ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

Low-Latency Threshold Implementations for Side-Channel Protected Cryptographic Hardware

Dušan Božilov

Supervisors:
Prof. dr. ir. Bart Preneel
Prof. dr. ir. Vincent Rijmen

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

March 2023

Low-Latency Threshold Implementations for Side-Channel Protected Cryptographic Hardware

Dušan BOŽILOV

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Science (PhD): Electrical Engineering

March 2023

Preface

Firstly, I would like to express my gratitude to Miroslav, Ventzi and Vincent for guiding me throughout my PhD. You brought me into the world of cryptographic hardware design. You have taught me so much, and allowed me to grow both professionally and as a person. The patience and understanding you had with me cannot be described in words, even long after my PhD was overdue. I can never repay you. I can only hope that I will be patient with somebody else later in life as you were with me.

Dear Bart, thank you taking me in as your student. Your comments were always insightful, and your lectures and talks were always inspirational and interesting.

Dear jury members, thank you for finding the time to go through the manuscript, spotting mistakes and providing the most useful feedback. The text is all the better because of your effort.

Dear COSICs, I was privileged to have been a part of such an amazing research group. I will keep many fond memories of this place, the conversations we had, the trips we went to. Péla, thank you so much so helping me with various administrative processes over the years, and for always bringing joy and happiness to wherever you are; I am sorry that I left those snoepjes go bad.

Barracks team, you made the PhD years such a joy with all the nonsensical things we did together. You are amazing friends, filled with joy, positivity and laughter. Danilo, thank you for enduring all those years as my flatmate. I will always look forward to our meaningful and meaningless discussions. Sara and Arthur, thank you for all the fun times we had and for the help you have provided me, be it translating my abstract or providing much useful advice to a new parent in a foreign country.

I want to express my gratitude to the European Commission for funding this research through the Horizon 2020 research and innovation programme Marie Skłodowska-Curie ITN ECRYPT-NET (Project Reference 643161). It was a privilege and pleasure to be one of the ECRYPT-NET fellows. I would also like to thank NXP Semiconductors and my former colleagues for providing an amazing work environment. Simon, I am glad to have had you as a fellow ECRYPT-NET researcher at NXP. Thank you for introducing me to the world of climbing and bouldering, and for all those fun squash matches.

I would like to thank my wife Nađa for loving and supporting me during the second half of my PhD. Particularly in the final year when she went out of her way to accommodate me while constantly taking care of our infant twin boys, Bogdan and Jakov. I do not know how you managed, but I am eternally grateful to you for it, ljubavi. Also many thanks to my family for being with me during this journey, helping me, guiding me and making me a better person as I matured.

Finally, I wish to thank my grandfather Stojan for instilling the love of mathematics and natural sciences in me when I was just a preschool child. Deda, you always wanted to see me complete my studies. I am sorry I was not fast enough to finish while you were alive, but hopefully I brought you joy wherever you are.

Abstract

Embedded applications that require security often rely on hardware cryptography to enable several security services. Cryptographic implementations should be transparent to the end-user, with minimal impact on the usability and performance of the enabled service. Therefore, the cost of the cryptographic layer should be minimized. Many applications require additional protection against tampering and side-channel attacks, which introduces additional costs to the implemented cryptographic service. This additional cost needs to be kept under control and should not significantly impact the performance of the system.

Achieving side-channel protection is a challenging task. The assumptions about the leakage from the target platform used to devise a countermeasure need to be sufficiently accurate and the countermeasure themselves need to provide adequate protection against the assumed leakage. Otherwise, the final device might still exhibit side-channel vulnerabilities, which can be seen from the amount of vulnerabilities uncovered in side-channel protected implementations over the years.

Low latency is critical for many practical applications. Secure boot and memory encryption are just some of the examples in which low latency is needed to ensure the responsiveness of the system, while the processing of sensitive data mandates the use of side-channel protection to thwart any side-channel attacks. Thus, providing low-latency side-channel protection is paramount in many cases.

In this thesis, we examine the aspects of low-latency design for side-channel protection. We specifically focus on side-channel protected hardware implementations. Several methods to achieve low-latency Threshold Implementations (TI) are presented. The main focus is on the algorithms that enable minimal area implementation of single-cycle nonlinear Boolean functions. The presented algorithms are generic and computationally feasible to a broad spectrum of functions used in cryptographic primitives, as shown by the application to relevant classes of Boolean functions of up to 8 bits. This thesis focuses on first- and second-order secure designs. However, higher-order secure designs can be constructed using the same methods.

The thesis also includes several hardware implementations of the PRINCE block cipher and detailed comparisons between them, emphasizing low-latency and low-energy aspects and their relation to the overall area of the final side-channel protected design. We also show that low-latency design can result in low-energy implementation in spite of the area penalty. The impact of the structure of the Algebraic Normal Form (ANF) of side-channel protected circuits is also examined on a single-cycle masked S-Box of AES (Advanced Encryption Standard); a large impact is shown on area and critical paths, with up to 50% difference in area and 10% difference in critical path.

We also discuss the optimizations of single S-Box serialized AES implementations which is the de facto standard for side-channel hardened AES realizations. The solutions presented allow for a full S-Box pipeline during the round, resulting in twenty cycles per round execution if the S-Box cycle latency is less than ten cycles.

Beknopte samenvatting

Geïntegreerde toepassingen die beveiliging vereisen, zijn vaak afhankelijk van hardware-cryptografie voor verschillende beveiligingstoepassingen. Cryptografische implementaties moeten onzichtbaar zijn voor de eindgebruiker en een minimale impact hebben op de bruikbaarheid en prestaties van de ingeschakelde toepassing. Bijgevolg moeten de kosten van de cryptografische laag zo minimaal mogelijk gehouden worden. Veel toepassingen hebben extra bescherming nodig tegen manipulatie en nevenkanaalsaanvallen wat extra kosten met zich meebrengt. Deze extra kosten moeten beteugeld worden en mogen de prestaties van het systeem niet significant beïnvloeden.

Beschermen tegen nevenkanaalsaanvallen is een uitdagende taak. De aannames over de lekkage van het doelplatform die gebruikt worden om een tegenmaatregel te bedenken, moeten voldoende nauwkeurig zijn; en de tegenmaatregelen moeten voldoende bescherming bieden tegen de veronderstelde lekkage. Zoniet, kan het uiteindelijke apparaat nog steeds kwetsbaar zijn voor nevenkanaalsaanvallen, zoals blijkt uit het aantal kwetsbaarheden dat door de jaren heen ontdekt is in nevenkanaalbeveiligde implementaties.

Lage latentie is van cruciaal belang voor veel praktische toepassingen. Veilig opstarten en geheugenencryptie zijn slechts enkele voorbeelden van toepassingen waarbij lage latentie noodzakelijk is voor een minimale reactietijd van het systeem, maar de verwerking van gevoelige gegevens tegelijk het gebruik van nevenkanaalsbescherming vereist. Daarom is het vaak van het grootste belang om nevenkanaalsbescherming met lage latentie te bieden.

In dit proefschrift onderzoeken we de aspecten van het ontwerpen van nevenkanaalsbescherming met lage latentie. We richten ons specifiek op nevenkanaalsbeschermde hardware-implementaties en presenteren verschillende methoden om threshold implementaties met lage latentie te realiseren. Dit proefschrift spitst zich vooral toe op de algoritmen die een minimale-oppervlakte implementatie van niet-lineaire Booleaanse functies met één cyclus toelaten.

De voorgestelde algoritmen zijn algemeen, en rekenkundig haalbaar voor een breed spectrum aan functies die gebruikt worden in cryptografische primitieven, zoals blijkt uit de toepassing ervan op relevante klassen van Booleaanse functies van maximaal acht bits. Dit proefschrift richt zich op veilige ontwerpen van de eerste en tweede orde, maar veilige ontwerpen van een hogere orde kunnen opgebouwd worden volgens dezelfde methoden.

Dit proefschrift bevat ook verschillende hardware-implementaties van het PRINCE blokvercijferingsalgoritme en een gedetailleerde vergelijking van deze verschillende implementaties. Hierbij ligt de nadruk op lage latentie en energiezuinigheid, en hun verhouding tot de totale oppervlakte van het uiteindelijke nevenkanaalsbeschermde ontwerp. We tonen hierbij aan dat een ontwerp met lage latentie kan resulteren in een energiezuinige implementatie ondanks de toename in oppervlakte. De impact van de structuur van de algebraïsche normaalvorm (ANF) van nevenkanaalsbeschermde stroomkringen werd onderzocht op een AES (Advanced Encryption Standard) gemaskeerde S-Box met één cyclus; we vonden hierbij een grote invloed op de oppervlakte en het kritieke pad, met tot 50% verschil in oppervlakte en 10% verschil in kritiek pad.

Ten slotte bespreken we ook de optimalisaties van AES-implementaties met één geserialiseerde S-box, de de facto standaard voor nevenkanaalsbeschermde AES-realisaties. De gepresenteerde oplossingen laten een volledige S-Box-pijplijn tijdens de ronde toe, resulterend in twintig cycli per uitvoeringsronde als de S-Box-latentie minder is dan tien cycli.

List of Symbols

$\boldsymbol{f}$	Shared representation of function f
$\boldsymbol{x}$	Shared representation of variable x
$\mathcal{I}$	Set of enumerated input shares
$\mathcal{S}$	Output sharing set
HD	Hamming distance
$\overline{x}$	First complement value of x
ρ	Percentage of covering sets dropped during neighborhood search in simulated annealing set covering solver
A_{io}	Input and output affine transform between PRINCE S-Box and its inverse
C_{131}	4-bit cubic permutation class
d	Security order
$d+1$	Threshold implementations with minimum $d+1$ input shares
D^n-table	Table representing output $d+1$ sharing of a degree n function
M	PRINCE forward round diffusion matrix
M'	PRINCE middle round diffusion matrix
n	Number of input bits
Q_{294}	4-bit quadratic permutation class
s_{in}	Number of input shares
s_{out}	Number of output shares

SR	PRINCE state nibble permutation
$td+1$	Threshold implementations with minimum $td+1$ input shares
wt	Hamming weight
x	Multi-bit variable
t	Algebraic degree

List of Abbreviations

AES Advanced Encryption Standard

ALU Arithmetic Logic Unit

ANF Algebraic Normal Form

ARX Addition-Rotation-XOR

ASIC Application Specific Integrated Circuit

CCS Composite Current Source

CGT Correctness Generator Table

CMOS Complementary Metal Oxide Semiconductor

CMS Consolidating Masking Schemes

CP Constraint Programming

CPA Correlation Power Analysis

DOM Domain-Oriented Masking

DPA Differential Power Analysis

EDA Electronic Design Automation

EMFI Electromagnetic Fault Injection

GE Gate Equivalent

IG Iterated Greedy

ISW	Ishai, Sahai and Wagner
LFSR	Linear Feedback Shift Register
LUT	Lookup Table
MAC	Message Authentication Code
MILP	Mixed Integer Linear Programming
ML	Machine Learning
NLRC	NonLinear-Register-Compression
PRNG	Pseudo-Random Number Generator
PVT	Power Voltage Temperature
SCA	Side-Channel Analysis
SCADA	Supervisory Control and Data Acquisition
SCP	Set Covering Problem
TI	Threshold Implementations
TVLA	Test Vector Leakage Assessment
WDDL	Wave Dynamic Differential Logic
XOF	Extendable-Output Function

Contents

List of Figures

List of Tables

Chapter 1

Introduction

"Speed is irrelevant if you are going in the wrong direction."

Mahatma Gandhi

The technological expansion that started in the second half of the twentieth century brought a new digital era to civilization, with electronic systems being an essential component in the lives of ordinary individuals. Digital communication became inexpensive, easily accessible, and fast, which led to the rise of various systems and applications leveraging novel channels of communication. The ability to quickly and reliably exchange information created a need to protect information from potentially malicious third parties. Cryptographic systems, originally exclusively used in a military and government setting, found their way into the everyday life. Online shopping, eBanking, and secure messaging applications rely on the ability to securely connect two or more physically distant parties over a digital channel, using cryptographic protocols and primitives to protect the data in transit. Cryptographic primitives also protect data at rest, i.e., sensitive databases such as medical or financial records.

Recently, with the expansion of the Internet of Things (IoT) devices and smart vehicles, the volume of data exchanged over public networks is drastically increasing. While allowing for previously unparalleled levels of connectivity, these devices often find themselves at the mercy of malicious parties. Smart devices with an internet connection have already been used as nodes in large *Distributed Denial of Service* (DDOS) attacks against widely used websites and online services [47]. Moreover, many IoT devices serve as sensors whose readings should only be performed by authorized parties. An unprotected smart meter with remote reading capability can be used as an indicator of residential presence, with low energy, gas, and water consumption providing a good indication a household is currently unoccupied and can be burglarized. Hence, protecting these devices and their data is of utmost importance. Additionally, low manufacturing costs must be maintained while stringent requirements for power, energy, and chip area have to be met, especially when a device is battery or passively powered. Thus, cryptographic protection also needs to be lightweight, low-cost, and effective. Finally, many IoT devices will by nature be located in remote locations, and they should also employ countermeasures against physical tampering while maintaining their low-cost and performance.

Industrial control systems are designed to ensure timely responsiveness, and a high degree of robustness. However, many of them were designed without considering security, as is the case for Supervisory Control and Data Acquisition (SCADA), the most widespread protocol for monitoring and control of various industrial systems. Consequently, numerous attacks on SCADA have been documented, with even more potential vulnerabilities known. Moreover, adding security to the existing protocol is an arduous task, as the cost of consuming the additional latency introduced by the security-providing component is too high. Thus, security considerations should be included during the design stage of any system/protocol operating in a distributed way. At the same time, cryptographic components ensuring security should impose as little strain on the system operation as possible, i.e., they should be lightweight and fast.

1.1 Cryptography

Cryptography is a science of studying methods that allow secure communication between several parties over an untrusted channel by ensuring confidentiality (secrecy) and integrity of data in transit or at rest. Additionally, cryptography provides mechanisms used for authentication of entities, allowing for restricted access of sensitive data in a system. Cryptography is also used for non-repudiation, i.e., providing proof of origin of data or action.

Encryption is a method that transforms a message (plaintext) into a seemingly random looking string (ciphertext) using an algorithm, a *cipher*. A cipher takes an additional secret input, called the *key*, to produce the ciphertext. Decryption is a method of retrieving the original message by applying the inverse cipher operation on the ciphertext, using the key as additional input. Message decryption should not be possible to any party which does not have the decryption key. Modern encryption algorithms rely on *Kerckhoffs' principle* which states that secrecy of the plaintext relies solely on the input *key* being unknown to the adversary, with the assumption that the details of encryption and decryption method are publicly known and available.

Cryptographic primitives can be divided into three categories:

- Symmetric key primitives.
- Asymmetric key primitives.
- Keyless primitives.

Symmetric key cryptography allows for secure communication between two parties if they share the same secret key. Figure 1.1 depicts both the encryption and decryption operation in the symmetric key setting, in which the same key is used both during encryption and decryption.

Symmetric primitives are efficient to implement in both software and hardware, typically using only bit manipulation instructions, arithmetic operations, and lookup tables (LUTs). *Block ciphers* are symmetric encryption algorithms which take an input block of the fixed size of m bits, and they produce the output ciphertext block, also consisting of m bits. The most famous block cipher example is Advanced Encryption Standard (AES) based on Rijndael [28]. AES is the predominant block cipher, with fast implementations both in software and in hardware. Processor manufacturers such as Intel, AMD and ARM even employ specialized instructions to expedite the AES computation in their CPUs. Specialized block ciphers designed for constrained devices are other notable examples, such as PRESENT [10] and PRINCE [12]. These ciphers have a smaller block size than AES and do not guarantee the same level of security, but they are less costly to deploy in hardware. Authenticity of transmitted messages between two parties is achieved by appending a *message authentication code* (MAC).

The most prominent keyless primitives are *cryptographic hash functions* which are typically used to create digital fingerprints as they provide a fixed length output regardless of the input size. Most commonly used hashing algorithms are SHA-3 [60] based on KECCAK permutation [4], and SHA-2 [35]. MAC algorithms often rely on cryptographic hash functions, e.g., HMAC-SHA-256 [52] is a hash-based MAC algorithm. Extendable-Output Functions (XOF) are another widely used keyless primitive, which takes an input bit string as input and can provide an output of any length. SHAKE-128 and SHAKE-256 [60] are KECCAK based examples of XOFs. An asymmetric encryption cryptographic primitives rely on using a pair of two distinct keys, and are predominantly used for establishing a secure channel between two parties. Similarly, digital

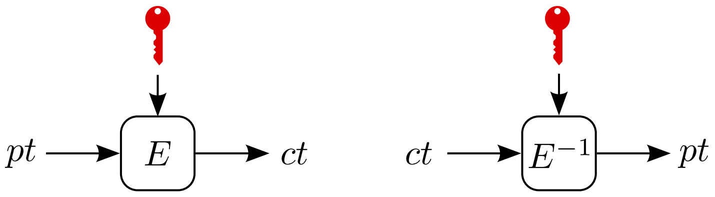


Figure 1.1: Symmetric encryption and decryption operations.

signatures leverage assymetric cryptography by using one key to create a signature during the signing step, and another paired key during signature verification step.

1.2 Modeling the Cryptographic Attacker

The original Kerckhoffs principle naturally lends itself to the *black-box* attacker model in which the attacker has access to the plaintext, ciphertext, and the algorithm itself, but not the key. The black-box attacker model is sufficient for attackers with access to the communication channel but not the device that performs the cryptographic operation. If the device is accessible to the attacker, the black-box model is invalidated because the adversary can exploit the physical nature of the implementation of the algorithm to recover the key. The adversary can either observe the physical state of the device or even attempt to disrupt the regular operation of the device. This more potent attacker model is referred to as *gray-box* model. The attacker's primary goal in the gray-box model is not to mathematically break the underlying cryptographic algorithm but to recover the key, whose value indirectly manifests itself via its physical state. The gray-box model is presented in Figure 1.2.

Gray-box attacks can be characterized into two groups:

- Passive attacks are characterized by the non-interference of the adversary with the operation of the device [51, 18, 37]. Instead, the device behavior is monitored while it executes a cryptographic operation. The attacker then tries to uncover the secret key from the obtained additional data. The most commonly measured characteristics of the device are execution time, instantaneous power consumption and electromagnetic radiation. Passive attacks are also called *side-channel* attacks, as they can be interpreted as an additional channel which the adversary has access to, in addition to the communication channel.

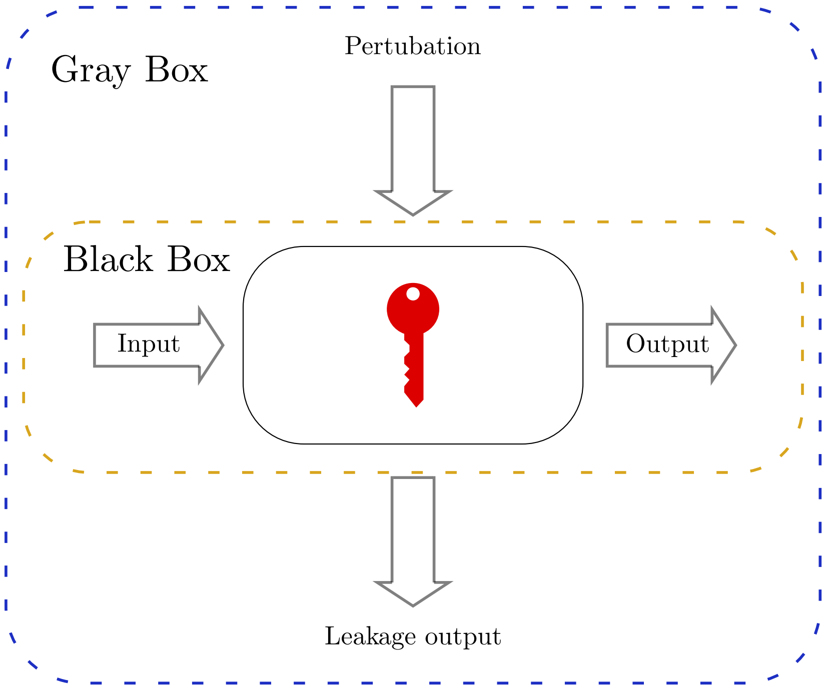


Figure 1.2: Gray-box Cryptography.

- Active attacks involve a perturbation of the internal state of the device while the cryptographic algorithm is being executed, with the goal of producing incorrect output [11, 6]. The faulty output is either used to undermine the integrity of the system if undetected or to recover the secret key if it can be paired with the known correct output. Voltage and clock glitching, and Electromagnetic Fault Injection (EMFI) are the most common methods used to induce a fault by causing multiple storage elements to update their outputs incorrectly[70, 3]. Laser fault injection is another method, which can be used to precisely target any single gate on the chip, allowing the adversary a much finer control over the attack. Active attacks are invasive if the device cannot return to normal operation after the attack, e.g., by removing the packaging of the chip or by shortcircuiting PRNG inputs of the device. Non-invasive attacks do not permanently impact the device. Hence the device can normally operate after the attack. Invasive attacks are often used to profile the device, to discover the most suitable points, both on die and in time, which can be used to mount a successful attack.

- Combined attacks perform a combination of both passive and active attacks to recover the key or invalidate device's security. The best example of a combined attack is faulting of the Pseudo Random Number Generator (PRNG) output to a constant value, which diminishes the protection level of side-channel countermeasures, followed by mounting a side-channel attack against the weakened implementation.

Both passive and active attacks pose a severe threat to the security of many embedded devices today. Moreover, the cost of mounting a successful attack is continuously decreasing, making these attacks more readily available. Hence, correctly applying countermeasures against such attacks is becoming increasingly important.

The device's operating environment should be carefully evaluated to estimate potential attacker capabilities and appropriate countermeasures should be implemented accordingly. An inadequate attacker model can lead to exploitable vulnerabilities, e.g., a side-channel only protected device can be easily broken if the adversary can mount active attacks. Conversely, a too powerful attack model would lead to an unnecessarily expensive device, with considerable power consumption and chip area.

1.3 Motivation

Many digital systems mandate both fast performance and side-channel protection, a daunting task, as most countermeasures introduce additional latency, notwithstanding the area overhead of the countermeasures. However, many applications do require both side-channel resilience and low latency. Thus, minimizing the area overhead while limiting the performance impact of side-channel countermeasures is critical for many applications in practice.

Memory encryption is probably the most obvious example where encryption (decryption) latency has a significant impact on the overall system performance. The standard memory encryption implementation is given in Figure 1.3. Especially when dealing with external memory, it is of utmost importance to protect the memory encryption key against side-channel attacks. The volume of data being encrypted/decrypted with one key in a memory encryption setting is high, allowing adversaries to mount side-channel attacks and easily recover the key if no countermeasures are in place.

Secure boot is a method to ensure that the code image being run on a processor comes from a trusted source and has not been tampered with on disk. Typically, the secure boot is achieved by signing the hash of the boot image and checking

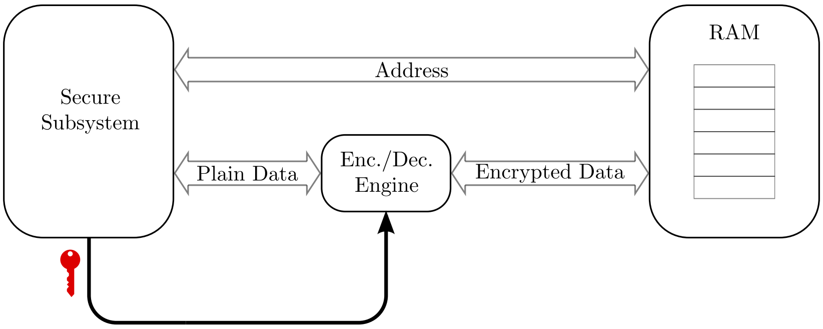


Figure 1.3: Memory encryption diagram.

the signed hash against the computed signature after power up. Alternatively, a MAC can be used to create a tag of the boot image, that again needs to be checked at power up. Similar to memory encryption, a large amount of data is processed during the hash computation using the same key, allowing adversaries to mount side-channel attacks during the MAC computation. Hence side-channel countermeasures should be in place to thwart side-channel attacks while maintaining the overall speed of the boot image verification. The secure boot speed is critical in scenarios in which the device idling in low power mode most of the time, only occasionally waking up to quickly perform a specific task and returning to low power mode. As waking up from low power mode initiates a secure boot process, keeping the secure boot execution time minimal is paramount to responsiveness and low energy consumption of the device.

Secure test and debug is another application in which a testing entity needs to authenticate itself to the device to gain elevated access control of the device. Allowing only authenticated parties to access the device test and debug ports while keeping the access time low is vital as it translates directly into the silicon manufacturing cost.

1.4 Organization

The work presented in this thesis focuses on reducing the cost of low-latency side-channel countermeasures in hardware implementations, evaluating the impact of side-channel countermeasures on the performance and chip area, and examining trade-offs during the design stage of side-channel protected implementations concerning latency, area, and power/energy consumption. The main contributions from the thesis have been published internationally [14, 17, 15, 79]. I was the main author in three publications [14, 17, 15] while in Shahmirzadi et al. [79] I contributed with an efficient TI sharing of an 8-bit cubic Boolean function, and a construction method for achieving such a sharing. The thesis is organized in the following manner:

- Chapter 2 gives necessary background on hardware design, side-channel attacks and countermeasures, focussing on Threshold Implementations (TIs) [64] and state of the art in low-latency side-channel design.
- Chapter 3 explores the construction of optimal TIs for Boolean functions of various algebraic degrees, and for different security orders. These constructions are then practically explored in subsequent sections, and they present the foundation of the thesis. The work presented in this chapter is based on work published in [14, 79, 15].
- Chapter 4 discusses several side-channel protected implementations of the PRINCE [12]. The main focus of the chapter is the comparison of standard hardware metrics between variants of the protected circuits, with special attention being given to latency and energy consumption. This Chapter is based on work presented in [14, 15, 17].
- Chapter 5 discusses a first-order side-channel resistant single cycle AES S-Box implementation and examines how the choice of the ANF of the shares impact on area and latency. Additionally, an algorithm for the efficient pipeline of serialized single S-Box AES is presented, allowing for minimal latency of one encryption operation, despite the multi-stage layout of the protected S-Box. The work presented in this chapter is based on findings published in [15] as well as previously unpublished results.
- Chapter 6 summarizes the work presented in this thesis and explores the future research directions of low-latency side-channel protected implementations.

Chapter 2

Preliminaries

"To acquire knowledge, one must study; but to acquire wisdom, one must observe."

Marilyn vos Savant

In this chapter, we provide an overview of the theoretical notions required to understand subsequent chapters. We start with the overview of digital hardware design, followed by the basis of Boolean algebra for cryptography. The bulk of the chapter, however, is the background on side-channel attacks, as well as protection against such attacks, with particular attention given to TI, the most commonly used Boolean masking technique for hardware circuits.

2.1 Hardware Design

Digital circuits are the dominant electronic circuits type used to compute discrete values. They are characterized by their ability to distinguish between two values. Thus, the underlying physical properties could be separated from the functionality of the circuit, allowing for the abstraction of the two signal levels by interpreting them as logical zero and logical one. Hence, the designer needs not know the physical properties of the technology in great detail, but can still abstract the two signals level into binary ones and zeros and apply discrete mathematical operations on such hardware. The rise of Electronic Design Automation (EDA) during the 1980s brought an even larger expansion

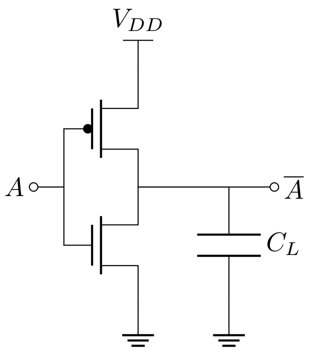


Figure 2.1: Logic inverter in CMOS technology.

of digital design. Complex digital circuitry could now be efficiently translated to available logic cells within a well-characterized and tested library of cells. Thus, the designers could focus their effort on the intricacies of the design instead of manually optimizing Boolean functions using Karnaugh maps.

The dominant digital electronics technology is Complementary Metal Oxide Semiconductor (CMOS), due to its extreme noise resistance and traditionally low static power consumption. While the static power consumption has increased with the decrease in technology size, it has remained one of the most used fabrication processes in digital design since the late 1960s.

The total power of a CMOS circuit is $P = P_{static} + P_{dynamic}$. It has two major components, static and dynamic power. Static power is the power due to parasitic leakage currents, while dynamic power is the power drawn from the voltage source while the circuit is computing a new output value. Typically, static power is several orders of magnitudes smaller compared to dynamic power, but with the increasing scaling, static power consumption is comparable to dynamic power consumption in smaller technology processes.

The dynamic power consumption of a CMOS circuit can be explained using a CMOS inverter, shown in Figure 2.1. This example, while simple, showcases how dynamic power is dependent on the data processed by the circuit. The energy consumption during the $0 \rightarrow 1$ transition on the output can be calculated as the integral of the instantaneous power, equal to voltage and current product during the transition, computed as follows:

$$E_{tot} = \int_0^T P_{DD}(t)dt = V_{DD} \int_0^T I_{DD}(t)dt = V_{DD} \int_0^{V_{DD}} C_L dV_{out} = C_L V_{DD}^2. \tag{2.1}$$

On the other side the energy E_C stored in the output capacitor C_L is

$$E_C = \int_0^T P_C(t)dt = \int_0^T V_C(t)I_C(t)dt = \int_0^{V_{DD}} V_{out}C_L dV_{out} = \frac{1}{2}C_L V_{DD}^2 \tag{2.2}$$

We can see that the total energy of the capacitor in the end E_C is only half of the energy provided by the voltage source E_{tot}. The rest of the energy dissipated during the transition in the upper PMOS transistor. Similarly during the $1 \rightarrow 0$ transition, the output capacitor will be discharged through the lower NMOS transistor, dissipating the stored energy in the process. In the simple model presented here, the energy that dissipates when the output changes its value is the same for both $0 \rightarrow 1$ and $1 \rightarrow 0$ transitions. However, due to multiple effects not taken into account, such as parasitic resistance of the source and other parasitic elements, the dissipation energy will slightly differ in practice for these two transitions.

The dynamic power of a CMOS circuit can be computed as the average energy of a transition times the switching frequency of the gate. The switching activity directly depends on the inputs of the CMOS logic circuit. This relation between the dynamic power and the processed values is the basis of side-channel attacks, as explained in Section 2.4.

Although EDA tremendously reduced the burden of hardware designers, they still need to be aware of the various physical aspects of the circuit. The number of logic cells used impacts the area of the implementation, which is often limited by the available die surface on a chip. Moreover, equivalent capacitance of the implementation is quadratically proportional to the total area [71], which increases the power and energy consumption.

The time between the output value change as the result of an input change of a logic cell is called *output delay*. Different cells have different output delays, and even the output delay of a single cell varies between different inputs, as well as different transitions of the same input. Thus, digital circuits utilizing multiple logic cells have to be adequately synchronized to operate correctly. The most common way to achieve synchronization is to incorporate a periodic pulse signal, called the *clock*, with one period of the clock signal often being called clock cycle. To ensure synchronization the clock signal is paired with logic cells that only update their value once per clock period, called *registers* or *flip-flops*. Register outputs are updated at every positive edge (0 to 1 transition) of the clock signal. More rarely, some register designs update their values on the negative clock edge, or both, to achieve synchronization. The digital logic between register outputs is frequently referred to as combinatorial logic. Combinatorial logic typically determines the maximal operating frequency of the circuit, which impacts the power and energy consumption.

The number of clock cycles needed to complete an operation is referred to as *cycle latency*. The absolute amount of time needed to complete an operation is referred to as simply *latency*. The number of output bits a design can output per cycle is called *throughput*. In many applications, overall latency and operating frequency are given as a requirement. Thus, the designer only has freedom to investigate implementations with different cycle latencies. A circuit can be designed for maximal throughput or minimal cycle latency, depending on the requirements of a specific application. Cycle latency directly impacts power and energy consumption, since requiring fewer cycles to complete an operation might decrease the energy consumption. Lower cycle latency might lead to increased power consumption due increased logic switching activity within one clock cycle, negatively impacting the energy. Conversely, higher cycle latency by introducing additional register stages might allow for higher maximal operating frequency, which can result in lower energy consumption of the design. Hence, the appropriate design choice between cycle latency and minimizing energy consumption is application specific.

2.2 Mathematical Background

Classical cryptography relies on the foundation of discrete mathematics, and here we will quickly introduce some basic notions of Boolean algebra. Boolean algebra operates with variables that can take two different values, 0 or 1. A Boolean function takes several input variables and produces a single output. Alternatively, a Boolean function is mapping from the vectorspace $\mathbb{F}_2^n$ of all binary vectors of length n, to the finite field $\mathbb{F}_2$ [21]. Vectorial Boolean function takes a number of input Boolean variables and produces a multi-dimensional output, a binary vector x. Each bit of x can be considered as a separate Boolean function on its own. Alternatively, a vectorial Boolean function is a mapping of vectorspace $\mathbb{F}_2^n$ to the vectorspace $\mathbb{F}_2^m$ for some positive integers n and m [22]. Elements of vectorspace $\mathbb{F}_2^n$ with 2^n elements are represented using n-bit values.

To argue about Boolean functions further we need to first introduce the notions of *Hamming weight* and *Algebraic normal form*.

Definition 2.2.1. *The Hamming weight $wt(x)$ of a positive integer x is the number of 1s in the binary representation of x.*

Every n-bit Boolean function S can be represented uniquely by its *Algebraic Normal Form* (ANF) [21]:

$$S(x) = \bigoplus_{i=(i_1,\ldots,i_n)\in\mathbb{F}_2^n} a_i x_1^{i_1} x_2^{i_2} \cdots x_n^{i_n}. \tag{2.3}$$

Here $x_1, \ldots, x_n$ represent the input bits of input variable x. The power function $x_k^{i_l}$ is equal to x_k if i_l is 1, or 1 otherwise. The *algebraic degree* of a Boolean function S is defined as the maximal value of the Hamming weight $wt(i)$ in Equation (2.3) for which the coefficient a_i is not zero. The notion of algebraic degree extends to vectorial Boolean functions in which the algebraic degree is the maximum of algebraic degrees of the output coordinates. Boolean functions whose algebraic degree is 1 are referred to as *linear*. Boolean functions whose algebraic degree is greater than 1 are referred to as *nonlinear*. Boolean functions whose algebraic degree is equal to 2 are also referred to as quadratic Boolean functions, while Boolean functions of degree 3 are also referred to as cubic Boolean functions. Nonlinear vectorial Boolean functions are one of the crucial components of many symmetric cryptographic algorithms, and are also called *substitution boxes* or *S-Boxes*. Linear vectorial Boolean functions with constant coefficient of 0 are also referred to as *linear* transforms. Linear vectorial Boolean with nonzero constant coefficient are called *affine* transforms. S-Boxes are often represented with their LUTs, which are frequently suitable for software and hardware implementations.

A Boolean function is balanced if its output takes values of 0 and 1 for an equal amount of inputs [21]. A Vectorial Boolean function is balanced if and only if all of the linear combinations of its output component functions are balanced. Balanced vectorial Boolean functions that map elements of the field $\mathbb{F}_{2^n}$ to the same field $\mathbb{F}_2^n$ are called permutations. An affine permutation is a permutation with algebraic degree equal to one.

A *finite* or *Galois* field is a field with a with finite number of elements. The number of elements, also known the order of a field, in a finite field is a prime, or a power of a prime. We refer to a finite field with an order p^n, p being a prime, and n being a positive integer, as $\mathbb{F}_{p^n}$. All finite fields of the same order are identical up to an isomorphism. A finite field $\mathbb{F}_{p^n}$ can be represented using an irreducible polynomial of degree n, whose coefficients are elements of the field $\mathbb{F}_p$.

It is well known that for a finite field with N elements, $a^N = a$ for each element a of the field. Hence $a^{N-1} = 1$ for any nonzero element a, and multiplicative inverse element of a is $a^{-1} = a^{N-2}$. AES S-Box uses multiplicative inversion in F_{2^8} field represented by irreducible polynomial $x^8 + x^4 + x^3 + x + 1$.

Another representation of a finite field is by using a normal basis. A field element β is normal basis generator in finite field $\mathbb{F}_{p^n}$ if and only if

$$(\beta_{n-1}, \beta_{n-2}, \ldots, \beta_1, \beta_0) = (\beta^{p^{n-1}}, \beta^{p^{n-2}}, \ldots, \beta^{p^1}, \beta^{p^0})$$

forms a vector space basis for $\mathbb{F}_{p^n}$ over $\mathbb{F}_p$ [44]. Each element a of the field can be expressed in the normal basis using its coefficients $(a_{n-1}, \ldots, a_0)$:

$$a = a_{n-1}\beta_{n-1} + \ldots + a_0\beta_0 \,.$$

Two vectorial Boolean functions S and S' are affine equivalent if and only if there exists affine permutations A and B satisfying $S' = A \circ S \circ B$. The operation $\circ$ indicates composition of permutations. We refer to A as the output and B as the input transformation. All permutations over the vectorspace $\mathbb{F}_2^n$ can be separated into multiple equivalence classes. Affine equivalence classes are frequently used as several important cryptographic properties are preserved within one class, most notably the algebraic degree.

2.3 Block Ciphers

Here we will briefly describe AES and PRINCE block ciphers, whose implementations are discussed in Chapters 3, 4 and 5.

2.3.1 AES

As mentioned in Section 1.1 encryption standard AES based on the Rijndael cipher is a round-based block cipher design, supporting 128, 192 and 256 bit key sizes, with a block size of 128 bits. The number of rounds is 10, 12, and 14, depending on the size of the key. The state consists of 16 bytes $s_0, \ldots, s_{15}$, and is organized as a 4×4 matrix of bytes in the following manner:

$$\begin{bmatrix} s_0 & s_4 & s_8 & s_{12} \\ s_1 & s_5 & s_9 & s_{13} \\ s_2 & s_6 & s_{10} & s_{14} \\ s_3 & s_7 & s_{11} & s_{15} \end{bmatrix}$$

AES round structure consisting of four steps:

- **AddRoundKey**: each byte of the state is XORed with the corresponding byte of the round key. AddRoundKey is also performed before the first round.

- **SubBytes**: Each byte is subjected to the permutation specified by the AES S-Box. The S-Box is based on an affine transformation of the multiplicative inverse in the Galois field $\mathbb{F}_{2^8}$ represented with the polynomial $x^8 + x^4 + x^3 + x + 1$. Zero value of the inversion is mapped to itself. S-Box $y = S(x)$ can then be expressed in a following manner

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \\ y_6 \\ y_7 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \\ z_4 \\ z_5 \\ z_6 \\ z_7 \end{bmatrix} + \begin{bmatrix} 1 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

  where $y_7, \ldots, y_0$ and $z_7, \ldots, z_0$ are individual bits of y and z, respectively, and z is the multiplicative inverse of x.

- **ShiftRows**: Four bytes of the i-th row of the state are rotationally shifted by i positions to the left.

- **MixColumns**: Each column consisting of four bytes (x_0, x_1, x_2, x_3) is transformed to four bytes (y_0, y_1, y_2, y_3) by the invertible linear transformation:

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

  MixColumns operation is not performed in the last round.

AES key schedule

AES round keys are obtained by expanding the input key using the AES key schedule.

If we define the following:

- AES round constants are 32-bit values defined as 4 bytes $rcon_i = [rc_i \; 0 \; 0 \; 0]$, where $rc_1 = 1$ and $rc_i = x \cdot rc_{i-1}$ in the AES finite field represented with polynomial $x^8 + x^4 + x^3 + x + 1$.

- R_n as number of rounds

- N as number of 32-bit words in the input key

Algorithm 1: AES key expansion algorithm.

```
Input: Input key $k = (K_0, \dots, K_{N-1})$, number of rounds $R_n$
Result: Expanded key $w = (W_0, \dots, W_{4R_n-1})$
for $i = 0$ to $N - 1$ do
    $W_i = K_i$
for $i = N$ to $4R_n - 1$ do
    if $i \equiv 0 \mod N$ then
        $W_i = W_{i-N} \oplus \text{SubWord}(\text{RotWord}(W_{i-1})) \oplus rcon_{i/N}$
    else if $i \equiv 4 \mod N$ then
        $W_i = W_{i-N} \oplus \text{SubWord}(\text{RotWord}(W_{i-1}))$
    else
        $W_i = W_{i-N} \oplus W_{i-1}$
```

- RotWord as a one byte rotational left shift of a 32-bit word
- SubWord as AES S-Box operation on each byte in 32-bit word
- $K_0, \dots, K_{N-1}$ as input key separated into N 32-bit words
- $W_0, \dots, W_{4R_n-1}$ as 32-bit words of expanded key

then Algorithm 1 represents the key expansion of AES.

2.3.2 PRINCE

PRINCE [12] is a block cipher specifically designed for low-latency hardware implementations. The PRINCE block size is 64 bits, with a 128-bit key size, and the input key is used to derive three 64-bit internally used keys k_0, k_0' and k_1. Its α-reflection property allows reuse of the same circuitry for both encryption and decryption. Although not designed to be efficient in software, a bit-sliced software implementation of PRINCE is surprisingly fast and can even be executed in fewer clock cycles compared to other lightweight block ciphers such as PRESENT and KATAN [66].

The input key is split into two 64-bit parts $k_0||k_1$ and expanded to $k_0||k_0'||k_1$ as shown below.

$$(k_0||k_0'||k_1) = k_0||((k_0 \ggg 1) \oplus (k_0 \gg 63))||k_1 \,.$$

The PRINCE block diagram is shown in Figure 2.2. As depicted, k_0 and k_0' are used as whitening keys at the start and at the end of the cipher, while k_1 is used

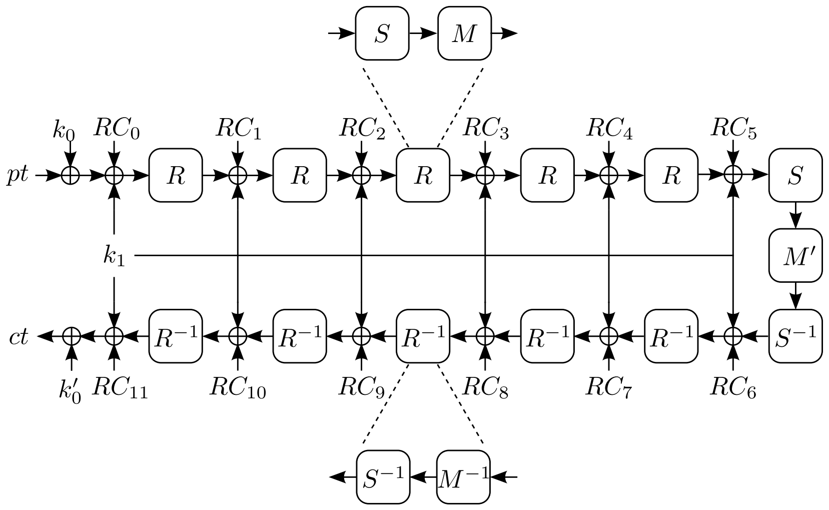


Figure 2.2: PRINCE cipher.

as round key in PRINCE$_{core}$ which consists of 12 rounds. More precisely, 12 rounds are divided into 6 “forward” rounds, followed by the middle involution layer, and finally 6 inverse rounds are applied at the end.

S-Box layer. The PRINCE S-Box is a 4-bit permutation of algebraic degree 3 and its look-up table is given in Equation (2.4).

$$S(x) = [B, F, 3, 2, A, C, 9, 1, 6, 7, 8, 0, E, 5, D, 4]\,. \tag{2.4}$$

The S-Box inverse is in the same affine equivalence class as the S-Box itself. Moreover, the input and output transformations are the same:

$$S^{-1} = A_{io} \circ S \circ A_{io}\,. \tag{2.5}$$

The LUT representation of the affine transformation A_{io} is given by

$$A_{io}(x) = [5, 7, 6, 4, F, D, C, E, 1, 3, 2, 0, B, 9, 8, A]\,.$$

Linear layer. The Matrices M and M' define the diffusion layer of PRINCE. M' is an involution, and the matrix M can be constructed from M' by applying the shift-rows operation SR so that $M = SR \circ M'$. Recall that SR is a linear operation that permutes the nibbles of the PRINCE state.

The RC_i addition is a 64-bit round constant addition. The round constants $RC_0, \ldots, RC_{11}$ are chosen such that $RC_i \oplus RC_{11-i} = \alpha$ with α being a 64-bit constant, $\alpha =$ `c0ac29b7c97c50dd`. Observing the structure of the PRINCE$_{core}$, it can be seen that the PRINCE$_{core}$ inverse with key $k_1 = k$ is equal to PRINCE$_{core}$ with key $k_1 = k \oplus \alpha$, making the decryption circuit easy to implement in hardware.

2.4 Side-Channel Attacks

As already stated in Section 1.2, side-channel attacks focus on exploiting a weakness in an implementation by collecting additional information from the device under attack, opposed to breaking the underlying cryptographic primitive, to recover the secret information. Side-channel attackers merely observe one or more characteristics of the device while computing the output of a cryptographic primitive, and then try to recover the key from the collected additional data. The type of auxiliary data can vary depending on the implementation and specific attack, the most common ones being power consumption, electromagnetic radiation, and execution time. The model where the attacker obtains extra side-channel information while the device is executing is called the *gray-box model*, while the classical *black-box model* assumes that only inputs or outputs are visible to the attacker, mandating the use of attacks that mathematically break the underlying cryptographic primitives.

Side-channel attacks first emerged in the 1990s, with the discovery of timing attacks by Kocher [51] on various public key primitives by exploiting the differences in execution time that could be associated with the private key value. Kocher et al. [50] then showed how to recover the key from the chip within a smart card using power consumption. As discussed in Section 2.1 the power consumption depends on the switching activity of the circuit, which in turn depends on the values circuit is processing. More formally, the power consumption can be expressed as follows:

$$P(x) = \mathcal{L}(x) + \mathcal{N}. \tag{2.6}$$

Here, $\mathcal{L}$ is the leakage function, expressing the dependency of the power consumption on the chosen intermediate value, while $\mathcal{N}$ is the noise component of the power consumption. The noise function consists of actual device and measurement noise, present in any electronic circuit, and the power consumption of elements of the circuits not operating on the intermediate value x under attack.

Another widely used leakage model is the probing model of Ishai et al. [45] which assumes an attacker who places a fixed amount of probes in the circuit, and can read out their exact value as the circuit operates. The model is also referred to as d-probing model where the number of probes available to the attacker is upper bounded by d. The probing model is extremely useful due to its simplicity and it lends itself to the scrutiny of mathematical formalism. Furthermore, side-channel security in the probing model by Duc et al. [34] implies side-channel security in the noisy leakage model.

Differential Power Analysis

Side-channel attacks follow a simple but effective divide and conquer strategy. The key is partitioned into several smaller length blocks, and the attacker tries to recover each block independently. In the original power side-channel attack, blocks were only one bit. The device which stores the secret key is tasked with running multiple encryption operations with varying inputs. The power consumption is recorded during each encryption. We refer to the power consumption signature of one encryption operation as a power trace. Next the attacker chooses an attack point, a value that is dependent on the secret key bit k_b the attacker is trying to recover, most notably the input or the output of an S-Box. Next, for both potential values of the key bit k_b, traces are partitioned into two groups. The first group is all the traces in which the key-dependent value v_k is 0 when k_b is 0. The second group contains all the traces where v_k is equal to 1 if k_b is 0. Two average traces are then computed for the two groups, and their difference is used to create a differential trace Δtr_{kb_0}. The same procedure is then repeated to obtain differential trace Δtr_{kb_1}, where the partitioning of traces is done assuming k_b is equal to 1. If the power consumption does depend on the value of k_b, then if the correct key bit k_b is 0 there should exist a noticeable peak in differential trace Δtr_{kb_0} as the partitioning is done correctly and the power consumption will differ at the time position at which key bit v_k is computed. Trace Δtr_{kb_1} should not exhibit any significant peaks, as the partitioning is not correct, so the final averaging should produce only noise. Calculation of differential traces for two values of k_b is given below

$$\Delta tr_{kb_0} = mean(tr(i), v_k(k_b = 0, pt(i)) = 0) - mean(tr(i), v_k(k_b = 1, pt(i)) = 1)$$

$$\Delta tr_{kb_1} = mean(tr(i), v_k(k_b = 1, pt(i)) = 0) - mean(tr(i), v_k(k_b = 1, pt(i)) = 1)\,.$$

Correlation Power Analysis

The original attack method was quickly improved, allowing the possibility to target multiple bits at once. *Correlation Power Analysis* (CPA) [18] expands on DPA by assuming a leakage model, and then leveraging the statistical correlation tools to exploit the dependency between the key material and power traces. In correlation-based attacks, the attacker introduces additional assumptions on the noisy leakage model to enhance the potency of the attack. The most commonly used leakage models are based on Hamming weight and Hamming weight distance. The first one accurately models the leakage of memory cells. The second one is more beneficial when modeling the temporary storage registers in digital circuits because in most fabrication processes registers consume significantly more power when their output value changes.

Attack point, ideally a register or memory element output, is targeted, as they contribute more than the combinatorial logic, reducing the noise seen by the attacker. Additionally, the output of nonlinear operations is shown to be favorable for the attacker [69].

Once the attack point is chosen, the attack collects n power traces during which encryption (or decryption) is run on the target device with one key. Using a leakage model of choice, the attacker calculates the leakage value based on the guess of the key and input given to the device. The Pearson correlation is then independently computed for each time sample of the traces. If the leakage model sufficiently accurate represents the leakage of the device, the correct key guess will have the highest absolute correlation value at the time samples during which the targeted intermediate value is being evaluated.

Template attacks and machine learning

Another improvement on the original DPA attack is template attacks [23], where the attacker profiles the power consumption of the target device to create a dictionary of power signatures associated with the leaky intermediate values. If profiling is successful, template attacks require significantly fewer traces than CPA or DPA to recover the key. However, the downside of template attacks is the long profiling phase, during which the power consumption of the target device is characterized. Additionally, template attacks lack scalability between different devices because the profiling is valid only for a specific version of the device being profiled.

Recently, machine learning (ML) methods have been investigated as means to mount side-channel attacks. However, at the moment of writing, the more traditional CPA and DPA seem to be several orders of magnitude more efficient compared to attacks based on machine learning techniques. Machine learning has been successful at reducing the noise level of hiding countermeasures. As such, currently ML is primarily used as a preprocessing step on the collected traces before the attacker applies classical attack methods.

Higher-order attacks

Classical DPA and CPA presented previously exploited averages of power traces to reduce the measurement noise and extract the key. Hence, early countermeasures focused on removing the dependence of the mean power consumption on the key, reducing the effectiveness of the attacks relying on that dependence. However, a generalized type of attack, leveraging second-order statistical moments of the power traces, variance, to uncover the leakage was soon discovered [54, 46]. In a d-th order attack, d samples from the same power trace are used in the attempt to correlate their combination with the key. We say that a d-th order attack is univariate if the d-order statistic of the sample is used or the sample value is raised to the d-th power. If d different samples are combined, e.g., via product combining, such an attack is considered multivariate.

Higher-order attacks are a powerful tool at the attacker's disposal. Theoretically, higher-order attacks can be used to thwart any countermeasure. However, due to nonlinear combination of the samples, the physical noise of the device, and the quantization noise of the measurement probe, the noise level is significantly amplified, effectively hiding the correlation leakage from the attacker.

Leakage detection

While DPA and CPA pose a serious tool to the attacker, they can be difficult to use from the countermeasure design standpoint. Namely, different leakage models should be used to ensure the design is not vulnerable to any of them. Moreover, multiple intermediate variables also need to be investigated, as the exploitable leakage might only be observable in a few of them if it exists. Thus, a more generic method is preferred in order to discover possible exploitable leakage points. Test Vector Leakage Assessment (TVLA) [25] is a methodology based on the statistical method of evaluating the probability that two sets are distinguishable from one another based on the student's distribution. Given two sets $\mathcal{S}_0$ and $\mathcal{S}_1$, with their means $\mu(\mathcal{S}_0)$ $\mu(\mathcal{S}_1)$, variances $\sigma(\mathcal{S}_0)$ and $\sigma(\mathcal{S}_1)$, and cardinalities $|\mathcal{S}_0|$ and $|\mathcal{S}_1|$, the *t-values* are computed as

$$t = \frac{\mu(\mathcal{S}_1) - \mu(\mathcal{S}_0)}{\sqrt{\frac{\sigma^2(\mathcal{S}_0)}{|\mathcal{S}_0|} + \frac{\sigma^2(\mathcal{S}_1)}{|\mathcal{S}_1|}}}.$$

The t-statistic of two sets $\mathcal{S}_0$ and $\mathcal{S}_1$ can then be used to quantify the probability that these two sets come from populations with different means. Namely, larger t values mean a larger probability that the populations for $\mathcal{S}_0$ and $\mathcal{S}_1$ can be distinguished. According to the TVLA methodology [25] the established threshold of $t = 4.5$ is sufficient to distinguish between two sets, as the associated probability that these two sets come from populations with different means is $p = 0.99999$.

In TVLA, one set of power traces, $\mathcal{S}_0$, is chosen such that all traces belonging to that set have fixed inputs to the implementation under test. The second set of power traces, $\mathcal{S}_1$, contain power traces of execution where the inputs were taken randomly.

The collection of the traces for sets $\mathcal{S}_0$ and $\mathcal{S}_1$ should ideally be interleaved to minimize the effect of external factors during measurement, such as temperature variations. If instead of inputs, a particular intermediate value in the algorithm is chosen to differentiate $\mathcal{S}_0$ and $\mathcal{S}_1$, such as an S-Box input or output, the t-test becomes *specific*. Otherwise, the test is *non-specific*.

Similarly, as with higher-order CPA and DPA attacks, TVLA was extended to higher-orders by using higher order moments instead of the means of two sets [76].

2.5 Side-Channel Countermeasures

With the discovery of side-channel attacks, the need for protection against these attacks became apparent. Most countermeasures rely on increasing the noise level to the attacker by increasing the noise level of the power traces, making the successful attack more difficult to realize. Some countermeasures try to limit the number of traces that can be collected in a specific time frame, usually a couple of seconds, slowing down the trace collection needed in the first stage of the attack.

There are several options available to the designer to increase the noise level or reduce the leakage that occurs. Noise addition can be done at the physical level, where the process technology used reduces as much as possible the amount of leakage. The best known side-channel resistant process technology is Wave Dynamic Differential Logic (WDDL) [80] which builds upon the standard CMOS technology. In Table 2.1 we can see the Hamming weight of the 0 and 1 encoding is the same, which as discussed helps to reduce the observable leakage. Complementary logic helps to reduce the leakage because in the Hamming weight leakage model WDDL does not leak due to its outputs always being differential. Additionally, WDDL uses precharge/evaluation operation phases to reduce the amount of switching during the computation, thus preventing glitches that are increasing the leakage levels of the circuit.

Table 2.1: WDDL dual outputs with precharge logic.

x	$\overline{x}$	Encoding	$wt(x\|\overline{x})$
0	0	Precharge	0
0	1	0	1
1	0	1	1
1	1	Error	2

While the physical layer countermeasures significantly reduce the amount of leakage observable to the attacker, it was shown that side-channel attacks in which a sufficient number of traces were acquired were still successful at recovering the key.

Noise addition can also be achieved at the algorithmic level by modifying the cryptographic operation's algorithm so that the functionality remains unchanged, with the execution generating significantly more noise. Algorithmic countermeasures can be separates into two groups, *hiding* and *masking*.

Hiding countermeasures strive to prevent the attacker from pinpointing the time sample at which the leakage occurs. The simplest way of hiding is to introduce dummy operations during the execution of the algorithm at randomly chosen clock cycles for each new execution. Moreover, even the entire algorithm can run with dummy input data multiple times, with the actual leaky execution being hidden among them. Depending on the algorithm, the internal operations could be reordered for each execution. In AES, the S-Box operation order can be shuffled for each round, for example. Most dummy-based hiding countermeasures add multiple clock cycles to the execution. Additionally, dummies must not be distinguishable. Namely, if the power characteristic of dummy cycles differs from the cycles when intermediate values are processed, they could be filtered out. As mentioned in Section 2.4, machine learning techniques have been successfully used to filter out dummy cycles from the traces, nullifying any perceived protection.

Masking countermeasure is an algorithmic approach to reduce noise in an implementation: one separates each intermediate variable into several parts, *shares*, which when all combined together reveal the intermediate value. There are several ways to achieve masking operation, depending on how the shares are combined. In Table 2.2 the difference between two share Boolean masking, arithmetic masking over integers with two shares, and multiplicative masking using finite field multiplication to generate a mask is shown. The choice of the type of masking again depends on the algorithm to which the masking is applied. There are even techniques to translate masking in one domain

Table 2.2: Different types of masking.

Masking type	Unmasking operation
Boolean	$x = x_0 \oplus x_1$
Multiplicative	$x = x_0 x_1$
Arithmetic	$x = x_0 + x_1$

Table 2.3: Hamming weight of two shared Boolean variable.

x	x_0	x_1	$wt(\boldsymbol{x}) = wt(x_0) + wt(x_1)$	Mean$(wt(\boldsymbol{x}))$	Var$(wt(\boldsymbol{x}))$
0	0	0	0	1	1
	1	1	2		
1	0	1	1	1	0
	1	0	1		

into masking of another domain. Masked HMAC-SHA-256 implementations often use both Boolean and arithmetic masking to implement different parts of the protected round function while performing Boolean to arithmetic and arithmetic to Boolean conversions when masks of one domain need to be used in the other. In what follows, we will focus exclusively on Boolean masking, which is most widely used in side-channel protected implementations of symmetric key cryptographic primitives. Table 2.3 shows again the average Hamming weight for a 1-bit value $\boldsymbol{x}$ shared with two shares. While the average Hamming weight is equal for both 0 and 1 values, the variance, or more general the second statistical moment, differs in the two cases.

Boolean masking is linear for any function f that is linear, i.e., if f is applied independently to each share, Boolean addition of the outputs would be the value of $f(x)$. Thus, linear operations are simple to mask using Boolean masking. However, masking of a Boolean multiplication (2-input AND), or more generally masking of nonlinear Boolean functions, is not as trivial. One of the first examples of an SCA-protected 2-input AND gate in the literature came from Trichina [81] whose proposed solution is depicted in Figure 2.3. A more generic construction for SCA protected AND gate of any order using Boolean masking is the ISW scheme [45]; it is summarized in Algorithm 2. Both Trichina AND gate and the ISW scheme require the internal operations to be evaluated in specific order to guarantee security. For Trichina AND gate XORs in Figure 2.3 should be evaluated in top to bottom fashion in the diagram. For the ISW scheme it is essential that the $r_{j,i}$ calculation is achieved by computing $r_{i,j} \oplus a_i b_j$ before the final XOR with $a_j b_i$. These requirements on the order of evaluations are not easily met during the design in CMOS technology. Namely, CMOS

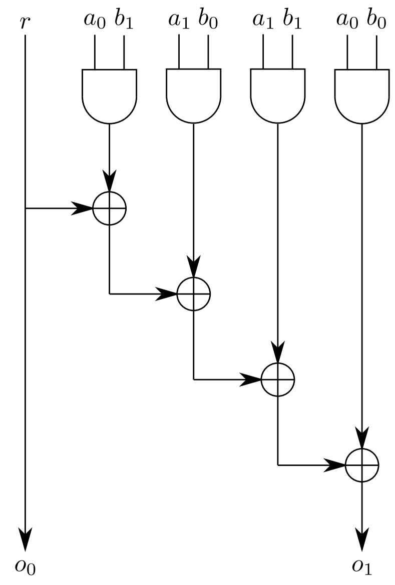


Figure 2.3: Trichina masked AND gate realization.

circuits exhibit the phenomena of glitches, where during one clock cycle, a gate can have its output change value multiple times due to different timings of the input signals. Glitches are an undesired effect in a digital circuit, as from the traditional design viewpoint they increase the power consumption of the circuit. From the security standpoint, they pose an additional problem, as the power consumption increase when glitches occur could be exploited in a side-channel attack. We will assume the most extreme potential gain of glitches to the attacker. Namely, for each output of a gate, we assume that all inputs are also known, and any intermediate Boolean function of the inputs that is also featured in the final output of the gate. This is also known as the glitch extended probing model [36].

2.6 Threshold Implementations

Threshold implementation emerged as another masking technique, particularly suitable to hardware implementations due to its resilience to glitches. The main idea behind TI comes from multi-party computation in which each player only computes on a fraction of the masked input, disallowing him from learning the

Algorithm 2: ISW AND circuit implementation.

```
Input: AND inputs $\boldsymbol{a} = a_0 \oplus \ldots \oplus a_{n-1}$ and $\boldsymbol{b} = a_0 \oplus \ldots \oplus a_{n-1}$ shared
       with $n$ shares.
Result: SCA protected value $\boldsymbol{c}$ equal to $\boldsymbol{c} = \boldsymbol{ab}$.
for $i = 0$ to $n - 1$ do
    for $j = i + 1$ to $n - 1$ do
        $r_{i,j} := Random()$
        $r_{j,i} := (r_{i,j} \oplus a_i b_j) \oplus a_j b_i$
for $i = 0$ to $n - 1$ do
    $c_i = a_i b_i$
    for $j = 0$ to $n - 1$ do
        if $j \neq i$ then
            $c_i = c_i \oplus r_{i,j}$
return $\boldsymbol{c} = \{c_0, c_1, \ldots, c_{n-1}\}$
```

unmasked input. In addition to glitch resilience, another appealing notion of TI is its focus on the protection of any Boolean function with a given algebraic degree, rather than protection of an AND gate. TI can also be used to protect against higher-order attacks; in this thesis we will refer to TI designed to protect against the d-th order attack as d-th order TI. We will represent the TI of a function f using n output shares $\boldsymbol{f} = (f_0, f_1, \ldots f_{n-1})$, and we will interchangeably use terms sharing or shared function to refer to $\boldsymbol{f}$. Each of the components of $\boldsymbol{f}$, $f_0, f_1, \ldots f_{n-1}$ is referred to as output component function, or output share. Unmasking or unsharing retrieves the unprotected value x from its masked representation $\boldsymbol{x}$ represented by the following definition:

Definition 2.6.1. *Given a masked representation bmx shared with n shares $\boldsymbol{x} = (x_0, x_1, \ldots x_{n-1})$, operation unmask$(\boldsymbol{x})$ computes the unprotected value $x = unmask(\boldsymbol{x}) = x_0 \oplus x_1 \oplus \ldots \oplus x_{n-1}$.*

Threshold implementations comply to the three properties

1. *Correctness*
2. *Non-completeness*
3. *Uniformity*

These properties are explained in what follows.

2.6.1 Correctness

The correctness property is necessary in any masking technique as the unmasking of the output $\boldsymbol{f}(\boldsymbol{x})$ should always produce the value equal to $f(x)$. Definition 2.6.2 describes correctness in a more formal manner.

Definition 2.6.2. *Given a Boolean function $f(x)$; $\boldsymbol{f}(\boldsymbol{x})$ is a correct sharing of the function $f(x) \iff (\forall \boldsymbol{x})(unmask(\boldsymbol{x}) = x \implies unmask(\boldsymbol{f}(\boldsymbol{x})) = f(x))$.*

2.6.2 Non-completeness

The non-completeness property of TI imposes restrictions on the component functions of a sharing $\boldsymbol{f}$. Namely, in order to achieve d-th order non-completeness, any combination of up to d component functions should be independent of at least one input share. It was shown in the glitch extended probing model that non-completeness has to be achieved in ordered to prevent exploitable side-channel leakage [33]. Thus, any non-complete circuit retains its side-channel security even in the presence of glitches. The non-completeness property gives a lower bound for the number of input shares required to secure a given function. Two strategies in TI accomplish non-completeness, differentiated by the minimal number of input shares required for a d-th order side-channel secure implementation of a function with a given algebraic degree.

Traditional TI [64, 7] mandates the use of at least $td+1$ input shares ($td+1$ TI) in order to secure a function with algebraic degree t against d-th order attacks. Non-completeness is achieved by ensuring that for any d-th order TI $\boldsymbol{f} = (f_0, f_1, \dots, f_{n-1})$ any combination of d component functions should be independent of at least one of the input shares completely, i.e., one input share should not be present in any of the input variables featured in d component functions. Figure 2.5 demonstrates the construction of a first-order $td+1$ TI of any quadratic function using three input and three output shares, while Figure 2.6 shows a first-order $td+1$ TI of an AND circuit, also using three input and three output shares. The number of output shares if $td+1$ input shares are used is shown to be $\binom{td+1}{t}$ [7]. It should be noted that while $td+1$ is the minimal number of input shares, correct and non-complete TI circuits can also have more than $td+1$ input shares if that is for any reason beneficial to the designer. In Chapter 3 we will use second-order $td+1$ TI with 8 input shares instead of 7 to reduce the number of output shares.

Consolidated Masking Scheme (CMS) [72] uses $d+1$ input shares ($d+1$ TI), building upon ISW and TI, while further demonstrating how to ensure glitch resistance with fewer input shares available. The biggest trade-off with $d+1$ shares is the increased number of output shares, which is always at least

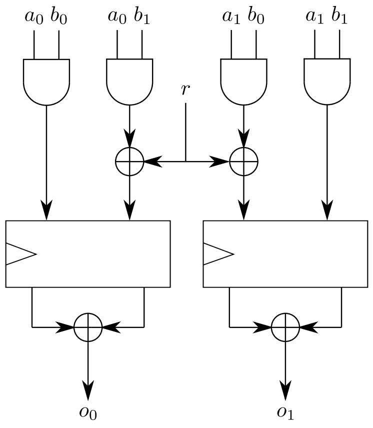


Figure 2.4: First-order $d + 1$ TI of AND circuit including remasking, synchronization and compression phase.

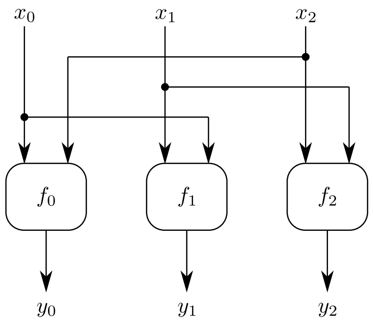


Figure 2.5: First-order $td + 1$ TI of a generic quadratic circuit.

$(d + 1)^t$. A first-order $d + 1$ TI AND implementation is given in Figure 2.4, showcasing the Nonlinear-register-compression layer structure (NLRC). While using $d + 1$ TI the relation between the input shares needs to obey a stronger requirement compared to $td + 1$ TI, namely shared input variables need to be *independent* [72]. If shared input variables are not independent, independence can be achieved by remasking some of the inputs or by using a technique proposed by Gross et al. [41].

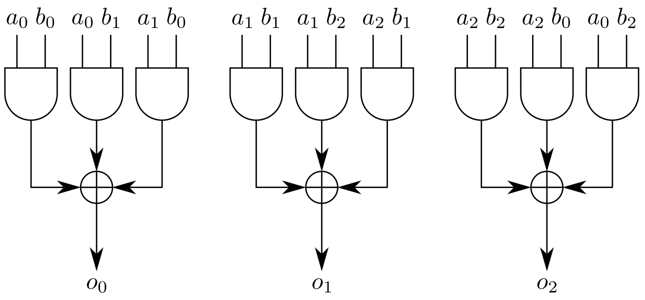

$a_0\,b_0$ $a_0\,b_1$ $a_1\,b_0$
$a_1\,b_1$ $a_1\,b_2$ $a_2\,b_1$
$a_2\,b_2$ $a_2\,b_0$ $a_0\,b_2$
o_0
o_1
o_2

Figure 2.6: First-order three share $td + 1$ TI of AND circuit.

2.6.3 Uniformity

While the masking of nonlinear operations constitutes the central part of any masking method, another critical element of a secure side-channel design is the procedure of cascading two or more nonlinear blocks. More precisely, how can the output of one nonlinear layer be used as input to another nonlinear layer without the loss of side-channel resistance? As most symmetric cryptographic primitives are realized by repeating a particular round function, the secure composition is a crucial part of any masking technique. In TI, the uniformity property is used to ensure the composability of subsequent nonlinear stages by defining conditions under which the output of one nonlinear stage can be used as input to the next stage. The classical definition of uniformity is given in Definition 2.6.3. An interesting implication of Definition 2.6.3 is that the sharing of a permutation with the same number of input and output shares is uniform if and only if the output sharing is itself a permutation.

Definition 2.6.3. *An output sharing $\boldsymbol{f}(\boldsymbol{x})$ of a Boolean function $f(x)$ is uniform if and only if*

$$(\forall x)(f(x) = y \implies (\forall \boldsymbol{x}, \boldsymbol{y})(unmask(\boldsymbol{x}) = x \wedge unmask(\boldsymbol{y}) = y \\ \implies \Pr(\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{y}) = const)).$$

Alternatively, for each x, its masking $\boldsymbol{x}$ and $y = f(x)$, all possible different outputs $\boldsymbol{y}$ of the $\boldsymbol{f}(\boldsymbol{x})$ must be equiprobable.

The uniformity property is mostly investigated in first-order $td + 1$ TI circuits, as it was shown that it does not guarantee resilience against multivariate attacks [72]. A brute force uniformity check is computationally quite expensive, with the exponential complexity on the product of number input bits and number of input shares, although specific optimizations for quadratic Boolean functions with 3 shares exists [5].

Additionally, the uniformity property is not preserved among the members on an affine equivalence class, but Bilgin et al. [9] showed how to create a uniform sharing for all members of an affine class if one of the members has uniform sharing:

Lemma 2.6.1. *If there exists a uniform sharing for any member of affine equivalence class S, a uniform sharing can be constructed for all members of that class.*

Proof. if S has a uniform sharing $\boldsymbol{S}$, and $S_1 = A \circ S \circ B$ where A and B are affine transforms, then the sharing $\boldsymbol{S_1} = \boldsymbol{A} \circ \boldsymbol{S} \circ \boldsymbol{B}$ is also uniform because $\boldsymbol{A}$, $\boldsymbol{B}$ and $\boldsymbol{S}$ are all uniform, hence their composition is also uniform. $\square$

Some functions have been shown not to have a uniform TI sharing, such as a simple AND circuit if 3 input shares are used. Uniformity of a quadratic permutation has been extensively investigated [9, 16], and a uniform sharing has been found for some of them with 3 shares, and for all of them with 4 shares. Since the number of invertible affine transformations of n bits is

$$2^n \times \prod_{i=0}^{n-1} (2^n - 2^i),$$

brute force search across all members of a quadratic permutation class is practically impossible for permutations of 4 or more bits. Furthermore, it was shown that for an n-bit quadratic permutation there exist $2^{9(n+\binom{n}{2})}$ different TI sharings with three shares, making an exhaustive search for uniform sharing impossible even for 3-bit permutations as the total number of possible sharings with three shares is 2^{54}. Thus, finding a uniform sharing of a given vectorial Boolean function or proving that such a sharing does not exist in a computationally efficient manner is an open problem.

There are several options to compensate for the lack of uniformity. The straightforward method is to refresh the output shares using a mask refreshing technique. This method requires a Pseudo Random Number Generator (PRNG) to generate random masks during algorithm execution. Moreover, mask refreshing significantly increases the power consumption of the circuit (cf. Chapters 4 and 5). Another option is to increase the number of input shares. Using the notation findings by Bilgin et al. [9], the quadratic 4-bit permutation Q_{300} denoted with its lookup table $[0, 1, 2, 3, 4, 5, 8, 9, 6, 7, 12, 13, 14, 15, 10, 11]$ does not have any 3-share uniform sharings, but a 4-share uniform sharing of Q_{300} is known. Finally, a change of guards method presented by Daemen [27] significantly reduces the amount of added randomness to produce a uniform sharing of the permutation-based nonlinear layer in the round function. Namely, the input shares of adjacent inputs to the S-Box are reused as output masks. The state needs to be extended initially with random masks used for the first S-Box evaluation, but further rounds do not need fresh randomness as the sharing of the last S-Box input is used to refresh masks of the first S-Box in all the following rounds.

From Definition 2.6.3 we can deduce that the $d+1$ TIs of nonlinear functions cannot be uniform, as the number of output shares would always be higher than the number of input shares, thus different shared outputs cannot be equiprobable. Hence, the output shares need to be refreshed in order to achieve uniform output distribution for $d+1$ TI.

2.6.4 Composability

In addition to uniformity, cascading of nonlinear functions in TI requires register stages to be used after each nonlinear block. Registers serve to stop glitch propagation from one block to another, which could lead to leakage, even if both blocks are fully side-channel resistant.

When sharing a nonlinear function, the number of output shares is typically larger than the number of input shares. The discrepancy between the number of input and output shares is likely to occur when applying $td+1$ TI, and it always occurs when applying $d+1$ TI. In order to minimize the number of output shares, we need to refresh and recombine (compress) some shares by adding several of them together. To prevent glitches from revealing unmasked values, decreasing the number of shares can only be done after storing these output shares into a register. The output shares that are recombined still need to be carefully chosen such that they do not reveal any unmasked value, e.g., by combining output shares remasked with the same random mask.

In round-based designs, in which the output of a TI circuit in one cycle is fed back to the same circuit in the following cycle, care needs to be taken in order to prevent transient leakage. Let us assume TI of a function $y = f(x)$ implemented with 3 shares as shown on the right side of Figure 2.7:

$$y_0 = f_0(x_1, x_2)$$

$$y_1 = f_1(x_2, x_0)$$

$$y_2 = f_2(x_0, x_1)\,.$$

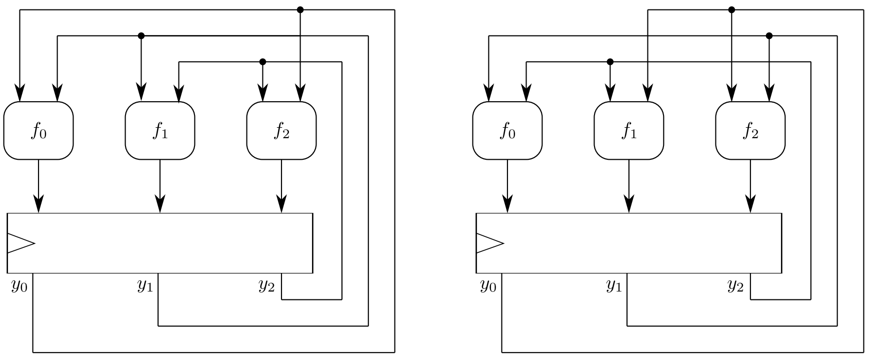


Figure 2.7: Safe (left) and potentially leaky (right) option on how to implement iterating $td + 1$ circuit.

Let $\boldsymbol{z}$ be the output of shared function $\boldsymbol{f}$ in the subsequent cycle, i.e., $z = f(y)$:

$$z_0 = f_0(y_1, y_2)$$

$$z_1 = f_1(y_2, y_0)$$

$$z_2 = f_2(y_0, y_1)\,.$$

The Hamming distance of the change in the output register of share 0 is equal to:

$$\mathrm{HD}(z_0, y_0) = z_0 \oplus y_0 = f_0(y_1, y_2) \oplus y_0\,.$$

From the previous equation we can see that Hamming distance of the subsequent register values contains the information from all three shares, which can be detectable in the power signature. We can notice the same dependency on all three shares in other two output registers of shares 1 and 2. Hence, it is recommended that the input share i is featured in the computation of the output share i to mitigate the possibility of the Hamming distance leakage. If the sharing is done in the following manner (left side of Figure 2.7):

$$y_0 = f_0(x_0, x_1)$$

$$y_1 = f_1(x_1, x_2)$$

$$y_2 = f_2(x_2, x_0)\,,$$

the Hamming distance of the subsequent outputs of share 0 is equal to:

$$\mathrm{HD}(z_0, y_0) = z_0 \oplus y_0 = f_0(y_0, y_1) \oplus y_0\,.$$

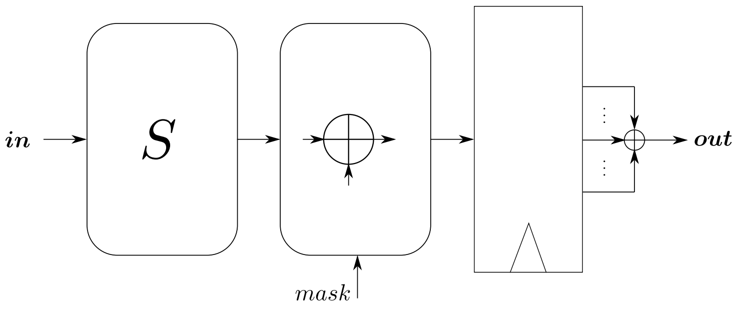


Figure 2.8: Direct side-channel protected S-Box implementation.

Thus, in this construction the Hamming distance leakage of all three shares is reduced.

2.7 Low Latency SCA Design

Similar to other side-channel countermeasures, the area overhead of applying TI increases polynomially with the security order and exponentially with the algebraic degree of the function we are trying to protect. To keep the large overheads caused by exponential dependency under control, designers often use decomposition of the higher degree functions into several lower-degree functions. This approach has originally been demonstrated by Poschmann et al. [68] whose implementation of a TI-protected PRESENT block cipher [10] included decomposing its cubic S-Box into two simpler quadratic S-Boxes. Finally, decomposition of the cubic 4-bit S-Boxes into chains of smaller quadratic S-Boxes was given in [9], which eventually enables compact, side-channel secure implementations of all 4-bit S-Boxes. Many side-channel protected implementations of AES S-Box use decomposition to reduce the area overhead, most of which is derived from the decomposition presented by Canright [20] in which the field inversion in $\mathbb{F}_{2^8}$ is implemented with field inversion in $\mathbb{F}_{2^4}$ which is, in turn, implemented using field inversion in $\mathbb{F}_{2^2}$. A generic side-channel design is showed in Figure 2.9 in which output computes function $out = S_m \circ \ldots \circ S_2 \circ S_1(in)$. Conversely, a monolithic approach without decomposition is featured in Figure 2.8. A more intermediate approach would reduce in the number of nonlinear stages during decomposition by merging some stages.

Although a decomposition of nonlinear functions into several simpler functions

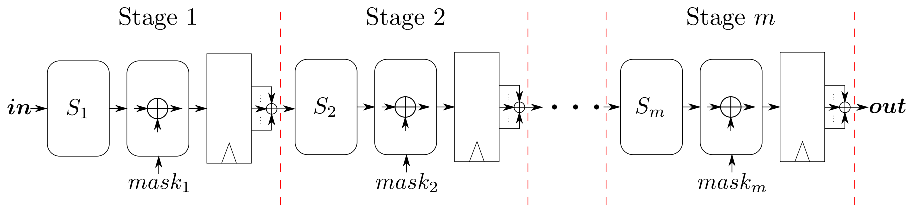


Figure 2.9: Decomposition based side-channel protected S-Box implementation.

of smaller algebraic degree is the proper approach to use for area reduction of the TI-protected implementations, its side-effect is the increased cycle latency of the S-Box evaluation, and hence the entire implementation. Recall that TI requires registers to be placed between the nonlinear operations in order to prevent glitch propagation, increasing the latency. Additionally, realizing side-channel implementations using decomposition is easier from the designer's perspective, as the number of output shares that need to be coded is smaller with less complex ANFs.

Therefore, most of the effort the scientific community has initially been spent on designing secure implementations with minimal area overhead. Another critical metric is the amount of randomness used in protected implementations. While both of these metrics are important, the performance and energy consumption of secure implementations have been unjustly treated as less significant. It has been widely accepted that performance is the metric to sacrifice in order to achieve the lowest possible gate count. Contrary to this view, most practical applications nowadays require high-speed execution, and it is often latency of the actual implementation that matters rather than throughput. Energy consumption is another equally important metric, and, unlike power consumption, it cannot be well controlled by keeping the area low while sacrificing performance. Optimizing for energy consumption is one of the most challenging optimization problems in (secure) circuit design since the perfect balance between circuit power consumption and execution speed needs to be hit.

The latency is directly proportional to the number of clock cycles a certain operation takes to execute. Additionally, the absolute latency is inversely proportional to the clock frequency of the system. While the clock frequency is determined by taking into account multiple factors from the whole system, most important of which is the overall power/energy consumption, the number of clock cycles a certain algorithm takes to execute is under the full control of the designer. Especially when considering embedded devices, the tendency is to keep the clock frequency as low as possible while still meeting the performance requirements. That is why minimizing the number of clock cycles of a certain algorithm is the most important strategy for minimizing the overall latency of that algorithm.

As the field of side-channel protection and masking matured, researchers turned their focus more and more on low latency side-channel designs, despite the difficulties during the design process. Moradi and Schneider [59] explored the extreme case of single-cycle side-channel secure implementations of PRINCE and Midori. The AES S-Box design by Ueno et al. [82] does not decompose inversion in field $\mathbb{F}_{2^4}$ to reduce S-Box latency by one clock cycle. Gross et al. [38] presented the first single-cycle implementation of the AES S-Box with 256 output shares. The side-channel resistant KECCAK [4] design by Arribas et al. [1] implements two rounds of KECCAK using second-order $td+1$ TI without a register stage in between to achieve a first-order secure implementation. The AES design by Sasdrich et al. [75] combines $td+1$ TI and precharge logic to remove the need for registers between nonlinear blocks since precharge logic designs inherently do not suffer from glitches.

2.8 Mask Refreshing Techniques

Section 2.6 states how mask refreshing is often needed at the output of a TI circuit to ensure its composability. Remasking is realized by XORing the output of a TI circuit with a random sharing of the zero value, which can be achieved in several ways. Here we reiterate the most common mask refreshing techniques.

Given n shares $x_1, \dots, x_n$ of a masked value $\boldsymbol{x}$ a (first- and second-order) refreshing can be realized by mapping $(x_1, \dots, x_n)$ to $(y_1, \dots, y_n)$ using n random values $r_1, \dots, r_n$ as follows:

$$y_1 = x_1 \oplus r_1 \oplus r_n \qquad y_i = x_i \oplus r_{i-1} \oplus r_i, \qquad i \in \{2, \dots, n\}\,. \tag{2.7}$$

This refreshing scheme is called *ring remasking*. A simpler refreshing using $n-1$ random values exists especially for the first-order secure implementations, and is achieved in the following way:

$$y_i = x_i \oplus r_i, \qquad i \in \{1, \dots, n-1\}, \qquad y_n = x_n \oplus r_1 \oplus \dots \oplus r_{n-1}\,. \tag{2.8}$$

An improvement regarding the number of random bits used when multiplication gate is shared has been achieved in Domain Oriented Masking (DOM) [41] in which the amount of randomness required is halved compared to the ring remasking techniques [72]; it is shown here:

$$y_i = x_i \oplus r_i, i \in \{1, \dots \lfloor n/2 \rfloor\}, \qquad y_i = x_i \oplus r_{n-i}, i \in \{\lceil n/2 \rceil + 1, \dots, n\}\,. \tag{2.9}$$

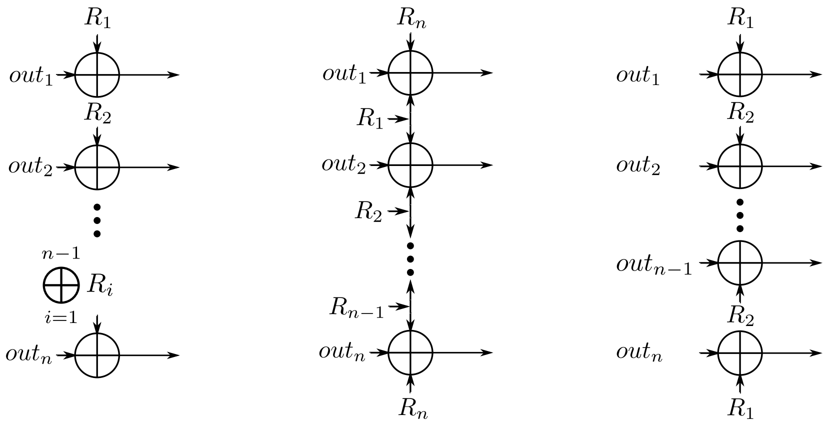


Figure 2.10: Different remasking options. Left (simple remasking, first-order only), middle (ring remasking), right (DOM remasking) remasking.

Gross et al. [40] showed that the amount of randomness for sharing a multiplication gate can be further reduced to one-third, although this comes at a significant performance cost. Throughout the rest of the thesis, we will interchangeably use the terms mask refreshing, resharing and remasking. Diagrams of three different remasking options are given in Figure 2.10.

2.9 Pseudo Random Number Generators

Sections 2.6 and 2.8 emphasized the need to inject randomness into masked circuits to ensure side-channel resilience and composability. The number of random bits needed is often quite high, mandating a high throughput PRNG. However, the PRNG used is not included in many of the published designs, creating confusion about the actual cost of those designs. PRNGs impact area, power and side-channel resilience, so their design must be carefully considered.

A PRNG is a circuit that provides a deterministic, yet seemingly random sequence of bits of length l starting from some initial seed value of length s, with $l \gg s$. Since a PRNG is deterministic, knowledge of the initial seed is sufficient to recreate the sequence.

The most well known PRNG design is the *Mersenne Twister* by Matsumoto and Nishimura [53]. It has a sequence period of $2^{19937} - 1$ and can provide 32-bits of output per update. The Mersenne twister is the default PRNG in many operating systems. In hardware implementations the Mersenne twister is seldom used as the large internal state translates to a large area, while the throughput is limited to 32 bits per cycle.

A *linear-feedback shift register* (LFSR) is a cheap way one can generate randomness, and it is used as such in some publications, e.g., by De Meyer et al. [32] and Shahmirzadi et al. [79]. An LFSR is implemented by using a shift register whose feedback is a linear Boolean function of the register state, i.e., it can be realized using only flip-flops and XOR cells. The downside of using an LFSR as a PRNG is that the sequence can be completely recreated if the attacker knows m output bits of the LFSR, where m is the order of the LFSR.

Stream ciphers such as Trivium by Cannière and Preneel [30] can also be used as a PRNG for side-channel resistant implementations. They are more robust than LFSRs as knowing parts or entirety of the output sequence should not lead to recovery of the internal state, i.e., the future output cannot be predicted better than a uniform guess. Stream ciphers typically mandate more area, and in many cases an initialization sequence before they can be used.

Most reported PRNGs in side-channel designs are not themselves side-channel protected. The reason is that side-channel attacks that exploit the PRNG are not considered to be cost effective. The attacker would first need to extract the PRNG output, and only then use that information to break the underlying masking scheme. Since the environment is presumed to be noisy, it is expected that the attacker cannot accurately enough recover the PRNG output sequence to mount a successful attack. However, these assumptions might not always hold and a more detailed study is needed on the impact of the PRNG design on the side-channel resilience of masked implementations.

Chapter 3

Optimal Sharing of Any Boolean Function

“No tool, no craft.”

Serbian proverb

The work presented in this chapter is based on several publications [14, 79, 15]. I was the main author and contributor for the work presented in [14, 15], and I provided an optimal sharing for the work presented by Shahmirzadi et al. [79].

Masking nonlinear Boolean functions with n variables and algebraic degree t, $t \geq 3$ poses a challenging task to the designer. While all the constraints required for masking quadratic functions are also present when protecting higher degree functions, several new difficulties arise. The number of shares grows exponentially in $d+1$ TI and polynomially in $td+1$ TI with the algebraic degree, increasing the area and number of random bits during remasking. Besides, the established theory only provides a trivial lower bound on the number of output shares for $d+1$ TI, and a rather inefficient construction for $td+1$ TI [7]. Thus, the output sharing might become unnecessarily large, and minimizing the number of output shares is essential since it directly impacts area and randomness needed in a hardware implementation.

In this chapter, we present several heuristics that can be used to minimize the number of output shares for $d+1$ and $td+1$ TI, together with an optimal method for $d+1$ TI when the algebraic degree of an n-bit function is $t = n-1$. We also connect the problem of finding the minimal output sharing for $d+1$ TI with the known NP-hard discrete optimization problem of set covering. For $td+1$ sharing, finding a minimal output sharing can be reduced to another NP-hard problem, a variant of the vertex cover problem. Thus, these problems are difficult but can be made practical with techniques described in this chapter, in many cases even solved to optimality. Moreover, even when the optimal solution cannot be identified, the following heuristics still provide output sharing with a number of output shares that is close to minimal. Efficient sharing of arbitrary Boolean functions is crucial for low latency applications, hence methods presented in this chapter should be utilized in side-channel low latency hardware circuits.

Intuitively, sharing quadratic Boolean functions or functions with only a small number of high degree terms in their ANF is quickly done by hand, for both flavors of TI, $d+1$ and $td+1$. Nevertheless, as the ANF becomes more complex, the proper way to create a minimal sharing becomes more elusive. Consequently, the effort needed to find minimal output sharing becomes increasingly difficult. Here we present several techniques that can help the designers to find the most efficient or close to optimal sharing solutions for both $td+1$ and $d+1$ sharing for any security order d.

3.1 Efficient First- and Second- Order $td+1$ Sharing

Each $td+1$ TI implementation contains two distinct phases:

a) The expansion phase in which the shared function f uses $s_{in} \geq td+1$ input shares and produces s_{out} output shares. The output share functions f_i are referred to as component functions.

b) The compression phase in which remasked s_{out} output shares stored in a register are combined again to s_{in} shares.

A register layer precedes the compression phase; for higher security order ($d > 2$), it needs to be followed by another register layer to ensure composability. If $s_{in} = s_{out}$, compression is omitted. Furthermore, if the TI sharing is uniform, the refreshing step can also be omitted.

A d-th order TI (more specifically, its *non-completeness* property) requires that any combination of up to d component functions f_i is missing at least one input share in each component function. The method presented in [7] demonstrates how to find a sharing with the minimum number of input shares, i.e., $s_{in} = td+1$, which results in $s_{out} = \binom{s_{in}}{t}$ output shares. However, this approach does not guarantee that s_{out} is indeed the theoretical minimum. Even more, there are examples which show that by increasing s_{in} it is possible to decrease s_{out}.

Throughout this thesis, we will use the number s_{out} of output shares as a figure of merit against which to optimize the implementation since the number of registers required to store the output shares and the number of random bits required for refreshing increases with the number of output shares.

As already discussed in Section 2.6, to comply with the $td + 1$ TI non-completeness output component functions can only contain a subset of input shares featured in their ANFs. Definition 3.1.2 introduces a notion of output sets that uniquely define all output shares by indicating which input shares can be featured in a given output, while Definition 3.1.1 introduces a notion of the shared monomial, i.e., an ANF term comprised of products of different input shares. Each unmasked ANF term of degree t produces s_{in}^t.

Definition 3.1.1. *A **Shared monomial** or **shared term** is a single term featured in the ANF of the output sharing, constituted by a product of one or more shares of input variables.*

Definition 3.1.2. *Consider a $td + 1$ TI sharing of $\boldsymbol{f}$ with s_{in} and s_{out} output shares. We can enumerate the input shares as elements of a set $\mathcal{I} = \{0, \dots s_{in} - 1\}$. With each output share f_o we can associate a set $\mathcal{O}$, which is a subset of I, containing indices of allowed input shares that can appear in the ANF of f_o. We refer to O as the **output set** of the o-th output share of $\boldsymbol{f}$. The set $\mathcal{S}$ containing all output sets of a sharing as elements is an **output sharing set**.*

Output sets do not impose any restriction for any particular input variable, but only to the indices of input shares that can be appear in the ANF of the output share. Table 3.1 shows an example of a second-order secure sharing of function $xy \oplus z$ with 6 input shares and 7 output shares. An illustration of this sharing is additionally represented by its output sets on the left. Note that the output sets dictate which indexes of variables are allowed in the corresponding output share. For example, for o_0 only input shares (i.e., indexes) 0, 1 and 2 are allowed. That requirement is indeed fulfilled by the formula describing o_0. Note that these sets do not uniquely define the sharing. We could move the term x_0y_1 from o_0 to o_4 or o_5, as $\{0, 1\}$ is a subset of both $\{0, 1, 4\}$ and $\{0, 1, 5\}$. Output sets would still be the same and the sharing would still be a correct second-order $td + 1$ sharing. More generally, a shared monomial can appear in an output share if and only if the set of input shares present in the shared monomial is a subset of the output set. It is up to the designer to choose the exact distribution of shared ANF monomials. As we will see in Chapter 5, the choice impacts the area and latency of the final circuit.

Table 3.1: Second-order $td+1$ TI sharing of $xy \oplus z$ using 6 input and 7 output shares.

Output set	Output share ANF
$\{0,1,2\}$	$o_0 = x_0y_1 \oplus x_1y_0 \oplus x_0y_2 \oplus x_1y_2 \oplus x_2y_1$
$\{0,3,4\}$	$o_1 = z_4 \oplus x_4y_4 \oplus x_0y_3 \oplus x_0y_4 \oplus x_3y_4 \oplus x_4y_3$
$\{1,3,5\}$	$o_2 = z_3 \oplus x_3y_3 \oplus x_1y_3 \oplus x_3y_1 \oplus x_1y_5 \oplus x_3y_5 \oplus x_5y_3$
$\{2,4,5\}$	$o_3 = z_5 \oplus x_5y_5 \oplus x_2y_4 \oplus x_4y_2 \oplus x_2y_5 \oplus x_5y_2 \oplus x_4y_5 \oplus x_5y_4$
$\{0,1,4\}$	$o_4 = z_0 \oplus x_0y_0 \oplus x_4y_0 \oplus x_1y_4 \oplus x_4y_1$
$\{0,1,5\}$	$o_5 = z_1 \oplus x_1y_1 \oplus x_0y_5 \oplus x_5y_0 \oplus x_5y_1$
$\{0,2,3\}$	$o_6 = z_2 \oplus x_2y_2 \oplus x_2y_0 \oplus x_3y_0 \oplus x_2y_3 \oplus x_3y_2$

An output sharing set fully determines compliance to the TI properties of correctness and non-completeness. First, we will introduce several definitions used in the rest of the section to ease the notation. Definition 3.1.3 just names any set of cardinality k as a k-set, which will be useful during the characterization of the $td+1$ sharing, as the construction method we propose uses output sets with the same cardinality to generate an output sharing. Definition 3.1.4 introduces a notion useful when checking the correctness property of $td+1$ TI sharings. Lemma 3.1.1 states the necessary and sufficient condition a $td+1$ sharing has to meet to fulfil the correctness property.

Definition 3.1.3. *A k-set is a set containing exactly k elements.*

Definition 3.1.4. ***The Correctness Generator Table*** $\mathrm{CGT}(s_{in}, t)$ *of a sharing with s_{in} input shares of a function of degree t is a set whose elements are $\binom{s_{in}}{t}$ t-sets representing all different combinations of input shares with t elements.*

Lemma 3.1.1. *An output sharing set $\mathcal{S}$, with s_{in} input shares of a function of degree t, is correct if and only if all sets from the correctness generator table $\mathcal{T} = \mathrm{CGT}(s_{in}, t)$ are subsets of at least one output set of $\mathcal{S}$.*

Proof. If $\mathcal{S}$ contains at least one output set for each set from $\mathcal{T}$, any shared monomial of degree t can appear in at least one of the output sets. Thus, any function of degree t can be correctly shared using output sharing set $\mathcal{S}$. If $\mathcal{S}$ is a correct sharing of any function with degree t, then all sets from $\mathcal{T}$ need to be a subset of at least one output set of $\mathcal{S}$. Otherwise, there would be a set $\mathcal{C} \in \mathcal{T}$ which is not a subset of any output set of $\mathcal{S}$. Hence, a shared monomial whose indices are from $\mathcal{C}$ could not belong to any output set of $\mathcal{S}$, which is impossible if the sharing is correct. $\square$

In Table 3.1, the unshared function is of degree 2, and we can verify that all of the $\binom{6}{2}$ 2-sets of the correctness generator table are contained in sharing output sets.

Non-completeness can be inferred from the output sharing set $\mathcal{S}$ in the following manner:

Lemma 3.1.2. *An output sharing set $\mathcal{S}$ with s_{in} input shares represents a d-th order non-complete sharing if and only if no union of d output sets is equal to the set $\mathcal{I} = \{0, \ldots, s_{in} - 1\}$.*

Proof. If there exists d output sets whose union is equal to the set $\mathcal{I}$ then d output component functions represented by these output sets would contain all input shares, which is contradicting with the $td+1$ non-completeness property as explained in Section 2.6 because these d output component functions are not independent of at least one input share. Conversely, if the sharing is non-complete, any d output sets represent d output component functions which are independent of at least one input share. Hence, union of any d output sets does not contain at least one value from $\{0, \ldots, s_{in} - 1\}$. $\square$

In Table 3.1 we can verify that the output is second order non-complete as for $d = 2$ no union of two output sets gives the set of all input shares $\{0, 1, 2, 3, 4, 5\}$.

While we have discussed *correctness* and *non-completeness* using the output sets, we did not look at the number of output sets themselves. As was already mentioned, the number of output sets directly determines the number of storage registers needed to prevent glitch propagation, as well as the amount of randomness required for mask refreshing. Since any subset of output shares that contains all possible t-sets also contains all possible sets of smaller length and smaller output sets do not contribute to the generation of a correct sharing of a degree t function, output sets of length smaller than t do not have to be considered.

Lemma 3.1.3. *The maximal cardinality of an output set from the d-th order non-complete output sharing $\mathcal{S}$ for functions of degree t with s_{in} input shares is $s_{in} - (t(d-1)+1)$.*

Proof. Let us assume otherwise, i.e., there is an output set $\mathcal{O}$ of size at least $s_{in} - t(d-1)$. Since the minimal cardinality of output sets is t, $t(d-1)$ input shares not in $\mathcal{O}$ can be separated into $d-1$ disjunct t-sets, which are covered by at most $(d-1)$ other output sets. This would mean that the non-completeness property would be violated since these $(d-1)$ output sets together with $\mathcal{O}$ combined comprise d output sets whose union is the set of all input shares. $\square$

Finally, the uniformity of a given sharing cannot be investigated using its output sets since uniformity is dependent on the exact ANF expressions of all output shares.

Algorithm 3 finds a $td + 1$ sharing by adding output sets to the solution until all the sets of the correctness generator table are subsets of at least one of output sets of the solution. According to Lemma 3.1.1, the constructed output sharing will fulfill the correctness property. For brevity, the procedures used in Algorithm 3 are listed below.

- kSets(k, s_{in}): Creates correctness generator table for degree k with s_{in} input shares, i.e., combinations without repetition of k elements from $\{0, \ldots, s_{in} - 1\}$.
- doesNotCover$(s_{in}, t, \mathcal{S})$: Indicates if the set $\mathcal{S}$ of output shares given ensures the correctness property in $td + 1$ TI with s_{in} input shares and algebraic degree t, i.e., all sets of CGT(s_{in}, t) are subsets of at least one output set of $\mathcal{S}$.
- removeCompletenessShares$(\mathcal{U}, \mathcal{S}, d)$: Takes as input a set of candidate sets $\mathcal{U}$ that are not chosen as output shares, and the set $\mathcal{S}$ representing a partially constructed output sharing. The procedure removes all output sets from $\mathcal{U}$ that would, if chosen to be part of output sharing, violate the non-completeness property.
- chooseGreedy$(\mathcal{U}, \mathcal{S}, \mathcal{T})$: Chooses the next output share in a greedy manner, given a partial sharing $\mathcal{S}$, the list of remaining shares $\mathcal{U}$, and a set of all not yet covered degree t sets from CGT. The next output set $\mathcal{C}$ is chosen as the output set from $\mathcal{U}$ which is a superset of the highest number of sets from $\mathcal{T}$. If there are multiple output sets that cover the same number of uncovered t-sets, the output set $\mathcal{C}$ is chosen uniformly at random among them. Finally, $\mathcal{T}$ is updated by removing all the t-sets that are subsets of $\mathcal{C}$.

```
Algorithm 3: Greedy Algorithm for efficient $td + 1$ sharing
Input: Number $s_{in}$ of input shares, security order $d$, algebraic degree $t$
Result: Found $td + 1$ sharing $\mathcal{S}$
$\mathcal{S} = \{\}$
$k = s_{in} - (t(d-1) + 1)$
$\mathcal{O} = kSets(k, s_{in})$
$\mathcal{T} = kSets(t, s_{in})$
while $doesNotCover(s_{in}, t, \mathcal{S})$ do
    $\mathcal{O} = removeCompletenessShares(\mathcal{O}, \mathcal{S})$
    $\mathcal{C} = chooseGreedy(\mathcal{O}, \mathcal{S}, \mathcal{T})$
    $\mathcal{S} = \mathcal{S} \cup \{\mathcal{C}\}$
return $\mathcal{S}$
```

Procedure chooseGreedy in Algorithm 3 involves randomly choosing a k-set $\mathcal{C}$ among all possible k-sets which are a superset of the maximal number of uncovered t-sets in $\mathcal{T}$. This non-deterministic behavior leads to different output sharings with potentially different cardinalities if we restart the algorithm multiple times. Hence, we restart the greedy algorithm 100 times and choose the output sharing $\mathcal{S}$ with the smallest cardinality among all executions. We have used 100 iterations to create the second-order $td + 1$ TI in Section 4.2.7, since using up to more iterations did not yield a sharing with fewer output shares.

An example single pass of the Algorithm 3 is given in Table 3.2, for $S_{in} = 6$, $d = 2$ and $t = 2$, with the set $\mathcal{U}$ containing all sets of size $k = s_{in} - (t(d-1) + 1) = 3$. In each step of the greedy algorithm, these sets are scored according to the procedure chooseGreedy. The chosen set to be added to the output sharing is marked in bold, and in light gray we highlight the sets that must be removed according to the removeCompletenessShares procedure, because if they remain in the next steps of the algorithm, they would violate non-completeness of the constructed sharing if chosen as part of the output sharing. The left column in each table is the score of a given k-set, or the number of t-sets it contains, that are not present in $\mathcal{S}$. The right column contains all remaining k-sets that do not violate non-completeness if added to $\mathcal{S}$. In the same column above the horizontal line is a partially constructed set $\mathcal{S}$ represented by k-sets that are added to it. As an example, in the fourth table, k-set $\{0, 1, 4\}$ has a score of 1 as only $\{1, 4\}$ is the new t-set it would add, given $\{0, 1\}$ and $\{0, 4\}$ are already subsets of output shares $\{0, 1, 2\}$ and $\{0, 3, 4\}$. On the other hand, k-set $\{2, 4, 5\}$ has a score of 3 since none of the t-sets $\{2, 4\}$, $\{2, 5\}$ and $\{4, 5\}$ are present in any of the output shares. For this particular order of the k-sets, we end up with an output sharing that contains 7 shares.

Table 3.2: Example execution of the greedy algorithm for the case $s_{in} = 6$, $d = 2$, $t = 2$. Each table shows a single step of the algorithm execution. Left column is the amount of t-sets not covered in $\mathcal{S}$ by the set on the right. Sets above the horizontal line are partially constructed sharing sets $\mathcal{S}$.

	{0, 1, 2}
3	**{0, 1, 2}**
3	{0, 1, 3}
3	{0, 1, 4}
3	{0, 1, 5}
3	{0, 2, 3}
3	{0, 2, 4}
3	{0, 2, 5}
3	{0, 3, 4}
3	{0, 3, 5}
3	{0, 4, 5}
3	{1, 2, 3}
3	{1, 2, 4}
3	{1, 2, 5}
3	{1, 3, 4}
3	{1, 3, 5}
3	{1, 4, 5}
3	{2, 3, 4}
3	{2, 3, 5}
3	{2, 4, 5}
3	{3, 4, 5}

	{0, 1, 2}
	{0, 3, 4}
2	{0, 1, 3}
2	{0, 1, 4}
2	{0, 1, 5}
2	{0, 2, 3}
2	{0, 2, 4}
2	{0, 2, 5}
3	**{0, 3, 4}**
3	{0, 3, 5}
3	{0, 4, 5}
2	{1, 2, 3}
2	{1, 2, 4}
2	{1, 2, 5}
3	{1, 3, 4}
3	{1, 3, 5}
3	{1, 4, 5}
3	{2, 3, 4}
3	{2, 3, 5}
3	{2, 4, 5}

	{0, 1, 2}
	{0, 3, 4}
	{1, 3, 5}
1	{0, 1, 3}
1	{0, 1, 4}
2	{0, 1, 5}
1	{0, 2, 3}
1	{0, 2, 4}
2	{0, 2, 5}
2	{0, 3, 5}
2	{0, 4, 5}
2	{1, 2, 3}
2	{1, 2, 4}
2	{1, 3, 4}
3	**{1, 3, 5}**
3	{1, 4, 5}
2	{2, 3, 4}
3	{2, 3, 5}
3	{2, 4, 5}

	{0, 1, 2}
	{0, 3, 4}
	{1, 3, 5}
	{2, 4, 5}
0	{0, 1, 3}
1	{0, 1, 4}
1	{0, 1, 5}
1	{0, 2, 3}
2	{0, 2, 5}
1	{0, 3, 5}
2	{0, 4, 5}
1	{1, 2, 3}
2	{1, 2, 4}
1	{1, 3, 4}
2	{1, 4, 5}
2	{2, 3, 4}
2	{2, 3, 5}
3	**{2, 4, 5}**

	{0, 1, 2}
	{0, 3, 4}
	{1, 3, 5}
	{2, 4, 5}
	{0, 1, 4}
1	**{0, 1, 4}**
1	{0, 1, 5}
1	{0, 2, 3}
1	{0, 2, 5}
1	{0, 3, 5}
1	{0, 4, 5}
1	{1, 2, 3}
1	{1, 2, 4}
1	{1, 3, 4}
1	{1, 4, 5}
1	{2, 3, 4}
1	{2, 3, 5}

	{0, 1, 2}
	{0, 3, 4}
	{1, 3, 5}
	{2, 4, 5}
	{0, 1, 4}
	{0, 1, 5}
1	**{0, 1, 5}**
1	{0, 2, 3}
1	{0, 2, 5}
1	{0, 3, 5}
1	{0, 4, 5}
1	{1, 2, 3}
0	{1, 2, 4}
0	{1, 3, 4}
0	{1, 4, 5}
1	{2, 3, 4}

	{0, 1, 2}
	{0, 3, 4}
	{1, 3, 5}
	{2, 4, 5}
	{0, 1, 4}
	{0, 1, 5}
	{0, 2, 3}
1	**{0, 2, 3}**
0	{0, 2, 5}
0	{0, 3, 5}
0	{0, 4, 5}
1	{1, 2, 3}
0	{1, 2, 4}
0	{1, 3, 4}
0	{1, 4, 5}

3.2 Optimal $d+1$ Sharing for Functions with Degree $n-1$

Achieving d-th order security using a $d+1$ sharing for a single term of degree t, i.e. a product of t variables, mandates exactly $(d+1)^t$ shares for the product [72]. Alternatively, for $s_{in} = d+1$ input shares and a product of t variables one gets $s_{out} = (d+1)^t$ output shares.

The main difference with $td+1$ sharing is how the *non-completeness* property is interpreted in $d+1$ TI. Unlike with $td+1$ TI sharing, in the $d+1$ TI sharing

Table 3.3: First-order $d+1$ sharing and table for the 3-bit function $xy \oplus z$.

x	y	z	Output share ANF
0	0	0	$o_1 = x_0y_0 \oplus z_0$
0	1	$*$	$o_2 = x_0y_1$
1	0	$*$	$o_3 = x_1y_0$
1	1	1	$o_4 = x_1y_0 \oplus z_1$

each output share should contain only one share per input variable. In other words, if in an output share there are two shares of an input variable then the d-th order non-completeness is considered violated. Recall that for the $td+1$ TI using more shares per input variable is possible since the number of input shares is larger. We observe the difference between $td+1$ and $d+1$ TI in Table 3.1 and 3.4. The first output share of Table (3.1) contains 3 input shares of x: x_0, x_1 and x_2. In contrast, the $d+1$ sharing of Equation (3.4) has only one input share of x in the first output share: x_0. Therefore, *non-completeness* in $d+1$ TI is satisfied if we have only one share (or zero) of each input variable present in any given output share. We will assume that the independence of input shares is always satisfied for the $d+1$ case, an assumption that is not needed for $td+1$ TI.

Correctness of the sharing in the $d+1$ case is achieved by verifying that each monomial of a shared term (product) in the ANF of the unshared function f is present in one of the output shares.

Let us consider again the simple function $xy+z$. One possible first-order $d+1$ sharing of it is given in Table 3.3. The sharing can also be represented with a table, as shown on the left side of Table 3.3. Each output share is a row of a table, and each column represents the shares of a different input variable. The entry in row i and column j is the allowed input share of the j-th input variable for the i-th output share.

Columns are representing the variables x, y, and z, respectively. Compared to the $td+1$ set representation, the table representation restricts input shares for each variable separately, while output sets impose a restriction that is the same for all input variables.

The asterisk values indicate that we do not care about what input share of z is there, because the sharing of the linear term z is ensured by combining rows 1 and 4 of the table. Additionally, the table representation of the sharing does not uniquely determine the exact formula for each output share, and there is a certain freedom in determining where we can insert the input shares.

Table 3.4: First-order $d + 1$ sharing and table for 3-bit function $xy + xz + yz$.

x	y	z	Output share ANF
0	0	0	$o_1 = x_0y_0 + x_0z_0 + y_0z_0$
0	1	1	$o_2 = x_0y_1 + x_0z_1 + y_1z_1$
1	0	0	$o_3 = x_1y_0 + x_1z_0$
1	1	1	$o_4 = x_1y_0 + x_1z_1$
$*$	0	1	$o_5 = y_0z_1$
$*$	1	0	$o_6 = y_1z_0$

For example, we can use Table 3.3 to share the function $x + y + xy + z$. There are two options for the terms x_0 and x_1: rows 1 and 2, and rows 3 and 4, respectively. The same holds for the terms y_0 and y_1: y_0 can be either in output share 1 or 3, and y_1 can be in output share 2 or 4.

The *non-completeness* and *correctness* properties can be easily argued from the table representation. Since for every table row, each column entry in the table can represent only one input share of that column's variable, first-order *non-completeness* is automatically satisfied. For row 3 in Table 3.3 we ensure that only x_1 and y_0 can occur in that output sharing by fixing the entries representing x to 1 and y to 0. Hence, there is no way that x_0 or y_1 can be a part of that particular output share, which is the only way to violate *non-completeness* in a $d + 1$ sharing. *Correctness* of the table can be verified by checking the correctness for every monomial in the unshared function f individually. If the combined columns representing variables of the monomial contain all possible combinations of share indexes, the sharing is correct. Indeed, if this is the case, all terms of the shared product for each monomial can be present in the output sharing. Following the example from Table 3.3, for the monomial xy we see that all four combinations $\{(0, 0), (0, 1), (1, 0), (1, 1)\}$ are present in two columns representing the variables x and y. Hence, all of the terms of the shared product $xy = (x_0 + x_1)(y_0 + y_1) = x_0y_0 + x_0y_1 + x_1y_0 + x_1y_1$ can be present in at least one output share. The same holds for $z = z_0 + z_1$ as both combinations $\{(0), (1)\}$ are present in the output table of Table 3.3. Also, it is easy to see that the number of rows in the correct sharing table is lower-bounded by $(d + 1)^t$.

Now, consider a function $xy + xz + yz$. One possible first-order $d + 1$ sharing and its table is given in Table 3.4 with share indexes of the input variables on the left. The columns represent x, y, and z, respectively.

The table now has 6 rows representing the different output shares, more than the theoretically minimal 4 shares. The sharing given by Table 3.4 is also easily obtained when we try to derive it by hand. A naive approach is to start by sharing xy into four shares. Next, we try to incorporate xz into these four shares by setting all indexes of z to be equal to y. The problem arises when we now try to add the sharing of yz. In the existing four output shares, z and y have the same indexes. Thus we need to add two more shares for the terms y_0z_1 and y_1z_0.

Further on, we will show that for any function with n input variables of degree $t = n-1$ it is possible to have a $d+1$ sharing with the minimal $(d+1)^t$ shares.

Definition 3.2.1. *The table with n columns representing an output sharing of a function of degree t with n input variables is referred to as a D^n-table. The number of rows of the table is the number of output shares for a given sharing. If the output sharing is correct then the D^n-table is a t-degree correct D^n-table. A t-degree correct D^n-table with minimal numbers of rows is called an* optimal *D^n-table. An optimal D^n-table that has $(d+1)^t$ rows is called an* ideal *D^n-table, denoted D^n_t-table.*

The concept of D^n-table will be utilized in the rest of this thesis to succinctly present $d+1$ TIs, both in Chapters 4 and 5.

Obviously, for $t = n$ the ideal D^n_n-table is just a table that contains all different $(d+1)^t$ indexes of input variables in terms of the shared product that occur when sharing a function of degree t. We can also consider each row of a D^n-table as an ordered tuple of size n. The i-th value in such a tuple represents the i-th input variable, and its value is the allowed input share of that variable in the output share represented by the tuple. All tuple entries can have values from the set $\{0, \ldots, d\}$.

Definition 3.2.2. *A D^t-table D_1 is a t-subtable of a D^n-table D_2 for t given columns if D_2 reduced to these t columns is equal to D_1.*

We have shown with the sharing in Table 3.3 how one can check the correctness of the table. Now we generalize this by showing how to check if a given D^n-table can be used to share any function of degree t. It turns out that it is sufficient to check correctness only for the terms of degree t, since if we can share a product of t variables with a given number of output shares, we can also always share any product of a subset of these t variables using the same output shares.

It is easy to see that a D^n-table D can be used to share any function of degree t if and only if for any combination of t columns, the D^t-table formed by t chosen columns contains all possible $(d+1)^t$ ordered tuples of size t. In, other words, a t-subtable of D for any t columns is a t-degree correct D^t-table.

Table 3.5: D^3-table and its 3 2-subtables.

xyz	xy	xz	yz
000	00	00	00
011	01	01	11
100	10	10	00
111	11	11	11
001	00	01	01
110	11	10	10

Namely, a D^t-table that contains all possible $(d+1)^t$ ordered t-tuples represents the correct sharing for functions of degree t. If this is true for any combination of t columns of D we can correctly share any combination of products of size t from n input variables.

An example is given in Table 3.5 in which the D^3-table on the left can be used for a first-order sharing of any function of degree 2 since all 3 D^2-tables obtained from it have all 4 possible ordered 2-tuples $(0,0)$, $(0,1)$, $(1,0)$ and $(1,1)$ as at least one of its rows.

Next we show how one can construct an ideal D^n-table for any function for given n, d and $t = n-1$. To recap, we first build a $(d+1)^t \times n$ table D, where every row is a tuple of indexes (in a single row no variable index is allowed to be missing and, naturally, no variable index is duplicated); any t-subtable of D for any t columns is a t-degree correct D^t-table. Since $t = n-1$ we can consider t-subtable generation as column removal from D. Such a D^n-table D is then equivalent to a sharing which fulfills the *correctness* and the *non-completeness* properties of TI. Constructing an ideal D^n_n-table is trivial by enumerating all ordered index n-tuples. Its number of rows is $(d+1)^n$.

Showing that a particular D^n-table with $(d+1)^{n-1}$ rows is a D^n_{n-1}-table becomes equivalent to proving that removal of any single column (restriction to $n-1$ columns or, equivalently, variables) from the D^n-table yields a D^{n-1}_{n-1}-table. Alternatively, any $(n-1)$-subtable of a D^n_{n-1}-table is a D^{n-1}_{n-1}-table.

Here we will show how to build the D^n_t-table for the case when $t = n-1$. For any given D^n_{n-1}-table and security order d we will prove the existence of d other D^n_{n-1}-tables such that no n-tuple exists in more than one table. In other words, no two tables contain rows that are equal. We call such $d+1$ D^n_{n-1}-tables *conjugate tables*, and the sharings produced from them *conjugate sharings*. Having all rows different implies that these $d+1$ D^n_{n-1}-tables cover $(d+1)(d+1)^{n-1} = (d+1)^n$ index n-tuples, i.e. all possible index n-tuples.

Therefore, these $d+1$ D^n_{n-1}-tables together form a D^n_n-table.

We build the $d+1$ conjugate D^n_{n-1}-tables inductively. For a given d we build $d+1$ conjugate D^2_1-tables, then assuming $d+1$ conjugate D^n_{n-1}-tables exist, we construct $d+1$ conjugate D^{n+1}_n-tables.

The *initial step* is simple: D^2_1 has two columns (for the variables x and y) and in each row i (enumerated from 0 to d) of each conjugate table j (enumerated from 0 to d) we set the value in the first column to be i, and the value of the second column to be $(i+j) \mod (d+1)$, hence obtaining the $(d+1)$ conjugate D-tables with $d+1$ rows. Indeed, both columns of any of the constructed D^2_1-tables contain all values between 0 and d, so by removing either column, we always obtain a correct D^1_1-table. Also, this construction ensures that the second column never has the same index value in one row for different tables, therefore no two rows for different tables are the same, ensuring that the formed tables are indeed conjugate.

Induction step - assume we have $d+1$ conjugate D^n_{n-1}-tables. Using them we now build $d+1$ conjugate D^{n+1}_n-tables using two procedures:

- initDTable(n): initializes an empty D-table with n variables, with zero rows.
- appendRows(D_i, D_j, idx): appends $(d+1)^{n-1}$ additional rows to output D^{n+1}-table D_i, by taking D^n_{n-1}-table D_j and adding a new column at the end whose values are idx.

Algorithm 4 explains the iterative step of the induction. An example of the iterative step from Algorithm 4 is given in Figure 3.1.

Algorithm 4: Algorithm for optimal $d+1$ sharing

```
Input: $d+1$ conjugate $D^n_{n-1}$-tables $D_{(0,n)}, \dots, D_{(d,n)}$
Result: $d+1$ conjugate $D^{n+1}_n$-tables $D_{(0,n+1)}, \dots, D_{(d,n+1)}$
for $0 \leq i \leq d$ do
    $D_{(i,n+1)}$ := initDTable($n+1$)
    for $0 \leq j \leq d$ do
        appendRows($D_{(i,n+1)}, D_{(j,n)}, (i+j) \mod (d+1)$)
```

Lemma 3.2.1. *Given $d+1$ conjugate D^n_{n-1}-tables Algorithm 4 constructs $d+1$ conjugate D^{n+1}_n-tables.*

Proof. First, let us show that the constructed $d+1$ D^{n+1}_n-tables are conjugate, i.e., there is no $(n+1)$-tuple which belongs to more than one of them. Let us

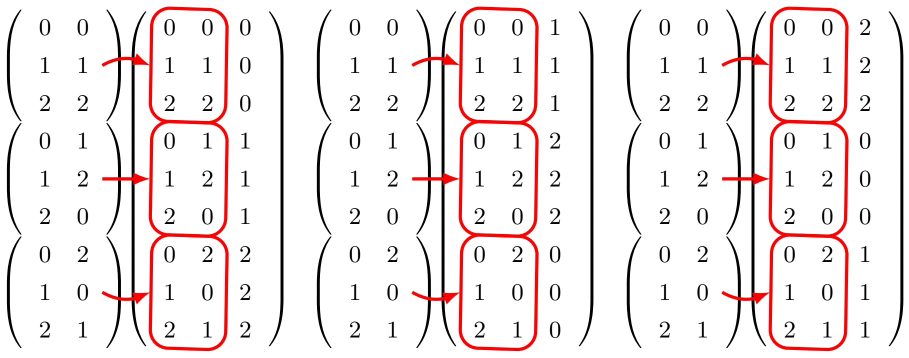


Figure 3.1: Generating conjugate D_2^3-tables from D_1^2-tables.

assume there exists an $(n+1)$-tuple which belongs to two D_n^{n+1}-tables. This implies the existence of an n-tuple which belongs to two of the initial $d+1$ D_{n-1}^n-tables, contradicting the fact that these initial tables are conjugate.

Finally, any restriction to a particular set of columns has to have all the combinations of index n-tuples, to satisfy the *correctness* property. In fact, it is sufficient to prove that any set of n columns in any of the new conjugate tables contains all possible n-tuples. Indeed, if we remove the last column in any of the so constructed tables, we get the union of the original $d+1$ D_{n-1}^n-tables forming one D_n^n-table. By definition a D_n^n-table satisfies this property. Lastly, we are left with the other case of removing one of the first n columns, which results in a table of dimensions $(d+1)^n \times n$. If we prove there are no duplicates among the $(d+1)^n$ tuples within this table, all combinations will be part of the table, making it again a D_n^n-table. Consider two n-tuples. If they are equal, their last indexes are also equal. By Algorithm 4, equality of the last indexes (these are in the $(n+1)$-st column) implies that the two $(n-1)$-tuples belong to one of the starting conjugate D_{n-1}^n-tables, i.e., they cannot be in different conjugate D_{n-1}^n-tables. However, for the $(n-1)$-tuples which belong to one of the starting D_{n-1}^n-tables by assumption it is known that there are no duplications and hence the considered two $(n-1)$-tuples cannot be equal. □

Theorem 3.2.1. *Any of the conjugate D_{n-1}^n-tables constructed by Algorithm 4 provides optimal sharing for given n, d and $t = n-1$.*

Proof. The algorithm is applied inductively for the number of variables from 2 till n. Since one D_{n-1}^n-table contains exactly $(d+1)^{n-1}$ rows, we conclude it is optimal because this is the theoretical lower bound for the number of output shares for the case $t = n-1$. □

A significant benefit of using an algorithmic solution is that it can easily be automated using a computer, removing the possibility of human error that is likely to occur, for functions with a complex ANF.

It is well known that a balanced Boolean function of n variables has degree at most $n-1$ [21]. Therefore all $n \times n$ S-Boxes which are permutations, have a degree of at most $n-1$. Indeed nearly all bijective S-Boxes used in symmetric ciphers are chosen to have the maximum degree $n-1$. In particular, inversion in the field always has maximum degree $n-1$, the most notable example being the AES S-Box. In the particular case of AES inversion, applying the algorithm shown here will produce the minimal number of shares: 128. In Chapter 5 we will explore a single-cycle $d+1$ TI of AES S-Box with an output sharing generated using Algorithm 4.

The most notable example where a low-degree function is used within a round structure is KECCAK's [4] χ-function which is a 5×5 S-Box of degree 2. A sharing with 8 shares can be easily found for χ by hand while a conjugate D^5-table will have 16 entries which correspond to the optimal sharing for degree 4. The heuristics given in the following section will help bridge this gap by providing minimal or near minimal sharings of functions where the degree is less than $n-1$.

3.3 Optimal $d+1$ Sharing for Functions of up to 8 Bits

When $t < n-1$ as was already shown from the KECCAK example, we see that the sharing obtained using Algorithm 4 does not give a solution with the minimal number of output shares. Alternatively, the previous section's method is not optimal when the degree of the function is lower than $n-1$. Therefore, a different strategy is needed to find the optimal sharing for functions with a degree lower than $n-1$. In order to find a solution for this particular case, we must first reformulate our problem.

By using a D^n-table to represent an output sharing, we showed in Section 3.2 that any function of degree t could be shared using a D^n-table D if all different t-subtables of D are correct D^t-tables. For a function f with n input bits, a D^n-table D can be used to correctly share f if for each ANF term of f of degree t, the corresponding t-subtable of D is a t-degree correct D^t-table. In other words, for each term l of t variables, columns representing output shares should contain all different $(d+1)^t$ combinations with repetitions. We evaluate the shares of l from 1 to $(d+1)^t$ in lexicographic order and say that output share

S covers the i-th share of term l if columns of S representing variables of l form the i-th share of t variables.

In order to find sharings of arbitrary functions we can transform the problem into a *set covering problem* (SCP). The SCP is a well-known discrete optimization problem, appearing in various applications, e.g., logic minimizer in EDA tools, mobile network base station placement, etc. Hence, we can leverage well-known methods for solving SCP instances to search for a minimal output sharing. Instead of trying to construct a correct sharing using smaller correct sharings, as in Algorithm 4, we will instead aim to choose a set of output shares among all possible output shares, making sure that the chosen output shares satisfy the correctness property.

Each different output share among all possible $(d+1)^n$ shares will be considered as a set, and the family of these sets will be referred to as $\mathcal{D}$. The universe $\mathcal{U}$ of all elements to be covered is created by going through the ANF of f, and for each term l of degree t we add $(d+1)^t$ elements, representing all shared terms of l. In other words, each shared monomial of each term is a separate element to be covered. Set S from $\mathcal{D}$ will contain an element e from $\mathcal{U}$ if the output share represented by S covers the shared term from f represented by e. Now given $\mathcal{D}$ and $\mathcal{U}$ we need to find a subfamily $\mathcal{C} \subseteq \mathcal{D}$ with minimal cardinality such that union of sets from $\mathcal{C}$ is $\mathcal{U}$. Concerning elements e from $\mathcal{U}$, in a valid solution, there exists at least one set S from $\mathcal{C}$ that contains e.

We can further represent SCP in terms of decision variables. With all possible output shares from $\mathcal{D}$ $1 \dots (d+1)^n$ we associate a $\{0,1\}$ variable x_S denoting if share S is chosen. The goal of finding correct and non-complete minimal sharing can then be formulated as:

$$\text{minimize} \sum_{S \in \mathcal{D}} x_S \tag{3.1}$$

$$\text{subject to} \sum_{S : e \in S} x_S \geq 1, \ (\forall e, e \in \mathcal{U}) \,. \tag{3.2}$$

Expression (3.1) is referred to as the objective function, while the inequalities given by (3.2) are called constraints.

The size of the search space for $d+1$ optimal sharing grows with the number of input variables and security order. Namely, there are $(d+1)^n$ decision variables for n-bit functions. For each decision variable the solver has to either include it into the solution or not, meaning the search space for the solver is $2^{(d+1)^n}$. As such the smallest search space for the sharing problems investigated in this Chapter is $2^{2^4} = 2^{16}$ for first-order secure quadratic 4-bit functions. Conversely, the largest search space investigated is $2^{3^8} = 2^{6561}$ for second-order sharings

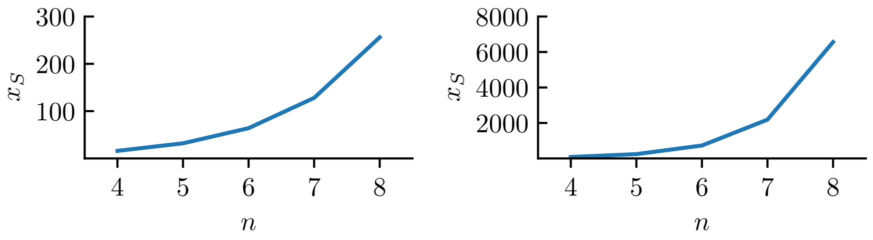


Figure 3.2: The number of decision variables x_S for the first- (left) and second-order (right) SCP sharing problem for n-bit functions. The size of the search space for the SCP solver is 2^{x_S}.

of 8-bit functions. Figure 3.2 shows the exponential growth of the decision variables x_S with the number of input bits.

The number of elements to cover from universe $\mathcal{U}$, i.e., $\binom{n}{t}$ for the sharings we are searching for, also impacts the solver in its ability to find optimal solution quickly. Namely, having more elements to cover slows down the time needed to find optimal solution.

3.3.1 Set Covering discrete optimization techniques

We have applied four different techniques to solve the underlying set covering problem: Constraint Programming [74], Mixed Integer Programming [78], Randomized Greedy with restarts [26], and Simulated Annealing [48] with a greedy heuristic. Since discrete optimization solvers provide varying levels of success depending on the problem instance, testing and comparing different techniques is the only way to discover which technique is the most suitable for minimizing the number of output shares in $d+1$ TI.

Two of the investigated techniques, Constraint Programming and Mixed Integer Programming, can find the optimal solution and prove its optimality, given enough time. However, for large SCP instances it might take a unreasonable amount of time to explore the entire search space. Hence, we restricted the running time of these two solvers to one hour on a regular CPU for all SCP instances we ran, unless stated otherwise. The other two techniques, Randomized Greedy with restarts, and Simulated Annealing with a greedy heuristic, are heuristic-based approaches. Hence they cannot prove the optimality of the found solution. However, they can potentially tackle bigger instances of SCP due to their computation speed.

Constraint Programming (CP) [74] focuses on finding a feasible solution given several constraints. Its original use is to determine if a problem is satisfiable. Nevertheless, it can also be used for minimization optimization by finding a feasible solution with objective cost N, then adding new constraints such that the objective function has to be smaller than N, and restarting the process. The cycle is repeated until the problem becomes infeasible, and the objective value of the final feasible solution is minimal, as the problem with smaller objective value is proven infeasible by the solver. We have used the freely available MiniZinc [61] software to solve our set covering problem.

Mixed Integer Linear Programming (MILP) [78] relaxes the problem such that decision variables become non-binary, but continuous real values, $x_S \in [0, 1]$, then tries to solve the underlying Linear Programming Problem [29] to establish a lower bound of the objective function. Afterwards, it tries to find a smallest solution such that all decision variables are integers, satisfying the original problem constraints. Similar to CP, MILP can prove the optimality of the solution. We have used the Gurobi 9.0 [43] solver for this part.

Randomized Greedy heuristic with restarts, or Iterated Greedy (IG) is a technique where a solution is constructed in a greedy manner, and in each step we take the set S that covers most uncovered elements so far. We stop when all elements are covered. All ties are broken randomly: if multiple sets cover an equal number of still uncovered elements, the algorithm randomly chooses one of them to add to the solution being built. We loop this approach multiple times and take the solution from the iteration that has the smallest number of sets. Since ties are broken randomly, solutions will differ from run to run. The optimality of this technique cannot be proven since the greedy algorithm finds local minima, not a global one for SCP. However, in practice, its outputs are often close to optimal.

Simulated Annealing [48] is a meta-heuristic where a neighbor solution with a higher objective function cost is accepted with a probability that gradually decreases during execution time. Intuitively, accepting worse solutions allows us to explore more of the search space and escape local minima. Over time lowering of the acceptance probability guides the search more and more toward good solutions, while the earlier rounds are used to explore a large search space more indiscriminately. The probability parameter is called temperature. We utilized the implementation approach given in [19, 55] where we separate execution into multiple rounds. After each round, the temperature is decreased by a constant factor *cool*. In each round, a number I of neighbors is explored. A neighbor is constructed by removing some of the sets from the solution, then constructing a new solution using the remaining sets as a starting point, and adding new ones until all elements are covered again. We accept the new solution if it is better than the previous one, or if not, we still accept it with probability

$\exp(-\delta/temp)$ where δ is the difference in objective function of the new and the current solution.

Here we reiterate the implementation details from [55]. The parameters are:

- **A**: data structure providing relation information of which sets cover which element, normally given as a $\{0,1\}$ matrix.
- **cool**: Temperature reduction ratio between rounds $temp_{r+1} = temp_r \times cool$
- ρ: used to determine the percentage of shares to be dropped from the current solution, during neighbor construction.
- **R**: number of rounds to run.
- **I**: number of neighbors examined in each round.
- **rand()**: function that returns a uniformly distributed random real number in range $[0,1]$.
- **temp**: initial temperature parameter used to determine the probability of accepting a bad neighbor.
- **Construct()**: greedy heuristic method used to construct a new covering after a percentage of shares has been removed from it.
- **Perturb()**: neighbor defining function. A factor ρ of shares is stripped from the current solution. Shares removed have the least amount of uniquely covered elements.
- **RemoveRedundant()**: executed after a new solution has been constructed. It can happen that some of the chosen shares are redundant since all of the elements covered by it are already fully covered by other shares in the candidate solution.

The simulated annealing algorithm for SCP is shown in Algorithm 5.

3.3.2 Sharing solutions

We have applied four discrete optimization techniques to all algebraic functions of up to 8 bits, of algebraic degree t which contain all t-degree terms in their ANFs. Since almost all symmetric key designs utilize a nonlinear S-Box of 8 or fewer bits, the obtained results would be applicable to most symmetric key primitives. Since optimal sharing has already been solved in Section 3.2 for

Algorithm 5: Simulated Annealing algorithm for the set cover problem.

```
Input: $A$, $R$, $I$, $temp$, $cool$, $\rho$
Result: Smallest found sharing $\mathcal{C}_{best}$
$\mathcal{C}^* := \{\}$
$\mathcal{C}^* := Construct(A, \mathcal{C}^*)$
$r := 0$
repeat
    $i := 0$
    repeat
        $\mathcal{C}' := Perturb(A, \mathcal{C}^*, \rho)$
        $\mathcal{C}' := Construct(A, \mathcal{C}')$
        $\mathcal{C}' := RemoveRedundant(A, \mathcal{C}')$
        $\delta := ObjectiveCost(\mathcal{C}') - ObjectiveCost(\mathcal{C}^*)$
        $rnd = rand()$
        if $(\delta \leq 0)$ $or$ $(rnd \leq e^{-\delta/temp})$ then
            $\mathcal{C}^* := \mathcal{C}'$
            if $ObjectiveCost(\mathcal{C}') < ObjectiveCost(\mathcal{C}_{best})$ then
                $\mathcal{C}_{best} := \mathcal{C}'$
        $i := i + 1$
    until $i == I$
    $temp := temp \times cool$
    $r := r + 1$
until $r == R$
return $\mathcal{C}_{best}$
```

degree $t = n - 1$, we have focused on algebraic functions where $t < n - 1$. In order for the solution to be generic, we further assume the most extreme case. That is, given n variables and degree t, we search for a sharing of a function with all $\binom{n}{t}$ t-degree terms present in the ANF. Sharing of such a function can be used for any n-bit function of degree t. However, a more efficient sharing for a given n-bit function f of degree t might exist, depending on the number of t-degree terms that are present in the ANF and their structure, and the same discrete optimization methodology can be applied on f to find a possibly better sharing.

First, we have applied CP using Minizinc [61] with Chuffed 0.10.4 [24] and Google OR-tools [67] solvers. For $d = 1$, both options can find optimal sharings for all cases, except for 8-bit functions of degree 5, where a solution with 52 shares is found after a few hours but without proof of optimality, even after running the CP solver for several days on a regular PC. Optimality for other

Table 3.6: Number of shares found using the CP solver. Values in bold mean that the solver proved optimality.

	$d = 1$					$d = 2$				
t	2	3	4	5	6	2	3	4	5	6
$n = 4$	**5**					**9**				
$n = 5$	**6**	**10**				15	44			
$n = 6$	**6**	**12**	**21**			–	–	–		
$n = 7$	**6**	**12**	**24**	**42**		–	–	–	–	
$n = 8$	**6**	**12**	**24**	52	**85**	–	–	–	–	–

Table 3.7: Number of shares found using the MILP solver. Values in bold mean that the solver proved optimality.

	$d = 1$					$d = 2$				
t	2	3	4	5	6	2	3	4	5	6
$n = 4$	**5**					**9**				
$n = 5$	**6**	**10**				**11**	**33**			
$n = 6$	**6**	**12**	**21**			**12**	**33**	119		
$n = 7$	**6**	**12**	**24**	**42**		**12**	45	153	440	
$n = 8$	**6**	**12**	**24**	52	**85**	14	63	–	–	–

cases is proven within several tens of minutes. For the second-order case $d = 2$, the CP solver is only able to prove optimality for the simplest of cases of 4 bits and degree 2. It can find a solution when $n = 5$, but without proof of optimality. For $n > 5$, the solver cannot provide any solutions within a few minutes, so we deem it not suitable for those cases due to the size of the search space. We did not see much difference in solution times between Chuffed and OR-tools solvers, although Chuffed seems to finish the search slightly faster. The resulting number of shares using CP solvers is given in Table 3.6.

Next we have tried to use the MILP solver, in particular the Gurobi 9.0 solver [43]. For $d = 1$ the solver was able to find same solutions and prove optimality in less time. However, for the case of 8 bits and degree 5, a solution of 52 was found again, but without proof of optimality. When $d = 2$, the MILP solver was more successful than the CP solver, finding optimal solutions for degree 2 functions for all $n = 4, 5, 6, 7$, and degree three functions of 5 and 6 bits. However, more complex cases seem to quickly become difficult for the solver. The resulting number of shares using MILP solver is given in Table 3.7.

Table 3.8: Number of shares found using the IG heuristic.

	$d = 1$					$d = 2$				
t	2	3	4	5	6	2	3	4	5	6
$n = 4$	5					9				
$n = 5$	6	12				11	36			
$n = 6$	6	12	22			13	40	128		
$n = 7$	6	12	24	47		14	45	138	409	
$n = 8$	6	12	30	56	96	15	45	135	405	1387

It becomes apparent that CP and MILP solvers struggle with $d = 2$, which is unsurprising because the number of decision variables increases exponentially. Hence for the more difficult cases, heuristics are the only possible way to find good solutions. The Iterated Greedy approach consists of 100 000 greedy algorithm runs and choosing the best solution among them. It finishes in just a matter of seconds, even for harder cases. The program can sometimes find optimal solutions of the CP and MILP approaches, but even when it does not, the solutions it finds are within 30% of minimal, where the minimal solution is known. The IG run results are given in Table 3.8.

To improve on the results of the IG heuristic, we have also tested the simulated annealing technique. First, the IG technique was used with 20 000 runs to provide an initial solution, and then SA was run with $R = 150$ rounds and $I = 100$ iterations in each round. The program ran extremely fast on a regular PC, and it was the slowest for 8-bit functions of degree 6 where it finished in about 5 minutes. SA program was able to find the same results for $d = 1$ as the CP and MILP solvers, finding optimal solutions faster. For $d = 2$ it improved on some of the instances compared to the IG approach, but the solution quality was at most 10% better in instances where CP and MILP were unable to provide solutions in a reasonable amount of time. In some instances with 8 bits and degrees 3, 4 and 5, it was unable to improve upon the solution provided by the IG approach. Modifying the SA annealing parameters had a limited impact on the quality of the solution. For minimal sharing problems we experimented with cooling factors in the range of $[0.8, 0.95]$, initial temperature between $[20, 200]$ and ρ in range $[0.1, 0.5]$. For these parameter ranges the cooling factor of 0.91, starting temperature of 100, and removal coefficient ρ of 0.2 seemed to be working well in almost all cases. The coefficient ρ had the most impact on the improvements, and we noticed that smaller values are more beneficial for larger t values, about 0.1, while values of 0.3 work better on smaller t values. The SA run results are given in Table 3.9.

Finally, we can put together the solutions of all four approaches to collect the

Table 3.9: Number of shares found using the SA heuristic.

	$d=1$					$d=2$				
t	2	3	4	5	6	2	3	4	5	6
$n=4$	5					9				
$n=5$	6	10				11	33			
$n=6$	6	12	21			12	33	115		
$n=7$	6	12	24	42		12	40	130	379	
$n=8$	6	12	24	52	85	14	45	135	405	1234

Table 3.10: Best sharing using all four approaches. Values in bold mean that solver proved optimality.

	$d=1$					$d=2$				
t	2	3	4	5	6	2	3	4	5	6
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				**11**	**33**			
$n=6$	**6**	**12**	**21**			**12**	**33**	115		
$n=7$	**6**	**12**	**24**	**42**		**12**	40	130	379	
$n=8$	**6**	**12**	**24**	52	**85**	14	45	135	405	1234

best ones. The aggregate results are presented in Table 3.10. Examining the best solutions found, we can determine that the optimal sharing does not have significantly more output shares compared to the trivial bound of $(d+1)^t$. For $d = 1$ the increase is up to 50 percent, except in the hardest case with 8 variables and degree 5 functions, in which we have 52 shares compared to the bound of 32 shares, an increase of a little over 60%. A similar situation happens with $d = 2$ where found solutions are within 70% increase. Due to the particular case of SCP for sharing being somewhat pathological, where all shares cover the equal amount of elements, it is difficult for the solver to find optimal solutions when the number of total shares increases. In contrast, good solutions are still relatively easily discovered using a greedy heuristic.

A comparison of our result with the recent ones presented in Section 3.2 and [84] is presented in Table 3.11. Obviously, the method presented in Section 3.2 achieves optimality when $t = n-1$, but it is ineffective for $t < n-1$. The greedy heuristic given in [84] finds solutions that are multiples of $(d+1)^t$. However, the authors only presented the solution for a specific case of an 8-bit degree 3 function, and for $(n = 4, t = 2, d = 2)$, $(n = 4, t = 2, d = 3)$, $(n = 5, t = 2, d = 4)$, $(n = 5, t = 3, d = 3)$, $(n = 6, t = 3, d = 3)$ the optimal solution is found

Table 3.11: Comparison to previously known results.

	$d=1$					$d=2$				
t	2	3	4	5	6	2	3	4	5	6
Proposed methodology										
$n=4$	**5**					**9**				
$n=5$	**6**	**10**				**11**	**33**			
$n=6$	**6**	**12**	**21**			**12**	**33**	115		
$n=7$	**6**	**12**	**24**	**42**		**12**	40	130	379	
$n=8$	**6**	**12**	**24**	52	**85**	14	45	135	405	1234
Construction from Algorithm 4										
$n=4$	8					27				
$n=5$	16	16				81	81			
$n=6$	32	32	32			243	243	243		
$n=7$	64	64	64	64		729	729	729	729	
$n=8$	128	128	128	128	128	2187	2187	2187	2187	2187
Construction from [84]										
$n=4$	8					**9**				
$n=5$	8	16				18	54			
$n=6$	8	16	32			18	54	162		
$n=7$	8	16	32	64		18	54	162	486	
$n=8$	8	16	32	64	128	18	54	162	486	1458

while not providing more information. Hence in Table 3.11 we indicated the smallest number of output shares the algorithm [84] could find. Our method provides better results in all cases for first and second security order.

Concrete sharings are given by Tables A.1, A.2, A.3, A.4 and A.5 in the Appendix A. If we examine the solutions for $d = 1$ we can see that many solutions have symmetric order: if an output share has a binary representation x, then the solution also contains the output share with binary representation equal to one's complement of x. One could assume that optimal solutions will always have sharings with such structure, but this is not the case. For example, if we provide this additional constraint to the CP solver, it will no longer find the sharing of 7-bit functions of degree 2 to be 12, but 14. The symmetric structure is obtained from the CP solver, probably based on the heuristic it used to parse the search space.

3.3.3 Sharing for AES cubic power functions

As an example of an application of the generic method presented here, we demonstrate the improved sharing of the power functions x^{26} and x^{49}, used to create AES inversion by De Meyer et al. and Nikova et al. [84, 63]. Both of them are cubic Boolean functions since Hamming weight of both 26 and 49 is 3 [21], and De Meyer et al. [84] presented a sharing using 16 output shares is used to produce a first-order secure $d+1$ implementation. The sharing is only valid for the lowest bit of the power functions, but using the rotational symmetry of power functions in normal basis, we can use the same sharing for other output bits as well.

The normal basis generator chosen by De Meyer et al. [84] is $\beta = 205$, with 205 being the representation of the generator in the AES polynomial basis. Table 3.10 gives a sharing with 12 shares which is 4 shares or 25% less than for a generic cubic function. This approach was applied to the first-order $d+1$ TI design by Shahmirzadi et al. [79]. However, if we apply the smallest sharing on the exact ANF of the cubic power functions, we can improve upon this result. Using the MILP set covering program, we were able to find a first-order sharing for both power functions x^{26} and x^{49} with 10 shares each, 6 shares or 37.5% less than the original sharing. Furthermore, we have applied the same method to find a second-order $d+1$ sharing of the same power function and the same normal basis generator. Surprisingly, both functions can be shared with $(d+1)^t = 27$ shares, the theoretical minimum. In order to make sure this is the best sharing, we have investigated 7 other pairs of cubic functions that yield inversion when composed together: $(13, 98), (19, 161), (38, 208), (52, 152), (67, 137), (76, 104), (134, 196)$. In addition, we have expanded the search across all 128 normal basis generators. An exhaustive search showed that 10 shares are the smallest number of shares in every case. Other generators that can be used to produce the sharing with 10 shares of the cubic power functions are $\{36, 96, 117, 124, 140, 199, 202\}$. Interestingly enough, in all the pairs of cubic power functions that produce inversion, the generators that have 10 shares in the output sharing are always the same. To make the notation as succinct as possible, we will only enumerate the chosen shares of the power functions x^{26} and x^{49} with normal basis generator 205 in their lexicographical order, the first share having index 0, and the last share having index $(d+1)^n - 1$. In other words, we look at all possible shares as n-digit words in base $d+1$. For example if we had a sharing with $d = 2, n = 4, t = 2$ given as $[2, 12, 25, 31, 44, 45, 60, 64, 77]$, it means that the actual nine shares are

$$(0,0,0,2)\ (0,1,1,0)\ (0,2,2,1)$$
$$(1,0,1,1)\ (1,1,2,2)\ (1,2,0,0)$$
$$(2,0,2,0)\ (2,1,0,1)\ (2,2,1,2)\,.$$

A shortcut to getting actual output shares for first-order designs using this notation where allowed shares are 0 and 1, is to observe the binary representation of the index value when each bit represents the allowed share of that input variable. It should be noted that this representation of the sharing, while succinct, is not unique with respect to the actual distribution of ANF terms, as low degree monomials have multiple possible output shares they can be a part of. For a first-order sharing of an 8-bit function, there are $2^8 = 256$ possible shares, while for the second-order sharing there are $3^8 = 6561$ total shares. The chosen shares for first-order sharing are given below:

$$shares(x^{26}) = [0, 30, 43, 93, 114, 154, 181, 195, 232, 246]$$
$$shares(x^{49}) = [0, 30, 79, 115, 124, 165, 187, 214, 217, 234]\,.$$

The chosen shares for the second-order sharing are given below:

$$\begin{aligned} shares(x^{26}) = &[0, 439, 626, 796, 1145, 1338, 1511, 1938, 2044, 2388, 2575, 2690,\\ &3103, 3290, 3474, 3863, 3975, 4162, 4533, 4639, 5069, 5212, 5408,\\ &5754, 5927, 6111, 6550]\\ shares(x^{49}) = &[0, 338, 673, 930, 1025, 1324, 1617, 1919, 2011, 2218, 2553, 2882,\\ &3148, 3231, 3542, 3745, 4053, 4148, 4436, 4762, 5097, 5276, 5368,\\ &5676, 5963, 6271, 6354]\,. \end{aligned}$$

3.4 Conclusion and Outlook

We have introduced several methods for optimizing Threshold Implementations, which makes low latency, low energy, and higher throughput of side-channel secure designs practical. First, we provided an algorithm that produces a $d+1$ TI sharing with the optimal (minimum) number of output shares for any n-input Boolean function of degree $t = n-1$ and for any security order d. Second, when

$t < n - 1$ we presented a discrete optimization based methodology that can be used to find good, and in many cases optimal, sharings of Boolean functions up to 8 bits. We also reduced the problem of finding optimal sharing to an instance of the set covering problem, which can be efficiently solved using various discrete optimization techniques. Third, we presented a heuristic for minimizing the number of output shares for higher-order $td + 1$ TI. Minimizing the number of output shares is of general interest since the method of minimizing the number of output shares can be applied to any cryptographic design.

We would like to summarize that the generic algorithm for achieving the minimal number of output shares is an essential tool for the side-channel secure circuit designer. However, finding the minimal number of output shares is not the only design criterion to investigate when designing for low latency and low energy applications. Chapters 4 and 5 rely on the sharing techniques presented in this chapter to provide TI designs with a minimal number of output shares.

Chapter 4

Low Latency Side-Channel Protected PRINCE Cipher

"Measure twice, cut once."

English proverb

The work presented in this chapter is based on three publications in which I was the main author [14, 17, 15], and my contributions were designing, realizing and analyzing presented PRINCE implementations.

In this chapter, we take a closer look at side-channel protected implementations focusing on low latency applications, both with respect to cycle latency and overall latency. We also demonstrate how low cycle latency can lead to energy-efficient design. Specifically, we will look at the PRINCE block cipher due to its low latency oriented design.

Applications such as memory encryption, in which low latency is essential and additional cycles during processing considerably reduce the memory response time, mandate the use of highly performant low latency cryptographic solutions. Moreover, the critical path must remain short to meet the RAM system's stringent frequency requirements. Being designed specifically with these criteria in mind, PRINCE is the optimal choice for such an application. Many commercially available products today incorporate a hardware PRINCE implementation to meet strict low latency encryption needs, including NXP Semiconductors' LPC55S general-purpose IoT microcontrollers [65].

However, side-channel attacks could pose a serious threat even for such an application, mandating the use of countermeasures in these extreme environments. Hence, we investigate the overhead caused by TI side-channel countermeasure in a case study of PRINCE. Most of the hardware relevant metrics are examined in detail, namely the number of gate equivalents needed to implement a circuit, power and energy consumption, and the PRNG requirements imposed by the countermeasure. Several design options are discussed, based on the S-Box decomposition, as well as on the S-Box implementation without decomposition using $d+1$ and $td+1$ TI sharings described in Chapter 3. All design options are thoroughly analyzed and should provide sufficient insight for any future design, with its drawbacks and fortes clearly indicated. The chapter consists of a brief explanation of the structure of the PRINCE S-Box, followed by the description of several TI versions, both $d+1$ and $td+1$, including first- and second-order security. We also explain the method to reduce the randomness for round-based implementations leveraging the mixing layer structure within PRINCE. Synthesis results are elaborated, presented, and compared using a commercial TSMC 90 nm library. Finally, two $d+1$ TI designs without S-Box decomposition are tested for their side-channel resilience using standard TVLA methodology.

4.1 PRINCE S-Box Decomposition

The PRINCE S-Box has an algebraic degree three and belongs to the class $\mathcal{C}_{131}$ [9]. According to Bilgin et al. [9] and the tables provided by Nikova [62] there are several hundreds of decompositions into three quadratic S-Boxes and four affine transformations.

We choose a decomposition where all three quadratic S-Boxes are the same, belonging to class Q_{294}. Such a decomposition can reuse the Q_{294} circuit during evaluation, leading to a smaller area footprint of the design. Decomposition leads to lower area and randomness requirements as they depend on the algebraic degree of the function when applying TI. However, the performance is penalized. The PRINCE S-Box ANF $(o_1, o_2, o_3, o_4) = F(x, y, z, w)$ is given by:

$$\begin{aligned}
o_1 &= 1 \oplus wz \oplus y \oplus zy \oplus wzy \oplus x \oplus wx \oplus yx \\
o_2 &= 1 \oplus wy \oplus zy \oplus wzy \oplus zx \oplus zyx \\
o_3 &= w \oplus wz \oplus x \oplus wx \oplus zx \oplus wzx \oplus zyx \\
o_4 &= 1 \oplus z \oplus zy \oplus wzy \oplus x \oplus wzx \oplus yx \oplus wyx \,. \qquad (4.1)
\end{aligned}$$

The S-Box and its inverse decompositions used in our implementation are:

$$S = A_1 \circ Q_{294} \circ A_2 \circ Q_{294} \circ A_3 \circ Q_{294} \circ A_4$$

$$S^{-1} = A_5 \circ Q_{294} \circ A_2 \circ Q_{294} \circ A_3 \circ Q_{294} \circ A_6 \,. \tag{4.2}$$

Here A_1 to A_6 are affine transformations and their respective look-up tables are:

$$A_1(x) = [C, E, 7, 5, 8, A, 3, 1, 4, 6, F, D, 0, 2, B, 9]$$

$$A_2(x) = [6, D, 9, 2, 5, E, A, 1, B, 0, 4, F, 8, 3, 7, C]$$

$$A_3(x) = [0, 8, 4, C, 2, A, 6, E, 1, 9, 5, D, 3, B, 7, F]$$

$$A_4(x) = [A, 1, 0, B, 2, 9, 8, 3, 4, F, E, 5, C, 7, 6, D]$$

$$A_5(x) = [B, 8, E, D, 1, 2, 4, 7, F, C, A, 9, 5, 6, 0, 3]$$

$$A_6(x) = [9, 3, 8, 2, D, 7, C, 6, 1, B, 0, A, 5, F, 4, E] \,.$$

The ANF of the Q_{294} quadratic permutation, $(o_1, o_2, o_3, o_4) = F(a, b, c, d)$ has the following form:

$$o_1 = a$$

$$o_2 = b$$

$$o_3 = ab \oplus c$$

$$o_4 = ac \oplus d \tag{4.3}$$

It should be noted that the decomposition of the PRINCE S-Box using the Q_{294} quadratic S-Box is not unique. For example, in the work by Moradi and Schneider [59] Q_{294} is also used to decompose the PRINCE S-Box, but the affine transforms A_1 to A_6 used there are different from the ones we described here.

We recall that, for a secure implementation with this decomposition method, nonlinear operations need to be separated by registers, making the evaluation of a single S-Box take 3 clock cycles. If the S-Box needs to be separated from the linear layer to avoid transient leakage, the number of cycles is further doubled to 6.

We will explore the implementation of the decomposed S-Box in order to address the issue of low-area and low-power applications but, in what follows, we also explore the sharing of the non-decomposed S-Box to address the issue of low latency and low energy.

4.2 Implementations, Results and Evaluation

Next we describe in detail eight different side-channel protected versions of PRINCE. The implementations are parametrized over three factors: first- and second-order protection, using $td + 1$ and $d + 1$ TI sharings, with and without S-Box decomposition. The round-based unprotected version of PRINCE is explained, and it serves as a reference, followed by designs of S-Boxes used in each of eight versions. Finally, the round structure of the PRINCE side-channel protected designs is elaborated.

4.2.1 PRINCE unprotected implementation

Figure 4.1 represents the architecture of the unprotected round-based PRINCE. In the unprotected design, the S-Box evaluation takes one cycle. One encryption is performed in 12 clock cycles. Due to the S-Box and its inverse being in the same equivalence class, they can share circuitry for both “forward” and “inverse” rounds by utilizing the affine transformation circuits A_{io} during the inverse S-Box evaluation. By adding an additional multiplexer, the design can perform the decryption operation as well. To minimize the overhead of added decryption functionality, we implement the round counter as explained in the design by Moradi and Schneider [59], XORing the round constant multiplexer selector to the input decryption indicator.

Following Figure 4.1, to correctly evaluate the S-Box, the data travels through multiplexers $\alpha_1 - \beta_2 - \delta_1$, except in the first round where the path is $\alpha_1 - \beta_1 - \delta_1$. Similarly, when evaluating the inverse S-Box, the active path is $\alpha_2 - \gamma_1 - \delta_2$, except in the last round when the chosen path is $\alpha_1 - \gamma_2 - \delta_2$.

4.2.2 TIs of Q_{294}

We have implemented $td + 1$ and $d + 1$ variants of TI for both the first- and the second-order Q_{294} implementations. We use the first-order $td + 1$ direct TI sharing [9], the second-order $td + 1$ sharing with 5 input shares and 10 output shares, as explained by Bilgin et al. [7]. For the $d + 1$ first- and second-order implementations, we used the sharing by Reparaz et al. [72]. Compression (cf. Section 2.6) is applied to the second-order $td + 1$ and both $d + 1$ versions.

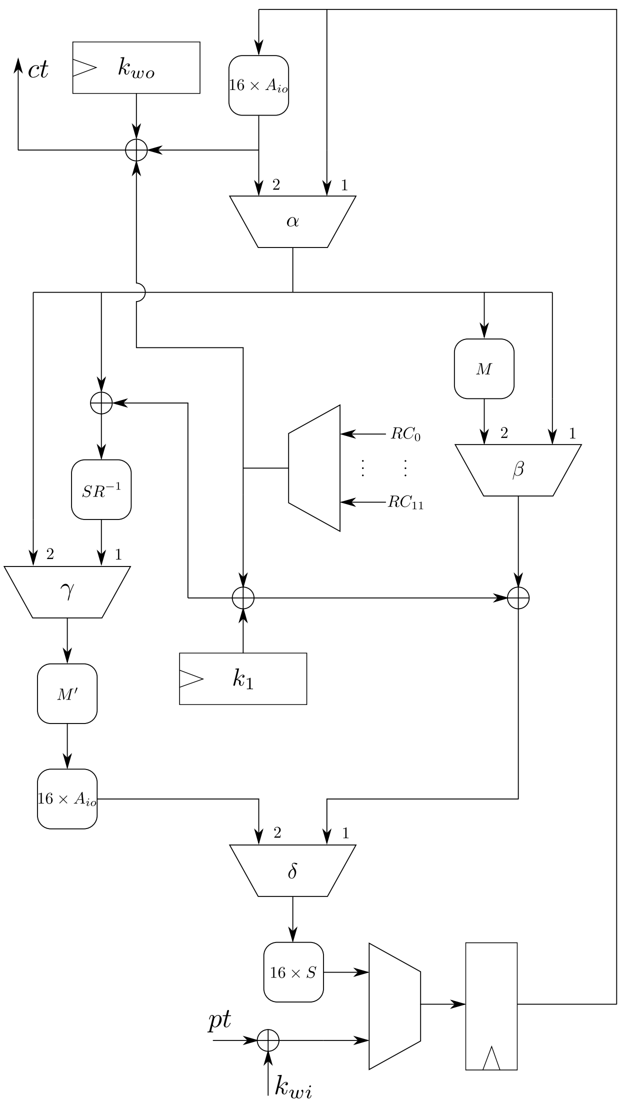


Figure 4.1: Unprotected PRINCE round based architecture.

The $td + 1$ first- and second-order hardware diagrams are shown in Figure 4.2 and Figure 4.4, respectively. Due to the complexity of the second-order $td + 1$ implementation, the partial share evaluation circuit is isolated in Figure 4.3, while Figure 4.4 references it as a block. Similarly, $d + 1$ first- and second-order

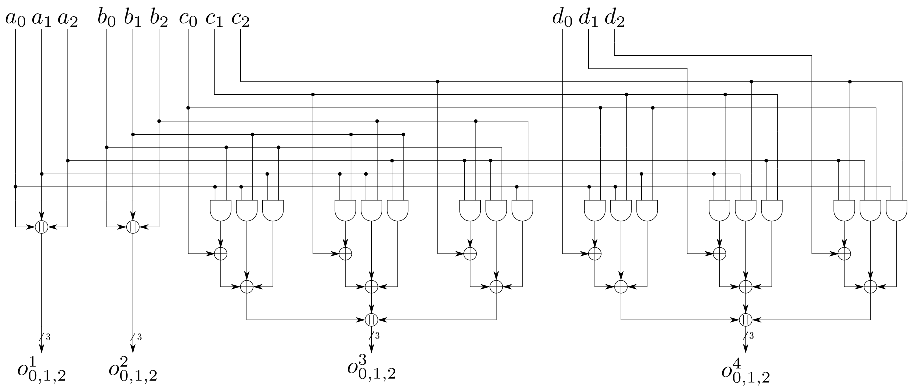


Figure 4.2: Q_{294} first-order $td + 1$ TI hardware circuit.

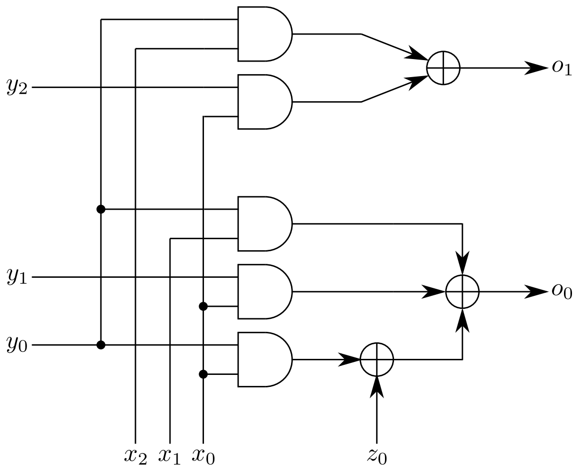


Figure 4.3: Partial quadratic evaluation circuit of $xy + z$.

hardware diagrams can be seen in Figure 4.5 and Figure 4.6, respectively.

4.2.3 First-order secure $td + 1$ TI of the PRINCE S-Box

For the first-order $td + 1$ design of the PRINCE S-Box without decomposition, we generated a sharing with 4 input and output shares following the sharing method presented by Bilgin et al. [9]. Since the number of inputs and outputs

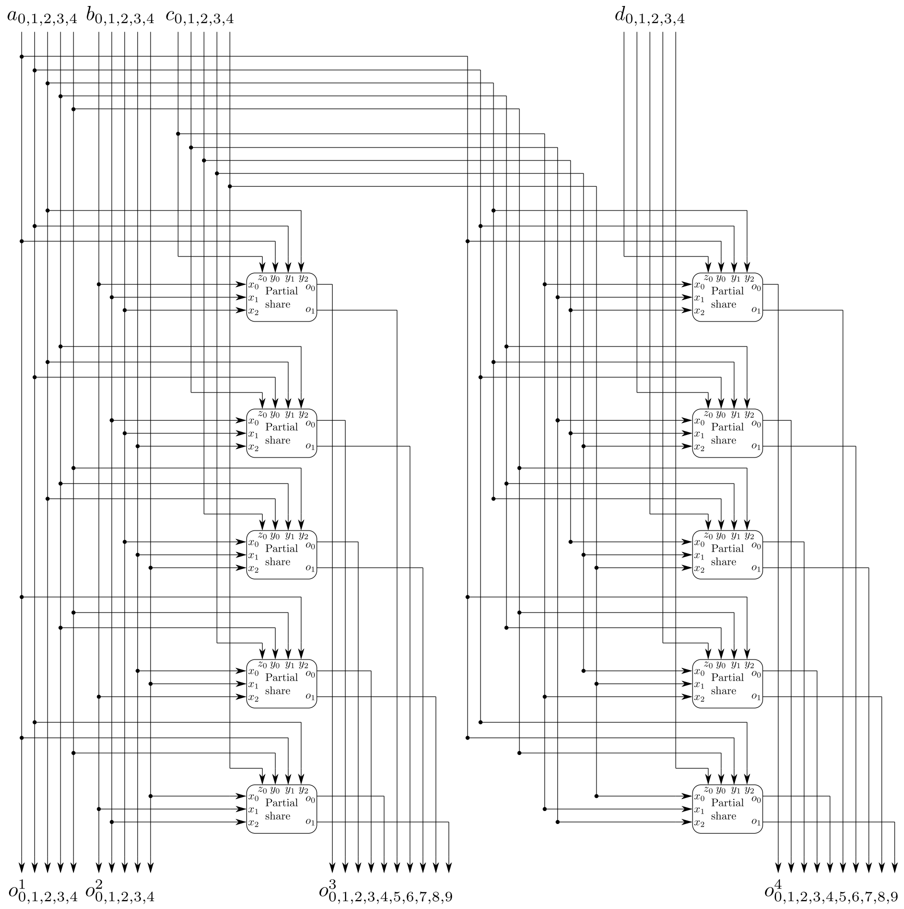


Figure 4.4: Q_{294} second-order $td+1$ TI hardware circuit.

is equal, again, there is no need for compression after the register stage. The direct sharing we used is not uniform; hence the output bits are refreshed using the simple remasking method given in Equation (2.8), requiring 3 random bits per output S-Box bit, or 12 random bits in total to refresh the entire S-Box.

4.2.4 Second-order secure $td+1$ TI of the PRINCE S-Box

To create a second-order secure $td+1$ masking for the PRINCE S-Box we have applied the iterated greedy algorithm described in Section 3.1. This algorithm

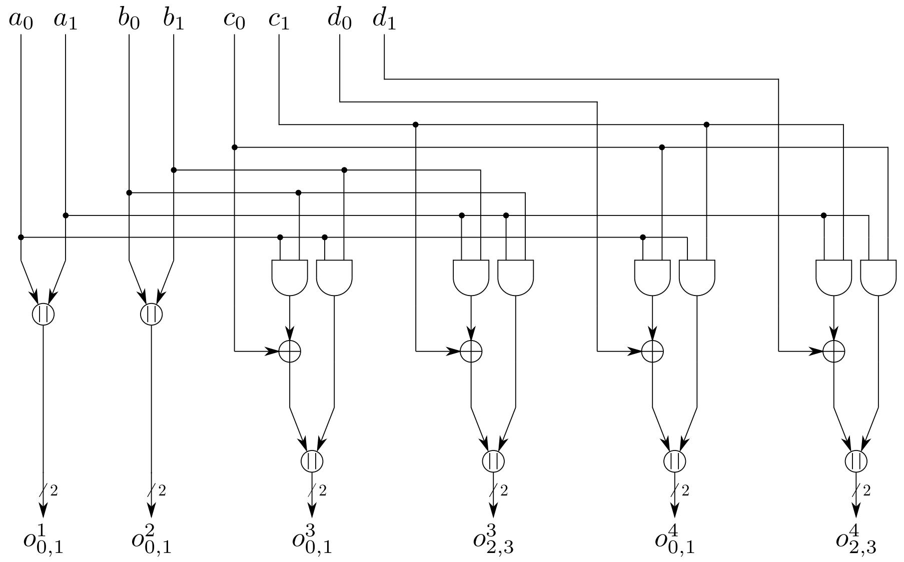


Figure 4.5: Q_{294} first-order $d + 1$ TI hardware circuit.

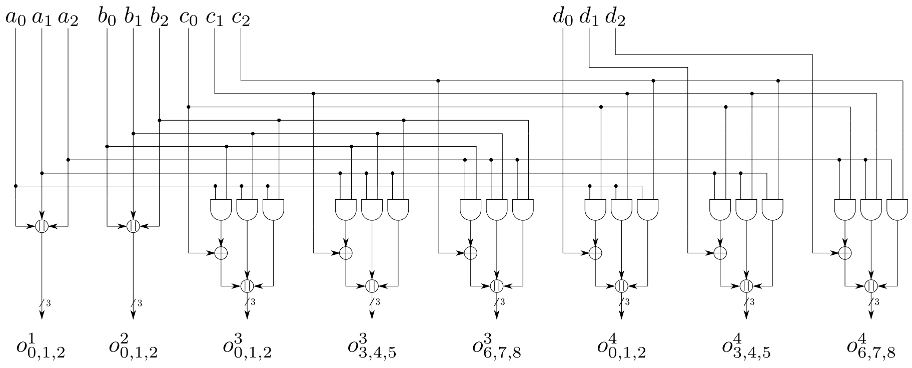


Figure 4.6: Q_{294} second-order $d + 1$ TI hardware circuit.

provides a solution that has 17 output shares and 8 input shares. Compared to the higher-order $td + 1$ method given by Bilgin et al. [7], which produces 35 output shares with 7 input shares, the total number of shares is reduced by almost a half. All output bits are refreshed using the ring remasking from Equation (2.7), requiring 68 random bits per S-Box. Since the rest of the PRINCE core uses three shares (see Section 4.2.7), we generate five extra shares before the S-Box input which consumes extra 20 random bits. Therefore, the whole S-Box evaluation uses 88 random bits.

4.2.5 First-order secure $d+1$ TI of the PRINCE S-Box

To implement the first-order secure masking of the PRINCE S-Box, we use the algorithm described in Section 3.2 to obtain the first-order conjugate D_3^4-table. This table represents an optimal solution for 2 input shares with 8 output shares for each input/output bit of the S-Box. Recall that the PRINCE S-Box is a 4×4-bit S-Box and that it has algebraic degree 3. All output bits can be refreshed using the mask refreshing of Equation (2.8) similar to the first-order $td+1$ S-Box implementation. Simple refreshing uses 7 bits of randomness per output bit, or 28 bits per S-Box in total. The optimal sharing is given below in Equation (4.4) as a first-order conjugate D_3^4-table. If we reorder the shares in the conjugate D_3^4-table by sorting them in lexicographical order, we obtain the sharing of Equation (4.5). Notice the symmetry in the ordering, as the indices in the first output share are complementary to the indices in the last output share. In Section 4.2.8 we describe the mask refreshing method that can decrease the amount of random bits needed to remask one output bit from 7 to 3.

$$\begin{gathered}
(x, y, z, w) \\
(0,0,0,0) \\
(1,1,0,0) \\
(0,1,1,0) \\
(1,0,1,0) \\
(0,0,1,1) \\
(1,1,1,1) \\
(0,1,0,1) \\
(1,0,0,1)\,.
\end{gathered} \tag{4.4}$$

$$\begin{aligned}
&(x,y,z,w)\\
&(0,0,0,0)\\
&(0,0,1,1)\\
&(0,1,0,1)\\
&(0,1,1,0)\\
&(1,0,0,1)\\
&(1,0,1,0)\\
&(1,1,0,0)\\
&(1,1,1,1)\,. \qquad (4.5)
\end{aligned}$$

A first-order $d+1$ sharing of the first coordinate function of PRINCE in Equation (4.1): $o^1 = 1 \oplus wz \oplus y \oplus zy \oplus wzy \oplus x \oplus wx \oplus yx$ is presented in Equation (4.6).

$$\begin{aligned}
o^1_1 &= 1 \oplus w_0z_0 \oplus y_0 \oplus z_0y_0 \oplus w_0z_0y_0 \oplus x_0 \oplus w_0x_0 \oplus y_0x_0\\
o^1_2 &= w_1z_1 \oplus w_1z_1y_0 \oplus w_1x_0\\
o^1_3 &= w_1z_0 \oplus w_1z_0y_1\\
o^1_4 &= w_0z_1 \oplus z_1y_1 \oplus w_0z_1y_1 \oplus y_1x_0\\
o^1_5 &= w_1z_0y_0\\
o^1_6 &= z_1y_0 \oplus w_0z_1y_0 \oplus y_0x_1\\
o^1_7 &= y_1 \oplus z_0y_1 \oplus w_0z_0y_1 \oplus x_1 \oplus w_0x_1 \oplus y_1x_1\\
o^1_8 &= w_1z_1y_1 \oplus w_1x_1\,. \qquad (4.6)
\end{aligned}$$

Continuing for the second bit's algebraic function $o^2 = 1 \oplus yw \oplus yz \oplus xz \oplus yzw \oplus xyz$ the optimal sharing is:

$$\begin{aligned}
o^2_1 &= 1 \oplus y_0w_0 \oplus y_0z_0 \oplus x_0z_0 \oplus y_0z_0w_0 \oplus x_0y_0z_0 \\
o^2_2 &= y_0w_1 \oplus y_0z_1 \oplus x_0z_1 \oplus y_0z_1w_1 \oplus x_0y_0z_1 \\
o^2_3 &= y_1w_1 \oplus y_1z_0 \oplus y_1z_0w_1 \oplus x_0y_1z_0 \\
o^2_4 &= y_1w_0 \oplus y_1z_1 \oplus y_1z_1w_0 \oplus x_0y_1z_1 \\
o^2_5 &= x_1z_0 \oplus y_0z_0w_1 \oplus x_1y_0z_0 \\
o^2_6 &= x_1z_1 \oplus y_0z_1w_0 \oplus x_1y_0z_1 \\
o^2_7 &= y_1z_0w_0 \oplus x_1y_1z_0 \\
o^2_8 &= y_1z_1w_1 \oplus x_1y_1z_1 \,. \qquad (4.7)
\end{aligned}$$

The optimal sharing for the third bit with algebraic function $o^3 = w \oplus x \oplus zw \oplus xw \oplus xz \oplus xzw \oplus xyz$ is:

$$\begin{aligned}
o^3_1 &= w_0 \oplus x_0 \oplus z_0w_0 \oplus x_0w_0 \oplus x_0z_0 \oplus x_0z_0w_0 \oplus x_0y_0z_0 \\
o^3_2 &= w_1 \oplus z_1w_1 \oplus x_0w_1 \oplus x_0z_1 \oplus x_0z_1w_1 \oplus x_0y_0z_1 \\
o^3_3 &= z_0w_1 \oplus x_0z_0w_1 \oplus x_0y_1z_0 \\
o^3_4 &= z_1w_0 \oplus x_0z_1w_0 \oplus x_0y_1z_1 \\
o^3_5 &= x_1 \oplus x_1w_1 \oplus x_1z_1 \oplus x_1z_0w_1 \oplus x_1y_0z_0 \\
o^3_6 &= x_1w_0 \oplus x_1z_1 \oplus x_1z_1w_0 \oplus x_1y_0z_1 \\
o^3_7 &= x_1z_0w_0 \oplus x_1y_1z_0 \\
o^3_8 &= x_1z_1w_1 \oplus x_1y_1z_1 \,. \qquad (4.8)
\end{aligned}$$

Finally, for the fourth bit of the PRINCE S-Box and its function $o^4 = 1 \oplus z \oplus x \oplus yz \oplus xy \oplus yzw \oplus xzw \oplus xyw$ the optimal sharing is given by:

$$
\begin{aligned}
o_1^4 &= 1 \oplus z_0 \oplus x_0 \oplus y_0 z_0 \oplus x_0 y_0 \oplus y_0 z_0 w_0 \oplus x_0 z_0 w_0 \oplus x_0 y_0 w_0 \\
o_2^4 &= z_1 \oplus y_0 z_1 \oplus y_0 z_1 w_1 \oplus x_0 z_1 w_1 \oplus x_0 y_0 w_1 \\
o_3^4 &= y_1 z_0 \oplus x_0 y_1 \oplus y_1 z_0 w_1 \oplus x_0 z_0 w_1 \oplus x_0 y_1 w_1 \\
o_4^4 &= y_1 z_1 \oplus y_1 z_1 w_0 \oplus x_0 z_1 w_0 \oplus x_0 y_1 w_0 \\
o_5^4 &= x_1 \oplus x_1 y_0 \oplus y_0 z_0 w_1 \oplus x_1 z_0 w_1 \oplus x_1 y_0 w_1 \\
o_6^4 &= y_0 z_1 w_0 \oplus x_1 z_1 w_0 \oplus x_1 y_0 w_0 \\
o_7^4 &= x_1 y_1 \oplus y_1 z_0 w_0 \oplus x_1 z_0 w_0 \oplus x_1 y_1 w_0 \\
o_8^4 &= y_1 z_1 w_1 \oplus x_1 z_1 w_1 \oplus x_1 y_1 w_1 \,.
\end{aligned}
\tag{4.9}
$$

To make the differences easier to notice between the shares, each shared monomial derived from the original unshared monomial is located one under the other. Otherwise an empty space is left, indicating the absence of that particular shared monomial in the observed output share. Such a visual representation makes comparison between output shares effortless and shared monomials originating from the same unshared monomial immediately apparent.

Note that the sharing of the cubic terms is unique while there are more options for the sharings of the lower degree terms. Repetitions of lower degree terms across different output shares could be interesting as they could lead to more hardware-efficient implementations because the logic synthesizer could create a better simplification depending on the ANF and cells available in the target library. However, they can only be added an even number of times, similar to the correction terms in $td + 1$ TI.

4.2.6 Second-order secure $d + 1$ TI of the PRINCE S-Box

For the second-order secure masking of the PRINCE S-Box, we again use the algorithm described in Section 3.2 to obtain a second-order conjugate D_3^4-table. This table represents an optimal solution with 3 input shares and 27 output shares for each input/output bit of the S-Box. All output bits are refreshed using the ring remasking of Equation (2.7), requiring 108 random bits for the entire S-Box. The optimal sharing is given below in Equation (4.10) as a conjugate D_3^4-table; the same rules are used as in the previous subsection. For brevity, we omit the exact ANF equations because of the large number of output shares. We note again the symmetry in the output shares, as for each output share there exist another output share whose share indices at the same column c are equal to $(d+1) - idx$, with idx being the input share index.

$$\begin{array}{lll}
(x, y, z, w) & & \\
(0,0,0,0) & (0,0,1,1) & (0,0,2,2) \\
(1,1,0,0) & (1,1,1,1) & (1,1,2,2) \\
(2,2,0,0) & (2,2,1,1) & (2,2,2,2) \\
(0,1,1,0) & (0,1,2,1) & (0,1,0,2) \\
(1,2,1,0) & (1,2,2,1) & (1,2,0,2) \\
(2,0,1,0) & (2,0,2,1) & (2,0,0,2) \\
(0,2,2,0) & (0,2,0,1) & (0,2,1,2) \\
(1,0,2,0) & (1,0,0,1) & (1,0,1,2) \\
(2,1,2,0) & (2,1,0,1) & (2,1,1,2)\,.
\end{array} \tag{4.10}$$

4.2.7 Protected implementations of the PRINCE cipher

Figure 4.7 depicts the data path of the hardware implementation for the four protected round-based implementations of PRINCE, which use the S-Box decomposition. All the data lines have a width of $64 \times s$ bits, in which s is the number of input shares. The exception to this is the S-Box output, which has more output shares than input shares in all cases, except for the first-order $td + 1$ implementations. The RC constant output is not divided into shares since the value is public and thus known to the attacker. The RC addition is achieved by adding its value to the first share of the shared state. Aside from already mentioned first-order $td + 1$ implementations, all protected designs feature a compression layer to reduce the number of shares used in the linear operations. Remasking is applied to the S-Box output in all versions except the first-order $td+1$ TI with decomposition, as the Q_{294} sharing provided is uniform. Figure 4.7 shows the round structure of the hardware implementation with blocks in red color potentially absent in some versions as

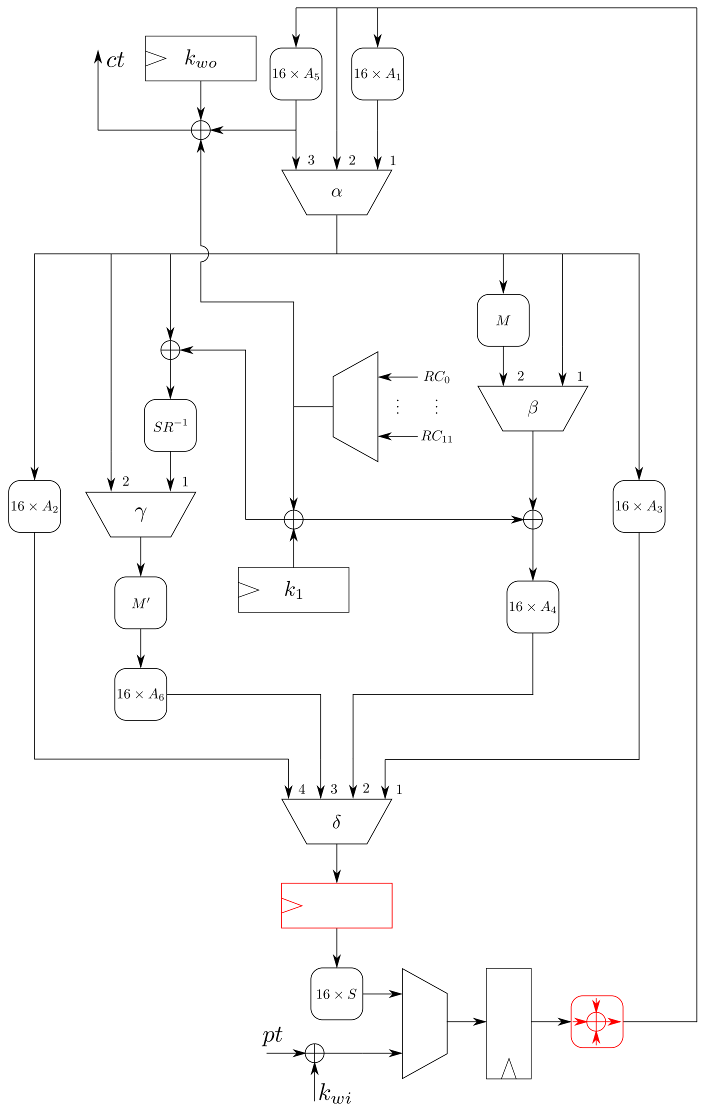


Figure 4.7: TI PRINCE round based architecture with decomposition.

discussed. Hardware implementations of combinatorial logic of Q_{294} TIs are detailed in Figures 4.2, 4.4, 4.5 and 4.6.

In order to support both encryption and decryption operations, input and output whitening keys, k_{wi} and k_{wo} are either k_0 or k_0' during encryption and decryption, respectively. We only require one extra multiplexer to implement this feature. During the S-Box evaluation the data path of the multiplexers is $\alpha_1 - \beta_2 - \delta_2$ in the first, $\alpha_2 - \delta_1$ in the second, and $\alpha_2 - \delta_4$ in the third Q_{294} computation, except in the first round where the third Q_{294} computation path is $\alpha_1 - \beta_1 - \delta_2$. Similarly, when evaluating the inverse S-Box, the active inputs of multiplexers are $\alpha_3 - \gamma_1 - \delta_3$ in the first, $\alpha_2 - \delta_1$ in the second, and $\alpha_2 - \delta_4$ in the third Q_{294} computation, except in the last round where the path during the Q_{294} computation is $\alpha_2 - \gamma_2 - \delta_3$.

For the $td + 1$ Q_{294} implementations, we use 3 and 5 shares respectively for the affine operations to reduce the amount of randomness required for the execution. This incurs an additional penalty in the area occupied by the implementation. Recall that the output of the S-Box component functions for $td + 1$ TI is shared with 3 and 10 shares respectively for the first- and second-order secure implementations. Remasking and compression are done only for the second-order $td + 1$ TI, since the first-order $td + 1$ sharing of Q_{294} is uniform. The $d + 1$ implementations use 2 and 3 shares for the first- and second-order secure implementation, respectively. The output of the S-Box component functions is shared with 4 and 9 shares for the first- and second-order secure implementations. Remasking and compression are required in both cases.

The round constant is added to only one of the shares. The key is shared with the same number of shares as the plaintext. We focus on the round-based implementation instead of the serialized one, which is more commonly described in the literature [31, 83, 58, 8, 84]. This greatly reduces the execution time at the expense of increased area and the required amount of randomness per clock cycle. In order to decrease the area, we employ multiplexers to avoid instantiating additional registers for the three stages of the S-Box evaluation. Since PRINCE has 12 rounds and each round has three stages of the S-Box evaluations, with stages taking one (for first-order $td + 1$ TI) or two (for $d + 1$ TI and second-order $td + 1$ TI) cycles, the total execution takes 36 cycles if a first-order $td + 1$ TI implementation is used or 72 clock cycles in all other proposed designs with decomposition.

Figure 4.8 represents the architecture for the four protected round-based implementations of PRINCE without S-Box decomposition. Comparing to the Figure 4.1, we can see that the two architectures are almost indentical, without taking into account that the data path in Figure 4.1 is unshared, while it is shared in Figure 4.8. An additional difference is that protected TIs without S-Box decomposition mandate the use of fresh masks during S-Box evaluation, and in $d + 1$ TI designs and second-order $td + 1$ TI design, compression is needed as well as adding a register layer before the S-Box to prevent transient leakage effects. The implementations of the NLRC layers of PRINCE S-Box

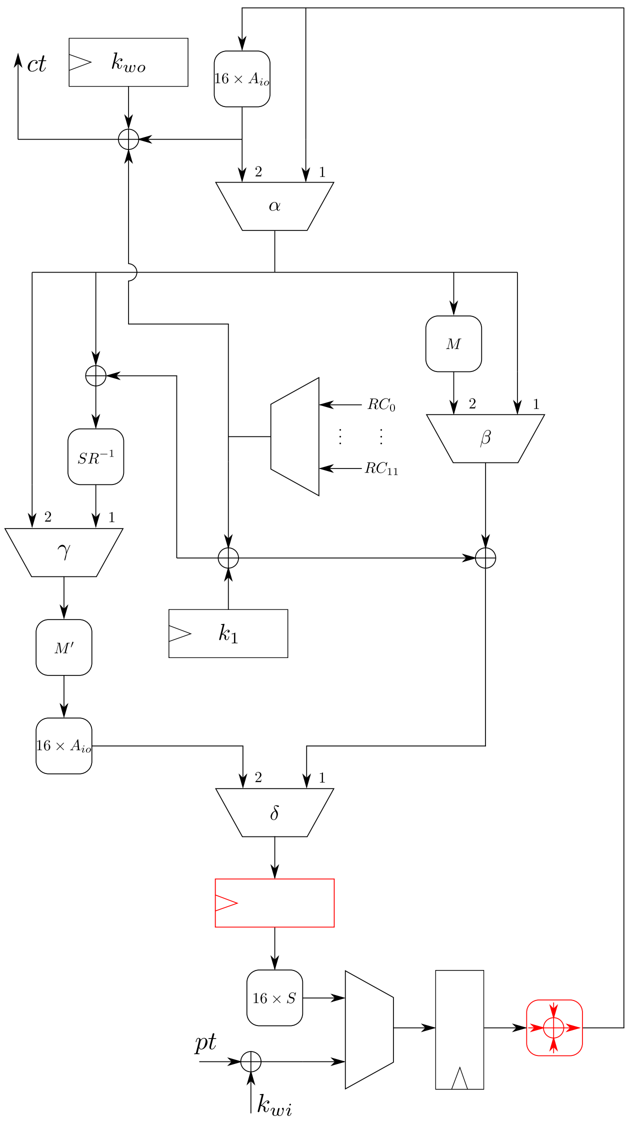


Figure 4.8: TI PRINCE round based architecture without decomposition.

are discussed in Sections 4.2.3-4.2.6.

To evaluate the S-Box, the data path of the multiplexers is $\alpha_1 - \beta_2 - \delta_1$ except in the first round where the path in the first clock cycle is $\alpha_1 - \beta_1 - \delta_1$.

Similarly, when evaluating the inverse S-Box, the active inputs of multiplexers are $\alpha_2 - \gamma_1 - \delta_2$, except in the first cycle of the last round where the path is $\alpha_1 - \gamma_2 - \delta_2$. Unlike in the unprotected version, the S-Box evaluation takes two cycles (S-Box layer and linear layer are separated into two cycles); hence it takes 24 cycles for one encryption/decryption operation. The exception is the first-order $td + 1$ implementation where the S-Box evaluation takes one cycle, making the encryption/decryption latency 12 cycles.

For $td + 1$ implementations, we use 4 and 3 shares respectively for the affine operations. Recall that the output of the S-Box component functions for $td + 1$ TI is shared with 4 and 17 shares respectively for the first- and second-order secure implementations. Compression is required only for the second-order $td + 1$ implementation, while remasking is applied for both of them. The $d + 1$ implementations use 2 and 3 shares for the first- and the second-order secure implementation, respectively. The output of the S-Box component functions is shared with 8 and 27 shares, respectively, for $d + 1$ implementations without S-Box decomposition. Remasking and compression are required in both cases.

4.2.8 Randomness reduction

The resharing of the first-order secure implementation without decomposition is performed according to the DOM [41] rules, in which complementary domains are remasked using the same randomness, with no remasking for output shares containing only one domain. It can be noticed from Equation (4.5) that output shares o_1, o_2, o_3, o_4 have complementary domains of shares o_8, o_7, o_6, o_5, respectively. If we consider 8 4-bit output shares, remasking is given by Equation (4.11)

$$\begin{aligned}
ro_1 &= o_1 \\
ro_2 &= o_2 \oplus r_1 \\
ro_3 &= o_3 \oplus r_2 \\
ro_4 &= o_4 \oplus r_3 \\
ro_5 &= o_5 \oplus r_3 \\
ro_6 &= o_6 \oplus r_2 \\
ro_7 &= o_7 \oplus r_1 \\
ro_8 &= o_8 \,,
\end{aligned} \tag{4.11}$$

in which o_i, ro_i are S-Box outputs output before and after remasking, and r_i are random 4-bit values, accounting for 12 random bits to remask the S-Box. Recombination is achieved by adding shares ro_1, ro_2, ro_3, ro_4 into one, and ro_5, ro_6, ro_7, ro_8 into another recombined share.

The randomness usage can be decreased even further if we take advantage of the structure of the PRINCE round. As explained in Section 2.3.2 the mixing layer consists of the matrices M, M' or M^{-1}. Recall that M can be obtained from M' using the nibble shuffling operation SR, i.e. $M = \mathrm{SR} \circ M'$. The 64×64 involution matrix M' is constructed as a block diagonal matrix with entries (M_0, M_1, M_1, M_0) where M_0 and M_1 are 16×16 matrices. This structure implies that 16-bit chunks of the state are processed independently. Therefore, we can use the same randomness for all four 16-bit blocks for the attacker case of $d = 1$ and $d = 2$.

Namely, if we enumerate the PRINCE state nibbles from 0 to 15, the following 4 groups can be formed: $(0, 1, 2, 3)$, $(4, 5, 6, 7)$, $(8, 9, 10, 11)$ and $(12, 13, 14, 15)$. Let R be the number of random bits needed to refresh masks of a single S-Box. Masks r_1, r_2, r_3, r_4, all containing R random bits, are reused 4 times to reshare nibbles $(0, 4, 8, 12)$, $(1, 5, 9, 13)$, $(2, 6, 10, 14)$ and $(3, 7, 11, 15)$, respectively. Hence, when evaluating the S-Boxes in a given group, the randomness required can be reused for the evaluation of the S-Boxes in the other groups. It can also be observed that the nibble shuffling

$$\mathrm{SR} : (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \rightarrow$$

$$\rightarrow (0, 5, 10, 15, 4, 9, 14, 3, 8, 13, 2, 7, 12, 1, 6, 11)$$

$$\rightarrow (r_1, r_2, r_3, r_4, r_1, r_2, r_3, r_4, r_1, r_2, r_3, r_3, r_1, r_2, r_3, r_4)$$

$$\mathrm{SR}^{-1} : (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15) \rightarrow$$

$$\rightarrow (0, 13, 10, 7, 4, 1, 14, 11, 8, 5, 2, 15, 12, 9, 6, 3)$$

$$\rightarrow (r_1, r_2, r_3, r_4, r_1, r_2, r_3, r_4, r_1, r_2, r_3, r_4, r_1, r_2, r_3, r_4)$$

does not cause mixing of the S-Boxes outputs obtained with the same randomness. This happens because 16-bit blocks newly formed groups after the SR or SR^{-1} operation still have their inputs remasked with 4 different randomness inputs. Hence, using this structure in a round-based implementation reduces the consumed randomness by a factor of four. We will use the probing model in the argumentation to showcase that side-channel resilience has not been diminished by applying randomness reuse described here.

When we consider the first-order attacker, he can probe one share out of two at a given cycle, thus the reuse of randomness is not exploitable. For the case of the second-order attacker, he is able to get either 2 shares out of 3 of one nibble or 1 share of the 2 nibbles using the same randomness at a given cycle. In the first case, again the attacker cannot exploit the reuse of randomness, since he does not know anything about this second value which can be combined with his own knowledge for the first value. In the second case, the attacker is unable to mount a bivariate attack using points from different rounds (and hence cycles) due to the remasking after each operation and the key addition (all of them done in the same cycle), and since the nibble shuffling does not cause mixing of the S-Box outputs in the same round.

4.2.9 Implementation Results

To demonstrate our results, we first use the 90 nm CMOS library provided by TSMC and consider the worst PVT corner case (a temperature of +125° C and a supply voltage of 1.0 V). The worst corner case is used almost exclusively in industrial applications. Conversely, scientific publications tend to report typical corner case, which yields a more optimistic estimate of what would be practically viable. In order to have a fair comparison and emphasize the difference between typical and worst case, we have also synthesized our designs as well as the previously existing TI PRINCE implementation by Moradi and Schneider [59] using the TSMC 90 nm library for the typical case of +25° C. Moradi and Schneider [59] provided their implementations, allowing for an apple to apple comparison of the best designs using the same compiler and library, as the synthesis results presented in this section for design presented by Moradi and Schneider [59] differ from the original paper.

For synthesis, we use the Cadence Encounter RTL Compiler version 14.20-s034 to evaluate the proposed architectures. The designs are synthesized using an operating frequency of 10 MHz, and the power consumption is estimated by simulating a back-annotated post-synthesis netlist with 100 random test vectors using the Cadence Incisive Enterprise Simulator version 15.10.006. The energy estimation is calculated for one complete encryption/decryption operation. Table 4.1 shows the area, power and energy consumption, the number of random bits required per clock cycle, and the maximum frequency for all the hardware implementations measured at the worst PVT corner case. All designs have their unconstrained critical paths well below 100 ns. Thus collecting area figures and power/energy consumption at the frequency of 10 MHz guarantees a fair comparison. The maximal operating frequency and the minimal latency are presented for the fully constrained implementations, which have significantly larger area than their timing unconstrained counterparts.

Table 4.1: Area/power/energy/randomness/latency/max frequency comparison at worst case PVT.

PRINCE	Area[1]	Power[1]	Energy[1]	Rand/ Cycle	Clock #	$f_{\max}$	Latency @ $f_{\max}$
	(GE)	(µW)	(pJ)	(bits)	(cycle)	(MHz)	(ns)
Unprotected	3597	57	69	0	12	285	42.2
[59] 1st $(td+1)$[2]	9484	66	264	0	40	328	122
1st $(d+1)$[2,4]	8701	97	698	24	72	260	277
1st $(td+1)$[2,4]	14153	75	270	0	36	268	134
1st $(d+1)$[3,4]	11613	99	238	112	24	285	84.0
1st $(td+1)$[3,4]	31116	576	691	48	12	204	58.8
2nd $(d+1)$[2,4]	13421	161	1159	72	72	250	288
2nd $(td+1)$[2,4]	18767	232	1670	40	72	243	296
2nd $(d+1)$[3,4]	32444	374	898	432	24	292	82.2
2nd $(td+1)$[3,4]	177647	1533	3679	352	24	282	85.1

[1] area, power and energy figures given at 10 MHz operating frequency
[2] with S-Box decomposition
[3] without S-Box decomposition
[4] designs presented in Section 4.2.7

The area, power and energy consumption of the PRNG are not included in Table 4.1 and Table 4.2, thus making the obtained results favoring solutions with more randomness. In practice, one must take the impact of PRNG into account since it is expected that higher throughput PRNGs consume more area, power and energy. However, in most security applications, PRNG is a component shared between multiple resources, making its impact on the overall area, power and energy consumption limited.

As expected, the first-order $d+1$ TI design with S-Box decomposition occupies the smallest area compared to other secure implementations. Compared to the first-order $td+1$ TI architecture with S-Box decomposition, this comes at the cost of extra randomness required.

We report an interesting observation when comparing the energy consumption of different architectures. The smallest energy consumption of 238 pJ has been achieved for the first-order secure $d+1$ TI architecture without S-Box decomposition presented here. This is closely followed by design from Moradi and Schneider [59] with S-Box decomposition 264 pJ. We attribute its low power consumption to the absence of randomness needed for resharing in this specific design, despite the area of both versions of first-order $td+1$ TI architectures with S-Box decomposition being larger compared to several other designs in Table 4.1. The absence of randomness greatly reduces the switching activity of the circuit lowering the power consumption considerably. Another interesting observation is that the first-order secure designs consume considerably less energy compared to second-order designs.

For second-order designs, those without S-Box decomposition lead to large area overheads (particularly in the $td+1$ scenario) and a high number of random bits consumed during remasking compared to simpler designs. We conclude that the $d+1$ designs are still interesting implementation choices if enough randomness can be provided to ensure side-channel resilience. Second-order $td+1$, on the other hand, seems quite unpractical due to its large area overheads and considerable power and energy consumption.

One can see that all protected designs except first-order $td+1$ without S-Box decomposition have their maximal frequency within 20% of each other. The reason for the first-order $td+1$ without S-Box decomposition smaller maximum frequency is the absence of the register before the S-Box operation. Also, the implementation by Moradi and Schneider [59] has a smaller critical path compared to our designs. The critical path for all implementations goes from the round counter to the S-Box input register. For the first-order $td+1$ without S-Box decomposition, we do not have the S-Box input register, making the critical part longer. Still, even with this limitation, the $td+1$ first-order version achieves smaller total latency compared to other designs.

Compared to the designs presented in Section 4.2.7, the design described by Moradi and Schneider [59] stores the key in an unshared register, requiring less area for key storage. This may lead to vulnerabilities to template-based side-channel attacks, but it certainly reduces the area, power, and energy consumption. Also, the authors proposed a more efficient affine transformation of decomposed S-Boxes, and the architecture has simplified interface and control logic. That is why their design is considerably smaller and has lower power consumption than the first-order $td+1$ version of our proposal with S-Box decomposition. When the energy consumption is compared, the two designs perform similarly with the design of Moradi and Schneider [59] being 2.3% more efficient. This is because our first-order $td+1$ design with S-Box decomposition is 10% faster in terms of the required number of clock cycles.

Another interesting observation is that our first-order $td+1$ design with S-Box decomposition has lower power consumption than four other designs from Table 4.1, while having larger area. As discussed previously, this is because no additional randomness is required during the encryption/decryption process. A quick experiment with the $d+1$ TI PRINCE without S-Box decomposition in which random inputs are all set to zero shows that the mask refreshing accounts for a significant amount of the total power consumption. Namely, the power/energy consumption drops by 40% if the random inputs are set to

Table 4.2: Area/power/energy/randomness/latency/max frequency comparison at normal case PVT.

PRINCE	Area (GE)	Power (uW)	Energy (pJ)	Rand/ Cycle (bits)	Clock # (cycle)	$f_{\max}$ (MHz)	Latency @ $f_{\max}$ (ns)
Unprotected	3596	57	68	0	12	381	31.5
[59] 1st $(td+1)$ [1]	9502	66	264	0	40	421	95.1
1st $(d+1)$ [2]	11634	100	241	48	24	379	63.3
2nd $(d+1)$ [2]	32477	364	874	1728	24	375	64.0

[1] with S-Box decomposition
[2] without S-Box decomposition

zero. Hence, when achieving lower power/energy is a major requirement using uniform $td+1$ sharing is the best approach if such a sharing can be found. Also, we note the conflicting nature of designing hardware-efficient side-channel protected circuits. We prefer additional switching activity to hide the useful signal from the attacker, but at the same time, the added switching largely contributes to the power consumption of the circuit.

Table 4.1 also clearly shows the difference between power and energy consumption. The most extreme example is the comparison between the first-order $td+1$ design without S-Box decomposition and the $d+1$ design with S-Box decomposition. Although the $td+1$ design without S-Box decomposition has almost 6 times the power consumption, it has slightly smaller energy consumption, as it takes 6 times fewer clock cycles to complete.

As can be seen by the reported figures, adding side-channel countermeasures increases the size by at least a factor of 2.5 compared to the unprotected PRINCE. One has the penalty of extra clock cycles as well in all the cases except the first-order $td+1$ without S-Box decomposition version. However, even in that particular implementation, the minimal latency is higher compared to the unprotected design due to its longer critical path.

The fastest unprotected PRINCE with worst-case PVT synthesis takes 42.2 ns, followed by the first-order $td+1$ TI without decomposition, which takes 58.8 ns, i.e., a 39% latency increase; next is the second-order $d+1$ TI without decomposition, which takes 82.2 ns, i.e., an additional 41% latency increase. Moreover, all designs without S-Box decomposition have significantly smaller latency compared to the implementation by Moradi and Schneider [59], ranging from 1.4 to 2 times less delay.

Table 4.2 shows the area, power and energy consumption, the number of random bits required per clock cycle and the maximum frequency for 3 hardware implementations, one given by Moradi and Schneider [59], and two $d+1$ designs without S-Box decomposition all measured at the typical PVT case.

Again, at the maximum frequency, our first-order design surpasses the previous state of the art by reducing the latency by almost a third. The energy consumption of our first-order at the frequency of 10 MHz is almost 10% lower. On the other hand, the implementation of Moradi and Schneider [59] beats our version with respect to area, power consumption, maximal running frequency, and randomness required during the remasking process. It can achieve higher throughput, with minor modifications to the finite state machine, so it processes three messages at once. Given that our goal was to minimize implementation latency, these results are not surprising. As was expected comparing the same designs using the typical and the worst-case corner case, we observe that the maximal frequency has increased by about a third when using the typical corner case. The area, power, and energy values differ only insignificantly because they are reported for a running frequency at 10 MHz, i.e., unconstrained timing. Since the same library is used, it is expected that both typical and worst-case corner case would synthesize to a similar minimal area.

4.2.10 Side-channel evaluation

We first provide an evaluation of the first-order PRINCE without S-Box decomposition using optimal $d+1$ sharing described in Section 3.2 which design was programmed into a Xilinx Spartan-6 FPGA. The platform used is a Sakura-G board. The design is separated into two FPGAs to minimize the noise: one performs the PRINCE encryption, and the second FPGA handles the I/O and the start signal. Our core runs at a low frequency of 3.072 MHz, while the sampling rate is 500 million samples per second. Since one trace consists of 2500 points, we can cover the first seven rounds of the execution. The power waveform is given in Figure 4.9.

We performed a non-specific leakage detection test [25] on the input plaintext following the standard methodology [73], and the resulting t-test graphs are shown in Figure 4.10. Initially, the PRNG is turned off to verify the validity of the setup, and leakage is clearly detected with one million traces. The left-hand side in the Figure 4.10 demonstrates a substantial first-order leakage during the loading of the plaintext and the key. This can be attributed to the one share of both the key and the plaintext being equal to the unshared value, while the other share is zero. Another strong peak is during the first S-Box execution as there is still a high correlation to the input in the PRINCE state in the first round. Leakage is also present in later rounds as well due to lack of additional

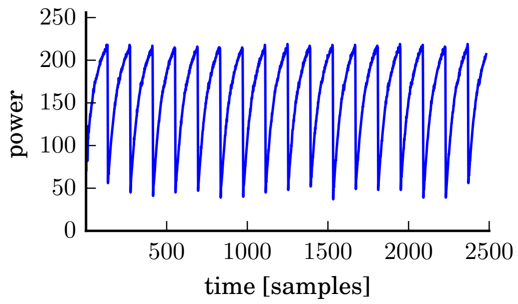


Figure 4.9: Example power trace waveform used to perform the t-test on first-order PRINCE.

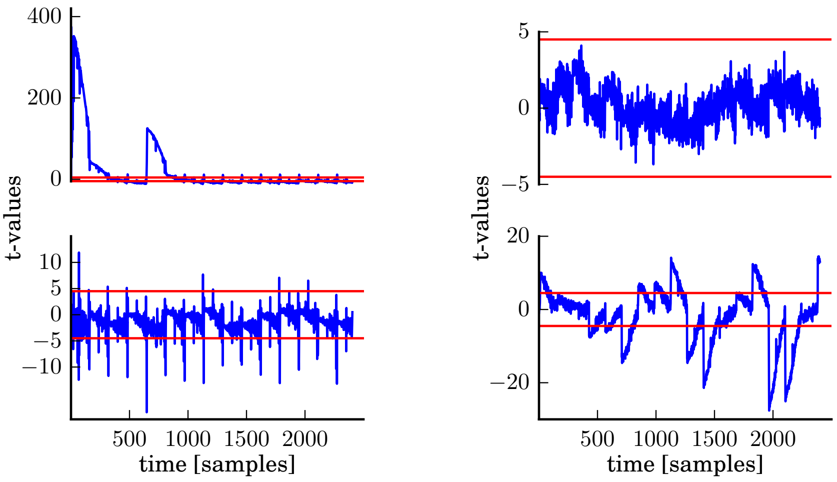


Figure 4.10: Leakage detection test results on first-order PRINCE. PRNG off (left) and PRNG on (right). First- (top) and second- (bottom) order t-test results.

randomness, although it becomes smaller. Second-order leakage can also be observed when the masks are off. When the PRNG is on, no first-order leakage is detected after 100 million traces, while second-order leakage is present as expected.

Due to the size and the randomness needed, the second-order design did not

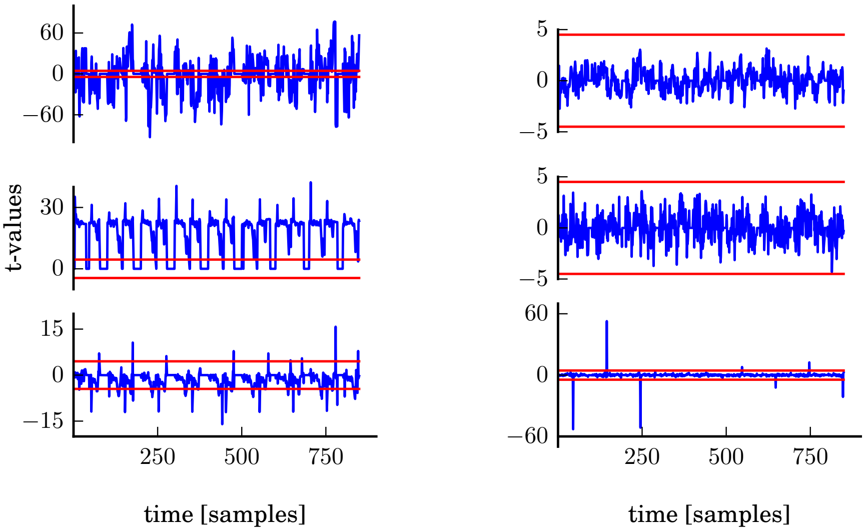


Figure 4.11: Leakage detection test results on second-order PRINCE. PRNG off (left) and PRNG on (right). First, second and third-order (top - middle - down) t-test results.

fit on the same FPGA board. Instead, the design is tested against simulated power traces. We measured the estimated power consumption by running a post-synthesis simulation with a back-annotated netlist. Input-to-output timing delays and the current consumption of every gate in the netlist were taken into account and modeled as specified by the technology liberty timing file. In our simulations, one clock cycle is represented by 50 sample points and the first seven rounds of the execution are covered. One million traces have been obtained with PRNG switched on, and 2 000 traces with PRNG off. Simulated traces are perfectly aligned, they do not contain any measurement noise, and numerical noise of the samples is minimized by having a precision of 32-bit floating point representation compared to 8-bit obtained from the FPGA setup.

The second-order implementation t-test results are shown in the Figure 4.11. We note that with PRNG off, leakage occurs in all orders with only 2 000 traces. With PRNG on, the design is leakage-free in first- and second-order, while several points leak in the third order. More precisely, third-order leakage occurs during the writing of the S-Box output to the register every other cycle.

4.3 Conclusion

As discussed in the work of Moradi and Schneider [59] designing low-latency side-channel protection in general, and for PRINCE block cipher in particular, has been identified as an open problem. In this chapter, we have shown the fastest round-based first- and second-order secure implementations of PRINCE using $td + 1$ and $d + 1$ TI sharing correspondingly, leveraging the sharing techniques from Chapter 3. Additionally, we showed how low cycle latency could lead to an energy-efficient design by demonstrating the most energy-efficient round-based first-order secure implementation of PRINCE using a $d + 1$ TI sharing. We have investigated several trade-offs that occur in side-channel secure designs. Particularly, we discuss the energy consumption of the implementations, an important factor in several applications, such as battery-powered devices.

We reported, evaluated, and compared hardware figures for eight different TI protected round-based versions of the PRINCE cipher, namely $d + 1$ and $td + 1$ TI versions, first- and second-order secure, with or without the S-Box decomposition. The $td+1$ TI versions tend to consume less randomness. The $d+1$ TI versions with decomposition achieve lower area and power consumption. The first-order designs without decomposition have favorable energy consumption. The comparison with the state of the art showed that our designs have more than 30% lower latency compared to the architecture presented by Moradi and Schneider [59] while the energy consumption is lower by about 10% It should, however, be noted that the previous TI design of PRINCE [59] still has the highest power efficiency reported in the literature while using more hardware efficient affine transformations to achieve decomposition.

As can be seen from the investigated TI designs of PRINCE cipher, many factors attribute to the characteristics of the final design in different ways. TI-protected functions with high algebraic degree reduce the final clock count and the latency and energy consumption during one operation. Conversely, the associated increased circuit complexity burdens both the area and the critical path, negatively impacting energy consumption and latency, respectively. A hardware designer must consider all these parameters since the optimal design choice heavily depends on the algorithm in question, alongside the constraints imposed upon the design. Our work shows, for the case of PRINCE block cipher, to achieve low latency, it is more efficient not to perform S-Box decomposition.

Chapter 5

Low Latency Side-Channel Protected AES Solutions

"Shortcut is noxious, longer path is closer."

Serbian proverb

The work presented in this chapter is based on findings published in [15] in which I was the main author. It also contains previously unpublished results from joint work with Danilo Šijačič, presented in Sections 5.1, 5.2, and 5.3. My contribution was the idea to test different ANF term distributions, and realization of the two single cycle masked AES S-Box implementations.

Low-latency masked implementations present a considerable challenge, as was discussed already in previous chapters. The biggest and most interesting challenge of low-latency side-channel protected design is an AES implementation due to the high algebraic degree of its S-Box. With the algebraic degree 7 of the S-Box, the minimal number of shares in the first-order $d+1$ TI implementation is 128. However, due to the complex algebraic structure, the only known implementation has 256 output shares [38], twice more than optimal. We try to improve upon this result by applying the optimal $d+1$ sharing presented in Section 3 to reduce the number of output shares to the theoretical minimum of 128. Also, we reduce the number of random bits needed for mask refreshing from 2048 down to 512. The result is the smallest reported single cycle side-channel protected AES S-Box up to date. We must indicate that, while it is possible to implement first-order $td+1$ TI AES S-Box with 8 input and output shares, the number of shared ANF terms for a single unshared monomial of degree 7 is $8^7 = 2\,097\,152$ or $252\,144$ per output share. This complexity makes it practically infeasible to use the $td+1$ approach as the critical path would become too long, and the synthesis tool would likely struggle to compile such a complex Boolean function.

The masked shares are typically given in the literature using the ANF for each share, however, we introduced a table notation in Chapter 3 in order to more succinctly present the output sharing. Nevertheless, while the table notation does allow us to evaluate the TI properties of *non-completeness* and *correctness*, it does not uniquely determine the ANF of each output share, due to the possibility to distribute shared monomials of the ANF in multiple different ways. The table notation of a sharing can be viewed as a placeholder for ANF terms. To reiterate, to satisfy the $d+1$ correctness property of TI, for each monomial of the shared ANF m_{sh}, there has to exist at least one row r in the D-table which can hold m_{sh}. But, often there exists multiple rows $r_1, \ldots, r_n$ that can hold m_{sh}. In that case, it is left to the designer to choose the output row in which the shared monomial m_{sh} will be located. The notion of redistributing shared monomials across different shares is not unknown in the scientific community. It was previously explored in $td+1$ TI strictly from a SCA resistance point of view by Bilgin et al. [9] to find a uniform $td+1$ sharing. In this chapter, however, we investigate the impact of several distribution strategies for shared ANF terms in low cycle latency TI designs. We address this issue with respect to hardware design metrics such as area, latency, as well as side-channel security. In particular, we study the impact of different ANF monomial allocations to output shares of a single-cycle AES S-Box. We demonstrate and quantify the trade-offs between area and latency in ASIC designs that can be made using commercial hardware design tools. For the given AES S-Box, we cover edge cases for both area and latency and discuss implications of such design choices on side-channel security. Combining two ANF distribution strategies with two hardware optimization goals (area/latency), we obtain results for four AES S-Box implementations that can be either as small as 21 kGE or as fast as 330 MHz in a 90 nm TSMC library.

In the end, we introduce a method to efficiently schedule the S-Box evaluation during the AES rounds in fully serialized AES implementations, allowing for the S-Box pipeline to always be full. Effectively, such a scheduling allows for the round to be completed in 20 cycles. Using a CP solver, we prove that there is such a scheduling for S-Boxes that take up to 10 cycles to complete. And for S-Box cycle latency between 6 and 10 cycles we provide an S-Box scheduling during one round that achieves 20 cycles per round.

5.1 Non-Uniqueness of TI

The TI properties listed in the Chapter 2 and the Chapter 3 are required for a secure implementation, however, they do not fully determine the ANF of the output component functions of a TI circuit. Depending on the ANF of the unshared function, several degrees of freedom are left to the designer as to in which output share to place certain ANF terms. A simple example of the first-order $d+1$ TI of the OR gate, with the ANF $c = ab + a + b$, can showcase this non-uniqueness. Two different solutions are given in Equation (5.1) and Equation (5.2).

$$\begin{aligned} c_0 &= a_0b_0 + a_0 + b_0 \\ c_1 &= a_0b_1 \\ c_2 &= a_1b_0 \\ c_3 &= a_1b_1 + a_1 + b_1\,. \end{aligned} \tag{5.1}$$

$$\begin{aligned} c_0 &= a_0b_0 + a_0 \\ c_1 &= a_0b_1 + b_1 \\ c_2 &= a_1b_0 + b_0 \\ c_3 &= a_1b_1 + a_1\,. \end{aligned} \tag{5.2}$$

The resulting hardware implementations of the four shares for these two TI solutions are given in Figure 5.1 and Figure 5.2. The sharing using Equation (5.1) produces two shares that contain three terms (c_0 and c_3), and two shares that contain one term (c_1 and c_2) in their ANF representation. In contrast, all output shares obtained using Equation (5.2) contain two terms each in their ANFs. We refer to the former one as *unbalanced*, and the latter one as *balanced* concerning the distribution of ANF terms among shares. In the unbalanced sharing, certain output shares absorb most of the shared terms, while others contain few shared terms, and in some cases, only the terms that can not belong to any other output share. In the balanced sharing, each output share contains an equal or roughly equal number of shared ANF monomials. The provided two examples clearly demonstrate that the distribution of ANF monomials among output shares can greatly influence the target circuit. It should also be noted that the choice of ANF distribution does not directly correlate with the logic depth of the circuit. While it might be intuitive that balanced strategy produces lower logic depth, that is not always true. Balanced TI OR in Figure 5.2 has a logic depth of 3, while unbalanced TI OR in Figure 5.1 has a logic depth of 2. The reason for such an outcome is that ANF representation uses XOR circuit which is more complex than simple NOR and NAND circuits in most technologies.

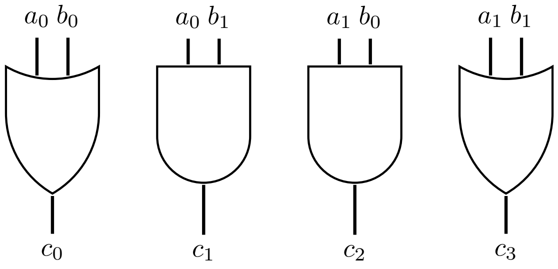


Figure 5.1: Unbalanced $d + 1$ TI OR gate

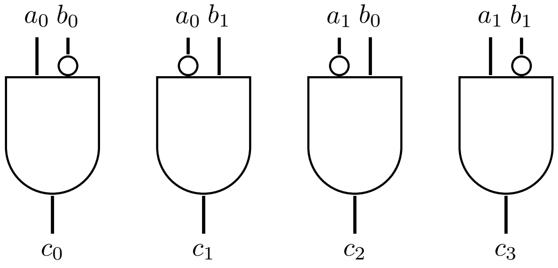


Figure 5.2: Balanced $d + 1$ TI OR gate

5.2 Hardware Design Strategies

The nonlinear layer, i.e., the S-Box, is the most challenging part in designing a Boolean masking scheme of a block cipher. First, the designer needs to decide if the appropriate route is decomposition, which yields low area and low randomness needed for remasking, or direct sharing, which yields a low latency side-channel implementation. Second, in the case of direct sharing, the designer further has to find an appropriate output sharing table, a non-trivial task, as already discussed in Chapter 3. In the AES implementation by Gross et al. [38], a sharing was derived by hand without using a more algorithmic approach, resulting in 256 output shares. Third, for the selected sharing scheme, the exact ANF of the sharing is not unique and has to be chosen.

We assume the ANF of the sharing is determined for two AES S-Boxes, and we illustrate the ramifications of different distributions of ANF monomials. Two extreme cases are presented. In the first case, ANF terms are as evenly distributed as possible, i.e., balanced. In the second case, ANF terms are distributed as unevenly as possible, i.e., unbalanced. Naturally, designers can opt for any of the intermediate options as per their needs and constraints. We resort to the edge cases to showcase their difference, and by extension, to introduce ANF monomial redistribution as a valuable hardware design parameter.

5.2.1 First-order single cycle side-channel protected AES S-Box

The AES S-Box is an 8-bit S-Box of algebraic degree 7. Hence, the $d+1$ sharing obtained using the construction method described with Algorithm 4 in Chapter 3 is optimal with respect to the number of output shares. It guarantees the minimal number of output shares, 128 in this case, and consumes 504 bits of randomness. We give the matrix representation of the 128 output shares in Appendix B. The exact ANFs of the two S-Box implementations are omitted as they have 24 302 ANF terms each across 128 output shares.

Figure 5.3 depicts the principal architecture of the chosen S-Box. 128 output shares of the S-Box and the adjacent remasking layer as constitute the majority of the combinatorial logic, followed by a 1024-bit register which also has a large area contribution. On the right-hand side of the register is the compression layer, that can be safely composed without additional register layers. The design is first-order secure, so composability problems mentioned by Moos et al. [57] affecting higher-order implementations do not apply here. Using the domain-oriented masking (DOM) remasking presented in Figure 2.10, we can reduce the number of random bits required to securely remask the sharing to 504. This is viable as the shares are again complementary, similar to shares of the PRINCE S-Box described in Section 4.2.8; complementary shares can be remasked with the same masks.

5.2.2 Implementation Results

Implementation results are summarized in Table 5.1. We use the Synopsys Design Compiler v2019.03 to synthesize the gate-level netlist. For static timing analysis, we use the Synopsis PrimeTime v2019.03 and its PX plugin for event-driven power simulation with CCS library models. We use a 90 nm TSMC library, in particular, the tcbn90lphp “flavor”.

We synthesize both S-Box options with two design goals in mind: optimized for low area and low latency. In either case, we push the design constraint to the extreme case. We report two timing paths t_1 and t_2, as shown in Figure 5.3.

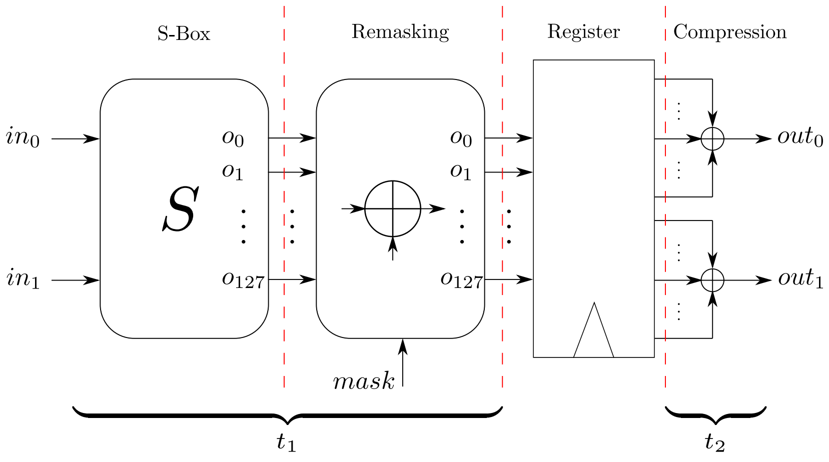


Figure 5.3: Principle architecture of the AES S-Box.

Path t_1 is the critical path dictating the maximal frequency, but t_2 cannot be neglected as it cuts into the slack of adjacent round logic. We report the average power consumption, assuming the 1 MHz operating frequency and averaging 10 000 fully annotated PrimeTime PX traces with and without randomness. Thus we illustrate the impact of remasking on power consumption. As we use post-synthesis, i.e., pre-layout, results, the design does not include a clock tree. Also, we use statistical wire-load models for timing calculation. For the area-optimized implementations, we use TSMC32K_Lowk_Conservative, and for the latency-optimized ones, we use TSMC64K_Lowk_Conservative. We set the minimal D-flip-flop DFQD1 as the driving cell of all data inputs and the loading cell of all data outputs. Lastly, all reports are generated for the worst-case process corner.

We include the first-order secure implementation reported by Gross et al. [38] in Table 5.1. However, in that work, the 90 nm Low-K UMC library is used, synthesized using a different compiler. The used process corner case was not reported. Hence the direct comparison should be taken with a grain of salt. Nonetheless, we see that the S-Box by Gross et al. [38] is nearly three times larger than the smallest design presented here. Even with the library, toolchain, and setup differences, a threefold smaller area is a strong indicator that the designs presented in this chapter are more area efficient.

Table 5.1: Implementation results for the low latency AES S-Box. t_1 is the critical path of the design, based on the combinatorial logic prior to the register. t_2 is the timing path of the output compression logic. f_{max} is the maximal frequency at which the synthesized circuit can operate. P_1 and P_2 are the average power consumption when the PRNG is turned on and off, respectively.

$\text{Design}_{\text{Goal}}$	t_1 [ns]	t_2 [ns]	f_{max} [MHz]	Area [kGE]	P_1 [μW][1]	P_2 [μW][1,2]
$\text{Balanced}_{\text{A}}$	18.58	7.20	53.83	32.37	54.52	24.88
$\text{Balanced}_{\text{L}}$	3.03	1.86	**330.03**	91.78	121.48	40.12
$\text{Unbalanced}_{\text{A}}$	15.79	7.20	63.33	**21.14**	40.22	22.02
$\text{Unbalanced}_{\text{L}}$	3.26	1.86	306.75	73.28	94.59	37.21
Related work						
Gross et al. [38][3]	2.81	N/A	356	60.73	N/A	N/A

[1] average power at 1 MHz operating frequency
[2] mask refreshing inputs set to zero during power evaluation
[3] the results are copied from original work [38] that used a different library, different technology and wire-load models

On the importance of timing constraints for reporting.

The large number of output shares in low latency S-Boxes yields a high fanout for each of the data-input drivers. Not providing design constraints, especially the realistic driving cells, while beneficial for low-area synthesis, leads to manifold inaccurate timing reports. Concretely, for our low-area designs, given the default (ideal) input drive, the reported t_1 would be 2 to 3 times smaller. Similarly, achieving good low latency results requires the synthesis tool to employ many buffer cells to mitigate the delay caused by the large fanout of input drivers. Concretely, for our low latency designs, given the default (ideal) input drive, the reported design area would be 30% to 40% smaller. These effects are highly prominent due to the unusually high fanout of the input drivers in designs such as this one, with a large number of output shares.

Similarly, generating the timing report naively using `report_timing` without additional arguments (e.g., `-from`, `-through` and `-to`), reports the timing of the default timing path. The default timing path is often the combinatorial path that ends with design outputs. In this case, this would be t_2. Thus the reported critical path would be wrongly reported if the timing report is not processed properly.

5.3 Security Evaluation

Table 5.1 shows different tradeoffs that can be made for a single low latency S-Box, as a function of two ANF distributions and two hardware optimization strategies. In this section, we perform a side-channel security evaluation to ensure that such optimizations do not compromise security. We use the TVLA [25] fixed versus random leakage detection test, with partitioning based on unshared inputs.

We perform our security evaluations based on logic simulations with the 10 ps precision using the CASCADE framework [85] to generate and process traces. This approach was computationally efficient and reliable for design-time side-channel evaluations of countermeasures implemented in a standard-cell ASIC. While simulated traces do not perfectly reflect all of the physical effects in an actual chip, we believe they are more suitable than an FPGA platform in this particular case, as we study the effect of design optimizations forASIC implementation. A similar study for FPGA platforms is also important, but we leave this for future work.

In order to validate our setup, 10 thousand TVLA traces are collected with masks set to 0. The results of the first- and the second-order TVLA are given in Figure 5.4 and Figure 5.5. As expected, all designs leak in all orders when masks are not random.

Next, we simulate 1 million traces using uniform random masks. Results of the first- and the second-order TVLA are given in Figure 5.6 and Figure 5.7. As expected, no leakage can be found in the first-order TVLA, while the second-order TVLA shows significant information leakage. Note that unbalanced designs consistently reach much higher t-values in the second order.

Additionally, we include the t-statistic evolution for one million traces in increments of 10 000 traces in Figure 5.8 and Figure 5.9. The figures show a constant trend within the $|t| < 4.5$ confidence interval for the first-order TLVA, indicating first-order security.

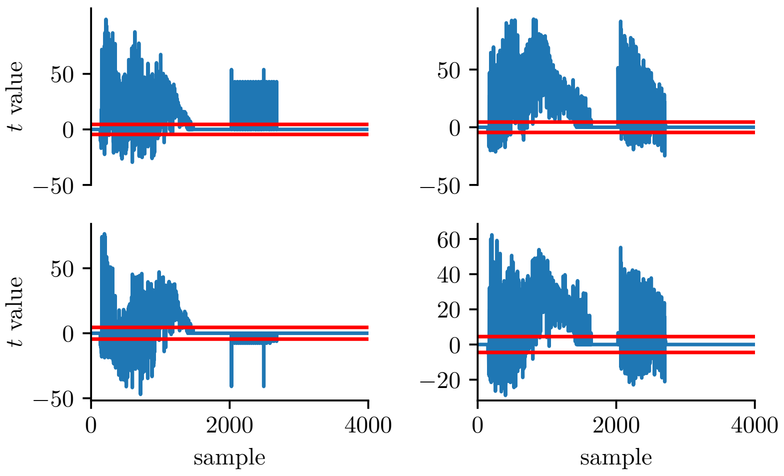


Figure 5.4: First-order TVLA (top) and second-order TVLA (bottom) using 10 000 simulated traces without randomness for low-area unbalanced (left) and balanced (right) AES S-Boxes.

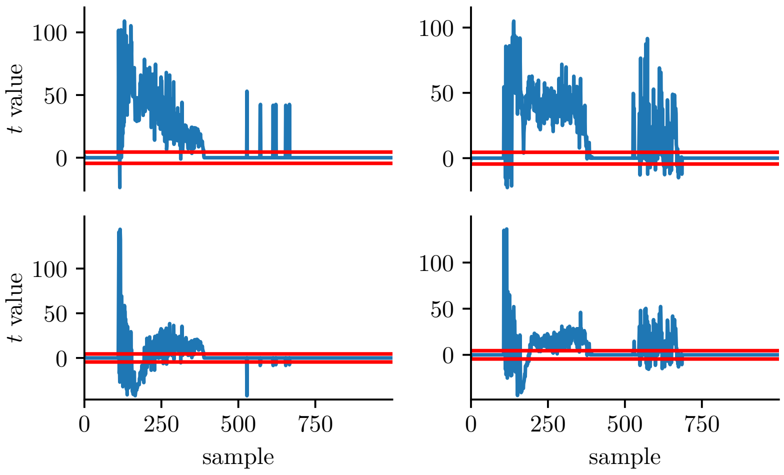


Figure 5.5: First-order TVLA (top) and second-order TVLA (bottom) using 10 000 simulated traces without randomness for low latency unbalanced (left) and balanced (right) AES S-Boxes.

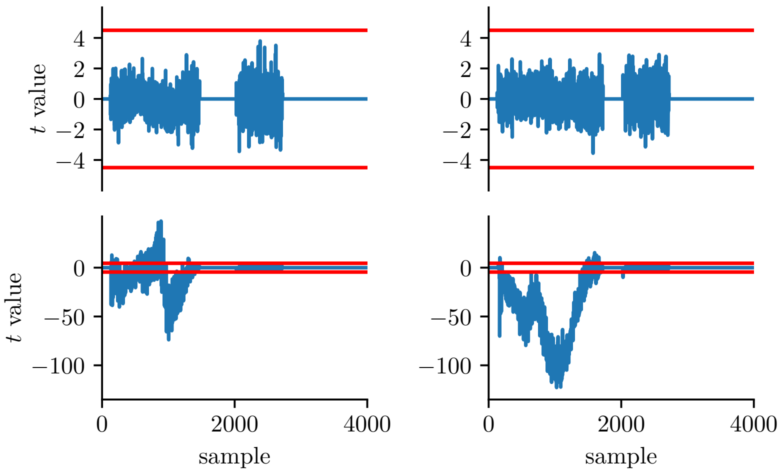


Figure 5.6: First- (top) and second-order (bottom) TVLA using 1 million simulated traces with randomness for low-area unbalanced (left) and balanced (right) AES S-Boxes.

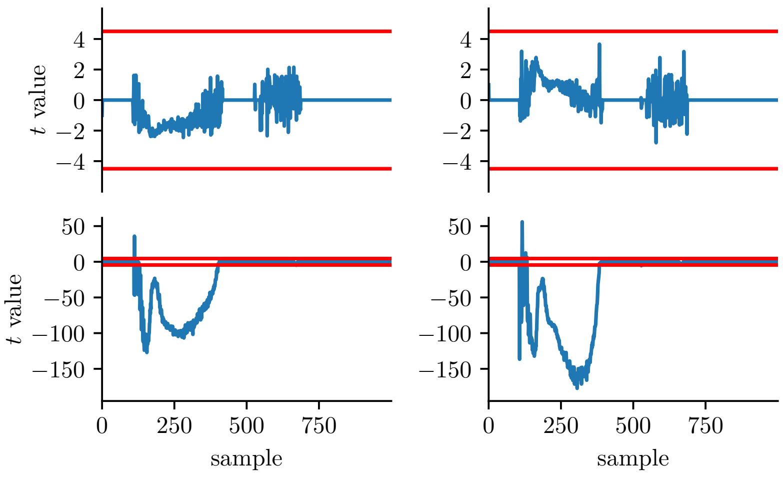


Figure 5.7: First- (top) and second-order (bottom) TVLA using 1 million simulated traces with randomness for low latency unbalanced (left) and balanced (right) AES S-Boxes.

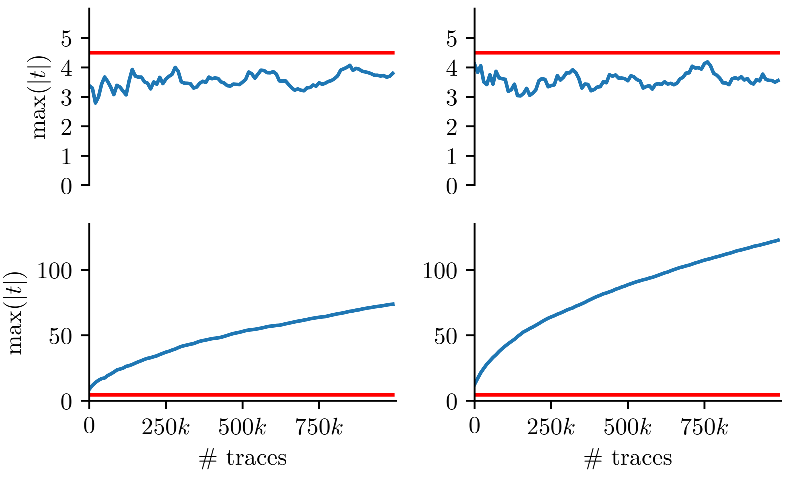


Figure 5.8: Evolution of the first-order (top) and the second-order (bottom) TVLA maximum using 1 million simulated traces with randomness for low-area unbalanced (left) and balanced (right) AES S-Boxes.

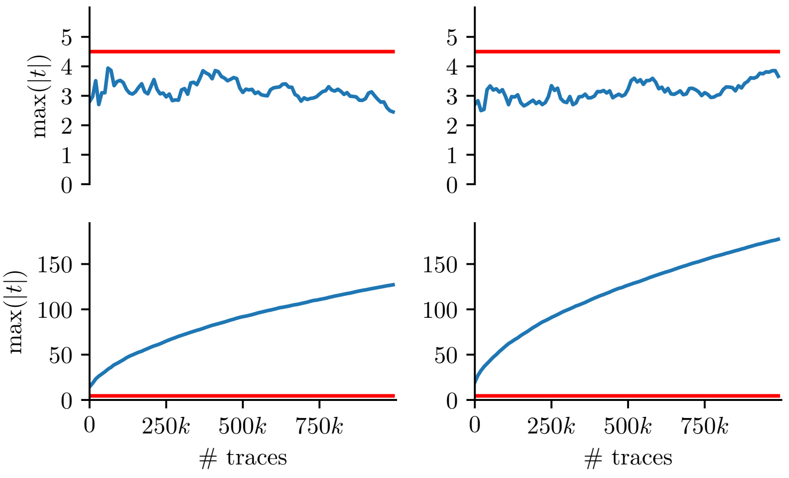


Figure 5.9: Evolution of the first-order (top) and the second-order (bottom) TVLA maximum using 1 million simulated traces with randomness for low latency unbalanced (left) and balanced (right) AES S-Boxes.

5.4 Optimizing Secure AES Schedule for Maximum Throughput

While the design of a low latency AES in Section 5.2 focused on designing the fastest possible side-channel resistant S-Box, we can also take an alternative approach when dealing with S-Box serialized AES implementations, in which a single S-Box instance computes both the SubBytes and round key update of AES key schedule sequentially. While this approach is justified, as there are multiple applications such as RFID devices, where area and power are pretty constrained, it still fares rather poorly with respect to latency and throughput. In the case of AES-128, the AES variant that is most prominent in the academic implementations, taking this approach limits the execution time asymptotically to 200 cycles per execution, with each round performing 20 S-Box operations. To ease the notation, we will refer to AES-128 simply as AES. While it is possible to use the S-Box design from Section 5.3, the area and the power consumption are quite significant for the single-cycle SCA protected S-Box implementation, even if the number of output shares is theoretically minimal. Thus the previously mentioned application, such as RFID devices, could rarely accommodate that much area or power consumption. Hence, we propose a solution focused on optimizing the S-Box evaluation schedule in a serialized implementation of AES.

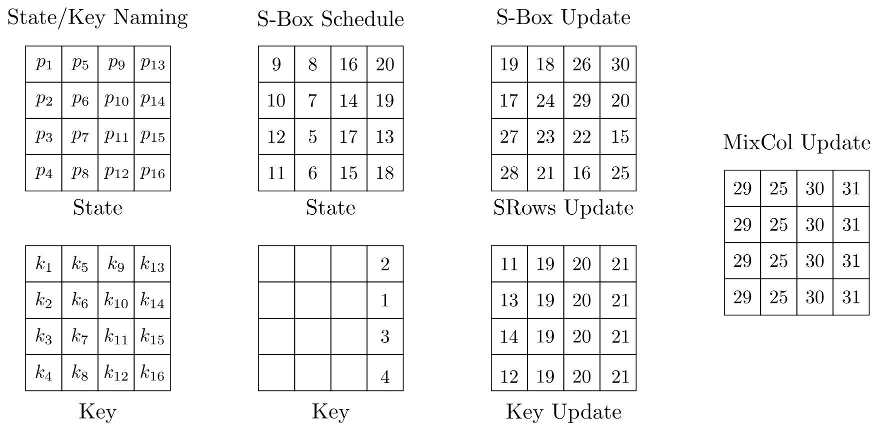
Adding the required ShiftRows, MixColumns, key schedule operations, and the several clock cycle latency of the S-Box in side-channel protected implementations increases the total latency even further. Several side-channel AES implementations [58, 8, 31, 42] require at least 246 cycles to complete one AES encryption. Recently, several approaches have been proposed that achieve the throughput of one encryption per 200 cycles while achieving latency of 216 cycles [42, 83]. But they are only feasible for S-Boxes that compute the output in 5 or fewer cycles, which is a restrictive requirement since many existing side-channel secure AES S-Box implementations take 6 or more cycles, going as far as 9 cycles [32]. Automated generation of masked hardware also frequently produces S-Boxes with high number of pipeline stages. The automatically generated implementation presented by Knichel et al. [49] takes 454 cycles to complete, while Momin et al. [56] present an implementation which can finish one encryption in 322 cycles. Here we propose a new method of scheduling that can achieve a throughput of 20 cycles per round for S-Box latency up to 10 cycles, allowing for a serialized AES implementation with the lowest possible latency. First, we declare a set of data dependencies that need to be followed in order to have a correct AES round implementation using a serialized S-Box:

- All state bytes that are to be overwritten by the S-Box output need to either be in the S-Box pipeline or to have finished the S-Box operation.
- All MixColumns input operation column bytes need to have finished the S-Box operation. However, the last byte can be collected straight from the S-Box output and not from the state registers.
- A state byte in the next round can only be used as S-Box input if the MixColumns operation for that byte's column has been performed.
- A key byte must not be updated before it is used in the current round.
- The first 4 bytes of the key during the key schedule are updated when the corresponding S-Box operation of the last 4 bytes of the key has been completed.
- Except for the first 4 bytes of the key, each subsequent byte i of the key can only be updated if byte $i-4$ has been updated.

We can program these rules in the MiniZinc constraint modeling language [61], providing the latency of the S-Box as a parameter. MiniZinc will then try to satisfy all constraints and output a solution if it exists, or if it cannot find a solution after exploring the entire search space, it will state that the problem is unsatisfiable. We have run our constrained model for different latencies, and it has always found a scheduling for a latency up to 10 clock cycles. The model is unsatisfiable for an S-Box latency of 11, proving that we cannot find a solution for any S-Box latency of 11 or more cycles.

We illustrate our solution on the 9-cycle latency S-Box that can, for example, be used to improve the scheduling of the M&M implementation [32]. The state and the key update schedule is shown in Figure 5.10, while the corresponding timing diagram can be seen in Figure 5.11. In the AES round control flow, the ShiftRows operation is performed together with the S-Box output. The output of the S-Box is written to the state byte in which it would be stored after performing the ShiftRows operation. Alternatively, it means that each column is written back diagonally with rotational wrapping around to the state matrix, not back to itself. This can be seen in Figures 5.10 and 5.11. The MixColumns operation is performed as soon as all column bytes are ready, i.e., in the 28th, 24th, 29th and 30th cycle, respectively. The result of MixColumns is ready in the following cycle. The key addition is performed in the next round iteration, and the final key addition is computed during the read-out operation. From the timing diagram in Figure 5.11 we see that the actual output for all the state bytes is ready 10 cycles after the beginning of the following round.

It should be noted that the trade-off in using our approach is an additional multiplexer at the S-Box input as we are required to read from and write to



State/Key Naming

p_1	p_5	p_9	p_{13}
p_2	p_6	p_{10}	p_{14}
p_3	p_7	p_{11}	p_{15}
p_4	p_8	p_{12}	p_{16}

State

k_1	k_5	k_9	k_{13}
k_2	k_6	k_{10}	k_{14}
k_3	k_7	k_{11}	k_{15}
k_4	k_8	k_{12}	k_{16}

Key

S-Box Schedule

9	8	16	20
10	7	14	19
12	5	17	13
11	6	15	18

State

			2
			1
			3
			4

Key

S-Box Update

19	18	26	30
17	24	29	20
27	23	22	15
28	21	16	25

SRows Update

11	19	20	21
13	19	20	21
14	19	20	21
12	19	20	21

Key Update

MixCol Update

29	25	30	31
29	25	30	31
29	25	30	31
29	25	30	31

Figure 5.10: S-Box pipeline schedule.

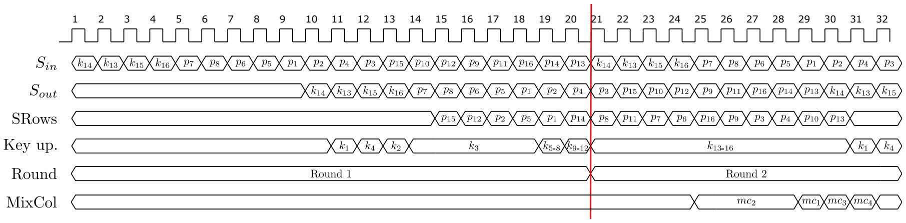

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
S_{in}
k_{14} k_{13} k_{15} k_{16} p_7 p_8 p_6 p_5 p_1 p_2 p_4 p_3 p_{15} p_{10} p_{12} p_9 p_{11} p_{16} p_{14} p_{13} k_{14} k_{13} k_{15} k_{16} p_7 p_8 p_6 p_5 p_1 p_2 p_4 p_3
S_{out}
k_{14} k_{13} k_{15} k_{16} p_7 p_8 p_6 p_5 p_1 p_2 p_4 p_3 p_{15} p_{10} p_{12} p_9 p_{11} p_{16} p_{14} p_{13} k_{14} k_{13} k_{15}
SRows
p_{15} p_{12} p_2 p_5 p_1 p_{14} p_8 p_{11} p_7 p_6 p_{16} p_9 p_3 p_4 p_{10} p_{13}
Key up.
k_1 k_4 k_2 k_3 $k_{5\text{-}8}$ $k_{9\text{-}12}$ $k_{13\text{-}16}$ k_1 k_4
Round
Round 1
Round 2
MixCol
mc_2 mc_1 mc_3 mc_4

Figure 5.11: Timing diagram of S-Box usage in the pipeline schedule.

arbitrary bytes of the state and the key matrix, which slightly increases the occupied area. However, we can achieve speed-ups of roughly 20% to 35% for existing implementations with S-Box latency greater than 6.

5.5 Conclusion

In this chapter, we investigated low latency SCA protected hardware AES implementations by analyzing single cycle $d + 1$ TI AES implementation as well as improvements in the S-Box pipeline scheduling in serialized AES implementations. The presented Unbalanced$_A$ implementation is currently the smallest single cycle SCA protected AES S-Box, with 3 times smaller area compared to the previous result by Gross et al. [38].

Additionally, we demonstrate the implementation difference of TI circuits caused by the choice of ANF during the design phase. We elaborate on two distinct strategies. The unbalanced strategy allocates ANF terms to output shares in an uneven manner, in which the first chosen output share absorbs as much shared ANF terms as possible, the next chosen output share absorbs as much of the remaining shares until all of the shared terms are associated to one of the output shares. On the other hand, the balanced strategy tries to distribute the shared ANF monomials as equally as possible across all output shares.

To quantify the effects of the two strategies, we examine a low latency AES S-Box designed using each of the two strategies. We then apply two hardware optimization strategies using commercial tools, low-area and low latency, to both the balanced and the unbalanced S-Box. Thus we create four extreme corner cases of the ASIC hardware design space. The results show that the unbalanced strategy achieves a smaller area than the balanced one, both when synthesized for lowest area and for minimal latency. The impact is significant as the area difference between the two S-Boxes is about 50% at the low area end, unbalanced S-Box requiring 21 kGE compared to 32 kGE of the balanced one. However, if the latency is of utmost importance in an application, the S-Box with balanced ANF is roughly 10% faster than the unbalanced one. The maximal frequencies are 330 MHz and 307 MHz for a balanced and an unbalanced S-Box, respectively, when synthesized for the minimal latency.

We evaluate the side-channel security of both S-Boxes using TVLA to ensure that such optimizations do not degrade the side-channel security. Our evaluation yields similar resistance levels, i.e., they are both secure against first-order attackers, both in unconstrained implementations and in low latency implementations.

Regarding the optimization of the S-Box pipeline schedule in serialized AES implementations, we show a how full pipeline can be achieved, i.e., 20 cycles to process one round, for S-Boxes with cycle latency of up to 10 cycles. The MiniZinc model shows that no 20-cycle solution exists for S-Boxes with cycle latency greater than 10. An example of scheduling that can be applied to M&M AES [32] is also presented. All scheduling solutions for the S-Box latency between 6 and 10 cycles are listed in Appendix C. Improved S-Box scheduling can also be used to reduce the number of clock cycles for masked implementations created using automated methods.

The choice of which low latency option is best is again dependent on the application. If the latency of 200 cycles per AES encryption is sufficient, optimizing the pipeline schedule of a slower S-Box in a serialized AES implementation with a single S-Box instance should be the preferred option of the designer. However, if the latency needs to be reduced maximally, a round-based implementation with single-cycle protected S-Box has to be used. In the end, there is also an option of a partial round-based design, where 2, 4 or even 8 S-Boxes are running in parallel to reduce the computation time of a single round. It is up to the designer to consider all the implications and to choose a design strategy most suitable to the target application's needs.

Chapter 6

Conclusion

“What you know is just a point of departure. So let’s move!”

Keorapetse Kgositsile

In this thesis, we have explored the means of minimizing the overhead of side-channel protection in a low-latency setting. We have developed novel construction methods aiming to provide efficient Threshold Implementations for a variety of Boolean function classes. Next, we have quantified the overhead of low-latency side-channel protection in hardware. Finally, we have provided multiple examples of low-latency masked designs.

In Chapter 3, we investigated construction methods for TI sharings of arbitrary Boolean functions, both for $td+1$ and $d+1$ flavors of TI. We introduced the $td+1$ notation of sharing via output sets and demonstrated how to analyze correctness and non-completeness properties in the output set notation. Algorithm 3 provided a construction method for $td+1$ TI sharing, which was used to generate to our knowledge the smallest second-order $td+1$ TI for cubic Boolean functions, with 8 input shares and 17 output shares. For the case of $d+1$ TI, we have established a succinct notation for the sharing and presented several construction methods of optimal or near-optimal $d+1$ TI sharings. The table notation of $d+1$ sharing can be easily used to check both the correctness and non-completeness properties of TI while avoiding the need to provide the complete ANF of a TI sharing. Algorithm 4 presented a construction that ensures a minimal number of output shares for any Boolean function of n bits of degree $n-1$, which can be used to construct a minimal sharing for any security order d. As many cryptographic primitives use n-bit S-Boxes of degree $n-1$, an optimal $d+1$ sharing construction is valuable to anyone aiming to design a $d+1$ TI of these primitives. Finally, Chapter 3 demonstrated how to transform a $d+1$ sharing table problem into a well-known discrete optimization problem of set covering. We then applied multiple discrete optimization techniques, some of which could guarantee the solution's optimality (constraint programming, mixed integer linear programming), and others that could solve larger instances (greedy heuristic, simulated annealing). Leveraging the power of discrete optimization techniques, we were able to find first-order secure optimal generic sharings and many good second-order sharings of any Boolean function of up to 8 bits. The first-order optimal sharing of cubic 8-bit functions has been used on a first-order secure AES design which decomposes the S-Box into two cubic 8-bit S-Boxes [79]. Moreover, the sharing tables of first- and second-order secure TIs of Boolean functions of up to 8 bits can be used for side-channel implementations of many other symmetric key primitives, as nearly all block ciphers use S-Boxes no larger than 8 bits.

Chapter 4 explored trade-offs during the design of side-channel protected hardware circuits. Specifically, we focused on comparing standard hardware figures of merit of power/energy, performance, and area across different side-channel protected TIs with first- and second-order security, realized using $td+1$ and $d+1$ versions of TI. The PRINCE block cipher was chosen for this case study due to its design being suitable for low-latency hardware applications. A round-based implementation of PRINCE is used for all versions of the implementations. A baseline unprotected version of PRINCE is also presented and compared to masked versions to showcase the cost of side-channel protection. Masked implementations are diversified across three orthogonal parameters: first- or second-order security, $td+1$ or $d+1$ TI, and masked S-Box realization with and without decomposition. Versions without S-Box decomposition leverage Algorithm 3 and Algorithm 4 for the TIs of the cubic PRINCE S-Box. To the best of our knowledge, the S-Box implementations without S-Box decomposition introduced in this chapter, aiming to reduce the overall latency, were not previously available in the literature. At the end of the chapter, TVLA assessments of $d+1$ implementations without S-Box decomposition are presented, indicating that the expected security level is reached in both variants. The main findings from the chapter are listed below:

- S-Box decomposition should be used when low area and low power are necessary for the target application. The smallest first-order protected version with S-Box decomposition is about 30% smaller than the smallest first-order protected design without S-Box decomposition.

- Low-latency mandates the use of masked S-Box without decomposition.

If operated at maximal frequency, the fastest implementation with S-Box decomposition takes more than twice the time to finish compared to the fastest masked version without S-Box decomposition.

- The lowest energy consumption is achieved by the masked $d + 1$ TI PRINCE without S-Box decomposition, by a small margin. While having a little less than 50% higher power consumption in comparison with $td + 1$ TI PRINCE by Moradi and Schneider [59], it finishes in fewer clock cycles, resulting in a lower final energy consumption.
- Mask refreshing accounts for up to half of the total power consumption of the circuit. Thus, power-aware implementations should strive for masked constructs that maximally reduce the number of XOR operations right before the register stage, such as uniform $td + 1$ TIs.
- The cost of side-channel countermeasures is still high when compared to the unprotected design. The smallest masked version of round-based PRINCE is two and a half times larger than the unmasked version, with the most power-efficient version still consuming 14% more power. Finally, the unprotected version of PRINCE is faster, about 40%, when clocked at a maximal frequency than the fastest side-channel protected version.
- A randomness reduction scheme suitable from PRINCE is proposed to reduce the number of random bits consumed by the masked implementation by a factor of four.

In Chapter 5 we introduced the smallest single-cycle AES S-Box, implemented using sharing obtained from Algorithm 4, with 128 output shares and two input shares. The chapter also discussed two design strategies for ANF term distribution among permitted output shares, which significantly impact the physical characteristics of the final circuit. The first strategy denoted unbalanced sharing produces a circuit that has about 50% lower area footprint compared to a circuit obtained via the balanced strategy. However, the balanced circuit can achieve 10% higher maximal frequency. Security evaluation was performed on both circuits using simulated traces, and no first-order leakage was observed in either circuit, as expected.

The second part of Chapter 5 investigated low-latency optimizations of S-Box serialized AES implementation with pipelined multi-cycle S-Box implementation, which is the typical S-Box architecture for masked AES implementations. Using a constraint programming model of execution dependencies within two rounds, we found optimal scheduling orders of S-Box input interleaved with MixColumns and ShiftRows operations for S-Box latencies of up to 10 cycles. Since most side-channel protected S-Boxes have less than 10 register stages, the obtained scheduling orders are applicable to nearly all existing masked AES S-Boxes. The cost of custom scheduling order of S-Box inputs is in additional multiplexers needed at the input of the S-Box, and in the control logic driving the selector signals of these multiplexers. The latency reduction amounts to at least 20%: round implementations of S-Box serialized designs with separate cycles for ShiftRows and MixColumn finish in 25 cycles for S-Box latencies of five cycles.

6.1 Future Work

The work presented in this dissertation advances the state of the art of low-latency side-channel protection. However, many aspects of it can be further improved and explored in more detail.

The sharing construction methods presented in Chapter 3 can be applied to many other cryptographic designs. Some primitives use arithmetic addition operation to compute their output. These primitives are notoriously difficult to mask using Boolean masking, e.g., the algebraic degree of the most significant bit and carry-out bit of an n-bit adder is equal to n and $n+1$, respectively. Thus, even a 32-bit adder poses a major challenge for straightforward Boolean masking, with only a handful of side-channel protected multi-bit adders available in the literature, implemented using the Kogge-Stone adder [77, 39]. Boolean to arithmetic and arithmetic to Boolean masking conversions have also been used to mask arithmetic operations. However, they introduce a latency overhead on both ends of the conversion, making such design extremely slow. Side-channel protected Addition-Rotation-XOR (ARX) designs, such as the SHA-2 family of hash functions, can be achieved fully with Boolean masking, with pipelined masked ripple-carry adders implemented using $d+1$ TI with sharing obtained using discrete optimization techniques, circumventing the need for masking conversion entirely. Post-quantum schemes such as Kyber [13] also use arithmetic adders. Thus its side-channel implementation would need an efficient masked adder. Side-channel implementation of an arithmetic multiplier could also be realized entirely in the Boolean masking domain. Such a multiplier could be used inside of a side-channel protected ALU.

Another interesting extension of this work is the investigation and improvement of SCA resistant software implementations, both on algorithmic and on architecture level. The current microcontroller design is aimed towards fast execution and performance. On a surface level, faster microcontroller architecture should imply faster side-channel resistant software running on the microcontroller. However, many architecture components responsible for faster execution also introduce leakage into the side-channel software implementation, mandating the use of higher-order countermeasures [2], which has a severe negative impact on the performance of the implementation. Thus, it is worthwhile to investigate microcontroller architectures that have hardened leakage points of the implementation, such as logical functions of the ALU, pipeline registers, and branch predictors. In particular, the RISCV platform is interesting as the architecture specification is fully open source and can be easily modified. Fully realized side-channel RISCV cores have already been proposed in the literature [39], but they add significant area and performance cost, which is not suitable for all applications. A middle-ground solution in which software countermeasures are running on a microcontroller with SCA hardening of critical components might be suitable in many cases. It would negate the need for higher-order software implementation while minimally impacting the performance and size of the microcontroller for general purpose use.

While examining the power consumption of several hardware implementations in Chapter 4 and Chapter 5 we have discovered that the power consumption is heavily influenced by the mask refreshing operation. For the AES S-Box from Chapter 5 turning mask refreshing off reduced the total power consumption by 70%. In the case of PRINCE, the power consumption was reduced by half also by switching off mask refreshing. Hence, reducing the number of mask refreshing operations should significantly improve the power and energy consumption. Moreover, it would reduce the complexity of the PRNG used to provide fresh masks, which would further decrease power consumption. One example where no mask refreshing is needed is uniform $td+1$ TI sharing. However, we currently have a limited understanding of how to efficiently construct a uniform TI sharing of an arbitrary function. Hence, a more thorough study of schemes with no need for mask refreshing is needed to reduce the power consumption overhead of side-channel protection.

In Chapter 5 we showcased the dependency of the sharing ANF to the area and latency of its hardware implementation. However, the two distribution strategies for shared ANF terms can be improved to provide TI circuits that provide even more efficient implementations regarding area or latency. A better strategy would involve a more elaborate metric instead of the number of ANF terms, also focusing on logic function minimization while considering the available cells from the library used during synthesis.

Finally, due to the increased complexity of higher-order TIs, it is essential to develop automated methods that can create masked circuit components using a computer program, as manual code development is highly error-prone due to the sheer number of shared ANF terms. The increased number of shares further exacerbates the issue in low-latency SCA protected designs which typically mask Boolean functions of high degree. Moreover, an automated toolbox for side-channel code generation would facilitate widespread adoption of masked circuit design and lower the barrier of expertise needed to design side-channel hardened circuits. It would also reduce the development time during design, consequently reducing the cost of the design as well. Such a tool should leverage the table notation of the sharing, provide several shared ANF term distribution strategy options, and translate the input Boolean function into a masked circuit protected up to the security order required.

Appendix A

Sharing Tables

We provide a quick reference for the sharings of the cases examined in Section 3.3.2. To make the notation as succinct as possible, we only enumerate the chosen shares in their lexicographical order, the first share having index 0. For example if we had a sharing with $d = 2, n = 4, t = 2$ given as $[2, 12, 25, 31, 44, 45, 60, 64, 77]$, it means that the actual nine shares are

$$(0,0,0,2)\ (0,1,1,0)\ (0,2,2,1)\ (1,0,1,1)\ (1,1,2,2)$$
$$(1,2,0,0)\ (2,0,2,0)\ (2,1,0,1)\ (2,2,1,2)$$

Table A.1: Sharing indices of best shares for security order $d = 1$, for $n = 4, 5, 7, 8$ bit functions, and algebraic degree $t = 2, \ldots, n-2$.

	t	Sharing indices
$n = 4$		
	2	$(1, 6, 8, 11, 13)$
$n = 5$		
	2	$(3, 12, 20, 24, 29, 30)$
	3	$(2, 5, 8, 11, 14, 17, 20, 23, 26, 29)$
$n = 6$		
	2	$(2, 21, 30, 35, 45, 56)$
	3	$(3, 9, 10, 15, 20, 27, 36, 43, 48, 53, 54, 60)$
	4	$(1, 4, 7, 10, 13, 16, 19, 22, 25, 28, 31, 34, 37, 40, 43, 46, 49, 52, 55, 58, 61)$
$n = 7$		
	2	$(24, 43, 54, 66, 85, 109)$
	3	$(6, 25, 37, 43, 50, 60, 67, 76, 85, 90, 96, 127)$
	4	$(0, 9, 14, 21, 23, 26, 35, 36, 47, 50, 57, 60, 67, 70, 77, 80, 91, 92, 101, 104, 106, 113, 118, 127)$
	5	$(1, 6, 10, 12, 15, 16, 19, 21, 25, 30, 32, 35, 37, 41, 46, 50, 52, 55, 56, 59, 61, 66, 68, 71, 72, 75, 77, 81, 86, 90, 92, 95, 97, 102, 106, 108, 111, 112, 115, 117, 121, 126)$
$n = 8$		
	2	$(15, 64, 119, 154, 177, 236)$
	3	$(12, 27, 33, 54, 85, 106, 130, 189, 207, 216, 228, 243)$
	4	$(9, 20, 31, 36, 42, 51, 66, 71, 88, 109, 113, 126, 129, 142, 146, 167, 184, 189, 204, 213, 219, 224, 235, 246)$
	5	$(1, 6, 8, 15, 19, 28, 36, 43, 50, 53, 57, 62, 69, 74, 80, 87, 89, 94, 96, 99, 109, 110, 118, 122, 124, 127, 128, 131, 133, 137, 145, 148, 154, 159, 161, 162, 173, 174, 183, 184, 193, 198, 203, 204, 210, 221, 231, 232, 241, 244, 251, 254)$
	6	$(3, 5, 6, 9, 10, 12, 17, 18, 20, 24, 31, 33, 34, 36, 40, 47, 48, 55, 59, 61, 62, 65, 66, 68, 72, 79, 80, 87, 91, 93, 94, 96, 103, 107, 109, 110, 115, 117, 118, 121, 122, 124, 129, 130, 132, 136, 143, 144, 151, 155, 157, 158, 160, 167, 171, 173, 174, 179, 181, 182, 185, 186, 188, 192, 199, 203, 205, 206, 211, 213, 214, 217, 218, 220, 227, 229, 230, 233, 234, 236, 241, 242, 244, 248, 255)$

Table A.2: Sharing indices of best shares for security order $d = 2$ of n-bit functions of degree t part 1.

	t	Sharing indices
$n = 4$		
	2	$(2, 12, 25, 31, 44, 45, 60, 64, 77)$
$n = 5$		
	2	$(0, 49, 71, 93, 106, 110, 139, 185, 195, 199, 234)$
	3	$(0, 5, 17, 19, 31, 38, 51, 61, 64, 66, 77, 87, 94, 101, 109, 116, 117, 131, 138, 152, 153, 160, 169, 171, 183, 188, 192, 203, 205, 208, 218, 231, 238)$
$n = 6$		
	2	$(0, 41, 143, 238, 295, 369, 408, 470, 555, 580, 674, 676)$
	3	$(0, 52, 68, 104, 112, 145, 159, 178, 201, 209, 224, 269, 275, 280, 312, 331, 336, 369, 380, 416, 438, 481, 499, 534, 547, 560, 586, 611, 624, 653, 672, 676, 711)$
	4	$(7, 11, 23, 24, 32, 34, 39, 44, 46, 54, 67, 75, 80, 82, 86, 96, 103, 108, 121, 128, 132, 143, 146, 147, 158, 160, 165, 176, 178, 180, 188, 191, 199, 204, 210, 220, 222, 225, 233, 235, 246, 251, 256, 261, 268, 271, 276, 281, 292, 302, 307, 312, 317, 324, 331, 338, 344, 359, 360, 367, 372, 382, 395, 397, 402, 407, 415, 420, 427, 436, 446, 450, 458, 459, 466, 471, 482, 488, 490, 498, 503, 505, 513, 529, 536, 537, 547, 549, 554, 562, 573, 577, 588, 593, 598, 608, 609, 613, 623, 624, 637, 639, 655, 657, 671, 678, 683, 686, 700, 703, 707, 715, 719, 722, 726)$
$n = 7$		
	2	$(0, 483, 632, 679, 872, 995, 1144, 1257, 1525, 1667, 1812, 2050)$
	3	$(1, 131, 173, 222, 288, 349, 380, 445, 521, 552, 570, 611, 721, 753, 873, 877, 920, 977, 1009, 1043, 1086, 1209, 1264, 1316, 1393, 1419, 1435, 1488, 1501, 1532, 1547, 1642, 1762, 1794, 1863, 1916, 1953, 2053, 2103, 2174)$
	4	$(0, 23, 52, 65, 66, 97, 111, 118, 149, 157, 166, 186, 198, 209, 224, 255, 268, 287, 302, 315, 330, 365, 371, 379, 389, 404, 425, 439, 453, 472, 496, 515, 517, 519, 565, 572, 585, 601, 636, 665, 687, 694, 702, 725, 737, 748, 769, 807, 822, 852, 857, 871, 873, 887, 905, 919, 944, 948, 976, 999, 1022, 1043, 1064, 1069, 1088, 1099, 1110, 1152, 1176, 1190, 1200, 1213, 1224, 1258, 1272, 1285, 1289, 1297, 1322, 1334, 1344, 1363, 1383, 1399, 1409, 1441, 1473, 1490, 1503, 1513, 1541, 1564, 1583, 1614, 1629, 1633, 1653, 1669, 1690, 1694, 1703, 1738, 1740, 1761, 1777, 1805, 1813, 1833, 1845, 1866, 1870, 1880, 1882, 1901, 1931, 1951, 1955, 1965, 1997, 2012, 2038, 2040, 2052, 2087, 2098, 2108, 2145, 2149, 2164, 2184)$

Table A.3: Sharing indices of best shares for security order $d = 2$ of n-bit function of degree t part 2.

	t	Sharing indices
$n = 7$		
	5	(0, 8, 14, 19, 24, 30, 36, 43, 47, 49, 56, 58, 69, 77, 83, 90, 95, 97, 103, 109, 113, 123, 134, 135, 143, 148, 154, 156, 163, 165, 179, 184, 185, 196, 199, 201, 206, 207, 221, 222, 227, 241, 245, 250, 256, 258, 266, 276, 282, 289, 296, 300, 307, 314, 315, 325, 327, 341, 342, 349, 353, 365, 367, 372, 382, 385, 389, 393, 401, 410, 414, 421, 425, 429, 432, 436, 443, 457, 460, 467, 472, 478, 480, 481, 485, 490, 497, 502, 507, 514, 521, 527, 531, 545, 546, 552, 560, 565, 572, 573, 577, 579, 593, 597, 601, 603, 614, 616, 622, 635, 637, 645, 648, 655, 662, 667, 677, 688, 690, 698, 699, 705, 711, 719, 724, 731, 736, 741, 746, 751, 764, 767, 769, 779, 780, 784, 786, 799, 801, 808, 809, 814, 816, 820, 830, 837, 841, 849, 853, 856, 860, 871, 872, 873, 878, 888, 896, 900, 910, 915, 920, 925, 933, 940, 945, 958, 962, 965, 966, 972, 977, 982, 992, 997, 1000, 1016, 1020, 1032, 1035, 1040, 1048, 1061, 1064, 1066, 1068, 1074, 1087, 1089, 1100, 1102, 1104, 1107, 1112, 1123, 1126, 1128, 1133, 1135, 1146, 1160, 1164, 1169, 1171, 1175, 1179, 1190, 1192, 1203, 1211, 1213, 1216, 1224, 1238, 1239, 1242, 1246, 1257, 1262, 1267, 1277, 1280, 1281, 1282, 1288, 1297, 1310, 1314, 1321, 1328, 1333, 1340, 1344, 1352, 1353, 1358, 1365, 1372, 1376, 1380, 1385, 1388, 1393, 1399, 1405, 1410, 1416, 1430, 1438, 1441, 1445, 1449, 1461, 1468, 1484, 1487, 1492, 1500, 1504, 1506, 1517, 1518, 1521, 1526, 1534, 1540, 1544, 1554, 1560, 1565, 1572, 1577, 1579, 1584, 1591, 1596, 1603, 1610, 1613, 1628, 1631, 1633, 1638, 1645, 1647, 1651, 1661, 1673, 1675, 1686, 1690, 1697, 1698, 1705, 1708, 1713, 1718, 1719, 1722, 1728, 1733, 1739, 1741, 1753, 1756, 1763, 1769, 1771, 1775, 1779, 1789, 1791, 1801, 1805, 1806, 1812, 1819, 1826, 1832, 1838, 1842, 1848, 1850, 1858, 1865, 1869, 1873, 1877, 1885, 1889, 1897, 1905, 1910, 1911, 1920, 1926, 1934, 1936, 1946, 1950, 1958, 1961, 1963, 1974, 1981, 1993, 1997, 1998, 2005, 2013, 2019, 2021, 2025, 2036, 2038, 2041, 2048, 2053, 2060, 2064, 2076, 2083, 2090, 2097, 2098, 2104, 2110, 2121, 2126, 2127, 2131, 2138, 2142, 2144, 2149, 2152, 2162, 2166, 2173, 2186)
$n = 8$		
	2	(0, 1255, 1923, 2045, 2347, 3100, 3210, 3599, 3844, 4729, 4796, 4926, 5490, 5909)
	3	(33, 424, 512, 635, 740, 919, 1191, 1348, 1534, 1608, 1706, 1830, 1907, 2025, 2170, 2261, 2277, 2354, 2589, 2626, 2710, 2784, 3148, 3227, 3269, 3423, 3852, 3982, 4193, 4318, 4510, 4667, 4846, 5037, 5223, 5287, 5353, 5445, 5633, 5730, 5825, 5861, 5939, 6138, 6511)
	4	(5, 33, 92, 151, 188, 207, 238, 268, 318, 378, 446, 495, 613, 644, 682, 807, 839, 904, 954, 984, 1015, 1037, 1102, 1133, 1140, 1162, 1273, 1310, 1338, 1397, 1495, 1560, 1651, 1682, 1703, 1731, 1799, 1936, 1997, 2016, 2026, 2057, 2085, 2122, 2172, 2261, 2268, 2299, 2364, 2497, 2562, 2594, 2653, 2674, 2705, 2733, 2792, 2820, 2857, 2888, 2955, 2986, 3023, 3042, 3073, 3083, 3133, 3194, 3250, 3368, 3396, 3426, 3448, 3486, 3635, 3652, 3770, 3789, 3857, 3906, 3937, 3975, 4028, 4062, 4093, 4115, 4145, 4195, 4291, 4320, 4387, 4418, 4474, 4593, 4645, 4676, 4710, 4741, 4763, 4797, 4859, 4908, 4949, 5064, 5086, 5103, 5165, 5252, 5317, 5369, 5457, 5488, 5544, 5605, 5627, 5692, 5742, 5752, 5783, 5811, 5858, 5877, 5908, 5946, 5968, 6005, 6033, 6144, 6181, 6203, 6241, 6374, 6439, 6504, 6557)

Table A.4: Sharing indices of best shares for security order $d = 2$, functions of $n = 8$ bits and algebraic degree $t = 5$.

	t	Sharing indices
$n = 8$		
	5	$(6, 9, 21, 35, 38, 61, 76, 100, 123, 140, 152, 188, 202, 214, 216, 245, 271, 295, 309, 330, 333, 374, 388, 409, 424, 447, 450, 464, 476, 479, 502, 516, 545, 560, 569, 581, 595, 607, 619, 633, 645, 657, 669, 683, 724, 734, 775, 798, 801, 810, 834, 839, 851, 863, 865, 877, 898, 901, 939, 956, 979, 982, 994, 1005, 1020, 1037, 1049, 1070, 1084, 1110, 1125, 1146, 1163, 1175, 1213, 1239, 1256, 1268, 1270, 1303, 1318, 1332, 1358, 1394, 1397, 1408, 1420, 1434, 1446, 1449, 1471, 1485, 1497, 1538, 1550, 1562, 1573, 1614, 1623, 1635, 1652, 1667, 1678, 1690, 1693, 1719, 1745, 1759, 1771, 1795, 1807, 1809, 1821, 1833, 1838, 1862, 1871, 1886, 1900, 1923, 1952, 1955, 1967, 1981, 1993, 2004, 2007, 2028, 2072, 2095, 2098, 2107, 2133, 2157, 2174, 2204, 2207, 2218, 2259, 2280, 2297, 2323, 2347, 2356, 2359, 2382, 2385, 2397, 2414, 2426, 2452, 2466, 2478, 2492, 2516, 2528, 2531, 2542, 2554, 2568, 2583, 2616, 2633, 2647, 2659, 2673, 2697, 2714, 2726, 2728, 2740, 2752, 2761, 2787, 2802, 2819, 2840, 2852, 2881, 2904, 2929, 2941, 2943, 2955, 2972, 2984, 2996, 3020, 3034, 3046, 3057, 3081, 3096, 3122, 3136, 3162, 3174, 3206, 3229, 3241, 3291, 3308, 3320, 3329, 3344, 3355, 3358, 3370, 3381, 3384, 3413, 3436, 3439, 3477, 3486, 3498, 3501, 3515, 3530, 3553, 3556, 3577, 3591, 3620, 3644, 3651, 3680, 3695, 3709, 3730, 3742, 3745, 3768, 3771, 3785, 3797, 3809, 3859, 3873, 3885, 3890, 3902, 3916, 3940, 3942, 3954, 3966, 3978, 4004, 4007, 4045, 4066, 4069, 4080, 4124, 4135, 4150, 4173, 4190, 4238, 4252, 4266, 4278, 4302, 4314, 4328, 4331, 4343, 4354, 4369, 4375, 4387, 4425, 4442, 4463, 4478, 4492, 4504, 4515, 4518, 4530, 4539, 4568, 4583, 4606, 4620, 4632, 4649, 4661, 4664, 4687, 4690, 4711, 4723, 4725, 4778, 4790, 4813, 4839, 4854, 4883, 4897, 4920, 4923, 4959, 4985, 4988, 4999, 5023, 5035, 5047, 5049, 5061, 5078, 5102, 5112, 5138, 5164, 5179, 5200, 5214, 5226, 5258, 5267, 5279, 5291, 5293, 5317, 5331, 5343, 5348, 5372, 5386, 5400, 5436, 5448, 5462, 5465, 5488, 5491, 5503, 5527, 5550, 5553, 5567, 5593, 5608, 5619, 5631, 5634, 5648, 5660, 5672, 5684, 5722, 5736, 5757, 5801, 5815, 5827, 5849, 5852, 5863, 5875, 5878, 5889, 5904, 5913, 5937, 5954, 5968, 6016, 6030, 6056, 6059, 6082, 6123, 6140, 6152, 6161, 6187, 6202, 6225, 6237, 6249, 6261, 6278, 6290, 6292, 6304, 6330, 6342, 6347, 6397, 6406, 6409, 6432, 6447, 6461, 6464, 6476, 6497, 6511, 6523, 6552)$

Table A.5: Sharing indices of best shares for security order $d = 2$, functions of $n = 8$ bits and algebraic degree $t = 6$. (Part 1)

Shares used
(1, 12, 20, 24, 33, 38, 41, 43, 50, 54, 58, 62, 68, 75, 79, 84, 87, 90, 97, 103, 107, 108, 113, 115, 116, 119, 125, 127, 129, 137, 138, 145, 148, 159, 162, 170, 173, 175, 185, 193, 201, 207, 209, 214, 223, 226, 230, 231, 236, 245, 246, 260, 265, 277, 279, 283, 288, 289, 296, 302, 307, 321, 328, 332, 335, 336, 344, 353, 357, 361, 374, 379, 392, 393, 396, 403, 410, 414, 421, 424, 429, 433, 437, 449, 450, 453, 459, 462, 465, 470, 471, 481, 485, 494, 496, 500, 504, 511, 515, 516, 528, 535, 544, 549, 552, 556, 560, 568, 572, 582, 583, 588, 598, 605, 606, 618, 621, 628, 638, 640, 644, 651, 655, 657, 665, 668, 670, 681, 684, 689, 691, 694, 701, 704, 715, 726, 732, 738, 743, 745, 751, 758, 760, 768, 774, 782, 790, 794, 798, 802, 806, 815, 821, 825, 831, 835, 840, 847, 850, 860, 861, 865, 872, 876, 882, 892, 897, 903, 909, 917, 925, 927, 935, 938, 940, 945, 950, 958, 969, 973, 980, 987, 990, 995, 1004, 1005, 1010, 1024, 1028, 1029, 1035, 1039, 1043, 1044, 1053, 1060, 1063, 1070, 1074, 1078, 1081, 1089, 1096, 1100, 1101, 1112, 1113, 1118, 1119, 1129, 1133, 1136, 1138, 1140, 1148, 1159, 1162, 1164, 1169, 1173, 1183, 1185, 1195, 1197, 1204, 1207, 1211, 1215, 1222, 1227, 1235, 1239, 1243, 1245, 1255, 1259, 1261, 1263, 1274, 1275, 1279, 1287, 1295, 1302, 1309, 1310, 1315, 1322, 1331, 1337, 1341, 1348, 1352, 1354, 1359, 1366, 1371, 1382, 1387, 1392, 1399, 1404, 1415, 1417, 1419, 1427, 1432, 1439, 1445, 1451, 1452, 1456, 1458, 1466, 1471, 1476, 1481, 1483, 1490, 1495, 1500, 1505, 1507, 1513, 1515, 1525, 1529, 1536, 1541, 1543, 1549, 1553, 1554, 1558, 1574, 1578, 1584, 1591, 1599, 1602, 1607, 1609, 1613, 1615, 1623, 1627, 1629, 1637, 1642, 1648, 1652, 1653, 1658, 1660, 1663, 1666, 1676, 1686, 1693, 1695, 1700, 1706, 1707, 1712, 1713, 1717, 1720, 1727, 1728, 1732, 1745, 1749, 1762, 1764, 1769, 1775, 1777, 1780, 1783, 1791, 1805, 1806, 1812, 1816, 1822, 1823, 1826, 1828, 1835, 1838, 1846, 1851, 1857, 1871, 1876, 1883, 1884, 1892, 1900, 1905, 1913, 1915, 1917, 1921, 1925, 1934, 1935, 1941, 1945, 1948, 1959, 1965, 1967, 1979, 1980, 1985, 1987, 1990, 1994, 1995, 2000, 2004, 2009, 2016, 2020, 2028, 2033, 2036, 2043, 2050, 2052, 2062, 2067, 2072, 2074, 2080, 2084, 2092, 2094, 2105, 2108, 2110, 2116, 2120, 2128, 2130, 2138, 2140, 2145, 2150, 2151, 2163, 2169, 2176, 2183, 2187, 2192, 2194, 2200, 2204, 2206, 2208, 2217, 2223, 2230, 2234, 2243, 2251, 2264, 2265, 2270, 2276, 2278, 2286, 2291, 2293, 2296, 2304, 2309, 2310, 2321, 2326, 2328, 2334, 2339, 2341, 2353, 2363, 2364, 2368, 2375, 2378, 2383, 2387, 2398, 2400, 2406, 2411, 2412, 2419, 2426, 2434, 2436, 2439, 2442, 2450, 2455, 2458, 2465, 2471, 2478, 2484, 2495, 2499, 2500, 2506, 2510, 2512, 2524, 2528, 2535, 2541, 2549, 2554, 2556, 2560, 2570, 2572, 2574, 2585, 2586, 2594, 2595, 2599, 2602, 2615, 2619, 2632, 2634, 2638, 2645, 2647, 2660, 2666, 2670, 2678, 2684, 2688, 2689, 2695, 2699, 2700, 2704, 2712, 2717, 2724, 2725, 2728, 2730, 2733, 2741, 2745, 2757, 2761, 2763, 2768, 2779, 2781, 2789, 2794, 2800, 2804, 2810, 2818, 2821, 2823, 2829, 2834, 2836, 2843, 2847, 2853, 2867, 2868, 2872, 2882, 2883, 2889, 2893, 2906, 2914, 2924, 2926, 2934, 2939, 2941, 2944, 2957, 2965, 2967, 2974, 2976, 2982, 2987, 2990, 3001, 3003, 3014, 3017, 3024, 3026, 3031, 3037, 3043, 3045, 3056, 3060, 3067, 3073, 3077, 3080, 3090, 3094, 3100, 3108, 3113, 3115, 3120, 3123, 3128, 3133, 3143, 3153, 3157, 3160, 3162, 3170, 3185, 3186, 3191, 3196, 3201, 3208, 3220, 3227, 3232, 3237, 3248, 3252, 3256, 3258, 3262, 3266, 3269, 3271, 3273, 3284, 3292, 3294, 3304, 3309, 3313, 3317, 3318, 3326, 3330, 3340, 3345, 3355, 3362, 3368, 3369, 3377, 3378, 3388, 3392, 3393, 3400, 3406, 3416, 3417, 3421, 3423, 3432, 3437, 3441, 3445, 3449, 3458, 3465, 3469, 3478, 3481, 3483, 3485, 3493, 3506, 3507, 3511, 3515, 3521, 3525, 3527, 3532, 3536, 3540, 3544, 3545, 3551, 3555, 3557, 3564, 3567, 3571, 3575, 3577, 3590, 3593, 3595, 3600, 3615, 3616, 3624, 3630, 3634, 3637, 3641, 3648, 3656, 3660, 3667, 3674, 3679, 3689, 3693, 3698, 3699, 3707, 3713, 3715, 3718, 3733, 3735, 3739, 3747, 3752, 3753, 3758, 3759, 3763, 3770, 3773, 3775, 3781, 3783, 3791, 3804, 3808, 3812, 3813, 3816, 3827, 3831, 3834, 3838, 3846, 3851, 3859, 3868, 3874, 3876, 3879, 3884, 3890, 3895, 3898, 3905, 3906, 3911, 3921, 3924, 3928, 3935, 3940, 3943, 3947, 3957, 3963, 3974, 3975, 3980, 3984, 3988, 3997, 4008, 4019, 4020, 4027, 4031, 4037, 4039, 4041, 4050, 4054, 4064, 4067, 4071, 4075, 4080, 4085, 4088, 4093, 4095, 4106, 4110, 4114, 4116, 4126, 4130, 4131, 4139, 4144, 4155, 4161, 4165, 4169, 4176, 4177, 4181, 4189, 4195, 4197, 4202, 4205, 4209, 4213, 4216, 4224, 4229, 4232, 4241, 4255, 4260, 4272, 4275, 4289, 4291, 4299, 4302, 4309, 4312, 4316, 4321, 4334, 4335, 4342, 4346, 4348, 4352, 4358, 4368, 4373, 4377, 4383, 4396, 4400, 4402, 4409, 4414, 4419, 4426, 4435, 4442, 4443, 4448, 4460, 4462, 4467, 4472, 4474, 4486, 4492, 4502, 4506, 4509, 4520, 4525, 4532, 4534, 4538, 4542, 4546, 4557, 4561, 4566, 4573, 4578, 4580, 4586, 4591, 4595, 4602, 4603, 4608, 4612, 4616, 4618, 4622, 4633, 4638, 4647, 4655, 4660, 4666, 4668, 4675, 4677, 4680, 4683, 4688, 4690, 4694, 4698, 4709, 4713, 4721, 4723, 4727, 4732, 4737, 4739, 4744, 4751, 4755, 4760, 4762, 4765, 4776, 4780, 4787, 4793, 4797, 4801, 4805, 4810)

Continued on next page

Table A.6: Sharing indices of best shares for security order $d = 2$, functions of $n = 8$ bits and algebraic degree $t = 6$. (Part 2)

Shares used
(4812, 4815, 4826, 4831, 4835, 4849, 4854, 4862, 4864, 4866, 4871, 4872, 4879, 4883, 4892, 4894, 4897, 4902, 4907, 4914, 4922, 4927, 4935, 4939, 4941, 4954, 4955, 4959, 4961, 4967, 4968, 4985, 4989, 4993, 4996, 5000, 5001, 5004, 5015, 5017, 5027, 5031, 5038, 5046, 5049, 5050, 5057, 5062, 5067, 5079, 5083, 5087, 5091, 5095, 5099, 5105, 5110, 5116, 5118, 5124, 5136, 5141, 5146, 5153, 5157, 5162, 5167, 5169, 5175, 5179, 5183, 5185, 5193, 5197, 5210, 5216, 5223, 5228, 5229, 5236, 5240, 5244, 5245, 5252, 5259, 5265, 5269, 5276, 5279, 5284, 5289, 5293, 5305, 5313, 5318, 5322, 5327, 5328, 5335, 5339, 5352, 5355, 5359, 5366, 5371, 5377, 5385, 5390, 5391, 5392, 5399, 5401, 5408, 5416, 5421, 5423, 5429, 5430, 5441, 5446, 5448, 5451, 5454, 5462, 5467, 5469, 5477, 5485, 5490, 5501, 5506, 5515, 5518, 5520, 5525, 5531, 5540, 5546, 5551, 5553, 5562, 5576, 5577, 5584, 5588, 5594, 5599, 5606, 5607, 5618, 5623, 5625, 5630, 5638, 5640, 5646, 5650, 5654, 5658, 5666, 5671, 5675, 5676, 5682, 5686, 5692, 5700, 5704, 5707, 5717, 5724, 5735, 5737, 5741, 5743, 5748, 5754, 5759, 5760, 5771, 5772, 5776, 5784, 5790, 5795, 5797, 5801, 5807, 5809, 5815, 5823, 5831, 5836, 5838, 5842, 5846, 5852, 5859, 5863, 5871, 5876, 5878, 5888, 5893, 5895, 5909, 5910, 5913, 5924, 5935, 5937, 5941, 5943, 5948, 5956, 5963, 5972, 5980, 5982, 5986, 5988, 5993, 6002, 6006, 6010, 6012, 6017, 6023, 6028, 6030, 6035, 6041, 6043, 6045, 6051, 6052, 6054, 6058, 6065, 6073, 6075, 6083, 6089, 6097, 6099, 6104, 6109, 6112, 6117, 6125, 6132, 6140, 6142, 6147, 6155, 6160, 6164, 6168, 6172, 6176, 6188, 6189, 6194, 6201, 6205, 6210, 6211, 6227, 6233, 6235, 6240, 6248, 6252, 6256, 6265, 6277, 6285, 6290, 6296, 6298, 6300, 6310, 6315, 6321, 6327, 6335, 6338, 6343, 6346, 6351, 6358, 6366, 6371, 6377, 6388, 6391, 6396, 6401, 6406, 6409, 6414, 6422, 6430, 6432, 6434, 6435, 6440, 6445, 6450, 6460, 6464, 6465, 6471, 6481, 6485, 6491, 6493, 6501, 6506, 6510, 6518, 6523, 6532, 6534, 6542, 6548, 6549, 6554, 6556)

Appendix B

AES Single Cycle $d+1$ S-Box Sharing

We present a sharing used to generate a single-cycle first-order $d+1$ AES S-Box. The sharing has 128 shares, and is created using Algorithm 4, represented using D_7^8-table below.

$(0,0,0,0,0,0,0,0)$	$(0,0,0,0,0,0,1,1)$	$(0,0,0,0,0,1,0,1)$	$(0,0,0,0,0,1,1,0)$
$(0,0,0,0,1,0,0,1)$	$(0,0,0,0,1,0,1,0)$	$(0,0,0,0,1,1,0,0)$	$(0,0,0,0,1,1,1,1)$
$(0,0,0,1,0,0,0,1)$	$(0,0,0,1,0,0,1,0)$	$(0,0,0,1,0,1,0,0)$	$(0,0,0,1,0,1,1,1)$
$(0,0,0,1,1,0,0,0)$	$(0,0,0,1,1,0,1,1)$	$(0,0,0,1,1,1,0,1)$	$(0,0,0,1,1,1,1,0)$
$(0,0,1,0,0,0,0,1)$	$(0,0,1,0,0,0,1,0)$	$(0,0,1,0,0,1,0,0)$	$(0,0,1,0,0,1,1,1)$
$(0,0,1,0,1,0,0,0)$	$(0,0,1,0,1,0,1,1)$	$(0,0,1,0,1,1,0,1)$	$(0,0,1,0,1,1,1,0)$
$(0,0,1,1,0,0,0,0)$	$(0,0,1,1,0,0,1,1)$	$(0,0,1,1,0,1,0,1)$	$(0,0,1,1,0,1,1,0)$
$(0,0,1,1,1,0,0,1)$	$(0,0,1,1,1,0,1,0)$	$(0,0,1,1,1,1,0,0)$	$(0,0,1,1,1,1,1,1)$
$(0,1,0,0,0,0,0,1)$	$(0,1,0,0,0,0,1,0)$	$(0,1,0,0,0,1,0,0)$	$(0,1,0,0,0,1,1,1)$
$(0,1,0,0,1,0,0,0)$	$(0,1,0,0,1,0,1,1)$	$(0,1,0,0,1,1,0,1)$	$(0,1,0,0,1,1,1,0)$
$(0,1,0,1,0,0,0,0)$	$(0,1,0,1,0,0,1,1)$	$(0,1,0,1,0,1,0,1)$	$(0,1,0,1,0,1,1,0)$
$(0,1,0,1,1,0,0,1)$	$(0,1,0,1,1,0,1,0)$	$(0,1,0,1,1,1,0,0)$	$(0,1,0,1,1,1,1,1)$

$(0,1,1,0,0,0,0,0)$ $(0,1,1,0,0,0,1,1)$ $(0,1,1,0,0,1,0,1)$ $(0,1,1,0,0,1,1,0)$

$(0,1,1,0,1,0,0,1)$ $(0,1,1,0,1,0,1,0)$ $(0,1,1,0,1,1,0,0)$ $(0,1,1,0,1,1,1,1)$

$(0,1,1,1,0,0,0,1)$ $(0,1,1,1,0,0,1,0)$ $(0,1,1,1,0,1,0,0)$ $(0,1,1,1,0,1,1,1)$

$(0,1,1,1,1,0,0,0)$ $(0,1,1,1,1,0,1,1)$ $(0,1,1,1,1,1,0,1)$ $(0,1,1,1,1,1,1,0)$

$(1,0,0,0,0,0,0,1)$ $(1,0,0,0,0,0,1,0)$ $(1,0,0,0,0,1,0,0)$ $(1,0,0,0,0,1,1,1)$

$(1,0,0,0,1,0,0,0)$ $(1,0,0,0,1,0,1,1)$ $(1,0,0,0,1,1,0,1)$ $(1,0,0,0,1,1,1,0)$

$(1,0,0,1,0,0,0,0)$ $(1,0,0,1,0,0,1,1)$ $(1,0,0,1,0,1,0,1)$ $(1,0,0,1,0,1,1,0)$

$(1,0,0,1,1,0,0,1)$ $(1,0,0,1,1,0,1,0)$ $(1,0,0,1,1,1,0,0)$ $(1,0,0,1,1,1,1,1)$

$(1,0,1,0,0,0,0,0)$ $(1,0,1,0,0,0,1,1)$ $(1,0,1,0,0,1,0,1)$ $(1,0,1,0,0,1,1,0)$

$(1,0,1,0,1,0,0,1)$ $(1,0,1,0,1,0,1,0)$ $(1,0,1,0,1,1,0,0)$ $(1,0,1,0,1,1,1,1)$

$(1,0,1,1,0,0,0,1)$ $(1,0,1,1,0,0,1,0)$ $(1,0,1,1,0,1,0,0)$ $(1,0,1,1,0,1,1,1)$

$(1,0,1,1,1,0,0,0)$ $(1,0,1,1,1,0,1,1)$ $(1,0,1,1,1,1,0,1)$ $(1,0,1,1,1,1,1,0)$

$(1,1,0,0,0,0,0,0)$ $(1,1,0,0,0,0,1,1)$ $(1,1,0,0,0,1,0,1)$ $(1,1,0,0,0,1,1,0)$

$(1,1,0,0,1,0,0,1)$ $(1,1,0,0,1,0,1,0)$ $(1,1,0,0,1,1,0,0)$ $(1,1,0,0,1,1,1,1)$

$(1,1,0,1,0,0,0,1)$ $(1,1,0,1,0,0,1,0)$ $(1,1,0,1,0,1,0,0)$ $(1,1,0,1,0,1,1,1)$

$(1,1,0,1,1,0,0,0)$ $(1,1,0,1,1,0,1,1)$ $(1,1,0,1,1,1,0,1)$ $(1,1,0,1,1,1,1,0)$

$(1,1,1,0,0,0,0,1)$ $(1,1,1,0,0,0,1,0)$ $(1,1,1,0,0,1,0,0)$ $(1,1,1,0,0,1,1,1)$

$(1,1,1,0,1,0,0,0)$ $(1,1,1,0,1,0,1,1)$ $(1,1,1,0,1,1,0,1)$ $(1,1,1,0,1,1,1,0)$

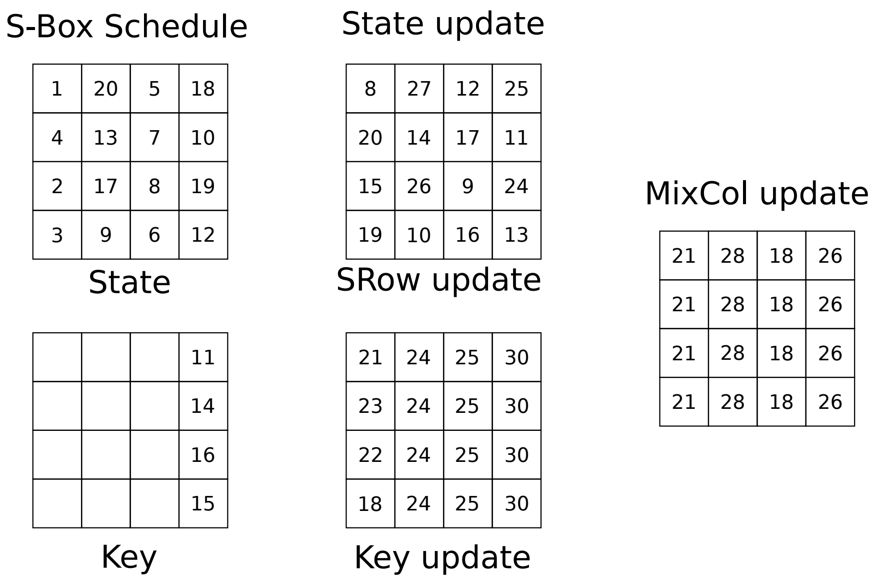
$(1,1,1,1,0,0,0,0)$ $(1,1,1,1,0,0,1,1)$ $(1,1,1,1,0,1,0,1)$ $(1,1,1,1,0,1,1,0)$

$(1,1,1,1,1,0,0,1)$ $(1,1,1,1,1,0,1,0)$ $(1,1,1,1,1,1,0,0)$ $(1,1,1,1,1,1,1,1)$

Appendix C

AES S-Box scheduling for S-Box latency of 6, 7, 8, and 10

We provide AES pipeline schedules that enable maximal throughput of 20 cycles per round as discussed in Section 5.4, for S-Box latencies of 6, 7, 8 and 10 cycles.



S-Box Schedule / State

1	20	5	18
4	13	7	10
2	17	8	19
3	9	6	12

Key

			11
			14
			16
			15

State update

8	27	12	25
20	14	17	11
15	26	9	24
19	10	16	13

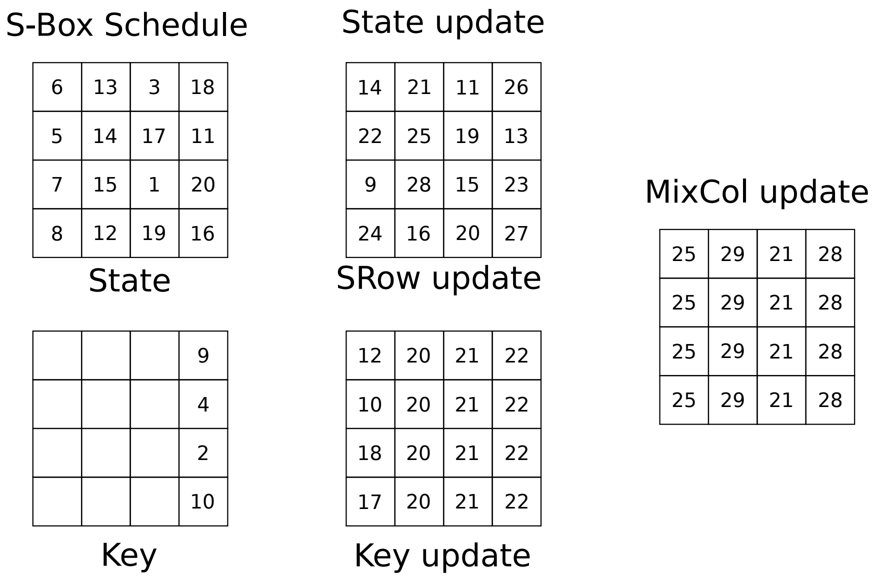
SRow update / Key update

21	24	25	30
23	24	25	30
22	24	25	30
18	24	25	30

MixCol update

21	28	18	26
21	28	18	26
21	28	18	26
21	28	18	26

Figure C.1: S-Box pipeline schedule for S-Box latency of 6 cycles.



S-Box Schedule / State

6	13	3	18
5	14	17	11
7	15	1	20
8	12	19	16

Key

			9
			4
			2
			10

State update

14	21	11	26
22	25	19	13
9	28	15	23
24	16	20	27

SRow update / Key update

12	20	21	22
10	20	21	22
18	20	21	22
17	20	21	22

MixCol update

25	29	21	28
25	29	21	28
25	29	21	28
25	29	21	28

Figure C.2: S-Box pipeline schedule for S-Box latency of 7 cycles.

S-Box Schedule

7	12	16	19
5	11	18	10
8	13	15	20
6	14	17	9

State

			4
			1
			2
			3

Key

State update

16	21	25	28
20	27	19	14
24	29	17	22
18	15	23	26

SRow update

10	19	20	21
11	19	20	21
12	19	20	21
13	19	20	21

Key update

MixCol update

25	30	26	29
25	30	26	29
25	30	26	29
25	30	26	29

Figure C.3: S-Box pipeline schedule for S-Box latency of 8 cycles.

S-Box Schedule

9	15	5	17
8	14	19	12
10	16	6	20
7	13	18	11

State

			4
			3
			1
			2

Key

State update

20	26	16	28
25	30	23	19
17	31	21	27
22	18	24	29

SRow update

14	19	20	21
12	19	20	21
13	19	20	21
15	19	20	21

Key update

MixCol update

26	32	25	30
26	32	25	30
26	32	25	30
26	32	25	30

Figure C.4: S-Box pipeline schedule for S-Box latency of 10 cycles.

Bibliography

[1] Arribas, V., Bilgin, B., Petrides, G., Nikova, S., and Rijmen, V. Rhythmic Keccak: SCA security and low latency in HW. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 1 (Feb. 2018), 269–290.

[2] Balasch, J., Gierlichs, B., Grosso, V., Reparaz, O., and Standaert, F.-X. On the cost of lazy engineering for masked software implementations. In *Smart Card Research and Advanced Applications* (Cham, 2015), M. Joye and A. Moradi, Eds., Springer International Publishing, pp. 64–81.

[3] Bar-El, H., Choukri, H., Naccache, D., Tunstall, M., and Whelan, C. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE 94*, 2 (2006), 370–382.

[4] Bertoni, G., Daemen, J., Peeters, M., and Assche, G. V. The Keccak reference, January 2011. `http://keccak.noekeon.org/` last consulted on October 21, 2022.

[5] Beyne, T., and Bilgin, B. Uniform first-order threshold implementations. In *Selected Areas in Cryptography – SAC 2016* (Cham, 2017), R. Avanzi and H. Heys, Eds., Springer International Publishing, pp. 79–98.

[6] Biham, E., and Shamir, A. Differential fault analysis of secret key cryptosystems. In *Advances in Cryptology — CRYPTO '97* (Berlin, Heidelberg, 1997), B. S. Kaliski, Ed., Springer Berlin Heidelberg, pp. 513–525.

[7] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., and Rijmen, V. Higher-order threshold implementations. In *Advances in Cryptology – ASIACRYPT 2014* (Berlin, Heidelberg, 2014), P. Sarkar and T. Iwata, Eds., Springer Berlin Heidelberg, pp. 326–343.

[8] Bilgin, B., Gierlichs, B., Nikova, S., Nikov, V., and Rijmen, V. A more efficient AES threshold implementation. In *Progress in Cryptology – AFRICACRYPT 2014* (Cham, 2014), D. Pointcheval and D. Vergnaud, Eds., Springer International Publishing, pp. 267–284.

[9] Bilgin, B., Nikova, S., Nikov, V., Rijmen, V., and Stütz, G. Threshold implementations of all 3 ×3 and 4 ×4 S-boxes. In *Cryptographic Hardware and Embedded Systems – CHES 2012* (Berlin, Heidelberg, 2012), E. Prouff and P. Schaumont, Eds., Springer Berlin Heidelberg, pp. 76–91.

[10] Bogdanov, A., Knudsen, L. R., Leander, G., Paar, C., Poschmann, A., Robshaw, M. J. B., Seurin, Y., and Vikkelsoe, C. PRESENT: An ultra-lightweight block cipher. In *Cryptographic Hardware and Embedded Systems - CHES 2007* (Berlin, Heidelberg, 2007), P. Paillier and I. Verbauwhede, Eds., Springer Berlin Heidelberg, pp. 450–466.

[11] Boneh, D., DeMillo, R. A., and Lipton, R. J. On the importance of checking cryptographic protocols for faults. In *Advances in Cryptology — EUROCRYPT '97* (Berlin, Heidelberg, 1997), W. Fumy, Ed., Springer Berlin Heidelberg, pp. 37–51.

[12] Borghoff, J., Canteaut, A., Güneysu, T., Kavun, E. B., Knezevic, M., Knudsen, L. R., Leander, G., Nikov, V., Paar, C., Rechberger, C., Rombouts, P., Thomsen, S. S., and Yalçin, T. PRINCE – a low-latency block cipher for pervasive computing applications. In *Advances in Cryptology – ASIACRYPT 2012* (Berlin, Heidelberg, 2012), X. Wang and K. Sako, Eds., Springer Berlin Heidelberg, pp. 208–225.

[13] Bos, J., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J. M., Schwabe, P., Seiler, G., and Stehle, D. CRYSTALS - Kyber: A CCA-secure module-lattice-based KEM. In *2018 IEEE European Symposium on Security and Privacy (EuroS&P)* (2018), pp. 353–367.

[14] Božilov, D., Knežević, M., and Nikov, V. Optimized threshold implementations: Minimizing the latency of secure cryptographic accelerators. In *Smart Card Research and Advanced Applications* (Cham, 2020), S. Belaïd and T. Güneysu, Eds., Springer International Publishing, pp. 20–39.

[15] Božilov, D., Knežević, M., and Nikov, V. Optimized threshold implementations: Securing cryptographic accelerators for low-energy and low-latency applications. *Journal of Cryptographic Engineering*, 12 (2022), 15–51.

[16] Božilov, D., Bilgin, B., and Sahin, H. A. A note on 5-bit quadratic permutations' classification. *IACR Transactions on Symmetric Cryptology 2017*, 1 (Mar. 2017), 398–404.

[17] Božilov, D., Nikov, V., and Rijmen, V. Design trade-offs in threshold implementations. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)* (2019), pp. 751–754.

[18] Brier, E., Clavier, C., and Olivier, F. Correlation power analysis with a leakage model. In *Cryptographic Hardware and Embedded Systems - CHES 2004* (Berlin, Heidelberg, 2004), M. Joye and J.-J. Quisquater, Eds., Springer Berlin Heidelberg, pp. 16–29.

[19] Brusco, M., Jacobs, L., and Thompson, G. A morphing procedure to supplement a simulated annealing heuristic for cost- and coverage-correlated set-covering problems. *Annals of Operations Research 86* (1999), 611–627.

[20] Canright, D. A very compact S-box for AES. In *Cryptographic Hardware and Embedded Systems – CHES 2005* (Berlin, Heidelberg, 2005), J. R. Rao and B. Sunar, Eds., Springer Berlin Heidelberg, pp. 441–455.

[21] Carlet, C. *Boolean Functions for Cryptography and Error-Correcting Codes.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2010, p. 257–397.

[22] Carlet, C. *Vectorial Boolean Functions for Cryptography.* Encyclopedia of Mathematics and its Applications. Cambridge University Press, 2010, p. 398–470.

[23] Chari, S., Rao, J. R., and Rohatgi, P. Template attacks. In *Cryptographic Hardware and Embedded Systems - CHES 2002* (Berlin, Heidelberg, 2003), B. S. Kaliski, ç. K. Koç, and C. Paar, Eds., Springer Berlin Heidelberg, pp. 13–28.

[24] Chu, G., and Stuckey, P. J. Chuffed solver description, 2014. `https://github.com/chuffed/chuffed` last consulted on October 21, 2022.

[25] Cooper, J., DeMulder, E., Goodwill, G., Jaffe, J., Kenworthy, G., and Rohatgi, P. Test vector leakage assessment (TVLA) methodology in practice. In *International Cryptographic Module Conference* (2013).

[26] Culberson, J. C., and Luo, F. Exploring the k-colorable landscape with iterated greedy. In *Dimacs Series in Discrete Mathematics and Theoretical Computer Science* (1995), American Mathematical Society, pp. 245–284.

[27] Daemen, J. Changing of the guards: A simple and efficient method for achieving uniformity in threshold sharing. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (Cham, 2017), W. Fischer and N. Homma, Eds., Springer International Publishing, pp. 137–153.

[28] Daemen, J., and Rijmen, V. *The Design of Rijndael: AES - The Advanced Encryption Standard*, 1st ed. Information Security and Cryptography. Springer Berlin, Heidelberg, 2002.

[29] Dantzig, G. *Linear programming and extensions.* Rand Corporation Research Study. Princeton Univ. Press, Princeton, NJ, 1963.

[30] De Cannière, C., and Preneel, B. *Trivium.* Springer Berlin Heidelberg, Berlin, Heidelberg, 2008, pp. 244–266.

[31] De Cnudde, T., Reparaz, O., Bilgin, B., Nikova, S., Nikov, V., and Rijmen, V. Masking AES with $d+1$ shares in hardware. In *Cryptographic Hardware and Embedded Systems – CHES 2016* (Berlin, Heidelberg, 2016), B. Gierlichs and A. Y. Poschmann, Eds., Springer Berlin Heidelberg, pp. 194–212.

[32] De Meyer, L., Arribas, V., Nikova, S., Nikov, V., and Rijmen, V. M&M: Masks and macs against physical attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2019*, 1 (November 2018), 25–50.

[33] De Meyer, L., Bilgin, B., and Reparaz, O. Consolidating security notions in hardware masking. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2019*, 3 (May 2019), 119–147.

[34] Duc, A., Dziembowski, S., and Faust, S. Unifying leakage models: From probing attacks to noisy leakage. In *Advances in Cryptology – EUROCRYPT 2014* (Berlin, Heidelberg, 2014), P. Q. Nguyen and E. Oswald, Eds., Springer Berlin Heidelberg, pp. 423–440.

[35] Eastlake, D., and Hansen, T. US Secure Hash Algorithms (SHA and SHA-based HMAC and HKDF). RFC 6234, RFC Editor, May 2011.

[36] Faust, S., Grosso, V., Pozo, S. M. D., Paglialonga, C., and Standaert, F. Composable masking schemes in the presence of physical defaults & the robust probing model. *IACR Transaction Cryptographic Hardware Embedded Systems 2018*, 3 (2018), 89–120.

[37] Gandolfi, K., Mourtel, C., and Olivier, F. Electromagnetic analysis: Concrete results. In *Cryptographic Hardware and Embedded Systems — CHES 2001* (Berlin, Heidelberg, 2001), Ç. K. Koç, D. Naccache, and C. Paar, Eds., Springer Berlin Heidelberg, pp. 251–261.

[38] Gross, H., Iusupov, R., and Bloem, R. Generic low-latency masking in hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2018*, 2 (May 2018), 1–21.

[39] Gross, H., Jelinek, M., Mangard, S., Unterluggauer, T., and Werner, M. Concealing secrets in embedded processors designs. In *Smart Card Research and Advanced Applications* (Cham, 2017), K. Lemke-Rust and M. Tunstall, Eds., Springer International Publishing, pp. 89–104.

[40] Gross, H., and Mangard, S. Reconciling $d+1$ masking in hardware and software. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (Cham, 2017), W. Fischer and N. Homma, Eds., Springer International Publishing, pp. 115–136.

[41] Gross, H., Mangard, S., and Korak, T. Domain-oriented masking: Compact masked hardware implementations with arbitrary protection order. In *Proceedings of the 2016 ACM Workshop on Theory of Implementation Security* (New York, NY, USA, 2016), TIS '16, Association for Computing Machinery, p. 3.

[42] Gross, H., Mangard, S., and Korak, T. An efficient side-channel protected AES implementation with arbitrary protection order. In *Topics in Cryptology – CT-RSA 2017* (Cham, 2017), H. Handschuh, Ed., Springer International Publishing, pp. 95–112.

[43] Gurobi Optimization, L. Gurobi optimizer reference manual, 2022. `http://www.gurobi.com` last consulted on October 21, 2022.

[44] Hensel, K. Ueber die darstellung der zahlen eines gattungsbereiches für einen beliebigen primdivisor. *Journal für die reine und angewandte Mathematik*, 103 (1888), 230–237.

[45] Ishai, Y., Sahai, A., and Wagner, D. Private circuits: Securing hardware against probing attacks. In *Advances in Cryptology - CRYPTO 2003* (Berlin, Heidelberg, 2003), D. Boneh, Ed., Springer Berlin Heidelberg, pp. 463–481.

[46] Joye, M., Paillier, P., and Schoenmakers, B. On second-order differential power analysis. In *Cryptographic Hardware and Embedded Systems – CHES 2005* (Berlin, Heidelberg, 2005), J. R. Rao and B. Sunar, Eds., Springer Berlin Heidelberg, pp. 293–308.

[47] Kambourakis, G., Kolias, C., and Stavrou, A. The mirai botnet and the iot zombie armies. In *2017 IEEE Military Communications Conference, MILCOM 2017, Baltimore, MD, USA, October 23-25, 2017* (2017), IEEE, pp. 267–272.

[48] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science 220*, 4598 (1983), 671–680.

[49] Knichel, D., Moradi, A., Müller, N., and Sasdrich, P. Automated generation of masked hardware. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2022*, 1 (Nov. 2021), 589–629.

[50] Kocher, P., Jaffe, J., and Jun, B. Differential power analysis. In *Advances in Cryptology — CRYPTO' 99* (Berlin, Heidelberg, 1999), M. Wiener, Ed., Springer Berlin Heidelberg, pp. 388–397.

[51] Kocher, P. C. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Advances in Cryptology — CRYPTO '96* (Berlin, Heidelberg, 1996), N. Koblitz, Ed., Springer Berlin Heidelberg, pp. 104–113.

[52] Krawczyk, H., Bellare, M., and Canetti, R. HMAC: Keyed-Hashing for Message Authentication. RFC 2104, RFC Editor, February 1997.

[53] Matsumoto, M., and Nishimura, T. Mersenne twister: A 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation 8*, 1 (jan 1998), 3–30.

[54] Messerges, T. S. Using second-order power analysis to attack DPA resistant software. In *Cryptographic Hardware and Embedded Systems — CHES 2000* (Berlin, Heidelberg, 2000), Ç. K. Koç and C. Paar, Eds., Springer Berlin Heidelberg, pp. 238–251.

[55] Minotra, D. A study of heuristic-algorithms for set-covering problems. Tech. rep., School of Computer Science, University of Windsor, June 2008.

[56] Momin, C., Cassiers, G., and Standaert, F.-X. Handcrafting: Improving automated masking in hardware with manual optimizations. In *Constructive Side-Channel Analysis and Secure Design* (Cham, 2022), J. Balasch and C. O'Flynn, Eds., Springer International Publishing, pp. 257–275.

[57] Moos, T., Moradi, A., Schneider, T., and Standaert, F.-X. Glitch-resistant masking revisited: or why proofs in the robust probing model are needed. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2019*, 2 (Feb. 2019), 256–292.

[58] Moradi, A., Poschmann, A., Ling, S., Paar, C., and Wang, H. Pushing the limits: A very compact and a threshold implementation of aes. In *Advances in Cryptology – EUROCRYPT 2011* (Berlin, Heidelberg, 2011), K. G. Paterson, Ed., Springer Berlin Heidelberg, pp. 69–88.

[59] Moradi, A., and Schneider, T. Side-channel analysis protection and low-latency in action. In *Advances in Cryptology – ASIACRYPT 2016* (Berlin, Heidelberg, 2016), J. H. Cheon and T. Takagi, Eds., Springer Berlin Heidelberg, pp. 517–547.

[60] National Institute of Standards and Technology. SHA-3 standard: Permutation-based hash and extendable-output functions. Tech. Rep. Federal Information Processing Standards Publications (FIPS PUBS) 202, 2015, U.S. Department of Commerce, Washington, D.C., 2015.

[61] Nethercote, N., Stuckey, P. J., Becket, R., Brand, S., Duck, G. J., and Tack, G. Minizinc: Towards a standard cp modelling language. In *Principles and Practice of Constraint Programming – CP 2007* (Berlin, Heidelberg, 2007), C. Bessière, Ed., Springer Berlin Heidelberg, pp. 529–543.

[62] Nikova, S. TI tools for the 3x3 and 4x4 S-boxes, 2012. `http://homes.esat.kuleuven.be/~snikova/ti_tools.html` last consulted on October 21, 2022.

[63] Nikova, S., Nikov, V., and Rijmen, V. Decomposition of permutations in a finite field. *Cryptography and Communications*, 11 (2019), 379–384.

[64] Nikova, S., Rechberger, C., and Rijmen, V. Threshold implementations against side-channel attacks and glitches. In *Information and Communications Security* (Berlin, Heidelberg, 2006), P. Ning, S. Qing, and N. Li, Eds., Springer Berlin Heidelberg, pp. 529–545.

[65] NXP. AN12278 LPC55S00 Security Solutions for IoT, 2020. `https://www.nxp.com/docs/en/application-note/AN12278.pdf` last consulted on October 21, 2022.

[66] Papapagiannopoulos, K. High throughput in slices: The case of PRESENT, PRINCE and KATAN64 ciphers. In *Radio Frequency Identification: Security and Privacy Issues* (Cham, 2014), N. Saxena and A.-R. Sadeghi, Eds., Springer International Publishing, pp. 137–155.

[67] Perron, L., and Furnon, V. OR-Tools, 2020. `https://developers.google.com/optimization/` last consulted on October 21, 2022.

[68] Poschmann, A., Moradi, A., Khoo, K., Lim, C.-W., Wang, H., and Ling, S. Side-channel resistant crypto for less than 2,300 ge. *Journal of Cryptology 24*, 2 (2011), 322–345.

[69] Prouff, E. Dpa attacks and s-boxes. In *Fast Software Encryption* (Berlin, Heidelberg, 2005), H. Gilbert and H. Handschuh, Eds., Springer Berlin Heidelberg, pp. 424–441.

[70] Quisquater, J.-J., and Samyde, D. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Smart Card Programming and Security* (Berlin, Heidelberg, 2001), I. Attali and T. Jensen, Eds., Springer Berlin Heidelberg, pp. 200–210.

[71] Rabaey, J. M., Chandrakasan, A., and Nikolić, B. *Digital integrated circuits- A design perspective*, 2nd ed. Prentice Hall, 2004.

[72] Reparaz, O., Bilgin, B., Nikova, S., Gierlichs, B., and Verbauwhede, I. Consolidating masking schemes. In *Advances in Cryptology – CRYPTO 2015* (Berlin, Heidelberg, 2015), R. Gennaro and M. Robshaw, Eds., Springer Berlin Heidelberg, pp. 764–783.

[73] Reparaz, O., Gierlichs, B., and Verbauwhede, I. Fast leakage assessment. In *Cryptographic Hardware and Embedded Systems – CHES 2017* (Cham, 2017), W. Fischer and N. Homma, Eds., Springer International Publishing, pp. 387–399.

[74] Rossi, F., Beek, P. v., and Walsh, T. *Handbook of Constraint Programming (Foundations of Artificial Intelligence).* Elsevier Science Inc., USA, 2006.

[75] Sasdrich, P., Bilgin, B., Hutter, M., and Marson, M. E. Low-latency hardware masking with application to AES. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2020*, 2 (March 2020), 300–326.

[76] Schneider, T., and Moradi, A. Leakage assessment methodology. In *Cryptographic Hardware and Embedded Systems – CHES 2015* (Berlin, Heidelberg, 2015), T. Güneysu and H. Handschuh, Eds., Springer Berlin Heidelberg, pp. 495–513.

[77] Schneider, T., Moradi, A., and Güneysu, T. Arithmetic addition over Boolean masking. In *Applied Cryptography and Network Security* (Cham, 2015), T. Malkin, V. Kolesnikov, A. B. Lewko, and M. Polychronakis, Eds., Springer International Publishing, pp. 559–578.

[78] Schrijver, A. *Theory of Linear and Integer Programming.* John Wiley & Sons, Inc., USA, 1986.

[79] Shahmirzadi, A. R., Božilov, D., and Moradi, A. New first-order secure AES performance records. *IACR Transactions on Cryptographic Hardware and Embedded Systems 2021*, 2 (February 2021), 304–327.

[80] Tiri, K., and Verbauwhede, I. A logic level design methodology for a secure DPA resistant ASIC or FPGA implementation. In *2004*

Design, Automation and Test in Europe Conference and Exposition (DATE 2004), 16-20 February 2004, Paris, France (2004), IEEE Computer Society, pp. 246–251.

[81] Trichina, E. Combinational Logic Design for AES SubByte Transformation on Masked Data. Cryptology ePrint Archive, Report 2003/236, 2003.

[82] Ueno, R., Homma, N., and Aoki, T. A systematic design of tamper-resistant Galois-field arithmetic circuits based on threshold implementation with (d + 1) input shares. In *2017 IEEE 47th International Symposium on Multiple-Valued Logic (ISMVL)* (2017), pp. 136–141.

[83] Ueno, R., Homma, N., and Aoki, T. Toward more efficient DPA-resistant AES hardware architecture based on threshold implementation. In *Constructive Side-Channel Analysis and Secure Design* (Cham, 2017), S. Guilley, Ed., Springer International Publishing, pp. 50–64.

[84] Wegener, F., De Meyer, L., and Moradi, A. Spin me right round rotational symmetry for FPGA-specific AES: Extended version. *Journal of Cryptology 33*, 3 (01 2020), 1114–1155.

[85] Šijačić, D., Balasch, J., Yang, B., Ghosh, S., and Verbauwhede, I. Towards efficient and automated side channel evaluations at design time. *Journal of Cryptographic Engineering 10* (2020), 305–319.

CV

Education

KU Leuven, Faculty of Engineering Science, Leuven, Belgium, 2015–2022
PhD in Cryptography
Adviser: Bart Preneel
Funded by Marie Skłodowska-Curie Scholarship

Research Visit at University of Bochum, Germany, April-July 2018

University of Belgrade, School of Electrical Engineering, Belgrade, Serbia, 2013–2014
Master's in Electrical Engineering
Thesis: Power Management in Android Operating System
Advisers: Lazar Saranovac and Strahinja Janković

University of Belgrade, School of Electrical Engineering, Belgrade, Serbia, 2009–2013
Bachelor's in Electrical Engineering
Graduated valedictorian of the class
Thesis: GPS Signal Parsing Using BeagleBoad and GPS Dongle
Advisers: Lazar Saranovac and Strahinja Janković

Teaching Experience

KU Leuven, Faculty of Engineering Science
TA for Design of Digital Platforms, Fall 2016, 2017, 2018

University of Belgrade, School of Electrical Engineering
TA for Fundamentals of Electrical Engineering Laboratory, Spring 2014

Talks

1. *Threshold Implementations of PRINCE*
   NIST Lightweight Cryptography Workshop 2016
   `https://www.nist.gov/system/files/documents/2016/10/04/agenda-lwc2016.pdf`
   Gaithersburg, Maryland, USA, October 2016

2. *A Note on 5-bit Quadratic Permutations' Classification*
   COSIC Seminar
   Leuven, Belgium, February 2017

3. *A Note on 5-bit Quadratic Permutations' Classification*
   Fast Software Encryption 2017
   `https://www.nuee.nagoya-u.ac.jp/labs/tiwata/fse2017/program.html`
   Tokyo, Japan, March 2017

4. *Optimized Threshold Implementations: Minimizing the Latency of Secure Cryptographic Accelerators*
   Smart Card Research and Advanced Application Conference 2019
   `https://cardis2019.fit.cvut.cz/program.html`
   Prague, Czech Republic, November 2019

5. *Design Trade-offs in Threshold Implementations*
   International Conference on Electronics, Circuits, and Systems Implementations 2019
   `https://researchr.org/publication/icecsys-2019`
   Genova, Italy, November 2019

6. *On Optimality of d+1 TI Shared Functions of 8 Bits or Less*
   COSIC Seminar
   Leuven, Belgium, June 2020

List of Publications

1. BOŽILOV D., BILGIN B., SAHIN H. A. A Note on 5-bit Quadratic Permutations' Classification. *In IACR Transactions on Symmetric Cryptology*, 2017(1), 398-404.

2. BOŽILOV D., NIKOV V., AND RIJMEN V. Design Trade-offs in Threshold Implementations. *In 2019 26th IEEE International Conference on Electronics, Circuits, and Systems Implementations*, 2019, IEEE, 751-754.

3. BOŽILOV D., KNEŽEVIĆ M., NIKOV V. Optimized Threshold Implementations: Minimizing the Latency of Secure Cryptographic Accelerators. In Belaïd, S., Güneysu, T. (eds) *Smart Card Research and Advanced Application. CARDIS 2019. Lecture Notes in Computer Science*, vol 11833. Springer, Cham. pp. 20-39.

4. BOŽILOV D., EICHLSEDER M. KNEŽEVIĆ M., LAMBIN B., LEANDER G., MOOS T., NIKOV V., RASOOLYADEH S., TODO Z., WIEMER F. PRINCEv2: More Security for (Almost) No Overhead. In Dunkelman, O., Jacobson, Jr., M.J., O'Flynn, C. (eds) *Selected Areas in Cryptography. SAC 2020. Lecture Notes in Computer Science*, vol 12804. Springer, Cham. pp. 483-511

5. SHAHMIRZADI A. R., BOŽILOV D., MORADI A. New First-Order Secure AES Performance Records. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2021 (2), 304-327.

6. BOŽILOV D., KNEŽEVIĆ M., NIKOV V. Optimized Threshold Implementations: Securing Cryptographic Accelerators for Low-Energy and Low-Latency Applications. In *Journal of Cryptographic Engineering* 2022 (12), 15-51.

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF ELECTRICAL ENGINEERING
COSIC
Kasteelpark Arenberg 10 bus 2452
B-3001 Leuven
dusan.bozilov@esat.kuleuven.be
http://www.cosic.esat.kuleuven.be