Domain-Oriented Masking–
Generically Masked Hardware Implementations

by

Hannes Groß

A PhD Thesis
Presented to the Faculty of Computer Science
in Partial Fulfillment of the Requirements for the
PhD Degree

Assessors

Prof. Stefan Mangard (Graz University of Technology, Austria)
Prof. Lejla Batina (Radboud University, Netherlands)

June 2018

Institute for Applied Information Processing and Communications (IAIK)
Faculty of Computer Science
Graz University of Technology, Austria

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present doctoral thesis.

Date

Signature

Abstract

An ever-growing number of devices are threatened by low-cost Side-Channel Analysis (SCA) attacks and therefore require efficient protection mechanisms. Masking provides a high level of resistance against SCA at an adjustable level of security. A high level of SCA resistance, however, goes hand in hand with an increasing demand for chip area and fresh randomness which drastically increases the overall implementation costs. In this thesis, we investigate efficient ways to protect security-sensitive hardware implementations against SCA by means of Boolean masking, and show formal methods to prove the security claims of our masked hardware implementations.

In the first part of this thesis, we discuss different hardware-based masking schemes and possible trade-offs to make masking less demanding in terms of randomness requirements or faster by reducing the latency of the masked circuits. As basis of our investigation we introduce the so-called Domain-Oriented Masking (DOM) scheme, which allows for generic protection against SCA at low randomness requirements compared to other masking schemes. By simply adjusting its security parameter (d), masked hardware implementations protected by DOM can be synthesized for arbitrary protection orders without changing the underlying hardware designs. Regardless of its generic appearance, DOM uses the minimum number of random shares, which is beneficial for the chip area costs of the designs, and requires only $d(d+1)/2$ fresh random bits for the protection of a single-bit finite field multiplication.

On the basis of DOM, we then introduce two variants called Unified Masking (UMA) and Low-Latency Masking (LOLA). With UMA we show that with a better control over the signal flow of security-critical data in hardware (by using more registers), the randomness costs of DOM can be reduced to only around $d(d+1)/4$ fresh random bits. For this purpose, we use algorithms known from software-based masking and combine them with the ideas of DOM to lower the randomness costs even further and to make their algorithms suitable for hardware designs.

Contrary to UMA, LOLA allows to loosen the control over the data flow and thus to reduce the number of register stages. Less register stages result in less latency for the resulting masked hardware designs at the cost of higher randomness requirements, more data redundancy, and additional circuitry. The higher randomness costs result from the fact that we allow the number of shares to grow in our design at every nonlinear operation and perform a randomness-costly share reduction as late as the randomness constraints allow.

The second part of this thesis investigates the benefits and drawbacks of the masking schemes from the first part. At first, we compare DOM to other hardware-based masking schemes by using generic implementations of the Advanced Encryption Standard (AES) before demonstrating how LOLA can be used to reduce the latency of the AES S-box from at least five cycles of the DOM variants to only one or two cycles.

We also participated in the evaluation of the next generation of symmetric ciphers by designing hardware implementations for the so-called CAESAR competition, which has the goal to find suitable candidates for a portfolio of so-called Authenticated Encryption (AE) schemes. Besides the introduction of three unprotected variants of the AE scheme ASCON that were designed to fulfill the needs to a wide variety of AE applications, we also compare different masked variants of ASCON for which we compare DOM, UMA, and LOLA designs. Furthermore, we show optimizations for DOM on the case study of the hash function KECCAK which has become the new Secure Hash Algorithm 3 (SHA3) standard. We complete this part of the thesis with a protected implementation of a 32-bit RISC-V processor to demonstrate the versatility of our DOM approach.

Although, masking theoretically provides a high level of resistance against side-channel analysis, in practice there are many possible pitfalls when masking schemes are applied, and implementation flaws are easily overlooked. Over the recent years, the formal verification of masked software implementations has made substantial progress. In contrast to software implementations, hardware implementations are inherently susceptible to glitches. Therefore, the same methods tailored for software implementations are not readily applicable.

In the third part of this thesis, we introduce a method to formally verify the security of masked hardware implementations that takes glitches into account. Our approach does not require any intermediate modeling steps of the targeted implementation. The verification is performed directly on the circuit's netlist in the probing model with glitches and covers also higher-order flaws. For this purpose, a sound but conservative estimation of the Fourier coefficients of each gate in the netlist is calculated, which characterizes statistical dependence of the gates on the inputs and thus allows to predict possible leakages. In contrast to existing practical evaluations, like t-tests, this formal verification approach makes security statements beyond specific measurement methods, the number of evaluated leakage traces, and the evaluated devices. Furthermore, flaws detected by the verifier are automatically localized. We have implemented our method on the basis of a SAT solver and demonstrate the suitability on a range of correctly and incorrectly protected circuits of different masking schemes and for different protection orders. Our approach is efficient enough to prove the security of a full DOM masked first-order AES S-box, and of the DOM KECCAK S-box up to the third protection order.

"By His hand, we are all fed.
Give us Lord, our daily bread.
Please know that we are truly grateful.
For every cup and every plateful."
— Texan Mealtime Prayer (TBBT)

Acknowledgements

This dissertation is the sum of many scientific and personal contributions which have formed this thesis, and for which I am very thankful. In the following, I want to say thank you to all the people without whom this PhD thesis would not have been possible. Keeping their contributions in mind, I refrain from using the personal pronoun "I" or any derived form in the remaining parts of this thesis and instead use "we" to underline that many people have contributed to the presented work.

I would not have started this five year long journey to receive my PhD without Thomas Plos and Martin Feldhofer, and therefore would have missed many great experiences for which I feel very blessed today. Many thanks to both of you.

Dear Stefan, as my advisor you have always encouraged me to reconsider what seemed to me to be carved in stone. Ever since I presented my work on threshold implementations of arithmetic logic units and you said: "There must be a way to do this with less shares...", my scientific journey has taken a direction I could have never imagined when I started my PhD. Only we know how much passion and effort we put into our work on the Domain-Oriented Masking scheme and how hard we fought to get this work published. Regardless of the tough headwind we had to overcome, your faith in my work has given me the right motivation to focus my entire research on masking. I will always be grateful to you for giving me the time to realign my research topic and for your guidance on my way to this thesis.

Dear Raphael, you were the best "bunkmate" I could have had, and over the years you have become a very dear friend to me. Thank you for cheering me up after bad reviews, for proof-reading many of my papers, watering my rubber tree while I was on vacation, and thanks for the laughs and rums we shared.

I want to thank all my colleagues from the Secure Systems Group, who have made my life at the institute more enjoyable with our "little board" meetings early in the morning, our board game evenings, our "almost weekly" beer rounds, and at many other occasions. I also want to thank all my colleagues outside the Secure Systems group with whom I really enjoyed working together: Roderick Bloem, Rinat Iusupov, Bettina Könighofer, Daniel Slamanig, Johannes Winter, and my master students Manuel Jelinek and David Schaffenrath.

Special thanks to the people from Cafe Zapo for providing a family-like lunch atmosphere, and especially to Raimund, the best cook in Graz (and far beyond) who has become a very dear friend to me. Your great food has helped me more than once to get through a rough day.

There are a lot of people outside my work environment who I would like to thank for their support. Listing them all, along with the things I am thankful for would be an unmanageable task. So I apologize upfront for not mentioning everybody personally. Above all, I want to thank my beloved family: My parents who always supported me in every thinkable way without ever putting any pressure on me, and who constantly showered me with their unconditional love. My two brothers who enrich my life and believe in me much more than I ever could. My grandparents for always being there for me and for being role models in so many important aspects of life. My in-laws who welcomed me in their family from the very first day on. All my friends who I very much consider as part of my family.

Finally, I want to thank my beloved wife Angelika, who I feel deserves my gratitude most of all. Dearest Angelika, ever since the day we met, regardless of what was going on in my life, I always felt to be the luckiest man to have you in my life. Thank you so much for cheering me up and supporting me after every setback, and for providing me with a safe haven where I could forget all worries and felt that nothing really bad could ever happen in my life, as long as you are with me. Thank you for sharing so many good times with me, for enjoying the things that I enjoy most, and for laughing with me over things that nobody else could ever understand.

I dedicate this thesis to you and my everlasting love for you.

January 2018, Graz

Table of Contents

III Verification of Masking 105

List of Tables

List of Figures

List of Abbreviations

AE	Authenticated Encryption
AES	Advanced Encryption Standard
ALU	Arithmetic-Logic Unit
ASIC	Application-Specific Integrated Circuit
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
DOM	Domain-Oriented Masking
DPA	Differential Power Analysis
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
ISA	Instruction-Set Architecture
LOLA	Low-Latency Masking
LSFR	Linear-Feedback Shift Register
LUT	Look-Up Table
PRNG	Pseudo-Random Number Generator
RNG	Random Number Generator
SCA	Side-Channel Analysis
SHA3	Secure Hash Algorithm 3
SPA	Simple Power Analysis
TI	Threshold Implementations
TRNG	True-Random Number Generator
UMA	Unified Masking

"One cannot not communicate."
— Paul Watzlawick

Introduction

The security of our entire digital world that increasingly becomes interwoven with our everyday life strongly relies on the ability to keep secrets. Without secrets, even the strongest cryptographic primitives and protocols fail in providing confidentiality, integrity, and authenticity. Many amenities we gained through digitalization, like electronic payment, e-commerce, eGovernment but also safety-relevant applications like medical devices, are unthinkable without sound cryptographic implementations that conceal the secrets on which they rely. So-called Side-Channel Analysis (SCA) attacks can easily reveal a device's secrets when no appropriate countermeasures are implemented.

In general, the term SCA refers to an attack that does not target the main communication channel but exploits information that is produced as a by-product of computation or data transmission with the goal to reveal security sensitive information. Before a more precise classification of the SCA attacks that are considered in this thesis is given, we first briefly introduce the idea of SCA with some real-world examples.

Side-Channel Analysis

"Every computation leaks information" — this famous variation of a quote stated by Micali and Reyzin in [MR04] is one of the most fundamental assumptions of the side-channel community. However, SCA is not an invention of the computer sciences or even of our time. During World War I, field telephones to send orders of the day to different military corps used only one wire for communication [MC14]. The earth was used as voltage reference for the communication channel and thus the return current was measurable by enemy intelligence corps by using rods that were buried in the earth to eavesdrop on the enemy's orders. During World War II, Bell Labs were aware of electromagnetic emanation attacks to eavesdrop on secured communications over distances of more than 24 meters.

SCA is not even a human invention. In Paul Watzlawick's famous book [Wat76] "How real is real?" he discusses the human inability to not communicate as one of the most fundamental principles in communication science. In human communication, even the absence of words can therefore be a strong statement and other communication channels exist beside speech that are much harder to control, like facial expression or gesture.

As an example, Watzlawick retold a story of a mathematics teacher and amateur horse trainer named Wilhelm von Osten who lived around the beginning of the 20th century. Von Osten had an Orlov trotter horse which became famous as the Clever Hans after which even a physiological effect was named. Wilhelm von Osten claimed that his horse was able to perform simple arithmetic and other intellectual tasks, by answering by tapping with its hoof or by nodding or shaking its head. These claims were investigated by a committee formed around the psychologist Carl Stumpf. The result of the investigation was that no tricks were involved and thus confirming von Osten's claims regarding the abilities of his horse. The investigation was passed on to the psychologist Oskar Pfungst who performed a couple of experiments that should disconnect the horses reaction from environmental influences like willful or unintentional clues from the audience or his master. As a result of the experiments Pfungst concluded that the horse only answered correctly, if the answer is also known by the questioner and if the horse could see the person that asked the question. Further investigations revealed that the only possibility for the horses correct answers was its ability to read, even unintentionally given clues from persons, like a change in the posture or the facial expression, when the number of given hoof taps reached the correct result. This case could thus be the first documented SCA performed by an animal on people.

There exists a broad variety of documented SCA attacks like the examples stated above performed on different targets and by exploiting different side-channels. In the following, we want to provide a differentiation on what SCA means in the context of this work.

Classification

We follow the path given by the underlined keywords in Figure 1 from top to bottom which denote the attack classification attributes which are relevant for the remainder of the thesis. Even though the outcome of the research that was conducted for this thesis is applicable to protect a broader class of security sensitive applications, most examples within this thesis are given on the basis of symmetric cryptographic primitives. More precisely, the kind of attacks for which we provide countermeasures are attacks on the implementation of these cryptographic primitives in Complementary Metal-Oxide-Semiconductor (CMOS) hardware, rather than attacks targeting the mathematical structure of these primitives like linear or differential cryptanalysis.

Active and Passive Attacks. Side-channel attacks are so-called passive attacks that work by only observing the behavior of an implementation and by concluding from the observed behavior information about the processed secret data (like the used symmetric key). Such information exploited by an attacker can be, for example, the power consumption or the electromagnetic emanation of a device during security-sensitive computations. The goal of an attacker is to either directly determine the used secret information or to narrow down the search space so that the secret information can be calculated within reasonable time.

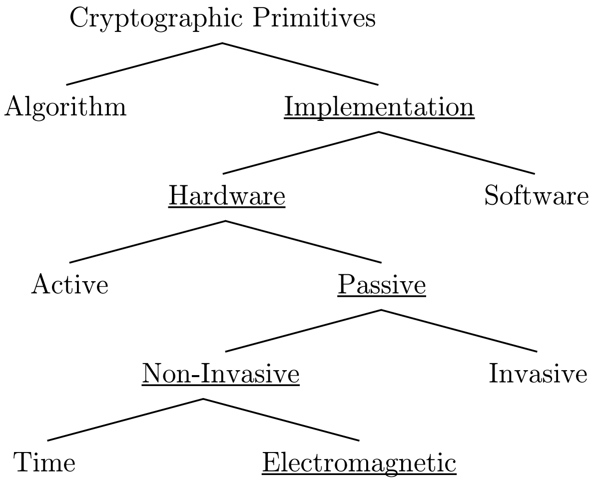

Figure 1: Classification of the attacks considered in this thesis

Strong symmetric primitives are designed in such a way that, even for an observation of multiple plaintext and ciphertext pairs, the remaining key space is reasonably larger than the complexity that can be handled by today's computers. For example, it is known from cryptanalysis of the 128-bit key variant of the Advanced Encryption Standard (AES) that the key space that needs to be tried out by an attacker can be reduced from 128 bits to about 126.1 bits [Bog+11]. This is, however, still much more than what can be brute-force searched by conventional (non-quantum) computers in the foreseeable future. By using SCA, on the other hand, the possible key space can be reduced significantly depending on the targeted platform and the concrete implementation.

Contrary to passive attacks, there are also active attacks that not only observe the behavior of an implementation but try to manipulate the execution in such a way that it results in an advantageous misbehavior from an attacker's perspective. Active attacks include, for example, fault analysis attacks where a device is used outside its specification ranges (clock speed, temperature, supply voltage) to produce faulty behavior. Another way to disturb the correct execution of a device is to use lasers or other electromagnetic sources. For the AES-128 encryption for instance, which consists of ten consecutive round transformations before the ciphertext is returned, a fault attack can be used to skip several of these rounds. The produced ciphertext thus reveals the internal state of the cipher at an early point during the execution which allows to determine the key or to cut down the possible key space to one that can be brute-force searched in reasonable time. Active attacks are very powerful and countermeasures against these attacks are thus intensively researched in the literature. However, defense mechanisms against active attacks are out of scope for this thesis.

Invasive and Non-Invasive Attacks. Side-channel attacks as considered for this thesis are assumed to be non-invasive, which means that the attack is performed on the device as it is without altering the device or even (partially) destroying it. Invasive attacks, on the other hand, are performed for example by opening the package of a chip. This is done by using etching and polishing techniques with the goal to gain access to the surface of the IC's die. Up to this point, the attack would still qualify as a semi-invasive attack, since the chip die itself is not penetrated. If the motivation for getting access to the chip surface is not only to study the architecture of the chip, but to the get access to the internal structures of the chip the attack is considered to be invasive. An example for such an invasive attack is the usage of probing needles or focused ion beams to read-out or manipulate information on chip wires. Techniques to counteract invasive attacks are not considered in this thesis.

Types of Side-Channels. There exist many possible types of side channels that can be exploited by an attacker. Some of the first side-channel attacks on cryptographic implementations were attacks on the execution time [Koc96]. If there are operations whose execution time varies with regard to the used key or internal state, then the execution time allows to draw conclusions on security-sensitive information. Even though the awareness for the execution time as side-channel has increased over the last years, there are still new attacks coming up that exploit variations in the runtime, like cache timing attacks [Lip+16; Lip+18]. A basic requirement for cryptographic implementations is thus constant runtime, which is also the case for the side-channel protected implementations we introduce in this thesis. Nevertheless, the techniques we present in this thesis to prevent SCA are not suited to seal leakages that are caused by non-constant runtime. We focus on side-channels caused by electromagnetic effects like power consumption [Koc+99] and electromagnetic emanation [QS01]. Such attacks are in the literature often referred to as power analysis attacks or electromagnetic emanation analysis attacks, respectively. For the remainder of this thesis, however, we generally refer to this kind of attacks as SCA.

Attack Scenario

A typical measurement setup to extract side-channel leakage information from a cryptographic hardware implementation is depicted in Figure 2. During the computation, the chip generates electromagnetic radiation that is produced by changes in the electric current flowing through the circuit (gates, transistors, wires). These changes in the current flow produce changes in the electromagnetic field that are sensed by the probe in form of a small induced current. The produced electromagnetic radiation is thus a result of the changes in the power consumption of the chip or the part of the chip that is probed. The amplifier that is connected to the probe is used to amplify the small current that flows through the electromagnetic probe so that the recording of the signal uses the optimal dynamic range of the oscilloscope. The recorded traces are then evaluated using, for example, statistical analysis, machine learning algorithms or pattern recognition. Because of the different attack methods and ways to exploit leakages, there exists a whole variety of names for side-channel attacks. A coarse distinction can be made between so-called Simple Power Analysis (SPA) and Differential Power Analysis (DPA).

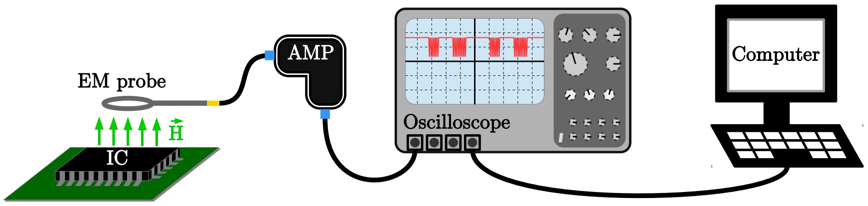

Figure 2: Typical measurement setup for an SCA attack

Simple Power Analysis. According to [Man+07], SPA attacks are attacks that work by exploiting key-dependent differences or patterns within a single leakage trace, for example, the recording of a device's power consumption during one encryption or a subsequence of the encryption procedure. This does not mean that for an SPA attack just a single power trace is used. Multiple traces could be used to lower the signal to noise ratio between the actual leakage signal and the electronic and measurement noise, for instance. However, the term "simple" refers to the fact that in an SPA, the relation between multiple traces and changes within intermediate values that produce the recorded leakage trace is not exploited.

A classical example for an SPA attack is the recognition of leakage patterns that are caused by individual commands of a processor. A square-and-multiply algorithm, as it is used in binary exponentiation of insecure implementations of elliptic curve point multiplications or also in RSA encryption, shows recognizable patterns depending on whether a simple squaring (shift operation) or a more complex multiplication is performed. Since these patterns can even emerge in pure visual inspection of a power trace, the exponent that was used for the exponentiation (which could be the key itself or key-related) can be directly extracted from the power trace.

Differential Power Analysis. In contrast to SPA, DPA exploits the relation among power traces to find evidence for the usage of a certain key or other secret data. A classical DPA can be coarsely divided into three phases: leakage gathering phase, hypotheses building phase, and hypotheses matching phase. In the following we only briefly summarize how a typical DPA attack works. For more details we refer the interested reader to [Man+07].

In the first phase leakage traces are collected under varying inputs, e.g. different plaintexts that are encrypted using the same secret key. The exploitation of differences in leakage traces under different inputs requires an alignment of the individual traces to each other that needs to be as exact as possible for the hypotheses matching phase. For practical attacks, there often exists no exact reference point in time for each encryption. This phase could thus involve a post-processing phase in which the leakage traces are first aligned. More advanced attacks require an even more complex post-processing that involves filtering or the combination of leakage points, but for the remainder of this introduction to power analysis we assume perfectly aligned traces that do not require post-processing for the attack to work.

Since no knowledge on the used secret key is assumed, the next step is to generate hypotheses on the basis of the unknown secret key. This phase is usually performed in a divide-and-conquer manner because building hypothesis based on a 128-bit key space as required for a full AES-128, for example, would be too complex. Therefore, a suitable point in the attacked algorithm needs to be found at first for which there exists a relation to the attacked secret key and the known input data. A suitable point for building this hypothesis is for example the result of the addition of the key and the plaintext in the first round of an AES that is followed by the 8-bit S-box lookup. Hypotheses can thus be calculated individually for all 8-bit chunks of the 128-bit key. The hypotheses contain the potential values of the attacked intermediate result of the algorithm that are calculated for each leakage trace of the targeted key byte. As the dynamic power consumption of a CMOS circuit depends on changes of a signal rather than the absolute values of the intermediate value, the hypothetical intermediate values are then mapped to a power consumption model. The mapping to a power model can again be rather simple, for example just calculating the number of bits that are nonzero (Hamming-weight model) or the number of bits that changed their value (Hamming-distance model), but can also become more complex and use power consumption characteristics of the attacked device.

In the matching phase, the hypothetical power model for the specific key guesses are statistically evaluated against the actual observations in the leakage traces. In practice, there exists a vast number of so-called side-channel distinguishers to select the most probable key candidate from the set of hypothetical keys. These distinguishers are based on different statistical methods like the Pearson correlation, difference of means, or mutual information analysis, and have varying practical properties and implications. The goal of all of these distinguishers is, however, to find and quantify dependencies between the hypothetical power models and leakage traces in order to determine the used key.

Masking as Countermeasure to SCA

There have been many methods researched over the last almost 20 years to counteract SCA, which can be roughly categorized into masking and hiding countermeasures (e.g. randomized execution paths). However, masking is the best researched countermeasure against SCA and favored because it severely impedes the exploitation of side-channel leakage. The core idea of masking is to make the produced side-channel leakage statistically independent from the data beeing processed. In classical masking this independence is achieved by masking a sensitive variable e.g. x as the sum of the variable and some fresh random masks (m).

$$x_m = x \oplus m_0 \oplus m_1 \oplus \dots$$

We will use this masking notation later in Part III. However, for the first parts of this thesis we use a sharing based notation which is just a different way to denote the masking. For this purpose, the secret data is assumed to be split into a number of so-called shares, which when recombined through addition over $GF(2^n)$ result in the original data again. The sharing is again based on uniformly distributed random masks. The sharing of a variable x can be written as shown in Equation 1, where the shares are denoted by numbers in the subscript index.

$$x = x_0 \oplus x_1 \oplus x_2 \oplus \dots \tag{1}$$

We can easily convert between the two representation forms, e.g. if we assign $x_0 = x \oplus m_0 \oplus m_1 \oplus \dots$ and the remaining shares to be just one of the masks ($x_1 = m_0$, $x_1 = m_1$, et cetera). Both forms are equal but the sharing-based notation is more general because it hides how the masking was initially performed. For the sharing-based form, it is just relevant that only the combination of all shares of one masked variable will leak any information on the underlying secret.

The sharing does not only affect the representation of the data but also of the functions that are applied to this data. An unshared function “F” is split up into a number of component functions denoted by the original function’s name with a number in the index. Again, the sum over the component functions must give the same result as for the unshared variables (see Equation 2).

$$\mathrm{F}(x, y) = \mathrm{F}_0 \oplus \mathrm{F}_1 \oplus \mathrm{F}_2 \oplus \dots \tag{2}$$

Independence and the Probing model. A basic requirement of all masking schemes is that each intermediate signal needs to be statistically independent of all unshared inputs and outputs. Often maintaining this independence requires the addition of a fresh random share to intermediate results. In this thesis, we always use r shares to refer to randomly picked shares with the intention to provide statistical independence. The independence requirement is strongly related to the so-called probing model.

The probing model introduced by Ishai, Sahai and Wagner [Ish+03] is the de facto standard model in which the side-channel resistance of a Boolean masked circuit is analyzed. Informally speaking, a circuit is said to be d^{th}-order secure in the probing model, if an attacker with the ability to place up to d probing needles on to any wire or gate of a circuit (to continuously record the signal transitions over time) is not able to combine the recorded information to reveal any (unshared) critical information. For example, the sharing of the variable x with $d+1$ shares is by definition d^{th}-order secure in the probing model because it would require more than d needles to collect all $d+1$ shares. The inherent goal of a masked circuit is to keep this independence throughout the entire circuit.

It was demonstrated by Faust et al. [Fau+10] and Rivain et al. [RP10] that there indeed exists a relation between the number of probed wires in the probing model and the attack order for a differential DPA attack. As it was shown by Chari et al. [Cha+99], there exists an exponential relation between the protection order and the number of required leakage traces to unmask the secrets.

While linear functions over $GF(2^n)$ can be implemented trivially, the implementation of nonlinear parts is quite challenging. In the past, Galois field (GF) multipliers have shown to be a good reference to compare masking schemes. Multipliers are also of particular interest because on the basis of a simple one-bit multiplier, which corresponds to an AND gate, every boolean logic gate can be realized and in consequence every possible circuit. In the following, we explain a classical masking approach based on the Trichina gate [Tri03] and the Threshold Implementations (TI) scheme, as examples for different masking schemes, on the basis of a Galois field multiplier that is protected against first-order attacks.

Classical Boolean Masking. First-order masking in general requires only two shares. Accordingly, a shared multiplication of two inputs x and y over $GF(2^n)$ can be written as the multiplication of two shared finite field elements as demonstrated in Equation 3, where $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$.

$$\begin{aligned} q = x \cdot y &= (x_0 \oplus x_1)(y_0 \oplus y_1) \\ &= x_0y_0 \oplus x_0y_1 \oplus x_1y_0 \oplus x_1y_1 \end{aligned} \tag{3}$$

While each partial product is statistically independent of x and y, the resulting sum is not. A fresh random share r_0 needs to be added to the first multiplication result of Equation 3 to maintain first-order security in the probing model. Figure 3 shows a classical masked $GF(2^n)$ multiplier introduced by Trichina [Tri03].

Even though this multiplier seems to be secure in the probing model, this implementation is still not free from first-order leakages. Consider the results of the two multipliers on the left, for example, that calculate x_0y_0 and x_0y_1. If these intermediate signals reach the exclusive-OR gate before r_0, then the resulting signal is no longer statistically independent of y. As a result, the security of the masked multiplier depends on signal transition times caused by wire lengths, transistor speeds, et cetera, which is hard to control for digital designers. For this reason, the classical Boolean masking scheme is considered to be flawed [Man+05]. As shown by Mangard et al. [MS06] the problems are not caused by the four multipliers or their additive leakage. The only reason this sharing produces first-order leakages are glitches that are caused by the addition of the multiplier results.

Researchers have tried to repair the masked multiplier in Figure 3 [Ala+09; Gho+07; Kum+07]. These works mainly focused on balancing and reordering the signals and gates in such a way that no glitches can occur any longer. These

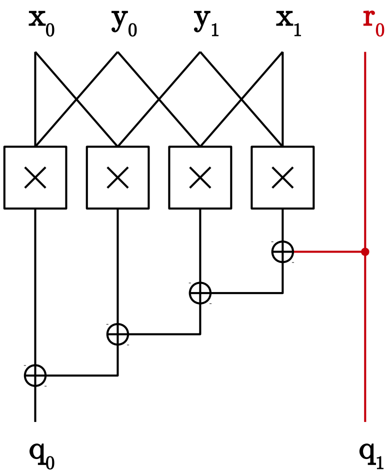

Figure 3: Classical masked $GF(2^n)$ multiplier according to [Tri03]

approaches, however, require an enormous effort in the backend of the hardware design flow in order to guarantee the correctness of the signal timings.

Threshold Implementations. The TI scheme by Nikova et al. [Nik+06] was the first provably secure masking scheme resistant to glitches. TI focuses on so-called component functions (cf. Equation 2), and on the properties these component functions have to fulfill to guarantee security in the probing model. These properties are called correctness, non-completeness, and uniformity. The *correctness* property of TI simply requires that the sum over the component functions must give the same result as the original function for the unshared variables.

The main idea of TI to prevent glitches that reduce the security in the probing model, is to only feed a subset of shares per variable into a component function (*non-completeness* property). As a direct consequence, the realization of nonlinear functions with TI always requires more than $d+1$ shares. While functions that are linear over $GF(2^n)$ can be implemented in a first-order secure manner with only two shares, Nikova et al. [Nik+06] state the lower bound for nonlinear functions with two variables to be at least three shares. In general, the number of input shares required for higher-order security [Bil+14c] is given by $s_{in} \geq d \cdot t + 1$ where s is the number of shares, d is the protection order, and t the degree of the function. The number of output shares for TI is given with $s_{out} \geq \binom{s_{in}}{t}$. In order to ensure that glitches do not propagate within adjacent component functions, registers are required at the output of each component function.

The property that is usually the hardest to achieve is *uniformity* which demands all share inputs and outputs of component functions to be uniformly distributed regardless of which unshared values they represent. For first-order TI, uniformity of the component functions can often be achieved without performing a complete resharing of the outputs by using more shares, or correction terms, or using fresh random shares in more than one component function.

As an example for a first-order secure TI, the $GF(2)$ multiplier in Figure 4 uses three shares per variable and one fresh random share for achieving uniformity of all output shares as shown by Bilgin [Bil+15a]. Equation 4 defines the corresponding component functions.

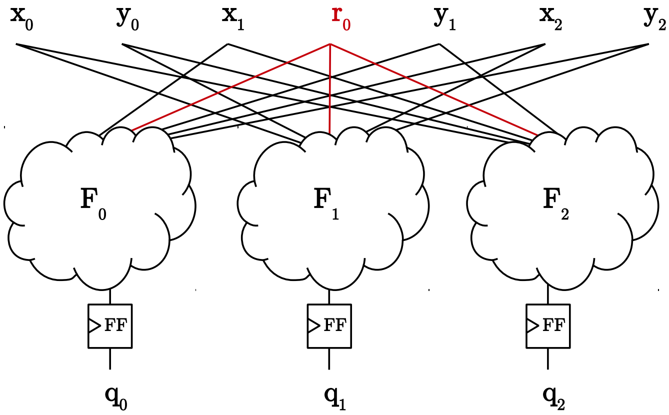

Figure 4: TI multiplier with component functions

$$\begin{aligned}
\mathrm{F}_0(\mathrm{x}_1, \mathrm{x}_2, \mathrm{y}_1, \mathrm{y}_2) &= x_1y_1 \oplus x_1y_2 \oplus x_2y_1 \oplus r_0 \\
\mathrm{F}_1(\mathrm{x}_0, \mathrm{x}_2, \mathrm{y}_0, \mathrm{y}_2) &= x_2y_2 \oplus x_0y_2 \oplus x_2y_0 \oplus x_0r_0 \oplus y_0r_0 \\
\mathrm{F}_2(\mathrm{x}_0, \mathrm{x}_1, \mathrm{y}_0, \mathrm{y}_1) &= x_0y_0 \oplus x_0y_1 \oplus x_1y_0 \oplus x_0r_0 \oplus y_0r_0 \oplus r_0
\end{aligned} \tag{4}$$

For the sake of completeness we note that the TI multiplier could also be implemented by using two fresh random shares and fewer logic gates, or using more shares and an increased gate count.

The non-completeness rule can be directly observed in the notation of the component functions of Equation 4. The first component function F_0, e.g., is independent of all shares with share index 0 (except the r shares), the second component function F_1 of all shares with index 1, and so forth. The glitch resistance, however, comes at higher costs in terms of gate count. While the classical multiplier requires only four AND gates and four XOR gates, the TI variant in Equation 4 consumes 13 AND gates, 12 XOR gates, and three registers.

In [Rep+15], Reparaz et al. introduced an extension of TI to higher orders, and mentioned that given an independent input sharing and carefully designed component functions with more calculation steps and registers, the number of shares can be lowered to $d+1$ for TI. However, they stated that they avoid giving a generic construction for the $d+1$ case, because extreme care has to be taken in order to not unmask any intermediate value.

Thesis Overview

The present dissertation examines the secure construction of generically masked hardware designs with a scalable protection level and their formal verification. In Part I, we introduce the Domain-Oriented Masking (DOM) scheme in Chapter 1, which is the basis for all other introduced masking schemes and practical implementations. We successively refine the idea of DOM in the remainder of this chapter, and research possible trade-offs to either save online randomness with the Unified Masking (UMA) scheme in Chapter 2 or ways to reduce the latency of DOM with our Low-Latency Masking (LOLA) scheme in Chapter 3.

Part II is about the practical comparison of the masking schemes introduced in Part I, which is performed on a wide variety of different masked hardware implementations. In Chapter 5 we start off with a DOM implementation of the AES which is the most widely deployed symmetric cryptographic primitive and thus used in many practical applications. In addition, we introduce different low-latency designs of the AES S-box and show the overhead costs of our LOLA scheme. The upcoming generation of symmetric primitives is considered in Chapter 6 in the form of the Authenticated Encryption (AE) scheme ASCON. We introduce many different protected and unprotected hardware implementations of ASCON and use them to compare all DOM based masking schemes among each other.

The application of DOM to the Secure Hash Algorithm 3 (SHA3) KECCAK along with different optimizations is shown in Chapter 7, before we conclude this part with a protected implementation of a RISC-V processor in Chapter 8.

The last part of this thesis (Part III) is about the formal verification of masking and presents a comparison with empirical verification methods for masked hardware implementations. We introduce a formal verification approach on the basis of the Fourier representation (or Walsh transformation) of Boolean functions which allows us to make a worst-case estimation of the data leakage under all possible signal timings. Our formal modeling approach is then used to evaluate practical masking examples.

For each part, we first briefly introduce the respective topics, we highlight our contributions in this area, and give closing remarks at the end of each part. We close this thesis with a discussion on open research questions and give an outlook on future research directions.

Part I

Generic Masking Schemes

In this part of the thesis, we first introduce the concept of Domain-Oriented Masking (DOM), a generic masking scheme that leads to hardware designs which can be synthesized for arbitrary Side-Channel Analysis (SCA) protection orders. DOM thereby realizes the same theoretical bounds for fresh randomness as the private circuits scheme [Ish+03] without being vulnerable to glitches. Therefore, we introduce the concept of share domains and apply the idea of keeping each domain independent from other share domains.

On the basis of DOM, we introduce two derived variants to explore different randomness, area, and latency trade-offs. With the Unified Masking (UMA) scheme, we show how the randomness usage can be lowered for the costs of better control over the transient behavior of the circuit. With the Low-Latency Masking (LOLA) scheme, we reduce the latency that is usually required in DOM for the remasking step in the nonlinear parts of the circuit.

This part is based on the following papers.

- **Chapter 1: Domain-Oriented Masking (DOM)**

  ▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order." In:* CT-RSA*. vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112*

  which is a reduced version of our IACR's ePrint archive paper:

  ▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order." In:* IACR Cryptology ePrint Archive *(2016)*

  that was also presented at the TIS workshop of the CCS and published as extended abstract:

  ▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order." In:* TIS@CCS*. ACM, 2016, p. 3*

- **Chapter 2: Unified Masking (UMA)**

  ▶ *Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In:* CHES*. vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136*

- **Chapter 3: Low-Latency Masking (LOLA)**

  ▶ *Hannes Groß, Rinat Iusupov, and Roderick Bloem.* Generic Low-Latency Masking in Hardware*. CHES 2018 (in press)*

Contribution. For the parts of the papers that were used in these chapters the author of this thesis is the main author.

> *"When you change the way you look at things, the things you look at change."*
>
> — Wayne Dyer

1

Domain-Oriented Masking (DOM)

In this chapter we introduce the general concept behind the DOM scheme. In contrast to Threshold Implementations (TI), which consider properties at function level, our DOM approach is based on the concept of share domains. In DOM, each share of a variable is associated with exactly one share domain. This is also reflected in the notation that is used. The shares x_0 and x_1 of a variable x, for example, are associated with the domains 0 and 1, respectively.

A DOM implementation uses $d+1$ shares per variable in order to achieve d^{th}-order security. There are $d+1$ domains in this case. The basic idea of the DOM approach is to keep the shares of all domains independent of shares of other domains. This independence ensures d^{th}-order security according to the d-probing model. If, for example, in a first-order security setting, a component function takes the inputs x_0 and y_0 from domain 0, all intermediate values calculated by this function are independent of the corresponding unshared inputs $x = x_0 \oplus x_1$ and $y = y_0 \oplus y_1$. This is a consequence of the fact that x_1 and y_1 are not part of this function and are combined by another component function working on domain 1.

In case of linear functions, the independence of the domains is trivial to achieve because linear functions only require to combine shares within one share domain. If only linear functions are used in a circuit, $d+1$ disjoint domains are formed, with no wires from one domain to another, that only contain one share of each input variable, and calculating one share per output variable. The critical part, like in all masking schemes, are the non-linear functions. In the case of non-linear functions, domain borders need to be crossed and dedicated measures need to be taken in order to maintain the independence of the shares in the different domains. The basic idea of DOM is to secure domain crossings by adding a fresh random share r and by using a register in order to prevent glitches from propagating from one domain to another. By carefully using the randomness, the signals that are derived from different domains can be reintegrated into the domains in a correct and secure manner.

In the following, we detail the concept for a two-input GF DOM multiplier, which serves as a basis to protect arbitrary circuits, and which is also the most critical part of masked Advanced Encryption Standard (AES) implementations, for instance. As for many other masked multipliers in the literature, one basic requirement of the DOM multiplier is that the inputs are shared independently.

As an example for a violation of share independence consider the classical masked GF multiplier (see Figure 3 in the introduction of this thesis). If this multiplier calculates $x \cdot x$ for the same sharing of both inputs x, then this would result in the multiplication terms x_0x_0, x_0x_1, x_1x_0, and x_1x_1. The terms x_0x_0 and x_1x_1 use only one share of x and are thus uncritical. The terms x_0x_1 and x_1x_0, on the other hand, violate the share independence by bringing together shares from different domains of one sharing. The share independence of course only requires the shares of the inputs to be independent not the variables themselves. It is therefore possible to calculate, for example, $x \cdot x$ with the DOM multiplier, as long as both inputs are shared independently.

In practice, dependencies between shares can be less obvious and are often just temporary. As an example, consider an unshared 2-bit transformation that is defined as follows: The first output bit p of this transformation is the linear combination of two of the input bits $x \oplus y$ and the second output bit is just the first input bit $q = x$. Due to signal delays it is possible that both output bits are temporarily formed by the same input bit x only. If those bits are the shared inputs of a non-linear multiplication, then this again results in a temporary violation of the share independence in form of so-called glitches.

We start to introduce DOM by means of the first-order secure DOM multiplier in Section 1.1 before we extend it in Section 1.2 to arbitrary protection orders.

1.1 First-Order Secure DOM Multiplier

A first-order secure DOM multiplier (see Figure 1.1) consists of two share domains. The inputs x and y are provided to the multiplier by the shares x_0 and x_1, and y_0 and y_1, respectively. The sharings for x and y are required to be uniformly random and independent of each other. The multiplier returns the shares q_0 and q_1 of the output q. A DOM multiplier performs three steps in order to map the input shares to the output shares. We refer to these steps as *calculation*, *resharing* and *integration*.

Calculation: In the calculation step, the actual multiplication is performed and the product terms x_0y_0, x_0y_1, x_1y_0 and x_1y_1 are calculated. In DOM, we distinguish between inner-domain terms (x_0y_0, x_1y_1) and cross-domain terms (x_0y_1, x_1y_0). The calculation of inner-domain terms only combines shares within one domain. These terms are not critical from a security point of view. Any function that is computed based on shares that are independent of the shares in

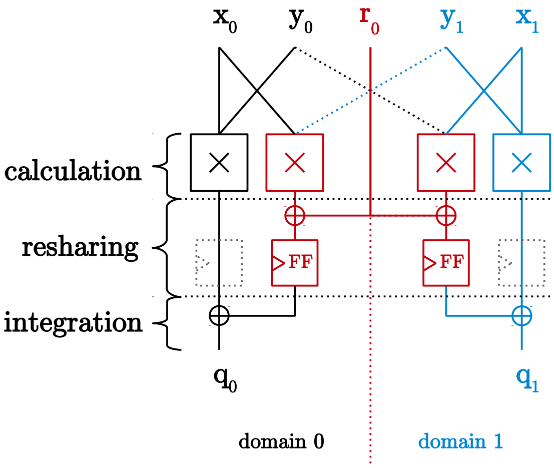

Figure 1.1: First-order DOM $GF(2^n)$ multiplier

other domains only leads to outputs that are also independent of the shares of the other domains.

In case of cross-domain calculations, there is less freedom. In fact, in a DOM scheme, cross-domain calculations can only be done for independently shared variables. If shares of the same variable were combined for example, the scheme would be trivially broken. For example, the product $x_0 x_1$ would leak information about x. However, shares from different domains corresponding to different variables can be combined without violating the requirement for d^{th}-order security. In fact, there is no leakage about x or y when calculating any function of x_0 and y_1. This results from the requirement that x and y are independently shared. There is also no leakage caused by any function of x_1 and y_0 for an independent sharing of x and y. Circuit parts that operate on inputs from multiple domains are plotted red in Figure 1.1. These parts are not assigned to a specific domain and contain the cross-domain terms.

Resharing: In DOM, the integration of cross-domain terms into a domain is prepared during the resharing step. By adding a fresh random r share to a cross-domain term, it becomes statistically independent of all other values and can therefore be added to any arbitrary domain in a next step. However, using a new share for each cross-domain term would lead to a high overhead. In DOM, the goal is to minimize the number of fresh shares. In case of the first-order secure multiplier, the same fresh share r_0 is used for the resharing of the product terms $x_0 y_1$ and $x_1 y_0$. This does not lead to a first-order leakage and at the same time allows for building a very efficient design.

In order to prevent any glitches from propagating through the resharing step, there is always a register included in DOM as last part of the resharing step.

The two registers in gray dotted lines are optional registers for the inner-domain terms and are only required in case pipelining is used. At first sight, the registers in Figure 1.1 seem to add an additional delay compared to the TI variant of the multiplier (see Figure 4, in the introduction). However, the TI scheme also requires registers at the output of each component function. Otherwise no cascading of functions is possible. In case of a DOM multiplier, the output can be directly plugged into the next nonlinear function, as long as the second input of this nonlinear function is not one of the inputs of the first nonlinear function (otherwise a register stage is required between the first and the second nonlinear function). The number of register stages is thus the same for both schemes.

Integration: During the integration phase, the reshared cross-domain terms are added to the domains, which concludes the GF multiplication. This addition leads to glitches at the XOR gate at the output of the domain. However, as the resharing finishes with a register, no glitches depending on x or y can occur. In terms of correctness of the scheme, it is important to point out that the fresh share r_0 becomes part of both domains of the multiplier. Hence, it holds that $q = q_0 \oplus q_1$ and no additional share is needed.

In summary, the security against a first-order probing attacker is ensured because each domain contains only inner-domain terms and cross-domain terms that are reshared with a fresh random share which is only used once in each domain. An attacker thus always needs to combine two or more intermediate signals to get one signal that depends on one of the independently shared inputs x or y. Problems caused by different signal propagation times are prevented through registered outputs in the resharing phase.

1.2 Higher-Order Secure DOM Multiplier

The first-order DOM multiplier can be extended to arbitrary protection orders. The generalization requires to first extend the *calculation* phase in order to produce a correct sharing with $d + 1$ shares for any given protection order d. In the *resharing* phase, it needs to be ensured that the fresh random r shares are distributed over the domains in a way that (1) each cross-domain term is reshared with an r share that is unique inside the targeted domain, and (2) none of the signal combinations created in the *integration* phase reveals more than the inner-domain terms or shares.

Calculation: The same rules as for the first-order DOM apply for the higher-order generalization. Again, any combination of inner-domain shares can be used for the multiplication terms inside their associated domain without any restrictions. Cross-domain multiplication terms are restricted to be originated from independently shared variables in order to prevent two shares of the same sharing from being combined.

Considering these restrictions, the $GF(2^n)$ multiplication formula can easily be generalized for $d + 1$ input shares per variable as shown in Equation 1.1.

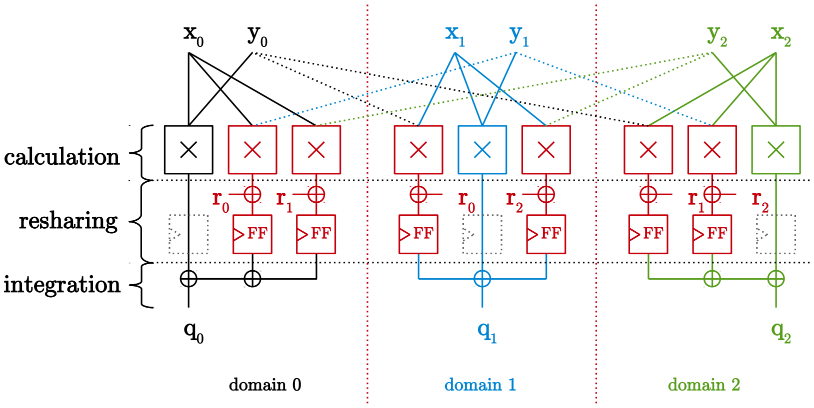

Figure 1.2: Second-order secure DOM $GF(2^n)$ multiplier

$$
\begin{aligned}
q = x \cdot y =& (x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4 \oplus \dots)(y_0 \oplus y_1 \oplus y_2 \oplus y_3 \oplus y_4 \oplus \dots) \\
=& \underbrace{\boldsymbol{x_0 y_0} \oplus x_0 y_1 \oplus x_0 y_2 \oplus x_0 y_3 \oplus x_0 y_4 \oplus \dots}_{q_0} \\
& \underbrace{x_1 y_0 \oplus \boldsymbol{x_1 y_1} \oplus x_1 y_2 \oplus x_1 y_3 \oplus x_1 y_4 \oplus \dots}_{q_1} \\
& \underbrace{x_2 y_0 \oplus x_2 y_1 \oplus \boldsymbol{x_2 y_2} \oplus x_2 y_3 \oplus x_2 y_4 \oplus \dots}_{q_2} \\
& \quad \vdots \qquad \vdots \qquad \vdots \qquad \vdots \qquad \vdots
\end{aligned}
\tag{1.1}
$$

Each row of this formula stands for one component function calculating one share of the output q. The multiplications in the diagonal (bold) are the inner-domain multiplication terms containing only shares from one specific domain and hence only leak about shares of this domain. The cross-domain products do not leak more information on the inputs x and y than probing of one share of x of one share of y directly. Hence, with this formula the sharing for the *calculation* step for the GF multiplier is secure in the d-probing model and can be realized for arbitrary numbers of shares. An example for a second-order DOM multiplier is given in Figure 1.2.

Resharing: A core property for the generalization of the DOM scheme is how the required fresh random r shares can be distributed among the cross-domain terms efficiently and correctly. From Equation 1.1 it can be seen that there are exactly $d(d + 1)$ cross-domain terms which need to be reshared. It is also important to note that there are exactly two product terms that combine shares from two given domains. For example, shares from domain 0 and 1 are only combined in the terms x_0y_1 and x_1y_0. In DOM, we use the same fresh share for product terms that combine shares from the same domain, see Equation 1.2. Hence, we use $d(d+1)/2$ fresh shares for a d^{th}-order DOM implementation of the multiplier (like Ishai et al. [Ish+03]).

Since no probing of any intermediate value created in the *calculation* phase contains more than one share of each input variable x or y, and we only add fresh random shares to the cross-domain terms in the *resharing* phase, no advantage to a d-probing attacker is given during these phases.

Integration: During the integration phase, the multiplication terms of each component function are added up at the output of the domain. Since a digital designer has no influence on the sequence in which these terms are added up, the higher-order DOM multiplier needs to provide security for each possible partial sum of these terms. In particular, it has to be taken care that each of the possible partial sums an attacker could probe only reveals the shares of the domain it is probed in. This is ensured by the resharing shown in Equation 1.2, where each r share is only reused for cross-domain multiplication terms with the same domain associations.

In order to exploit the reuse, it would be necessary to probe the two component functions that use the terms with the reused share. However, the two component functions that use the two terms are associated with the same domains as the terms in the cross-domain products. Hence, there is no advantage for the attacker due to the reuse of the r shares.

For example, share r_0 in Figure 1.2 is used on the terms x_0y_1 and x_1y_0 and these two terms only occur in the component functions for q_0 and q_1. An attacker that probes any partial sum of the terms of q_0 learns only about shares in domain 0. When probing any partial sum of the terms q_1, there is only information about shares of domain 1. A second-order attacker that learns about partial sums of q_0 and q_1 learns about shares from domains 0 and 1 in any case. The fact that partial products x_0y_1 and x_1y_0 reuse r_0 does not provide any advantage to an attacker.

Based on Equation 1.2, the fact that a DOM multiplier fulfills d^{th}-order security can be verified visually. In this multiplication matrix, the diagonal terms are formed by the inner-domain product terms. These inner-domain terms also divide the multiplication matrix into an upper and lower triangular matrix in which each of the fresh random r shares is used exactly once. The triangle formed by the r shares is mirrored along the diagonal. The mirroring of the r shares ensures that each possible combination of partial sums from any two component functions removes at most one fresh random share, and reveals only the inner-domain shares of both domains. As this applies for all combinations of partial sums of all different domains, an attacker restricted to d probes obtains at most d shares per variable. The higher order multiplier is thus secure in the d-probing model.

$$
\begin{array}{rlllll}
\mathrm{q}_0 = & \boldsymbol{x_0y_0} & \oplus\, (x_0y_1 \oplus r_0) & \oplus\, (x_0y_2 \oplus r_1) & \oplus\, (x_0y_3 \oplus r_3) & \oplus\, (x_0y_4 \oplus r_6) \oplus \dots \\
\mathrm{q}_1 = & (x_1y_0 \oplus r_0) \oplus & \boldsymbol{x_1y_1} & \oplus\, (x_1y_2 \oplus r_2) & \oplus\, (x_1y_3 \oplus r_4) & \oplus\, (x_1y_4 \oplus r_7) \oplus \dots \\
\mathrm{q}_2 = & (x_2y_0 \oplus r_1) & \oplus\, (x_2y_1 \oplus r_2) \oplus & \boldsymbol{x_2y_2} & \oplus\, (x_2y_3 \oplus r_5) & \oplus\, (x_2y_4 \oplus r_8) \oplus \dots \\
\mathrm{q}_3 = & (x_3y_0 \oplus r_3) & \oplus\, (x_3y_1 \oplus r_4) & \oplus\, (x_3y_2 \oplus r_5) \oplus & \boldsymbol{x_3y_3} & \oplus\, (x_3y_4 \oplus r_9) \oplus \dots \\
\mathrm{q}_4 = & (x_4y_0 \oplus r_6) & \oplus\, (x_4y_1 \oplus r_7) & \oplus\, (x_4y_2 \oplus r_8) & \oplus\, (x_4y_3 \oplus r_9) \oplus & \boldsymbol{x_4y_4} \oplus \dots \\
 & \vdots & \vdots & \vdots & \vdots & \vdots
\end{array}
\tag{1.2}
$$

The component functions of the multiplication matrix can also be written in closed form as shown in Equation 1.3.

$$
\mathrm{q_i} = x_iy_i \oplus \sum_{j>i}^{d} (x_iy_j \oplus r_{(i+j(j-1)/2)}) \oplus \sum_{j<i}^{d} (x_iy_j \oplus r_{(j+i(i-1)/2)}) \tag{1.3}
$$

This equation is the basis for the scalable hardware designs in Part II.

1.3 Summary

In this chapter, we introduced the DOM scheme which allows generic protection against d^{th}-order SCA. The main idea of DOM is to transform an unprotected circuit into a masked circuit by splitting not only the input and output variables of the circuit, but also the circuit itself, into $d+1$ domains. Each domain (or subcircuit) thus has only one share per masked variable as input. This trivially ensures security in the d-probing model for circuits that contain only linear operations. For the calculation of nonlinear operations, however, it is also required to combine shares from different domains in a secure manner. For this purpose, we introduced a secure and scalable DOM $GF(2^n)$ multiplier, which for $GF(2)$ corresponds to a securely masked AND gate and can thus be used to implement arbitrary circuits. The DOM multiplier is secure under the assumption of independently shared inputs, which is however not a strong practical limitation as we will demonstrate in Part II of this thesis. For example, the calculation of x^2 which is often required in practice is a linear operation in $GF(2^n)$ and does not require a DOM multiplier at all. Share independence is broken by linear combination of independently shared variables, for example, $(x \oplus y) \cdot x$ is secure in DOM as long as a register is used for the calculation of $x \oplus y$ to hinder glitch propagation. Furthermore, the DOM multiplier only outputs shares from and to the same domain (or securely remasked share combinations), and also completely breaks the share dependence to the input variables of the multiplier after an additional register stage at the output. The additional register ensures that all of the randomness used in the DOM multiplier is added to the signals before further propagation. This also ensures independence to variables used in any linear operations that are performed directly at the output of the multiplier with respect to the share domains (between the output of the multiplier and an additional register stage). However, this additional register stage is not always required between cascaded DOM multipliers as long as share independence can be ensured, for example, for calculating $x \cdot y \cdot z$ under the assumption that all variables are independently shared. A generic strategy for avoiding register stages (even inside DOM multipliers) is introduced in Chapter 3. Before this, we discuss how randomness can be saved in DOM for the cost of more register stages in the next chapter.

2

Unified Masking (UMA)

In this chapter, we combine the DOM approach with the most randomness-efficient masking approaches from software in a unified masking approach (UMA). The basis of the generic UMA algorithm is the algorithm of Barthe et al. [Bar+17b] which we combine with DOM. The randomness requirements of UMA are in all cases less or equal to generic software masking approaches. As a non-generic optimization, for the second protection order, we also take into account the solution of Belaïd et al. [Bel+16]. We then show how the UMA algorithm can be efficiently ported to hardware and thereby reduce the asymptotic randomness costs from $d(d+1)/2$ to $d(d+1)/4$. For this purpose, we analyze the parts of the algorithm that are susceptible to glitches and split the algorithm into smaller independent hardware modules that can be calculated in parallel. As a result, the latency in hardware is at most five cycles. Finally, we compare the implementation costs and randomness requirements of UMA to the costs of DOM and point out practical differences.

2.1 Randomness Gap in Hardware and Software

As a starting point of our randomness considerations we recapitulate the multiplication matrix of the DOM scheme for independently shared inputs. For the moment, however, we do not consider registers but instead assume a software implementation for which all stated equations are evaluated from left to right and with respect to the parentheses. An example for the multiplication of the independently shared variables a and b is given in Equation 2.1.

$$
\begin{aligned}
q_0 &= a_0b_0 \oplus (a_0b_1 \oplus r_0) \oplus (a_0b_2 \oplus r_1) \oplus (a_0b_3 \oplus r_3) \oplus (a_0b_4 \oplus r_6) \dots \\
q_1 &= (a_1b_0 \oplus r_0) \oplus a_1b_1 \oplus (a_1b_2 \oplus r_2) \oplus (a_1b_3 \oplus r_4) \oplus (a_1b_4 \oplus r_7) \dots \\
q_2 &= (a_2b_0 \oplus r_1) \oplus (a_2b_1 \oplus r_2) \oplus a_2b_2 \oplus (a_2b_3 \oplus r_5) \oplus (a_2b_4 \oplus r_8) \dots \\
q_3 &= (a_3b_0 \oplus r_3) \oplus (a_3b_1 \oplus r_4) \oplus (a_3b_2 \oplus r_5) \oplus a_3b_3 \oplus (a_3b_4 \oplus r_9) \dots \\
q_4 &= (a_4b_0 \oplus r_6) \oplus (a_4b_1 \oplus r_7) \oplus (a_4b_2 \oplus r_8) \oplus (a_4b_3 \oplus r_9) \oplus a_4b_4 \dots \\
&\dots
\end{aligned} \tag{2.1}
$$

This shared multiplication requires $d(d+1)/2$ fresh random bits which results from the fact that multiplication terms where a and b have the same index (inner-domain terms) do not require fresh randomness, and for the remaining cross-domain terms the same random bit is used as for terms with mirrored indices. The DOM algorithm is especially suited for hardware implementations because it does not require control over the order in which the (remasked) multiplications terms are summed up in each domain, which would require additional registers. However, by introducing a better control over the order in which these terms are summed up (more registers or using software implementations) the randomness requirement can be lowered as demonstrated by the software masking algorithm of Barthe et al. , for instance.

2.1.1 Barthe et al.'s Algorithm

The essence of Barthe et al.'s algorithm is that by grouping the multiplication terms and the used random bits in a certain way, the randomness can be reused securely and in a more efficient manner than in Equation 2.1. Please note that for the moment we just consider the original software implementation of the algorithm for which all stated equations are evaluated from left to right. Parentheses indicating registers are thus omitted.

Instead of using one random bit to protect two mirrored terms (a_ib_j and a_ib_j, where $i \neq j$) as in Equation 2.1, the same fresh random bit can be used again to protect another pair of mirrored terms. The multiplication matrix for the shared q can thus be written according to Equation 2.2 for $d = 4$ as an example. Here, the random bit r_0 is used to secure the absorption of the terms a_0b_1 and a_1b_0 in q_0 as well as for the terms a_4b_1 and a_1b_4 in q_4.

$$
\begin{aligned}
q_0 &= a_0b_0 \oplus r_0 \oplus a_0b_1 \oplus a_1b_0 \oplus r_1 \oplus a_0b_2 \oplus a_2b_0 \\
q_1 &= a_1b_1 \oplus r_1 \oplus a_1b_2 \oplus a_2b_1 \oplus r_2 \oplus a_1b_3 \oplus a_3b_1 \\
q_2 &= a_2b_2 \oplus r_2 \oplus a_2b_3 \oplus a_3b_2 \oplus r_3 \oplus a_2b_4 \oplus a_4b_2 \\
q_3 &= a_3b_3 \oplus r_3 \oplus a_3b_4 \oplus a_4b_3 \oplus r_4 \oplus a_3b_0 \oplus a_0b_3 \\
q_4 &= a_4b_4 \oplus r_4 \oplus a_4b_0 \oplus a_0b_4 \oplus r_0 \oplus a_4b_1 \oplus a_1b_4
\end{aligned} \tag{2.2}
$$

A vectorized version of Barthe et al.'s algorithm is given in Equation 2.3 where all operations are performed share-wise from left to right and bold letters indicate a vector of shares ($\boldsymbol{q} = \{q_0, q_1, \dots, q_d\}$). Accordingly, the vector multiplication is the multiplication of the shares with the same share index, e.g. $\boldsymbol{ab} = \{a_0b_0, a_1b_1, \ldots, a_db_d\}$. Additions in the subscript indicate an index offset of the vector modulo $d+1$ which equals a rotation of the vector elements inside the vector, e.g. $\boldsymbol{a}_{+1} = \{a_1, a_2, \ldots, a_0\}$. Superscript indices refer to different and independent randomness vectors with a size of $d+1$ random bits for each vector.

$$\begin{aligned}\boldsymbol{q} = \boldsymbol{ab} \oplus \boldsymbol{r^0} \oplus \boldsymbol{ab}_{+1} \oplus \boldsymbol{a}_{+1}\boldsymbol{b} \oplus \boldsymbol{r^0}_{+1} \oplus \boldsymbol{ab}_{+2} \oplus \boldsymbol{a}_{+2}\boldsymbol{b}\\ \oplus \boldsymbol{r^1} \oplus \boldsymbol{ab}_{+3} \oplus \boldsymbol{a}_{+3}\boldsymbol{b} \oplus \boldsymbol{r^1}_{+1} \oplus \boldsymbol{ab}_{+4} \oplus \boldsymbol{a}_{+4}\boldsymbol{b}\\ \oplus \boldsymbol{r^2} \oplus \boldsymbol{ab}_{+5} \oplus \boldsymbol{a}_{+5}\boldsymbol{b} \oplus \boldsymbol{r^2}_{+1} \oplus \boldsymbol{ab}_{+6} \oplus \boldsymbol{a}_{+6}\boldsymbol{b} \ldots\end{aligned} \tag{2.3}$$

At the beginning of the algorithm, the shares of $\boldsymbol{q}$ are initialized with the terms resulting from the share-wise multiplication $\boldsymbol{ab}$. Then there begins a repeating sequence that ends when all multiplication terms are absorbed inside one of the shares of $\boldsymbol{q}$. The first sequence starts with the addition of the random bit vector $\boldsymbol{r^0}$. Then a multiplication term and mirrored term pair (a_ib_j and a_jb_i, where $i \neq j$) is added, before the rotated $\boldsymbol{r^0}_{+1}$ vector is added followed by the next pair of terms. The next (up to) four multiplication terms are absorbed using the same sequence but with a new random bit vector $\boldsymbol{r^1}$. This procedure is repeated until all multiplication terms are absorbed. There are thus $\lceil \frac{d}{4} \rceil$ random vectors required with a length of $d+1$ bits each. So, in total the randomness requirement is $\lceil \frac{d}{4} \rceil (d+1)$. In any case, for the last sequence Barthe et al.'s algorithm introduces a new randomness vector that is added once normally and once rotated by one element. This makes the randomness usage of the last sequence less efficient if the last sequence uses less than four multiplication terms per share of $\boldsymbol{q}$. We shall take this up again in Section 2.2 in order to extend Barthe et al.'s algorithm by making it more randomness-efficient.

2.1.2 Randomness Bounds and Optimal Solutions

Barthe et al.'s generic algorithm, even though it has the best asymptotic randomness requirement so far, is known to be not optimal in all cases. The work of Belaïd et al. [Bel+16] proves a lower bound for the randomness requirement of masked multiplications being $d+1$ for $d \leq 3$ (and d for the cases $d \leq 2$). This work also introduces a generic masking algorithm along with some brute-force searched "optimal" solutions which achieve the stated randomness bound.

Nevertheless, it remains unclear whether or not this stated randomness bound is tight. Furthermore, this lower randomness bound (for $d > 1$) is so far only reached by Belaïd et al. 's brute-force searched optimal solutions up to order 4, and in the case of ($d = 3$ and $d = 4$) by Barthe et al.'s generic algorithm. For protection orders above $d = 4$, there is no algorithm known to reach this lower bound. In addition, Barthe et al.'s algorithm requires one fresh random bit more than Belaïd et al.'s generic algorithm in case $d = 1 \mod 4$.

The randomness requirement is depicted in Figure 2.1 for which the so far least randomness demanding masked multiplication algorithms in software were merged together, and then compared to the so far least randomness demanding

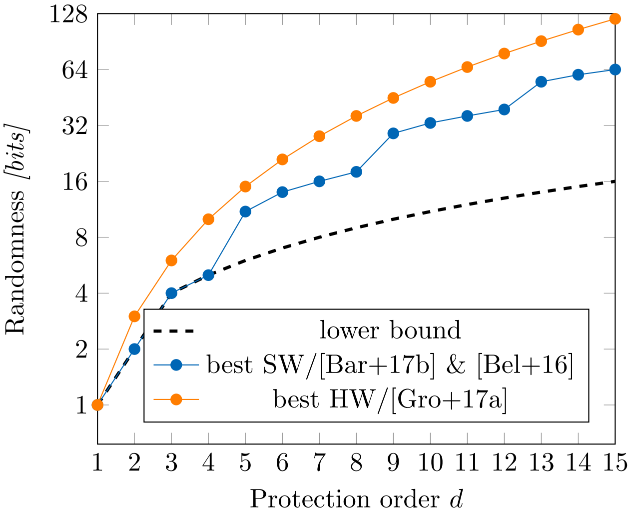

Figure 2.1: Randomness requirements for the best known masked multiplication algorithms

masked multiplication algorithm suitable for hardware implementations (DOM). As it shows, the gap between the state of the art in software and hardware-based masking is quite significant, and in some cases it is as wide as twice the amount of random bits required for software (for $d = 0 \mod 4$).

In the next sections, we introduce the UMA multiplication algorithm that closes the randomness gap between software and hardware-based masking. In addition, the UMA algorithm also lowers the randomness requirement for software for $d = 1 \mod 4$.

2.2 Unified Masked Multiplication in Software

For the assembly of the UMA algorithm, we extend Barthe et al.'s algorithm with optimizations from Belaïd et al. and DOM. We differentiate between four cases for handling the last sequence in Barthe et al.'s algorithm: (1) if the protection order d is an integral multiple of 4, we call the last sequence *complete*, (2) if $d \equiv 3 \mod 4$, we call it *pseudo-complete*, (3) if $d \equiv 2 \mod 4$, we call it *half-complete*, and (4) if $d \equiv 1 \mod 4$, we call it *incomplete*. We first introduce each case briefly before giving a full algorithmic description of the whole algorithm.

Complete and Pseudo-Complete. Complete and pseudo complete sequences are treated according to Barthe et al.'s algorithm. In contrast to the complete sequence, the pseudo-complete sequence contains only three multiplication terms per share of $\boldsymbol{q}$. See the following example for $d = 3$:

$$\boldsymbol{q} = \boldsymbol{ab} \oplus \boldsymbol{r^0} \oplus \boldsymbol{ab}_{+1} \oplus \boldsymbol{a}_{+1}\boldsymbol{b} \oplus \boldsymbol{r^0_{+1}} \oplus \boldsymbol{ab}_{+2}$$

Half-Complete. Half-complete sequences contain two multiplication terms per share of $\boldsymbol{q}$. For handling this sequence, we consider two different optimizations. The first optimization requires d fresh random bits and is hereafter referred to as Belaïd's optimization because it is the non-generic solution in [Bel+16] for the $d = 2$ case. An example for Belaïd's optimization is given in Equation 2.4. The trick to save randomness here is to use the accumulated randomness used for the terms in the first functions in order to protect the last function of $\boldsymbol{q}$. It needs to be ensured that r_0^0 is added to r_1^0 before the terms a_2b_0 and a_0b_2 are added.

$$\begin{aligned} q_0 &= a_0b_0 \oplus r_0^0 \oplus a_0b_1 \oplus a_1b_0 \\ q_1 &= a_1b_1 \oplus r_1^0 \oplus a_1b_2 \oplus a_2b_1 \\ q_2 &= a_2b_2 \oplus r_0^0 \oplus r_1^0 \oplus a_2b_0 \oplus a_0b_2 \end{aligned} \tag{2.4}$$

Unfortunately, Belaïd's optimization cannot be generalized to higher orders to the best of our knowledge. As a second optimization we thus consider the DOM approach for handling this block which is again generic. DOM requires one addition less for the last $\boldsymbol{q}$ function for $d = 2$ but requires one random bit more than the Belaïd's optimization (see Equation 2.5) and thus the same amount as Barthe et al.'s original algorithm. However, for the hardware implementation the DOM approach saves area in this case because it can be parallelized.

$$\begin{aligned} q_0 &= a_0b_0 \oplus r_0^0 \oplus a_0b_1 \oplus r_2^0 \oplus a_0b_2 \\ q_1 &= a_1b_1 \oplus r_1^0 \oplus a_1b_2 \oplus r_0^0 \oplus a_1b_0 \\ q_2 &= a_2b_2 \oplus r_2^0 \oplus a_2b_0 \oplus r_1^0 \oplus a_2b_1 \end{aligned} \tag{2.5}$$

Incomplete. Incomplete sequences contain only one multiplication term per share of $\boldsymbol{q}$. Therefore, in this case one term is no longer added to its mirrored term. Instead, the association of each term with the shares of $\boldsymbol{q}$ and the usage of the fresh random bits is performed according to the DOM scheme. An example for $d = 1$ is given in Equation 2.6.

$$\begin{aligned} q_0 &= a_0b_0 \oplus r_0^0 \oplus a_0b_1 \\ q_1 &= a_1b_1 \oplus r_0^0 \oplus a_1b_0 \end{aligned} \tag{2.6}$$

2.2.1 Full Description of UMA

Algorithm 1 shows the pseudo-code of the proposed UMA algorithm. The inputs of the algorithm are the two operands $\boldsymbol{a}$ and $\boldsymbol{b}$ split into $d + 1$ shares each. The randomness vector $\boldsymbol{r}^*$ (we use $*$ to make it explicit that $\boldsymbol{r}$ is a vector of

Algorithm 1 : UMA multiplication algorithm

```
Input: $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{r}^*$
Output: $\boldsymbol{q}$
    Initialize q:
 1: $\boldsymbol{q} = \boldsymbol{ab}$
    Handle complete sequences:
 2: for $i = 0 < \lfloor d/4 \rfloor$ do
 3:     $\boldsymbol{q} \oplus= \boldsymbol{r}^i \oplus \boldsymbol{ab}_{+2i+1} \oplus \boldsymbol{a}_{+2i+1}\boldsymbol{b} \oplus \boldsymbol{r}^i_{+1} \oplus \boldsymbol{ab}_{+2i+2} \oplus \boldsymbol{a}_{+2i+2}\boldsymbol{b}$
 4: end for
    Handle last sequence:
 5: $l = \lfloor d/4 \rfloor$
      Pseudo-complete sequence:
 6: if $d \equiv 3 \mod 4$ then
 7:     $\boldsymbol{q} \oplus= \boldsymbol{r}^l \oplus \boldsymbol{ab}_{+2l+1} \oplus \boldsymbol{a}_{+2l+1}\boldsymbol{b} \oplus \boldsymbol{r}^l_{+1} \oplus \boldsymbol{ab}_{+2l+2}$
        Half-complete sequence:
 8: else if $d \equiv 2 \mod 4$ then
 9:     if $d = 2$ then
10:         $\boldsymbol{z} = \{r^l_0, r^l_1, r^l_0 \oplus r^l_1\}$
11:         $\boldsymbol{q} \oplus= \boldsymbol{z} \oplus \boldsymbol{ab}_{+2l+1} \oplus \boldsymbol{a}_{+2l+1}\boldsymbol{b}$
12:     else
13:         $\boldsymbol{q} \oplus= \boldsymbol{r}^l \oplus \boldsymbol{ab}_{+2l+1} \oplus \boldsymbol{r}^l_{+2l+2} \oplus \boldsymbol{ab}_{+2l+2}$
14:     end if
        Incomplete sequence:
15: else if $d \equiv 1 \mod 4$ then
16:     $\boldsymbol{z} = \{\boldsymbol{r}^l, \boldsymbol{r}^l\}$
17:     $\boldsymbol{q} \oplus= \boldsymbol{z} \oplus \boldsymbol{ab}_{+2l+1}$
18: end if
19: return $q$
```

vectors) contains $\lceil d/4 \rceil$ vectors with $d+1$ random bits each. Please note that all operations, including the multiplication and the addition, are again performed share-wise from left to right.

At first, the return vector q is initialized with the multiplication terms that have the same share index for a and b in Line 1. In Line 2 to 4, the *complete* sequences are calculated according to Barthe et al.'s original algorithm. We use the superscript indices to address specific vectors of r^* and use again subscript indices for indexing operations on the vector. Subscript indices with a leading "+" denote a rotation by the given offset.

From Line 5 to 17, the handling of the remaining multiplication terms is performed according to the description above for the *pseudo-complete*, *half-complete*, and *incomplete* cases. In order to write this algorithm in quite compact form, we made the assumption that for the last random bit vector $\boldsymbol{r}^l$ only the required random bits are provided. In Line 10 where Belaïd's optimization is used for $d = 2$, a new bit vector $\boldsymbol{z}$ is formed that consists of the concatenation of the two elements of vector $\boldsymbol{r^l}$ and the sum of these bits. So, in total the $\boldsymbol{z}$ vector is again $d+1$ (three) bits long. In a similar way, we handle the randomness in Line 16. We concatenate two copies of $\boldsymbol{r^l}$ of the length $(d+1)/2$ to form $\boldsymbol{z}$ which is then added to the remaining multiplication terms.

Randomness requirements. Table 2.1 shows a comparison of the randomness requirements of UMA with other masked multiplication algorithms. The comparison shows that UMA requires the least amount of fresh randomness in all generic cases. With the non-generic optimization by Belaïd et al., the algorithm reaches lower bounds of $d+1$ for $d>2$ and of d for $d \leq 2$ below the fifth protection order.

Table 2.1: Randomness requirement comparison

d	**UMA**	Barthe et al.	Belaïd et al.	DOM
1	**1**	2	**1**	**1**
2	3 (**2**[1])	3	3 (**2**[1])	3
3	**4**	**4**	5 (**4**[1])	6
4	**5**	**5**	8 (**5**[1])	10
5	**9**	12	11	15
6	**14**	**14**	15	21
7	**16**	**16**	19	28
8	**18**	**18**	24	36
9	**25**	30	29	45
10	**33**	**33**	35	55
11	**36**	**36**	41	66
12	**39**	**39**	48	78
13	**49**	56	55	91
14	**60**	**60**	63	105
15	**64**	**64**	71	120

[1] non-generic solution

Compared to Barthe et al.'s original algorithm, UMA saves random bits in the cases where the last sequence is *incomplete*. More importantly, since we target efficient higher-order masked hardware implementations, UMA has much lower randomness requirements than the original DOM scheme. Up to half of the randomness costs can thus be saved compared to DOM. In the next section we show how UMA can be securely and efficiently implemented in hardware.

2.3 UMA in Hardware

Directly porting UMA to hardware by emulating what a processor would do, i.e. ensuring the correct order of instruction execution by using registers in between every operation, would introduce a tremendous area and performance overhead over existing hardware masking approaches. To make this algorithm more efficient while still keeping it secure in hardware, it needs to be sliced into smaller portions of independent code parts than can be translated to hardware modules which can be evaluated in parallel.

Inner-domain block. The assignment of the inner-domain terms ($\boldsymbol{q} = \boldsymbol{ab}$) in Line 1 of Algorithm 1 can thus be considered uncritical in terms of d^{th}-order probing security. Only shares with the same share index are multiplied and stored at the same index position of the share in $\boldsymbol{q}$. The *inner-domain* block is depicted in Figure 2.2 and consist of $d + 1$ AND gates that are evaluated in parallel. Hence, each share stays in its respective share domain. So even if the sharings of the inputs of a and b were the same, this block would not compromise the security because neither a_0a_0 nor b_0b_0, for example, would provide any additional information on a or b. We can thus combine the *inner-domain* block freely with any other secure masked component that ensures the same domain separation.

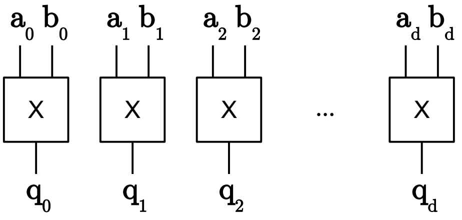

Figure 2.2: Inner-domain block

(Pseudo-)Complete blocks. For the security of the implementation in hardware, the order in which the operations in Line 3 (and Line 7) are performed is essential. Since the calculation of one *complete* sequence is subdivided by the addition of the random vector in the middle of this code line, it is quite tempting to split this calculation into two parts and to parallelize them in order to speed up the calculation.

However, if we consider Equation 2.3, and omit the inner domain-terms that would have already been calculated in a separate inner-domain block, a probing attacker could get (through glitches) the intermediate results from the probe $p_0 = r_0 \oplus a_0b_1 \oplus a_1b_0$ from the calculation of q_0 and $p_1 = r_0 \oplus a_4b_1 \oplus a_1b_4$ from the calculation of q_4. By combining the probed information from p_0 and p_1 the attacker would already gain information on three shares of a and b. With the remaining two probes, the attacker could just probe the missing shares of a or b to fully reconstruct them. The *complete* sequence and for the same reasons also the *pseudo-complete* sequence can thus not be further parallelized.

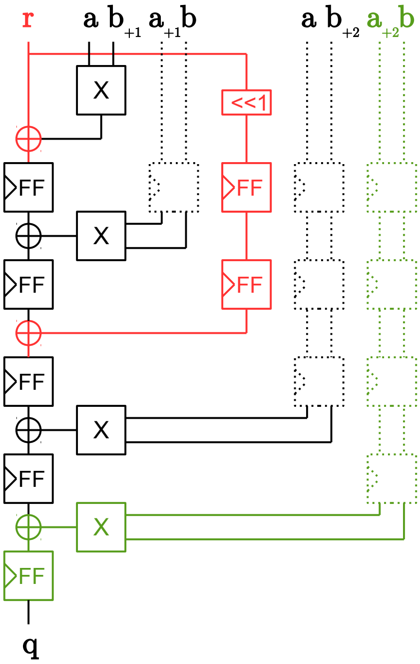

Figure 2.3: Complete block

Figure 2.3 shows the vectorized *complete* block that consists of five register stages. Optional pipeline registers are depicted with dotted lines where necessary that make the construction more efficient in terms of throughput. For the *pseudo-complete* block, the last XOR is removed and the most right multiplier including the pipeline registers before the multiplier (marked green).

The security of this construction has already been analyzed by Barthe et al. [Bar+17b] in conjunction with the inner-domain terms (which have no influence on the probing security) and for subsequent calculation of the sequences. Since the scope of the randomness vector is limited to one block only, a probing attacker does not gain any advantage (information on more shares than probes used) by combining intermediate results of different blocks, even if they are calculated in parallel. Furthermore, each output of these blocks is independently and freshly masked and separated in $d + 1$ domains which allows for the combination with other blocks.

Half-complete block. Figure 2.4 shows the construction of the *half-complete* sequence in hardware when Belaïd's optimization is used for $d = 2$. The creation of the random vector $\boldsymbol{z}$ requires one register and one XOR gate. The security of this construction was formally proven by Belaïd et al. in [Bel+16]. For protection orders other than $d = 2$, we use instead the same DOM construction as for the incomplete block.

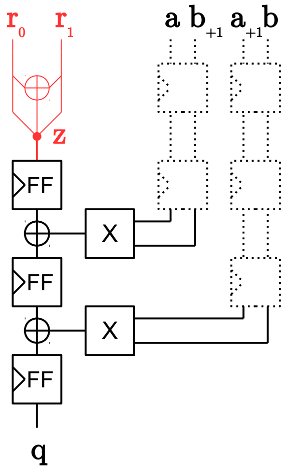

Figure 2.4: Half-complete block (Belaïd's opt.)

Incomplete block. For the *incomplete* block (and the half-complete block without Belaïd's optimization) each random bit is only used to protect one multiplication term and its mirrored term. The term and the mirrored term are distributed in different domains to guarantee probing security. Figure 2.5 shows the construction of an *incomplete* block following the construction principles of DOM for two bits of $\boldsymbol{q}$ at the same time. For *half-complete* blocks (without Belaïd's optimization), two instances of the *incomplete* constructions are used with different indexing offsets and the resulting bits are added together (see Line 13). No further registers are required for the XOR gate at the output of this construction because it is ensured by the registers that all multiplication terms are remasked by r before the results are added.

Assembling the UMA AND Gate. Figure 2.6 shows how the UMA AND gate is composed from the aforementioned building blocks. Except for the *inner-domain* block which is always used, all other blocks are instantiated and connected depending on the given protection order which allows for a generic construction of the masked AND gate from $d = 0$ (no protection) to any desired protection order. Connected to the *inner-domain* block, there are $\lfloor \frac{d}{4} \rfloor$ *complete* blocks, and either one or none of the *pseudo-complete*, *half-complete*, or *incomplete* blocks.

Table 2.2 gives an overview of the hardware costs of the different blocks that form the masked AND gate. All stated gate counts need to be multiplied by

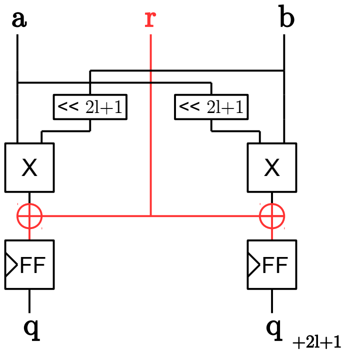

Figure 2.5: Incomplete block

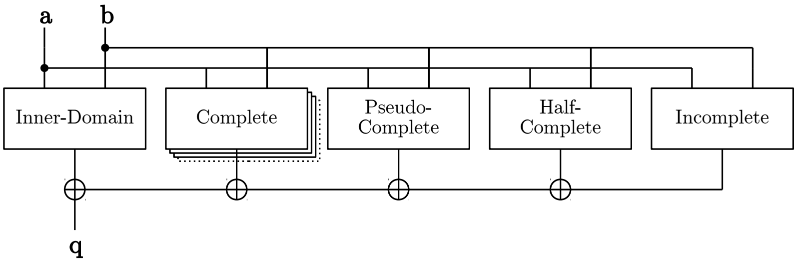

Figure 2.6: Fully assembled UMA AND gate

the number of shares $(d+1)$. The XOR gates which are required for connecting the different blocks are accounted to the *inner-domain* block. In case pipelining is used, the input shares of a and b are pipelined instead of pipelining the multiplication results inside the respective blocks. The required pipelining registers for the input shares are also added to the *inner-domain* block's register requirements, since this is the only fixed block of every masked AND gate. The number of pipelining registers is determined by the biggest latency required for one block. In case one or more *complete* blocks are instantiated, there are always five register stages required which gives a total amount of $10(d+1)$ input pipelining registers. However, for $d < 4$ the number of input pipelining registers is always twice the amount of cycles for the instantiated block which could also be zero for the unprotected case where the masked AND gate consists only of the *inner-domain* block. The *inner-domain* block itself does not require any registers except for the pipelining case and thus has a latency of zero.

For the cost calculation of the UMA AND gate, the gate counts for the *complete* block need to be multiplied by the number of instantiated *complete* blocks ($\lfloor \frac{d}{4} \rfloor$) and the number of shares $(d+1)$. The other blocks are instantiated at most once. The *pseudo-complete* block in case $d \equiv 3 \mod 4$, the *half-complete*

Table 2.2: Overview of the hardware costs of different blocks

Block	AND	XOR	FF $\cdot(d+1)$		Latency
	$\cdot(d+1)$	$\cdot(d+1)$	w/o pipel.	pipelined	[Cycles]
Inner-domain	1	$\lceil \frac{d}{4} \rceil$	0	$0-10$	0
Complete	4	5	5	7	5
Pseudo-complete	3	4	4	6	4
Half-complete:					
Belaïd's optimization	2	$2+\frac{1}{3}$	3	3	3
DOM	2	3	2	2	1
Incomplete	1	1	1	1	1

block in case $d \equiv 2 \mod 4$ (where Belaïd's optimization is only used for $d = 2$), and the *incomplete* block in case $d \equiv 1 \mod 4$.

Comparison with DOM. Table 2.3 shows a comparison of the UMA AND gate with a masked AND gate from the DOM scheme. For the generation of these numbers, we used Table 2.2 to calculate the gate counts for the UMA AND gate. For DOM, we have $(d+1)^2$ AND gates, $2d(d+1)$ XOR gates, and $(d+1)^2$ registers ($-d-1$, for the unpipelined variant). For calculating the gate equivalence, we used the 90 nm UMC library from Faraday as reference. Accordingly, a two-input AND gate requires 1.25 GE, an XOR gate 2.5 GE, and a D-type flip-flop with asynchronous reset 4.5 GE.

Since in both implementations AND gates are only used for creating the multiplication terms, both columns for the UMA AND gate construction and the DOM AND are equivalent. The gate count for the XORs in the UMA implementation is lower than for the DOM gate which results from the reduced randomness usage compared to DOM. The reduced XOR count almost compensates for the higher register usage in the unpipelined case. The difference for the 15^{th} order, for example, is still only 8 GE. However, the latency of the UMA AND gate is in contrast to the DOM AND gate, except for $d = 1$, not always one cycle but increases up to five cycles. Therefore, in the pipelined implementation more registers are necessary, which results in an increasing difference in the required chip area for higher protection orders.

Practical Differences to DOM. There is another very important practical difference between the DOM and UMA masked AND gates regarding their security. While both masked AND gates are secure in the probing model under the assumption of independently shared inputs, the reduced amount of randomness required for the UMA variant does not achieve a complete separation of the shared input variables at the output, even after the insertion of another register stage. More care has thus to be take in practice when combining UMA gates. For example, the calculation of $(x \cdot y) \cdot x$ is secure when DOM ANDs are used, but not for all protection orders if UMA gates are used. A DOM gate can therefore not always be replaced with a UMA gate in practice. At this point, we want to thank Moos et al. [Moo+18] for pointing out this difference prior to the publishing of this thesis.

Table 2.3: Comparison of the UMA AND gate with DOM

	UMA AND						DOM AND					
d	AND	XOR	Registers		GE		AND	XOR	Registers		GE	
			unpipel.	pipel.	unpipel.	pipel.			unpipel.	pipel.	unpipel.	pipel.
1	4	4	2	6	24	42	4	4	2	4	24	33
2	9	10	9	27	77	157	9	12	6	9	68	82
3	16	20	16	56	142	322	16	24	12	16	134	152
4	25	30	25	85	219	489	25	40	20	25	221	244
5	36	48	36	108	327	651	36	60	30	36	330	357
6	49	70	49	133	457	835	49	84	42	49	460	492
7	64	88	72	184	624	1,128	64	112	56	64	612	648
8	81	108	90	216	776	1,343	81	144	72	81	785	826
9	100	140	110	250	970	1,600	100	180	90	100	980	1,025
10	121	176	132	286	1,185	1,878	121	220	110	121	1,196	1,246
11	144	204	168	360	1,446	2,310	144	264	132	144	1,434	1,488
12	169	234	195	403	1,674	2,610	169	312	156	169	1,693	1,752
13	196	280	224	448	1,953	2,961	196	364	182	196	1,974	2,037
14	225	330	270	510	2,321	3,401	225	420	210	225	2,276	2,344
15	256	368	304	592	2,608	3,904	256	480	240	256	2,600	2,672

3

Low-Latency Masking (LOLA)

In this chapter, we show how to reduce the latency of DOM implementations. From the last two chapters we can conclude that the main causes which hinder the calculation of a DOM masked circuit in fewer clock cycles are: 1) the compression to $d+1$ shares after nonlinear operations (like the DOM multiplier in Figure 1.1) which require registers for the resharing of the cross-domain terms, and 2) the temporary or permanent dependencies (variable collisions) at the inputs of a nonlinear circuit part (cf. Section 1.1, in Chapter 1). Our LOLA approach thus works by skipping the share compression step and avoiding variable collisions at the input of nonlinear functions.

3.1 Compression Skipping

Our main observation is that the resharing and compression to $d+1$ shares (and therefore also the randomness and additional circuitry) is not a necessity from the probing model itself. It is solely performed for practical reasons and to some extent to make the result independent of the shared operands without having an explicit mask refreshing. We extend the domain separation requirement of the DOM approach insofar as we still constrain each domain to use at most one share per variable but with the addition to allow domains with mixed share indices.

For example, the shared multiplication $q = x \cdot y$ without a subsequent resharing and compression step thus results in four share domains for the result variable q (Figure 3.1). Each domain contains only one multiplication term from the calculation step. Any subsequent linear operations on the shares of q that only involve shares that are already used in the respective domain can be performed without violating the probing model. If q is multiplied by another variable, e.g.

z with $d+1$ shares, the number of shares and domains increases to $(d+1)^3$ and so on. To keep this exponential blowup of shares and domains within reasonable bounds, the number of consecutive nonlinear operations needs to be minimized, or otherwise at some point a secure share compression needs to be performed if the blowup becomes unacceptable.

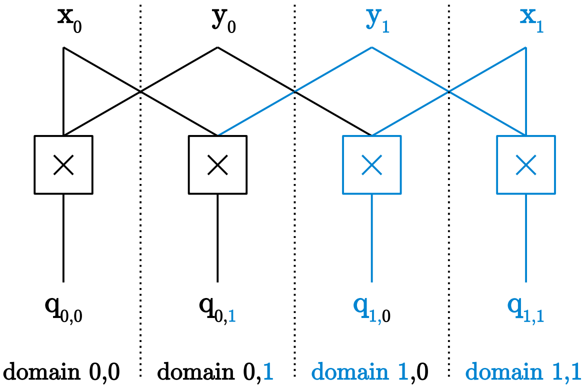

Figure 3.1: First-order LOLA multiplication with compression skipping, resulting in four domains

The security in the probing model for the compression skipping approach is given because any masked circuit that can be divided into at least $d+1$ independent subcircuits (without any wires to the other subcircuits), where each subcircuit uses at most one of the $d+1$ input shares from each variable, requires at least $d+1$ probes to combine all shares of one variable.

3.2 Avoiding Collisions

The up to now tacit assumption that allows for the nonlinear combination of shares without compression and mask refreshing is that all operands have independent sharings. Meaning that it is relatively straightforward to apply this approach to a masked circuit that calculates $x \cdot y \cdot z$ if all involved shares are produced using independent and fresh randomness. The calculation of $(x \cdot y) \cdot x$, on the other hand, requires more attention (see Figure 3.2, left). One of the resulting multiplication terms would be $x_0 \cdot y_0 \cdot x_1$ ($q_{0,0,1}$) which brings two shares of x together and thus violates the domain separation requirement. This circumstance is indicated in Figure 3.2 by the different coloring of the shares for x which when combined result in an insecure sharing (colored red). One approach to circumvent the violation of the probing model, that is used by the threshold implementations scheme for instance, is to use more than $d+1$ shares and to ensure that in the worst case the probing attacker gets access to at most d shares when using up to d probing needles. Efficient sharings that fulfill the

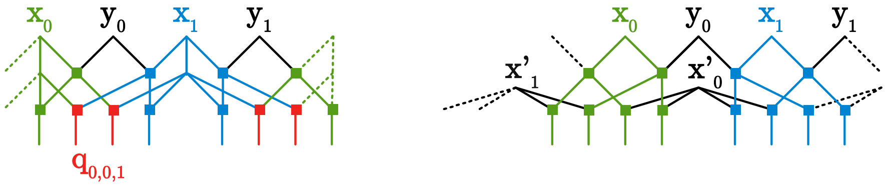

Figure 3.2: Example for an insecure first-order masked circuit calculating $(x \cdot y) \cdot x$ (left), and a secure circuit $(x \cdot y) \cdot x'$ (right). The shares of x are colored green (x_0) and blue (x_1) for clarity reasons

properties required by the TI [Nik+06] scheme (correctness, independence, and uniformity) at the same time are, however, not trivial to find.

Instead of increasing the share count per variable, we propose the duplication of colliding variables (and gates) by using multiple shared instances of the same variable with independent sharings. Instead of calculating $(x \cdot y) \cdot x$, we thus calculate the equivalent $(x \cdot y) \cdot x'$ where $x = \sum_{i=0}^{d} x_i = \sum_{i=0}^{d} x'_i$, and all involved shares are picked independently and uniformly at random. As Figure 3.2 (right) shows, mixing of the shares of x is circumvented this way. While at first sight this may seem as if we were using a sledgehammer to crack a nut, it has the same randomness costs for sharing a variable than e.g. a first-order ($d = 1$) TI with three shares and does not require additional (online) randomness. We would thus share x into $x_0 = x \oplus r_0$ and $x_1 = r_0$, and x' into $x'_0 = x \oplus r_1$ and $x'_1 = r_1$. The probing security for this simple example can be easily observed by writing down all resulting shares $q_{i,j,k} = x_i y_j x'_k$. Since none of the shares of q contain two shares of the same variable, the sharing is secure.

More generally, the probing security of any circuit with $d + 1$ input shares and protection order d is given, if at no point in the circuit there exists a path from one share of a variable to another share of the same variable (assuming that all variables are independently shared).

3.3 Resolving Gate Collisions

When looking at complex circuits, collisions can no longer entirely be resolved by duplication of the input variables. For the purpose of illustration, we consider a purely combinatorial unmasked circuit (Figure 3.3).

We note that collisions that would be caused when a circuit is masked using the DOM scheme are already evident in the unshared circuit. The first collision in this circuit (1) is caused because there exists a path (over other gates) from one input of the circuit to both inputs of a nonlinear gate. Since this collision is directly caused by the circuit input i_3, we could simply duplicate the input that causes the collision as before and connect the copy (i'_3) accordingly (Figure 3.3, right). The second collision is caused by a gate (2) that has a path to both inputs of a nonlinear gate. In this case, simply duplicating the inputs would not be

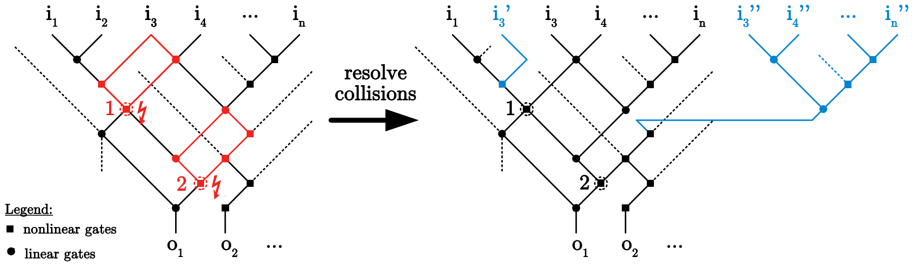

Figure 3.3: Example for collisions directly caused by inputs (1) and collisions caused by gates (2), collisions (left) and resolved collisions (right)

enough to avoid this collision. Instead, the gate causing the collision needs to be duplicated including its entire fan-in circuitry and the respective inputs. The output of the duplicated circuit then needs to be used in one path instead of the output wire of the gate that caused the collision.

In the next sections, we demonstrate the suitability of our LOLA approach on practical examples and discuss trade-offs and possible pitfalls.

3.4 A Low-Latency Ascon S-box

As a first proof of concept, we introduce a masked Ascon S-box that requires a single clock cycle while existing $d + 1$ share implementations [GM17] require at least three clock cycles. The S-box is equivalent to the Keccak S-box except for an affine transformation on the input that produces temporary variable collisions making it a viable first practical example for our approach. We first transform the unshared S-box circuit to free the circuit from variable collisions, and then share the S-box according to our LOLA approach.

Collision-free S-box. The structure of the S-box is depicted in Figure 3.4, which corresponds to Equation 3.1.

$$\begin{aligned}
a'' &= (a \oplus e) \oplus \underline{(\neg b \wedge (b \oplus c))} \oplus (d \oplus e) \oplus (\neg (a \oplus e) \wedge b) \\
b'' &= b \oplus (\neg (b \oplus c) \wedge d) \oplus (a \oplus e) \oplus \underline{(\neg b \wedge (b \oplus c))} \\
c'' &= \neg \Big((b \oplus c) \oplus \underline{(\neg d \wedge (d \oplus e))} \Big) \\
d'' &= d \oplus \underline{(\neg (d \oplus e) \wedge (a \oplus e))} \oplus (b \oplus c) \oplus \underline{(\neg d \wedge (d \oplus e))} \\
e'' &= (d \oplus e) \oplus (\neg (a \oplus e) \wedge b)
\end{aligned} \tag{3.1}$$

By looking at the equations one can observe that there is a variable collision in the AND gates in five cases (underlined parts in Equation 3.1). These are the nonlinear gates that would produce a violation in the probing model due to glitches in case we would share the S-box using DOM. For example $(\neg b \wedge (b \oplus c))$

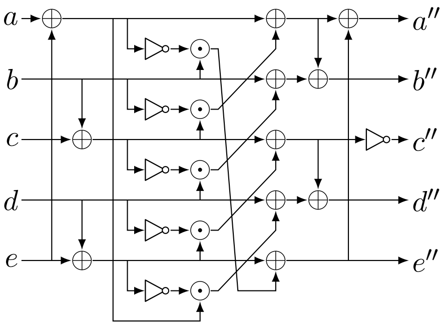

Figure 3.4: Ascon's original S-box, with collisions in a to d

in a'' combines the variable b with itself in an AND gate which would combine shares with different share index (e.g. b_0b_1) when the S-box is shared and thus create a violation.

To avoid collisions in the AND gates we duplicate the signals b, d, and e (b', d', and e') and replace one of the operands of the AND gates accordingly as shown in Equation 3.2.

$$\begin{aligned}
a'' &= (a \oplus e) \oplus (\neg b' \wedge (b \oplus c)) \oplus (d \oplus e) \oplus (\neg\, (a \oplus e) \wedge b) \\
b'' &= b \oplus (\neg\, (b \oplus c) \wedge d) \oplus (a \oplus e) \oplus (\neg b' \wedge (b \oplus c)) \\
c'' &= \neg\, ((b \oplus c) \oplus (\neg d' \wedge (d \oplus e))) \\
d'' &= d \oplus (\neg\, (d \oplus e) \wedge (a \oplus e')) \oplus (b \oplus c) \oplus (\neg d' \wedge (d \oplus e)) \\
e'' &= (d \oplus e) \oplus (\neg\, (a \oplus e) \wedge b)
\end{aligned} \tag{3.2}$$

Sharing of the S-box. Since the S-box description is now free from any variable collisions, the S-box can safely be shared following our LOLA approach. We assume that each of the five inputs and the two duplicated inputs are shared using $d+1$ shares. As there is a single layer of AND gates, the shares and thus the domains for each of the outputs grow from $d+1$ to $(d+1)^2$, and we use two indices (i and j) to denote the according output share.

$$
\begin{aligned}
a''_{i,j} &= \begin{cases} (a_i \oplus e_i) \oplus \left(\neg^i b'_i \wedge (b_i \oplus c_i)\right) \oplus (d_i \oplus e_i) \oplus \left(\neg^i (a_i \oplus e_i) \wedge b_i\right), & \text{if } i = j. \\ \left(\neg^i b'_i \wedge (b_j \oplus c_j)\right) \oplus \left(\neg^i (a_i \oplus e_i) \wedge b_j\right), & \text{otherwise.} \end{cases} \\
b''_{i,j} &= \begin{cases} b_i \oplus \left(\neg^i (b_i \oplus c_i) \wedge d_i\right) \oplus (a_i \oplus e_i) \oplus \left(\neg^i b'_i \wedge (b_i \oplus c_i)\right), & \text{if } i = j. \\ \left(\neg^i (b_i \oplus c_i) \wedge d_j\right) \oplus \left(\neg^j b'_j \wedge (b_i \oplus c_i)\right), & \text{otherwise.} \end{cases} \\
c''_{i,j} &= \begin{cases} \neg^i \left((b_i \oplus c_i) \oplus \left(\neg^i d'_i \wedge (d_i \oplus e_i)\right)\right), & \text{if } i = j. \\ \left(\neg^i d'_i \wedge (d_j \oplus e_j)\right), & \text{otherwise.} \end{cases} \\
d''_{i,j} &= \begin{cases} d_i \oplus \left(\neg^i (d_i \oplus e_i) \wedge (a_i \oplus e'_i)\right) \oplus (b_i \oplus c_i) \oplus \left(\neg^i d'_i \wedge (d_i \oplus e_i)\right), & \text{if } i = j. \\ \left(\neg^i (d_i \oplus e_i) \wedge \left(a_j \oplus e'_j\right)\right) \oplus \left(\neg^j d'_j \wedge (d_i \oplus e_i)\right), & \text{otherwise.} \end{cases} \\
e''_{i,j} &= \begin{cases} (d_i \oplus e_i) \oplus \left(\neg^i (a_i \oplus e_i) \wedge b_i\right), & \text{if } i = j. \\ \left(\neg^i (a_i \oplus e_i) \wedge b_j\right), & \text{otherwise.} \end{cases}
\end{aligned}
\tag{3.3}
$$

For each output variable we consider two cases:

1) The case $i = j$ covers the inner-domain terms where only variables with the same share index appear. To ensure correctness of the sharing, the negation e.g. $\neg^i$ is only effective if the corresponding variable in the superscript equals to zero so that only the first share of a variable is inverted.

2) For the remaining output shares, we need to be more careful to fulfill the domain separation requirement. By the duplication of the according inputs we ensured that there are no two paths for any of the input variables that are combined in a nonlinear AND gate, which would result in a flaw that could not be avoided in this case. However, for linear gates we still need to ensure that we do not combine shares with different share indices from the same variable in the same domain (domain separation requirement). For example $(b'_i \wedge (b_j \oplus c_j)) \oplus ((a_i \oplus e_i) \wedge b_j)$ in a'' would produce a flaw in case we would switch the share index variable of one of the b variables (i to j) in this equation so that we have $(\ldots(b_i \ldots)) \oplus ((\ldots b_j)$. For this reason, we also need to set the indices in b'' and d'' for the last AND gate terms accordingly.

The correctness of the sharing is given by the fact that the sums over i and j over each output variable result in Equations 3.1 when b' is set to b, d' is set to d, and e' is set to e. The security is given by the fact that we do not have any domain crossings.

3.5 A Low-Latency Masked AES S-box

The efficient (masked) implementation of the AES S-box has proven to be a difficult practical problem and a huge variety of papers have been published on efficient S-box constructions. Most of the recent works on masked AES implementations use the S-box design of David Canright [Can05] as basis. The original design goal of Canright's S-box design is low chip area for an unmasked implementation which does not automatically result in the lowest area costs for a side-channel protected implementation. For our LOLA approach, the maximum logic depth and in particular the nonlinear gate depth (number of AND gates or GF multipliers in the logic path) seems to be the natural major design criterion because at each nonlinear gate the number of shares is increased. The S-box design of Boyar and Peralta [BP12] addresses low logic depth which results in a total logic depth of 16 and a nonlinear gate depth of 4. This design was most recently used in another work on low-latency masking by Ghoshal et al. [GC17] with a latency of three to four cycles. Canright's S-box on the other hand has a logic depth of 25 to 27, and a nonlinear gate depth of 4 (in the variant as it is used by most masked implementations). Another important aspect that needs to be taken into account for our approach is the number of bit collisions because it determines the number of input duplicates we need to provide in order to guarantee collision freeness.

Choosing the most promising S-box design. As analyzing a circuit with respect to its collision behavior is rather time-consuming, we developed a tool that simply traces all inputs and gate outputs through a given circuit and checks for conflicts. We analyzed the Canright S-box (original design), the Boyar-Peralta S-box and the design of Edwin NC Mui [Mui07]. As it turns out, even despite the fact that the Boyar-Peralta S-box was designed for low circuit depth, it implies lots of gate dependencies which require quite a number of sub-circuit copies and input copies. Furthermore, the Canright and the Mui S-box designs do not break down the complete design of the AES S-box into single gates but rather consist of larger self-contained structures like Galois field multipliers which can be shared more efficiently than by sharing each AND gate separately. The circuit that showed the fewest dependencies is the design by Mui which we then chose to take it as basis for our own design. However, we note that we do not consider this choice of the S-box or our LOLA implementations of it to be optimal.

Mui's design is depicted in Figure 3.5. The black and red (security-critical) paths correspond to the original design by Edwin NC Mui. The gray dotted circuit elements are used for the collision-free S-box design and replace the red paths. For the design of the S-box without collisions, we took an iterative approach for which we implemented the circuit from the inputs onwards to the next nonlinear part of the circuit and checked for collisions. We thus also split the explanation accordingly.

S-box inputs to inverter. After the input transformation mapping the S-box input x, which is interpreted as a polynomial in $GF(256)$, to two elements in $GF(16)$ the transformed input is split into two halves. The two halves are nonlinearly combined in the $GF(16)$ multiplier. Since the linear input mapping and the XOR in front of the first GF multiplier mixes many of the input bits (cf. [Mui07] for details), it requires to duplicate all bits of x (x') except for the fifth bit and the circuitry that causes the flaw (the gray colored and dotted input mapping). Otherwise, an input collision would be caused in the multiplier as

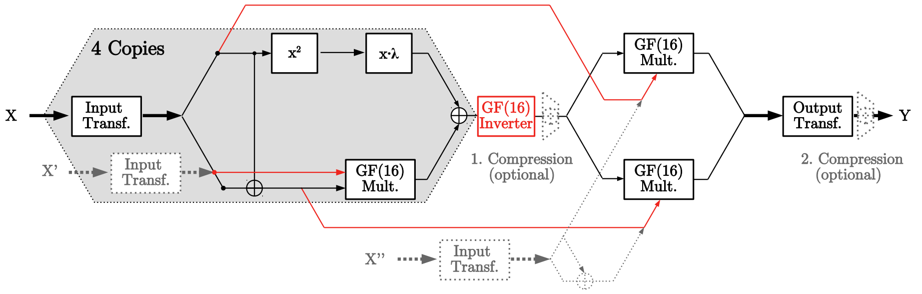

Figure 3.5: Mui S-box design (black and red parts are from the original design), gray dotted paths and elements replace the red paths to which they are connected in the collision-free design

indicated by the red wire. For the shared S-box variant, the number of shares is increased from $d+1$ to $(d+1)^2$ after the multiplier and the linearly transformed parts (x^2 and $x\lambda$) are added with respect to their share domain.

GF(16) inverter. In Mui's S-box design, the $GF(16)$ inverter is given as Boolean equation instead of finite field arithmetic as e.g. in Canrights S-box. The mathematical description is stated in Equation 3.4. The inversion in $GF(16)$ results in collisions for all S-box input bits which requires to separate the calculation of all input bits of the inversion by copying the fan-in circuit (dotted gray hexagon, "4 Copies") four times including the changes as described above. Up to this point, the S-box circuit requires in total four full copies of the input x and four partial copies (x', each bit except for the fifth bit) to avoid collisions.

$$\begin{aligned}
a' &= a \oplus abc \oplus ad \oplus b \\
b' &= abc \oplus abd \oplus ad \oplus b \oplus bc \\
c' &= a \oplus abc \oplus acd \oplus b \oplus bd \oplus c \\
d' &= \underline{abc} \oplus \underline{abd} \oplus ac \oplus \underline{acd} \oplus ad \oplus b \oplus bc \oplus bcd \oplus c \oplus d
\end{aligned} \tag{3.4}$$

In contrast to the Ascon S-box example, the equations for the inverter are free from any internal collisions of the inverter inputs (there is no path from one input variable to both inputs of an AND gate). In order to avoid the combination of two or more shares of one input for the shared S-box representation, care needs to be taken also for the linear gates. Again we avoid collisions in the linear parts by associating with each variable one share index which we keep throughout the entire calculation. To keep the number of output shares to a minimum we try to use as few share indices as possible. However, as can already be observed in the underlined parts of the unshared calculation of d', using only three indices is not always possible.

Reduced example for flawed indexing. To demonstrate the resulting problem for d' in the shared variant, we consider a reduced example that contains only the problematic parts:

$$q = abc \oplus abd \oplus acd$$

If we want to calculate the shared representation of q, we need to combine all shares (given by the indices i, j, and k) of the variables connected by an AND gate as given in the following example. We assume, as for the inverter inputs, that the input share count is already increased to $(d+1)^2$.

$$q_{(i,j,k)} = a_i b_j c_k \oplus a_i b_j d_k \oplus a_i c_j d_k$$

The problem arises in the XOR gates because we combine shares from the same variable c one time with the share index k and another time with index j which violates the mixed domains assumption. Since there is no way to overcome this issue by associating the share indices differently, the calculation is split into two parts. Splitting up the calculation in two parts as shown in Equation 3.5 increases the amount of shares from $(d+1)^2$ to $2(d+1)^6$ (the curly braces indicate a concatenation of shares).

$$q_{(i,j,k)} = \{a_i b_j c_k \oplus a_i b_j d_k, a_i c_j d_k\} \tag{3.5}$$

By applying this solution to the equation of the inversion (Equation 3.4), we can denote the sharing of the inverter as in Equation 3.6. The curly braces under the equations ensure correctness of the sharing and denote that certain terms are only present in certain output shares (when the stated constraint is fulfilled).

$$\begin{aligned}
a'_{(i,j,k)} =& \underbrace{a_{(i)}}_{j=k=0} \oplus a_{(i)}b_{(j)}c_{(k)} \oplus \underbrace{a_{(i)}d_{(k)}}_{j=0} \oplus \underbrace{b_{(j)}}_{i=k=0} \\
b'_{(i,j,k)} =& a_{(i)}b_{(j)}c_{(k)} \oplus a_{(i)}b_{(j)}d_{(k)} \oplus \underbrace{a_{(i)}d_{(k)}}_{j=0} \oplus \underbrace{b_{(j)}}_{i=k=0} \oplus \underbrace{b_{(j)}c_{(k)}}_{i=0} \\
c'_{(i,j,k)} =& \{ \underbrace{a_{(i)}}_{j=k=0} \oplus a_{(i)}b_{(j)}c_{(k)} \oplus \underbrace{b_{(j)}}_{i=k=0} \oplus \underbrace{c_{(k)}}_{i=j=0}, a_{(i)}c_{(j)}d_{(k)} \oplus \underbrace{b_{(i)}d_{(k)}}_{j=0} \} \\
d'_{(i,j,k)} =& \{ a_{(i)}b_{(j)}c_{(k)} \oplus a_{(i)}b_{(j)}d_{(k)} \oplus \underbrace{a_{(i)}c_{(k)}}_{j=0} \oplus \underbrace{a_{(i)}d_{(k)}}_{j=0} \oplus \underbrace{b_{(j)}}_{i=k=0} \oplus \underbrace{b_{(j)}c_{(k)}}_{i=0} \oplus \underbrace{c_{(k)}}_{i=j=0} \\
& \oplus \underbrace{d_{(k)}}_{i=j=0}, a_{(i)}c_{(j)}d_{(k)} \oplus b_{(i)}c_{(j)}d_{(k)} \}
\end{aligned} \tag{3.6}$$

Final multiplier stage to output transformation. For the final multiplier stage we avoid collisions by using an additional set of freshly masked copies of the S-box inputs (x'', with $d+1$ shares). These copies are then combined with outputs of the $GF(16)$ inverter in the multipliers. As these multiplications occur in parallel and no nonlinear transformation follows in the S-box, only one additional copy of the inputs x'' suffices for both multiplications. The adjacent linear transformations are applied share-wise and with respect to the share domains to avoid collisions at this stage. Up to this point, no additional online randomness or registers are required and further linear transformations are still possible within the same clock cycle. As a drawback, the number of shares is increased to $2(d+1)^7$ at the output. For practical implementation of a full AES, a resharing and compression stage is thus required at some point as we will discuss in Section 5.2. However, our goal in this chapter was to demonstrate that DOM masked circuits can be implemented without the necessity for register stages in nonlinear operations, and thus giving designers the possibility to trade a higher number of shares against less latency.

"The art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable."

— Sun Tzu

4

Conclusions

In this part of the thesis, we introduced the DOM scheme as countermeasure against SCA and showed possible trade-offs in the form of the derived masking schemes UMA and LOLA. The essence of the domain-oriented masking perspective is to shift the design perspective of masked hardware implementations from a functional level as in TI to the circuit level. This offers the advantage that hardware designers can stay in their preferred design paradigm because the relation between the unmasked and the masked circuits remains natural. From an abstract point of view, we achieve share independence as required for secure masking just by copying the original circuit into a number of independent subcircuits (domains) the same way as we split up sensitive information into the according number of shares in masking. By associating each share per variable with one of these domains, and by keeping the shares in their respective domains the share independence as required for secure masked circuits, even in higher-order cases, can be achieved quite easily. The domain-oriented design approach allows us to create generic hardware designs that can be synthesized for any desired protection order without changing the design itself but only by adjusting the security parameter. We will demonstrate this in the next part of the thesis.

Another benefit of DOM is that the randomness costs are significantly reduced compared to other schemes, especially in the higher-order case. In practice, the generation of online randomness is one of the biggest obstacles that hinder the efficient implementation of higher-order masking. The generation of fresh randomness not only increases the chip area required for Pseudo-Random Number Generators (PRNGs) or True-Random Number Generators (TRNGs), but also requires more energy and power. Saving randomness is thus essential to make higher protection orders suitable for practical implementations.

With UMA we investigated how the randomness costs for generic masking in hardware can be reduced even further. The cost for reducing the amount of required online randomness is an increase in the number of delay cycles from one (for DOM) to up to five cycles, and an increase in the chip area which, however, in practice is at least compensated by less circuitry required for producing the randomness.

Nevertheless, there exist practical applications where the latency of a masked implementation is more important than reducing the overall implementation costs. With LOLA we thus explored how the latency of a generically masked hardware design can be traded against an increased number of shares.

Part II

Masked Implementations

In this part of the thesis, we investigate the hardware overhead costs for the DOM and the derived schemes UMA and LOLA that were introduced in Part I. We show implementations of the most relevant symmetric-key primitives like the AES, as well as for the Secure Hash Algorithm 3 (SHA3) KECCAK, but also for the next generation of symmetric-key primitives in the form of the so-called Authenticated Encryption (AE) scheme ASCON. Not only cryptographic primitives require protection against unwanted data leakage. Also the protection of other parts of a system that manipulate and process sensitive data requires appropriate measures. We show the versatility of DOM on the example of a 32-bit RISC-V processor.

Area and throughput comparisons as well as randomness costs are considered for the comparison of the schemes. Furthermore, we compare our implementations to related masked implementations to provide a comprehensive view on the benefits and drawbacks of our schemes.

The following papers provide the content for the respective chapters.

- **Chapter 5: Advanced Encryption Standard (AES)**

  This chapter uses the paper:

  ▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order." In:* CT-RSA*. vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112*

  and its extended version:

  ▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order." In:* IACR Cryptology ePrint Archive *(2016)*

  for the original DOM implementations of the AES in Section 5.1. The LOLA implementation of the AES S-box in Section 5.2 is based on the work:

  ▶ *Hannes Groß, Rinat Iusupov, and Roderick Bloem.* Generic Low-Latency Masking in Hardware*. CHES 2018 (in press)*

  Contribution. The author of this thesis is the main author of the papers used in this chapter.

- **Chapter 6: Ascon—Authenticated Encryption**

  The ASCON AE scheme is introduced with some unprotected implementation variants based on:

  ▶ *Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. "Suit up! — Made-to-Measure Hardware Implementations of ASCON." in:* DSD*. IEEE Computer Society, 2015, pp. 645–652*

  Before that, DOM, UMA, and LOLA variants are presented based on:

▶ *Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In:* CHES*. vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136*

and

▶ *Hannes Groß, Rinat Iusupov, and Roderick Bloem.* Generic Low-Latency Masking in Hardware*. CHES 2018 (in press)*

Contribution. The author of this thesis is the main author of the parts of the papers used in this chapter. Two of the hardware implementations and their description in the first paper were provided by Erich Wenger and some content with regard to the description of Ascon was contributed by Christoph Dobraunig.

- **Chapter 7: Keccak Secure Hash Algorithm (SHA3)**

  ▶ *Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK." in:* DSD*. IEEE Computer Society, 2017, pp. 205–212*

  Contribution. The author of this thesis is the main author of the paper. The implementation-related parts and the DOM-protected hardware implementations were provided by David Schaffenrath in the course of his master's project.

- **Chapter 8: RISC-V Processor**

  ▶ *Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs." In:* CARDIS*. vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104*

  Contribution. The author of this thesis is the main author of the paper. The description of the implementation and the RISC-V DOM implementation were provided by Manuel Jelinek in the course of his master's thesis.

"Der Worte sind genug gewechselt,
Laßt mich auch endlich Taten sehn!"
("Enough words have been exchanged,
now at last let me see some deeds!")
— J. W. von Goethe

5

Advanced Encryption Standard (AES)

The AES is the most frequently used symmetric-key primitive and was designed by Joan Daemen and Vincent Rijmen originally under the name Rijndael before being standardized as the AES in 2000. The widespread distribution makes the AES an interesting target for benchmarking new masking schemes. We first introduce a DOM implementation of the full AES and compare it to existing AES implementations before we show the suitability of our LOLA approach on the S-box of the AES.

5.1 DOM-Protected AES

To compare the efficiency of the DOM scheme to other schemes, we implemented a variant of the AES encryption-only design suggested by Moradi and Poschmann [Mor+11]. Moradi's design was also used and modified by Bilgin et al. [Bil+14a; Bil+15b] resulting in a more efficient first-order TI, and by De Cnudde et al. [Cnu+15] for a second-order TI of the AES S-box following the CMS scheme [Rep15].

The control path of our modified AES design consists of a linear-feedback shift register (LFSR), the round constant generation module (RCON), and some additional logic gates to generate the control signals (see [Mor+11] for more details). Our LFSR module has a cycle length of 23. In each round, the first 16 cycles are spent on *AddRoundKey* and *SubBytes*. Then there are four cycles used for *MixColumns* and to calculate the first four bytes of the next round key. Then there are two dummy rounds inserted to bring the state register in correct position for further processing before in the final cycle the ShiftRows transformation is performed. The datapath (Figure 5.1) mainly consists of the

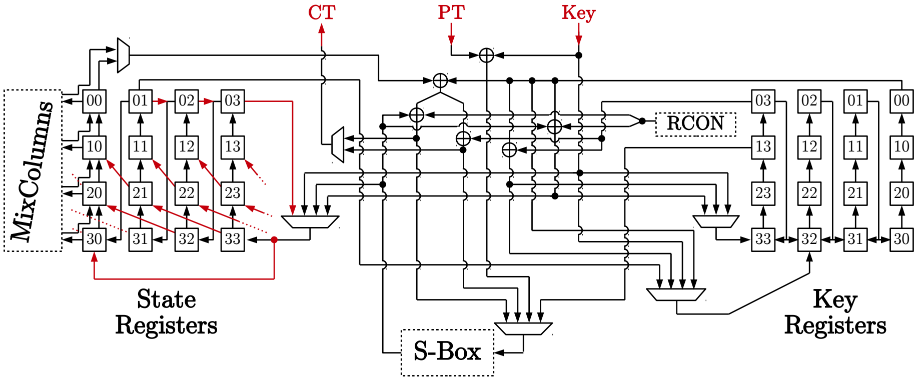

Figure 5.1: Datapath of the DOM AES implementation (all data signals are 8 bits wide)

S-box, the key and state registers which are implemented as shift registers, the *MixColumns* module, and some multiplexers.

5.1.1 DOM Design of the AES S-box

The by far most complex and most security critical part of the AES implementation is the S-box. Figure 5.2 shows our design of a 1^{st}-order protected variant of Canright's [Can05] AES S-box design. The S-box consists of many linear operations like the linear mappings at the input and the output, the square scalers, the sub-field inverters, and the adders. These are the parts that can be implemented share-wise for both domains in a straightforward way. The Galois field multipliers with different field order form the non-linear parts of the S-box. Canright's S-box makes repeated use of a finite field isomorphism to express $GF(2^8)$ elements as multiple elements in lower subfields—down to eight elements in $GF(2)$. These $GF(2^n)$ multipliers are replaced by the masked DOM GF multipliers. Therefore, the standard-cell library AND cells used for the calculation step in the masked AND gate are simply replace by the according GF multipliers.

To maximize the efficiency of the implementation, seven pipelining stages are added to the S-box. The pipelining registers are marked with circles and appear along the red and green dotted lines in Figure 5.2. Red dotted lines indicate multiplier related stages which are also labeled Stage 1-5 in order to refer to them more easily. The green marked registers are required to ensure independence in the presence of glitches for the inputs of the adjacent GF gates. To make the S-box secure and efficient at the same time, it is necessary to pinpoint all GF gates that have related input sharings. These gates need to be treated more carefully than the one with independent inputs. We now discuss the security of each multiplication stage individually which reveals that the additional pipeline

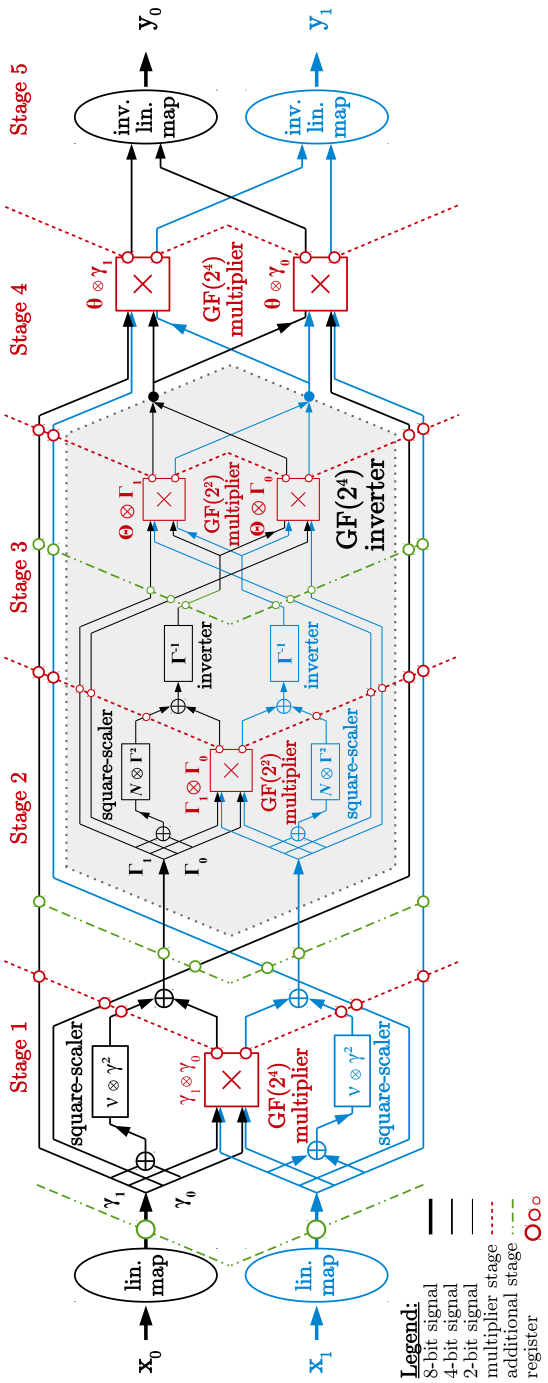

Figure 5.2: First-order DOM design of the AES S-box

stages (plotted in green) are required at multiplication stages 1, 2, and 3, but not at 4 and 5.

Stage 1. The $GF(2^4)$ gate in Stage 1 receives its inputs from the linear mapping at the S-box input. The linear mapping takes the 8-bit input shares x_0 and x_1 and linearly combines these eight bits inside their respective domain (see [Can05] for more details). Because of the different signal transition times and gate delays, it is therefore possible that the output of the linear mapping temporarily consists of bits with related sharing. Applying these bits directly to the GF gate from Figure 1.1—while the linear mapping has not yet settled—would thus violate the independence in the cross-domain terms associated GF multipliers. To avoid these glitches, registers are inserted after the linear maps to ensure the signals are settled before the bits are applied to the GF gate.

Stage 2 and 3. The situation is similar at Stage 2 and Stage 3. At these stages, glitches can occur from the combination of the square scaler outputs with the outputs of the GF gate. Again these glitches can be avoided by inserting pipelining stages at the marked positions in Figure 1.2.

Stage 4. For the GF gates in Stage 4, the inputs are the pipelined S-box inputs and the output of the GF gates of the previous stage. The output of the GF gate of Stage 3 originate from the inputs of the $GF(2^4)$ inverter which is remasked in Stage 1 (the masking is effective at latest at Stage 2). Therefore, the inputs of the Stage 4 GF gates are clearly independent and so no registers are required here.

Stage 5. The output mapping in this stage is again a linear transformation and uncritical as long as it is not followed by a nonlinear transformation that is unprepared for related sharing of its inputs. However, in our design of the AES core the output of the S-box is either stored in the key or state registers before it is used again, or fed into the S-box which is also uncritical because the input multiplier of either S-box variant is already prepared to process related input sharings.

The rest of the S-box is implemented according to the original Canright design but without some of its optimizations that would not be beneficial for our implementation. Canright's design, for example, reuses some temporary results in other parts of the S-box. Storing temporary results would lead to many additional pipelining registers for our design of the S-box and is therefore not suitable. For the generalization of the S-box to higher protection orders, the black (or blue) parts in Figure 5.2 are basically duplicated and the secure GF gates are generated as described in Chapter 1.

Table 5.1: First-order secure AES-128 implementation results

Design/Module	Chip Area *[%]*	Chip Area *[kGE]*	Randomness *[Bits/S-box]*	Cycles
Our Implementation (90 nm)				
Overall	100.0	**6.0**	**18**	**246**
S-box	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 TI (180 nm)				
[Mor+11]		11.0 / 10.8[a]	48	266
[Bil+14a]		9.1 / 8.2[a]	44	246
[Bil+15b]		8.1 / 7.3[a]	32	246
d+1 TI (45 nm)				
[Cnu+16]		6.7 / 6.3[a]	54	276

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

5.1.2 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 result 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 [Gro16].

Anyway, for a masked hardware design the number required fresh random bits is even more crucial for the efficiency of an implementation than the stated chip area of the designs. The generation of fresh random bits with high entropy requires additional hardware and involves, e.g., complex analog circuitry or pseudo random number generators based on symmetric primitives. Both options have a critical influence on the chip area requirements, the energy budget, and on the delay or throughput.

First-order secure AES. Table 5.1 compares our first-order secure AES hardware implementation with existing related work. The $d+1$ share designs of [Cnu+16] 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

Table 5.2: Second-order secure AES-128 implementation results

Design/Module	Chip Area		Randomness	Cycles
	[%]	*[kGE]*	*[Bits/S-box]*	
Our Implementation (90 nm)				
Overall	100.0	**10.0**	**54**	**246**
S-box	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 TI(estimated [Cnu+16], 45 nm)				
[Cnu+15]		*18.6 / 14.9*[a]	126	276
d+1 TI (45 nm)				
[Cnu+16]		10.5 / 10.3[a]	162	276

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

requires at least three shares for securely calculating non-linear functions while the first-order $d+1$ share designs require only two shares.

In comparison with $d+1$ TI design [Cnu+16] which requires 54 random bits per S-box 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 5.2, 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. [Cnu+16]. 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 Implementations 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 5.3 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 S-box

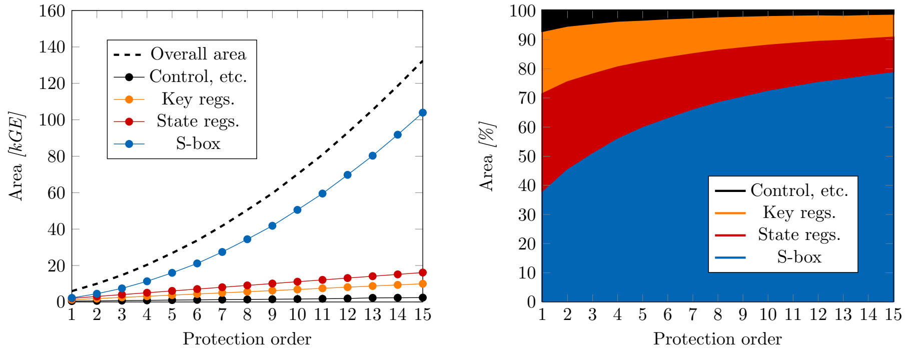

Figure 5.3: Area requirements absolute (left) and in percent (right) per protection order

and the contained *GF* gates grow quadratically. For the S-box, 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.

5.2 LOLA-Protected AES S-box

In this section, we discuss the costs for reducing the latency of the DOM AES S-box using LOLA, and present a comparison with related work and our DOM implementation. We present three variants of Mui's AES S-box depicted in Figure 3.5. The first variant does not consider share compression at the output of the S-box. We call this variant the zero-latency variant which denotes that further linear operations on the output shares (like *ShiftRows*, *MixColumns*, or *AddRoundkey*) are still possible within the same cycle. However, this variant hides the costs for share compression and we thus consider also two additional variants. The first-order zero latency S-box requires 17.83 kGE of chip area for a 90 nm UMC process with a maximum clock frequency of 228 MHz.

For the one-cycle variant of the S-box we compress the shares at the output of the zero latency S-box by using a CMS compression function after the output transformation. The number of shares is thus reduced from $2(d+1)^7$ to $d+1$ again which requires $16(d+1)^7$ random bits and registers in total. This variant would help in a full implementation of the AES to reduce the number of subsequent linear transformations and registers, for the cost of the CMS resharing stage. The chip area requirements for the single-cycle masked S-box variant with first-order protection are 61 kGE at a maximum clock frequency of 356 MHz, and it requires 2 kbits of fresh randomness.

The chip area costs for the single-cycle S-box variant are admittedly very high given the fact that one round unrolled variant of AES-128 requires 16 of these S-boxes. The costs can be reduced by performing an intermediate resharing and compression step after the inverter in Figure 3.5. The number of shares is thus reduced from $2(d+1)^6$ to $d+1$ before the last two multiplications are performed which saves many of the area-consuming $GF(16)$ multipliers and linear transformations at the output. The final compression requires $8(d+1)^2$ fresh random bits. In total, this variant requires $6(d+1)^6 + 8(d+1)^2$ random bits (416 bits for first-order protection) and the chip area is reduced to 6.7 kGE. For second order, the amount of required randomness is 4,446 bits and the chip area is 57 kGE.

5.2.1 Comparison with DOM and Related Work

A summary of the results for our low-latency AES S-box variants and related work is given in Table 5.3. All of our stated results are post-synthesis results for a 90 nm Low-K UMC process with 1 V supply and a 20 MHz clock, synthesized with the Cadence Encounter RTL compiler v14.20. The used cell library and tool chain vary among the stated related work and the numbers should not be compared directly.

As the comparison shows, our low-latency AES S-box variants are the first published constructions that reduce the latency below three cycles per S-box calculation. The price is a significant increase of both chip area and randomness requirements, especially for the single-cycle S-box variant with 60.73 kGE and 2 kbit of randomness. The zero latency variant requires with 17.8 kGE almost nine times more area than the smallest design. The chip area overhead for the first-order AES S-box with two cycles is relatively moderate with about a factor of three times the area of the smallest known S-box construction. Furthermore, our designs are generic.

Comparing the randomness requirements is difficult since most of the stated work uses a different amount of input shares which is usually not considered to be part of the required (online) randomness. In this context, our zero latency variant requires no additional online randomness but it requires of course additional randomness for the sharing and the duplication of the input variables. In case of our two-cycle variant, the online randomness costs for calculating one S-box are with 416 (and 4,446 bits, respectively) significantly increased over the state of the art.

However, we note that our primary goal was to demonstrate that for generic higher-order protection a reduction of the latency is indeed possible even in complex designs like the AES S-box. The most efficient design choices and the best point at which the shares can be again compressed remains to be an open problem.

Furthermore, we denote that we used the CMS scheme by Reparaz et al. [Rep+15] for the estimation of the randomness requirements of the generic protection case. However, as it was demonstrated by Moos et al. [Moo+18], the CMS scheme is only secure up to the second protection order. An efficient and generic compression algorithm thus also remains to be an open problem.

Table 5.3: Results and comparison of masked AES S-box implementations

Design	Order *[d]*	Size *[kGE]*	Cycles */S-box*	Max. Clock *[MHz]*	Randomness *[bits] (online)*
Zero Latency	first	17.83	0	228	0
Zero Latency	d		0		0
Single Cycle	first	60.73	1	356	2,048
Single Cycle	d		1		$16(d+1)^7$
Two Cycle	first	6.74	2	584	416
Two Cycle	second	57.11	2	517	4,446
Two Cycle	d		2		$6(d+1)^6 + 8(d+1)^2$
			Related work		
[Bil+14a]	first	3.71	3		44
[Bil+15b]	first	2.84	3		32
[Cnu+15]	second	7.9 - 11.2	6		126
[Cnu+16]	first	1.98	6		54
[GC17]	first	4.61	4		0
[GC17]	first	3.63 - 3.80	4		34 - 68
[GC17]	first	2.91 - 3.34	3		20-24
DOM S-box	first	2.2	8		18
DOM S-box	second	4.5	8		54
DOM S-box	d		8		9d(d+1)
[Mor+11]	first	4.24	4		48

Discussion on the impact on full AES implementations. In the following, we want to briefly discuss the expected impact on the latency and throughput of a full AES-128 implementation on the basis of our S-box implementation results. We denote that we are fully aware of the fact that the provisioning of such high amounts of randomness, as required for multiple instances of our S-box implementations in parallel, as well as the required chip area and power consumption would exceed the capabilities of most practical applications. This comparison should merely serve as a basis for future comparisons and to demonstrate that a cycle count reduction not automatically leads to a reduction of the overall latency.

The impact on the throughput as well as on the latency highly depends on the concrete AES implementation whose assumptions can highly vary. For this reason, we make runtime estimations for two different corner cases, namely best-case and worst-case cycle count estimations. An AES-128 encryption consists of a pre-round (which only performs the AddRoundKey transformation) followed by nine full rounds, and the final round (which omits MixColumns). A full round consists of SubBytes (the S-box layer) followed by ShiftRows, MixColumns and AddRoundKey.

An overview on our cycle count estimation is given in Table 5.4. For the best-case runtime estimation, we assume that only the S-box layer introduces delay cycles due to non-linear calculations and that a full SubBytes transformation with 16 S-boxes in parallel is implemented. All other transformations are assumed to be performed implicitly and the round keys are already precomputed. For the best case, we thus estimate a runtime that is ten times (number of full rounds plus the final round) the delay of the used S-box transformation ($10 l_{sbox}$). This cycle count corresponds, for example, to Intel's AES instructions [Gue09].

Table 5.4: Cycle count estimation for full AES-128 hardware implementations with a variable numbers of cycles for the S-box (l_{sbox})

Round	SubBytes	ShiftRows	MixColumns	AddRoundKey	Key Schedule
0	–	–	–	$0 \ldots 16$	$0 \ldots 2 + 16 l_{sbox}$
$1 \ldots 9$	$(1 \ldots 16) l_{sbox}$	$0 \ldots 1$	$0 \ldots 4$	$0 \ldots 16$	$0 \ldots 2 + 16 l_{sbox}$
10	$(1 \ldots 16) l_{sbox}$	$0 \ldots 1$	–	$0 \ldots 16$	–
Overall	$10 l_{sbox} \ldots 224 + 320 l_{sbox}$				

For the worst-case cycle count estimation, we assume that only one S-box is implemented in hardware, that the key schedule is performed on the fly, and that all other transformations require one clock cycle. The key schedule is thus assumed to require at most two cycles (for RotWord and Rcon) plus 16 times the number of the assumed S-box delay for SubWord. Our worst-case estimation of $224 + 320 l_{sbox}$ approximately corresponds to Feldhofer et al.'s [Fel+05] low-power AES implementation which requires 1,032 cycles for a two-cycle S-box.

For our single-cycle S-box variant, we thus estimate the cycle count for a full AES-128 to be between 10 cycles and 544 cycles, and for the two-cycle S-box the estimations result in 20 cycles and 864 cycles, respectively. Given the maximum clock frequencies in Table 5.3 (and neglecting additional combinatorial delay introduced by other components of the AES), the estimated latency for the single-cycle S-box AES is between 1.53 µs and 28.09 ns, and for the two-cycle S-box AES variant between 1.48 µs and 34.25 ns. The reduction to a single-cycle S-box could thus, in the best of cases save about 18% of latency while for the worst case estimation the single-cycle S-box introduces an even 3.3% higher latency in the full AES than the two cycle S-box. Similarly, the estimated throughput is between 83.76 Mbps and 4.56 Gbps for the single-cycle variant, and between 86.52 Mbps and 3.74 Gbps for the two-cycle S-box variant.

In summary, even when keeping practical limitations regarding randomness provisioning, chip area, and power or energy consumption aside, it is not entirely clear whether or not a reduction of the latency for the nonlinear parts of a cryptographic implementation automatically leads to a reduction of the overall latency of the system. In practice, the most valuable design choices therefore need careful evaluation of the overall constraints of a system.

6

Ascon—Authenticated Encryption

Symmetric cryptography has a rich history of competitions to find good and secure cryptographic primitives. Winners of such competitions like the AES serve as a basis of our modern information and communication systems. In the so-called CAESAR competition over 45 different AE schemes competed in becoming the standard primitives for authenticated encryption. One of the finalists in this competition is the AE scheme Ascon.

6.1 Overview on Ascon

Ascon [Dob+16] has a sponge-like mode of operation as depicted in Figure 6.1. Its state size, the permutation p and mode of operation are chosen in a way that allows compact hardware implementations, while still providing high throughput. Ascon comes in two different versions, namely Ascon-128 and Ascon-128a

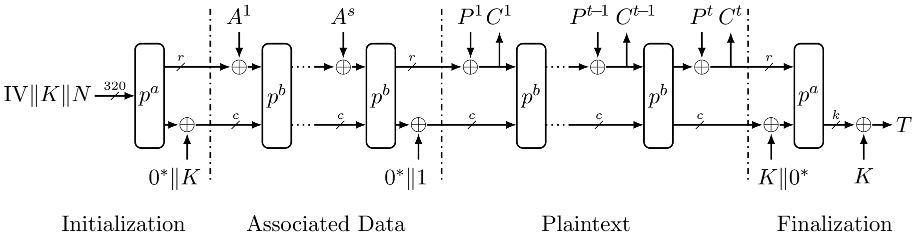

Figure 6.1: The encryption of Ascon-128

with both 128 bit security level but different rates. We focus on Ascon-128 in the remainder of this chapter.

6.1.1 Mode of Operation

Ascon has a state size of 320 bits (consisting of five 64-bit words $x^0, \dots, x^4$) that are updated in four phases: *Initialization*, *Processing of Associated Data*, *Processing of Plaintext/Ciphertext*, and *Finalization*. All phases use the same permutation function p that is applied twelve times during the *Initialization* and *Finalization* phase. The lighter variant of p with six rounds is used for processing the data and ensures high performance. The data is handled in 64-bit blocks.

The *Initialization* phase, takes the secret key K (128 bits) and the public nonce N (128 bits). This nonce has to be fresh for every encryption and must not be used twice. If the nonce is used twice or multiple times, then the confidentiality is jeopardized.

After the *Initialization* phase the optional associated data A^i is processed. Associated data is information, which does not need to be confidential, but must not be altered by an attacker. Each block A^i is added to the secret state. If there is no associated data to process, the whole step can be omitted.

In the *Encryption* phase, each plaintext block P^i is xored with the secret state to produce one ciphertext block C^i. Six consecutive round transformations p are executed for each of the 64-bit data blocks.

After the generation of the ciphertext, the *Finalization* starts. The output of the *Finalization* is the 128-bit tag T. With the help of this tag, modifications of the ciphertext and the associated data can be detected during decryption (validation).

Decryption is very similar to encryption. Just the part, where the ciphertext is processed instead of the plaintext differs slightly. Thus, no inverse of the permutation is needed for decryption. So, both encryption and decryption can be implemented with just a slight overhead compared to encryption only.

6.1.2 Permutation

Ascon-128 uses two permutations, p^6 and p^{12}. The two permutations are the 6 and 12 iterative executions of the round transformation p. The round transformation p consists of a constant addition to x^2, followed by an application of a substitution layer, and a linear layer.

The substitution layer is the parallel application of 64 5-bit S-boxes. The S-boxes used for Ascon are an affine transformation of the χ mapping of Keccak [Ber+11]. This affine mapping improves some cryptographic properties of the Keccak's χ mapping, while still leaving the core of the S-box and therefore the algebraic degree of 2 intact. Moreover, the Ascon S-box can be implemented using only a few logical operations, which are highly parallelizeable (Figure 6.2).

The linear layer consists of five applications of the function $\Sigma_{l^i,r^i}(x^i) = x^i \oplus (x^i \ggg l^i) \oplus (x^i \ggg r^i)$ to each 64-bit word of the state $(x^0, \dots, x^4)$. The Σ function is similar to the one used in SHA-2 [NIS95], except that other rotation

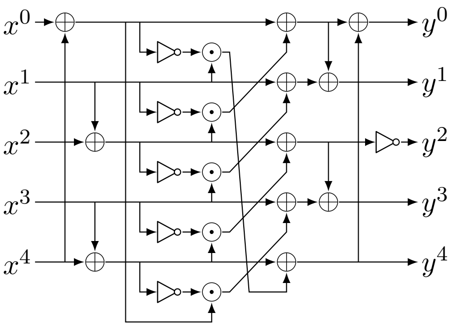

Figure 6.2: Substitution layer with 5-bit S-box Ascon

values (l^i, r^i) are used. The rotation values (l^i, r^i) are different for every 64-bit word in one round.

6.1.3 Hardware Security Properties of Ascon

In order to protect Ascon against SCA it is important to reflect on the properties of Ascon that make the life of an attacker hard. Ascon uses a mode of operation which is based on MonkeyDuplex [Dae12]. In contrast to MonkeyDuplex, Ascon uses a keyed *Initialization* and *Finalization.* This has the effect that a state recovery during the processing of data neither leads to the recovery of the secret key, nor allows universal forgeries.

Therefore, SCA attacks on the data processing phase may be applied in order to recover the internal states, but do not allow the attacker to recover the key. In addition, SCA on the *Finalization* are hard, since the attacker would have to attack both the key and many unknown state bits that have no influence on the emitted tag. In fact, the most vulnerable phase is the *Initialization.* For Ascon-128, three out of five input bits of the first S-boxes are publicly known and the other two bits belong to the secret key.

6.2 Unprotected Hardware Designs

Ascon allows to be optimized for many practical applications, where both confidentiality and authenticity are required. In the following, three implementation variants with different design goals are introduced.

6.2.1 High Throughput Design (Ascon-fast)

Due to the low complexity of Ascon's round transformation, it is possible to fully unroll a complete round transformation and still achieve high frequencies. As it turns out, the round transformations are so hardware-friendly that even multiple rounds can be computed in a single clock cycle. The Ascon-fast variants aim at a maximal data throughput with a minimum of latency. Therefore, at least one round transformation is performed in every clock cycle and no pipelining stages are used. Each Ascon-fast variant uses a different number of unrolled round transformations. The datapath of Ascon-fast as shown in Figure 6.3 mainly consists of the unrolled round transformations (five 64-bit state registers, 64 parallel S-boxes, and the linear diffusion layer). Only a few additional multiplexers and XOR-gates are needed to connect the unrolled round transformation with the data bus and the key register.

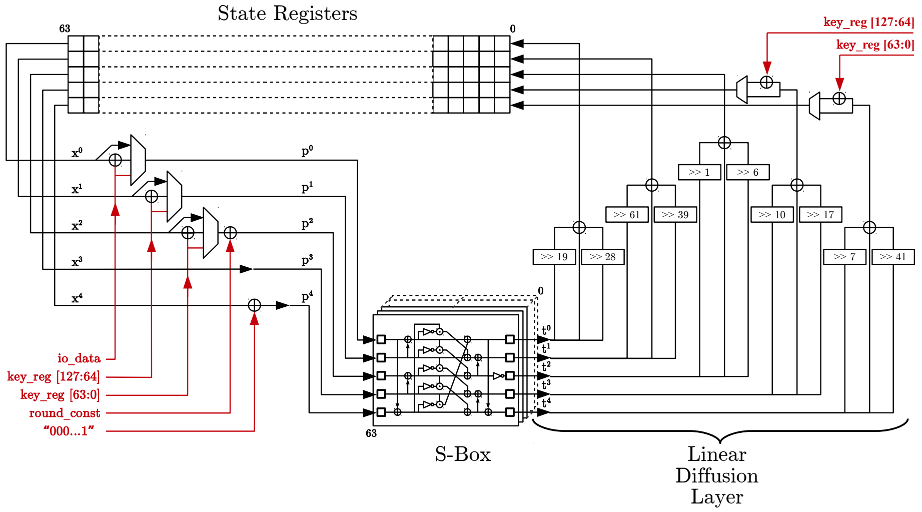

Figure 6.3: Datapath of the *fast* variant of Ascon with one round transformation per cycle

6.2.2 64-bit Datapath Design (Ascon-64-bit)

The design idea behind the Ascon-64-bit implementation is based on the inherent 64-bit structure of Ascon. Instead of a concrete implementation of the S-box and the linear diffusion layer, this design uses an arithmetic logic unit (ALU) comparable to a microcontroller design. Consequently, the controlpath works similarly to a sequential program code that is executed by the datapath in Figure 6.4.

Besides the five state registers, there exist also two temporary registers, which—together with the inputs from the controlpath—form the input operands of the ALU. The ALU itself consists of an iterative barrel-shifter unit, three logic operations, and a data-storage unit that takes the 64-bit bus data input and stores it either in the high or the low part of the selected operand. On the output of the ALU, the result of the operation is selected that is then applied to the destination register. During the execution, the S-box and linear layer are iteratively calculated using the operations of the ALU. Thus, one round operation takes 59 clock cycles.

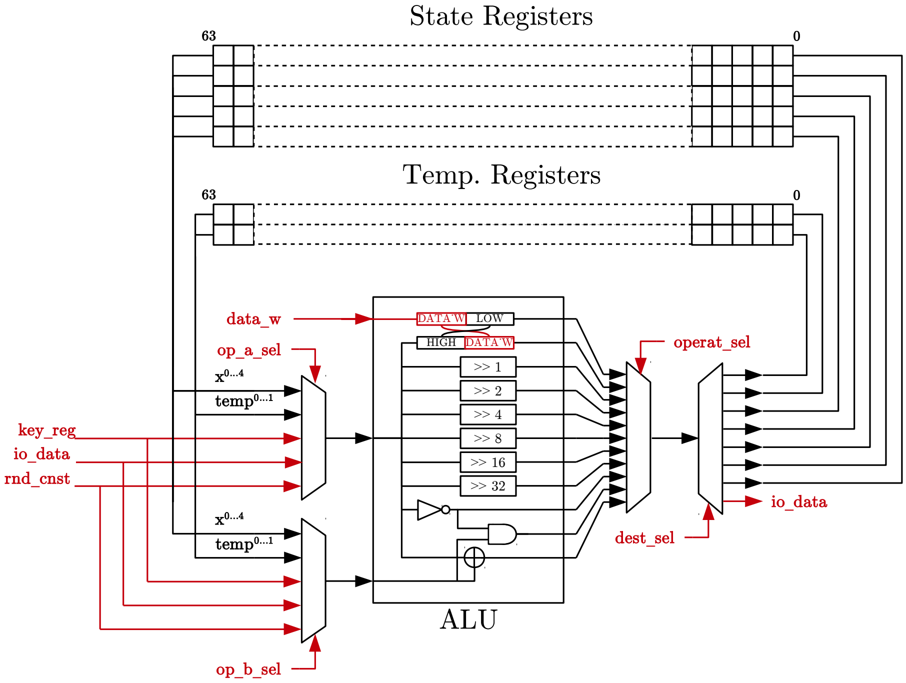

Figure 6.4: Datapath of the *64-bit* variant of Ascon

6.2.3 Low Area Design (Ascon-x-low-area)

The datapath of the so-called Ascon-x-low-area variant (see Figure 6.5) uses a radical low-area approach, which can be summarized as "one bit operation per cycle". The state consists of five clock-gated shift registers with independent shift-enable inputs. For the S-box calculation, all state registers are activated and shifted bit-slice-wise through the single S-box instance. The result is stored in the least-significant bits of the state. Accordingly, the whole S-box layer operation consumes 64 cycles. The subsequent linear diffusion layer is split up into five interleaved subiterations in which each state register is updated individually. As a single state bit depends on two other bits of the same state row, the linear layer cannot be calculated without temporarily storing either the results or the state row itself, respectively. Thus, another (temporary) shift register is needed that in one iteration holds the result of the current linear layer operation and in the next iteration is used to write the result back. Once the first subiteration of the linear layer is finished, the calculation of the next state row and the write-back operation can be done in parallel. This uncompromising low-area approach results in 512 clock cycles per round transformation.

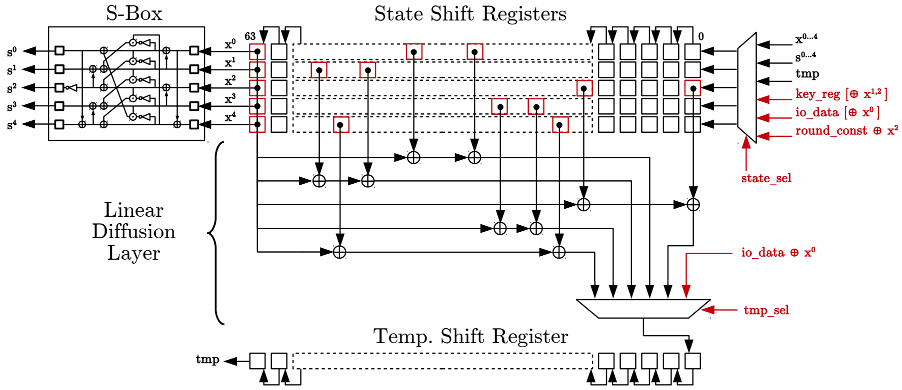

Figure 6.5: Datapath of the *x-low-area* variant of Ascon

6.2.4 Results

All Ascon designs are implemented in VHDL and evaluated using a Cadence-based ASIC design-flow. For the following results, a 90 nm UMC standard performance low-K library from Faraday is used with a global clock of 1 MHz and a 1 V power supply. The designs are compiled with the Cadence Encounter RTL compiler version v08.10-s28_1 and routed with Cadence NanoRoute v08.10-s155. The results of all practical evaluations are collected in Table 6.1.

Table 6.1: Characteristics of the Ascon-128 hardware implementations

Design	Chip Area w/o interface	Chip Area w/ interface	Throughput	Throughput	Power at 1 MHz	Energy
	[kGE]	*[kGE]*	*[cycles/byte]*	*[Mbps]*	*[µW]*	*[µJ/byte]*
Ascon-fast						
1 round	7.08	7.95	0.75	5,524	43	33
2 rounds	10.61	11.48	0.38	8,425	72	27
3 rounds	14.26	15.13	0.25	10,407	102	25
6 rounds	24.93	25.80	**0.13**	**13,218**	184	**23**
Ascon-64-bit	4.99	5.86	44.25	72	32	1,397
Ascon-x-low-area	**2.57**	**3.75**	384.00	14	**15**	5,706

Some implementations can process up to 8 bytes of data in a single clock cycle (0.125 cycles per byte) and other implementations are as small as 2.57 kGE. Especially the characteristics of Ascon-fast with one unrolled round are impressive. This implementation needs 7.08 kGE (7.95 kGE with key register and 64-bit bus interface), reaches a maximum clock frequency of 517 MHz, and can therefore process up to 5.5 Gbit per second. This means that the most straightforward design is easily sufficient to encrypt a gigabit Ethernet connection on the fly. At 100 MHz, the design only needs 529 µW (38 µW static leakage and 4.9µW/MHz dynamic power) and is therefore also suitable for mobile applications. As it

Table 6.2: Characteristics of related implementations

Design	Chip Area	Throughput	Power	Technology
	[kGE]	*[Mbps]*	*[μW/MHz]*	
AES-CCM [Bog+13]	3.77	57	5.12	STM 65 nm
AES-OCB2 [Bog+13]	5.92	113	8.11	STM 65 nm
AES-ALE [Bog+13]	2.70	244	10.55	STM 65 nm
Minalpher [Sas+14] low-area	2.81	369	—	
SILC [Iwa+14] V1	15.70	764	—	
AES-OCB [Par05]	22.55	854	—	TSMC 90 nm
SILC [Iwa+14] V2	23.10	2,635	—	
Scream ED [Gro+14b] 1 Round	6.23	4,577	—	STM 65 nm
Keccak MonkeyDupl. [YK13]	5.90	4,900	42	
Scream ED [Gro+14b] 2 Round	8.31	5,190	—	STM 65 nm
Minalpher [Sas+14] high-speed	14.32	6,104	—	
Norx [Aum+14]	59.00	10,000	—	UMC 180 nm
ICEPOLE [Mor+14]	—	41,364	—	FPGA (Xilinx Virtex 6)

also provides the best performance per throughput, it is perfectly suitable for *embedded systems*.

If higher throughput is required, ASCON-fast with six unrolled rounds can process more than 13 Gbit/sec at 206 MHz. This is more than sufficient even for 10 gigabit network connections. For *RFID* applications, where size as well as power matter, ASCON-x-low is only 2.57 kGE large and requires as little as 15 µW for a 1 MHz clock source. For a, for example for RFID tags more suitable, power saving 130 nm low-leakage UMC technology, the power consumption is reduced to 4.1 µW.

Energy, the most critical characteristic of *wireless sensor nodes*, is the product of power and runtime. The ASCON-fast implementations require the least amount of energy because of their low runtimes. Even though six unrolled rounds give the best energy results, it would probably be more reasonable to use ASCON-fast with one unrolled round for *wireless sensor nodes*.

Related Work A fair comparison of hardware designs is a difficult task. Different designers make diverging assumptions about, e.g., key registers or bus interfaces. Additionally, results are highly dependent on the used manufacturing technology, the used toolchain (e.g., Cadence, Synopsis, Menthor, Xilinx, or Altera), and the external operating conditions (e.g., power supply voltage or ambient temperature). Therefore, the following comparison with related work has to be interpreted with caution.

In Table 6.2, the hardware results of several AE designs are listed. There are standardized AES-based implementations [Bog+13; Par05] and CAESAR candidates based on sponge constructions [Aum+14; Mor+14; YK13] and block ciphers [Gro+14b; Iwa+14; Sas+14]. Figure 6.6 visually combines Tables 6.1 and 6.2. The horizontal axis shows the throughput and the vertical axis depicts the area footprint. The dashed equi-efficiency lines indicate a constant throughput per area ratio.

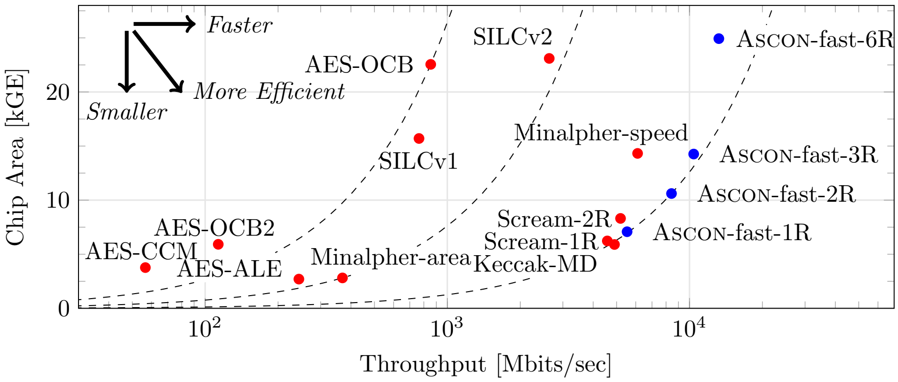

Figure 6.6: Throughput versus area comparison

It seems that ASCON provides, together with Keccak MonkeyDuplex [YK13], excellent performance per area. All AES implementations are a magnitude slower than even the slowest ASCON-fast implementation. Only Norx [Aum+14] and ICEPOLE [Mor+14] achieve similar or higher performance. However, Norx is more than twice as large (59 kGE) as the largest ASCON design (26 kGE). The table contains ASIC results available for ICEPOLE so far. In terms of size, ASCON needs six times fewer registers to store the state and the processed data (1280 + 1024 vs. 320 + 64) than ICEPOLE. ICEPOLE S-boxes also have an algebraic degree of four and are thus potentially harder to share than the ASCON S-boxes, which probably results in a higher overhead for protected implementations.

6.3 DOM- and UMA-Protected Implementations

An overview of the top module of our DOM and UMA hardware design is given in Figure 6.7 (left). It consists of a simple data interface to transfer associated data, plaintext or ciphertext data with ready and busy signaling which allows for simple connection with e.g. AXI4 streaming masters. Since the nonce input and the tag output have a width of 128 bit, they are transferred via a separate port. The assumptions taken on the key storage and the Random Number Generator (RNG) are also depicted. We assume a secure key storage that directly transfers the key to the cipher core in shared form, and an RNG that has the capability to deliver as many fresh random bits as required by the selected configuration of the core.

The core itself consists of the Finite-State Machine (FSM) that controls the general process (*control FSM*) and the *round counter* that form the control path, and the *state* module that forms the data path and is responsible for all state transformations. Figure 6.7 (right) shows a simplistic schematic of the state module. The state module has a separate FSM and performs the round

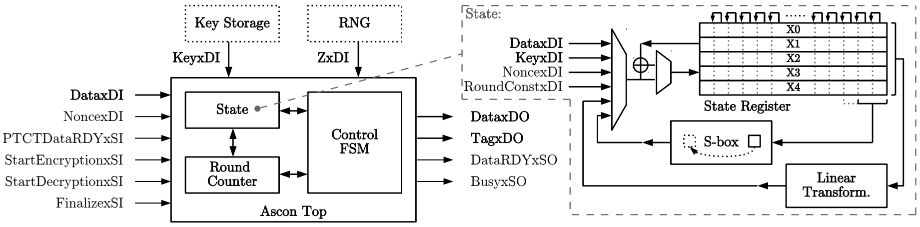

Figure 6.7: Overview of the Ascon core (left) and the state module (right)

transformation in four substeps: (1) during *IDLE*, the initialization of the state with the configuration constants, the key, and the nonce is ensured.

(2) in the *ADD_ROUND_CONST* state, the round constant is added, and optionally other required data is either written or added to the state registers like input data or the key. Furthermore, it is possible to perform the linear parts of the S-box transformation already in this state to save pipeline registers during the S-box transformation and to save one delay cycle. This option, however, is only used for the configuration of Ascon where all 64 possible S-box instances are instantiated.

(3) the *SBOX_LAYER* state provides flexible handling of the S-box calculation with a configurable number of parallel S-box instances. Since the S-box is the only non-linear part of the transformation, its size grows quadratically with the protection order and not linearly as the other data path parts of the design. The configurable number of S-boxes thus allows to choose a trade-off between throughput and chip area, power consumption, et cetera. During the S-box calculation, the state registers are shifted and the S-box module is fed with the configured number of state slices with five bits each slice. The result of the S-box calculation is written back during the state shifting. Since the minimum latency of the S-box changes with the protection order and whether the DOM or UMA approach is used, the S-box calculation takes one to 70 cycles.

(4) in the *LINEAR_LAYER* state, the whole linear part of the round transformation is calculated in a single clock cycle. The linear transformation simply adds two rotated copies of one state row with itself. It would be possible to break down this step into smaller chunks to save area. However, the performance overhead and the additional registers required to do so would relativize the chip area savings especially for higher orders.

S-box construction. Ascons's S-box is affine-equivalent to the Keccak S-box and takes five (shared) bits as an input (see Figure 6.8). The figure shows where the pipeline registers are placed in our S-box design (green dotted lines). The first pipeline stage (Stage 0, gray) is optionally already calculated in the *ADD_ROUND_CONST* stage. The registers after the XOR gate in State 0 are important for the glitch resistance and therefore for the security of the design. Without these registers, the second masked AND gate from the top (red paths), for example, could temporarily be sourced twice by the shares of x^1 for both inputs of the masked AND gate. Since the masked AND gate mixes shares from different domains, a timing-dependent violation (glitch) of the d-probing resistance could occur. Note that the XOR gates at the output do not require an additional register stage because they are fed into one of the state registers. As long as no share domains are crossed during the linear parts of the transformation the probing security is given. We assure this by associating each share and each part of the circuit with one specific share domain (or index) and keeping this for the entire circuit.

The other pipelining registers are required because of the latency of the masked AND gates which is one cycle for the DOM gate, and up to five cycles for the UMA AND gate.

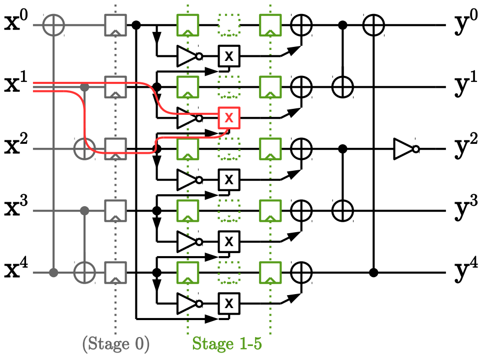

Figure 6.8: Ascon's S-box module with optional affine transformation at input (gray) and variable number of pipeline registers (green)

6.3.1 Implementation Results

All results stated in this section are post-synthesis results for a 90 nm Low-K UMC process with 1 V supply voltage and a 20 MHz clock. The designs were synthesized with the Cadence Encounter RTL compiler v14.20-s064-1. Figure 6.9 compares the area requirements of the UMA approach with DOM for the pipelined Ascon implementation with a single S-box instance. The figure on the left shows the comparison of single masked AND gates inside the Ascon design, while the figure on the right compares the whole implementations of the design. Comparing this results with Table 2.3 reveals that the expected gate counts for DOM match the practical results quite nicely. For the UMA approach, on the other hand, the practical results are always lower than the stated numbers. The reduction results from the fact that the amount of required pipelining registers for the operands is reduced because the pipelining register are shared among the masked AND gates. This does not affect the DOM implementation because the multiplication results are always calculated within only one delay cycle.

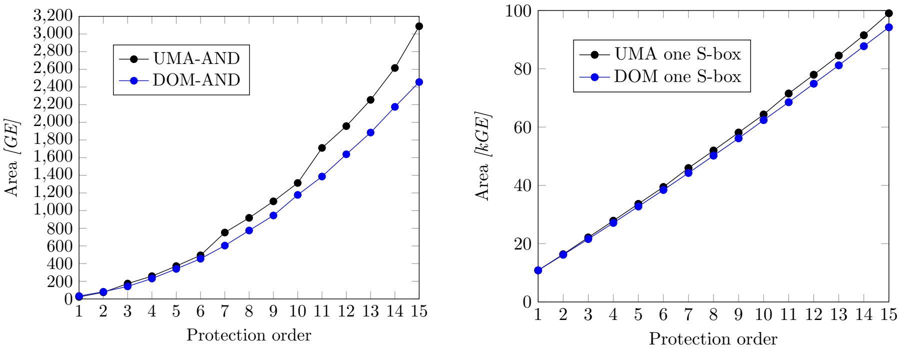

Figure 6.9: UMA versus DOM area requirements for different protection orders. Left figure compares masked AND gates, right figure compares full Ascon implementations

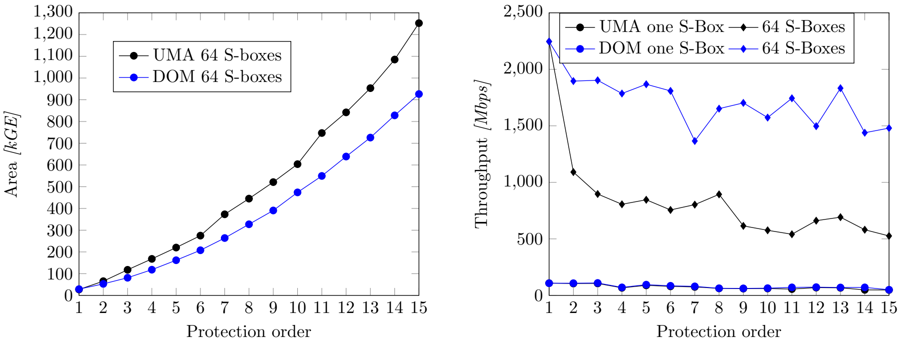

Figure 6.10: UMA versus DOM area requirements for Ascon at different protection orders and 64 parallel S-boxes (left) and throughput comparison in the right figure

The right figure shows that the difference for the single S-box Ascon implementation is relatively low, especially for low protection orders, and seems to grow only linearly within the synthesized range for d between 1 and 15. For the first order implementation, both designs require about 10.8 kGE. For the second order implementation, the difference is still only about 200 GE (16.2 kGE for DOM versus 16.4 kGE). The difference grows with the protection order and is about 4.8 kGE for $d = 15$ which is a size difference of about 5 %. The seemingly linear growth in area requirements for both approaches is observed because the S-box is only a relatively small part with 3-20 % of the design which grows quadratically, while the state registers that grow linearly dominate the area requirements with 96-80 %.

We also synthesized the design for 64 parallel S-boxes which makes the implementation much faster in terms of throughput but also has a huge impact on the area requirements (see Figure 6.10). The characteristics for UMA and

DOM look quite similar to the comparison of the masked AND gates in Figure 6.9 (left) and show a quadratic increase with the protection order. The chip area is now between 28 kGE ($d = 1$) and 1,250 kGE ($d = 15$) for UMA and 926 kGE for DOM. The S-box requires between 55 % and 92 % of the whole chip area.

Throughput. To compare the maximum throughput achieved by our designs, we calculated the maximum clock frequency for which our design is expected to work for typical operating conditions (1 V supply, and 25 °C) over the timing slack for the longest delay path. This frequency is then multiplied with the block size for our encryption (64 bits) divided by the required cycles for absorbing the data in the state of Ascon (for six consecutive round transformations).

The results are shown in Figure 6.10. The throughput of both masking approaches with only one S-box instance is quite similar which can be explained with the high number of cycles required for calculating one round transformation (402-426 cycles for UMA versus 402 cycles for DOM). The UMA approach achieves a throughput between 48 Mbps and 108 Mbps, and the DOM design between 50 Mbps and 108 Mbps for the single S-box variants.

For 64 parallel S-boxes, the gap between DOM and UMA increases because DOM requires only 18 cycles to absorb one block of data while UMA requires between 18 and 42 cycles which is an overhead of more than 130 %. Therefore, also the throughput is in average more than halved for the UMA implementation. The UMA design achieves between 0.5 Gbps and 2.3 Gbps, and DOM Ascon between 1.5 Gbps and 2.3 Gbps.

Randomness. The amount of randomness required for the UMA and DOM designs can be calculated from Table 2.1 by multiplying the stated number by five (for the five S-box bits), and additionally by 64 in case of the 64 parallel S-box version. For the single S-box design, the (maximum) amount of randomness required per cycle for the UMA design is thus between 5 bits for $d = 1$ and 320 bits for $d = 15$, and for DOM between 5 bits and 600 bits. For the 64 parallel S-boxes design, the first-order designs already require 320 bits per cycle, and for the 15^{th}-order designs the randomness requirements grow to 20 kbits and 37.5 kbits per cycle, respectively.

6.3.2 Discussion on the Randomness Costs

In practice, the generation of fresh randomness with high entropy is a difficult and costly task. It is, however, also difficult to put precise numbers on the cost of randomness generation because there exist many possible realizations. The following comparison should thus not be seen as statement of implementation results but reflects only one possible realization which serves as a basis for the discussion.

A common and performant way to generate many random numbers with high entropy is the usage of PRNGs based on symmetric primitives, like Ascon for example. A single cipher design thus provides a fixed number of random bits,

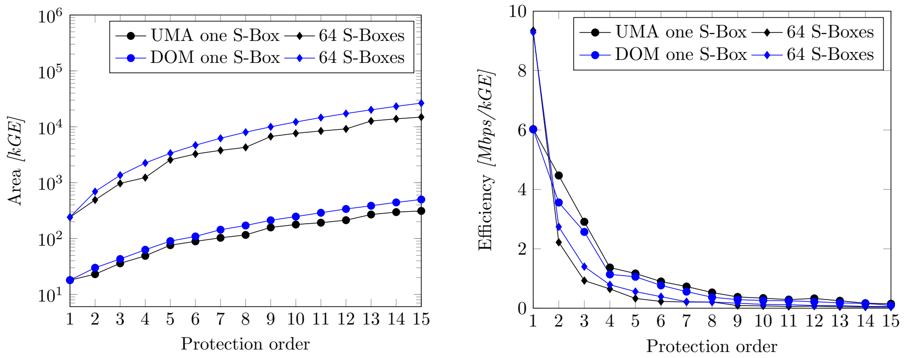

Figure 6.11: UMA versus DOM area requirements for Ascon including an area estimation for the randomness generation in the left figure, and an efficiency evaluation (throughput per chip area) on the right

e.g. 64 bits in the case of Ascon, every few cycles. In the following comparison, we assume a one-round unrolled Ascon implementation resulting in six delay cycles and 7.1 kGE of chip area [Gro+15b]. If more random bits are required, additional PRNGs are inserted, which increase the area overhead accordingly.

Figure 6.11 (left) shows the area results from including the overhead cost for the required PRNGs. Starting with $d = 2$ for DOM, $d = 3$ for UMA for the single S-box variants, and for all of the 64 parallel S-box variants, one PRNG is no longer sufficient to reach the maximum possible throughput the designs offer. The randomness generation thus becomes the bottleneck of the design and additional PRNGs are required, which result in the chip area differences compared to Figures 6.9 and 6.10, respectively. As depicted, both UMA variants require less chip area than their DOM pendants. However, this comparison does not take the throughput of the designs into account (see Figure 6.10).

Figure 6.11 (right) compares the efficiency, calculated as throughput (in Mbps) over the chip area (in kGE). By using this metric, it shows that UMA is the more efficient scheme when considering the single S-box variants, while DOM is the more efficient solution for the 64 S-box variants. However, the practicality of the 64 S-box implementations with up to a few millions of GE and between 30 and 3,600 additional PRNGs is very questionable.

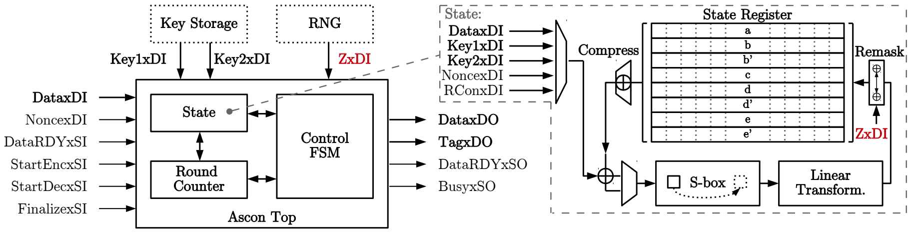

Figure 6.12: Hardware design overview of Ascon

6.4 LOLA-Protected Ascon Implementations

In this section, we integrate the LOLA S-box design from Section 3.4 into a round unrolled variant of Ascon-128. Since the S-box layer in Ascon is only preceded by the linear addition of key or data, and only followed by the linear transformation layer (which both can be securely realized by only operating on each share separately), the shared S-box description can be used to implement a full round transformation of Ascon without any registers in between.

For the sake of completeness, we remark that the combination of the shares created by the shared S-box in Equation 3.3 would not be secure because different share indices are used for some variables. However, this is not an issue for the one-round unrolled Ascon variant because the S-box is calculated column-wise over the state and is only followed by a linear transformation that operates inside one state row. Independence of the cipher rounds is ensured by a resharing after each round transformation. The resharing also includes the creation of the duplicated state rows for the next round by applying the compression two times in parallel for these variables with fresh and independent randomness (ZxDI) from the RNG.

Design description. Figure 6.12 depicts our top module of the Ascon core. The structure is based on the one used in Section 6.3 for the DOM and UMA designs. The majority of changes are done in the state module (right). The round transformation is no longer distributed over (at least) three clock cycles but is performed in a single step. Due to the S-box layer, the amount of shares increases from $d + 1$ to $(d + 1)^2$ for the linear layer which is followed by a remasking according to the CMS scheme of Reparaz et al. [Rep+15]. The CMS remasking requires one fresh random bit per share which amounts to $8 \cdot 64 \cdot (d + 1)^2$ bits in total for our design. Before the compression to $d+1$ terms can be performed, the $(d+1)^2$ refreshed shares are stored in the state registers which includes duplicated state rows needed for the S-box layer in the next round transformation. The number of state registers is therefore increased from $5 \cdot 64 \cdot (d+1)$ to $8 \cdot 64 \cdot (d+1)^2$ compared to the DOM and UMA variants which is partially compensated by the registers which are not required for the S-box layer.

Another change affects the key storage which now needs to supply an additional copy of the key since the key is combined with the state during the initialization and the finalization and is used in parts of the state that need to be copied for the secure S-box transformation.

Results and comparison. The post-synthesis results for a 90 nm Low-K UMC process with 1 V supply and a 20 MHz clock synthesized with the Cadence Encounter RTL compiler v14.20 of the LOLA designs are given in Table 6.3. The design is generic in terms of protection order (d), but since the number of registers grows quadratically with the protection order, we only considered results up to order five. For all protection orders, only six cycles per encryption or decryption are required which is three to seven times fewer than for the (64 parallel S-boxes) DOM and UMA designs.

Unrolling one round produces much more combinatorial delay which results in a lowered maximum clock frequency. Nevertheless, also the throughput is in all cases increased over related work. While for first-order the throughput is only slightly increased, the difference becomes much more significant for order five for which the throughput is almost doubled over the DOM design and 3.5 times higher than for the UMA design. The price for the reduced latency is an increased chip area (about 15 kGE overhead for the first-order variant, and double the amount of area over DOM for order five), and an increased randomness consumption which is between 5.2 (UMA, order five) and 6.4 (DOM/UMA, first order) higher.

Table 6.3: Results for ASCON-128 with one cycle per round (64 S-boxes)

Design	Size *[kGE]*	Cycles *[Cycles/Round]*	Max. Throughput *[Gb/s]*	Randomness *[bits/cycle]*
1st-order	42.75	1	2.77	2,048
2nd-order	90.94	1	3.35	4,608
3rd-order	153.91	1	3.34	8,192
4th-order	238.30	1	2.59	12,800
5th-order	339.82	1	2.99	18,432
Related work				
1st-order UMA	27.18	3	2.25	320
1st-order DOM	28.89	3	2.25	320
5th-order DOM	161.87	3	1.86	4,800
5th-order UMA	220.01	7	0.85	3,520

Discussion. We admit that the randomness requirements for the higher-order variants become very high but we denote two things:

1) Our LOLA approach offers a new design choice that a designer of a masked circuit can use to trade-off area and randomness against less latency. We used one corner case to demonstrate the feasibility of the approach by targeting one cycle

per round transformation. A designer, of course, could also target a two-cycle variant by using the resharing e.g. after the S-box or by inserting registers after the affine transformation in the S-box to save randomness and area.

2) The CMS resharing function is not an ideal choice from multiple perspectives. As it was pointed out by Moos et al. [Moo+18], the CMS resharing is only secure up to the second protection order, and therefore not scalable to any protection order as required by LOLA. We nevertheless decided to use the hardware and randomness numbers, even for protection orders greater than two, as estimation for the implementation costs because of the lack of suitable alternatives. A DOM-like resharing, for example, could possibly reduce the randomness amount, e.g. for first order by a factor of 4 which would reduce the randomness to 512 bits per cycle. On the other hand, using the DOM resharing would require a deeper analysis of the design over at least two rounds. Therefore, we made the choice to use the CMS resharing at this point and denote the use of a generic, secure, and more efficient resharing function as one interesting practical extension of our work.

7

KECCAK Secure Hash Algorithm (SHA3)

KECCAK is a family of sponge functions, from which several instantiations have been standardized by NIST as SHA-3, SHAKE and KECCAK-p in [NIS15], as a result from the SHA-3 hash competition. The SHA-3 specification describes four different instantiations of the KECCAK-f [1600] hash function. The KECCAK family [NIS15] consists of seven permutations KECCAK-f [b], for $b \in \{25, 50, 100, 200, 400, 800, 1600\}$, where b denotes the width of the permutation. These permutations are organized in a sponge construction, which also allows to express the KECCAK permutation in terms of rate r and capacity c as KECCAK [r,c], for $b = r + c$. The rate in the sponge construction corresponds to the block size and can be a multiple of a lane size, while its capacity determines the security level as $c/2$.

Although the seven permutations of KECCAK-f [b] have different permutation widths, their underlying round function is always the same. A full round of KECCAK consists of the five steps θ, ρ, π, χ and ι, which operate on the three-dimensional state in this order. A full permutation is defined as the repeated application of these five steps.

- θ is a linear diffusion step. It calculates the parity of each column in a slice and adds it to a neighboring column in the same and the next higher slice.
- ρ and π are also linear diffusion steps, most often implemented directly by wiring in a hardware implementation.
- χ is a degree-2 non-linear mapping. It operates on each row of the state independently and is implemented as $x^i \leftarrow x^i \oplus (x^{i+1} \oplus 1)x^{i+2}$.
- ι is a simple addition of a round constant to a lane.

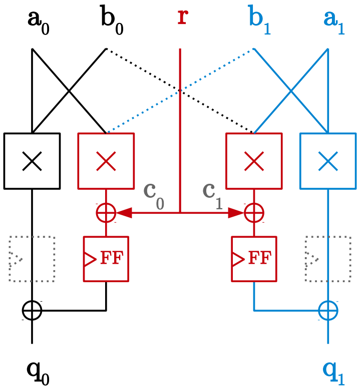

Figure 7.1: First-order DOM multiplier calculating $q = ab$, and with randomness optimization for $q = ab \oplus c$ calculation (gray)

7.1 DOM Optimizations

In order to make our protected KECCAK implementation more efficient, we introduce different optimizations to decrease the overhead in terms of area, delay, and required randomness.

Randomness Optimization. DOM was intended for the efficient and generic higher-order protection of hardware designs. This genericity, however, leads to an unnecessary overhead in terms of fresh randomness for first-order protection of some S-box constructions. In particular, this affects S-boxes of the form $ab \oplus c$ which is the case e.g. in the S-box of KECCAK, Present [Bog+07], Noekeon [Dae+00], and LowMC [Alb+15] .

The 5-bit KECCAK S-box is given as $x^i \oplus (x^{i+1} \oplus 1)x^{i+2}$ which is of the form $ab \oplus c$. In a first-order DOM-protected KECCAK S-box (see Figure 7.2), the straightforward implementation requires 5 bit of randomness for each S-box to calculate $x^{i+1}x^{i+2}$. However, the shares of x^i are independent from the ones of x^{i+1} and x^{i+2} and thus the shares of x^i fulfill the same property as a fresh random r share. Instead of adding r to the cross-domain terms in both domains, we add the shares associated with x^i as c_0 and c_1 to these cross-domain terms in their respective domain (see Figure 7.1).

Saving randomness by reusing unrelated state bits has already been reported for first-order threshold implementations by Bilgin et al. [Bil+14b] and more recently by Daemen et al. [Dae17]. The difference is that Daemen's changing-of-the-guards approach performs an explicit resharing at the end of the S-box function for two out of five S-box bits, which requires additional XOR gates, while we perform this implicitly.

While the probing security of the construction in Figure 7.1 is trivially given under the assumption that the sharings of the input bits (a, b, and c) are

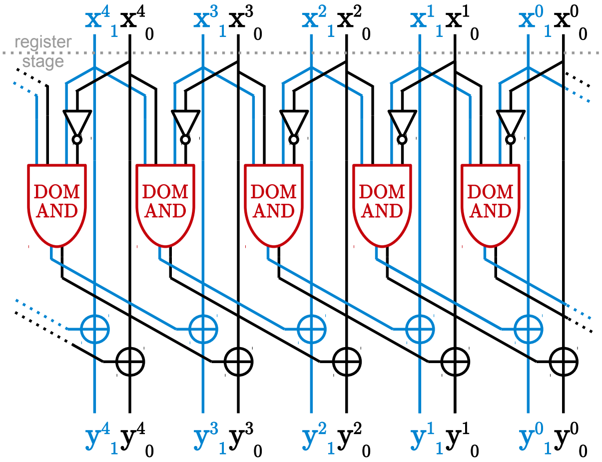

Figure 7.2: First-order protected S-box of Keccak with the DOM multiplier from Figure 7.1

independent, the security argumentation gets more difficult when the rest of the Keccak transformations and the full S-box implementation is considered. Indeed, the uniformity of the state bits and therefore their suitability for masking other operations degenerates over the rounds of Keccak as stated by Daemen. As this effect is considered to be minimal by Daemen [Dae16] and therefore unpractical to exploit, we still consider it as a valid option in our design. However, we note that, formally speaking, this optimization leads to a flaw in the probing assumption. Using this optimization requires careful investigation of the degeneration effect for the targeted design or usage of the changing-of-the-guards method from Daemen [Dae17]. Thus we made the usage of this optimization optional for our first-order Keccak instantiation.

Throughput and Area Optimization. The DOM multiplier always introduces a delay cycle through the resharing of the cross-domain terms. To make this calculation more efficient, another register could be added in the inner-domain paths of the multiplier to generate a pipeline stage (gray dotted registers in Figure 7.1). However, this has a negative influence on the required chip area if many S-box instances are used in parallel. We circumvent the additional register by clocking the cross-domain flip-flops on the negative clock edge, which effectively means doubling the clock frequency for the S-box. We investigate the effectiveness of this approach in Section 7.3 and investigate its influence on the maximum throughput. Furthermore, we note that this optimization is only a meaningful option if the critical path in the S-box is relatively small compared to the other circuit parts, or if the overall clock frequency of the chip is somewhat already constrained. Otherwise this optimization might have a negative influence on the layout of the design to meet the clock constraints.

7.2 Implementation

Our generic Keccak implementation allows to be customized for a variety of requirements for different security-critical applications. In the following, all possible configurations of the Keccak design are explained, and the main variants are introduced.

A high-level architectural view of the design is shown in Figure 7.3. It is important to note that each mapping is either directly connected to the next one *or* to the sponge state. The configuration of the connection is done at synthesis time. Hence the connection between the steps, respectively between a mapping and the sponge state, is done by simple wiring. Everything from a fully parallel implementation, in which all five steps are done in one clock cycle, to a fully iterative one, in which the output of each step is written back into the state, can be instantiated. For example, when omitting the gray connections shown in Figure 7.3, the Serial-Area configuration, which is described later in this Section, is obtained.

Iterative Application of Functions. The design allows to apply the individual round transformation steps in an iterative way, which takes either multiple cycles, but can also be performed in a single clock cycle. In case a step is handled iteratively it will act on a specified number of state slices (in powers of two) in parallel. This means that the state can be thought of as a number of FIFOs. The FIFO's output form the input of the iteratively applied step, and the output of the step gets either piped into the input of the following step (which would in turn need to be iterative), or back to the FIFO. The specified number of parallel slices is used for all iterative steps (except for an iterative ρ/π step which is explained further below). This simplifies the design by allowing to chain the slice-based iterative χ, ι, θ and absorption steps together.

The iterative version of the steps looks as follows.

- In θ, one output slice depends on the parity of a previously processed slice's columns. This can simply be handled by storing the parity of the highest processed slice of the previous cycle. This works for every slice except for the first one, which can only be finished once we look at the last slice. Thus the first slice is a special case and is finished together with the last one.
- An iterative ρ step means that each lane gets rotated until the desired offset is reached. This does imply some control overhead, but allows to save most of the multiplexers which are needed when $\pi \circ \rho$ is performed in one cycle, which requires the full state to be written at once.
- The π mapping is then applied together with χ, which works out nicely, since both are slice-based functions and π is implemented by simple wiring.
- The iterative ι step is simply done by adding only the relevant part of the round constant. In our implementation ι is always done concurrently with the S-box function χ.

It is also possible to choose the lane length freely, which means, that any Keccak-f [b] variant can be instantiated.

Absorption. A concrete instantiation of our design can perform the absorption either in a lane-based or a slice-based fashion. In case of a slice-based absorption, the absorbed slice(s) can be fed directly into an iterative θ step, which saves cycles that otherwise would be wasted solely for absorption. In the case of a lane-based absorption, such optimizations are not possible. However, lane-based absorption often fits much more naturally with how data is processed and sent over buses, hence possibly saving area or increasing overall performance, depending on the concrete system. To avoid having to include additional buffers in case of systems with bus widths unequal to the lane length, it is possible to adjust the number of bits absorbed in a single cycle (in powers of two). This means, that it is possible to absorb more than one lane at once, or only a fraction of a lane, as long as the word to absorb is a power of two, depending on the configuration.

Concrete instantiations. Since the number of configurations which are possible with this approach is huge, we focus on three corner cases.

- Serial-Tp All steps except ρ and π are performed iteratively. The absorption is done in a slice-based manner, in parallel with the θ step in the first round. ρ and π are done in a separate step. The χ and ι steps are chained together with the absorption XORs and the θ step. Thus the processing of a block takes $r(\frac{W}{SP}+1)$ cycles, where W is the lane length, SP are the number of parallel processed slices and r the number of rounds. This implementation is similar to the one described by Bilgin et al. [Bil+14b] but performs $\pi \circ \rho$ in a dedicated cycle instead of concurrently with the last cycle of θ.

- Serial-Area This variant is similar to Serial-Tp but every step is done iteratively (including ρ and π). In the iterative ρ step, each lane gets rotated until a counter exceeds the rotation offset of that lane, hence this step now takes W cycles to complete. This saves most of the multiplexers which are needed when $\pi \circ \rho$ is performed in one cycle, as the whole state does not have to be updated at once. This simple modification yields the smallest register-based implementation of Keccak to date. As a trade-off, throughput is decreased, compared to Serial-Tp, since ρ now takes W cycles to complete.

- Parallel This variant is a fully parallel implementation in which we try to achieve the highest possible throughput. The unprotected instantiation performs $\iota \circ \chi \circ \pi \circ \rho \circ \theta$ in one cycle. The DOM-masked instantiations require an additional flip-flop stage before the S-boxes, otherwise glitches in the θ step would lead to a violation of the probing security during the application of the χ step. Hence for one round of Keccak $\pi \circ \rho \circ \theta$ is performed in one cycle and $\iota \circ \chi$ in the next. Such a configuration yields the highest throughput but on the other hand requires $5 \cdot 2^l$ 5-bit S-boxes.

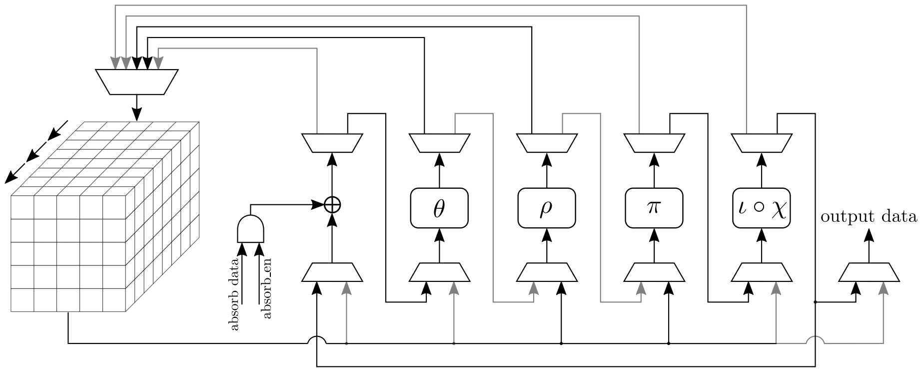

Figure 7.3: Simplified architecture of our implementation

S-Box Variants. For each of the previously described configurations, we further instantiated two different variants that differ in the implementation of the S-Box in the χ step.

- *Pipelined* DOM variant. In this variant we use additional inner-domain flip-flops together with the cross-domain flip-flops as a pipeline stage for the multiplier in Figure 7.1.
- *Double-clocked* DOM variant. Here we try to keep the area overhead of DOM minimal by saving the inner-domain flip-flops and clocking the cross-domain flip-flops on the negative clock edge as described in Section 7.1.

Both S-box variants use the optional optimization to reduce randomness in the first-order case, as described in Section 7.1. For the SERIAL configuration, the overhead of the inner-domain flip-flops is negligible when only one slice is processed in parallel. Thus only the pipelined S-box variant is shown in the results for these instantiations.

Area Estimation. For the protected implementations, the linear parts (except for the inverts) need to be replicated for each additional domain. Hence the linear θ, ρ, π mappings, as well as the state itself are expected to scale linearly with the number of shares $(d + 1)$. For the non-linear χ step, a more detailed look at the S-Box is required. An unprotected S-Box consists of 5 AND, 5 NOT and 5 XOR gates as can be obtained when only looking at domain A (black parts) in Figure 7.2 and replacing the DOM AND instances by normal AND gates. An estimation for the scaling of the χ step can be given by looking at the generic construction of the DOM AND gate in Equation 1.3, and replicating the XOR gates for each share, as illustrated in Figure 7.2 for two shares. Generally speaking, for $d+1$ shares, the combinatorial part of the S-Box consists of $5(d+1)^2$ AND gates, 5 NOT gates and $5(d + 1) + 5(d + 1)^2$ XOR gates. Simplifying this by looking only at the DOM AND gate ($(d + 1)^2$ ANDs and XORs), we can estimate the combinatorial part to increase with a factor $(d + 1)^2$. The number of flip-flops in the χ step depends on whether the implementation is pipelined (uses the inner-domain flip-flops) or not. The pipelined variant requires $(d+1)^2$ flip-flops, the variant without inner-domain flip-flops requires $d(d+1)$.

7.3 Results

The synthesis results are obtained with the configurations described in Section 7.2. We apply the configurations to the KECCAK [1088,512] permutation, since it is a SHA3 standard, and allows comparisons with other publications. All values have been obtained by synthesizing the design with Cadence RTL Compiler version 8.1 XL. We used the FSC0H_D and FSD0A_A libraries from FARADAY for the 130 nm and 90 nm designs respectively. The numbers given in plots correspond to designs synthesized with the 130 nm library.

A detailed look at the synthesis results, up to the second protection order, is given in Table 7.1. The KECCAK team itself were the first to provide a first-order protected KECCAK threshold implementation with three shares [Ber+12]. The implementation was later on improved by Bilgin et al. [Bil+14b] resulting in the smallest register-based[1] protected and unprotected KECCAK designs reported up to now.

Area Requirements. The SERIAL configurations in Table 7.1 show the resulting numbers when processing a single slice per cycle. This allows to directly compare our serial designs with the one of Bilgin et al. [Bil+14b], which also processes one slice per cycle. Their unprotected design has a size of just 10.6 kGE for their serial implementation while our unprotected variants use 11 kGE in case of SERIAL-TP, and 9.2 kGE in the SERIAL-AREA implementation for the cost of a doubled cycle count. This makes our SERIAL-AREA configuration the smallest register-based KECCAK implementation reported so far.

When looking at the SERIAL configurations' increase in size between unprotected (11.0/9.2 kGE), first-order (22.3/18.7 kGE) and second-order (34.6/28.8 kGE) protected SERIAL designs in Table 7.1, it can be seen that all linear parts grow linearly with the protection order as expected. This is also illustrated in Figure 7.4, which shows that the area requirement increases almost linearly with the protection order for the SERIAL designs. The only non-linear part of the design is the χ step, which only operates on one slice in the SERIAL configurations as shown in Table 7.1, and thus the χ transformation has only a marginal influence on the overall size.

For the PARALLEL configurations, the linear parts of the design grow linearly with the protection order as well. The non-linear χ step now operates on the full state, hence $64 \cdot 5$ 5-bit S-Boxes are required, making it the main contributor to the overall area.

As discussed in Section 7.2, the area of the DOM-protected χ step increases non-linearly with the number of shares. This can best be observed

---

[1] The smallest design this far is achieved by the usage of RAM macros and needs significantly more cycles per block [PH13].

		UMC 0.13μm					UMC 90nm						
		Area (kGE)				Freq.	Area (kGE)				Freq.		
Prot. order	Design	θ	χ	State/Other	$\sum$	(MHz)	θ	χ	State/Other	$\sum$	(MHz)	Cycles	Rand.
None	Parallel	8.6	6.4	16.2	31.2	919.1	7.4	6.0	14.0	27.4	1,287.0	24	-
	Serial-Tp	0.3	0.1	10.6	11.0	866.6	0.2	0.1	9.6	9.9	861.3	1624	-
	Serial-Area	0.3	0.1	8.8	9.2	861.3	0.2	0.1	7.4	7.7	900.9	3136	-
1st order	Parallel double-clocked [2]	17.2	44.2	38.9	100.5	803.9	15.0	38.4	32.2	85.7	891.3	48	-
	Parallel pipelined [2]	17.2	57.6	36.8	111.8	837.5	15.0	50.4	32.2	97.7	846.7	72	-
	Serial-Tp pipelined	0.6	0.9	20.8	22.3	812.3	0.4	0.8	18.9	20.1	864.3	1648	-
	Serial-Area pipelined	0.6	0.9	17.1	18.7	856.2	0.4	0.4	14.5	15.7	850.3	3160	-
2nd order	Parallel double-clocked [2]	26.0	139.9	60.1	226.0	811.7	22.5	114.0	51.1	188.1	897.7	48	4800/cycle
	Parallel pipelined [2]	25.8	157.2	55.1	238.4	840.3	22.5	138.0	48.2	208.9	848.9	72	4800/cycle
	Serial-Tp pipelined	1.0	2.5	31.1	34.6	844.6	0.6	2.2	28.1	30.9	845.3	1648	75/cycle
	Serial-Area pipelined	0.9	2.5	25.4	28.8	852.5	0.6	2.2	21.4	24.2	898.5	3160	75/cycle
Related Work													
None	Parallel [Bil+14b]	8.6	6.4	15.6	30.6	855	-	-	-	-	-	24	-
	Serial [Bil+14b]	0.1	0.1	10.4	10.6	752	-	-	-	-	-	1600	-
1st order	Parallel-3sh [Bil+14b]	25.7	52.8	56.7	135.2	746	-	-	-	-	-	25	4/round
	Parallel-4sh [Bil+14b]	34.2	61.6	61.8	157.6	735	-	-	-	-	-	24	-
	Serial-3sh [Bil+14b]	0.4	0.8	31.4	32.6	820	-	-	-	-	-	1625	4/round
	Serial-4sh [Bil+14b]	0.5	0.9	41	42.4	775	-	-	-	-	-	1600	-

Table 7.1: Synthesis results

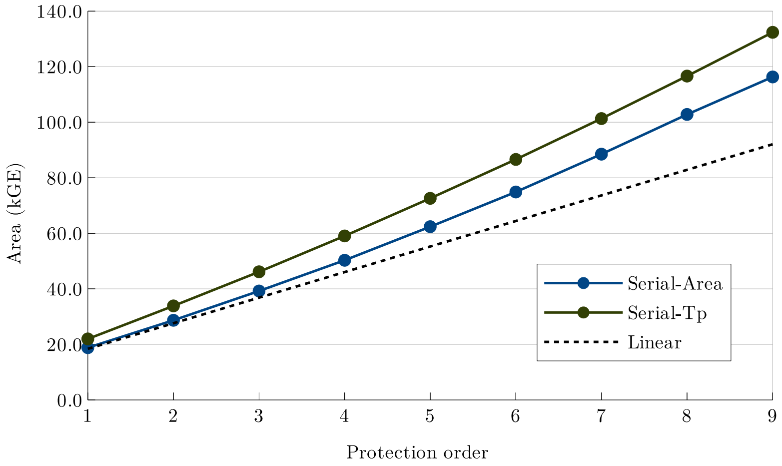

Figure 7.4: Area requirement for increasing number of share domains. Serial with 1 slice processed in parallel with pipelined S-box

in the area increase from the unprotected Parallel to the protected Parallel configurations. The unprotected χ step (6.4 kGE) consists solely of combinational logic, a 1600-bit state of small flip-flops is around 8.8 kGE (see Serial-Area in Table 7.1). Hence, as was discussed in Section 7.2, a rough area estimation for the χ step of the parallel Keccak implementation with $d+1$ shares would be $(d+1)^2 \cdot (6.4 \text{ kGE} + 8.8 \text{ kGE})$ for the pipelined and $(d+1)^2 \cdot 6.4 \text{ kGE} + d(d+1) \cdot 8.8 \text{ kGE}$ for the double-clocked variant. In case of a first-order protected implementation this corresponds to 60.8 kGE, respectively 43.2 kGE, which is close to the actually achieved results (57.6 kGE, resp. 44.2 kGE).

Compared to the existing implementations, our designs have a lower area overhead for the same protection order, while achieving similar throughput, in all configurations. The main reason for this difference is that related work uses the $dt+1$ TI approach which requires at least three input shares for first-order protection while our $d+1$ share implementations require only two shares in the first-order case.

Throughput Considerations In case higher throughput is desired it is also possible to increase the number of slices that are processed in parallel as mentioned in Section 7.2. Figure 7.5 and Figure 7.6 show how the area and maximum frequency develop, respectively, when doing so for the pipelined Serial-Tp configuration. Note that while the area and maximum frequency for Serial-Area would look similar, the throughput gain would be lower, due to ρ always taking 64 cycles.

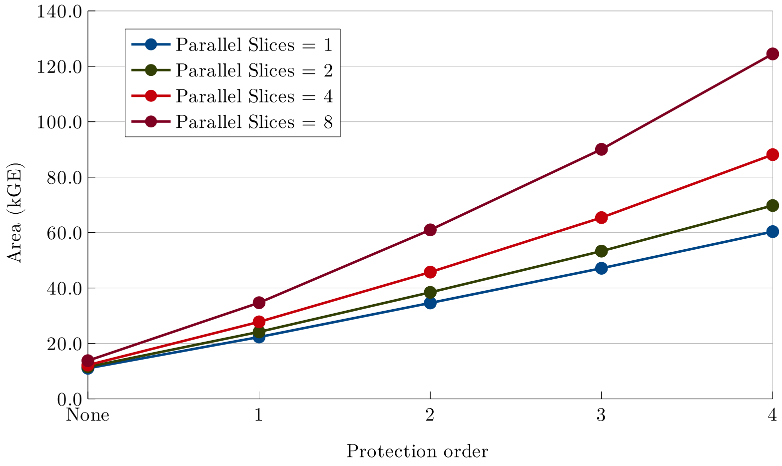

Figure 7.5: Serial-Tp: Area over the number of share domains for different number of parallel processed slices

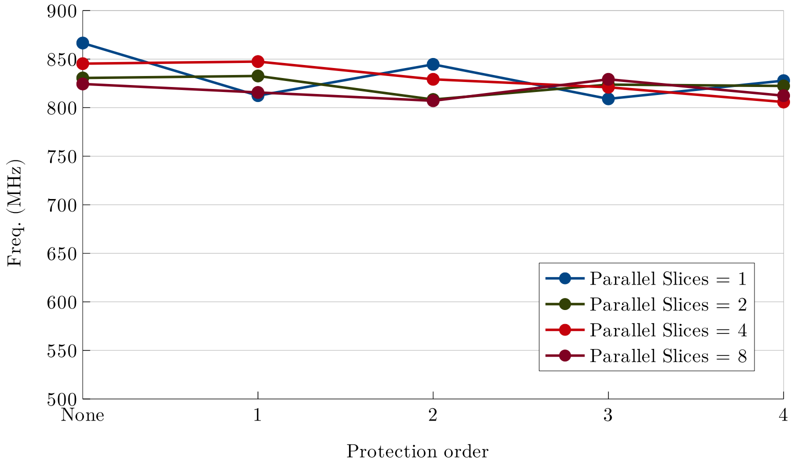

Figure 7.6: Serial-Tp: Frequency over the number of share domains for a varying number of parallel processed slices

The throughput of the Serial-Tp configuration doubles if the number of parallel processed slices doubles. This, of course, also doubles the number of needed S-boxes, thus the area increase with higher protection order becomes less linear. Compared to the first-order protected $dt + 1$ TI in Table 7.1 that needs 32.6 kGE , the DOM-protected counterpart uses just 22.3 kGE.

The highest throughput is achieved with the Parallel configuration, which needs the full 1600 S-boxes. The area requirement can be lowered by implementing the double clocking of the S-box as described in Section 7.1. As shown in Table 7.1, the area of the double-clocked S-box variant (100.5 kGE resp. 226.0 kGE) is notably smaller than the area of the pipelined S-box variant (111.8 kGE resp. 238.4 kGE).

8

RISC-V Processor

This implementation builds upon the V-scale processor that implements the RISC-V Instruction-Set Architecture (ISA), which was originally developed at the University of California, Berkeley. RISC-V is a customizable, modular, free and open RISC ISA. Its architecture is highly flexible, meaning that the register sizes (32, 64, or 128 bit), their number (16 or 32), the number of privilege levels (1 to 4), and the supported instructions can be chosen according to the desired use case.

The ISA defines the mandatory base-integer instruction set (I or E) which contains the most basic memory, arithmetic, logic, and control-flow instructions. Optionally, more complex instructions can be implemented and are defined via various standard extensions. These extensions include, for example, instructions for integer multiplication/division (M), atomic (A) operations, as well as single- (F) and double-precision (D) floating-point computations. The instructions in the base instruction set and the mentioned extensions are all encoded in 32 bits. However, both shorter and longer instructions are supported, too. The extension for compressed instructions (C), for example, defines 16-bit instructions, which map to the base instruction set, to increase code density. Furthermore, RISC-V also supports the addition of fully customized instructions as so called non-standard extensions (X).

The fact that RISC-V, unlike for example the AVR, x86, and the ARM ISA, has no status flags (carry, overflow, zero, ...) is noteworthy too, given that it simplifies the masking efforts. Carry propagation as well as comparisons are performed with dedicated instructions instead.

Like the ISA, also the V-scale processor core has been developed in Berkeley. V-scale is a Verilog implementation of the RV32IM instruction set, i.e., it is a RISC-V processor with 32 registers with a width of 32 bits featuring the base-

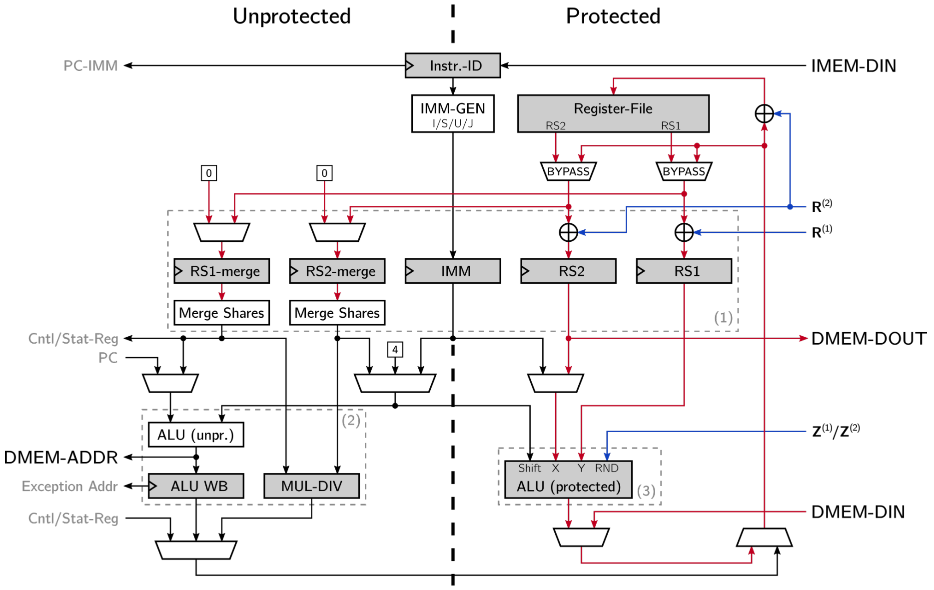

Figure 8.1: Overview of the V-scale core. Gray blocks are registers or use a register stage internal. Shared data connections are illustrated in red, unshared in black and the randomness in blue

integer instruction set and the integer multiplication extension. The core itself relies on a single-issue in-order 3-stage pipeline comprising a fetch, a combined decode+execute, and a write-back stage. Additionally, the data dependencies between consecutive instructions can be resolved using a bypass of the write-back stage which permits to maximize the utilization of the core. Communication with memory relies on separated AHB-Lite memory interfaces for instructions and data, permitting to build Harvard and von Neumann architectures.

8.1 Protected Implementation of V-scale

Our protected implementation of V-scale addresses the problem that data processed by the processor is subject to side-channel attacks. In our design, we solely protect the instructions of the base RV32I instruction set as it is the most versatile one. Nevertheless, the multiplication/division (M) extension of the original V-scale processor has been kept to maintain compatibility but is still unprotected.

Therefore, the register file, the majority of the Arithmetic-Logic Unit (ALU) and the data memory interface of the V-scale processor have been protected using the DOM scheme. Other parts, like the instruction memory interface and the decoder, have been left unprotected. The reason for this split is that in any case the implemented code must be written in such a way that it does not leak information about the processed data over the instruction sequence because different instructions show different power signatures in leakage traces. Otherwise, even on a fully shared processor, timing attacks would for example be possible.

The resulting processor's architecture is depicted in Figure 8.1. One major difference to the original V-scale processor is that the protected core now has four pipeline stages. The additional pipeline stage (see (1) in Figure 8.1) splits the previously combined decode+execute stage and is necessary to prevent leakage due to glitches when data shares are merged. This aspect is described in more detail in Section 8.1.1.

From another perspective, the processor is split into a part that operates on DOM-shared data and a part operating with merged data shares. Accordingly, the ALU itself has been split into a protected and an unprotected part. The unprotected ALU (see (2) in Figure 8.1) implements multiplication/division, address calculation, and data comparison for conditional jumps. Performing comparisons for conditional jumps in an unprotected way is legitimate as code is not allowed to branch on secure data anyway in order to avoid timing attacks. More details on the logic to securely merge the different DOM shares and on the unprotected ALU itself can be found in Section 8.1.2. All the remaining functionality being part of the base instruction set (e.g. AND, OR, XOR, ADD, ...) is implemented in the protected ALU in a DOM-protected way. The protected ALU is visualized in Figure 8.1 at (3) and is thoroughly described in Section 8.1.3.

8.1.1 Additional Pipeline Stage

The major change compared to the unprotected processor consists in the additional source registers shown in Figure 8.1 at (1). The main purpose of these buffer registers is to prevent glitches in the merging units connected to *RS1-merge* and *RS2-merge*. These merging units recombine the shares to the original value as shown in Equation 1.

Without the registers *RS1-merge* and *RS2-merge*, (de-)activation of the merging units can result in data-dependent glitches. This is illustrated using two basic scenarios. First, the output of the register file switches to sensitive data. This requires the merging units to be disabled by detaching their inputs from the source register. However, if the sensitive data is selected faster than the merging unit is disabled, sensitive data propagates into the merging unit which results in the leakage of sensitive data. Second, the output of the register file switches from sensitive data to data to be merged. This enables the merging unit by switching the multiplexer to the output of the register file. Here, if the multiplexer switches faster than the register file output is selected, the sensitive data from before glitches into the merging units which leaks information. Both scenarios are prevented by the additional buffer registers *RS1-merge* and *RS2-merge*. These effectively decouple the merging units from the register file selector by setting the input to the merging units to zero if not required. To adapt the delay of the protected to the unprotected data path, further buffer registers *RS1* and *RS2* are needed.

Another change to the processor design is the addition of fresh randomness to the processed values before the ALU result is written back to the register file and before the registers *RS1* and *RS2* are used as operands for the protected ALU. This allows to restore the independence of the sharings after unprotected operations and shifts operations which generate zeros or duplicate the most significant bit, respectively. Furthermore, the addition of fresh randomness is required right before operating on identical operand registers for protected ALU operations.

8.1.2 Unprotected Operations

Figure 8.1 shows at (2) the modules *MUL-DIV* and *ALU (unpr.)* providing the unprotected operations of our core. These modules operate natively with 32-bit word size and use the merged data as described in Section 8.1.1. The *MUL-DIV*-module is the unprotected hardware multiplication and division unit from the original V-scale processor design and kept to maintain compatibility.

The unprotected ALU implements different compare operations, e.g., for branch instructions. However, the comparison results can also be written back to a register. While all branch instructions use two source register inputs, instructions storing the comparison result allow to alternatively use an immediate value as the second source. Note that the compare functionality could have been implemented without merging the data, but branching on protected data must anyway be avoided due to possible timing attacks [Koc96]. This design decision should be kept in mind as it makes it necessary to avoid compare operations on protected data.

Furthermore, the unprotected ALU provides an adder to perform address calculations within load and store operations. Note, however, that the required merging of source register before the actual address computation does not reduce security. As the second operand is constant and determined by a known software implementation, the value of the source register can always be reconstructed, also if a masked adder was used and the shares of the memory address were merged afterwards. Besides, the unprotected adder is also used within two further instructions. First, the adder is used in the jump and link instruction to increment the program counter in the computation of the address of the following instruction. Second, in the “add upper immediate to program counter”-instruction both the program counter and the immediate input are publicly known making a masked adder obsolete.

8.1.3 Protected ALU

The protected ALU is shown in Figure 8.2 which provides the masked functionality for bit-wise logic operations and arithmetic operations. Both input sharing vectors $\mathbf{X}$ and $\mathbf{Y}$ are composed of $d+1$ independent shares (see Equation 8.1), where d is the protection order of the DOM implementation. For resharing purposes, the protected ALU has two additional inputs $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$ holding the required

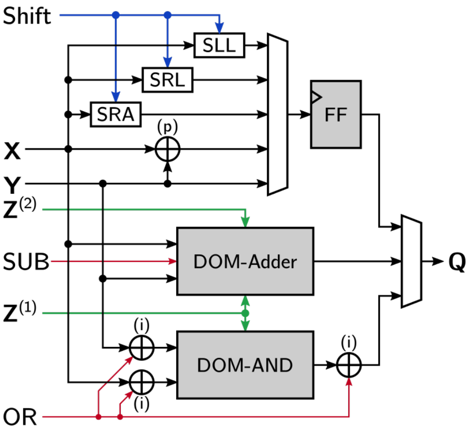

Figure 8.2: Protected ALU using a single DOM-AND for AND and OR operations. Shown XOR operations used in different manner: (p)airwise XOR operation of input shares (e.g. $x_0 \oplus y_0; x_1 \oplus y_1; \ldots$); (i)nverting the operand xoring the signal OR with every element of the corresponding first share

fresh random shares. The data width of the input shares and the fresh random z shares is 32 bits each.

$$\mathbf{X} = \underbrace{(x_0, x_1, x_2, \ldots)}_{d+1} \qquad \mathbf{Y} = \underbrace{(y_0, y_1, y_2, \ldots)}_{d+1} \tag{8.1}$$

$$\mathbf{Z}^{(1)} = \underbrace{(z_0^{(1)}, z_1^{(1)}, z_2^{(1)}, \ldots)}_{d(d+1)/2} \qquad \mathbf{Z}^{(2)} = \underbrace{(z_0^{(2)}, z_1^{(2)}, z_2^{(2)}, \ldots)}_{d(d+1)/2} \tag{8.2}$$

DOM-AND

The basis for all implemented non-linear operations is the DOM $GF(2)$ multiplier (see Section 1.1) which corresponds to a logic AND gate with two one-bit inputs. A basic requirement of the DOM multiplier is that the two inputs **X** and **Y** are independently shared which is ensured by design of the protected core.

The construction of the *DOM-AND* is generic and can thus be extended to arbitrary protection orders by adding additional shares. For the protection order d, $d+1$ shares per variable are required giving $d+1$ independent share domains. Every domain consists of $d+1$ AND gates and flip-flops which results in a quadratical growth of the chip area accordign to the protection order. The three steps (calculation, resharing, and integration) of the DOM implementation are applied independently for every bit position of the 32-bit shares. Therefore, a 32-bit AND gate consists of 32 *DOM-AND* gates.

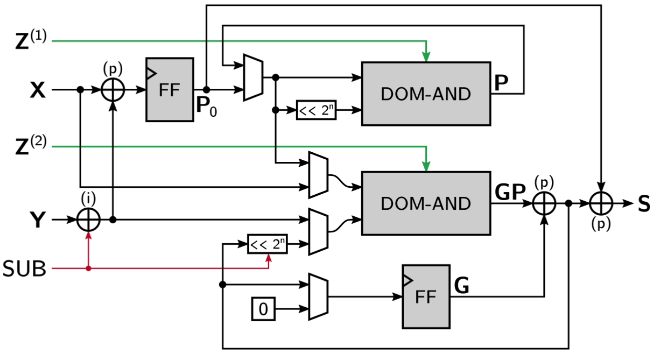

Figure 8.3: Masked adder using two DOM-AND. Shown XOR operations used in different manner: (p)airwise XOR operation of input shares (e.g. $x_0 \oplus y_0; x_1 \oplus y_1; \ldots$); (i)nverting the operand by xoring the signal SUB with every element of the first share of **Y** (y_0).

DOM-Adder

The protected adder is based on a Kogge-Stone design, similar to the construction of Schneider et al. [Sch+15]. The adder is a carry-lookahead-type adder using a tree-like structure separating the addition into propagation and carry generation. Figure 8.3 shows the secure DOM adder. It is composed of two *DOM-ANDs*, two bit shifts, and multiple XORs. The XOR as well as the shift operations can be performed independently for each share domain and each input. The nonlinear parts of the adder are formed by two *DOM-AND* gates. To make the illustration of the adder in Figure 8.3 more concise, the three steps for calculating the DOM-AND are only indicated by the respective function. The *DOM-AND*'s internal registers together with the **G** are used as the working registers for the iterative calculation of the sum. The *DOM-AND*'s internal registers are indicated by **GP** which belongs to the carry generation path and **P** which belongs to the propagation path.

For the carry generation path, the register **G** is used to store the previous value of the generation step as it is required in the next iteration. An important requirement of the used *DOM-AND* gate is an independent sharing of both inputs. This independence is ensured for both AND gates because the bit position of one operand is always shifted by at least one position. With the same argument, the random shares in each cycle are applied for both AND gates without violating the independence requirement.

The subtraction operation can easily be performed by calculating the two's-complement of the subtrahend. The subtraction is controlled by the SUB input. Therefore, the input signal SUB is xored with every bit of the first share of **Y** (y_0). Incrementing the result by one is done by connecting the carry-in of the adder with SUB which is active on a subtraction. This is done in the shifter of the generation path by appending the carry bit below the least significant bit of the first share and shifting it into the carry generation path. The following equations use the $\ll$ operation to indicate a left shift performed independently on every input share supporting only shifts with 2^n where $n \geq 0$. The calculation of the sum is performed in three steps called preprocessing, processing, and postprocessing.

An addition is started with the initial preprocessing step initializing the registers $\mathbf{G}$, $\mathbf{P}_0$ and $\mathbf{GP}$ according to Equation 8.3.

$$\mathbf{G}_0 = 0 \qquad \mathbf{P}_0 = \mathbf{X} \oplus \mathbf{Y} \qquad \mathbf{GP}_0 = \mathbf{XY} \tag{8.3}$$

The processing step is performed five times in a row (n =1 ... 5). The first and last steps are diverging from the normal processing operation. In the first step, the input register $\mathbf{P}$ is replaced by $\mathbf{P}_0$. In the last processing step, the register update of $\mathbf{P}$ is omitted (see Equations 8.4-8.6).

$$\mathbf{G}_n = \mathbf{G}_{n-1} \oplus \mathbf{GP}_{n-1} \qquad n = 1 \ldots N \tag{8.4}$$

$$\mathbf{P}_n = \mathbf{P}_{n-1}\ (\mathbf{P}_{n-1} \ll 2^{n-1}) \qquad n = 1 \ldots N-1 \tag{8.5}$$

$$\mathbf{GP}_n = \mathbf{P}_{n-1}(\mathbf{G}_n \ll 2^{n-1}) \qquad n = 2 \ldots N \tag{8.6}$$

In the final postprocessing step, the resulting sum is simply computed by a single XOR operation as shown in Equation 8.7.

$$\mathbf{S} = \mathbf{P}_0 \oplus (\mathbf{G}_N \ll 1) \tag{8.7}$$

Resharing of ALU Inputs and Outputs

To reduce the required fresh randomness, the two resharing values $\mathbf{R}^{(1)}$ and $\mathbf{R}^{(2)}$ in Figure 8.1 are generated from the random shares. Furthermore, the merged value of both $\mathbf{R}$ shares is always zero so that an addition of the shares with a sharing of the register file input or output always result in a resharing without changing the underlying value. For first-order protection, the resharing value is generated by duplicating a single random share as shown in Equation 8.8.

$$\mathbf{R}^{(1)} = (z_0^{(1)}, z_0^{(1)}) \qquad \mathbf{R}^{(2)} = (z_0^{(2)}, z_0^{(2)}) \tag{8.8}$$

For other protection orders, the randomness is composed as shown in Equation 8.9-8.10.

$$\mathbf{R}^{(1)} = (z_0^{(1)}, z_0^{(1)} \oplus z_1^{(2)}, z_2^{(1)} \oplus z_1^{(2)}, z_2^{(1)} \oplus z_3^{(2)}, z_4^{(1)} \oplus z_3^{(2)}, \ldots) \tag{8.9}$$

$$\mathbf{R}^{(2)} = (z_0^{(2)}, z_1^{(1)} \oplus z_0^{(2)}, z_1^{(1)} \oplus z_2^{(2)}, z_3^{(1)} \oplus z_2^{(2)}, z_3^{(1)} \oplus z_4^{(2)}, \ldots) \tag{8.10}$$

To guarantee the independence of both resharing values, the first sharing $\mathbf{R}^{(1)}$ uses the shares of $\mathbf{Z}^{(1)}$ with even and shares of $\mathbf{Z}^{(2)}$ with odd indexes, whereas the second sharing $\mathbf{R}^{(2)}$ uses the remaining shares of $\mathbf{Z}^{(1)}$ and $\mathbf{Z}^{(2)}$. This combination of both shares is necessary to prevent adding of two shares which are also used in the *DOM-AND* for the integration step. For example, if the second term of

$\mathbf{R}^{(1)}$ uses the same random z share $(z_0^{(1)} \oplus z_1^{(1)})$, it could be used to eliminate two random values in domain 0. This reduces the number of signals an attacker has to probe to reveal an unshared intermediate.

Other ALU Operations

The remaining operations of the protected ALU (see Figure 8.2) are the shift operations, the logic operations XOR and OR, and the pass-through path. The shift operations are represented by the blocks SLL, SRL and SRA, which perform a logical left or right shift or an arithmetic right shift. The $Shift$ operand uses a separate unshared input for selecting the shift width which is generated outside the module as shown in Figure 8.1. This is necessary to prevent an unwanted merging of the default-used shift operand $\mathbf{Y}$. The shifts are performed independently on every share of $\mathbf{X}$. For the arithmetic right shift, the most significant bit of every share is duplicated. The logical shift operations add zeros to the shares. Therefore, the shares must be refreshed, which is done before writing back the result into the register file or the buffer registers adding fresh randomness (see Figure 8.1).

The XOR operation is done in a straight-forward way by adding the input shares of $\mathbf{X}$ and $\mathbf{Y}$ share-wise. This leads to a zero result using the same input values. Again, the results are reshared using fresh randomness before storing them in the buffer registers *RS1* and *RS2* to guarantee independence of the shares.

The pass-through applies the second input $\mathbf{Y}$ unmodified to the output. To prevent a duplication of the sharing of $\mathbf{Y}$ in different registers, the sharing is again refreshed before writing it to a register.

The OR operation is combined with the AND operation formed by the *DOM-AND* to reduce the logic overhead. This is done by transforming the logical OR into an AND by inverting both inputs and the output. If the OR operation is used, the input *OR* is set which inverts the first share of both input operands as well as the resulting output of the *DOM-AND* by adding to all bits the *OR* signal.

8.2 Hardware Results

The hardware results are gathered for a Xilinx Kintex-7 FPGA with the Xilinx Vivado Design Suite 2014.3. Therefore, the synthesis was done for the unprotected core as well as for the protected V-scale core with protection orders from 1 up to 4. Figure 8.4 shows the evolution of required look-up tables (LUTs) (left) as well as the required registers (right) for increasing protection order. The overall area seems to grow only linearly with the protection order. The design of the DOM-AND gates which are part of the nonlinear modules of the protected ALU increase quadratically which, however, contributes only marginally to the overall size for lower protection orders. Table 8.1 shows the area result in numbers. Additionally, the required randomness is shown which increases quadratically

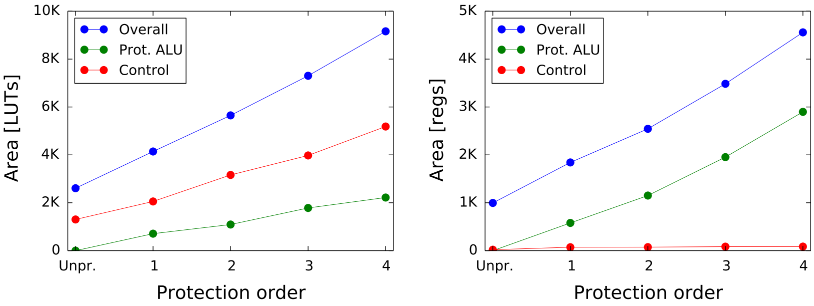

Figure 8.4: Required LUT (left) and registers (right) on an FPGA

Table 8.1: V-scale core implementation results

Prot. Order *d*	FPGA Logic *[LUTs]*	 *[regs]*	Randomness *[Bits]*	Max. Clock *[MHz]*
Unpr.	2,607	996	0	45.6
1	4,143	1,842	64	61.0
2	5,626	2,551	192	59.5
3	7,259	3,484	384	58.3
4	9,244	4,561	640	41.0

with the protection order. In particular the randomness required for the protected ALU is $32 \times d(d+1)$ bits in each cycle. The last column shows the maximum clock frequency which is higher for the protection orders 1 up to 3 than for the unprotected implementation. This results from the additional pipeline stage of the protected implementation which reduces the critical path but increases the delay at the same time.

"Talking isn't doing. It is a kind of good deed to say well; and yet words are not deeds."

— William Shakespeare

9

Conclusions

In this part of the thesis, several masked implementations of cryptographic primitives as well as masked S-box constructions and a masked RISC-V processor implementation have been introduced. These implementations allowed us to compare the DOM based masking schemes from Part I among each other and to compare our implementations with other existing implementations from many different points of view. All of our the stated designs are generic in terms of protection order while related hardware implementations are usually only designed for first and in some cases for second order SCA resistance. This genericity, however, is not bought by a huge overhead in terms of chip area or randomness costs. Our DOM AES and SHA3 implementations, for example, are still the least randomness demanding higher-order protected design to date, while our LOLA AES S-box constructions are the first to lower the latency below two cycles and even allows us to calculate a masked AES S-box in a single clock cycle.

With the Ascon hardware implementations, we have demonstrated that Ascon is not only a very versatile AE with in general low hardware costs but also that its comparably low side-channel protection costs makes it a suitable candidate for the next generation of symmetric-key primitives. Furthermore, both UMA as well as LOLA variants have shown to extend the DOM implementation meaningfully by either reducing the randomness costs or the latency significantly, and thus enabling Ascon for a wide range of security-critical applications with different constraints.

Giving fair comparisons for hardware designs is in general a difficult task given the fact that most of the stated hardware results in the literature are given for different combinations of standard-cell libraries, design tools, and synthesis parameters. The various assumptions on the different masked implementations, like the number of shares for a certain protection order or the relation of the produced random bits among each other, make this comparison even less transparent. A direct comparison of the stated numbers is thus only possible to a very limited extent. To make the comparison with our implementations as fair and transparent as possible, we thus decided to make all of our hardware implementations openly available:

► *Hannes Groß.* Collection of DOM-Protected Hardware Implementations. *https://github.com/hgrosz*

Part III

Verification of Masking

In this part of the thesis, we introduce a method to formally prove the security of masked hardware implementations in the presence of glitches. In contrast to existing formal or non-empirical verification approaches for hardware designs, the introduced approach does not require any additional modeling of the circuit or the leakage source and proves the security of a circuit directly on its netlist. Compared to empirical verification methods based on the statistical analysis of leakage traces, our formal approach allows direct localization of the detected flaws, and gives conclusive security statements that are independent of device- or measurement-specific conditions, or the amount of gathered leakage information.

We base our approach on the probing model of Ishai et al. [Ish+03] and take the effects of glitches into account. We introduce a circuit verification method that performs a conservative estimate of the data an attacker can learn by probing different gates and wires. The verification works directly on the gate-level representation of the circuit. It uses the Fourier expansion (or Walsh expansion) of the functions that are computed and uses the fact that a non-zero Fourier coefficient for a linear combination of variables indicates a correlation between the function and these variables (cf. [Bha+13]). A correlation with a linear combination of variables that contains secrets but no uniformly distributed masking variables corresponds to an information leak. By only keeping track of whether coefficients are zero or not, we circumvent the complexity of a full Fourier representation of all functions computed by all gates of the circuit, at the cost of a loss of precision that may lead to false alarms.

We start this part with an example for an empirical leakage assessment based on statistical t-tests and discuss its limitations. We then introduce our formal verification approach and apply the approach on practical examples.

This part of the thesis is based on the following papers:

- **Chapter 10: Empirical Side-Channel Evaluation**

  ▶ *Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In:* CHES*. vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136*

  Contribution. The author of this thesis is the main author of the paper.

- **Chapter 11: Formal Verification of Masking**

  ▶ *Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches." In:* EUROCRYPT (2)*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353*

  Contribution. The parts of the paper used for this thesis are joint work with equal contributions from the authors.

- **Chapter 12: Practical Formal Verification**

  ► *Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In:* CHES*. vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136*

  ► *Hannes Groß, Rinat Iusupov, and Roderick Bloem.* Generic Low-Latency Masking in Hardware*. CHES 2018 (in press)*

Contribution. The author of this thesis is the main author of the papers. The taint-checking-based verification tool was provided by Rinat Iusupov.

10

Empirical Side-Channel Evaluation

Before introducing our formal verification approach in the next section, we first perform an exemplary empirical leakage evaluation on the example of the UMA S-box of Ascon from Section 6.3 and discuss its limitations.

In order to analyze the SCA resistance of our UMA implementations empirically, we performed a statistical t-test according to Goodwill et al. [Goo+11] on leakage traces of the S-box designs of the UMA variants. We note that t-tests are unsuitable for proving any general statements on the security of a design (for all possible conditions and signal timings) as it would be required for a complete security verification. T-tests only allow statements for the tested devices and under the limitations of the measurement setup. Many works test masked circuits on a Field Programmable Gate Array (FPGA) and perform the t-test on the traces gathered from power measurements. This approach has the drawback that due to the relatively high noise levels the evaluation is usually limited to first and second-order multivariate t-tests.

However, in practice t-tests have proven to be very sensitive and useful to test the side-channel resistance of a design.

Setup. We use the recorded signal traces from the post-synthesis simulations of the netlists, which are noise-free and allow us to evaluate the designs up to the third order. Using post-synthesis leakage traces over traces collected from an FPGA or Application-Specific Integrated Circuit (ASIC) design shows some differences which are in some cases very beneficial but there are also drawbacks. First of all, post-synthesis leakage traces are totally free from environmental noise and variations in the operating conditions like temperature or the supply voltage. As a result, violations of the d^{th}-order security are found with much fewer leakage traces. Another big advantage is that the t-tests can be performed either on a rather coarse level, by taking all signals together into account, or on a very fine-grained level by using individual signals. The latter allows to directly locate the source of the leakage on signal level which makes it very easy to find the flaws in the design.

One disadvantage of this approach is that post-synthesis traces do not use a real existent leakage source. A t-test performed on a specific ASIC chip or FPGA design, however, also only allows to give a statement about this specific device and even does not give a guarantee about its behavior in the future, since signal delays may change under different environmental conditions and over the life cycle of a device. In case of the simulated synthesized netlist, the signal delays are based on unified gate delays which also result in signal glitches that appear from cascading logic gates. Glitches that could result from different wire lengths, and other parasitic effects, however, are not modeled and thus are more likely to show up on FPGA or ASIC-based t-tests.

The intuition of the t-test follows the idea that a Differential Power Analysis (DPA) attacker can only make use of differences in leakage traces. To test that a device shows no exploitable differences, two sets of traces are collected per t-test: (1) a set with randomly picked inputs, (2) a set with fixed inputs and the according t-value is calculated. Then the t-value is calculated according to Equation 10.1 where X denotes the mean of the respective trace set, S^2 is the variance, and N is the size of the set, respectively.

$$t = \frac{X_1 - X_2}{\sqrt{\frac{S_1^2}{N_1} + \frac{S_2^2}{N_2}}} \tag{10.1}$$

The null-hypothesis is that the means of both trace sets are equal, which is accepted if the calculated t-value is below the threshold of ±4.5. If the t-value exceeds this threshold, then the null-hypothesis is rejected with a confidence greater than 99.999% for large enough trace sets. A so-called centered product pre-processing step with trace points inside a six cycle time window is performed for higher-order t-tests. Beyond this time frame, the intermediates are ensured to be unrelated to the inputs. We thus combine multiple tracepoints by first normalizing the means of the trace points and then multiplying the resulting values with other normalized points inside the time window.

Results. Figure 10.1 shows the results of the t-tests for the time offsets which yielded the highest t-values for the UMA S-box implementations of Ascon. From top to bottom, the figures show the results for different protection orders from $d = 0$ to $d = 3$, and from left to right we performed different orders of t-tests starting from first order up to third order. Above $d = 3$ and third-order t-tests, the evaluation of the t-tests becomes too time-intensive for our setup.

On the y-axis of the figures, the t-values are drawn, and the y-axis denotes the used number of traces at a fraction of a million. The horizontal lines (green, inside the figures) indicate the ±4.5 confidence border. The protection border between the figures (the red lines) separates the t-tests for which the protection

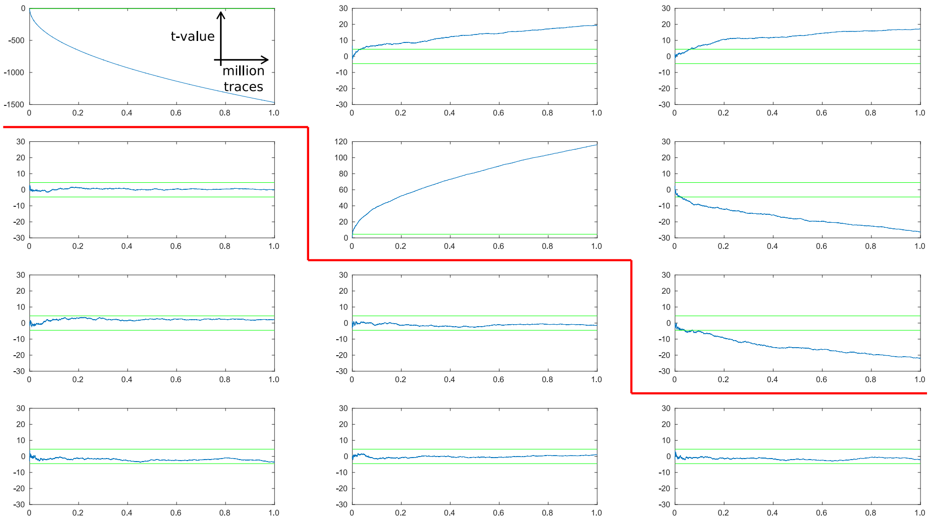

Figure 10.1: T-test evaluation for different protection orders $d = 0 \ldots 3$ (from top to bottom) and for different t-test orders (first to third, from left to right)

order of the design is below the performed t-test (left) from the t-tests for which the test order is above (right).

As intended, the t-values for the masked implementations below the protection border do not show any significant differences even after one million noise-free traces. For the unprotected implementation (top, left figure), for example, the first-order t-test fails with great confidence even after only a couple of traces, and so do the second and third-order t-tests on the right. The first-order t-test below the first-order protected S-box does not show any leakages anymore but the higher-order t-tests fail again as expected. The third-order implementation does not show any leakages anymore for the performed t-tests. We thus conclude that our implementations seem to be secure.

Discussion An empirical leakage evaluation like the t-test performed in this chapter, regardless of whether real leakage traces or simulated leakage traces were used, is never conclusive. This means that is that it is impossible to conclude from the t-test result that with more traces the t-test would still be negative below the stated security level. On the other hand, if the leakage test is positive, finding the cause of the leakage is usually not trivial because the leakage test usually takes the sum of the leakages of all or at least many signals into account. Furthermore, an empirical evaluation does not ensure the security of the implementation itself, but only states the security of the implementation for the tested devices and for the specific environmental and testing conditions which when varied may lead to different signal timings. Another drawback is that t-tests cannot be easily performed during the design phase of a circuit. Is is usually required to map the part that should be tested to a real implementation, e.g. an FPGA, and to run the statistical test on this. However, if an ASIC implementation is targeted, this mapping to an FPGA implementation is quite imprecise because the leakage behavior of logic gates of ASIC designs and the Look-Up Table (LUT) based design of an FPGA are quite different. To conclude this discussion, an empirical leakage assessment, even though it is required and very helpful in practice, can never guarantee the soundness of an implemented countermeasure under all possible environmental conditions and signal timings.

In the next chapter, we introduce a formal verification approach that overcomes these shortcomings.

"Just because it works in practice does not mean it also works in theory."
— Anonymous recipient of cruel reviews

11

Formal Verification of Masking

Automated verification of masked implementations has been intensively researched over the last years and recently many works targeting this topic have been published [Bar+15a; Bay+13; Bel+16; BM16; EW14; Eld+14b; Eld+14a; Mos+12]. Most of the existing work, however, targets software-based masking which does not include the effects of glitches.

Verification of masked software. One of the most groundbreaking works towards the efficient verification of masked software implementations is the work of Barthe et al. [Bar+16]. Instead of proving the security of a whole implementation at once, this work introduces the notion of strong non-interference (SNI). SNI is an extension to the more general non-interference (NI) notion introduced in [Bar+15b]. The SNI notion allows to prove the security of smaller code sequences (called gadgets) in terms of composability with other code parts. Gadgets fulfilling this SNI notion can be freely composed with other gadgets without interfering with their SCA resistance.

Verification of algorithms that fulfill this notion scale much better than other approaches but not all masking algorithms that are secure are also directly composable. As a matter of fact, the most efficient software masking algorithms in terms of randomness of Belaïd et al. [Bel+16; Bel+17], Barthe et al. [Bar+17b], and UMA, for example, do not achieve SNI directly.

In contrast to Barthe et al.'s work on SNI [Bar+16], our approach does not check for composability and is therefore less restrictive to the circuits and masking schemes that can be proven (similar to the NI approach of Barthe et al. [Bar+15b]). Since Barthe et al.'s work is designed to prove masked software implementations, it does not cover glitches. In our work, we introduce the necessary formal groundwork for the verification of masked circuits and in particular the propagation of glitches. Our approach is thereby not bound to our SAT realization but is also compatible with existing tools like easycrypt which is developed by Barthe et al. [Bar+17a].

Most recently, another formal verification approach by Coron [Cor] was introduced that builds on the work of Barthe et al.. Essentially, two approaches are discussed in this work. The first approach is basically the same as the approach in [Bar+15b] but written in Common Lisp language. The second approach is quite different and works by using elementary transformations in order to make the targeted program verifiable using the NI and SNI properties. Coron's work targets again only software based masking and does not take glitches into account.

Eldib et al. [Eld+14b] present an approach to verify masked software implementations. Similar to our approach, the verification problem is encoded into SMT and verified by checking the constraints for individual nodes (operations) inside the program. This approach allows direct localization of the vulnerable code parts. However, their approach targets software and therefore does not cover glitches. It also produces SMT formulas that are exponential in the number of secret variables, whereas the formulas that are produced by our approach are only linear.

Bhasin et al. [Bha+13] also use the Fourier transformation to estimate the side channel attack resistance of circuits. Their approach uses a SAT solver to construct low-weight functions of a certain resistance order. They have not used their approach to evaluate existing implementations of cryptographic functions, and they do not take glitching behavior into account.

Verification of masked hardware. Similar to our approach, Bertoni et al. [BM16] address verification of masked hardware implementations in the presence of glitches. In this work, all possible transients at the input of a circuit are considered and all resulting glitches that could occur at the gates are modeled. However, this approach focuses on first-order masking of purely combinatorial logic and uses a rather simple power model to measure the impact (transitions from 0 to 1 result in the same power consumption as transitions from 1 to 0). We note that focusing on combinatorial logic only, leaves out most of the existing hardware-based masking schemes such as [GM17; Gro+17a; Nik+06; Rep+15]. Bertoni et al.demonstrated their approach on a masked implementation of Keccak based on a masking scheme that is known to be insecure in the presence of glitches.

In contrast to Bertoni et al.'s work, our approach considers combinatorial logic as well as sequential gates (registers), covers also higher-order leakages, and is not restricted to circuits with only one output bit.

In the work of Reparaz [Rep16], a leakage assessment approach is introduced that works by simulating leakages of a targeted hardware implementation in software. At first, a high-level model of the hardware implementation is created, and the verification then works by simulating the model with different inputs and extracting leakage traces. The verification result is gathered by applying statistical significance tests (t-tests) to the simulated leakage traces. Compared to our approach, the leakage detection approach of Reparaz does not perform a formal verification but an empirical leakage assessment. Furthermore, the verification is not directly performed on the targeted hardware implementation but requires to model its (leakage) behavior in software.

11.1 Preliminaries

In the following, we make extensive use of the usual set notation, where $S \triangle T = S \setminus T \cup T \setminus S$ denotes the symmetric difference of S and T and for two sets of sets $\mathcal{S}$ and $\mathcal{T}$, we define $\mathcal{S} \triangle \mathcal{T} = \{S \triangle T \mid S \in \mathcal{S}, T \in \mathcal{T}\}$ to be the pointwise set difference of all elements. We write $\mathbb{B} = \{\mathsf{true}, \mathsf{false}\}$ for the set of Booleans. For a set X of Boolean variables, we identify an assignment $f : X \to \mathbb{B}$ with the set of variables x for which $f(x) = \mathsf{true}$. For a Boolean function $f(X, Y)$ and an assignment $x \subseteq X$, we write $f|_x$ to denote the function $f|_x(y) = f(x, y)$.

Fourier expansion of Boolean functions. There is a close connection between statistical dependence and the Fourier expansion of Boolean functions. First, we formally define statistical independence.

Definition 1 (Statistical independence)**.** *Let X, Y, and Z be sets of Boolean variables and let $f : 2^X \times 2^Y \to 2^Z$. We say that f is statistically independent of X if for all z there is a c such that for all x we have $|\{y \mid f(x, y) = z\}| = c$.*

Lemma 2. *Let $F : \mathbb{B}^X \times \mathbb{B}^Y \to \mathbb{B}^Z$. Function F is statistically independent of X iff for all functions $f : \mathbb{B}^Z \to \mathbb{B}$ we have that $f \circ F$ is statistically independent of X.*

To define Fourier expansions, we follow the exposition of [O'D14] and associate true with -1 and false with 1. We can then represent a Boolean function as a multilinear polynomial over the rationals.

Definition 3 (Fourier expansion)**.** *A Boolean function $f : \{-1, 1\}^n \to \{-1, 1\}$ can be* uniquely *expressed as a multilinear polynomial in the n-tuple of variables $X = (x_1, x_2, \ldots, x_n)$ with $x_i \in \{\pm 1\}$, i.e., the multilinear polynomial of f is a linear combination of monomials, called* Fourier characters*, of the form $\chi_T(X) = \prod_{x_i \in T} x_i$ for every subset $T \subseteq X$. The coefficient of $\chi_T \in \mathbb{Q}$ is called the* Fourier coefficient *$\widehat{f}(T)$ of the subset T. Thus we have the* Fourier representation *of f:*

$$f(X) = \sum_{T \subseteq X} \widehat{f}(T) \chi_T(X) = \sum_{T \subseteq X} \widehat{f}(T) \prod_{x_i \in T} x_i.$$

The Fourier characters $\chi_T : \{-1, 1\}^n \to \{-1, 1\}$ form an orthonormal basis for the vector space of functions in $f : \{-1, 1\}^n \to \{-1, 1\}$. The Fourier coefficients are given by the projection of the function to its basis, i.e., for

$f : \{-1, 1\}^n \to \{-1, 1\}$ and $T \subseteq X = (x_1, x_2, \dots, x_n)$, the coefficient $\widehat{f}(T)$ is given by $\widehat{f}(T) = 1/2^n \cdot \sum_{X \in \{\pm 1\}^n} (f(X) \cdot \chi_T(X))$. In order to prevent confusion between multiplication and addition on rationals and conjuction and XOR on Booleans, we write $\cdot$ and $+$ for the former and $\wedge$ and $\oplus$ for the latter.

As an example, the Fourier expansion of $x \wedge y$ is

$$1/2 + 1/2 \cdot x + 1/2 \cdot y - 1/2 \cdot x \cdot y. \tag{11.1}$$

If $x = \mathsf{false} = 1$ and $y = \mathsf{true} = -1$, for example, the polynomial evaluates to $1/2 + 1/2 - 1/2 + 1/2 = 1 = \mathsf{false}$ as expected for an AND function.

Let us note some simple facts. (1) The Fourier expansion uses the exclusive OR of variables as the basis: $x \oplus y = x \cdot y$. (2) $f^2 = 1$ for the Fourier expansion of any Boolean function f [O'D14]. (3) There are two linear functions of two arguments: $f = x \cdot y$ (XOR) and $f = -(x \cdot y)$ (XNOR). All other functions f are nonlinear and for them, each of $\widehat{f}(\emptyset)$, $\widehat{f}(\{x\})$, $\widehat{f}(\{y\})$, and $\widehat{f}(\{x, y\})$ is nonzero. (We are ignoring the constant and unary functions.) (4) The statistical dependence of the functions can be read off directly from the Fourier expansion: the conjunction has a constant bias, positively correlates with x and y, and negatively with its $x \oplus y$. This last fact can be generalized to the following lemma.

Lemma 4 (Xiao-Massey [XM88])**.** *A Boolean function $f : \{-1, 1\}^n \to \{-1, 1\}$ is statistically independent of a set of variables $X' \subseteq X$ iff $\forall T \subseteq X'$ it holds that if $T \neq \emptyset$ then $\widehat{f}(T) = 0$.*

11.2 Masking and the Probing Model

For this chapter, we change the so far used sharing-based notation to a masking based notation which is more convenient for our formal verification approach. Reconsider that the intention of masking is to impede SCA attacks by making side-channel information independent of the underlying security-sensitive information. This independence is achieved through the randomization of the representation of security-sensitive variables inside the circuit. For this purpose, randomly produced and uniformly distributed masks are added (XOR) to the security-sensitive variables prior to a security-critical computation. The number of used masks depends on the used masking scheme and is a function of the security order.

As a simple example, we consider the security-sensitive 1-bit variable s in Equation 11.2 that is protected by adding a uniformly random mask m_s, resulting in the masked representation s_m.

$$s_m = s \oplus m_s. \tag{11.2}$$

The masked value s_m is again uniformly distributed and statistically independent of s, i.e., it has the same probability to be 0 or 1 regardless of the value of s. Any operation that is performed only on s_m is statistically independent of s and so is thus also the produced side-channel information. Since the mask m_s is randomly produced, operations on the mask are uncritical. However, the combination of side-channel information on s_m and m_s can reveal information on s. The independence achieved through masking is thus only given up to a certain degree (the number of fresh masks used for masking s), and it is important to ensure this degree of independence throughout the entire circuit. The degree of independence is usually referred to as the protection order d.

Masked circuits. For the remainder of this chapter, let us fix an ordered set $X = \{x_0, \dots, x_n\}$ of input variables. We partition the input variables X into three categories:

- $S = \{s_1, \dots s_j\}$ are security-sensitive variables such as key material and intermediate values of cryptographic algorithms that must be protected against an attacker by means of masking.
- $M = \{m_1, \dots m_k\}$ are masks that are used to break the statistical dependency between the secrets S and the information carried on the wires and gates. Masks are assumed to be fresh random variables with uniform distribution and with no statistical dependency to any other variable of the circuit.
- $P = \{p_1, \dots p_l\}$ are all other variables including publicly known constants, control signals, et cetera. Unlike secret variables, these signals do not need to be protected by masks and are unsuitable for protecting secret variables.

We define a circuit $C = (\mathcal{G}, \mathcal{W}, R, f, \mathcal{I})$, where $(\mathcal{G}, \mathcal{W})$ is an acyclic directed graph with vertices $\mathcal{G}$ (gates) and edges $\mathcal{W} \subseteq \mathcal{G} \times \mathcal{G}$ (wires). Gates with indegree 0 are called inputs I, gates with outdegree 0 are called outputs O. Furthermore, $R \subseteq \mathcal{G}$ is a set of registers, f is a function that associates with any gate $g \in \mathcal{G} \setminus I$ with indegree k a function $f(g) : \mathbb{B}^k \to \mathbb{B}$, and $\mathcal{I} : I \to (2^X \to \mathbb{B})$ associates an externally computed Boolean function over X to each input. We require that registers have indegree 1 and that the associated function is the identity. In the following, we assume, wlog, that all gates, except inputs and registers, have indegree 2 and we partition these gates into a set L of linear gates (XOR, XNOR) and a set N of nonlinear gates (AND, NAND, OR, NOR, the two implications and their negations). We also require that for any gate g, any path from some input to g has the same number of registers.

The intuitive meaning of f is the local function computed by a gate. For instance, if g is an AND gate, $f(g)(x, y) = x \wedge y$. We associate with every gate another function $F(g) : 2^X \to \mathbb{B}$, which defines the function computed by the output of the gates in terms of the circuit inputs. The function $F(g)$ is defined by the functions of the predecessor gates and $f(g)$ in the obvious way. Given a sequence of gates $(g_1, \dots, g_d)$, we extend F pointwise to $F(g_1, \dots, g_d) : 2^X \to \mathbb{B}^d$: $F(g_1, \dots, g_d)(x) = (g_1(x), \dots, g_d(x))$. We often identify a gate with its function.

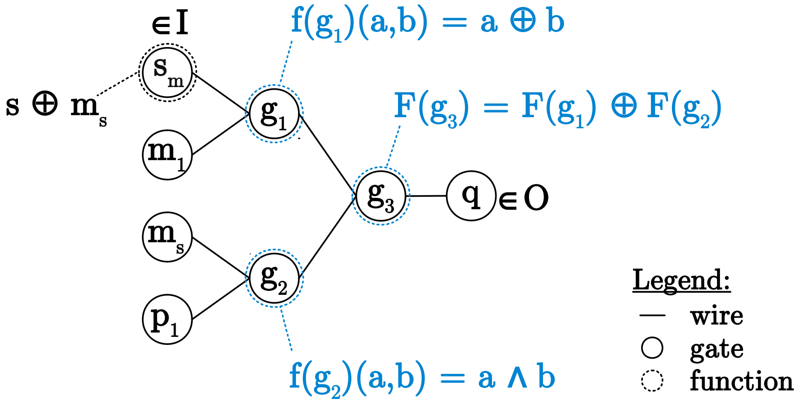

Figure 11.1: Circuit graph of circuit in Figure 11.2

As an example, consider the circuit graph in Figure 11.1 (which corresponds to the circuit depicted in Figure 11.2). We have $f(g_3)(a, b) = a \oplus b$ and $F(g_3) = (s_m \oplus m_1) \oplus (m_s \wedge p_1)$.

For a circuit C, a sequence of gates $G = (g_1, \dots, g_n)$, and a sequence of functions $F = (f_1, \dots, f_n)$ with $f_i \in \mathbb{B}^2 \to \mathbb{B}$, we write $C[G \mapsto F]$ for the circuit C in which gate g_i is replaced by a gate with the Boolean function f_i.

Security of masked circuits. The security of various masking schemes is often analyzed in the so-called probing model that was introduced by Ishai et al. [Ish+03]. It was shown by Faust et al. [Fau+10] and Rivain et al. [RP10] that the probing model is indeed suitable to model side-channel attacks and to describe the resistance of an implementation in relation to the protection order d. As it was shown by Chari et al. [Cha+99], there is an exponential relation between d and the number of leakage traces required to exploit the side-channel information.

In the probing model, an attacker is bound to d probing needles which can be freely placed on arbitrary circuit gates (or wires). Probes are placed permanently on these gates and monitor all signal states and signal transitions that occur at the probed circuit gate from the circuit reset onwards. Thus one probe records the probed signals at all time instances. The probing model quantifies the level of side-channel resistance of a circuit over the minimum number of probing needles an attacker requires to extract any secret information. More specifically, a circuit is secure in the probing model if an attacker cannot combine the information gathered from d probes over all points in time in an arbitrary function F such that F statistically depends on any of the secret variables in S. We model a probe as the ability to read the Boolean function produced by the probed gate or its associated wire. Since we assume that the masking variables are uniformly distributed, and the public variables are known, the circuit leaks information iff

F is statistically dependent on S regardless of the values that the public variables take.

Definition 5 (secure functions). *A function* $f : 2^X \to \mathbb{B}^d$ *is secure if* f *is for any assignment* $p \subseteq P$ *to the public variables,* $f|_p$ *is statistically independent of* S.

Definition 6 (d-probing security [Ish+03]). *A circuit* $C = (\mathcal{G}, \mathcal{W}, f, \mathcal{I})$ *is order* d *probing secure (d-probing secure) iff for any gates* $g_1, \ldots, g_d \in \mathcal{G}$, $F(g_1, \ldots, g_d)$ *is secure.*

Verification example using the Fourier expansion. According to Lemma 4, we can decide whether the values computed by a circuit are secure by computing the Fourier expansion of all its gates and checking whether there is a coefficient that contains only secret variables without a mask (and with or without public variables). Formally, we check that $\emptyset \neq S' \subseteq S \cup P$ such that $\widehat{F}(g)(S') \neq 0$. The first-order security of a circuit can thus be verified using the probing model by calculating the Fourier expansion of the whole circuit. As an example, consider the Fourier expansion of the circuit in Figure 11.2 for which we have:

$$\begin{aligned}
F(g_1) &= s_m \cdot m_1, \\
F(g_2) &= 1/2 + 1/2 \cdot m_s + 1/2 \cdot p_1 - 1/2 \cdot m_s p_1, \text{ and} \\
F(g_3) &= F(g_1) \cdot F(g_2) \\
&= 1/2 \cdot s_m m_1 + 1/2 \cdot m_s s_m m_1 + 1/2 \cdot p_1 s_m m_1 - 1/2 \cdot m_s p_1 s_m m_1.
\end{aligned}$$

Assuming that $s_m = s \oplus m_s$ and using the properties of the Fourier expansion this implies that

$$F(g_3) = 1/2 \cdot s m_s m_1 + 1/2 \cdot s m_1 + 1/2 \cdot s p_1 m_s m_1 - 1/2 \cdot s p_1 m_1. \qquad (11.3)$$

For the example circuit in Figure 11.2, if s is a secret and m_1 is a uniformly distributed random mask, then g_3 in Equation 11.3 computes a function that does not reveal any secret information. This follows from the fact that in $F(g_3)$ there are only (non-zero) Fourier coefficients for terms that contain s and at least one masked value.

Since the exact computation of Fourier coefficients is very expensive and the extension to higher-order probing security nontrivial, in the following we develop a method to estimate the Fourier coefficients of each gate and to check for higher-order security.

11.3 Verification for Stable Signals

In this section, we present a sound verification method for (d-)probing security for the steady-state of a digital circuit. It is assumed that the signals at the circuit input are fixed to a certain value and that all intermediate signals at the gates and the circuit output have reached their final (stable) state. This approach is later on extended in Sections 11.4 and 11.5 to cover transient signals and glitches.

Since the formal verification of the security order of masked circuits has proven to be a non-trivial problem in practice, the intention behind a circuit verifier is to have a method that correctly classifies a wide range of practically relevant and securely masked circuits but rejects all insecure circuits. Any circuit that is not secure according to Definition 6 is rejected. Our verification approach can be subdivided into three parts: (1) the labeling system, (2) the propagation rules, and (3) the actual verification.

11.3.1 Labeling

In order to check the security of a circuit we introduce a labeling over the set of input variables X for the stable signals $\mathcal{S} : \mathcal{G} \rightarrow 2^{2^X}$ that associates a set of sets of variables to every gate. This labeling system is based on the Fourier representation of Boolean functions (see Section 11.1) and intuitively, a label contains at least those sets $X' \subseteq X$ for which $\widehat{f}(X') \neq 0$ (the sets that correlate with the Boolean functions).

The initial labeling is derived from $\mathcal{I}$. For an input g which is fed by function $f_g = \mathcal{I}(g)$, we have $\mathcal{S}(g) = \{X' \subset X \mid \widehat{f_g}(X') \neq 0\}$. In practice, the initial labeling of the circuits is easy to determine as inputs are typically provided with either a single variable m or a masked secret $x \oplus m$. An example for the labeling of an example circuit is shown in Figure 11.2 (blue). Inputs containing security-sensitive variables contain a single set listing all security-sensitive variables and masks that protect these sensitive variables. For the masked signal $s_m = s \oplus m_s$, for example, the initial label is $\mathcal{S}(s_m) = \{\{s, m_s\}\}$. The meaning of the label is that by probing this input the attacker does not learn anything about s. In order to reveal any information on s, also some information on m_s needs to be combined with this wire in, either by the circuit itself (which would be a first-order flaw) or by the attacker by probing an according wire. If the attacker is assumed to be restricted to a single probing needle ($d = 1$), the signal s_m is secure against first-order attacks. Finally, the masked inputs m_s and m_1 in Figure 11.2 contain only the mask variables. Formally, for inputs $g \in I$ with function $\mathcal{I}(g) = f(X)$, we set $\mathcal{S}(g) = \{X' | X' = X\}$.

11.3.2 Propagation rules

To estimate the information that an attacker can learn by probing the output of a gate, we propagate the input labels through the circuit. For the verification, we conservatively estimate which coefficients of the Fourier representation are different from zero and correlate with the variables. We prove at the end of this section that our estimation is sufficient to point out all security-relevant information.

Nonlinear gates. To generate the labels for the outputs of each gate of the circuit, we introduce the *nonlinear gate* rule. The nonlinear gate rule corresponds

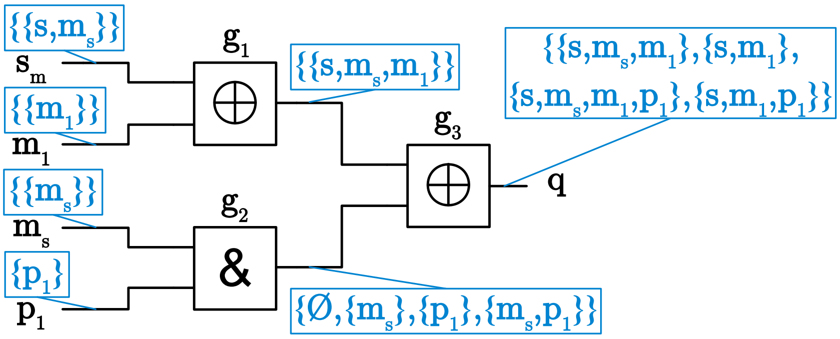

Figure 11.2: Masked circuit example with according labels after the propagation step

to a worst-case estimation of the concrete Fourier spectrum of the signals and trivially catches all flaws. The labeling for the output of the nonlinear gate $g \in N$, with inputs g_a and g_b is:

$$\mathcal{S}(g) = \{\emptyset\} \cup \mathcal{S}(g_a) \cup \mathcal{S}(g_b) \cup \mathcal{S}(g_a) \bigtriangleup \mathcal{S}(g_b).$$

See gate g_2 in Figure 11.2 for a simple example of an AND gate calculating $m_s \wedge p_1$. The resulting labels denote the information that can be learned by probing this gate which could be either m_s or p_1 alone, or together. The labeling reflects the Fourier spectrum of the AND gate (see Equation 11.1). In particular, the labeling shows all terms of the Fourier polynomial whose coefficients are different from zero and are therefore statistically dependent.

Linear gate rule. By following the Definition 3 of the Fourier expansions further we can also model linear gates that have a reduced spectrum compared to nonlinear gates. We model this circumstance by introducing a new rule for labeling a linear gate $g \in L$ with inputs g_a and g_b:

$$\mathcal{S}(g) = \mathcal{S}(g_a) \bigtriangleup \mathcal{S}(g_b).$$

Combined example. To demonstrate how the propagation step works in practice, we applied the propagation rules (summarized in Table 11.1) to an example circuit. The result is shown in Figure 11.2. The AND gate g_2 is a nonlinear gate, and the propagation rules are then iteratively applied to the gates g_1 to g_3. The output labeling of g_1 indicates that the security-critical variable s is here not only protected by m_s but also by the mask m_1. Combining the public signal p_1 with the mask m_s in the nonlinear gate results in a nonuniform output signal which is indicated by the $\{\emptyset\}$ label at the output of g_2. For the calculation of the labels of g_3, the linear rule is used on the output labels of g_1 and g_2 which results in a labeling that indicates that s is still protected by m_s, or m_1, or both, even in the worst-case.

Table 11.1: Propagation rules for the stable set $\mathcal{S}(g)$ connected to the gates g_a and g_b

Gate Type of g	*Stable set rule*
Input $\mathcal{I}(g) = f(X)$	$\mathcal{S}(g) = \{X' \mid X' = X\}$
Nonlinear gate	$\mathcal{S}(g) = \{\emptyset\} \cup \mathcal{S}(g_a) \cup \mathcal{S}(g_b) \cup \mathcal{S}(g_a) \triangle \mathcal{S}(g_b)$
Linear gate	$\mathcal{S}(g) = \mathcal{S}(g_a) \triangle \mathcal{S}(g_b)$
Register	$\mathcal{S}(g) = \mathcal{S}(g_a)$

11.3.3 Verification

For the verification step, in the first-order case, the circuit verifier checks if any of the sublabels created in the propagation step contain one or more secret variables without any masking variables (public variables are ignored since they are unable to mask secret data). If this is the case, the verifier rejects the circuit. In the example circuit in Figure 11.2, all of the labels that contain s also contain m_1 or m_s and therefore the circuit is accepted by the verifier.

Higher-order verification. For the generalization to d-order verification, it is quite tempting to model the attackers abilities by letting the attacker pick multiple labels from any gate and combining them in an arbitrary manner. However, we note that the labeling does not reflect the relation of the probed information among each other and thus does not give a suitable approximation of what can be learned when multiple gates are probed. As a trivial example, consider a circuit that calculates $q = (a \wedge b) \oplus c$ where all inputs are uniformly distributed. The labeling of the output q after the propagation step consists of the labels $\{c\}$, $\{a, c\}$, $\{b, c\}$, and $\{a, b, c\}$ for all of which an attacker probing q would indeed see a correlation. If an attacker restricted to two probes were to probe q with the first probe, the attacker obviously would not learn anything more by probing q a second time. In other words, if we would model a higher-order attacker by the ability to combine multiple labels, the attacker could combine the label $\{c\}$ with any other label of q, e.g. $\{a, b, c\}$, in order to get information on a or b which is of course not the case.

Instead of modeling higher-order verification by the straight-forward combination of labels, we check the nonlinear combination of any tuple of d gates. An attacker can thus pick any number of up to d gates and combines them in an arbitrary function. We then need to check that even the worst case function over the gates could never contain a secret variable without a mask. This causes an obvious combinatorial blowup.

In the next two sections, we extend the verifier to cover glitches which shows that the example circuit is actually insecure.

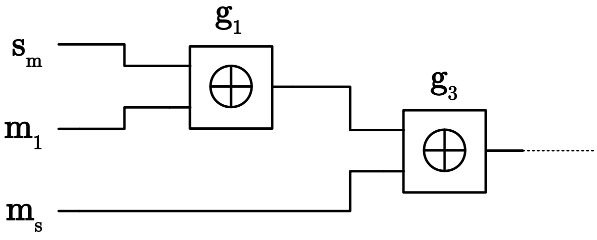

Figure 11.3: Masked circuit example, insecure due to glitches

11.4 Modeling Transient Timing Effects

So far, we have only considered the circuit's stable signals. We now discuss signal timing effects inside one clock cycle i.e. glitches and formalize how we model glitches in the probing model. Subsequently, we discuss how we model information that is collected from multiple clock cycles.

11.4.1 Glitches

As an example of what can go wrong when differences in the signal propagation times are not taken into account [MS06], consider the circuit in Figure 11.3. The depicted circuit is secure in the original probing model as introduced in [Ish+03].

The information on the outputs of the XOR gates is ($s_m = s \oplus m_s$):

$$\begin{aligned} g_1 &= s_m \oplus m_1 = s \oplus m_s \oplus m_1 \text{ and} \\ g_3 &= s_m \oplus m_1 \oplus m_s = s \oplus m_1. \end{aligned}$$

Since the other circuit gates (input terminals are modeled as gates) only carry information on the masked value s_m or the masks m_s and m_1, a single probe on any parts of the circuit does not reveal s and the circuit is thus first-order secure in the original probing model.

However, if we assume that in a subsequent clock cycle (Cycle 2 in Figure 11.4) a different secret s' is processed, the circuit inputs change accordingly from s_m, m_s, and m_1 to s'_m, m'_s, and m'_1, respectively. Figure 11.4 shows an example on how these changes propagate through the circuit. Due to signal timing variance caused by physical circumstances, like different wire lengths or different driving strengths of transistors, so-called glitches arise. As a result of this timing variance, m_1 changes its value later (t_2) than the other inputs (t_1) thus creating a temporary information leak (glitch). An attacker who places one probe on the output of g_3 firsts observes the original value $s \oplus m_1$ (at time t_0) and then $s' \oplus m_1$ (between t_1 and t_2). By combining the information the attacker obtains the information $(s \oplus m_1) \oplus (s' \oplus m_1)$ which is equivalent to $s \oplus s'$. Thus, the attacker learns the relation of two secret bits. This information could not be obtained by combining the stable signals in the two clock cycles. Indeed, the

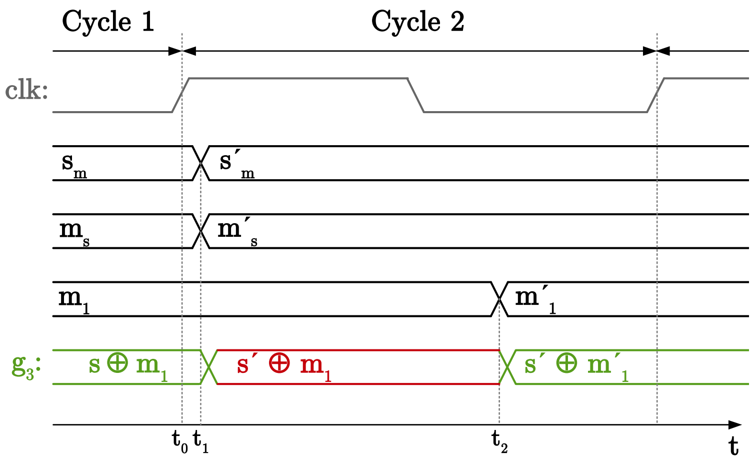

Figure 11.4: Waveform example for the circuit in Figure 11.3, showing security-critical glitch (red)

leakage critically depends on the temporary information provided by the glitch in the circuit. To verify the security of a circuit in the probing model with glitches, all possible signal combinations that could arise because of propagation delays of signals need to be considered.

11.4.2 Formalization of Probing Security with Glitches

To formalize the probing model with glitches in the first-order case, the attacker's abilities are extended as follows: The attacker can first replace any number of gates (except for registers) by a gate that computes an arbitrary Boolean function from the gate's original inputs, and may then place one probe on any wire such that there is no register between any replaced gate and the probe.

For higher-order attacks with $d > 1$, the formalization is a little more cumbersome. Intuitively, the attacker should be able to modify the behavior of arbitrary gates, but this effect should disappear when the signal passes through a register. We model this by copying the combinational parts of the circuit and allowing the attacker to change gates in the copy, whereas the original, unmodified signals are propagated by the unmodified gates. Figure 11.5 illustrates an example for the modeling of the glitches. The copied gates, which the attacker may modify, are drawn in blue. Note in particular that gate g_7 feeds into register g_8, but the copy g'_7 becomes a new primary output.

Formally, given a circuit $C = (\mathcal{G}, \mathcal{W}, R, f, \mathcal{I})$, we do the following.

(1) We define a circuit $C' = (\mathcal{G}', \mathcal{W}', R, f', \mathcal{I})$. We copy all the gates except inputs and registers: $\mathcal{G}' = \mathcal{G} \cup \{g' \mid g \in \mathcal{G} \setminus R \setminus I\}$. We introduce wires from the inputs and registers to the copied gates and introduce wires between the copied gates: $\mathcal{W}' = \mathcal{W} \cup \{(g, h') \mid (g, h) \in \mathcal{W}, g \in I \cup R\} \cup \{(g', h') \mid (g, h) \in \mathcal{W}, g \notin$

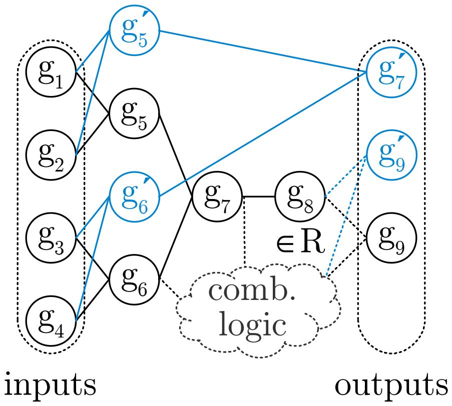

Figure 11.5: Example for modeling of glitches of a circuit C (without blue parts) in C'

$I \cup R, h \notin R\}$. Finally, the functions of the copied gates are the same as those of the originals: $f'(g') = f(g)$ for $g \in \mathcal{G}' \setminus \mathcal{G}$.

(2) The attacker may replace any gate copy g' by a gate that computes an arbitrary Boolean function. We model this by defining a set of circuits, one for any set of gates that the attacker may modify:

$$\mathcal{C}_{\text{glitch}}(C) = \{C'[(g'_1, \dots, g'_n) \mapsto (f_1, \dots, f_n)] \mid (g_1, \dots, g_n) \in \mathcal{G}^n, \forall i. f_i : \mathbb{B}^2 \to \mathbb{B}\}.$$

Definition 7 (d-probing security with glitches)**.** *A circuit C is order d probing secure with glitches iff for any $C_{glitch} = (\mathcal{G}', \mathcal{W}', R, f', \mathcal{I}) \in \mathcal{C}_{glitch}$ and any gates $g_1, \dots, g_d \in \mathcal{G}'$, $F(g_1, \dots, g_d)$ is secure.*

11.4.3 Modeling Information from Multiple Clock Cycles

The verification of higher-order probing security requires to model information that is obtained and combined over different clock cycles. In our verification approach, we consider dependencies between variables rather than concrete instantiation of these variables. The way we model glitches allows an attacker to exploit the worst case dependencies between the variables in between two register stages. We now state assumptions on masked circuits that ensure that the worst-case dependencies are the same in each clock cycle.

Assumptions on masked circuits. Without loss of relevance for masked circuits we make the following assumptions which are inspired by practical masked circuits: (1) We assume that the values on the inputs remain the same throughout a clock cycle, they toggle only once at the beginning of a new clock cycle (registered inputs). (2) The class of the variables that are used in the input functions and the functions themselves do not change over time. For the circuit in Figure 11.3, for example, the input s_m always contains a variable

$s \in S$ and the associated mask $m_s \in M$ even though in each clock cycle the variables may change (e.g. from s to s'). (3) Mask variables are fresh random and uniformly distributed at each clock cycle. (4) The circuits are feedback-free and loop-free, except for the inherent feedback loops of registers. (5) The register depth (number of registers passed, counting from the input of the circuit) for each variable combined in a gate function is the same. No information resulting from different clock cycles is thus combined apart from the effects of delays and glitches which may temporarily combine information from two successive clock cycles. This assumption is motivated by the fact that most of the masked hardware designs, e.g. common S-box designs, are designed in a pipelined way.

From these assumptions it follows that all variables change in each cycle (e.g. from s to s', and so on), however, at varying times and in an arbitrary order. The variable classes and functions remain the same, and as a result from the assumptions 4 and 5 it is ensured that only variables that are fed into the circuit at the same cycle or from the cycle before are combined. It is therefore enough to consider the propagation of dependencies instead of concrete instantiation of variables.

11.5 Extension for Transient Signals

In this section, we use the modeling of the transient timing effects from the previous section to complete our verification approach. We take glitches into account by extending the propagation rules accordingly. The modeling of information from different clock cycles, on the other hand, does not require any changes in the verification approach from Section 11.3.

The nonlinear gate rule in Table 11.1 already inherently covers glitches by propagating the labels of the inputs and all possible combinations of these labels directly to the output. To hinder the propagation of glitches, circuit designers use registers that propagate their input only on a specific clock event, and thus isolate the register input from the output during the evaluation phase. We model the glitching behavior of a circuit by introducing an additional transient set of labels $\mathcal{T}$ per gate. Each gate thus has two associated sets: $\mathcal{S}$ carries the information of the stable state of the circuit as before, and the transient set $\mathcal{T}$ describes the transient information that is only accessible to an attacker in between two registers (or an input and a register, or a register and an output). In between two registers, we also apply the nonlinear gate rule to linear gates to ensure we cover all possible effects of glitches.

Figure 11.6 illustrates the new linear gate rule for the stable (blue) and the transient (red) set of labels. The stable and transient sets of the inputs are equal at the beginning because the inputs are either circuit inputs or outputs of a register. When the signals propagate through the linear XOR gate, the transient set is calculated by applying the linear rule from Table 11.2 and the stable set with the linear rule from Table 11.1. After the signal passes the register, only the stable information remains and the transient set carries thus the same information as the stable set. Table 11.2 summarizes the rules for creating the

Table 11.2: Propagation rules for the transient set $\mathcal{T}(g)$ fed by the gates g_a and g_b

Gate Type of g	*Transient set rule*
Input	$\mathcal{T}(g) = \mathcal{S}(g)$
Nonlinear gate	$\mathcal{T}(g) = \{\emptyset\} \cup \mathcal{T}(g_a) \cup \mathcal{T}(g_b) \cup \mathcal{T}(g_a) \bigtriangleup \mathcal{T}(g_b)$
Linear gate	$\mathcal{T}(g) = \{\emptyset\} \cup \mathcal{T}(g_a) \cup \mathcal{T}(g_b) \cup \mathcal{T}(g_a) \bigtriangleup \mathcal{T}(g_b)$
Register	$\mathcal{T}(g) = \mathcal{S}(g_a)$

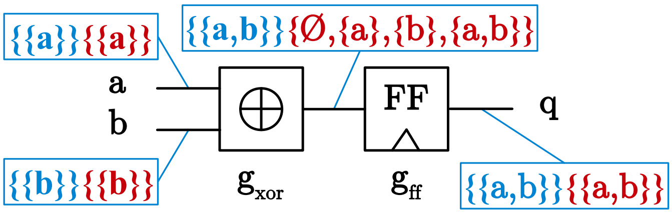

Figure 11.6: XOR gate rules for stable (blue) and transient (red) signal sets

transient-set labels $\mathcal{T}(g)$. Please note that introducing the transient set and the transient gate rules corresponds to the modeling of glitches from Section 11.4 as depicted in Figure 11.5 (blue), where the gates in between two registers are copied and their function can be changed in an arbitrary manner by the attacker. Replacing the transient labels with the stable labels at a register corresponds to connecting the copied gates to the circuit output to hinder the propagation of glitches.

Aside from the introduction of the transient set and the according propagation rules, the verification works as described in Section 11.3. The circuit inputs are initially labeled according to their input variables where both the stable and transient sets hold the same labels. Then, for all possible combinations of up to d gates, the propagation of the labels is performed according to the stable and transient propagation rules. The circuit is order-d probing secure if no combination of gates produces a label that only consists of secrets and public variables without masks.

Example. The transient labels $\mathcal{T}$ of the circuit in Figure 11.2 are shown in Figure 11.7 (the stable sets are omitted since they do not carry any additional information). Due to the transient set propagation rules, the functionality of the gates g_1 and g_3, which are linear gates in the underlying circuit in Figure 11.2, is replaced by the functionality of nonlinear gates. As can be observed at the output of the circuit, the verification under the consideration of glitches leads to a rejection of the circuit because the s variable (black labels) is in the output labeling without being masked by either m_s or m_1.

To make it clear that the circuit is indeed insecure, we assume that $p_1 = \texttt{true}$ and that s_m and m_s change their values to s'_m and m'_s, resp., but the value of m_1

and p_1 temporarily remains unchanged. Then, g_1 transitions from $s \oplus m_s \oplus m_1$ to $s' \oplus m_s' \oplus m_1$ and as a result g_3 transitions from $s \oplus m_1$ to $s' \oplus m_1$, thus leaking information about the relation of s and s' (cf. Figure 11.4). The flaw can be repaired easily by adding a register after g_1 which ensures that s_m is always remasked before m_s is combined with s_m in g_3, and the same labels as in Figure 11.2 for g_1 would thus be propagated.

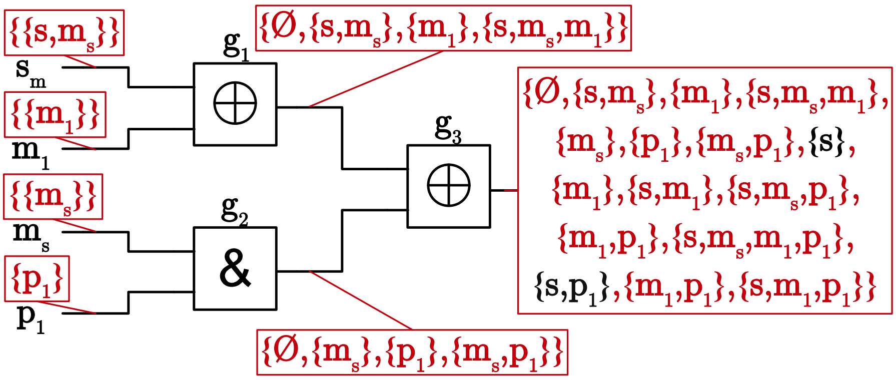

Figure 11.7: Masked circuit example from Figure 11.2 reevaluated with the transient rules (red) which leads to a flaw due to glitches (black labels)

12

Practical Formal Verification

For the analysis of the side-channel resistance of our UMA and LOLA Ascon S-box designs, we used the formal verification approach described in the previous chapter. For the sake of completeness, we denote that in the original paper of our formal verification approach [Blo+18] we also state more verification results, including but not limited to the DOM multipliers, the DOM AES S-box, and the DOM Keccak S-box. We do not include these results in this chapter because they were mainly gathered by Rinat Iusupov. The respective tool which was also implemented by Rinat Iusupov is publicly accessible online [Ius]. For the LOLA circuits we extended the verification approach towards a faster evaluation for these specific LOLA circuits (cf. taint checking below).

12.1 Formal Verification of UMA Circuits

The verification results for the UMA S-box design of ascon are shown in Table 12.1. We used the optimization introduced in [Blo+18] for which each shared input is checked individually. The time stated in the table is the average verification time over each of the five secret inputs. The first-order protected UMA S-box requires about one second per secret. The verification time increases significantly with the verification order. For the second order, the verification takes about 1.5 minutes, and for the third order it is already about 20 hours. Again, all verification results indicate a secure design of the S-box which confirms the results of the t-test in Section 10.

Table 12.1: Formal verification results of the UMA S-box

Design	Verification order	Time	Result
1^{st}-order UMA S-box	1	$\leq 1\,\mathrm{s}$	✓
2^{nd}-order UMA S-box	2	$\leq 1.5\,\mathrm{m}$	✓
3^{rd}-order UMA S-box	3	$\leq 20\,\mathrm{h}$	✓

12.2 Formal Verification of LOLA Circuits

For our further side-channel experiments, we verified the LOLA S-box designs of ASCON up to order three. The results are shown in Table 12.2 (column FV).

It shows that the first-order S-box design is verified in less than two seconds (parallel verification of the five secrets) on our Intel Xeon E5-2699v4 CPU with a clock frequency of 3.6 GHz and 512 GB of RAM running in a 64-bit Debian 9 operating system. For order two, the verification increases to about 18 seconds, and for order three the verification takes about 21 minutes. All verification results indicate securely masked circuits for the given protection order.

For the verification of the AES S-box, on the other hand, the circuit size exceeds the number of gates over the most complex circuit tested in the paper of Bloem et al. [Blo+18] (a DOM-protected AES S-box verified in 5 to 10 hours) by almost a factor of ten. Therefore, we could not finish the verification within one day and decided to use a verification approach specifically designed for our approach which we refer to as taint checking in the following.

12.3 Taint checking of the LOLA AES S-box

The basic idea for the taint checking verification approach follows from the design principle of our LOLA masking approach. Any $d+1$ masked circuit is trivially secure in the probing model if for any gate or wire of the circuit there is no path that connects any two shares of one variable. We could thus split the circuit into $d+1$ distinct sub-circuits that are never fed by two or more shares of one shared input variable. This approach only works, of course, if we do not use a share compression as it is the case for the ASCON S-box design and the zero latency AES S-box variant. Other variants of our designs that use the CMS share compression cannot be verified using this approach because the compression clearly creates paths that combine two or more shares (which are then of course first remasked to ensure independence). However, our main goal is to show the security of our LOLA masking approach and to demonstrate that even very complex designs like the AES S-box can be securely implemented this way. The other variants of the AES S-box suggested in Section 3.5 are introduced to analyze possible trade-offs and implementation costs of our approach.

We instantiated the taint checking approach by using the SAT-based verification tool as basis. We label all shared circuit inputs accordingly to the sharing and then simply propagate the input labels through the entire circuit so that every gate and wire that is somehow connected with the input share is tainted by assigning the label of the connected inputs. If at any point in the circuit two shares from the same variable are part of the labeling of one wire, our tool denotes a flaw and returns the causing gate and inputs. This tool has proven to be extremely helpful also during the design of the LOLA AES and Ascon circuits.

We also performed the taint checking verification approach for the Ascon circuits, for which the verification now takes less than a second. Furthermore, we managed to check the first-order zero latency AES S-box variant in a bit more than ten minutes. We, however, note that this approach works only for this specific kind of LOLA circuits without compression for which the security can be easily verified by ensuring a separation of shares throughout the entire circuit.

Table 12.2: Side-channel resistance verification results for the LOLA Ascon and the first-order zero latency AES S-box designs

S-box Design	Gates		Order	FV		Taint Checking	
	Lin	Non-lin		Time	Result	Time	Result
1st-order Ascon	34	22	1	$\leq 2\,\mathrm{s}$	✓	$\leq 1\,\mathrm{s}$	✓
2nd-order Ascon	58	48	2	$\leq 18\,\mathrm{s}$	✓	$\leq 1\,\mathrm{s}$	✓
3rd-order Ascon	88	84	3	$\leq 21\,\mathrm{m}$	✓	$\leq 1\,\mathrm{s}$	✓
Zero Latency AES	17,199	5,544	1	≥ 1 day	?	$\leq 11\,\mathrm{m}$	✓

"All the proof of a pudding,
is in the eating."
— William Camden

13
Conclusions

Having secure masking schemes whose assumptions and rules hold in the probing model is a basic requirement for having SCA-resistant circuits. Nevertheless, the usage of a masking scheme in practice remains a cumbersome and error-prone task which makes the verification of the SCA resistance of a masked implementation inevitable.

In this part, we have compared a typical empirical method that is widely used in practice, namely a t-test based leakage assessment on leakage traces of the UMA construction of the ASCON S-box (Chapter 10), against a formal verification method which we introduced in Chapter 11. While empirical methods like statistical leakage assessment or practical attacks can only increase the trust in the correctness of a masked implementation, a formal verification can give conclusive security statements. The security is thus ensured for all possible signal timings, environmental conditions, and beyond the number of tested leakage traces.

The advantage of empirical methods, on the other hand, is that they allow to consider a bigger part of a circuit for the verification like a whole chip during the processing of security-critical data. The formal verification method we introduced, despite the fact that it is the first method that is efficient enough to prove a whole masked DOM AES S-box, allows at least not yet for a verification of a whole AES circuit, for example.

However, in most of the existing symmetric ciphers, the S-box construction is the only nonlinear part which is the most likely to fail and the hardest to verify because this is the part of a circuit where share domains need to be mixed. For the remaining linear parts of a circuit, a much simpler approach can be used to ensure that no domain crossings are performed like the taint checking approach we introduced in Chapter 12.

A clear benefit of our formal verification approach over empirical methods is that a flaw can easily be localized and that our tool can handle netlists at different design stages (from a first RTL description up to a back-annotated netlist before the final tape out). This makes the approach helpful during the whole design process of a masked circuit and allows to try out new optimizations, new masking schemes, and to directly observe their impact.

"Prediction is very difficult,
especially about the future."
— Niels Bohr

14

Summary and Outlook

The increasing number of interconnected devices demand security not only on a cryptographic level but also on a physical level. Without appropriate countermeasures against physical attacks like Side-Channel Analysis (SCA) these devices are defenseless against attackers that have physical access. Such an attacker could extract secret information by just passively observing the physical side-channel information (like power consumption, electromagnetic emanation) that is unintentionally created during security-critical computations.

In this thesis, we discussed generic methods to protect hardware implementations against SCA by means of Boolean masking. We introduced a Domain-Oriented Masking (DOM) perspective in the first part of the thesis and derived different masking schemes that allow to trade implementation costs like chip area and randomness requirements against performance figures, and vice versa. All presented schemes form the basis for the design of hardware implementations that scale generically with the required level of resistance against SCA.

We discussed the implementation costs of our DOM-based schemes and several trade-offs on generically masked hardware implementations in the second part of this thesis. Our practical comparison includes implementations of the Advanced Encryption Standard (AES), the Secure Hash Algorithm 3 (SHA3), the Authenticated Encryption (AE) scheme Ascon, and a 32-bit RISC-V processor. Regardless of their generic appearance, our masked hardware implementations result in very randomness-efficient and low area demanding circuits, or in masked circuits that allow to calculate the outputs within only a few clock cycles, respectively.

In the third part of this thesis, we introduced a formal verification approach that verifies the soundness of the implemented masking scheme directly on the netlist of a hardware implementation. We compared this formal approach to an empirical leakage assessment based on statistical t-tests which is the predominant method for leakage evaluation in the recent literature. Besides the much stronger security argument that is provided by formal verification, our approach has proven to be very helpful during the whole design process of a masked circuit and for the creation of new masking schemes.

Open Research Questions

Masking as a countermeasure against SCA has a long history that reaches back almost as long as the Differential Power Analysis (DPA) paper of Kocher et al. [Koc+99]. Even though this research area seems relatively old for the fast growing and fast developing computer sciences field, substantial progress has been made over the past years and the progress still continues. In the following, we want to share our perspective on how this research field might progress over the next fex years and what questions remain unanswered after this thesis.

Randomness Requirements

The question of how much randomness is necessary to protect an implementation against a certain attack order remains remains an open issue. There are works that try to prove randomness bounds like [Bel+16]. However, it remains uncertain whether or not these bounds are tight and under what assumptions they hold. These proven bounds also just consider a completely isolated masked multiplication. So, even if the bounds were tight it is not clear how they map to practical implementations of larger circuits where randomness could probably be reused. For example, in our Low-Latency Masking (LOLA) scheme we have shown that theoretically no online randomness is needed at all if an exponential share blow-up is accepted. So, an even more intriguing question in this regard is how we can find the sweet spot between the number of sharing duplications and the required online randomness under the given constraints.

Better Compression Algorithms for LOLA

In regard of low-latency masking, we used the CMS resharing algorithm for the intermediate and final share compression in our hardware implementations. Given the extended domain separation principle, it should be possible to design a more efficient compression algorithm much like the original DOM compression that requires less randomness than CMS for LOLA circuits.

Smaller Technology Sizes

Complementary Metal-Oxide-Semiconductor (CMOS) technology continues to evolve and to produce smaller structure sizes which soon will reach 5 nm processes for semiconductor manufacturing. With the decrease of the structure sizes, the CMOS transistors produce more and more static leakage (current flows even though the transistors are turned off) and wire distances get smaller which provokes crosstalk. Both effects shake on the fundamental assumptions of masking which requires independent leakage of shared information or at least only additive Gaussian leakages. The extent to which cross talk and static leakage reduce the security level in practical implementations is largely unexplored and might lead to higher requirements for the SCA protection order.

Formal Verification

Practical formal verification of masked hardware implementations is quite a young research field compared to masking itself. The formal verification approach we presented in this thesis is the first to prove practical hardware implementations on the basis of their netlist and not on e.g. their mathematical equations, under the assumption of glitches, and for higher verification orders. Nevertheless, our approach is only a first step towards practical formal verification of masked hardware implementations, and we are optimistic that the outcome of our work will encourage research on how to improve and extend this approach.

Bibliography

[Ala+09] Monjur Alam, Santosh Ghosh, M. J. Mohan, Debdeep Mukhopadhyay, Dipanwita Roy Chowdhury, and Indranil Sengupta. “Effect of glitches against masked AES S-box implementation and countermeasure.” In: *IET Information Security* 3.1 (2009), pp. 34–44.

[Alb+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. “Ciphers for MPC and FHE.” In: *EUROCRYPT 2015.* 2015. DOI: `10.1007/978-3-662-46800-5_17`. URL: `http://dx.doi.org/10.1007/978-3-662-46800-5_17`.

[Aum+14] Jean-Philippe Aumasson, Philipp Jovanovic, and Samuel Neves. *NORX.* Submission to the CAESAR competition: `http://competitions.cr.yp.to/round1/norxv1.pdf`. 2014.

[Bar+15a] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, and Benjamin Gregoire. “Compositional Verification of Higher-Order Masking: Application to a Verifying Masking Compiler.” In: *IACR Cryptology ePrint Archive* 2015 (2015), p. 506. URL: `http://eprint.iacr.org/2015/506`.

[Bar+15b] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. “Verified Proofs of Higher-Order Masking.” In: *EUROCRYPT 2015, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I.* Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. LNCS. Springer, 2015, pp. 457–485. DOI: `10.1007/978-3-662-46800-5_18`. URL: `https://doi.org/10.1007/978-3-662-46800-5_18`.

[Bar+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. “Strong Non-Interference and Type-Directed Higher-Order Masking.” In: *Proceedings of the 2016 ACM SIGSAC CCS, Vienna, Austria, October 24-28, 2016.* 2016, pp. 116–129. DOI: `10.1145/2976749.2978427`. URL: `http://doi.acm.org/10.1145/2976749.2978427`.

[Bar+17a] Gilles Barthe, François Dupressoir, Benjamin Grégoire, Alley Stoughton, and Pierre-Yves Strub. *EasyCrypt: Computer-Aided Cryptographic Proofs.* `https://github.com/EasyCrypt/easycrypt`. 2017.

[Bar+17b] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model." In: *EUROCRYPT (1)*. Vol. 10210. Lecture Notes in Computer Science. 2017, pp. 535–566.

[Bay+13] Ali Galip Bayrak, Francesco Regazzoni, David Novo, and Paolo Ienne. "Sleuth: Automated Verification of Software Power Analysis Countermeasures." In: *CHES 2013, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*. 2013, pp. 293–310. DOI: `10.1007/978-3-642-40349-1_17`. URL: `http://dx.doi.org/10.1007/978-3-642-40349-1_17`.

[Bel+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Randomness Complexity of Private Circuits for Multiplication." In: *EUROCRYPT 2016*. 2016. DOI: `10.1007/978-3-662-49896-5_22`. URL: `http://dx.doi.org/10.1007/978-3-662-49896-5_22`.

[Bel+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields." In: *CRYPTO (3)*. Vol. 10403. LNCS. Springer, 2017, pp. 397–426.

[Ber+11] Guido Bertoni, Joan Daemen, Michael Peeters, and Gilles Van Assche. *Keccak Specifications*. Submission to NIST (Round 3). 2011. URL: `http://keccak.noekeon.org`.

[Ber+12] Guido Bertoni, Joan Daemen, Michael Peeters, Gilles Van Assche, and Ronny Van Keer. "Keccak implementation overview." In: *URL: http://keccak.neokeon.org/Keccak-implementation-3.2.pdf* (2012).

[Bha+13] Shivam Bhasin, Claude Carlet, and Sylvain Guilley. "Theory of masking with codewords in hardware: low-weight dth-order correlation-immune Boolean functions." In: *IACR Cryptology ePrint Archive* 2013 (2013), p. 303. URL: `http://eprint.iacr.org/2013/303`.

[Bil+14a] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "A More Efficient AES Threshold Implementation." In: *Progress in Cryptology - AFRICACRYPT 2014 - 7th International Conference on Cryptology in Africa, Marrakesh, Morocco, May 28-30, 2014. Proceedings*. Ed. by David Pointcheval and Damien Vergnaud. Vol. 8469. Lecture Notes in Computer Science. Springer, 2014, pp. 267–284.

[Bil+14b] Begül Bilgin, Joan Daemen, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Gilles Van Assche. "Efficient and First-Order DPA Resistant Implementations of Keccak." English. In: *Smart Card Research and Advanced Applications*. Ed. by Aurélien Francillon and Pankaj Rohatgi. Lecture Notes in Computer Science. Springer International Publishing, 2014, pp. 187–199. ISBN: 978-3-319-08301-8.

DOI: `10.1007/978-3-319-08302-5_13`. URL: `http://dx.doi.org/10.1007/978-3-319-08302-5_13`.

[Bil+14c] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Higher-Order Threshold Implementations." English. In: *Advances in Cryptology – ASIACRYPT 2014*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, pp. 326–343. ISBN: 978-3-662-45607-1. DOI: `10.1007/978-3-662-45608-8_18`. URL: `http://dx.doi.org/10.1007/978-3-662-45608-8_18`.

[Bil+15a] Begül Bilgin, Svetla Nikova, Ventzislav Nikov, Vincent Rijmen, Natalia Tokareva, and Valeriya Vitkup. "Threshold Implementations of Small S-boxes." In: *Cryptography and Communications* 7.1 (2015). DOI: `10.1007/s12095-014-0104-7`. URL: `http://dx.doi.org/10.1007/s12095-014-0104-7`.

[Bil+15b] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "Trade-Offs for Threshold Implementations Illustrated on AES." In: *IEEE Trans. on CAD of Integrated Circuits and Systems* 34.7 (2015), pp. 1188–1200.

[Blo+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches." In: *EUROCRYPT (2)*. Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353.

[BM16] Guido Bertoni and Marco Martinoli. "A Methodology for the Characterisation of Leakages in Combinatorial Logic." In: *SPACE 2016, Hyderabad, India, December 14-18, 2016, Proceedings.* 2016, pp. 363–382. DOI: `10.1007/978-3-319-49445-6_21`. URL: `https://doi.org/10.1007/978-3-319-49445-6_21`.

[Bog+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. "PRESENT: An Ultra-Lightweight Block Cipher." In: *CHES 2007*. 2007. DOI: `10.1007/978-3-540-74735-2_31`. URL: `http://dx.doi.org/10.1007/978-3-540-74735-2_31`.

[Bog+11] Andrey Bogdanov, Dmitry Khovratovich, and Christian Rechberger. "Biclique Cryptanalysis of the Full AES." In: *Advances in Cryptology – ASIACRYPT 2011: 17th International Conference on the Theory and Application of Cryptology and Information Security, Seoul, South Korea, December 4-8, 2011. Proceedings.* Ed. by Dong Hoon Lee and Xiaoyun Wang. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 344–371. ISBN: 978-3-642-25385-0. DOI: `10.1007/978-3-642-25385-0_19`. URL: `https://doi.org/10.1007/978-3-642-25385-0_19`.

[Bog+13] Andrey Bogdanov, Florian Mendel, Francesco Regazzoni, Vincent Rijmen, and Elmar Tischhauser. "ALE: AES-Based Lightweight Authenticated Encryption." In: *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Springer, 2013, pp. 447–466. ISBN: 978-3-662-43932-6. DOI: `10.1007/978-3-662-43933-3_23`. URL: `http://dx.doi.org/10.1007/978-3-662-43933-3_23`.

[BP12] Joan Boyar and René Peralta. "A Small Depth-16 Circuit for the AES S-Box." In: *SEC*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, 2012, pp. 287–298.

[Can05] David Canright. "A Very Compact S-Box for AES." In: *CHES*. Vol. 3659. Lecture Notes in Computer Science. Springer, 2005, pp. 441–455.

[Cha+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks." In: *CRYPTO*. Vol. 1666. Lecture Notes in Computer Science. Springer, 1999, pp. 398–412.

[Cnu+15] 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. DOI: `10.1007/978-3-319-31271-2_16`. URL: `http://dx.doi.org/10.1007/978-3-319-31271-2_16`.

[Cnu+16] 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. DOI: `10.1007/978-3-662-53140-2_10`. URL: `http://dx.doi.org/10.1007/978-3-662-53140-2_10`.

[Cor] Jean-Sebastien Coron. *Formal Verification of Side-channel Countermeasures via Elementary Circuit Transformations*. Cryptology ePrint Archive, Report 2017/879.

[Dae+00] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. "Nessie proposal: NOEKEON." In: *First Open NESSIE Workshop*. 2000, pp. 213–230.

[Dae12] Joan Daemen. *Permutation-based Encryption, Authentication and Authenticated Encryption*. DIAC – Directions in Authenticated Ciphers. 2012.

[Dae16] Joan Daemen. "On Non-uniformity in Threshold Sharings." In: *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security*. TIS '16. Vienna, Austria: ACM, 2016, pp. 41–41. ISBN: 978-1-4503-4575-0. DOI: `10.1145/2996366.2996374`. URL: `http://doi.acm.org/10.1145/2996366.2996374`.

[Dae17] Joan Daemen. “Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing.” In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 137–153.

[Dob+16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round3/asconv12.pdf`. 2016. URL: `http://ascon.iaik.tugraz.at`.

[Eld+14a] Hassan Eldib, Chao Wang, Mostafa M. I. Taha, and Patrick Schaumont. “QMS: Evaluating the Side-Channel Resistance of Masked Software from Source Code.” In: *DAC '14, San Francisco, CA, USA, June 1-5, 2014*. 2014, 209:1–209:6. DOI: `10.1145/2593069.2593193`. URL: `http://doi.acm.org/10.1145/2593069.2593193`.

[Eld+14b] Hassan Eldib, Chao Wang, and Patrick Schaumont. “SMT-Based Verification of Software Countermeasures against Side-Channel Attacks.” In: *TACAS 2014, 2014, Grenoble, France, April 5-13, 2014. Proceedings*. 2014, pp. 62–77. DOI: `10.1007/978-3-642-54862-8_5`. URL: `http://dx.doi.org/10.1007/978-3-642-54862-8_5`.

[EW14] Hassan Eldib and Chao Wang. “Synthesis of Masking Countermeasures against Side Channel Attacks.” In: *CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*. 2014, pp. 114–130. DOI: `10.1007/978-3-319-08867-9_8`. URL: `http://dx.doi.org/10.1007/978-3-319-08867-9_8`.

[Fau+10] Sebastian Faust, Tal Rabin, Leonid Reyzin, Eran Tromer, and Vinod Vaikuntanathan. “Protecting Circuits from Leakage: the Computationally-Bounded and Noisy Cases.” English. In: *EUROCRYPT 2010*. Vol. 6110. LNCS. 2010. ISBN: 978-3-642-13189-9. DOI: `10.1007/978-3-642-13190-5_7`. URL: `http://dx.doi.org/10.1007/978-3-642-13190-5_7`.

[Fel+05] Martin Feldhofer, Johannes Wolkerstorfer, and Vincent Rijmen. “AES Implementation on a Grain of Sand.” In: *IEEE Proceedings - Information Security* 152.1 (2005), pp. 13–20. ISSN: 1747-0722. DOI: `10.1049/ip-ifs:20055006`.

[GC17] Ashrujit Ghoshal and Thomas De Cnudde. “Several Masked Implementations of the Boyar-Peralta AES S-Box.” In: *INDOCRYPT*. Vol. 10698. Lecture Notes in Computer Science. Springer, 2017, pp. 384–402.

[Gho+07] Santosh Ghosh, Monjur Alam, Kundan Kumar, Debdeep Mukhopadhyay, and Dipanwita Roy Chowdhury. “Preventing the Side-Channel Leakage of Masked AES S-Box.” In: *Advanced Computing and Communications, 2007. ADCOM 2007. International Conference on*. 2007. DOI: `10.1109/ADCOM.2007.63`.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In: *CHES*. Vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136.

[GM18] Hannes Gross and Stefan Mangard. "A Unified Masking Approach." In: *Journal of Cryptographic Engineering* (2018). ISSN: 2190-8516. DOI: `10.1007/s13389-018-0184-y`.

[Goo+11] 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.

[GP12] Hannes Groß and Thomas Plos. "On Using Instruction-Set Extensions for Minimizing the Hardware-Implementation Costs of Symmetric-Key Algorithms on a Low-Resource Microcontroller." In: *RFIDSec*. Vol. 7739. Lecture Notes in Computer Science. Springer, 2012, pp. 149–164.

[Gro] Hannes Groß. *Collection of DOM-Protected Hardware Implementations*. `https://github.com/hgrosz`.

[Gro+] Hannes Groß, Rinat Iusupov, and Roderick Bloem. *Generic Low-Latency Masking in Hardware*. CHES 2018 (in press).

[Gro+14a] Hannes Groß, Erich Wenger, Honorio Martín, and Michael Hutter. "PIONEER - a Prototype for the Internet of Things Based on an Extendable EPC Gen2 RFID Tag." In: *RFIDSec*. Vol. 8651. Lecture Notes in Computer Science. Springer, 2014, pp. 54–73.

[Gro+14b] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, Kerem Varici, François Durvaux, Lubos Gaspar, and Stéphanie Kerckhof. *CAESAR candidate SCREAM Side-Channel Resistant Authenticated Encryption with Masking*. DIAC 2014: Directions in Authenticated Ciphers, Santa Barbara, USA [Accessed: 2014 09 30]. Aug. 2014. URL: `http://2014.diac.cr.yp.to/slides/leurent-scream.pdf`.

[Gro+15a] Hannes Groß, Marko Hölbl, Daniel Slamanig, and Raphael Spreitzer. "Privacy-Aware Authentication in the Internet of Things." In: *CANS*. Vol. 9476. Lecture Notes in Computer Science. Springer, 2015, pp. 32–39.

[Gro+15b] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. "Suit up! — Made-to-Measure Hardware Implementations of ASCON." In: *DSD*. IEEE Computer Society, 2015, pp. 645–652.

[Gro+16a] Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs." In: *CARDIS*. Vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104.

[Gro+16b] Hannes Groß, Stefan Mangard, and Thomas Korak. “Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order.” In: *IACR Cryptology ePrint Archive* (2016).

[Gro+16c] Hannes Groß, Stefan Mangard, and Thomas Korak. “Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order.” In: *TIS@CCS*. ACM, 2016, p. 3.

[Gro+17a] Hannes Groß, Stefan Mangard, and Thomas Korak. “An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order.” In: *CT-RSA*. Vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112.

[Gro+17b] Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. “Ascon hardware implementations and side-channel evaluation.” In: *Microprocessors and Microsystems - Embedded Hardware Design* 52 (2017), pp. 470–479.

[Gro+17c] Hannes Groß, David Schaffenrath, and Stefan Mangard. “Higher-Order Side-Channel Protected Implementations of KECCAK.” In: *DSD*. IEEE Computer Society, 2017, pp. 205–212.

[Gro15] Hannes Groß. “Sharing is Caring - On the Protection of Arithmetic Logic Units against Passive Physical Attacks.” In: *RFIDSec*. Vol. 9440. Lecture Notes in Computer Science. Springer, 2015, pp. 68–84.

[Gro16] Hannes Groß. *DOM-Protected Hardware Implementation of AES*. `https://github.com/hgrosz/aes-dom`. 2016.

[Gue09] Shay Gueron. “Intel’s New AES Instructions for Enhanced Performance and Security.” In: *FSE*. Vol. 5665. Lecture Notes in Computer Science. Springer, 2009, pp. 51–66.

[Ish+03] Yuval Ishai, Amit Sahai, and David Wagner. “Private Circuits: Securing Hardware against Probing Attacks.” English. In: *CRYPTO 2003*. Vol. 2729. LNCS. 2003. ISBN: 978-3-540-40674-7. DOI: `10.1007/978-3-540-45146-4_27`. URL: `http://dx.doi.org/10.1007/978-3-540-45146-4_27`.

[Ius] Rinat Iusupov. *REBECCA - Masking verification tool*. `https://github.com/riusupov/rebecca`.

[Iwa+14] Tetsu Iwata, Kazuhiko Minematsu, Jian Guo, Sumio Morioka, and Eita Kobayashi. *SILC: SImple Lightweight CFB*. DIAC 2014: Directions in Authenticated Ciphers, Santa Barbara, USA [Accessed: 2014 09 30]. Aug. 2014. URL: `http://2014.diac.cr.yp.to/slides/iwata-silc.pdf`.

[Koc+99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. "Differential Power Analysis." In: *Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology*. CRYPTO '99. London, UK, UK: Springer-Verlag, 1999, pp. 388–397. ISBN: 3-540-66347-9. URL: `http://dl.acm.org/citation.cfm?id=646764.703989`.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *CRYPTO*. Vol. 1109. Lecture Notes in Computer Science. Springer, 1996, pp. 104–113.

[Kum+07] Kundan Kumar, Debdeep Mukhopadhyay, and Dipanwita Roy-Chowdhury. "Design of a Differential Power Analysis Resistant Masked AES S-Box." English. In: *INDOCRYPT 2007*. Vol. 4859. LNCS. 2007. ISBN: 978-3-540-77025-1. DOI: `10.1007/978-3-540-77026-8_29`. URL: `http://dx.doi.org/10.1007/978-3-540-77026-8_29`.

[Lip+16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. "ARMageddon: Cache Attacks on Mobile Devices." In: *USENIX Security Symposium*. USENIX Association, 2016, pp. 549–564.

[Lip+18] Moritz Lipp et al. "Meltdown." In: *meltdownattack.com* (2018).

[Man+05] Stefan Mangard, Thomas Popp, and BerndtM. Gammel. "Side-Channel Leakage of Masked CMOS Gates." English. In: *CT-RSA 2005*. Vol. 3376. LNCS. 2005. ISBN: 978-3-540-24399-1. DOI: `10.1007/978-3-540-30574-3_24`. URL: `http://dx.doi.org/10.1007/978-3-540-30574-3_24`.

[Man+07] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. Springer, 2007. ISBN: 978-0-387-30857-9.

[MC14] Debdeep Mukhopadhyay and Rajat Subhra Chakraborty. *Hardware Security - Design, Threats, and Safeguards*. CRC Press, 2014.

[Moo+18] Thorben Moos, Amir Moradi, Tobias Schneider, and François-Xavier Standaert. *Glitch-Resistant Masking Revisited - or Why Proofs in the Robust Probing Model are Needed*. Cryptology ePrint Archive, Report 2018/490. `https://eprint.iacr.org/2018/490`. 2018.

[Mor+11] Amir Moradi, Axel Poschmann, San Ling, Christof Paar, and Huaxiong Wang. "Pushing the Limits: A Very Compact and a Threshold Implementation of AES." In: *Proceedings of the 30th Annual International Conference on Theory and Applications of Cryptographic Techniques: Advances in Cryptology*. EUROCRYPT'11. Tallinn, Estonia: Springer-Verlag, 2011, pp. 69–88. ISBN: 978-3-642-20464-7. URL: `http://dl.acm.org/citation.cfm?id=2008684.2008693`.

[Mor+14] Pawel Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wójcik. *ICEPOLE*. Submission to the CAESAR competition: `http://competitions.cr.yp.to/round1/icepolev1.pdf`. 2014.

[Mos+12] Andrew Moss, Elisabeth Oswald, Dan Page, and Michael Tunstall. “Compiler Assisted Masking.” In: *CHES 2012, Leuven, Belgium, September 9-12, 2012. Proceedings*. 2012, pp. 58–75. DOI: `10.1007/978-3-642-33027-8_4`. URL: `http://dx.doi.org/10.1007/978-3-642-33027-8_4`.

[MR04] Silvio Micali and Leonid Reyzin. “Physically Observable Cryptography.” In: *Theory of Cryptography: First Theory of Cryptography Conference, TCC 2004, Cambridge, MA, USA, February 19-21, 2004. Proceedings*. Ed. by Moni Naor. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 278–296. ISBN: 978-3-540-24638-1. DOI: `10.1007/978-3-540-24638-1_16`. URL: `https://doi.org/10.1007/978-3-540-24638-1_16`.

[MS06] Stefan Mangard and Kai Schramm. “Pinpointing the Side-Channel Leakage of Masked AES Hardware Implementations.” English. In: *CHES 2006*. Vol. 4249. LNCS. 2006. ISBN: 978-3-540-46559-1. DOI: `10.1007/11894063_7`. URL: `http://dx.doi.org/10.1007/11894063_7`.

[Mui07] Edwin NC Mui. “Practical implementation of Rijndael S-box using Combinational logic.” In: *Custom R&D Engineer Texco Enterprise Pvt. Ltd* (2007).

[Nik+06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. “Threshold Implementations Against Side-channel Attacks and Glitches.” In: *Proceedings of the 8th International Conference on Information and Communications Security*. ICICS’06. Raleigh, NC: Springer-Verlag, 2006, pp. 529–545. ISBN: 3-540-49496-0, 978-3-540-49496-6. DOI: `10.1007/11935308_38`. URL: `http://dx.doi.org/10.1007/11935308_38`.

[NIS15] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. 2015. URL: `http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.202.pdf`.

[NIS95] NIST. *FIPS PUB 180-4: Secure Hash Standard*. Apr. 1995.

[O’D14] Ryan O’Donnell. *Analysis of Boolean Functions*. Cambridge University Press, 2014. ISBN: 978-1-10-703832-5. URL: `http://www.cambridge.org/de/academic/subjects/computer-science/algorithmics-complexity-computer-algebra-and-computational-g/analysis-boolean-functions`.

[Par05] Milind M. Parelkar. “Authenticated Encryption in Hardware.” MA thesis. Fairfax, VA, USA: George Mason University, 2005.

[PH13] Peter Pessl and Michael Hutter. "Pushing the Limits of SHA-3 Hardware Implementations to Fit on RFID." In: *Cryptographic Hardware and Embedded Systems – CHES 2013, 15th International Workshop, Santa Barbara, CA, USA, August 20-23*. Vol. 8086. Springer, 2013. DOI: `10.1007/978-3-642-40349-1_8`.

[Plo+10] Thomas Plos, Hannes Groß, and Martin Feldhofer. "Implementation of Symmetric Algorithms on a Synthesizable 8-Bit Microcontroller Targeting Passive RFID Tags." In: *Selected Areas in Cryptography*. Vol. 6544. Lecture Notes in Computer Science. Springer, 2010, pp. 114–129.

[QS01] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-measures for Smart Cards." English. In: *Smart Card Programming and Security*. Vol. 2140. LNCS. 2001. ISBN: 978-3-540-42610-3. DOI: `10.1007/3-540-45418-7_17`. URL: `http://dx.doi.org/10.1007/3-540-45418-7_17`.

[Rep+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes." In: *CRYPTO 2015*. 2015. DOI: `10.1007/978-3-662-47989-6_37`. URL: `http://dx.doi.org/10.1007/978-3-662-47989-6_37`.

[Rep15] Oscar Reparaz. "A Note on the Security of Higher-Order Threshold Implementations." In: *IACR Cryptology ePrint Archive* 2015 (2015). URL: `http://eprint.iacr.org/2015/001`.

[Rep16] Oscar Reparaz. "Detecting Flawed Masking Schemes with Leakage Detection Tests." In: *FSE 2016, Bochum, Germany, March 20-23, 2016, Revised Selected Papers*. 2016, pp. 204–222. DOI: `10.1007/978-3-662-52993-5_11`. URL: `https://doi.org/10.1007/978-3-662-52993-5_11`.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES." English. In: *CHES 2010*. Vol. 6225. LNCS. 2010. ISBN: 978-3-642-15030-2. DOI: `10.1007/978-3-642-15031-9_28`. URL: `http://dx.doi.org/10.1007/978-3-642-15031-9_28`.

[Sas+14] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose. *Minalpher*. DIAC 2014: Directions in Authenticated Ciphers, Santa Barbara, USA [Accessed: 2014 09 30]. Aug. 2014. URL: `http://2014.diac.cr.yp.to/slides/matsui-minalpher.pdf`.

[Sch+15] Tobias Schneider, Amir Moradi, and Tim Güneysu. "Arithmetic Addition over Boolean Masking - Towards First- and Second-Order Resistance in Hardware." In: *ACNS*. Vol. 9092. Lecture Notes in Computer Science. Springer, 2015, pp. 559–578.

[Tri03] Elena Trichina. "Combinational Logic Design for AES SubByte Transformation on Masked Data." In: *IACR Cryptology ePrint Archive* 2003 (2003). URL: `http://eprint.iacr.org/2003/236`.

[Wat76] Paul Watzlawick. *How Real Is Real?: Confusion, Disinformation, Communication.* Vintage books. Random House, 1976. ISBN: 9780394498539.

[XM88] Guo-Zhen Xiao and James L. Massey. “A spectral characterization of correlation-immune combining functions.” In: *IEEE Trans. Information Theory* 34.3 (1988), pp. 569–571. DOI: `10.1109/18.6037`. URL: `http://dx.doi.org/10.1109/18.6037`.

[YK13] Tolga Yalçin and Elif Bilge Kavun. “On the Implementation Aspects of Sponge-based Authenticated Encryption for Pervasive Devices.” In: *Proceedings of the 11th International Conference on Smart Card Research and Advanced Applications.* CARDIS’12. Graz, Austria: Springer-Verlag, 2013, pp. 141–157. ISBN: 978-3-642-37287-2. DOI: `10.1007/978-3-642-37288-9_10`. URL: `http://dx.doi.org/10.1007/978-3-642-37288-9_10`.

[Zös+15] Lukas Zöscher, Jasmin Grosinger, Raphael Spreitzer, Ulrich Muehlmann, Hannes Groß, and Wolfgang Bösch. “Concept for a security aware automatic fare collection system using HF/UHF dual band RFID transponders.” In: *ESSDERC*. IEEE, 2015, pp. 194–197.

[Zös+16] Lukas Zöscher, Raphael Spreitzer, Hannes Groß, Jasmin Grosinger, Ulrich Muehlmann, Dominik Amschl, Hubert Watzinger, and Wolfgang Bösch. “HF/UHF dual band RFID transponders for an information-driven public transportation system.” In: *Elektrotechnik und Informationstechnik* 133.3 (2016), pp. 163–175.

About the Author

Author information as of June 2018.

Personal Information

Name:	Hannes Groß
Date of birth:	December 19th, 1986
Place of birth:	Wolfsberg, Austria

Education

- **06/2013 – present:** Doctoral studies, Graz University of Technology, Austria.
- **07/2011 – 04/2013:** Master studies in Information and Computer Engineering (Telematik), Graz University of Technology, Austria.
- **10/2007 – 07/2011:** Bachelor studies in Information and Computer Engineering (Telematik), Graz University of Technology, Austria.

Professional and Academic Experience

- **06/2013 – present:** Project assistant, Institute for Applied Information Processing and Communications (IAIK), Graz University of Technology, Austria.
- **2013:** Contractor, NXP, Graz, Austria.
- **2012 – 2013:** Master's Thesis, NXP, Graz, Austria.
- **Summer 2010 and 2011**: Internship as software developer for RFID systems, Graz University of Technology, Austria.

Author's Publications

Author's publications as of June 2018.

Publications Used in this Thesis

▶ *Hannes Groß, Rinat Iusupov, and Roderick Bloem.* Generic Low-Latency Masking in Hardware. *CHES 2018 (in press)*

▶ *Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches." In:* EUROCRYPT (2). *Vol. 10821. Lecture Notes in Computer Science. Springer, 2018, pp. 321–353*

▶ *Hannes Groß and Stefan Mangard. "Reconciling d+1 Masking in Hardware and Software." In:* CHES. *vol. 10529. Lecture Notes in Computer Science. Springer, 2017, pp. 115–136*

▶ *Hannes Groß, David Schaffenrath, and Stefan Mangard. "Higher-Order Side-Channel Protected Implementations of KECCAK." in:* DSD. *IEEE Computer Society, 2017, pp. 205–212*

▶ *Hannes Groß, Stefan Mangard, and Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order." In:* CT-RSA. *vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112*

▶ *Hannes Groß, Manuel Jelinek, Stefan Mangard, Thomas Unterluggauer, and Mario Werner. "Concealing Secrets in Embedded Processors Designs." In:* CARDIS. *vol. 10146. Lecture Notes in Computer Science. Springer, 2016, pp. 89–104*

▶ *Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. "Suit up! — Made-to-Measure Hardware Implementations of ASCON." in:* DSD. *IEEE Computer Society, 2015, pp. 645–652*

Other Publications

▶ *Hannes Gross and Stefan Mangard. "A Unified Masking Approach." In:* Journal of Cryptographic Engineering *(2018).* ISSN*: 2190-8516.* DOI*: `10.1007/s13389-018-0184-y`*

▶ *Hannes Groß, Erich Wenger, Christoph Dobraunig, and Christoph Ehrenhöfer. "Ascon hardware implementations and side-channel evaluation." In:* Microprocessors and Microsystems - Embedded Hardware Design *52 (2017), pp. 470–479*

▶ *Lukas Zöscher, Raphael Spreitzer, Hannes Groß, Jasmin Grosinger, Ulrich Muehlmann, Dominik Amschl, Hubert Watzinger, and Wolfgang Bösch. "HF/UHF dual band RFID transponders for an information-driven public transportation system." In:* Elektrotechnik und Informationstechnik *133.3 (2016), pp. 163–175*

▶ *Hannes Groß, Marko Hölbl, Daniel Slamanig, and Raphael Spreitzer. "Privacy-Aware Authentication in the Internet of Things." In:* CANS*. vol. 9476. Lecture Notes in Computer Science. Springer, 2015, pp. 32–39*

▶ *Lukas Zöscher, Jasmin Grosinger, Raphael Spreitzer, Ulrich Muehlmann, Hannes Groß, and Wolfgang Bösch. "Concept for a security aware automatic fare collection system using HF/UHF dual band RFID transponders." In:* ESSDERC*. IEEE, 2015, pp. 194–197*

▶ *Hannes Groß. "Sharing is Caring - On the Protection of Arithmetic Logic Units against Passive Physical Attacks." In:* RFIDSec*. Vol. 9440. Lecture Notes in Computer Science. Springer, 2015, pp. 68–84*

▶ *Hannes Groß, Erich Wenger, Honorio Martín, and Michael Hutter. "PIONEER - a Prototype for the Internet of Things Based on an Extendable EPC Gen2 RFID Tag." In:* RFIDSec*. Vol. 8651. Lecture Notes in Computer Science. Springer, 2014, pp. 54–73*

▶ *Hannes Groß and Thomas Plos. "On Using Instruction-Set Extensions for Minimizing the Hardware-Implementation Costs of Symmetric-Key Algorithms on a Low-Resource Microcontroller." In:* RFIDSec*. Vol. 7739. Lecture Notes in Computer Science. Springer, 2012, pp. 149–164*

▶ *Thomas Plos, Hannes Groß, and Martin Feldhofer. "Implementation of Symmetric Algorithms on a Synthesizable 8-Bit Microcontroller Targeting Passive RFID Tags." In:* Selected Areas in Cryptography*. Vol. 6544. Lecture Notes in Computer Science. Springer, 2010, pp. 114–129*