_{i+2}, a_{i+1}) = (0, 0, 1, 1, 1, 1, 1, 1)$, and at time t_1 are $(b_i, c_{i+2}, b_{i+1}, b_{i+2}, c_{i+1}, c_i, a_{i+2}, a_{i+1}) = (0, 1, 1, 0, 1, 0, 0, 0)$. Figure 4.47 depicts the situation with transients, computed with the glitch-counting algorithm, and literal transients for each gate, computed by literifiers. In the first function, at XOR gate there is a leakage of c_{i+2} and b_{i+2}, while in the second function at the same level there is a leakage of c_{i+2}, c_i, a_{i+2} and a_{i+1}: in this case, some information about $x_{i+2} = a_{i+2} \oplus b_{i+2} \oplus c_{i+2}$ can be recovered.

Now we explain how these leakages can be exploited to compute an attack using power consumption models. Referring to Figure 4.45, the atomic circuit has five inputs m_1, m_2, m_3, m_4, m_5 and four gates s_1, s_2, s_3, s_4. Time slots are:

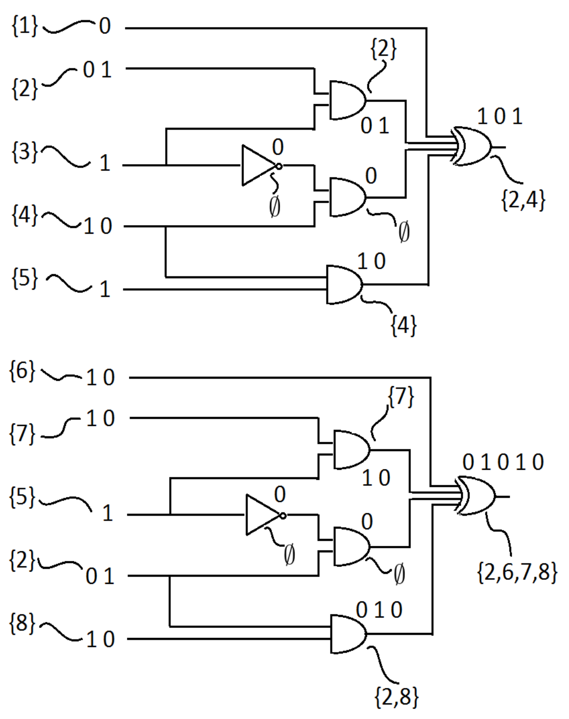

Figure 4.47: Example of LP model applied to two branches of χ with three shares: from the first circuit there can be some leakages of m_2 and m_4 at XOR level, and from the second circuit there can be some leakages of m_2, m_6, m_7 and m_8 at XOR level.

TS_1 $\rightarrow$ change of output bit of s_5 caused by the switch of m_1

TS_2 $\rightarrow$ change of output bit of s_1 caused by the switch of m_2

TS_3 $\rightarrow$ change of output bit of s_5 caused by the switch of the output of s_1 (in TS_2)

TS_4 $\rightarrow$ change of output bit of s_1 caused by the switch of m_3

TS_5 $\rightarrow$ change of output bit of s_5 caused by the switch of the output of s_1 (in TS_4)

TS_6 $\rightarrow$ change of output bit of s_2 caused by the switch of $m3$

TS_7 $\rightarrow$ change of output bit of s_3 caused by the switch of the output of s_2 (in TS_6)

TS_8 $\rightarrow$ change of output bit of s_5 caused by the switch of output of s_3 (in TS_7)

TS_9 $\rightarrow$ change of output bit of s_3 caused by the switch of m_4

TS_{10} $\rightarrow$ change of output bit of s_5 caused by the switch of output of s_3 (in TS_9)

TS_{11} $\rightarrow$ change of output bit of s_4 caused by the switch of m_4

TS_{12} $\rightarrow$ change of output bit of s_5 caused by the switch of output of s_4 (in TS_{11})

TS_{13} $\rightarrow$ change of output bit of s_4 caused by the switch of m_5

TS_{14} $\rightarrow$ change of output bit of s_5 caused by the switch of output of s_4 (in TS_{13})

Also in this case, there are two inputs variables (m_3 and m_4) that have effect on the last gate s_5 following two ways (for m_3 TS_4/TS_5 and TS_6/TS_7/TS_8, for m_4 TS_9/TS_{10} and TS_{11}/TS_{12}). Then, the set linked to the last gate is $I = \{1, 2, 3, 4, 5, 6, 7\}$, and the number of possible combinations is $7! = 5040$. However, the quotient set $\frac{Cb}{\sim_{Cb}}$ has only eight elements, and we choose as representatives of the classes:

- 1234567 for propagation sequence 1
- 1234576 for propagation sequence 2
- 1235467 for propagation sequence 3
- 1235476 for propagation sequence 4
- 1324567 for propagation sequence 5
- 1324576 for propagation sequence 6
- 1325467 for propagation sequence 7
- 1325476 for propagation sequence 8

We know that if we consider only one atomic circuit, there can be no leakage of some native variable, since each atomic circuit never processes all shares of x_i or x_{i+1} or x_{i+2}. Moreover, we can consider two atomic circuits together, but in this case it is clear that an attack implemented considering only one spatial point is impossible. If two circuits are considered, the number of propagation sequences grows to $8 * 8 = 64$.
Another time, without loss of generality, we can consider the two circuits in Figure 4.46, namely circuits that produce outputs a_i and b_i. In this case, there are two possibilities to recover a native value:

- native value x_{i+1} can be recovered in two cases:
  - if there is a leakage of b_{i+1} and c_{i+1} in the first circuit and a leakage of a_{i+1} in the second circuit;
  - if there is a leakage of b_{i+1} in the first circuit and a leakage of a_{i+1} and c_{i+1} in the second circuit;
- native value x_{i+2} can be recovered in two cases:
  - if there is a leakage of b_{i+2} and c_{i+2} in the first circuit and a leakage of a_{i+2} in the second circuit;
  - if there is a leakage of b_{i+2} in the first circuit and a leakage of a_{i+2} and c_{i+2} in the second circuit.

Instead, if we consider all three circuits together for an attack, the considered spatial points are at least three, and in this case it would be possible to recover a native value among x_i, x_{i+1} and x_{x+2} (since all the shares of the native values are processed by the three circuits).
In Table 4.8 the measures collected with **Power Consumption Model 4** are reported: they refer to power consumptions of a single circuit, with all possible inputs at time t_1 and all eight propagation sequences.
If we consider two circuits, there can be some leakages of x_{i+1} or x_{i+2}. Analysing all possible cases, no consumption can be exploited to recover any information about native values, also

Table 4.8: Consumptions of an atomic circuit of χ with three shares in accordance with Power Consumption Model 4, considering all propagation sequences.

m_1	m_2	m_3	m_4	m_5	consumption mean							
					PS 1234567	PS 1234576	PS 1235467	PS 1235476	PS 1324567	PS 1324576	PS 1325467	PS 1325476
0	0	0	0	0	3.5	3.5	2.5	2.5	3.5	3.5	2.5	2.5
0	0	0	0	1	3.5	4.5	2.5	3.5	3.5	4.5	2.5	3.5
0	0	0	1	0	4.5	3.5	4.5	3.5	4.5	3.5	4.5	3.5
0	0	0	1	1	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5
0	0	1	0	0	2.5	2.5	2.5	2.5	3.5	3.5	3.5	3.5
0	0	1	0	1	2.5	3.5	2.5	3.5	3.5	4.5	3.5	4.5
0	0	1	1	0	3.5	2.5	4.5	3.5	4.5	3.5	5.5	4.5
0	0	1	1	1	3.5	3.5	4.5	4.5	4.5	4.5	5.5	5.5
0	1	0	0	0	4.5	4.5	3.5	3.5	3.5	3.5	2.5	2.5
0	1	0	0	1	4.5	5.5	3.5	4.5	3.5	4.5	2.5	3.5
0	1	0	1	0	5.5	4.5	5.5	4.5	4.5	3.5	4.5	3.5
0	1	0	1	1	5.5	5.5	5.5	5.5	4.5	4.5	4.5	4.5
0	1	1	0	0	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5
0	1	1	0	1	3.5	4.5	3.5	4.5	3.5	4.5	3.5	4.5
0	1	1	1	0	4.5	3.5	5.5	4.5	4.5	3.5	5.5	4.5
0	1	1	1	1	4.5	4.5	5.5	5.5	4.5	4.5	5.5	5.5
1	0	0	0	0	3.5	3.5	2.5	2.5	3.5	3.5	2.5	2.5
1	0	0	0	1	3.5	4.5	2.5	3.5	3.5	4.5	2.5	3.5
1	0	0	1	0	4.5	3.5	4.5	3.5	4.5	3.5	4.5	3.5
1	0	0	1	1	4.5	4.5	4.5	4.5	4.5	4.5	4.5	4.5
1	0	1	0	0	2.5	2.5	2.5	2.5	3.5	3.5	3.5	3.5
1	0	1	0	1	2.5	3.5	2.5	3.5	3.5	4.5	3.5	4.5
1	0	1	1	0	3.5	2.5	4.5	3.5	4.5	3.5	5.5	4.5
1	0	1	1	1	3.5	3.5	4.5	4.5	4.5	4.5	5.5	5.5
1	1	0	0	0	4.5	4.5	3.5	3.5	3.5	3.5	2.5	2.5
1	1	0	0	1	4.5	5.5	3.5	4.5	3.5	4.5	2.5	3.5
1	1	0	1	0	5.5	4.5	5.5	4.5	4.5	3.5	4.5	3.5
1	1	0	1	1	5.5	5.5	5.5	5.5	4.5	4.5	4.5	4.5
1	1	1	0	0	3.5	3.5	3.5	3.5	3.5	3.5	3.5	3.5
1	1	1	0	1	3.5	4.5	3.5	4.5	3.5	4.5	3.5	4.5
1	1	1	1	0	4.5	3.5	5.5	4.5	4.5	3.5	5.5	4.5
1	1	1	1	1	4.5	4.5	5.5	5.5	4.5	4.5	5.5	5.5

with all propagation sequences. Indeed, the mean of consumptions in all cases is always 8, both if the native value (x_{i+1} or x_{i+2}) is 0 or 1.
Also if we consider 3 circuits together, and analysing all possible cases, no consumption can be exploited to recover any information about native values, also with all propagation sequences. Indeed, the mean of consumptions in all cases is always 12, whether the native value is 0 or 1.
With all three circuits, we can also think to relieve from consumptions some information about the XOR of the output bits of all three circuits (that is a native value). Even with this approach, it is not possible to distinguish if the native value is 0 or 1, with no propagation sequence or particular input. Once again, the mean of consumptions in all cases is always 12, whether the native value is 0 and if it is 1.
With Power Consumption Model 4, all possible leakages are analysed, and we have verified that there not can be any leakages exploitable to recover some information about native values.
Now we study the consumptions recovered with **Power Consumption Model 2**, knowing that the time slots for an atomic circuit of χ with 3 shares are 14 and considering all inputs at time t_1. With this analysis, we can notice that:

- In the table of consumptions of the circuit with propagation sequence 1234567, columns of time slot 4 and time slot 5 are proportional to m_2, columns of time slot 9 and time slot 10 are inversely proportional to m_3 and columns of time slot 13 and time slot 14 are proportional to m_4.
- In the table of consumptions of the circuit with propagation sequence 1234576, columns of time slot 4 and time slot 5 are proportional to m_2, columns of time slot 9 and time slot 10 are inversely proportional to m_3 and columns of time slot 11 and time slot 12 are proportional to m_5.

- In the table of consumptions of the circuit with propagation sequence 1235467, columns of time slot 4 and time slot 5 are proportional to m_2, columns of time slot 7 and time slot 8 are proportional to m_4 and columns of time slot 13 and time slot 14 are proportional to m_4.
- In the table of consumptions of the circuit with propagation sequence 1235476, columns of time slot 4 and time slot 5 are proportional to m_2, columns of time slot 7 and time slot 8 are proportional to m_4 and columns of time slot 11 and time slot 12 are proportional to m_5.
- In the table of consumptions of the circuit with propagation sequence 1324567, columns of time slot 2 and time slot 3 are proportional to m_3, columns of time slot 9 and time slot 10 are inversely proportional to m_3 and columns of time slot 13 and time slot 14 are proportional to m_4.
- In the table of consumptions of the circuit with propagation sequence 1324576, columns of time slot 2 and time slot 3 are proportional to m_3, columns of time slot 9 and time slot 10 are inversely proportional to m_3 and columns of time slot 11 and time slot 12 are proportional to m_5 (Table 4.9).
- In the table of consumptions of the circuit with propagation sequence 1325467, columns of time slot 2 and time slot 3 are proportional to m_3, columns of time slot 7 and time slot 8 are proportional to m_4 and columns of time slot 13 and time slot 14 are proportional to m_4.
- In the table of consumptions of the circuit with propagation sequence 1325476, columns of time slot 2 and time slot 3 are proportional to m_3, columns of time slot 7 and time slot 8 are proportional to m_4 and columns of time slot 11 and time slot 12 are proportional to m_5.

We can try to exploit information about a native variable in more ways, from two or three circuits, with different propagation sequences. We have decided to explain only one case, where we try to recover x_{i+1}, considering two circuits (first one producing a_i and second one producing b_i), both with propagation sequence 1324576. From the first circuit, we can recover information about m_3 (i.e. b_{i+1}) and m_5 (i.e. c_{i+1}), while from the second one we can recover information about m_5 (i.e. a_{i+1}). All of these consumptions are recovered at level of XOR gate, in both the circuits, and then in total two points have to be monitored, namely the XOR gates.
In the following steps, we explain how an attacker can exploit this information to recover five bits of the key with a CPA attack. At each clock cycle there is a leakage of x_{i+1}, and after the five χ operations all the five bits of the row of the key can be recovered.

- At the beginning, m traces are collected, for each possible value p_i that the row of KECCAK can assume, with $0 \leq i \leq 31$, and recovering consumptions in each time slot, and in all of the five rounds. All these consumptions are collected in a matrix **T**, that has $32m$ rows (number of traces) and $14 * 5 * 2$ columns, since there are 14 time slots, five rounds (as the number of bits of the KECCAK row) and two circuits that we are considering.
- In each round, an attacker is interested only on consumptions in time slot 3 and 12 for the first circuit, and consumption in time slot 12 in the second circuit. For each row of **T**

and for each round, these three consumptions are XORed, computing a matrix $\tilde{\mathbf{T}}$ with $32m$ rows and five columns, a column for each round.

- Since an attacker does not know inputs at time t_0, she computes the mean of all elements in each column inherent to the same input p_i in $\tilde{\mathbf{T}}$ computing a matrix $\bar{\mathbf{T}}$ with 32 rows (one for each p_i) and five columns. Finally, a vector t is constructed, such that the first five elements are the first row of $\bar{\mathbf{T}}$, the second five elements are the second ow of $\bar{\mathbf{T}}$, and so on.
- A matrix of hypothesis **H** is created, with the hypothetical consumptions; this matrix has 32 rows, that refer to input p_i at time t_1, and $32 * 5$ columns, that correspond to 32 hypothetical keys and 5 bits for each one. Consumptions are 0.375 if the corresponding bit is 0, 0.5 if it is 1.
- Correlation between t and **H** is computed, to reveal the key.

The attack that allows to recover one bit of the key at each round, in tridimensional notation, is a $(1, 2, 3)$ attack. Indeed:

- $sm = 1$, because to implement this attack an attacker uses the first statistical moment (but it works also using second and third statistical moments);
- $sp = 2$, because the useful consumptions are recovered in two points, namely at level of the XOR gate of the first circuit, and at level of the XOR gate of the second circuit;
- $ti = 3$, because the attacker recover the consumptions in three time slots, two time slots in one of the two circuits, and one time slot in the other one.

The attack explained before is an application of this $(1, 2, 3)$ attack on all five rounds.

4.2.8 Conclusions

In this section, two main arguments have been treated as an attempt to model glitches at design-time: Hazard Algebra and models that come from it and the order of an attack.
In subsection 4.2.4 the unrealistic worst-case scenario of propagation of glitches is overcome, thanks to the new approach with propagation sequences. The worst-case is defined on an “ideal circuit”, namely a circuit in which we are interested only in its gates and relative connections. Differently, the signal propagation described with the propagation sequences is defined on circuits that are not ideal, but of which are considered also the intrinsic properties, as the length of wires. Indeed, on the ideal circuit different propagation sequences are defined, such that they respect all possible different propagation of signal (and propagation of glitches) that can happen in a real circuit with the same gates and connections of the ideal one. In this chapter, we have given a mathematical description of propagation sequences as quotient of the set of combinations, and we defined also a new algorithm to count the glitches in the case of a particular propagation sequence, that operates similarly to the glitch-counting algorithm defined by Brzozowski and Ésik in [40]. In particular, the main contribution that propagation sequences give is the possibility to better quantify the amount of critical leakages in a circuit.

Table 4.9: Consumptions of an atomic circuit of χ with three shares, with propagation sequence 1324576 and in accordance with Power Consumption Model 2.

m_1	m_2	m_3	m_4	m_5	consumption in each time slot													
					T_1	TS_2	TS_3	TS_4	TS_5	TS_6	TS_7	TS_8	TS_9	TS_{10}	TS_{11}	TS_{12}	TS_{13}	TS_{14}
0	0	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
0	0	0	0	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
0	0	0	1	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
0	0	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
0	0	1	0	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
0	0	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
0	0	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
0	0	1	1	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
0	1	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	.5	0.5	0.25	0.25	0	0
0	1	0	0	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
0	1	0	1	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
0	1	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
0	1	1	0	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
0	1	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
0	1	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
0	1	1	1	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
1	0	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
1	0	0	0	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
1	0	0	1	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
1	0	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
1	0	1	0	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
1	0	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
1	0	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
1	0	1	1	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
1	1	0	0	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
1	1	0	0	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
1	1	0	1	0	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0	0
1	1	0	1	1	0.5	0.25	0.25	0	0	0.5	0.25	0.25	0.5	0.5	0.25	0.25	0.5	0.5
1	1	1	0	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
1	1	1	0	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5
1	1	1	1	0	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0	0
1	1	1	1	1	0.5	0.25	0.25	0.5	0.5	0.5	0.25	0.25	0	0	0.25	0.25	0.5	0.5

Subsection 4.2.5 is principally based on the redefinition of the order of an attack. The main effort was to give our interpretation of order respecting all the previous ones, namely the definitions based on the order of the statistical moment used, or the one based on the number of probes placed in the circuit to analyse the signal. The goal was achieved introducing the order of an attack as tridimensional point (sm, sp, ti), where sm is the order of the statistical moment used to implement the attack, sp is the number of points in the circuit that are analysed and ti is the number of time instances considered. The last two variables depend on the measuring instrument, and in particular sp depends on the spatial resolution of the instrument and ti depends on the time resolution. In this way, with the second variable of the triplet the attention is posed on the previous definition based on the number of probes placed on the wires of the circuit, while the other two respect the definition of order based on the statistical moment, with ti denoting a $v-$variate attack, where v are different time instances in the same clock cycle.
In subsection 4.2.7, these arguments have been validated with some explicative examples on the nonlinear part of KECCAK, in conjunction with the application of the power consumption models defined in subsection 4.2.6, that are introduced with the aim to create some models for the collection of consumptions during the execution of the cryptographic algorithms.
Some possible future developments about propagation sequences can be some insights of the topic. For example an interesting argument can be the deduction of an equation, applicable to any feedback-free circuit, with which calculate the number of propagation sequences in it, without recover each time all combinations and counting the number of propagation sequences as the cardinality of the quotient set. Another example can be a study about ”the quality” of the propagation sequences in a circuit, namely if there are some propagation sequences that are more possible or better detectable in it rather than some others.
Instead, the order of an attack has been a widespread topic in all the literature until now, and then we suppose that this is an argument that will be treated again in different forms. We

foresee the appearance of some further specifications about our interpretation of the order as a tridimensional point, or some other new and different definitions. However, we think that at the moment our characterization of the order is the most exhaustive ever appeared, being able to conjunct all the previous definitions in only one. An argument that can be developed starting from our definition is a study of the differences between an attack implemented placing the ideal probes in a unique clock cycle, or taking information in more clock cycles. Moreover, another argument that could be better study is the conjunction between ideal probes and real instruments, with different bandwidth.
We can also propose some developments about the power consumption models, since they do not hold all possibilities of consumptions collection. For example, in our four models there is none that represents an instrument with a high spatial resolution but a low time resolution, i.e. an instrument that reads only a sum of all consumptions, but that can relieve consumptions at level of all gates.

Chapter 5

Summary and Conclusion

In this deliverable, we presented the cryptographic primitives proposed by HECTOR partners to address the need for Authenticated Encryption. We elaborated about the reasons that make these algorithms more efficient, more secure, and simpler to use compared with the primitives usually used so far.
All the proposed algorithms, and in particular their latest variations fulfil the criteria listed in HECTOR Deliverable D1.1. Namely:

- the algorithms support Authenticated Encryption with Associated Data, which means that a portion of the data is protected for both integrity and confidentiality, and another portion for integrity only;
- they are Single Pass, which means that a single elaboration of the message and single primitive are sufficient;
- they can perform Online processing, which means that the size of the message does not have to be known in advance.

Besides fulfilling these requirements, the underlying sponge construction used in the proposed algorithms provides interesting benefits when compared to commonly used solutions. The same cryptographic primitive can be used for several purposes, such as hashing, reseedable pseudo-random generation, key derivation, encryption, MAC computation and authenticated encryption. Another benefit is that these algorithms no longer require to securely combining different primitives to accomplish a task, reducing the risk of vulnerabilities.

As an illustration, we explained how a poor combination of encryption and MAC catastrophically undermined the security of some largely used communication protocols. A final benefit is that it allows to focus the efforts for the protection against side-channel attacks onto a single primitive, while the sponge construction are also easier to protect, making it easier to derive such countermeasures.
Side-channel attacks constitute a very powerful class of attacks against cryptographic devices that exploit some kind of physical information leaked by the device itself (e.g. timing information, power consumption, or electromagnetic emanation). Nowadays side-channel attacks pose an important practical threat against devices in real-life use cases.

We spent the second part of this Deliverable analysing those attacks and proposing solutions in order to make devices more robust against them. We presented an overview of the

state of the art of those attacks and introduced some new techniques, which could undermine the security of devices if not properly taken into account. We focused on system-level countermeasures able to counter the known attacks.
Besides studying attacks and proposing countermeasures, we tackled the difficulties associated with the implementation of devices that need to be robust against side-channel attacks, again aiming at improving the overall efficiency. Side-channel protections affect costs and performances. Countermeasures usually require to increase either the area or the latency of the implemented cryptographic block (or both). Often more important, protections also significantly impact the development and verification schedule of designs.

Relying exclusively on silicon samples to verify the effectiveness of countermeasures is simply not acceptable, because of the late feedback and the costs involved in a silicon iteration. In order to overcome this issue, we proposed a hardware design methodology allowing to take into account and assess the side-channel properties of the resulting devices. Such methodology is based on functional languages, which can close the gap between the high-level specifications and the actual hardware implementation.
We investigated the methodology having in mind the requirements from HECTOR Deliverable D1.1:

- provide reasonably accurate early feedback;
- allow quick fact-based comparison of different solutions that can be implemented against a specific side-channel attack;
- help the refinement of attack and evaluation techniques, in order to prototype and ease the final on-silicon evaluations;
- provide detailed view of the side-channel behaviour of internal blocks, in order to easy the analysis and the fixing of vulnerabilities.

In view of supporting the design-time evaluation of side-channel properties, we also proposed a model to tackle one of the most critical sources of leakage in hardware implementations: glitches on combinational logic.

Design-time side-channel evaluation methodologies cannot and are not aiming at replacing on-silicon evaluations, but rather complementing them. A final assessment on the target physical device is still needed. The goal here is to guide the implementation of the secure solution, by supporting the designer in selecting the right trade-off between costs and protection, leading to the right practical protection in the most efficient and cost-optimized way.

Chapter 6

List of Abbreviations

AD	Associated Data
AE	Authenticated Encryption
AES	Advanced Encryption Standard
API	Application Programming Interface
ASIC	Application Specific Integrated Circuit
CCM	Counter with CBC-MAC
DPA	Differential Power Analysis
EC	European Commission
ECC	Elliptic Curve Cryptography
ECDSA	Elliptic Curve Digital Signature Algorithm
EMFI	Electromagnetic Fault Injection
FPGA	Field Programmable Logic Array
FSM	Finite State Machine
GCM	Galois/Counter Mode
GE	Gate Equivalent
HDL	Hardware Description Language
IP	Intellectual Property
IV	Initialization Vector
LUT	Look-Up Table
MAC	Message Authentication Code
NIST	National Institute of Standards and Technology
PUF	Physical Unclonable Function
RTL	Register-Transfer Level
SEI	Squared Euclidian Imbalance
SFA	Statistical Fault Attack
SHA	Secure Hash Algorithm
SPA	Simple Power Analysis
SUV	Secret and Unique Value
TBD	to be determined
TI	Threshold Implementation
TRNG	True Random Number Generator

Bibliography

[1] Diac - directions in authenticated ciphers. http://hyperelliptic.org/DIAC/.

[2] Farzaneh Abed, Scott R. Fluhrer, Christian Forler, Eik List, Stefan Lucks, David A. McGrew, and Jakob Wenzel. Pipelineable on-line encryption. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 205–223. Springer, 2014.

[3] Yasemin Acar, Michael Backes, Sven Bugiel, Sascha Fahl, Patrick Drew McDaniel, and Matthew Smith. SoK: Lessons Learned from Android Security Research for Appified Software Platforms. In *IEEE Symposium on Security and Privacy – S&P 2016*, pages 433–451, 2016.

[4] Megha Agrawal, Donghoon Chang, and Somitra Sanadhya. sp-AELM: Sponge based authenticated encryption scheme for memory constrained devices. In Ernest Foo and Douglas Stebila, editors, *Information Security and Privacy – ACISP 2015*, volume 9144 of *LNCS*, pages 451–468. Springer, 2015.

[5] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013.

[6] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Qingju Wang, and Kan Yasuda. PRIMATEs v1.02, Submission to the CAESAR Competition. http://primates.ae/wp-content/uploads/primatesv1.02.pdf.

[7] Elena Andreeva, Andrey Bogdanov, Atul Luykx, Bart Mennink, Elmar Tischhauser, and Kan Yasuda. Parallelizable and authenticated online ciphers. In Kazue Sako and Palash Sarkar, editors, *Advances in Cryptology - ASIACRYPT 2013 - 19th International Conference on the Theory and Application of Cryptology and Information Security, Bengaluru, India, December 1-5, 2013, Proceedings, Part I*, volume 8269 of *Lecture Notes in Computer Science*, pages 424–443. Springer, 2013.

[8] Elena Andreeva, Joan Daemen, Bart Mennink, and Gilles Van Assche. Security of keyed sponge constructions using a modular proof approach. In *FSE 2015*, pages 364–384, 2015.

[9] Ralph Ankele and Robin Ankele. Software benchmarking of the 2$^{\text{nd}}$ round CAESAR candidates. Cryptology ePrint Archive, Report 2016/740, 2016.

[10] Seyed Hosein Attarzadeh Niaki and Ingo Sander. Co-simulation of embedded systems in a heterogeneous MoC-based modeling framework. In *2011 6th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 238–247. IEEE, June 2011.

[11] Adam J. Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M. Smith. Practicality of Accelerometer Side Channels on Smartphones. In *Annual Computer Security Applications Conference – ACSAC 2012*, pages 41–50, 2012.

[12] Christiaan Baaij and Jan Kuper. Using rewriting to synthesize functional languages to digital circuits. In Jay McCarthy, editor, *Trends in functional programming*, volume 8322 of *Lecture notes in computer science*, pages 17–33, Berlin, Germany, 2014. Springer.

[13] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. DPA, Bitslicing and Masking at 1 GHz. In *Cryptographic Hardware and Embedded Systems – CHES 2015*, pages 599–619, 2015.

[14] Elaine B Barker, William C Barker, William E Burr, W Timothy Polk, and Miles E Smid. Sp 800-57. recommendation for key management, part 1: General (revised). 2007.

[15] AliGalip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. Sleuth: Automated verification of software power analysis countermeasures. In Guido Bertoni and Jean-Sbastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 293–310. Springer Berlin Heidelberg, 2013.

[16] Pierre Belgarric, Pierre-Alain Fouque, Gilles Macario-Rat, and Mehdi Tibouchi. Side-Channel Analysis of Weierstrass and Koblitz Curve ECDSA on Android Smartphones. In *Topics in Cryptology – CT-RSA 2016*, pages 236–252, 2016.

[17] Mihir Bellare, Anand Desai, E. Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th Annual Symposium on Foundations of Computer Science, FOCS '97, Miami Beach, Florida, USA, October 19-22, 1997*, pages 394–403. IEEE Computer Society, 1997.

[18] Mihir Bellare, Joe Kilian, and Phillip Rogaway. The security of the cipher block chaining message authentication code. *J. Comput. Syst. Sci.*, 61(3):362–399, 2000.

[19] Mihir Bellare, Phillip Rogaway, and David Wagner. The EAX mode of operation. In Bimal K. Roy and Willi Meier, editors, *Fast Software Encryption, 11th International Workshop, FSE 2004, Delhi, India, February 5-7, 2004, Revised Papers*, volume 3017 of *Lecture Notes in Computer Science*, pages 389–407. Springer, 2004.

[20] Daniel J. Bernstein and Tanja Lange (editors). eBACS: ECRYPT Benchmarking of Cryptographic Systems. `https://bench.cr.yp.to`. (accessed 15 September 2016).

[21] Guido Bertoni, Joan Daemen, Nicolas Debande, Than-Ha Le, Michaël Peeters, and Gilles Van Assche. Power Analysis of Hardware Implementations Protected with Secret Sharing. In *45th Annual IEEE/ACM International Symposium on Microarchitecture Workshops (MICROW)*, pages 9–16. IEEE Computer Society, 2012.

[22] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Keyak hardware implementations – git repository. https://github.com/guidobertoni/caesar_gmu_vhdl.

[23] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Sponge functions. In *ECRYPT hash workshop*, volume 2007. Citeseer, 2007.

[24] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Building power analysis resistant implementations of KECCAK. *Second SHA-3 condidate conference*, August 2010.

[25] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Cryptographic sponge functions (Version 0.1). http://sponge.noekeon.org/, 2011.

[26] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. The Keccak SHA-3 submission. http://keccak.noekeon.org/, 2011.

[27] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. Permutation-based encryption, authentication and authenticated encryption. In *DIAC 2012*, pages 159–170. na, 2012.

[28] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Caesar submission: KETJE V2. http://ketje.noekeon.org/, September 2016.

[29] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. Caesar submission: KEYAK V2. http://keyak.noekeon.org/, September 2016.

[30] Guido Bertoni, Joan Daemen, Michal Peeters, and Gilles Van Assche. Duplexing the sponge: Single-pass authenticated encryption and other applications. In *In Selected Areas in Cryptography*, pages 320–337.

[31] Guido Bertoni and Marco Martinoli. A Metodology for the Characterization of Leakages in Combinatorial Logic. 2016.

[32] Sebastian Biedermann, Stefan Katzenbeisser, and Jakub Szefer. Hard Drive Side-Channel Attacks Using Smartphone Magnetic Field Sensors. In *Financial Cryptography – FC 2015*, pages 489–496, 2015.

[33] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. Efficient and First-Order DPA Resistant Implementations of Keccak. In *CARDIS 2014*, LNCS. Springer, 2014.

[34] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. A More Efficient AES Threshold Implementation. In *AFRICACRYPT 2014*, volume 8469 of *LNCS*. Springer, 2014.

[35] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Higher-Order Threshold Implementations. In *ASIACRYPT 2014*, volume 8874 of *LNCS*. Springer, 2014.

[36] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Trade-Offs for Threshold Implementations Illustrated on AES. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 34(7), July 2015.

[37] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, and Georg Stütz. Threshold Implementations of All 3x3 and 4x4 S-Boxes. In *CHES 2012*, volume 7428 of *LNCS*. Springer, 2012.

[38] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. Lava: Hardware design in haskell. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming*, ICFP '98, pages 174–184, New York, NY, USA, 1998. ACM.

[39] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *CHES 2004*, pages 16–29, 2004.

[40] Janusz Brzozowski and Zoltán Ésik. Hazard algebras. *Formal Methods in System Design*, 23(3):223–256, 2003.

[41] Liang Cai and Hao Chen. TouchLogger: Inferring Keystrokes on Touch Screen from Smartphone Motion. In *USENIX Workshop on Hot Topics in Security – HotSec*, 2011.

[42] Suresh Chari, Josyula R. Rao, and Pankaj Rohatgi. Template attacks. In *CHES 2002*, pages 13–28, 2002.

[43] Thomas De Cnudde, Begül Bilgin, Oscar Reparaz, Ventzislav Nikov, and Svetla Nikova. Higher-Order Threshold Implementation of the AES S-Box. In *CARDIS 2015*, 2015.

[44] Thomas De Cnudde, Oscar Reparaz, Begül Bilgin, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. Masking AES with d+1 Shares in Hardware. In *CHES 2016*, 2016.

[45] Quynh H Dang. Secure hash standard. *National Institute of Standards and Technology, Gaithersburg, MD, Tech. Rep. August*, 2015.

[46] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, and Florian Mendel. On the security of fresh re-keying to counteract side-channel and fault attacks. In *CARDIS 2014*, pages 233–244, 2014.

[47] Christoph Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, and Thomas Unterluggauer. ISAP – Authenticated Encryption Inherently Secure Against Passive Side-Channel Attacks. https://eprint.iacr.org/2016/952. Accessed: 2017-01-20.

[48] Christoph Dobraunig, François Koeune, Stefan Mangard, Florian Mendel, and François-Xavier Standaert. Towards fresh and hybrid re-keying schemes with beyond birthday security. In *CARDIS 2015*, volume 9514 of *LNCS*, pages 225–241. Springer, 2015.

[49] Christoph Dobraunig, Florian Mendel, Martin Schläffer, and Maria Eichlseder. Ascon website. http://ascon.iaik.tugraz.at/index.html, 12 2013. Accessed: 2016/12/10.

[50] Christoph Dobraunig, Florian Mendel, Martin Schläffer, and Maria Eichlseder. Ascon v1.2 – submission to round 3 of the CAESAR competition. http://ascon.iaik.tugraz.at/files/asconv12.pdf, 2016.

[51] Christoph Dobraunig, Florian Mendel, Martin Schläffer, Hannes Gross, and Maria Eichlseder. Ascon hardware implementations – git repository. https://github.com/IAIK/ascon_hardware, 12 2014. Accessed: 2016/12/10.

[52] Thai Duong and Juliano Rizzo. Here come the ⊕ ninjas. https://bug665814.bugzilla.mozilla.org/attachment.cgi?id=540839, 2011. Unpublished manuscript.

[53] Stefan Dziembowski and Krzysztof Pietrzak. Leakage-resilient cryptography. In *FOCS 2008*, pages 293–302, 2008.

[54] Hassan Eldib, Chao Wang, and Patrick Schaumont. Formal verification of software countermeasures against side-channel attacks. *ACM Trans. Softw. Eng. Methodol.*, 24(2):11:1–11:24, December 2014.

[55] Levent Erkök, Magnus Carlsson, and Adam Wick. Hardware/software co-verification of cryptographic algorithms using Cryptol. In *Formal Methods in Computer Aided Design, FMCAD'09, Austin, TX, USA*, pages 188–191. IEEE, November 2009.

[56] Sebastian Faust, Krzysztof Pietrzak, and Joachim Schipper. Practical leakage-resilient symmetric cryptography. In *CHES 2012*, pages 213–232, 2012.

[57] Niels Ferguson. Collision attacks on ocb. http://www.cs.ucdavis.edu/~rogaway/ocb/fe02.pdf, 2002. Unpublished manuscript.

[58] Ewan Fleischmann, Christian Forler, and Stefan Lucks. Mcoe: A family of almost foolproof on-line authenticated encryption schemes. In Anne Canteaut, editor, *Fast Software Encryption - 19th International Workshop, FSE 2012, Washington, DC, USA, March 19-21, 2012. Revised Selected Papers*, volume 7549 of *Lecture Notes in Computer Science*, pages 196–215. Springer, 2012.

[59] Thomas Fuhr, Éliane Jaulmes, Victor Lomné, and Adrian Thillard. Fault attacks on AES with faulty ciphertexts only. In *Fault Diagnosis and Tolerance in Cryptography – FDTC 2013*, pages 108–118, 2013.

[60] Kris Gaj and ATHENa Team. ATHENa: Automated Tool for Hardware Evaluation, 2016. https://cryptography.gmu.edu/athena/.

[61] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A Survey of Microarchitectural Timing Attacks and Countermeasures on Contemporary Hardware. *IACR Cryptology ePrint Archive*, 2016:613, 2016.

[62] Catherine H. Gebotys, Simon Ho, and C. C. Tiu. EM Analysis of Rijndael and ECC on a Wireless Java-Based PDA. In *Cryptographic Hardware and Embedded Systems – CHES 2005*, pages 250–264, 2005.

[63] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA Key Extraction from Mobile Devices via Nonintrusive Physical Side Channels. In *Conference on Computer and Communications Security – CCS 2016*, pages 1626–1638, 2016.

[64] Andy Gill. Declarative FPGA circuit synthesis using Kansas Lava. In *The International Conference on Engineering of Reconfigurable Systems and Algorithms*, July 2011.

[65] Virgil D. Gligor and Pompiliu Donescu. Fast encryption and authentication: XCBC encryption and XECB authentication modes. In Mitsuru Matsui, editor, *Fast Software Encryption, 8th International Workshop, FSE 2001 Yokohama, Japan, April 2-4, 2001, Revised Papers*, volume 2355 of *Lecture Notes in Computer Science*, pages 92–108. Springer, 2001.

[66] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *J. ACM*, 33(4):792–807, 1986.

[67] Gabriel Goller and Georg Sigl. Side Channel Attacks on Smartphones and Embedded Devices Using Standard Radio Equipment. In *Constructive Side-Channel Analysis and Secure Design – COSADE 2015*, pages 255–270, 2015.

[68] Gilbert Goodwill, Benjamin Jun, Josh Jaffe, and Pankaj Rohatgi. A Testing Methodology for Side-Channel Resistance Validation. In *NIST Non-Invasive Attack Testing Workshop*, 2011.

[69] Louis Goubin and Jacques Patarin. DES and Differential Power Analysis The Duplication Method. In *Cryptographic Hardware and Embedded Systems*, volume 1717 of *LNCS*. Springer, 1999.

[70] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhofer. Suit up! - Made-to-measure hardware implementations of ASCON. In *Digital System Design – DSD 2015*, pages 645–652. IEEE Computer Society, 2015.

[71] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache Template Attacks: Automating Attacks on Inclusive Last-Level Caches. In *USENIX Security Symposium 2015*, pages 897–912, 2015.

[72] Shay Gueron and Yehuda Lindell. GCM-SIV: full nonce misuse-resistant authenticated encryption at under one cycle per byte. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-6, 2015*, pages 109–119. ACM, 2015.

[73] Frank K Gürkaynak, Kris Gaj, Beat Muheim, Ekawat Homsirikamol, Christoph Keller, Marcin Rogawski, Hubert Kaeslin, and Jens-Peter Kaps. Lessons learned from designing a 65 nm asic for third round sha-3 candidates. 2012.

[74] Viet Tung Hoang, Ted Krovetz, and Phillip Rogaway. Robust authenticated-encryption AEZ and the problem that it solves. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 15–44. Springer, 2015.

[75] Viet Tung Hoang, Reza Reyhanitabar, Phillip Rogaway, and Damian Vizár. Online authenticated-encryption and its nonce-reuse misuse-resistance. In Rosario Gennaro and Matthew Robshaw, editors, *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, volume 9215 of *Lecture Notes in Computer Science*, pages 493–517. Springer, 2015.

[76] Avesta Hojjati, Anku Adhikari, Katarina Struckmann, Edward Chou, Thi Ngoc Tho Nguyen, Kushagra Madan, Marianne S. Winslett, Carl A. Gunter, and William P. King. Leave Your Phone at the Door: Side Channels that Reveal Factory Floor Secrets. In *Conference on Computer and Communications Security – CCS 2016*, pages 883–894, 2016.

[77] Michael Hutter and Jörn-Marc Schmidt. The Temperature Side Channel and Heating Fault Attacks. In *Smart Card Research and Advanced Applications – CARDIS 2013*, pages 219–235, 2013.

[78] Yuval Ishai, Amit Sahai, and David Wagner. Private Circuits: Securing Hardware against Probing Attacks. In *Advances in Cryptology - CRYPTO 2003*, volume 2729 of *LNCS*. Springer, 2003.

[79] Suman Jana and Vitaly Shmatikov. Memento: Learning Secrets from Process Footprints. In *IEEE Symposium on Security and Privacy – S&P 2012*, pages 143–157, 2012.

[80] Charanjit S. Jutla. Encryption modes with almost free message integrity. In Birgit Pfitzmann, editor, *Advances in Cryptology - EUROCRYPT 2001, International Conference on the Theory and Application of Cryptographic Techniques, Innsbruck, Austria, May 6-10, 2001, Proceeding*, volume 2045 of *Lecture Notes in Computer Science*, pages 529–544. Springer, 2001.

[81] Charanjit S. Jutla. Encryption modes with almost free message integrity. *J. Cryptology*, 21(4):547–578, 2008.

[82] Paul C. Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In *Advances in Cryptology – CRYPTO 1996*, pages 104–113, 1996.

[83] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology – CRYPTO 1999*, pages 388–397, 1999.

[84] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is ssl?). In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 310–331. Springer, 2001.

[85] Xun Li, Vineeth Kashyap, Jason K. Oberg, Mohit Tiwari, Vasanth Ram Rajarathinam, Ryan Kastner, Timothy Sherwood, Ben Hardekopf, and Frederic T. Chong. Sapper: A language for hardware-level security policy enforcement. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 97–112, New York, NY, USA, 2014. ACM.

[86] Xun Li, Mohit Tiwari, Jason K. Oberg, Vineeth Kashyap, Frederic T. Chong, Timothy Sherwood, and Ben Hardekopf. Caisson: A hardware description language for secure information flow. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pages 109–120, New York, NY, USA, 2011. ACM.

[87] Zhonghai Lu, Ingo Sander, and Axel Jantsch. A case study of hardware and software synthesis in ForSyDe. In *Proceedings of the 15th International Symposium on System Synthesis*, pages 86–91, Kyoto, Japan, October 2002.

[88] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks - Revealing the Secrets of Smart Cards*. Springer, 2007.

[89] Philip Marquardt, Arunabh Verma, Henry Carter, and Patrick Traynor. (sp)iPhone: Decoding Vibrations From Nearby Keyboards Using Mobile Phone Accelerometers. In *Conference on Computer and Communications Security – CCS 2011*, pages 551–562, 2011.

[90] Marcel Medwed, Christophe Petit, Francesco Regazzoni, Mathieu Renauld, and François-Xavier Standaert. Fresh re-keying II: securing multiple parties against side-channel and fault attacks. In *CARDIS 2011*, pages 115–132, 2011.

[91] Marcel Medwed, François-Xavier Standaert, Johann Großschädl, and Francesco Regazzoni. Fresh re-keying: Security against side-channel and fault attacks for low-cost devices. In *AFRICACRYPT 2010*, pages 279–296, 2010.

[92] Maryam Mehrnezhad, Ehsan Toreini, Siamak Fayyaz Shahandashti, and Feng Hao. Stealing PINs via Mobile Sensors: Actual Risk versus User Perception. *CoRR*, abs/1605.05549, 2016.

[93] Yan Michalevsky, Aaron Schulman, Gunaa Arumugam Veerapandian, Dan Boneh, and Gabi Nakibly. PowerSpy: Location Tracking Using Mobile Device Power Analysis. In *USENIX Security Symposium 2015*, pages 785–800, 2015.

[94] Amir Moradi and Oliver Mischke. How far should theory be from practice? - Evaluation of a countermeasure. *CHES 2012. LNCS*, vol. 7428:92–106, 2012.

[95] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. Pushing the Limits: A Very Compact and a Threshold Implementation of AES. In *Eurocrypt 2011*, EUROCRYPT'11. Springer-Verlag, 2011.

[96] Tilo Müller and Michael Spreitzenbarth. FROST - Forensic Recovery of Scrambled Telephones. In *Applied Cryptography and Network Security – ACNS 2013*, pages 373–388, 2013.

[97] Yuto Nakano, Youssef Souissi, Robert Nguyen, Laurent Sauvage, Jean-Luc Danger, Sylvain Guilley, Shinsaku Kiyomoto, and Yutaka Miyake. A Pre-processing Composition for Secret Key Recovery on Android Smartphone. In *Information Security Theory and Practice – WISTP 2014*, pages 76–91, 2014.

[98] NewAE Technology Inc. Fault Injection Raspberry PI. `https://wiki.newae.com`. Accessed: 2016-08-03.

[99] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold Implementations Against Side-Channel Attacks and Glitches. In *Information and Communications Security*, volume 4307 of *LNCS*. Springer, 2006.

[100] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches. *Journal of Cryptology*, vol. 24:292–322, 2011.

[101] Colin O'Flynn. Fault Injection using Crowbars on Embedded Systems. *IACR Cryptology ePrint Archive*, 2016:810, 2016.

[102] S. Ordas, L. Guillaume-Sage, and P. Maurine. Electromagnetic Fault Injection: The Curse of Flip-Flops. *Journal of Cryptographic Engineering*, pages 1–15, 2016.

[103] Elisabeth Oswald, Stefan Mangard, Christoph Herbst, and Stefan Tillich. Practical second-order DPA attacks for masked smart card implementations of block ciphers. *Lecture notes in computer science*, pages 192–207, 2006.

[104] Daniel Otte. AVR crypto lib. http://avrcryptolib.das-labor.org. Accessed: 2016/01/13.

[105] Olivier Pereira, François-Xavier Standaert, and Srinivas Vivek. Leakage-resilient authentication and encryption from symmetric cryptographic primitives. In *CCS 2015*, pages 96–108, 2015.

[106] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. DRAMA: Exploiting DRAM Addressing for Cross-CPU Attacks. In *USENIX Security Symposium 2016*, pages 565–581, 2016.

[107] Krzysztof Pietrzak. A leakage-resilient mode of operation. In *EUROCRYPT 2009*, pages 462–482, 2009.

[108] Bart Preneel and Paul C. van Oorschot. On the security of two MAC algorithms. In *EUROCRYPT '96*, pages 19–32, 1996.

[109] Jean-Jacques Quisquater and David Samyde. ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards. In *Smart Card Programming and Security – E-smart 2001*, pages 200–210, 2001.

[110] Raphael Spreitzer, Veelasha Moonsamy, Thomas Korak, Stefan Mangard. SoK: Systematic Classification of Side-Channel Attacks on Mobile Devices. https://arxiv.org/abs/1611.03748. Accessed: 2017-01-19.

[111] Oscar Reparaz. Detecting flawed masking schemes with leakage detection tests. Cryptology ePrint Archive, Report 2016/282, 2016. http://eprint.iacr.org/2016/282.

[112] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. Consolidating Masking Schemes. In *Advances in Cryptology - CRYPTO 2015 - 35th Annual Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2015, Proceedings, Part I*, 2015.

[113] Lionel Rivière, Zakaria Najm, Pablo Rauzy, Jean-Luc Danger, Julien Bringer, and Laurent Sauvage. High Precision Fault Injections on the Instruction Cache of ARMv7-M Architectures. In *Hardware Oriented Security and Trust – HOST 2015*, pages 62–67, 2015.

[114] Phillip Rogaway. Efficient instantiations of tweakable blockciphers and refinements to modes OCB and PMAC. In Pil Joong Lee, editor, *Advances in Cryptology - ASIACRYPT 2004, 10th International Conference on the Theory and Application of Cryptology and Information Security, Jeju Island, Korea, December 5-9, 2004, Proceedings*, volume 3329 of *Lecture Notes in Computer Science*, pages 16–31. Springer, 2004.

[115] Phillip Rogaway, Mihir Bellare, and John Black. OCB: A block-cipher mode of operation for efficient authenticated encryption. *ACM Trans. Inf. Syst. Secur.*, 6(3):365–403, 2003.

[116] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *Advances in Cryptology - EUROCRYPT 2006, 25th Annual International Conference on the Theory and Applications of Cryptographic Techniques, St. Petersburg, Russia, May 28 - June 1, 2006, Proceedings*, volume 4004 of *Lecture Notes in Computer Science*, pages 373–390. Springer, 2006.

[117] Cyril Roscian, Alexandre Sarafianos, Jean-Max Dutertre, and Assia Tria. Fault Model Analysis of Laser-Induced Faults in SRAM Memory Cells. In *Fault Diagnosis and Tolerance in Cryptography – FDTC 2013*, pages 89–98, 2013.

[118] Lorenz Schwittmann, Viktor Matkovic, Matthäus Wander, and Torben Weis. Video Recognition Using Ambient Light Sensors. In *Pervasive Computing and Communication Workshops – PerCom 2016*, pages 1–9, 2016.

[119] Laurent Simon, Wenduan Xu, and Ross Anderson. Don't Interrupt Me While I Type: Inferring Text Entered Through Gesture Typing on Android Keyboards. *PoPETs*, 2016:136–154, 2016.

[120] Sergei Skorobogatov. The Bumpy Road Towards iPhone 5c NAND Mirroring. *CoRR*, abs/1609.04327, 2016.

[121] Sergei P. Skorobogatov and Ross J. Anderson. Optical Fault Induction Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2002*, pages 2–12, 2002.

[122] Chen Song, Feng Lin, Zhongjie Ba, Kui Ren, Chi Zhou, and Wenyao Xu. My Smartphone Knows What You Print: Exploring Smartphone-based Side-channel Attacks Against 3D Printers. In *Conference on Computer and Communications Security – CCS 2016*, pages 895–907, 2016.

[123] Raphael Spreitzer, Simone Griesmayr, Thomas Korak, and Stefan Mangard. Exploiting Data-Usage Statistics for Website Fingerprinting Attacks on Android. In *Security and Privacy in Wireless and Mobile Networks – WISEC 2016*, pages 49–60, 2016.

[124] François-Xavier Standaert, Olivier Pereira, Yu Yu, Jean-Jacques Quisquater, Moti Yung, and Elisabeth Oswald. Leakage resilient cryptography in practice. In *Towards Hardware-Intrinsic Security – Foundations and Practice*, pages 99–134. Springer, 2010.

[125] Mostafa M. I. Taha and Patrick Schaumont. Side-channel countermeasure for SHA-3 at almost-zero area overhead. In *HOST 2014*, pages 93–96. IEEE Computer Society, 2014.

[126] The CAESAR committee. CAESAR: Competition for authenticated encryption: Security, applicability, and robustness. `http://competitions.cr.yp.to/`, 2014.

[127] Karim Tobich, Philippe Maurine, Pierre-Yvan Liardet, Mathieu Lisart, and Thomas Ordas. Voltage Spikes on the Substrate to Obtain Timing Faults. In *Digital System Design – DSD 2013*, pages 483–486, 2013.

[128] Elena Trichina. Combinational logic design for AES subbyte transformation on masked data. *IACR Cryptology ePrint Archive*, 2003, 2003.

[129] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23:37–71, 2010.

[130] Jasper G. J. van Woudenberg, Marc F. Witteman, and Federico Menarini. Practical Optical Fault Injection on Secure Microcontrollers. In *Fault Diagnosis and Tolerance in Cryptography – FDTC 2011*, pages 91–99, 2011.

[131] Johannes Wolkerstorfer, Elisabeth Oswald, and Mario Lamberger. *An ASIC Implementation of the AES SBoxes*, pages 67–78. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[132] Qiuyu Xiao, Michael K. Reiter, and Yinqian Zhang. Mitigating Storage Side Channels Using Statistical Privacy Mechanisms. In *Conference on Computer and Communications Security – CCS 2015*, pages 1582–1594, 2015.

[133] Lin Yan, Yao Guo, Xiangqun Chen, and Hong Mei. A Study on Power Side Channels on Mobile Devices. *CoRR*, abs/1512.07972, 2015.

[134] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *USENIX Security Symposium 2014*, pages 719–732, 2014.

[135] Danfeng Zhang, Yao Wang, G. Edward Suh, and Andrew C. Myers. A hardware design language for timing-sensitive information-flow security. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '15, pages 503–516, New York, NY, USA, 2015. ACM.

[136] Nan Zhang, Kan Yuan, Muhammad Naveed, Xiao-yong Zhou, and XiaoFeng Wang. Leave Me Alone: App-Level Protection against Runtime Information Gathering on Android. In *IEEE Symposium on Security and Privacy – S&P 2015*, pages 915–930, 2015.

[137] Xiao-yong Zhou, Soteris Demetriou, Dongjing He, Muhammad Naveed, Xiaorui Pan, XiaoFeng Wang, Carl A. Gunter, and Klara Nahrstedt. Identity, Location, Disease and More: Inferring Your Secrets From Android Public Resources. In *Conference on Computer and Communications Security – CCS 2013*, pages 1017–1028, 2013.

[138] Tong Zhu, Qiang Ma, Shanfeng Zhang, and Yunhao Liu. Context-free Attacks Using Keyboard Acoustic Emanations. In *Conference on Computer and Communications Security – CCS 2014*, pages 453–464, 2014.