Optimizations in Symmetric Cryptography

Ko Stoffelen

ISBN 978-94-6421-722-3

Cover design by Marilou Maes, Persoonlijk Proefschrift
Printed by GVO Drukkers en Vormgevers

Optimizations in Symmetric Cryptography

Proefschrift

ter verkrijging van de graad van doctor
aan de Radboud Universiteit Nijmegen
op gezag van de rector magnificus prof. dr. J.H.J.M. van Krieken,
volgens besluit van het college voor promoties
in het openbaar te verdedigen op

woensdag 1 juni 2022
om 12:30 uur precies

door

Koos Wim Stoffelen

geboren op 7 april 1992
te Groningen

Promotoren

prof. dr. Joan Daemen

prof. dr. Peter Schwabe

Manuscriptcommissie

prof. dr. Bart Jacobs

prof. dr. Anne Canteaut
Inria, Frankrijk

prof. dr. ir. Vincent Rijmen
KU Leuven, België

dr. ir. Marc Stevens
Centrum Wiskunde & Informatica, Amsterdam

dr. Gilles Van Assche
STMicroelectronics, België

Acknowledgements

This journey would not have been possible without all the great people who came along the way. First of all, I express my thanks to Peter Schwabe, my supervisor. I recall a certain dinner at an Indian restaurant, after the Operating Systems Security course, where I first learned about post-quantum cryptography and the possibility of doing a PhD with you. Neither of us would then have thought that there would be so few mentions of post-quantum cryptography in this thesis. Thanks for always being so approachable and for providing guidance whenever asked, even if it required me to go through the Chicago Manual of Style again.

I also thank the manuscript committee consisting of Bart Jacobs, Anne Canteaut, Vincent Rijmen, Marc Stevens, and Gilles Van Assche, for taking the time to go through my thesis.

Most of the work was done together with a number of brilliant coauthors whom I am grateful to for all the discussions. Thanks to Joan Daemen, Lauren De Meyer, Benjamin Grégoire, Hannes Gross, Matthias Kannwischer, Thorsten Kranz, Martin Krenn, Gregor Leander, Stefan Mangard, Kostas Papagiannopoulos, Joost Rijneveld, Peter Schwabe, and Friedrich Wiemer.

Of the utmost importance was the presence of Brinda, Christoph, Dan, Engelbert, Erik, Fabian, Freek, Guillaume, Jon, Joost, Joost, Kostas, Léo, Marc, Matthias, Michael, Niels, Pedro, Pol, Ronny, Sander, and all the other nice colleagues in and around the Digital Security group – simply too many to list exhaustively! The 'around' extends even to the Eindhoven crew that was always there. I have fond memories of the countless coffee breaks, Friday drinks, group outings, movie nights, conferences, summer schools, working groups, weddings, and other events that we have enjoyed together. These really are the things that made this PhD possible and I can not imagine what it would have been like without. A huge thanks to all of you!

I thank Nick, Kris, and Alex of the Cloudflare crypto team for taking me on as an intern and for the valuable experience in London and San Francisco.

Next I am grateful to ’t Haasje for keeping me somewhat in shape, for being able to empty my mind during practice, for challenging competitions, but mostly for many fun events with its very enthusiastic members.

Then there is the Wijnnadeeltjes crew with whom I have shared many crazy adventures over at least as many drinks. Now spread out over the country and somewhat more mature, but may we continue doing this for a long time. Thanks!

From way back, thanks to Raphaël and Léon for being great friends for so long, regardless of how many times our lives have changed. It is hard to express how appreciated that is.

Finally, I thank my family for their love and support. We have shared many celebrations of life, but more importantly, you have always been there when we had to deal with the difficulties of life. Thanks to all of you.

Contents

Chapter 1

Introduction

Let me start with the title of this thesis. The title *Optimizations in Symmetric Cryptography* is only four words long, yet it expresses in a very compact way what this thesis is all about. To introduce this thesis and the topics that it contains, we can use the title and slowly unravel it word by word. For pedagogical reasons, it makes most sense to start on the right-hand side and move to the left.

The first and arguably the most important word that needs to be understood is *Cryptography*. History is filled with stories where protection of information plays a pivotal role. Whether this information concerns military operations, as with the invasion of Greece by the Achaemenid Empire under Xerxes I in the fifth century BC, or love letters, as between queen Marie Antoinette of France and count Axel von Fersen of Sweden, is of little interest here. Since long before the term 'cryptography' (or 'cryptology') was first coined, there have been people who tried to hide information by scrambling messages and there have been people who tried to decode these scrambled messages.

This is not any different from today's situation. In fact, protection of information may be more pivotal than ever in a largely digitalized society. What have changed considerably throughout history are the cryptographic methods that are used to scramble messages and the cryptanalytic methods that are used to decode messages. Modern-day cryptography is a lot more rigorous, mathematically speaking. Publicly available scrutiny of cryptographic systems has significantly improved the security of the systems we use today, despite more powerful attackers. Another trend is that modern-day cryptography has become more versatile: it considers more properties than only the confidentiality of information. For example, it can also give guarantees that a message was not modified in transit or who sent a particular message.

Now is a good moment to move to the next word of the title. The adjective *Symmetric* refers to a particular branch of cryptography. The confidentiality of an encrypted message usually depends on a cryptographic key and only the holder of this key should be able to decrypt the message. For some historical cryptographic systems this key may have been a code word, agreed in advance by the relevant parties. Nowadays that key is usually a string of ones and zeroes. When the same key is used to encrypt and to decrypt a message, this is a symmetric cryptographic system. This is unlike asymmetric cryptography, also called public-key cryptography, which was first realized in the 1970s. In public-key encryption, the key that is used to decrypt a message is different but mathematically related to the key that is used to encrypt a message. However, we need not submerge ourselves into this too much as this thesis only concerns symmetric cryptography.

Assuming that nobody will argue against skipping the preposition *in*, that just leaves *Optimizations*. This term is sufficiently vague that it spans across all core chapters of this thesis. Its use should immediately raise a few questions, such as what, why, and how. The how definitely goes too far for the introduction, so let's leave that for later chapters. The answers to what and why partially follow from the notion that adding cryptography always comes with certain costs. These costs can be any combination of more CPU cycles, more gates in an integrated circuit, more electrical energy required, more code complexity, and so on. Lowering these costs may lead to even more widespread use of cryptography, improving the security of information. Of course, it is an equally true statement that many computer scientists and mathematicians just like optimization problems.

1.1 Outline and Contributions

The core chapters of this thesis are based on a sequence of published papers. These papers contain contributions to multiple sub-areas in cryptographic research. While they are certainly related, it felt natural to categorize the work into three parts.

The chapters in Part I focus on building blocks that are used in the design of round functions of iterated cryptographic permutations and ciphers. The aim is to improve these small building blocks such that they increase the security of the complete scheme as much as possible using as few operations as possible. The three chapters study three types of cryptographic building blocks: S-boxes, maximum-distance separable matrices, and column-parity mixers.

Part II continues to optimize cryptographic permutations and ciphers, but at a different level. Once a scheme has been designed, it has to be implemented before it can be used in practice. This part covers hand-optimized software implementations in assembly language for two CPU architectures. Naturally, the choices that are made at the design stage of a permutation or cipher have a large impact on what one can do on the implementation level and, the other way around, design choices are typically made with software and/or hardware implementations in mind.

Finally, the chapters in Part III consider the common model where an attacker has access to a physical device and attempts to extract secret information, such as a secret key, using information provided by measurements of physical characteristics of that device, such as its power consumption. These information streams are called side channels. Masking is a well-studied countermeasure against this type of attacker, but it tends to be a costly one: implementations become much slower and/or bigger. The chapters in this part improve this with two different approaches. Once more, there is some interaction going on. Choices made during both the design and the implementation phases of a cipher impact the cost of applying side-channel countermeasures. Design and implementation choices can also be made with the cost of side-channel countermeasures in mind.

All core chapters are based on published papers with minor modifications. These modifications are mostly to make tables and figures fit within the dimensions of the current page size and to make the style and formatting more uniform. Some overlapping preliminaries have been moved to Chapter 2 to improve the flow for the reader. In a few cases, the scientific content has been updated, shortened, or otherwise modified. Such modifications are detailed at the beginning of the respective chapters.

Most chapters are based on joint work with various great coauthors. To clarify my personal contributions, I will describe per chapter what forms the basis of the chapter and how I contributed to it.

Chapter 3: S-boxes

This chapter is based on the following publication.

> Ko Stoffelen. "Optimizing S-Box Implementations for Several Criteria Using SAT Solvers". In: *Fast Software Encryption – FSE 2016*. Vol. 9783. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 140–160.

All work that is described in this publication is my own. The results on the multiplicative complexity of S-boxes were obtained while I was writing my Master's thesis [Sto15] under supervision of Lejla Batina. The work on all other optimization criteria, including the part on combining optimization criteria, was carried out as part of my PhD. The parts on multiplicative complexity, although technically not a part of my PhD, are only included for completeness.

Chapter 4: MDS Matrices

This chapter is based on the following publication.

> Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. "Shorter Linear Straight-Line Programs for MDS Matrices". In: *IACR Transactions on Symmetric Cryptology* 2017.4 (2017), pp. 188–211.

The work was a collaboration effort. Personally I worked on implementations of variants of what we call the BP heuristics and I ran many experiments. I also wrote parts of the paper and contributed to discussions. The results from the statistical analysis and the correlation figures were produced by Friedrich Wiemer and Thorsten Kranz.

Chapter 5: Column-Parity Mixers

This chapter is based on the following publication.

> Ko Stoffelen and Joan Daemen. "Column Parity Mixers". In: *IACR Transactions on Symmetric Cryptology* 2018.1 (2018), pp. 126–159.

The work was a collaboration effort. Personally I contributed to the development of the theory on column-parity mixers and their diffusion properties, the general design strategy, the Mixifer permutation and its security analysis, its software implementations, and to writing the paper. The work for the subsections on linear mask propagation, trail clustering, and on impossible differential cryptanalysis was done by Joan Daemen.

Chapter 6: ARM Cortex-M

This chapter is based on the following publication.

> Peter Schwabe and Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4". In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Vol. 10532. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 180–194.

The work was a collaboration effort. Personally I wrote all the code and I wrote most of the paper.

Chapter 7: RISC-V

This chapter is based on the following publication.

> Ko Stoffelen. "Efficient Cryptography on the RISC-V Architecture". In: *LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*. Vol. 11774. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2019, pp. 323–340.

All work that is described in this publication is my own.

Chapter 8: Vectorization

This chapter is based on the following publication.

> Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. "Vectorizing Higher-Order Masking". In: *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Vol. 10815. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 23–43.

The work was a collaboration effort. Personally I wrote the optimized implementations, helped with the lab setup, and wrote large parts of the paper. The side-channel evaluation was done by Kostas Papagiannopoulous and the new SNI refreshing and multiplication gadgets were contributed by Benjamin Grégoire.

Chapter 9: Reusing Randomness

This chapter is based on the following publication.

> Hannes Gross, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. "First-Order Masking with Only Two Random Bits". In: *Proceedings of ACM Workshop on Theory of Implementation Security*. TIS'19. ACM, 2019, pp. 10–23.

The work was a collaboration effort. A previous version of the `AND` gate and some results for hardware implementations were already put on IACR ePrint by the other authors before I joined. Since then, the work has changed direction and improved results considerably. Personally, I wrote the masked AES implementation, wrote parts of the paper, performed lab and simulator experiments, and contributed to discussions.

1.2 Research Data Management

Nearly all software and data that were used or produced for this thesis are available in online repositories that are publicly accessible. Unless a clearly marked license file in a repository states otherwise, copyrights and related rights are waived with the Creative Commons Zero v1.0 Universal waiver. This maximizes the potential for reuse of the software and data in the future.

References to the repositories are provided throughout this thesis, but they are also summarized in this section to make them easier to find.

S-box optimization tool
`https://github.com/Ko-/sboxoptimization`

Shorter linear SLPs for MDS matrices
`https://github.com/rub-hgi/shorter_linear_slps_for_mds_matrices`

Column-parity mixer trail search tool
`https://github.com/Ko-/cpm`

Mixifer implementations
`https://github.com/Ko-/mixifer`

AES implementations for ARM Cortex-M3 and M4
`https://github.com/Ko-/aes-armcortexm`

Implementations of cryptographic primitives for RV32I
`https://github.com/Ko-/riscvcrypto`

Masked AES implementations parallelized with vectorization
`https://github.com/Ko-/aes-masked-neon`

Masking with two random bits
`https://github.com/LaurenDM/TwoRandomBits`

In addition to the software and data referred to above, side-channel trace sets exist related to Chapter 8 and Chapter 9. These are not available online due to their large sizes. Instead, they are locally stored on a hard disk drive in the side-channel lab of the Digital Security group at Radboud University and will remain available upon request.

Chapter 2
Preliminaries

This chapter aims to provide some background information on symmetric cryptology and on closely-related scientific areas. This is also where we establish some notation commonly used in subsequent chapters. This chapter is by no means a complete account of everything that there is to know about the topics that are discussed, but it merely intends to cover the common ground that is required for other chapters. In those other chapters, these preliminaries are supplemented with information that is more relevant to a specific chapter. Still, the reader is referred to a proper textbook or a course if the reader desires information that is not covered here.

This chapter first explains some basic mathematical concepts in Section 2.1. It then proceeds with the fundamentals of symmetric cryptology in Section 2.2. Section 2.3 puts cryptographic algorithms into a physical device, such as a microcontroller, and explains why implementations are also scientifically interesting. Finally, Section 2.4 discusses pitfalls of cryptographic implementations, how they can be attacked, and how these attacks can be prevented.

2.1 Mathematics

Most constructs in cryptology use existing and well-understood concepts from number theory, algebra, and other branches of mathematics. This makes it easier to reason about these constructs and therefore to study their security properties, which is what cryptographers are usually interested in. It is assumed that the reader is familiar with basic linear algebra, probability theory, predicate logic and common algebraic structures. Nonetheless, some definitions are available here as a reference.

A *set* S is a collection of unique objects, such as different numbers. An object a is an *element* of a set S, denoted $a \in S$, when it is one of the objects in the collection. A set can be specified by listing its objects $\{a, b, c, d, \dots\}$, but that can get quite lengthy when a set contains an infinite number of objects. Some commonly used sets also have special names. For example, $\mathbb{Z}$ is the set of integers and $\mathbb{N}$ is the set of natural numbers. In order to avoid ambiguity, $0 \in \mathbb{N}$ in this thesis.

A *group* $(G, \cdot)$ is an algebraic structure that is given by a set G and a binary operation $\cdot : G \times G \rightarrow G$ that satisfies three criteria:

- *Associativity*: $\forall a, b, c \in G : (a \cdot b) \cdot c = a \cdot (b \cdot c)$.
- *Identity element*: $\exists e \in G : e \cdot a = a \cdot e = a$. This element e is unique.
- *Inverse element*: $\forall a \in G : \exists b \in G : a \cdot b = b \cdot a = e$. This b is called the inverse of a and denoted a^{-1}.

An example of a group is $(\mathbb{Z}, +)$, where $+$ is addition of integers. For this additive example, inverses are denoted $-a$. A group $(G, \cdot)$ is *Abelian* when $\cdot$ is also commutative: $\forall a, b \in G : a \cdot b = b \cdot a$.

A straightforward extension of groups adds another operation. A *ring* $(R, +, \cdot)$ is an algebraic structure where the following criteria hold:

- $(R, +)$ is an Abelian group.
- $\cdot$ is associative over R and has a (multiplicative) identity element.
- *Distributivity*: $\forall a, b, c \in R : a \cdot (b + c) = (a \cdot b) + (a \cdot c) \land (a + b) \cdot c = (a \cdot c) + (b \cdot c)$.

An example of a ring is $(\mathbb{Z}, +, \cdot)$, where $+$ and $\cdot$ are addition and multiplication of integers, respectively.

A *commutative ring* is a ring where the multiplication operation $\cdot$ is also commutative. A *field* is a commutative ring where every element of its set also has a multiplicative inverse under the multiplication operation $\cdot$. A group, ring, and field are *finite* when their set has a finite number of elements. In (symmetric) cryptology, finite fields are common algebraic structures.

The *order* of a finite field, often denoted by q, is defined as the number of elements in its set. A finite field of a certain order q exists if and only if q is some power of a prime number, i. e., $q = p^k$ with k a positive integer. Then p is also known as the *characteristic* of the field. In general, there are many ways to construct a finite field with a specific order q, but they are all isomorphic. Intuitively, this means that finite fields of equal order behave the same with respect to calculations, as there exist reversible mappings between them. This is why explicit constructions are sometimes ignored and one just speaks of 'the' finite field $\mathbb{F}_q$. The smallest finite field is the field with 2 elements, denoted $\mathbb{F}_2$, which can be used to model bits with `XOR` and `AND` operations.

2.2 Symmetric Cryptology

In this section we cover several cryptographic primitives that most cryptographic schemes in symmetric cryptology are built upon. Before we discuss these primitives, the properties that they provide, and some examples, we look at idealized abstractions that model the properties that we desire. Finally, we discuss one instantiation of a cryptographic primitive, AES, in more detail as it will occur in multiple chapters of this thesis.

2.2.1 Idealized Abstractions

The first notion is that of a *pseudorandom permutation* (PRP). In general, a permutation is a bijective function from some set to itself. A PRP $f : X \to X$ is a permutation f from some set X to itself, such that it cannot be distinguished with non-negligible probability from a permutation that is selected uniformly at random from the set of all permutations over X. More commonly, a PRP is defined using a PRP family $f : K \times X \to X$, where there is some key space K

and for every $k \in K$ it holds that $f(k, \cdot)$ is a PRP. It is usually required that f and f^{-1} are efficiently computable.

A more general notion is that of a *pseudorandom function* (PRF). The difference is that a PRF does not require that the function maps a set to itself, but it allows an arbitrary output set. Consequently, it does not need to be a bijective mapping and it may not have an inverse. Clearly, every PRP is also a PRF.

Such idealized abstractions are useful in security proofs. They also clarify what properties a particular instantiation of a cryptographic primitive should have.

2.2.2 Primitives

Arguably one of the simplest primitives is the (unkeyed) *cryptographic permutation*. This is a function $P : \{0,1\}^n \to \{0,1\}^n$ that processes fixed-length blocks such that P is a PRP. Typical examples are the Keccak-f permutations [BDPV11b].

Block ciphers also process fixed-length blocks, but add a fixed-length key. More precisely, a block cipher consists of an encryption function $E : \{0,1\}^{|k|} \times \{0,1\}^n \to \{0,1\}^n$ and a decryption function $D : \{0,1\}^{|k|} \times \{0,1\}^n \to \{0,1\}^n$ such that for all $k \in \{0,1\}^{|k|}$ and $x \in \{0,1\}^n$, $D(k, E(k, x)) = x$. A secure block cipher is a family of PRPs. A block cipher can be constructed from a cryptographic permutation [EM93], but direct instantiations of block ciphers are more common. The prime example of a block cipher that is heavily used today is AES. We will discuss its construction in more detail in the Section 2.2.3.

A downside of a block cipher is that one can only use the encryption or decryption function once there is an input of precisely n bits. *Stream ciphers* are more flexible and allow inputs of arbitrary length or at least arbitrary multiples of some block length. Stream ciphers can be constructed from block ciphers by putting the block cipher in a so-called mode of operation. AES-CTR is a stream cipher that works like this. They can also be constructed directly, such as the ChaCha ciphers [Ber08a].

A *cryptographic hash function*, or simply hash function in the context of this thesis, is another primitive that is used in many cryptographic schemes. A hash function is an efficiently-computable function $h : \{0,1\}^* \rightarrow \{0,1\}^n$ that maps arbitrary-length inputs to fixed-length outputs. h needs to satisfy the following additional criteria:

- *Pre-image resistance*: given $y = h(x)$, it should be hard to find any x' such that $h(x') = y$.
- *Second pre-image resistance*: given x, it should be hard to find any $x' \neq x$ such that $h(x) = h(x')$.
- *Collision resistance*: it should be hard to find any x, x' with $x \neq x'$ such that $h(x) = h(x')$.

Typical examples are the SHA-2 [NIS15a] and SHA-3 [NIS15b] hash functions.

2.2.3 AES

In this section we zoom in on one particular block cipher, as it appears in all chapters of this thesis. In 1997, NIST announced that they were looking for a successor to DES [DES77] and they put out a call for proposals for new ciphers. All submissions were to be block ciphers that support a block length of 128 bits and key sizes of 128, 192, and 256 bits. Out of fifteen submissions, Rijndael was selected as the winner and it was finally standardized as AES in 2001 [DR02].

The AES design is based on a number of rounds, where each round consists of a nonlinear substitution of the bytes of the state, followed by linear operations that aim to spread and mix parts of the state, followed by the addition of a round key that is derived from the key through a key schedule. These steps are called SubBytes, ShiftRows, MixColumns, and AddRoundKey in that order. The first round starts with an initial AddRoundKey and the final round is different by omitting MixColumns as it does not improve security anymore. AES-128, the variant with a key size of 128 bits, has 10 rounds, AES-192 12 rounds, and AES-256 14 rounds.

The security of AES has been established through years of thorough cryptanalysis. An attack that is always possible is brute-force search for the correct key. The amount of effort that this costs is determined by the size of the key, so for AES-128 this takes 2^{128} AES operations (or 2^{127} on average). Linear [Mat94] and differential [BS91] cryptanalysis are general methods that apply to all block ciphers and have proved themselves to be powerful attacks in the past. However, AES was designed with these attacks in mind and its designers have proven that these attacks require much more effort than brute-force search.

Currently, the best-known attack against AES-128, conveniently ignoring related-key attacks for the moment, is a biclique attack that requires $2^{126.13}$ operations and 2^{56} bits or 9 petabyte of storage [TW15]. Although this implies a theoretical break of AES, in practice, the biclique attack is unlikely to be faster than simple brute-force search of the key.

AES is widely deployed and used in many products, protocols, and industries to securely encrypt data.

2.3 Cryptographic Implementations

Cryptographic algorithms are implemented in a wide range of hardware and software. A very rough division can be made between ASICs, FPGAs, microcontrollers, smartphones, high-end CPUs, and GPUs. The topics in this thesis primarily consider the middle part of this range.

2.3.1 Assembly

A lot of cryptographic implementations are written in assembly language. This has several advantages. First of all, the implementation is likely to be faster. This is especially useful if the operation is to be performed many times. A human with knowledge of the mathematical properties of the cryptographic algorithm can think of optimizations that are beyond what generic compilers can do. Also note that compilers are optimized to perform well on average, without affecting the compilation time or binary size too much. This may not be what is wanted for a cryptographic implementation. Another reason for programming in assembly is that a compiler may unknowingly alter the behavior of a program. Statements such as removing a key from memory by overwriting it with zeroes may be removed by a compiler, because it thinks it is useless if that value is not read again later.

Some disadvantages of assembly are, of course, that you lose portability between CPU architectures and that you increase the code complexity. Usually the idea is that this effort is done once in a software library and everyone else can then reuse that library.

2.3.2 ARM Cortex-M

Various CPU architectures are mentioned frequently throughout this thesis, hence they are briefly described in this section. The first is a line of microprocessors called Cortex-M by ARM. This consists of a number of popular 32-bit low-cost processors for embedded applications that have been used in billions of devices. These processors generally run lightweight embedded operating systems or none at all. Although it is described as a RISC architecture, some models do come with multipliers, divide instructions, DSP instructions, and floating-point instructions. Noteworthy are the barrel-shift registers that provide free shifts and rotations of one input of arithmetic instructions.

2.3.3 ARM Cortex-A and NEON

Cortex-A is a group of 32-bit and 64-bit processors that are used in smartphones, tablets, and other applications that require more performance. These typically run a full-fledged operating system such as Linux.

The NEON media processing engine or Advanced SIMD extension extends a Cortex-A processor with a 64-bit or 128-bit SIMD vector unit that can accelerate media and signal processing, but it can also be useful for parallel processing in cryptographic implementations.

2.3.4 RISC-V

ARM designs processors, but they do not build them themselves. Instead, other companies purchase licenses to use their designs. The RISC-V project is about creating an open and free instruction set architecture and started originally at the University of California at Berkeley. There exist both open-source and proprietary processors that implement this instruction set architecture. There are 32-bit, 64-bit, and 128-bit variants of the instruction set, targeting a wide range of applications. It should be noted that the base contains very few instructions and that there are many standardized extensions that bring functionality such as multiplication, floating-point support, and vector operations, some of which are still in development.

2.4 Side-Channel Attacks and Countermeasures

Research on cryptographic implementations not only aims to lower the costs of adding more secure cryptography, but it also aims to make the implementations themselves more secure. A theoretically secure cipher is not as useful when its implementation contains flaws that can be exploited to easily extract a secret key. The attacker may use information about physical characteristics of a particular implementation, called *side channels*. To reason about implementation security it is important to have an explicit *attacker model* that states what an attacker is allowed to do. From this it can be determined whether a model is relevant to a particular use case. For example, on a regular PC a secret key can often be retrieved by a privileged user by extracting it from memory or by attaching a debugger to an encryption process. From the perspective of the implementation of a cryptographic operation in a desktop application, it is impossible to exclude these types of attacks without relying on assistance from special-purpose hardware. Hence an implementer may decide that this attacker model is not too relevant for their use case and accept these risks. On the other hand, a company that produces credit cards may take sophisticated side-channel attacks very seriously.

2.4.1 Timing Attacks

One branch of side-channel attacks uses timing information. A simple example is the case of testing whether two strings are equal by comparing them character by character and returning when a mismatch occurs. By measuring the time it takes to do this comparison, an attacker can determine at what position the strings begin to differ. This turns the hard problem of guessing the right string into the much simpler problem of guessing the right character a few times in a row. Kocher famously applied a similar idea to implementations of asymmetric cryptographic algorithms in 1996 [Koc96].

These timing attacks can be quite subtle. In 2005, Bernstein showed that full key recovery was possible against the AES implementation in OpenSSL because of the behavior of caches in CPUs [Ber05a]. The solution is the notion of *constant-time* implementations: the running time should be independent of secret values. But controlling the running time can be hard when there are many abstraction layers between the code and the physical world. Compilers may introduce timing variance in code that was intended to be constant-time for optimization purposes and even a single instruction may turn out to be less constant-time than one expected due to a hidden mechanism in complex and proprietary hardware.

Assembly implementations and publicly available documentation of hardware do help out a lot, as well as in-depth testing to the timing behavior of CPU instructions. Another direction is to put the whole cryptographic operation in a hardware circuit, such as with the AES-NI instruction set extension for Intel and AMD CPUs.

2.4.2 Power and Electromagnetic Attacks

Another branch of side-channel attacks uses other physical characteristics. In 1999, Kocher, Jaffe, and Jun showed how on many devices, the power consumption of the device correlates with the instructions and values that are being processed [KJJ99]. In cryptographic implementations, some of these values are supposed to be secret. However, by measuring the power consumption across a cryptographic operation, the implementation may leak the full cryptographic key. With *simple power analysis* (SPA), patterns are directly deduced from a single *trace* of measurements. Usually this is not enough to recover a key. *Differential power analysis* (DPA) is a more powerful attack that requires multiple traces and some statistics. The main idea is to guess some secret bits (for example, a byte of a round key in the case of AES) and to perform a statistical test to confirm whether the guess was correct. A notable variation is *correlation power analysis* (CPA) that uses the Pearson correlation coefficient to distinguish between a correct guess and a wrong guess [BCO04].

Similarly, simple and differential *electromagnetic analysis* are based on the fact that a current induces an electromagnetic field that can be measured by a probe [QS01]. Sometimes it is hard to measure the current going through a processor directly because of how the processor is integrated into a circuit. Its electromagnetic field may then be easier to measure. However, a downside can be that there may be more noise in the measurement of the electromagnetic field.

A high noise level can be combatted by generating more traces, but this also increases the amount of effort for an attacker. Some countermeasures against power and electromagnetic analysis are therefore based on increasing the amount of noise. The countermeasure that occurs most in this thesis is called *masking*. This aims to break the correlation between the physical measurement and the secret value by adding randomness. Secret values are split into shares that by themselves are uniformly random. A careful implementation then performs the computation on the shares in such a way that the computation remains correct and that secret intermediate values remain uniformly random.

Power and electromagnetic analysis, as well as masking, can be extended to *higher orders*. This means that the statistical properties of multiple aspects of a signal are studied jointly. This can be approached with multivariate statistics or by first mapping the problem to a univariate problem. For masking, protection against higher-order attacks means that secret values need to be split into more shares.

PART I

Cryptographic Building Blocks

Chapter 3

S-boxes

As a first cryptographic building block, we consider the nonlinear S-boxes that can be found in round functions of most iterated permutations or ciphers. This chapter introduces a technique to optimize implementations of such S-boxes for a number of optimization criteria. In comparison to the original publication [Sto16b], a few mistakes that were pointed out by Jérémy Jean are corrected. The S-boxes of RECTANGLE and its inverse were swapped and in an implementation of LAC there was a typo in a variable name. Another modification is that the appendices are omitted.

3.1 Introduction

Implementations of cryptographic algorithms are typically optimized for one or multiple criteria, such as latency, throughput, power consumption, memory consumption, etc., but also criteria such as the cost of adding masking countermeasures to protect against side-channel attacks. It is worthwhile to spend time on this optimization, as the implementations are typically used many times. It is usually a hard problem to find an implementation that is actually theoretically minimal with respect to the criteria, e. g., general circuit minimization is $\sum_2^P$-complete [BU08]. However, for small functions this is still possible, using, for instance, SAT solvers. Especially for building blocks that can be used in multiple cryptographic algorithms, such as S-boxes, it is useful to look at methods for finding minimal implementations with respect to some given criteria.

In Section 3.2, we first discuss the simpler problem of finding minimal implementations of linear functions. We give a brief overview of methods for finding the shortest linear straight-line program.

We then move toward S-boxes and in Section 3.3 we consider known methods [CMH13; Mou15] that manage to find minimal implementations for the relevant optimization criteria of multiplicative complexity [BPP00], bitslice gate complexity [CHM11], and gate complexity. The definitions of these criteria are given in Section 3.3. We study how feasible the methods actually are by applying them to S-boxes that are used in recent cryptographic algorithms, such as several candidates in the CAESAR competition[1] and lightweight block ciphers. Additionally, we provide tools that allow anyone to conveniently do the same to other small S-boxes.

Then we look at another optimization criterion: the circuit-depth complexity. This is relevant in hardware implementations to decrease the delay and to be able to increase the clock frequency. We suggest a new method for encoding the circuit-depth-complexity decision problem in SAT and we show how feasible this method is in practice by providing efficient low-depth S-box implementations for Joltik [JNP15], Piccolo [SIH+11], LAC [ZWW+14], Prøst [KLL+14], and RECTANGLE [ZBL+14] in Section 3.3.5.

Section 3.4 discusses how several optimization criteria can be combined, by first optimizing the S-box used by the PRIMATEs [ABB+14] for multiplicative complexity and then for gate complexity. This is done by taking the intermediate result after optimizing for multiplicative complexity, identifying the linear parts of this, and by treating these as instances of the shortest linear straight-line program problem.

Full listings of the optimized implementations that are obtained can be found in the appendices of the original publication [Sto16b].

Contributions presented in this chapter. To summarize, the contributions of this chapter are:

---

[1] `https://competitions.cr.yp.to/caesar.html`

- implementations of the S-boxes in Ascon, ICEPOLE, Joltik/Piccolo, Keccak/Ketje/Keyak, LAC, Minalpher, Prøst, and RECTANGLE with a provably minimal number of nonlinear gates;
- a new method for encoding the circuit-depth-complexity decision problem as an instance of SAT;
- optimized and in some cases even provably minimal implementations of the S-boxes in Joltik/Piccolo, LAC, Prøst, and RECTANGLE with respect to bitslice gate complexity, gate complexity, and circuit-depth complexity;
- a method to combine multiple optimization criteria;
- an implementation of the S-box used by the PRIMATEs that is first optimized for multiplicative complexity and then for (bitslice) gate complexity;
- tools and documentation to optimize implementations of small nonlinear functions such as S-boxes using SAT solvers, with respect to multiplicative complexity, bitslice gate complexity, gate complexity, or circuit-depth complexity, as described in Section 3.4.

3.2 Shortest Linear Straight-Line Programs

Before tackling the optimization of S-boxes, let us restrict ourselves to linear functions and let us consider the Shortest Linear Program (SLP) problem over $\mathbb{F}_2$. Let A be an $m \times n$ matrix of constants over $\mathbb{F}_2$ and let x be a vector of n variables over $\mathbb{F}_2$. The SLP problem is to find the program with the smallest number of lines that computes Ax, where every program line is of a certain form.

Let Z be a set of variables over $\mathbb{F}_2$, that initially contains the input variables $\{x_0, \ldots, x_{n-1}\}$. Let $z_i, z_j \in Z$. Then every program line is of the form

$$z' := z_i + z_j.$$

After executing this program line, the new variable z' is added to the set, $Z := Z \cup \{z'\}$. The new variable z' can therefore be used in the next program line. The program is said to compute Ax when there exists a vector $(z_1, \ldots, z_m) \in Z^m$ such that $Ax = (z_1, \ldots, z_m)^\intercal$.

Being able to find the shortest straight-line linear program has obvious applications to cryptology. Solving the SLP over $\mathbb{F}_2$ is equivalent to finding the shortest circuit to compute a function using only `XOR` gates. Optimizing implementations of linear operations, such as MixColumns in AES and the linear components of certain implementations of SubBytes, can therefore be seen as instances of the SLP problem over $\mathbb{F}_2$. However, this method does not apply to nonlinear operations such as S-boxes. We show in Section 3.3 what kind of methods can be used in such cases.

Solving the SLP problem. Boyar, Matthews, and Peralta showed that the SLP problem over $\mathbb{F}_2$ is NP-hard [BMP08]. Off-the-shelf SAT solvers can be used to find solutions for small instances of this problem. Fuhs and Schneider-Kamp presented a method [FS10] to encode the SLP problem as an instance of SAT and they show how this can be used to optimize the affine transformation of AES's SubBytes [FS10; FS12].

For larger instances, exact methods will quickly become infeasible. Alternatively, Boyar and Peralta published an approach to solve the SLP problem over $\mathbb{F}_2$ based on a heuristic [BP10]. In short, the heuristic method uses a base vector set S, initialized with unit vectors for all variables in x, and a distance vector *Dist* that keeps track of the minimal Hamming distance to S for each row in A. Repeatedly, the sum of the pair of base vectors in S that minimizes the sum of *Dist* is added to S and *Dist* is updated, until *Dist* is the all-zero vector. If there is a tie between two pairs of base vectors, the pair that maximizes the Euclidean length of the new *Dist* vector is chosen. This algorithm makes it possible to find solutions to larger instances of the SLP problem.

3.3 Optimizing S-box Implementations using SAT Solvers

For nonlinear functions such as S-boxes, known approaches based on heuristics [BP10] all exploit additional algebraic structure that may be available, e. g., as for the S-box of AES. However, in general this additional structure may not exist and one may need to fall back to generic methods such as SAT solvers.

S-box implementations in both software and hardware can be optimized with SAT solvers according to several criteria. In this chapter we consider the following four optimization goals:

Multiplicative complexity. The multiplicative complexity of a function [BPP00] is defined as the smallest number of nonlinear gates with fan-in 2 required to compute this function. If we restrict our S-box implementations to the {`AND`, `OR`, `XOR`, `NOT`} operations, we only need to consider the number of `AND`s and `OR`s. Optimizing for this goal is useful in the case of protecting against side-channel attacks using random masks, where nonlinear gates are typically more expensive to mask. There are also applications in multi-party computation and fully homomorphic encryption, where the cost of nonlinear operations is even more significant [ARS+15].

Bitslice gate complexity. The bitslice gate complexity of a function [CHM11] is defined as the smallest number of operations in {`AND`, `OR`, `XOR`, `NOT`} required to compute this function. This translates directly to efficient bitsliced software implementations, as on most common CPU architectures, there are no instructions for computing `NAND`, `NOR`, or `XNOR` immediately.

Gate complexity. The gate complexity of a function is defined as the smallest number of logic gates required to compute this function. Unlike for bitslice gate complexity, `NAND`, `NOR`, and `XNOR` gates are now also allowed. This translates to efficient hardware implementations, although the different amounts of area required by these types of gates and the different delays still need to be taken into account. Note that we only consider gates with a fan-in of at most 2.

Circuit-depth complexity. The depth of a circuit is defined as the length of the longest paths from an input gate to an output gate. Every function can be computed by a circuit with depth 2, e. g., by expressing the function in conjunctive or disjunctive normal form. However, this can lead to very wide circuits with a lot of gates, which is typically not desirable. There is somewhat of a trade-off between circuit depth and number of gates. Still, optimizing for this goal is useful in the case of hardware implementations, to be able to decrease the total delay and therefore to be able to increase the clock frequency. Again, only gates with a fan-in of at most 2 are considered.

These criteria come with corresponding decision problems. For example, given a function f and some positive integer k, the *multiplicative-complexity decision problem* is defined as:

> "Is there a circuit that implements f and that uses at most k nonlinear operations?"

The decision problems for the other three optimization goals can be defined analogously. Off-the-shelf SAT solvers can be used to solve these decision problems. When a SAT solver successfully finds a circuit for some value k but outputs `UNSAT` for $k - 1$, it is proven that k is the minimum value. Note that when a SAT solver outputs `SAT` for some value k, it also provides a satisfying valuation that can be used to reconstruct an implementation of f.

In order to use SAT solvers to solve these decision problems, the problems first have to be encoded in logical formulas in conjunctive normal form (CNF), because that is the input format that the SAT solver requires.

3.3.1 Notation

For the encoding, we use the notation of Mourouzis [Mou15]. We consider systems of multivariate equations over $\mathbb{F}_2$. In these equations, let

- x_i be variables representing S-box inputs;
- y_i be variables representing S-box outputs;
- q_i be variables representing gate inputs;
- t_i be variables representing gate outputs;
- a_i be variables representing wiring between gates; and
- b_i be variables representing wiring *inside* gates. This will become more clear when they are first used in Section 3.3.3.

In the implementations the *logical connectives* are used to denote the types of operations. Let $\wedge$, $\vee$, $\oplus$, $\neg$ denote `AND`, `OR`, `XOR`, `NOT`, respectively, and let $\uparrow$, $\downarrow$, $\leftrightarrow$ denote `NAND`, `NOR`, `XNOR`, respectively.

3.3.2 Optimizing for Multiplicative Complexity

Courtois, Mourouzis, and Hulme [CMH13; Mou15] suggested a method to encode the multiplicative-complexity decision problem. Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ be an S-box and let k be the multiplicative complexity that we want to test for. Then first create a set of equations C in ANF consisting of:

- $\forall i \in \{0, \ldots, k-1\}$: $t_i = q_{2i} \cdot q_{2i+1}$, to encode the k `AND` gates.
- $\forall i \in \{0, \ldots, 2k-1\}$: $q_i = a_l + \left(\sum_{j=0}^{n-1} a_{l+j+1} \cdot x_j\right) + \left(\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor - 1} a_{l+n+j+1} \cdot t_j\right)$, where $l = i(n+1) + \left\lfloor \frac{i^2-2i+1}{4} \right\rfloor$, to encode that the inputs of the `AND` gates can be any linear combination of S-box inputs and previous `AND`-gate outputs. The single a represents an optional `NOT` gate.

- $\forall i \in \{0, \dots, m-1\}$: $y_i = \left(\sum_{j=0}^{n-1} a_{s+j} \cdot x_j\right) + \left(\sum_{j=0}^{k-1} a_{s+n+j} \cdot t_j\right)$, where $s = 2k(n+1) + k(k-1) + i(n+k)$, to encode that the S-box outputs can be any linear combination of S-box inputs and `AND`-gate outputs.

For example, when $n = m = 4$ and $k = 3$, this leads to the following set of equations C:

$$\begin{aligned}
q_0 &= a_0 + a_1 \cdot x_0 + a_2 \cdot x_1 + a_3 \cdot x_2 + a_4 \cdot x_3 \\
q_1 &= a_5 + a_6 \cdot x_0 + a_7 \cdot x_1 + a_8 \cdot x_2 + a_9 \cdot x_3 \\
t_0 &= q_0 \cdot q_1 \\
q_2 &= a_{10} + a_{11} \cdot x_0 + a_{12} \cdot x_1 + a_{13} \cdot x_2 + a_{14} \cdot x_3 + a_{15} \cdot t_0 \\
q_3 &= a_{16} + a_{17} \cdot x_0 + a_{18} \cdot x_1 + a_{19} \cdot x_2 + a_{20} \cdot x_3 + a_{21} \cdot t_0 \\
t_1 &= q_2 \cdot q_3 \\
q_4 &= a_{22} + a_{23} \cdot x_0 + a_{24} \cdot x_1 + a_{25} \cdot x_2 + a_{26} \cdot x_3 + a_{27} \cdot t_0 + a_{28} \cdot t_1 \\
q_5 &= a_{29} + a_{30} \cdot x_0 + a_{31} \cdot x_1 + a_{32} \cdot x_2 + a_{33} \cdot x_3 + a_{34} \cdot t_0 + a_{35} \cdot t_1 \\
t_2 &= q_4 \cdot q_5 \\
y_0 &= a_{36} \cdot x_0 + a_{37} \cdot x_1 + a_{38} \cdot x_2 + a_{39} \cdot x_3 + a_{40} \cdot t_0 + a_{41} \cdot t_1 + a_{42} \cdot t_2 \\
y_1 &= a_{43} \cdot x_0 + a_{44} \cdot x_1 + a_{45} \cdot x_2 + a_{46} \cdot x_3 + a_{47} \cdot t_0 + a_{48} \cdot t_1 + a_{49} \cdot t_2 \\
y_2 &= a_{50} \cdot x_0 + a_{51} \cdot x_1 + a_{52} \cdot x_2 + a_{53} \cdot x_3 + a_{54} \cdot t_0 + a_{55} \cdot t_1 + a_{56} \cdot t_2 \\
y_3 &= a_{57} \cdot x_0 + a_{58} \cdot x_1 + a_{59} \cdot x_2 + a_{60} \cdot x_3 + a_{61} \cdot t_0 + a_{62} \cdot t_1 + a_{63} \cdot t_2
\end{aligned}$$

This set of equations does not depend on f yet, but only on the values of n and m. The equations in C have to be satisfied for all possible S-box inputs. An equation set C' is created that contains 2^n copies of the equations in C, in which all x_i, y_i, q_i, t_i are renumbered, but in which all a_i, b_i remain the same. f is *bound* to the problem description by adding its truth table as $2^n(n+m)$ constant equations, i. e., one for every bit in both the S-box input and the S-box output, to C'.

C' is in ANF. The method by Bard, Courtois, and Jefferson [BCJ07] for converting sparse systems of low-degree multivariate binary polynomials is used to convert C' to CNF, such that it is understood by the SAT solver.

Results. This method makes it feasible to find the multiplicative complexity of several 4-bit and 5-bit S-boxes. Finding the multiplicative complexity comes with an actual implementation that uses this minimal number of nonlinear gates. After Courtois, Hulme, and Mourouzis applied this method to the S-boxes of PRESENT and GOST [CHM11], we show that we can also find results for more recently introduced 4-bit and 5-bit S-boxes.

We consider the S-boxes, and if applicable, their inverses (denoted by $^{-1}$), in Ascon [DEMS16], ICEPOLE [MGH+14], Keccak [BDPV11b]/Ketje [BDP+16a]/Keyak [BDP+16b], all PRIMATEs [ABB+14], Joltik [JNP15]/Piccolo [SIH+11], LAC [ZWW+14], Minalpher [STA+15], Prøst [KLL+14], and RECTANGLE [ZBL+14]. Minalpher's and Prøst's S-boxes are involutory, which is why their inverses are not listed separately. The inverse S-boxes in Ascon, ICEPOLE, Keccak, Ketje, and Keyak are not actually used in decryption and are therefore not considered.

For all S-boxes except the one used by the PRIMATEs we are able to prove the multiplicative complexity. The results are summarized in Table 3.1. The actual implementations can be found in the original publication [Sto16b], but note that these should most likely not be used by themselves as we are being very generous with `XOR` gates. The linear parts should be optimized separately, as we will demonstrate in Section 3.4.

These and subsequent results are obtained using MiniSat 2.2.0[2] and CryptoMiniSat 2.9.10[3] using default parameters on a single core of an Intel Xeon E7-4870 v2 running at 2.30 GHz.

---

[2] `https://www.minisat.se/MiniSat.html`

[3] `https://www.msoos.org/cryptominisat2/`

Table 3.1: Multiplicative complexity of S-boxes.

S-box	Size	Multiplicative complexity
Ascon	5	5
ICEPOLE	5	6
Keccak/Ketje/Keyak	5	5
PRIMATEs	5	$\in \{6, 7\}$
PRIMATEs^{-1}	5	$\in \{6, 7, 8, 9, 10\}$
Joltik/Piccolo	4	4
Joltik^{-1} /Piccolo^{-1}	4	4
LAC	4	4
Minalpher	4	5
Prøst	4	4
RECTANGLE	4	4
RECTANGLE^{-1}	4	4

For the PRIMATEs S-box and inverse S-box, we find solutions for $k = 7$ and $k = 10$, respectively. Furthermore, we find for both S-boxes that the case for $k = 5$ yields `UNSAT`. We have started several attempts to find a decisive answer for $k = 6$, including:

- reducing the CNF, e. g., using NICESAT [CMV09];
- fine-tuning SAT solver parameters;
- trying other SAT solvers;
- trying other SAT solvers that can run in parallel on many cores, such as Plingeling and Treengeling;[4] and

---

[4] `http://fmv.jku.at/lingeling/`

- letting all of this run for several months on a machine with 120 cores and 3 TB of RAM.

Unfortunately, none of these attempts resulted in an answer as no solver instance has terminated yet. As these SAT solvers typically have much more difficulty with proving the `UNSAT` case than proving the `SAT` case, and as the `SAT` proof for $k = 7$ was found in less than 40 hours, we expect the $k = 6$ case to yield `UNSAT` and we therefore conjecture the multiplicative complexity of the PRIMATEs S-box to be 7. In Section 3.4 we go into more detail on optimizing the PRIMATEs S-box. For the inverse S-box, we did not manage to find solutions for $k \in \{6, 7, 8, 9\}$.

3.3.3 Optimizing for Bitslice Gate Complexity

In the same work [CMH13; Mou15], a method is given to optimize for bitslice gate complexity. However, it is only applied on the small CTC2 toy cipher and therefore it remains unclear how practical this method is for real-world ciphers. We investigate this by applying the method to the same S-boxes as in the previous section.

The encoding scheme for the bitslice-gate-complexity decision problem is slightly different compared to the multiplicative-complexity decision problem. Let $f : \mathbb{F}_2^n \to \mathbb{F}_2^m$ again be an S-box and let k now be the bitslice gate complexity that we want to test for. Then our first set of equations C in ANF consists of:

- $\forall i \in \{0, \ldots, k-1\}$: $t_i = b_{3i} \cdot q_{2i} \cdot q_{2i+1} + b_{3i+1} \cdot q_{2i} + b_{3i+1} \cdot q_{2i+1} + b_{3i+2} + b_{3i+2} \cdot q_{2i}$, to encode the k `AND`, `OR`, `XOR` or `NOT` gates. The b_i determine what kind of gate this will represent, as can be seen in Table 3.2.

- $\forall i \in \{0, \ldots, k-1\}$: $0 = b_{3i} \cdot b_{3i+2}$ and $0 = b_{3i+1} \cdot b_{3i+2}$, to make sure that the gate is either a unary `NOT` or a binary `AND`, `OR`, or `XOR`, but not one of them combined with a `NOT`. This excludes `NAND`, `NOR`, and `XNOR` gates.

- $\forall i \in \{0,\dots,2k-1\}$: $q_i = \left(\sum_{j=0}^{n-1} a_{l+j} \cdot x_j\right) + \left(\sum_{j=0}^{\lfloor \frac{i}{2} \rfloor -1} a_{l+n+j} \cdot t_j\right)$, where $l = in + \left\lfloor \frac{i^2-2i+1}{4} \right\rfloor$, to encode that the inputs of the gates can be any S-box input bit or any previously computed bit.
- $\forall i \in \{0,\dots,2k-1\}, \forall j \in \{l,\dots,l+n+\lfloor \frac{i}{2} \rfloor -2\}, \forall u \in \{j+1,\dots,l+n+\lfloor \frac{i}{2} \rfloor -1\}$: $0 = a_j \cdot a_u$, to encode an *at-most-one* constraint on the gate inputs.
- $\forall i \in \{0,\dots,m-1\}$: $y_i = \left(\sum_{j=0}^{n-1} a_{s+j} \cdot x_j\right) + \left(\sum_{j=0}^{k-1} a_{s+n+j} \cdot t_j\right)$, where $s = 2kn + k(k-1) + i(n+k)$, to encode that the S-box output bit can be any S-box input bit or any gate output.
- $\forall i \in \{0,\dots,m-1\}, \forall j \in \{s,\dots,s+n+k-2\}, \forall u \in \{j+1,\dots,s+n+k-1\}$: $0 = a_j \cdot a_u$, to encode an *at-most-one* constraint on the S-box outputs.

Table 3.2: Gate modifiers for bitslice gate complexity.

$b_{3i}b_{3i+1}b_{3i+2}$	Gate t_i function
`000`	0
`001`	$\neg q_{2i}$
`010`	$q_{2i} \oplus q_{2i+1}$
`011`	Prevented by constraint on b_{3i+2}
`100`	$q_{2i} \wedge q_{2i+1}$
`101`	Prevented by constraint on b_{3i+2}
`110`	$q_{2i} \vee q_{2i+1}$
`111`	Prevented by constraint on b_{3i+2}

Converting C to C' and then to CNF is the same process as with the multiplicative-complexity decision problem. Note that the constraint equations on a_i and b_j do not have to be duplicated 2^n times for C', as they are not renumbered. This saves a lot of redundant clauses.

Results. As the amount of CNF clauses that is necessary to describe the bitslice-gate-complexity decision problem becomes much larger compared to the multiplicative-complexity decision problem, it can take much more time for a SAT solver to actually solve a problem instance. Still, for some 4-bit and 5-bit S-boxes results can be obtained within minutes or within a few hours. Table 3.3 contains some examples. If a bitslice gate complexity is listed as $\leq k$, a solution was found for k, but we were unable to prove that this is the minimum because the SAT solver did not terminate within a reasonable amount of time for $k-1$. Similarly, in some cases a lower bound was found because the SAT solver was able to determine that no circuit exists with that numbers of gates. The actual implementations with the given number of operations can be found in the original publication [Sto16b].

Table 3.3: Bitslice gate complexity of S-boxes.

S-box	Size	Bitslice gate complexity	Implementation
Keccak/Ketje/Keyak	5	≤ 13	3 `AND`, 2 `OR`, 5 `XOR`, 3 `NOT`
Joltik/Piccolo	4	10	1 `AND`, 3 `OR`, 4 `XOR`, 2 `NOT`
Joltik^{-1} /Piccolo^{-1}	4	10	1 `AND`, 3 `OR`, 4 `XOR`, 2 `NOT`
LAC	4	11	2 `AND`, 2 `OR`, 6 `XOR`, 1 `NOT`
Minalpher	4	≥ 11	
Prøst	4	8	4 `AND`, 4 `XOR`
RECTANGLE	4	$\in \{10, 11, 12\}$	4 `OR`, 7 `XOR`, 1 `NOT`
RECTANGLE^{-1}	4	$\in \{11, 12\}$	1 `AND`, 3 `OR`, 7 `XOR`, 1 `NOT`

For Prøst, it is interesting to note that the SAT solvers are able to find the same implementations as its authors already suggested. We have proven that their bitsliced implementation is indeed minimal.

3.3.4 Optimizing for Gate Complexity

A method to encode the gate-complexity decision problem was also provided by Courtois, Mourouzis, and Hulme [CMH13; Mou15], but again, actual results were only given for the CTC2 toy cipher. We show that it is feasible to compute the gate complexity for real-world 4-bit S-boxes as well.

The encoding is very similar to the bitslice-gate-complexity decision problem. The first set of equations C in ANF only differs in two places:

- Instead of the previous rule for t_i, the gates are encoded differently: $\forall i \in \{0, \ldots, k-1\}$: $t_i = b_{3i} \cdot q_{2i} \cdot q_{2i+1} + b_{3i+1} \cdot q_{2i} + b_{3i+1} \cdot q_{2i+1} + b_{3i+2}$, to encode the k gates. The b_i determine what kind of gate this will represent, as can be seen in Table 3.4.
- The additional constraints on the b_i are completely omitted.

Converting C to C' and then to CNF is similar to the previous optimization goals.

Table 3.4: Gate modifiers for gate complexity.

$b_{3i}b_{3i+1}b_{3i+2}$	Gate t_i function
`000`	0
`001`	1
`010`	$q_{2i} \oplus q_{2i+1}$
`011`	$q_{2i} \leftrightarrow q_{2i+1}$
`100`	$q_{2i} \wedge q_{2i+1}$
`101`	$q_{2i} \uparrow q_{2i+1}$
`110`	$q_{2i} \vee q_{2i+1}$
`111`	$q_{2i} \downarrow q_{2i+1}$

Results. Our results on real-world 4-bit S-boxes are summarized in Table 3.5. The full implementations can be found in the original publication [Sto16b]. For the studied 5-bit S-boxes we did not manage to retrieve results. Note that all types of logic gates are considered equally expensive. There is no type of gate that is preferred over the other, because information such as differences in area consumption or time delay are not taken into account. The implementations found by the SAT solver should therefore not be used directly for hardware implementations. However, they serve as an optimal starting point from where to swap "expensive" gates for cheaper ones, depending on the specific technology that is to be used. For example, the designers of Piccolo suggested a hardware implementation [SIH+11] of their S-box that may or may not be more efficient than the implementation given here, depending on the specific technology.

Table 3.5: Gate complexity of 4-bit S-boxes.

S-box	Gate complexity	Implementation
Joltik/Piccolo	8	2 `OR`, 1 `XOR`, 2 `NOR`, 3 `XNOR`
Joltik^{-1} /Piccolo^{-1}	8	2 `OR`, 1 `XOR`, 2 `NOR`, 3 `XNOR`
LAC	10	1 `AND`, 3 `OR`, 2 `XOR`, 4 `XNOR`
Prøst	8	4 `AND`, 4 `XOR`
RECTANGLE	$\in \{10, 11\}$	1 `AND`, 1 `OR`, 6 `XOR`, 1 `NAND`, 1 `NOR`, 1 `XNOR`
RECTANGLE^{-1}	$\in \{10, 11\}$	1 `AND`, 1 `OR`, 2 `XOR`, 1 `NAND`, 1 `NOR`, 5 `XNOR`

3.3.5 Optimizing for Depth Complexity

There are many situations in high-speed hardware implementations where the implementer wants to keep the depth of the circuit as low as possible, in order to be able to increase the clock frequency, without having to use significantly more gates. We provide a novel method to find low-depth implementations of small functions such as S-boxes using SAT solvers. This method is inspired by the encoding of the gate-complexity decision problem, but modified in some important ways.

In the encoding of the gate-complexity decision problem, we expressed that every gate can use the S-box input and the outputs of previous gates as its input. The key idea here is to divide the circuit into *depth layers* and to encode the notion that a gate can only use the S-box input and the output of gates in the previous layers as its input. This is made more precise later.

First we note that it is useful to limit the potential increase in the number of gates when reducing the depth of a circuit, to simplify the encoding and to limit the search space. We introduce a fixed maximum layer width w to address this, so we allow at most w gates to be executed in parallel. For some function f, we want to be able to answer questions such as: "is there a circuit implementing f with depth k and with at most w gates on each depth layer?".

Using this fixed maximum layer width, we make our encoding method more precise by once more creating a set C of multivariate equations over $\mathbb{F}_2$ in ANF that consists of:

- $\forall i \in \{0, \ldots, kw-1\}$: $t_i = b_{3i} \cdot q_{2i} \cdot q_{2i+1} + b_{3i+1} \cdot q_{2i} + b_{3i+1} \cdot q_{2i+1} + b_{3i+2}$, to encode the kw gates. The b_i determine what kind of gate this will represent, as can be seen in Table 3.4.
- $\forall i \in \{0, \ldots, 2kw-1\}$: $q_i = \left(\sum_{j=0}^{n-1} a_{l+j} \cdot x_j\right) + \left(\sum_{j=0}^{v-1} a_{l+n+j} \cdot t_j\right)$, where $v = \left\lfloor \frac{i}{2w} \right\rfloor w$ and $l = in + v\,(i - v - w)$, to encode that the inputs of the gates can be any S-box input bit or any previously computed bit.
- $\forall i \in \{0, \ldots, 2kw-1\}, \forall j \in \{l, \ldots, l+n+v-2\}, \forall u \in \{j+1, \ldots, l+n+v-1\}$: $0 = a_j \cdot a_u$, to encode an *at-most-one* constraint on the gate inputs.
- $\forall i \in \{0, \ldots, m-1\}$: $y_i = \left(\sum_{j=0}^{n-1} a_{s+j} \cdot x_j\right) + \left(\sum_{j=0}^{kw-1} a_{s+n+j} \cdot t_j\right)$, where $s = kw(2n + kw - w) + i(n + kw)$, to encode that the S-box output bit can be any S-box input bit or any gate output.

- $\forall i \in \{0,\ldots,m-1\}, \forall j \in \{s,\ldots,s+n+kw-2\}, \forall u \in \{j+1,\ldots,s+n+kw-1\}$: $0 = a_j \cdot a_u$, to encode an *at-most-one* constraint on the S-box outputs.

Converting C to C' and subsequently expressing this in CNF is again the same process as before.

Results. Using our method, we are able to find low-depth implementations for our 4-bit S-boxes. The results are summarized in Table 3.6 and the corresponding implementations can be found in the original publication [Sto16b]. The last column in Table 3.6 lists scenarios that yield `UNSAT`, to show boundaries on what is possible. The trade-off between circuit depth and the number of gates is made here in such a way that reducing the depth by 1 would imply the implementation to have at least twice as many gates as is required by the gate complexity.

Table 3.6: Depth complexity of 4-bit S-boxes.

S-box	k	w	Implementation	`UNSAT` boundaries
Joltik/Piccolo	4	2	2 `OR`, 1 `XOR`,	$k = 4, w = 1$
			2 `NOR`, 3 `XNOR`	$k = 3, w = 10$
Joltik^{-1} /Piccolo^{-1}	4	3	3 `OR`, 5 `XOR`,	$k = 4, w = 2$
			1 `NOR`, 3 `XNOR`	$k = 3, w = 10$
LAC	3	6	3 `OR`, 4 `XOR`,	$k = 3, w = 4$
			4 `NAND`, 4 `XNOR`	$k = 2, w = 10$
Prøst	4	3	4 `AND`, 1 `OR`, 4 `XOR`,	$k = 4, w = 2$
			1 `NAND`, 1 `XNOR`	$k = 3, w = 10$
RECTANGLE	3	6	1 `OR`, 8 `XOR`,	$k = 3, w = 4$
			3 `NAND`, 2 `NOR`, 2 `XNOR`	$k = 2, w = 10$
RECTANGLE^{-1}	3	6	2 `AND`, 3 `OR`, 5 `XOR`,	$k = 3, w = 4$
			1 `NAND`, 1 `NOR`, 3 `XNOR`	$k = 2, w = 10$

3.4 Combining Criteria: Optimizing the PRIMATEs S-box

So far, we have seen how to optimize for one specific goal. However, a result that is optimized for multiplicative complexity may contain more `XOR` gates than is desired, and a result that is optimized for gate complexity may contain more nonlinear gates than is desired for a masked implementation. Here we show how multiple optimization goals can be combined by looking at the 5-bit PRIMATEs S-box. We first optimize for multiplicative complexity to have a minimal number of nonlinear gates, and subsequently we minimize the number of linear gates. The result is an implementation that has 4 `AND`, 3 `OR`, 31 `XOR`, and 5 `NOT` gates.

Listing 3.1: PRIMATEs S-box after optimizing for multiplicative complexity.

$$\begin{aligned}
q_0 &= x_0 \oplus x_3 \\
q_1 &= x_1 \\
t_0 &= q_0 \vee q_1 \\
q_2 &= \neg(x_1 \oplus x_3) \\
q_3 &= x_0 \oplus x_2 \\
t_1 &= q_2 \wedge q_3 \\
q_4 &= x_0 \oplus x_1 \oplus x_4 \\
q_5 &= x_0 \oplus x_2 \oplus x_3 \\
t_2 &= q_4 \wedge q_5 \\
q_6 &= \neg(x_0 \oplus x_2 \oplus x_3 \oplus x_4) \\
q_7 &= x_1 \oplus x_2 \oplus x_4 \\
t_3 &= q_6 \vee q_7 \\
q_8 &= x_0 \oplus x_1 \oplus x_2 \oplus x_3 \oplus x_4
\end{aligned}$$

$$\begin{aligned}
q_9 &= x_2 \oplus t_0 \oplus t_3 \\
t_4 &= q_8 \wedge q_9 \\
q_{10} &= x_0 \oplus x_3 \oplus x_4 \\
q_{11} &= \neg(x_0 \oplus x_4) \\
t_5 &= q_{10} \vee q_{11} \\
q_{12} &= \neg(x_1 \oplus x_2 \oplus t_0 \oplus t_2 \oplus t_3 \oplus t_4) \\
q_{13} &= x_2 \oplus x_3 \\
t_6 &= q_{12} \wedge q_{13} \\
y_0 &= x_1 \oplus x_3 \oplus t_2 \oplus t_3 \oplus t_5 \oplus t_6 \\
y_1 &= x_0 \oplus x_4 \oplus t_1 \oplus t_2 \oplus t_3 \oplus t_4 \oplus t_5 \oplus t_6 \\
y_2 &= x_1 \oplus x_2 \oplus x_4 \oplus t_1 \oplus t_3 \oplus t_4 \oplus t_5 \\
y_3 &= x_0 \oplus x_2 \oplus x_3 \oplus x_4 \oplus t_3 \oplus t_4 \oplus t_5 \oplus t_6 \\
y_4 &= \neg(x_2 \oplus t_0 \oplus t_2 \oplus t_3 \oplus t_4 \oplus t_5 \oplus t_6)
\end{aligned}$$

When the optimization method for multiplicative complexity is applied, we find a solution with multiplicative complexity 7 as shown in listing 3.1. It is not hard to see that there are a lot of redundant `XOR` operations in this implementation. We distinguish between `XOR` operations before the nonlinear gates (on x_i) and `XOR` operations after the nonlinear gates (on t_i). It is possible to view them as two straight-line linear programs, where the first describes the linear part of the S-box approached from the input and the second describes the linear part approached from the S-box output.

The problem of finding the shortest linear program Ax can be given by $x = (x_0, x_1, x_2, x_3, x_4)^\intercal$ and

$$A = \begin{array}{cc} \begin{array}{c} q_0 \\ q_1 \\ q_2 \\ q_3 \\ q_4 \\ q_5 \\ q_6 \\ q_7 \\ q_8 \\ q_9 \\ q_{10} \\ q_{11} \\ q_{12} \\ q_{13} \\ y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{array} & \begin{pmatrix} 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 \end{pmatrix} \end{array}.$$

The shortest linear straight-line program problem $A'x'$ can be given by $x' = (t_0, t_1, t_2, t_3, t_4, t_5, t_6)^\intercal$ and

$$A' = \begin{array}{c} q_9 \\ q_{12} \\ y_0 \\ y_1 \\ y_2 \\ y_3 \\ y_4 \end{array} \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}.$$

We are able to find a minimal straight-line program computing $A'x'$ using SAT solvers. We use the method suggested by Fuhs and Schneider-Kamp [FS10] to encode the SLP problem as a SAT instance in CNF. This yields a result that is incorporated in our implementation of the PRIMATEs S-box. Finding a minimal straight-line program computing Ax turned out to be infeasible using SAT solvers within a reasonable amount of time. Therefore, we apply the heuristic approach as suggested by Boyar and Peralta [BP10]. This does provide us with a short straight-line program. We combine both results and amend the original PRIMATEs S-box implementation to get a more efficient implementation. We are able to decrease the previous result of 58 `XOR` gates to only 31 `XOR` gates, as shown in listing 3.2, where z_i represent helper variables.

Tools. We provide tools to generate C' in ANF for all discussed optimization goals and to convert a SAT solver solution back to an S-box implementation. We place those tools into the public domain. They and additional documentation are available online at `https://github.com/Ko-/sboxoptimization`.

Listing 3.2: PRIMATEs S-box with 31 `XOR` gates.

$$\begin{aligned}
z_0 &= x_0 \oplus x_4 \\
z_1 &= x_1 \oplus x_2 \\
z_2 &= x_2 \oplus x_3 \\
q_0 &= x_0 \oplus x_3 \\
t_0 &= q_0 \vee x_1 \\
q_2 &= x_1 \oplus x_3 \\
q_3 &= \neg(x_0 \oplus x_2) \\
t_1 &= q_2 \vee q_3 \\
q_4 &= x_1 \oplus z_0 \\
q_5 &= x_0 \oplus z_2 \\
t_2 &= q_4 \wedge q_5 \\
q_6 &= \neg(x_4 \oplus q_5)
\end{aligned}$$

$$\begin{aligned}
q_7 &= x_4 \oplus z_1 \\
t_3 &= q_6 \vee q_7 \\
q_8 &= q_4 \oplus z_2 \\
z_9 &= t_0 \oplus t_3 \\
q_9 &= x_2 \oplus z_9 \\
t_4 &= q_8 \wedge q_9 \\
q_{10} &= \neg(x_3 \oplus z_0) \\
t_5 &= q_{10} \wedge z_0 \\
q_{12} &= \neg(z_1 \oplus z_9 \oplus t_2 \oplus t_4) \\
t_6 &= q_{12} \wedge z_2 \\
z_3 &= t_5 \oplus t_6 \\
z_4 &= t_3 \oplus z_3
\end{aligned}$$

$$\begin{aligned}
z_5 &= t_2 \oplus z_4 \\
z_6 &= t_1 \oplus t_6 \\
z_7 &= t_4 \oplus z_5 \\
z_8 &= t_1 \oplus z_7 \\
z_{10} &= t_0 \oplus z_7 \\
z_{11} &= t_4 \oplus z_4 \\
z_{12} &= z_6 \oplus z_{11} \\
y_0 &= \neg(q_2 \oplus z_5) \\
y_1 &= z_0 \oplus z_8 \\
y_2 &= q_7 \oplus z_{12} \\
y_3 &= q_6 \oplus z_{11} \\
y_4 &= x_2 \oplus z_{10}
\end{aligned}$$

3.5 Conclusion

SAT solvers can be used to find minimal implementations for small functions such as S-boxes with respect to criteria as the multiplicative complexity, bitslice gate complexity, gate complexity, and circuit-depth complexity. We have shown how this can be done and how multiple criteria can be combined. However, for 8-bit S-boxes and larger functions these methods quickly become infeasible. One will then have to resort to approaches based on heuristics.

Chapter 4

MDS Matrices

As a second cryptographic building block, we consider maximum-distance separable (MDS) matrices, which are commonly chosen as linear mixing layer in round functions of iterated permutations and ciphers. This chapter argues that researchers should focus on global optimization instead of local optimization and applies existing techniques from a different line of research to achieve lower `XOR` *counts. The original publication [KLSW17] contained a few wrong numbers that were later discovered and corrected by co-authors Thorsten Kranz and Friedrich Wiemer. Another wrong number for Toeplitz matrices caused by a typo in code was pointed out by Mohsen Mousavi. In this chapter those mistakes have been fixed. Another modification compared to the original publication is that the appendix has become a regular section.*

4.1 Introduction

Lightweight cryptography has been a major trend in symmetric cryptography for the last years. While it is not always exactly clear what lightweight cryptography actually is, the main goal can be summarized as very efficient cryptography. Here, the meaning of efficiency ranges from small chip size to low latency and low energy.

As part of this line of work, several researchers started to optimize the construction of many parts of block ciphers, with a special focus on the linear layers more recently and even more specifically the implementation of MDS matrices. That is, linear layers with an optimal branch number.

The first line of work focused solely on minimizing the chip area of the implementation. This started with the block cipher PRESENT [BKL+07] and goes over to many more designs, such as LED [GPPR11] and the hash function

PHOTON [GPP11], where in the latter MDS matrices were constructed that are especially optimized for chip area by allowing a serialized implementation. However, there seem to be only a few practical applications where a small chip area is the only optimization goal and for those applications very good solutions are already available by now.

Later, starting with FOAM [KPPY14], researchers focused on round-based implementations with the goal of finding MDS constructions that minimize the number of `XOR` operations needed for their implementation. Initially, the number of `XOR` operations needed was bounded by the number of ones in the binary representation of the matrix.

However, as the number of ones only gives an upper bound on the number of required `XOR`s, several papers started to deviate from this conceptually easier but less accurate definition of `XOR` count and started to consider more efficient ways of implementing MDS matrices. Considering an $n \times n$ MDS matrix over a finite field $\mathbb{F}_{2^k}$ given as $M = (m_{i,j})$ the aim was to choose the elements $m_{i,j}$ in such a way that implementing all of the multiplications $x \mapsto m_{i,j}x$ in parallel becomes as cheap as possible. In order to compute the matrix M entirely, those partial results have to be added together, for which an additional amount of `XOR`s is required. It became common to denote the former cost as the overhead and the later cost, i. e., the cost of combining the partial results as a fixed, incompressible part. A whole series of papers [BKL16; JPST17; LS16; LW16; LW17; SKOP15; SS16a; SS16b; SS17; ZWS17] managed to reduce the overhead.

From a different viewpoint, what happened was that parts of the matrix, namely the cost of multiplication with the $m_{i,j}$, were extensively optimized, while taking the overall part of combining the parts as a given. That is, researchers have focused on local optimization instead of global optimization.

Indeed the task of globally optimizing is far from trivial, and thus the local optimization is a good step forward.

Interestingly, trying to lower the cost of implementing the multiplication with a relatively large, e. g., 32×32 binary matrix, is another extensively studied line of research. It is known that the problem is NP-hard [BMP08; BMP13] and thus quickly renders infeasible for increasing matrix dimension. However, a number of heuristic algorithms for finding the shortest linear straight-line program, which exactly corresponds to minimizing the number of `XOR`s, have been proposed in the literature [BFP19; BMP08; BMP13; BP10; FS10; FS12; Paa97; VSP17]. Those algorithms produce competitive results with a reasonable running time for arbitrary binary matrices of dimension up to at least 32.

Thus, the natural next step in order to optimize the cost of implementing MDS matrices is to combine those two approaches. This is exactly what we are doing in our work. Our contribution, which we achieve by applying the heuristic algorithms to find a short linear straight-line program to the case of MDS matrices, is threefold.

First, we use several well-locally-optimized MDS matrices from the literature and apply the known algorithms to all of them. This is conceptually easy, with the main problem being the implementation of those algorithms. In order to simplify this for follow-up works, we make our implementation publicly available.

This simple application immediately leads to significant improvements. For instance, we get an implementation of the AES MixColumn matrix that outperformed all implementations in the literature at the time of the original publication [KLSW17], i. e., we use 97 `XOR`s while the best implementation before used 103 `XOR`s [JMPS17]. This result has later been improved and the best result is currently at 92 `XOR`s [Max19]. In the case of applying it to the other constructions, we often get an implementation using *fewer `XOR` operations than what was considered fixed cost before*. That is, when (artificially) computing it, the overhead would actually be negative. This confirms our intuition that the overhead was already very well optimized in previous work, such that now optimizing globally is much more meaningful.

Second, we took a closer look at how the previous constructions compare when being globally optimized. Interestingly, the previous best construction,

i. e., the MDS matrix with smallest overhead, was most of the time *not the one with the fewest* `XOR`*s*. Thus, with respect to the global optimum, the natural question was, which known construction actually performs best. In order to analyze that, we performed extensive experimental computations to compare the distribution of the optimized implementation cost for the various constructions. The, somewhat disappointing, result is that all known constructions behave basically the same. The one remarkable exception is the subfield construction for MDS matrices, first introduced in Whirlwind [BNN+10].

Third, we looked at finding matrices that perform exceptionally well with respect to the global optimization, i. e., which can be implemented with an exceptionally low *total* number of `XOR`s. Those results are summarized in Table 4.1. Compared to previously known matrices, ours improve on all – with the exception of one, where the best known matrix is the already published matrix from [SS16b].

Table 4.1: Best known MDS matrices. Matrices in the lower half are involutory.

Type	Previously best known		New best known	
$\mathrm{GL}(4,\mathbb{F}_2)^{4\times4}$	58	[JPST17; SS16b]	36*	Eq. (4.1) (Hadamard)
$\mathrm{GL}(8,\mathbb{F}_2)^{4\times4}$	106	[LW16]	72	Eq. (4.2) (Subfield)
$(\mathbb{F}_2[x]/\mathtt{0x13})^{8\times8}$	384	[SKOP15]	196†	Eq. (4.3) (Cauchy)
$\mathrm{GL}(8,\mathbb{F}_2)^{8\times8}$	640	[LS16]	392	Eq. (4.4) (Subfield)
$(\mathbb{F}_2[x]/\mathtt{0x13})^{4\times4}$	63	[JPST17]	42*	[SS16b]
$\mathrm{GL}(8,\mathbb{F}_2)^{4\times4}$	126	[JPST17]	84	Eq. (4.5) (Subfield)
$(\mathbb{F}_2[x]/\mathtt{0x13})^{8\times8}$	424	[SKOP15]	212†	Eq. (4.6) (Vandermonde)
$\mathrm{GL}(8,\mathbb{F}_2)^{8\times8}$	736	[JPST17]	424	Eq. (4.7) (Subfield)

* Computed with heuristic from [BMP13].

† Computed with heuristic from [Paa97].

Finally, we like to point out two restrictions of our approach. First, we do not try to minimize the amount of temporary registers needed for the implementation. Second, in line with all previous constructions, we do not minimize the circuit depth. The later restriction is out of scope of the current work but certainly an interesting task for the future.

All our implementations are publicly available on GitHub at
`https://github.com/rub-hgi/shorter_linear_slps_for_mds_matrices`.

4.2 Preliminaries

Before getting into details about the `XOR` count and previous work, let us recall some basic notations on finite fields [LN97], their representations [War94], and on matrix constructions.

4.2.1 Basic Notations

$\mathbb{F}_{2^k}$ is the finite field with 2^k elements, often also denoted as GF(2^k). Up to isomorphism, every field with 2^k elements is equal to the polynomial ring over $\mathbb{F}_2$ modulo an irreducible polynomial q of degree k: $\mathbb{F}_{2^k} \cong \mathbb{F}_2[x]/q$. In favor of a more compact notation, we stick to the common habit and write a polynomial as its coefficient vector interpreted as a hexadecimal number, i. e., $x^4 + x + 1$ corresponds to `0x13`.

It is well known that we can represent the elements in a finite field with characteristic 2 as vectors with coefficients in $\mathbb{F}_2$. More precisely, there exists a vector-space isomorphism $\Phi : \mathbb{F}_{2^k} \to \mathbb{F}_2^k$. Every multiplication by an element $\alpha \in \mathbb{F}_{2^k}$ can then be described by a left-multiplication with a matrix $T_\alpha \in \mathbb{F}_2^{k \times k}$ as shown in the following diagram.

$$\begin{tikzcd}
\mathbb{F}_{2^k} \arrow[r, "\cdot\alpha"] \arrow[d, "\Phi"'] & \mathbb{F}_{2^k} \\
\mathbb{F}_2^k \arrow[r, "T_\alpha"] & \mathbb{F}_2^k \arrow[u, "\Phi^{-1}"']
\end{tikzcd}$$

T_α is usually called the multiplication matrix of the element α. Given an $n \times n$ matrix $M = (\alpha_{i,j})$ with $\alpha_{i,j} \in \mathbb{F}_{2^k}$ for $1 \leq i, j \leq n$, we define $\mathcal{B}(M) := (T_{\alpha_{i,j}}) \subseteq \mathrm{GL}(k, \mathbb{F}_2)^{n\times n} \subseteq (\mathbb{F}_2^{k\times k})^{n\times n} \cong \mathbb{F}_2^{nk\times nk}$. Its corresponding binary $nk \times nk$ matrix is called the *binary representation*. Here, $\mathrm{GL}(k, \mathbb{F}_2)$ denotes the general linear group, that is the group of invertible matrices over $\mathbb{F}_2$ of dimension $k \times k$.

Given a matrix M and a vector u, the Hamming weights $\mathrm{hw}(M)$ and $\mathrm{hw}(u)$ are defined as the number of nonzero entries in M and u, respectively. In the case of a binary vector $v \in \mathbb{F}_2^{nk}$, we define $\mathrm{hw}_k(v) := \mathrm{hw}(v')$, where $v' \in (\mathbb{F}_2^k)^n$ is the vector that has been constructed by partitioning v into groups of k bits. Furthermore, the branch number of a matrix M is defined as $\mathrm{bn}(M) := \min_{u\neq 0}\{\mathrm{hw}(u) + \mathrm{hw}(Mu)\}$. For a binary matrix $B \in \mathbb{F}_2^{nk\times nk}$, the branch number for k-bit words is defined as $\mathrm{bn}_k(B) := \min_{u\in\mathbb{F}_2^{nk}\setminus\{0\}}\{\mathrm{hw}_k(u) + \mathrm{hw}_k(Mu)\}$.

In the design of block ciphers, MDS matrices play an important role.

Definition 1. An $n \times n$ matrix M is *MDS* if and only if $\mathrm{bn}(M) = n + 1$.

It has been shown that a matrix is MDS if and only if all its square submatrices are invertible [MS77, page 321, Theorem 8]. MDS matrices do not exist for every choice of n, k. The exact parameters for which MDS matrices do or do not exist are investigated in the context of the famous MDS conjecture which was initiated in [Seg55]. For binary matrices, we need to modify Definition 1.

Definition 2. A binary matrix $B \in \mathbb{F}_2^{nk\times nk}$ is *MDS* for k-bit words if and only if $\mathrm{bn}_k(M) = n + 1$.

MDS matrices have a common application in linear layers of block ciphers, due to the wide-trail strategy proposed for AES, see [Dae95; DR02]. We typically deal with $n \times n$ MDS matrices over $\mathbb{F}_{2^k}$ respectively binary $\mathbb{F}_2^{nk \times nk}$ matrices that are MDS for k-bit words where $k \in \{4, 8\}$ is the size of the S-box. In either case, when we call a matrix MDS, the size of k will always be clear from the context when not explicitly mentioned.

It is easy to see that, if $M \in \mathbb{F}_{2^k}^{n \times n}$ is MDS, then also $\mathcal{B}(M)$ is MDS for k-bit words. On the other hand, there might also exist binary MDS matrices for k-bit words that have no according representation over $\mathbb{F}_{2^k}$.

Other, non-MDS matrices are also common in cipher designs. To name only a few examples: PRESENT's permutation matrix [BKL+07], lightweight implementable matrices from PRINCE [BCG+12], or PRIDE [ADK+14], or the recently used almost-MDS matrices, e. g., in Midori [BBI+15], or QARMA [Ava17].

4.2.2 MDS Constructions

Cauchy and Vandermonde matrices are two famous constructions for building MDS matrices. They have the advantage of being provably MDS.

However, as known from the MDS conjecture, for some parameter choices, MDS matrices are unlikely to exist. E. g., we do not know how to construct MDS matrices over $\mathbb{F}_{2^2}$ of dimension 4×4.

Definition 3 (Cauchy matrix). Given two disjoint sets of n elements of a field $\mathbb{F}_{2^k}$, $A = \{a_1, \dots, a_n\}$, and $B = \{b_1, \dots, b_n\}$ with $a_i - b_j \neq 0$. Then the following matrix is a *Cauchy* matrix.

$$M = \operatorname{cauchy}(a_1, \dots, a_n, b_1, \dots, b_n) := \begin{pmatrix} \frac{1}{a_1 - b_1} & \frac{1}{a_1 - b_2} & \cdots & \frac{1}{a_1 - b_n} \\ \frac{1}{a_2 - b_1} & \frac{1}{a_2 - b_2} & \cdots & \frac{1}{a_2 - b_n} \\ \vdots & & \ddots & \vdots \\ \frac{1}{a_n - b_1} & \frac{1}{a_n - b_2} & \cdots & \frac{1}{a_n - b_n} \end{pmatrix}$$

Every Cauchy matrix is MDS, e. g., see [GR13, Lemma 1].

Definition 4 (Vandermonde matrix)**.** Given an n-tuple $(a_1, \dots, a_n)$ with $a_i \in \mathbb{F}_{2^k}$. Then the following matrix is a *Vandermonde* matrix.

$$M = \mathrm{vandermonde}(a_1, \dots, a_n) := \begin{pmatrix} a_1^0 & a_1^1 & \cdots & a_1^{n-1} \\ a_2^0 & a_2^1 & \cdots & a_2^{n-1} \\ \vdots & & \ddots & \vdots \\ a_n^0 & a_n^1 & \cdots & a_n^{n-1} \end{pmatrix}$$

Given two Vandermonde matrices A and B with fully distinct a_i, b_j, then the matrix AB^{-1} is MDS, see [LF04, Theorem 2]. Furthermore, if $a_i = b_i + \Delta$ for all i and an arbitrary nonzero Δ, then the matrix AB^{-1} is also involutory [LF04; SDMO12].

4.2.3 Specially Structured Matrix Constructions

Other constructions, such as circulant, Hadamard, or Toeplitz, are not per se MDS, but they have the advantage that they greatly reduce the search space by restricting the number of submatrices that appear in the matrix. For circulant matrices, this was already noted by Daemen, Knudsen, and Rijmen [DKR97].

In order to generate a random MDS matrix with one of these constructions, we can choose random elements for the matrix and then check for the MDS condition. Because of many repeated submatrices, the probability to find an MDS matrix is much higher then for a fully random matrix.

Definition 5 (Circulant matrices)**.** A *right circulant* $n \times n$ matrix is defined by the elements of its first row $a_1, \dots, a_n$ as

$$M = \mathrm{circ_r}(a_1, \dots, a_n) := \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ a_n & a_1 & \cdots & a_{n-1} \\ \vdots & & \ddots & \vdots \\ a_2 & \cdots & a_n & a_1 \end{pmatrix}.$$

A *left circulant* $n \times n$ matrix is analogously defined as

$$M = \operatorname{circ}_\ell(a_1, \ldots, a_n) := \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ a_2 & a_3 & \cdots & a_1 \\ \vdots & & \ddots & \vdots \\ a_n & a_1 & \cdots & a_{n-1} \end{pmatrix}.$$

While in the literature circulant matrices are almost always right circulant, left circulant matrices are equally fine for cryptographic applications. The often noted advantage of right circulant matrices, the ability to implement the multiplication serialized and with shifts in order to save `XOR`s, of course also applies to left circulant matrices. Additionally, it is easy to see that $\operatorname{bn}(\operatorname{circ}_\mathrm{r}(a_1, \ldots, a_n)) = \operatorname{bn}(\operatorname{circ}_\ell(a_1, \ldots, a_n))$, since the matrices only differ in a permutation of the rows. Thus, for cryptographic purposes, it does not matter if a matrix is right circulant or left circulant and we will therefore simply talk about circulant matrices in general. The common practice of restricting the matrix entries to elements from a finite field comes with the problem that circulant involutory MDS matrices over finite fields do not exist; see [JA09]. However, Li and Wang [LW16] showed that this can be avoided by taking the matrix elements from the general linear group.

Definition 6 (Hadamard matrix)**.** A *(finite field) Hadamard* matrix M over $\mathbb{F}_{2^k}$ is of the form

$$M = \begin{pmatrix} M_1 & M_2 \\ M_2 & M_1 \end{pmatrix},$$

where M_1 and M_2 are either Hadamard matrices themselves or one-dimensional [Bea75].

The biggest advantage of Hadamard matrices is the possibility to construct involutory matrices. If we choose the elements of our matrix such that the first row sums to one, the resulting matrix is involutory; see [GR13].

Definition 7 (Toeplitz matrix)**.** An $n \times n$ *Toeplitz* matrix M is defined by the elements of its first row $a_1, \dots, a_n$ and its first column $a_1, a_{n+1}, \dots, a_{2n-1}$ as

$$M = \mathrm{toep}(a_1, \dots, a_n, a_{n+1}, \dots, a_{2n-1}) := \begin{pmatrix} a_1 & a_2 & \cdots & a_n \\ a_{n+1} & a_1 & \ddots & a_{n-1} \\ \vdots & \ddots & \ddots & \vdots \\ a_{2n-1} & a_{2n-2} & \cdots & a_1 \end{pmatrix},$$

that is, every element defines one minor diagonal of the matrix.

To the best of our knowledge, Sarkar and Syed [SS16b] were the first to scrutinize Toeplitz matrices in the context of `XOR` counts.

Finally, the subfield construction was first used to construct lightweight linear layers in the Whirlwind hash function [BNN+10, Section 2.2.2] and later used in [ADK+14; CYK+12; JPST17; KPPY14; SKOP15]. As its name suggests, the subfield construction was originally defined only for matrices over finite fields: a matrix with coefficients in $\mathbb{F}_{2^k}$ can be used to construct a matrix with coefficients in $\mathbb{F}_{2^{2k}}$. Here, we use the natural extension to binary matrices.

Definition 8 (Subfield matrix)**.** Given an $n \times n$ matrix M with entries $m_{i,j} \in \mathbb{F}_2^{k \times k}$. The subfield construction of M is then an $n \times n$ matrix M' with

$$M' = \mathrm{subfield}(M) := \left(m'_{i,j}\right),$$

where each $m'_{i,j} = \begin{pmatrix} m_{i,j} & 0 \\ 0 & m_{i,j} \end{pmatrix} \in \mathbb{F}_2^{2k \times 2k}$.

This definition is straightforward to extend for more than one copy of the matrix M. The subfield construction has some very useful properties, see [BNN+10; JPST17; KPPY14; SKOP15].

Lemma 1. *For the subfield construction, the following properties hold:*

1. *Let M be a matrix that can be implemented with m `XOR`s. Then the matrix $M' = \mathrm{subfield}(M)$ can be implemented with $2m$ `XOR`s.*

2. *Let M be an MDS matrix for k-bit words. Then $M' = \text{subfield}(M)$ is MDS for $2k$-bit words.*

3. *Let M be an involutory matrix. Then $M' = \text{subfield}(M)$ is also involutory.*

Proof.

(1) Due to the special structure of the subfield construction, we can split the multiplication by M' into two multiplications by M, each on one half of the input bits. Hence, the `XOR` count doubles.

(2) We want to show that $\text{hw}_{2k}(u) + \text{hw}_{2k}(M'u) \geq n + 1$ for every nonzero u. We split u into two parts u_1 and u_2, each containing alternating halves of the elements of u. As described in [KPPY14], the multiplication of M' and u is the same as the multiplication of the original matrix M and each of the two u_i, if we combine the results according to our splitting. Let $t = \text{hw}_{2k}(u) > 0$. Then, we have $t \geq \text{hw}_k(u_1)$ and $t \geq \text{hw}_k(u_2)$. Without loss of generality, let $\text{hw}_k(u_1) > 0$. Since M is MDS for k-bit words, we have $\text{hw}_k(Mu_1) \geq n - t + 1$ which directly leads to $\text{hw}_{2k}(M'u) \geq n - t + 1$.

(3) As in the above proof, this property is straightforward to see. We want to show that $M'M'u = u$ for any vector u. Again, we split u into two parts, u_1 and u_2, each containing alternating halves of the elements of u. Now, we need to show that $MMu_i = u_i$. This trivially holds, as M is involutory.

□

With respect to cryptographic designs, this means the following: assume we have a linear straight-line program with m `XOR`s for an (involutory) $n \times n$ MDS matrix and k-bit S-boxes. We can then easily construct a linear straight-line program with $2m$ `XOR`s for an (involutory) $n \times n$ MDS matrix and $2k$-bit S-boxes.

4.3 Related Work

In 2014, [KPPY14] introduced the notion of `XOR` count as a metric to compare the area-efficiency of matrix multiplications. Following that, there has been a lot of work [BKL16; JPST17; LS16; LW16; LW17; SKOP15; SS16a; SS16b; SS17; ZWS17] on finding MDS matrices that can be implemented with as few `XOR` gates as possible in the round-based scenario.

In an independent line of research, the problem of implementing binary matrix multiplications with as few `XOR`s as possible was extensively studied [BFP19; BMP08; BMP13; BP10; FS10; FS12; Paa97; VSP17].

In this section, we depict these two fields of research and show how they can be combined.

4.3.1 Local Optimizations

Let us first recall the scenario. In a round-based implementation the matrix is implemented as a fully unrolled circuit. Thus, in the `XOR` count metric, the goal is to find a matrix that can be implemented with a circuit of as few (2-input) `XOR` gates as possible. Of course, the matrix has to fulfill some criteria, typically it is MDS. For finding matrices with a low `XOR` count, the question of how to create a circuit for a given matrix must be answered.

The usual way for finding an implementation of $n \times n$ matrices over $\mathbb{F}_{2^k}$ was introduced in [KPPY14]. As each of the n output components of a matrix-vector multiplication is computed as a sum over n products, the implementation is divided into two parts. First there are the single multiplications and then there is the addition of the products. As $\mathbb{F}_{2^k} \cong \mathbb{F}_2^k$, an addition of two elements from $\mathbb{F}_2^k$ requires k `XOR`s and thus adding up the products for all rows requires $n(n-1)k$ `XOR`s in the case of an MDS matrix where every element is nonzero. If one implements the matrix like this, these $n(n-1)k$ `XOR`s are a fixed part that cannot be changed. Accordingly, many papers [BKL16; JPST17; LS16; LW16;

ZWS17] just state the number of `XOR`s for the single field multiplications when presenting results. The other costs are regarded as inevitable. The goal then boils down to constructing matrices with elements for which multiplication can be implemented with few `XOR`s. Thus, the original goal of finding a global implementation for the matrix is approached by locally looking at the single matrix elements.

To count the number of `XOR`s for implementing a single multiplication with an element $\alpha \in \mathbb{F}_{2^k}$, the multiplication matrix $T_\alpha \in \mathbb{F}_2^{k\times k}$ is considered. Such a matrix can be implemented in a straightforward way with $\mathrm{hw}(T_\alpha) - k$ `XOR`s by simply implementing every `XOR` of the output components. We call this the *naive* implementation of a matrix and when talking about the naive `XOR` count of a matrix, we mean the $\mathrm{hw}(T_\alpha) - k$ `XOR`s required for the naive implementation. In [JPST17], this is called d-`XOR`. It is the easiest and most frequently used method of counting `XOR`s. Of course, in the same way we can also count the `XOR`s of other matrices over $\mathbb{F}_2^{k\times k}$, i. e., also matrices that were not originally defined over finite fields.

For improving the `XOR` count of the single multiplications, two methods have been introduced. First, if the matrix is defined over some finite field, one can consider different field representations that lead to different multiplication matrices with potentially different Hamming weights, see [BKL16; SKOP15; SS16a]. Second, by reusing intermediate results, a $k \times k$ binary matrix might be implemented with less than $\mathrm{hw}(M) - k$ `XOR`s, see [BKL16; JPST17]. In [JPST17], this is called s-`XOR`. The according definitions from [JPST17] and [BKL16] require that all operations must be carried out on the input registers. That is, in contrast to the naive `XOR` count, no temporary registers are allowed. However, as we consider round-based hardware implementations, there is no need to avoid temporary registers since these are merely wires between gates.

Nowadays, the `XOR` count of implementations is mainly dominated by the $n(n-1)k$ `XOR`s for putting together the locally optimized multiplications. Lastly, we seem to hit a threshold and new results often improve existing results only by very few `XOR`s. The next natural step is to shift the focus from local optimization of the single elements to the global optimization of the whole matrix. This was also formulated as future work in [JPST17]. As described in Section 4.2, we can use the binary representation to write an $n \times n$ matrix over $\mathbb{F}_{2^k}$ as a binary $nk \times nk$ matrix. First we note that the naive `XOR` count of the binary representation is exactly the naive `XOR` count of implementing each element multiplication and subsequently adding the results. However, if we consider the optimization technique of reusing intermediate results for the whole $nk \times nk$ matrix, there are many more degrees of freedom. For the MixColumn matrix there already exists some work that goes beyond local optimization. An implementation with 108 `XOR`s has been presented in [BBR16a; BBR16b; SMTM01] and an implementation with 103 `XOR`s in [JMPS17]. A first step to a global optimization algorithm was taken in [ZWZZ16]. However, their heuristic did not yield very good results and they finally had to go back to optimizing submatrices.

Interestingly, much better algorithms for exactly this problem are already known from a different line of research.

4.3.2 Global Optimizations

Implementing binary matrices with as few `XOR` operations as possible is also known as the problem of finding the *shortest linear straight-line program* [BMP13; FS10] over the finite field with two elements. Although this problem is NP-hard [BMP08; BMP13], attempts have been made to find exact solutions for the minimum number of `XOR`s. Fuhs and Schneider-Kamp [FS10; FS12] suggested to reduce the problem to satisfiability of Boolean logic. They presented a general encoding scheme for deciding if a matrix can be implemented with a certain number of `XOR`s. Now, for finding the optimal implementation, they repeatedly use SAT solvers for a decreasing number of `XOR`s. Then, when they know that a matrix can be implemented with ℓ `XOR`s, but cannot be implemented with $\ell - 1$ `XOR`s, they are able to present ℓ as the optimal `XOR` count. They used this technique to search for the minimum number of `XOR`s necessary to compute a binary matrix of size 21×8, which is the first linear part of the AES S-box, when it is decomposed into two linear parts and a minimal nonlinear core. While it worked to find a solution with 23 `XOR`s and to show that no solution with 20 `XOR`s exists, it turned out to be infeasible to prove that a solution with 22 `XOR`s does not exist and that 23 is therefore the minimum. In general, this approach quickly becomes infeasible for larger matrices. In Chapter 3 we applied it successfully to a small 7×7 matrix, but we did not manage to find a provably minimal solution with a specific matrix of size 19×5. However, there do exist heuristics to efficiently find short linear straight-line programs also for larger binary matrices.

Back in 1997, Paar [Paa97] studied how to optimize the arithmetic used by Reed-Solomon encoders. Essentially, this boils down to reducing the number of `XOR`s that are necessary for a constant multiplier over the field $\mathbb{F}_{2^k}$. Paar described two algorithms that find a local optimum. Intuitively, the idea of the algorithms is to iteratively eliminate *common subexpressions*. Let T_α be the multiplication matrix, to be applied to a variable $x = (x_1, \ldots, x_k) \in \mathbb{F}_2^k$. The first algorithm for computing $T_\alpha x$, denoted PAAR1 in the rest of this work, finds a pair (i, j), with $i \neq j$, where the bitwise `AND` between columns i and j of T_α has the highest Hamming weight. In other words, it finds a pair (x_i, x_j) that occurs most frequently as subexpression in the output bits of $T_\alpha x$. The `XOR` between those is then computed, and M is updated accordingly, with $x_i + x_j$ as newly available variable. This is repeated until there are no common subexpressions left.

The second algorithm, denoted PAAR2, is similar, but differs when multiple pairs are equally common. Instead of just taking the first pair, it recursively tries all of them. The algorithm is therefore much slower, but can yield slightly improved results. Compared to the naive `XOR` count, Paar noted an average reduction in the number of `XOR`s by 17.5% for matrices over $\mathbb{F}_{2^4}$ and by 40% for matrices over $\mathbb{F}_{2^8}$.

In 2009, Bernstein [Ber09] presented an algorithm for efficiently implementing linear maps modulo 2. Based on this and on [Paa97], a new algorithm was presented in [BC14]. However, the algorithms from [BC14; Ber09] have a different framework in mind and yield a higher number of `XOR`s compared to [Paa97].

Paar's algorithms lead to so-called *cancellation-free* programs. This means that for every `XOR` operation $u + v$, none of the input bit variables x_i occurs in both u and v. Thus, the possibility that two variables cancel each other out is never taken into consideration, while this may in fact yield a more efficient solution in terms of the total number of `XOR`s. In 2008, Boyar, Matthews, and Peralta [BMP08] showed that cancellation-free techniques can often not be expected to yield optimal solutions for nontrivial inputs. They also showed that, even under the restriction to cancellation-free programs, the problem of finding an optimal program is NP-complete.

Around 2010, Boyar and Peralta [BP10] came up with a heuristic that is not cancellation-free and that improved on Paar's algorithms in most scenarios. Their idea was to keep track of a distance vector that contains, for each targeted expression of an output bit, the minimum number of additions of the already computed intermediate values that are necessary to obtain that target. To decide which values will be added, the pair that minimizes the sum of new distances is picked. If there is a tie, the pair that maximizes the Euclidean norm of the new distances is chosen. Additionally, if the addition of two values immediately leads to a targeted output, this can always be done without searching further. This algorithm works very well in practice, although it is slower compared to PAAR1.

Next to using the Euclidean norm as tie breaker, they also experimented with alternative criteria. For example, choosing the pair that maximizes the Euclidean norm minus the largest distance, or choosing the pair that maximizes the Euclidean norm minus the difference between the two largest distances. The results were then actually very similar. Another tie-breaking method is to flip a coin and choose a pair randomly. The algorithm is now no longer deterministic and can be run multiple times. The lowest result can then be used. This performed slightly better, but of course processing again takes longer.

The results of [BMP08] and [BP10] were later improved and published in [BMP13]. In early 2017, Visconti, Schiavo, and Peralta [VSP17] explored the special case where the binary matrix is dense. They improved the heuristic on average for dense matrices by first computing a *common path*, an intermediate value that contains most variables. The original algorithm is then run starting from this common path. At BFA 2017, Boyar, Find, and Peralta [BFP19] presented an improvement that simultaneously reduces the number of `XOR`s and the depth of the resulting circuit. We refer to this family of heuristics [BFP19; BMP08; BMP13; BP10; VSP17] as the BP heuristics.

4.4 Results

Using the techniques described above, we now give optimized `XOR` counts and implementations of matrices described in the literature. After that, we analyze the statistical behavior of matrix constructions. Finally we summarize the to date best known matrices.

4.4.1 Improved Implementations of Matrices

Using the heuristic methods that are described in the previous section, we can easily and significantly reduce the `XOR` counts for many matrices that have been used in the literature. The running times for the optimizations are in the range of seconds to minutes. All corresponding implementations are available in the GitHub repository. Table 4.2 and Table 4.3 list results for matrices that have been suggested in previous works where it was an explicit goal to find a lightweight MDS matrix. While the constructions themselves will be compared in Section 4.4.2, this table deals with the suggested instances.

Table 4.2: Comparison of 4×4 MDS matrices over $\mathrm{GL}(4, \mathbb{F}_2)$ and $\mathrm{GL}(8, \mathbb{F}_2)$.

Matrix	Naive	Literature	Paar1	Paar2	BP
4×4 matrices over $\mathrm{GL}(4, \mathbb{F}_2)$					
[SKOP15] (Hadamard)	68	20 + 48	50	48	48
[LS16] (Circulant)	60	12 + 48	49	46	44
[LW16] (Circulant)[*]	60	12 + 48	48	47	44
[BKL16] (Circulant)[†]	64	12 + 48	48	47	**42**
[SS16b] (Toeplitz)	58	**10** + **48**	46	45	43
[JPST17]	61	**10** + **48**	48	47	43
[SKOP15] (Hadam., Invol.)	72	24 + 48	52	48	48
[LW16] (Hadam., Invol.)	72	24 + 48	51	48	48
[SS16b] (Involutory)	64	16 + 48	50	48	**42**
[JPST17] (Involutory)	68	**15** + **48**	51	47	47
4×4 matrices over $\mathrm{GL}(8, \mathbb{F}_2)$					
[SKOP15] (Subfield)	136	40 + 96	100	98	100
[LS16] (Circulant)	128	28 + 96[‡]	116	116	112
[LW16]	106	**10** + **96**	102	102	102
[BKL16] (Circulant)	136	24 + 96	116	112	110
[SS16b] (Toeplitz)	123	24 + 96[‡]	110	108	107
[JPST17] (Subfield)	122	20 + 96	96	95	**86**
[SKOP15] (Subf., Invol.)	144	40 + 96[‡]	104	101	100
[LW16] (Hadam., Invol.)	136	40 + 96	101	97	**91**
[LW16] (Circ., Invol.)	132	36 + 96	104	104	97
[SS16b] (Involutory)	160	64 + 96	110	109	100
[JPST17] (Subf., Invol.)	136	**30** + **96**	102	100	**91**

[*] We chose the matrix presented as an example in the paper.

[†] We chose the canonical candidate from its class of MDS matrices.

[‡] Reported by [JPST17].

A number of issues arise that are worth highlighting. First of all, it should be noted that without any exception, the `XOR` count for every matrix could be reduced with little effort. Second, it turns out that there are many cases where the $n(n-1)k$ `XOR`s for summing the products for all rows is not even a correct lower bound. In fact, all the 4×4 matrices over $\mathrm{GL}(4, \mathbb{F}_2)$ that we studied can be implemented in *at most* 48 `XOR`s.

What may be more interesting, is whether the `XOR` count as it was used previously is a good predictor for the actual implementation cost as given by the heuristic methods. Here we see some differences. For example, [LW16]'s circulant 4×4 matrices over $\mathrm{GL}(8, \mathbb{F}_2)$ first compared very favorably, but we now find that the subfield matrix of [JPST17] requires fewer `XOR`s.

Table 4.3: Comparison of 8×8 MDS matrices over $\mathrm{GL}(4, \mathbb{F}_2)$ and $\mathrm{GL}(8, \mathbb{F}_2)$.

Matrix	Naive	Literature	PAAR1	PAAR2	BP
8×8 matrices over $\mathrm{GL}(4, \mathbb{F}_2)$					
[SKOP15] (Hadamard)	432	**160 + 224***	210	209	**194**
[SS17] (Toeplitz)	394	170 + 224	205	205	201
[SKOP15] (Hadam., Invol.)	512	**200 + 224***	222	222	**217**
8×8 matrices over $\mathrm{GL}(8, \mathbb{F}_2)$					
[SKOP15] (Hadamard)	768	256 + 448*	474	—	467
[LS16] (Circulant)	688	**192 + 448***	464	—	447
[BKL16] (Circulant)	784	208 + 448*	506	—	498
[SS17] (Toeplitz)	680	232 + 448	447	—	**438**
[SKOP15] (Hadam., Invol.)	816	320 + 448*	430	—	**428**
[JPST17] (Hadam., Invol.)	1152	**288 + 448**	620	—	599

* Reported by [JPST17].

Regarding involutory matrices, it was typically the case that there was an extra cost involved to meet this additional criterion. However, the heuristics sometimes find implementations with even fewer `XOR`s than the non-involutory matrix that was suggested. See for example the matrices of [SS16b] in Table 4.2.

Table 4.4: MDS matrices used in ciphers or hash functions.

Cipher	Type	Naive	Literature	PAAR1	PAAR2	BP
AES [DR02][*]	$(\mathbb{F}_2[x]/\texttt{0x11b})^{4\times4}$	152	7 + 96[†]	108	108	97
ANUBIS [BR00a]	$(\mathbb{F}_2[x]/\texttt{0x11d})^{4\times4}$	184	80 + 96[‡]	121	121	106
CLEFIA M_0 [SSA+07]	$(\mathbb{F}_2[x]/\texttt{0x11d})^{4\times4}$	184	80 + 96[‡]	121	121	106
CLEFIA M_1 [SSA+07]	$(\mathbb{F}_2[x]/\texttt{0x11d})^{4\times4}$	208	—	121	121	111
FOX mu4 [JV04]	$(\mathbb{F}_2[x]/\texttt{0x11b})^{4\times4}$	219	—	144	143	137
Twofish [SKW+98]	$(\mathbb{F}_2[x]/\texttt{0x169})^{4\times4}$	327	—	151	149	129
FOX mu8 [JV04]	$(\mathbb{F}_2[x]/\texttt{0x11b})^{8\times8}$	1257	—	611	—	594
Grøstl [GKM+]	$(\mathbb{F}_2[x]/\texttt{0x11b})^{8\times8}$	1112	504 + 448[‡]	493	—	475
Khazad [BR00b]	$(\mathbb{F}_2[x]/\texttt{0x11d})^{8\times8}$	1232	584 + 448[‡]	488	—	507
Whirlpool [BR00c][§]	$(\mathbb{F}_2[x]/\texttt{0x11d})^{8\times8}$	840	304 + 448[‡]	481	—	465
Joltik [JNP15]	$(\mathbb{F}_2[x]/\texttt{0x13})^{4\times4}$	72	20 + 48[‡]	52	48	48
Small scale AES [CMR05]	$(\mathbb{F}_2[x]/\texttt{0x13})^{4\times4}$	72	—	54	54	47
Whirlwind M_0 [BNN+10]	$(\mathbb{F}_2[x]/\texttt{0x13})^{8\times8}$	488	168 + 224[‡]	218	218	212
Whirlwind M_1 [BNN+10]	$(\mathbb{F}_2[x]/\texttt{0x13})^{8\times8}$	536	184 + 224[‡]	246	244	235

[*] Also used in other primitives, e. g., its predecessor Square [DKR97] and MUGI [WFY+02].

[†] Reported by [JMPS17].

[‡] Reported by [JPST17].

[§] Also used in Maelstrom-0 [FBR06].

Aside from these matrices, we also looked at (MDS) matrices that are used by various ciphers and hash functions. Table 4.4 and Table 4.5 list their results. Not all MDS matrices that are used in ciphers are incorporated here. In particular, LED [GPPR11], PHOTON [GPP11], and PRIMATEs [ABB+14] use efficient serialized MDS matrices. Comparing these to our "unrolled" implementations would be somewhat unfair.

Table 4.5: Non-MDS matrices used in ciphers or hash functions.

Cipher	Type	Naive	Literature	Paar1	Paar2	BP
QARMA-128 [Ava17]	$(\mathbb{F}_2[x]/\texttt{0x101})^{4\times4}$	64	—	48	48	48
ARIA [KKP+04]	$(\mathbb{F}_2)^{128\times128}$	768	480*	416	—	—
Midori [BBI+15]†	$(\mathbb{F}_{2^4})^{4\times4}$	32	—	24	24	24
PRINCE $\widehat{M}_0$, $\widehat{M}_1$ [BCG+12]	$(\mathbb{F}_2)^{16\times16}$	32	—	24	24	24
PRIDE L_0–L_3 [ADK+14]	$(\mathbb{F}_2)^{16\times16}$	32	—	24	24	24
QARMA-64 [Ava17]	$(\mathbb{F}_2[x]/\texttt{0x11})^{4\times4}$	32	—	24	24	24
SKINNY-64 [BJK+16]	$(\mathbb{F}_{2^4})^{4\times4}$	16	12	12	12	12

* Reported by [BCL+04].

† Also used in other ciphers, e. g., MANTIS [BJK+16] and Fides [BBK+13].

The implementation of the MDS matrix used in AES with 97 `XOR`s is, to the best of our knowledge, the most efficient implementation so far and improves on the previous implementation of 103 `XOR`s, reported by [JMPS17]. As a side note, cancellations do occur in this implementation, we thus conjecture that such a low `XOR` count is not possible with cancellation-free programs.

4.4.2 Statistical Analysis

Several constructions for building MDS matrices are known. However, it is not clear which one is the best when we want to construct matrices with a low `XOR` count. In this section, we present experimental results on different constructions and draw conclusions for the designer. We also examine the correlation between naive and heuristically improved `XOR` counts. When designing MDS matrices with a low `XOR` count, we are faced with two major questions. First, which construction is preferable? Our intuition in this case is that a better construction has better statistical properties compared to an inferior construction. We are aware that the statistical behavior of a construction might not be very important for a designer who only looks for a single, very good instance. Nevertheless we use this as a first benchmark. Second, is it a good approach to choose the matrices as sparse as possible? To compare the listed constructions, we construct random instances of each and then analyze them with statistical means.

Building Cauchy and Vandermonde matrices is straightforward as we only need to choose the defining elements randomly from the underlying field. For the other constructions, we use the following backtracking method to build random MDS constructions of dimension 4×4. Choose the new random elements from $\mathrm{GL}(k, \mathbb{F}_2)$ that are needed for the matrix construction in a step-by-step manner. In each step, construct all new square submatrices. If any of these is not invertible, discard the chosen element and try a new one. In the case that no more elements are left, go one step back and replace that element with a new one, then again check the according square submatrices, and so on. Eventually, we end up with an MDS matrix because we iteratively checked that every square submatrix is invertible. The method is also trivially derandomizable, by not choosing the elements randomly, but simply enumerating them in any determined order.

Apart from applying this method to the above mentioned constructions, we can also use it to construct an *arbitrary*, i. e., unstructured, matrix that is simply defined by its 16 elements. This approach was already described in [JPST17].

In this manner, we generated 1000 matrices for each construction and computed the distributions for the naive `XOR` count, the optimized `XOR` count of Paar1, and BP. Table 4.6 lists the statistical parameters of the resulting distributions and Fig. 4.1 depicts them (the sample size N is the same for Table 4.6 and Figs. 4.1 through 4.5).

Table 4.6: Distributions for differently optimized `XOR` counts. By N we denote the sample size, μ is the mean, and σ^2 the variance.

Construction	N	Naive		Paar1		BP	
		μ	σ^2	μ	σ^2	μ	σ^2
4×4 matrices over $\mathrm{GL}(4, \mathbb{F}_2)$							
Cauchy	1 000	120.7	77.3	62.9	11.0	53.1	4.0
Circulant	1 000	111.8	117.1	60.4	19.2	52.1	7.1
Hadamard	1 000	117.5	99.6	60.2	17.8	52.4	6.9
Toeplitz	1 000	112.8	39.9	59.9	7.4	51.3	3.9
Vandermonde	1 000	120.6	87.6	62.2	8.1	52.9	3.1
Enumerated 4×4 matrices over $\mathrm{GL}(4, \mathbb{F}_2)$							
Circulant	1 000	82.9	53.0	54.9	13.5	50.1	6.7
Hadamard	1 000	102.1	76.0	56.7	20.6	50.6	8.0
Toeplitz	1 000	86.1	43.9	55.3	8.3	49.4	3.9
Arbitrary	1 000	80.5	8.3	49.7	3.2	44.5	1.8
4×4 matrices over $\mathrm{GL}(8, \mathbb{F}_2)$							
Cauchy	1 000	454.1	467.2	215.1	39.6	—	—
Vandermonde	1 000	487.3	597.4	220.2	44.3	—	—
4×4 subfield matrices over $\mathrm{GL}(4, \mathbb{F}_2)$							
Cauchy	1 000	241.1	312.1	125.8	44.2	—	—
Vandermonde	1 000	240.6	452.8	121.8	47.1	—	—

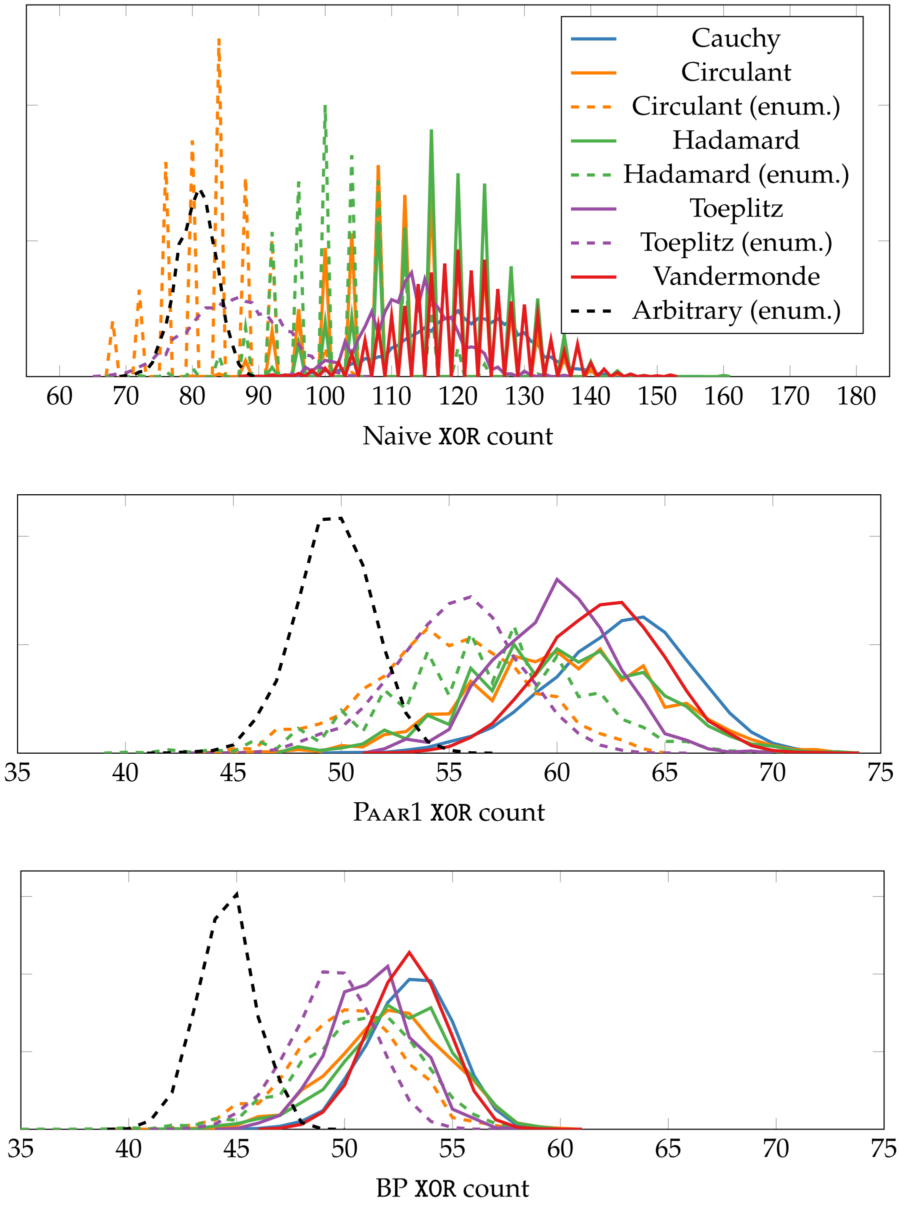


Figure 4.1: XOR count distributions for 4×4 MDS matrices over $\mathrm{GL}(4, \mathbb{F}_2)$.

The most obvious characteristics of the statistical distributions are that the means μ do not differ much for all randomized constructions. The variances σ^2 on the contrary differ much more. This is most noticeable for the naive `XOR` count, while the differences get much smaller when the `XOR` count is optimized with the Paar1 or BP heuristic. One might think that the construction with the lowest optimized average `XOR` count, which is for matrices over $\mathrm{GL}(4, \mathbb{F}_2)$ the arbitrary construction with enumerated elements, yields the best results. However, the best matrix we could find for these dimension was a Hadamard matrix. An explanation for this might be the higher variance that leads to some particularly bad and some particularly good results.

The graphs in Fig. 4.1 convey a similar hypothesis. Looking only at the naive `XOR` count, we can notice some differences. For example circulant matrices seem to give better results than, e. g., Hadamard matrices. Additionally, the naive `XOR` count increases step-wise as not every possible count occurs. However, the distributions get smoother and more similar when optimizing the `XOR` count.

We conclude that all constructions give similarly good matrices when we are searching for the matrix with the lowest `XOR` count, with one important exception. For randomly generated matrices the `XOR` count increases by a factor of four, if we double the parameter k. Table 4.6 covers this for Cauchy and Vandermonde matrices. We do not compute the statistical properties for Circulant, Hadamard, and Toeplitz matrices with elements of $\mathrm{GL}(8, \mathbb{F}_2)$, as the probability to find a random MDS instance for these constructions is quite low. Thus, generating enough instances for a meaningful statistical comparison is computationally tough and – as we deduce from a much smaller sample size – the statistical behavior looks very similar to that of the Cauchy and Vandermonde matrices. Instead, as was already mentioned in Lemma 1, the subfield construction has a much more interesting behavior. It simply doubles the `XOR` count. The lower half of Table 4.6 confirms this behavior.

Thus, when it is computationally infeasible to exhaustively search through all possible matrices, it seems to be a very good strategy to use the subfield construction with the best known results from smaller dimensions. This conclusion is confirmed by the fact that our best results for matrices over $\mathrm{GL}(8, \mathbb{F}_2)$ are always subfield constructions based on matrices over $\mathrm{GL}(4, \mathbb{F}_2)$.

Next, we want to approach the question if choosing MDS matrices with entries with low Hamming weight is a good approach to finding low-`XOR`-count implementations. For each MDS-matrix family and for each optimization algorithm, we plot the naive `XOR` count against the optimized one.

In Figs. 4.2 through 4.5 one can see that several options can occur. While there is a general tendency of higher naive `XOR` counts leading to higher optimized `XOR` counts, the contrary is also possible. For example, there are matrices which have a low naive `XOR` count (left in the figure), while still having a somewhat high optimized `XOR` count (top part of the figure). However, there are also matrices where a higher naive `XOR` count results in a much better optimized `XOR` count. The consequence is that we cannot restrict ourselves to very sparse matrices when searching for the best `XOR` count, but also have to take more dense matrices into account. A possible explanation for this behavior is that the heuristics have more possibilities for optimizations, when the matrix is not sparse.

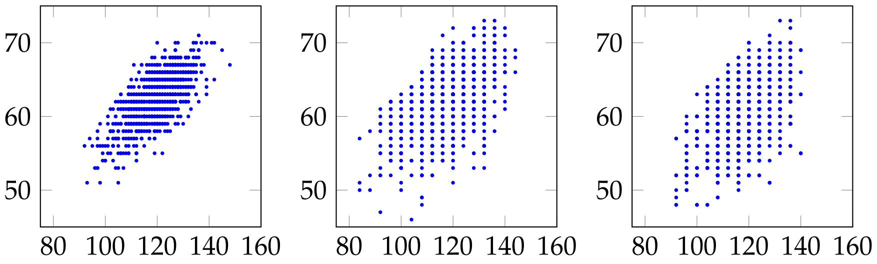
4.4.3 Best results

Let us conclude by specifying the currently best MDS matrices. The notation $M_{n,k}$ denotes an $n \times n$ matrix with entries from $\mathrm{GL}(k, \mathbb{F}_2)$, an involutory matrix is labeled with the superscript i. Table 4.1 covers non-involutory and involutory matrices of dimension 4×4 and 8×8 over $\mathrm{GL}(4, \mathbb{F}_2)$ and $\mathrm{GL}(8, \mathbb{F}_2)$. $M_{8,4}$ and $M^i_{8,4}$ are defined over $\mathbb{F}_2[x]/$`0x13`.

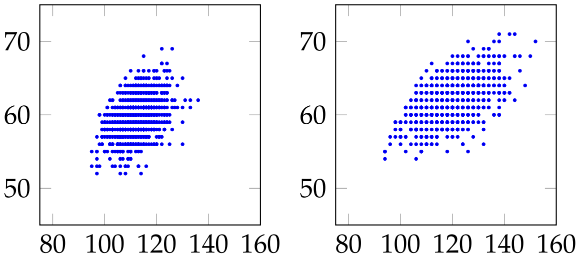
The matrices mentioned there are the following:

$$M_{4,4} = \mathrm{hadamard}(\begin{pmatrix} 0&0&0&1 \\ 0&0&1&0 \\ 0&1&0&0 \\ 1&0&0&0 \end{pmatrix}, \begin{pmatrix} 0&0&1&1 \\ 1&0&0&1 \\ 1&1&0&0 \\ 0&1&0&0 \end{pmatrix}, \begin{pmatrix} 1&1&0&1 \\ 1&1&0&0 \\ 0&1&0&1 \\ 0&0&1&0 \end{pmatrix}, \begin{pmatrix} 1&1&0&0 \\ 0&1&0&1 \\ 1&0&1&1 \\ 0&0&0&1 \end{pmatrix}) \tag{4.1}$$

$$M_{4,8} = \mathrm{subfield}(M_{4,4}) \tag{4.2}$$



(a) Cauchy **(b)** Circulant **(c)** Hadamard



(d) Toeplitz **(e)** Vandermonde

Figure 4.2: Correlations between naive (x-axis) and Paar1 (y-axis) `XOR` counts for randomly generated matrices.

$$M_{8,4} = \text{cauchy}\begin{pmatrix} x^3{+}x^2, x^3, x^3{+}x{+}1, x{+}1, 0, x^3{+}x^2{+}x{+}1, x^2, x^2{+}x{+}1, \\ 1, x^2{+}1, x^3{+}x^2{+}x, x^3{+}1, x^3{+}x^2{+}1, x^2{+}x, x^3{+}x, x \end{pmatrix} \tag{4.3}$$

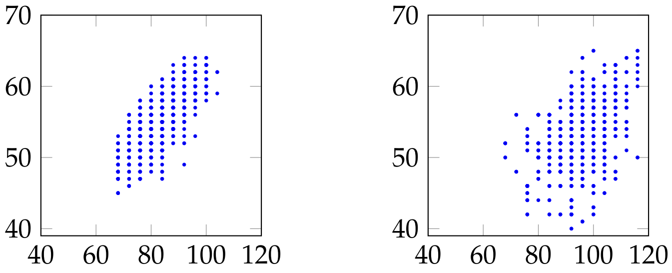
$$M_{8,8} = \text{subfield}(M_{8,4}) \tag{4.4}$$

$$M^i_{4,8} \text{ is the subfield construction applied to [SS16b, Example 3]} \tag{4.5}$$

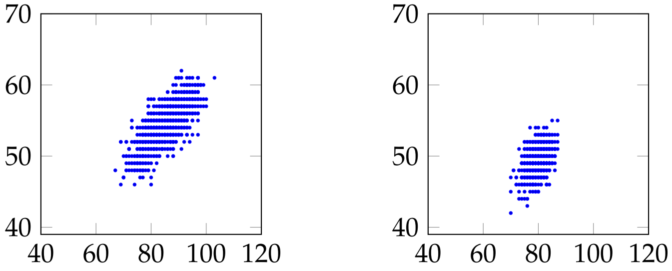
$$M^i_{8,4} = \text{vandermonde}\begin{pmatrix} x^3{+}x{+}1, x{+}1, x^3{+}x^2{+}x, x^3{+}x^2{+}1, x^3{+}1, x^3, 0, x^3{+}x \\ x^2{+}x{+}1, x^3{+}x^2{+}x{+}1, x, 1, x^2{+}1, x^2, x^3{+}x^2, x^2{+}x \end{pmatrix} \tag{4.6}$$

$$M^i_{8,8} = \text{subfield}(M^i_{8,4}) \tag{4.7}$$

All these matrices improve over the previously known matrices, except for the involutory matrix from [SS16b] of dimension 4 × 4 over $\mathrm{GL}(4, \mathbb{F}_2)$. $M_{4,4}$ was



(a) Circulant (enum.) **(b)** Hadamard (enum.)

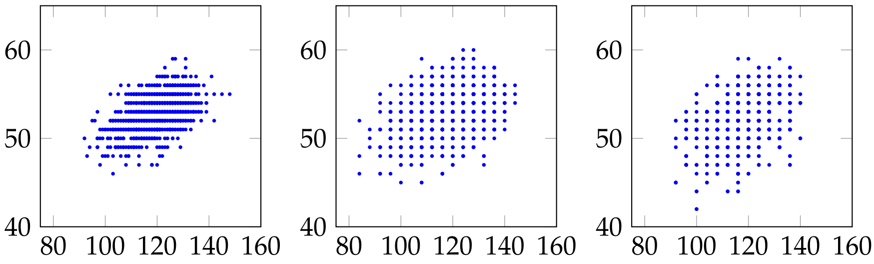


(c) Toeplitz (enum.) **(d)** Arbitrary (enum.)

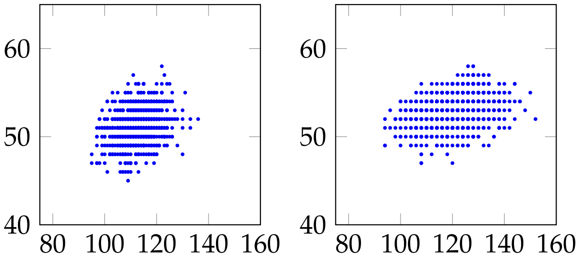
Figure 4.3: Correlations between naive (x-axis) and Paar1 (y-axis) `XOR` counts for enumerated matrices.

found after enumerating a few thousand Hadamard matrices, while $M_{8,4}$ and $M^i_{8,4}$ are randomly generated and were found after a few seconds. Every best matrix over $\mathrm{GL}(8, \mathbb{F}_2)$ uses the subfield construction.

With these results we want to highlight that, when applying global optimizations, it is quite easy to improve (almost) all currently best known results. We would like to mention that our results should not be misunderstood as an attempt to construct matrices that cannot be improved. Another point that was not covered in this work is the depth of the critical path as considered in [BFP19].

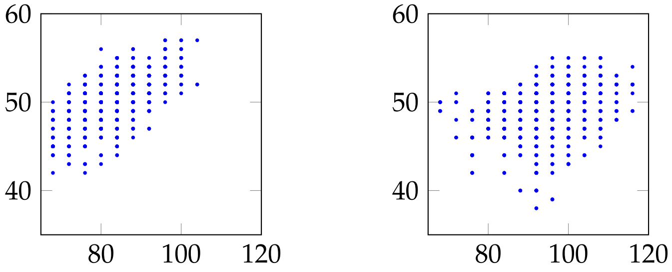


(a) Cauchy **(b)** Circulant **(c)** Hadamard

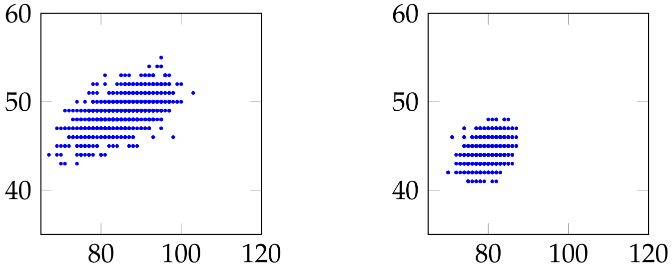


(d) Toeplitz **(e)** Vandermonde

Figure 4.4: Correlations between naive (x-axis) and BP (y-axis) `XOR` counts for randomly generated matrices.



(a) Circulant (enum.) **(b)** Hadamard (enum.)



(c) Toeplitz (enum.) **(d)** Arbitrary (enum.)

Figure 4.5: Correlations between naive (x-axis) and BP (y-axis) `XOR` counts for enumerated matrices.

Chapter 5

Column-Parity Mixers

As a third and final cryptographic building block, we formalize column-parity mixers and study their cryptographic properties. They are suitable as an alternative to MDS matrices for linear mixing layers in round functions of iterated permutations and ciphers. We then outline a full design strategy that incorporates column-parity mixers. In comparison to the original publication [SD18], some appendices have been removed and some have been merged into the regular text.

5.1 Introduction

In recent years, there has been a lot of research on lightweight cryptography. A need is perceived for cryptographic primitives that can be implemented on resource-constrained platforms, such as devices in the Internet of Things. Historically the most important primitives were block ciphers, but since a few years cryptographic permutations gain in popularity. Most of these ciphers consist of the repeated application of an invertible round function, where in the case of block ciphers each round takes a round key. As inspired by DES [DES77] and later AES [DR02], often the round function consists of a number of separate layers, each with a particular task. There is typically a nonlinear S-box layer and a linear diffusion layer. In some modern ciphers, this linear layer consists of a sub-layer that mixes the bits (or bytes) and typically has a high branch number [DR02], and a transposition sub-layer that moves bits (or bytes around). In AES, the mixing layer MixColumns consists of the parallel application of a maximum-distance separable (MDS) mapping on each of the 4-byte columns. This mapping has branch number 5, the maximum attainable value. In combination with the transposition layer ShiftRows, this allows to give a simple proof for a strong upper bound on the differential probability of 4-round differential trails (characteristics) and the correlation of 4-round linear trails.

With lightweight cryptography in mind, there has been a substantial amount of publications in the last few years on constructions for lightweight MDS mappings, see e. g., [BKL16; GPP11; LS16; LW16]. This has led to a better understanding of the implementation cost of such mappings in relation to their dimensions: the number and size of elements (e. g., 4 bytes in MixColumns). We can now build block ciphers and permutations with the same design philosophy as AES, using more lightweight components and leading to bounds that are easy to prove.

Another cipher where the designers emphasize the mixing layer is Keccak-f, the permutation underlying Keccak and the SHA-3 functions [BDPV11b; NIS15b]. Its mixing layer θ does not operate on separate subblocks as MixColumns does, and it has a branch number of 4. However, despite the fact that it requires only 2 `XOR` operations per bit, in combination with the other layers of the round function it appears to have quite good diffusion. In particular, [MDA17] reports on computer-aided proofs for quite promising upper bounds for the differential probability of differential trails. It appears that θ-like mappings would be a good candidate for a linear mixing layer, or in general for a mixing layer with a good trade-off between implementation cost and mixing power.

There is a remarkable difference between AES and Keccak-f. In the former, all step mappings are defined in terms of operations on bytes and in the latter each step mapping treats groupings of bits along different axes. The AES design approach has received quite some following and it can be called byte-oriented (although there are also ciphers where these units are 4-bit nibbles or even 5-bit units). We call the design approach of Keccak-f bit-oriented. The byte-oriented design approach has the advantage that one can easily prove bounds. We believe that θ-like mappings are suitable for both design approaches.

5.1.1 Our Contributions

In this chapter we present a generalization of the θ mixing layer in Keccak-f called column-parity mixers (CPMs). CPMs operate on two-dimensional arrays and in their definition parities computed over the columns play a central role, hence the name *column-parity mixers*. In Section 5.2 we provide an elegant description using matrix arithmetic, allowing us to easily derive algebraic, diffusion, and mask-propagation properties. We also show that column-parity mixers operating on states with an even number of rows have quite different properties from those operating on an odd number of rows.

The former are involutions and are ideally suited for block ciphers and permutations that need to have an efficient inverse. Coincidentally, those are the ones that are typically required to have a permutation width (or block size) that is a power of two, and this is nicely compatible with an even number of rows. An example is disk encryption where the size of a disk sector is a power of two.

The latter may have an inverse with high implementation cost but also with very interesting diffusion properties (see Section 5.4). They are suited for permutations used in MACs or in stream encryption, or in conjunction with authenticated encryption modes that do not require the inverse such as the sponge and duplex constructions [BDPV11a], where the permutation width is also less bound to a power of two.

We see Keccak-f as a representative example of a bit-oriented design that makes use of a column-parity mixing layer and we make the case that they are also suitable for byte-oriented designs. In Section 5.5 we outline a general design strategy with attention for how strong bounds can be attained for differential and linear trails. We apply this strategy concretely to design a 256-bit permutation called Mixifer with an efficient inverse. This can be used as a permutation in an Even-Mansour block cipher [EM93] or in modes such as proposed in [Men16]. The width of 256 is large enough to make the birthday bound 2^{128} far enough to not pose a practical problem and it is small enough to still be called lightweight.

Our permutation design, presented in Section 5.6, uses a number of ideas that are of independent interest. It operates on an array of 4 rows of 16 nibbles each and we arrange the bits of the nibbles in such a way that a bitsliced implementation is immediate. Its design can be seen as a hybrid between that of AES and that of Keccak-f. We show in Section 5.6.6 that in comparison with AES and some established permutations, the performance of our permutation is better or comparable.

5.2 Column-Parity Mixers and their Properties

Column-parity mixers are generalizations of the mixing layer θ in Keccak-f. Therefore, we adopt the terminology proposed in [BDPV11b]. Unlike the descriptions therein, we will treat the states that these mixing layers operate on as two-dimensional arrays that we will interpret as matrices. In the context of this section we limit ourselves to matrices with elements of $\mathbb{F}_2$, but one can easily generalize our treatment to elements of $\mathbb{F}_p$ with p a prime or even to elements of an arbitrary finite ring.

5.2.1 Matrices

We use $\mathbf{I}$ to denote a (square) identity matrix and $\mathbf{0}$ to denote an all-zero matrix. We assume that the dimensions of these matrices are determined by context. The transpose of a matrix A is denoted as A^T.

We use $\mathbf{1}_x$ to denote a column vector of x components that are all equal to 1. Consequently, $\mathbf{1}_x^\mathsf{T}$ is an all-1 row vector with x components. We use $\mathbf{1}_x^y$ to denote a matrix with x rows and y columns with all components 1. Clearly, $\mathbf{1}_x^y = \mathbf{1}_x \mathbf{1}_y^\mathsf{T}$.

The element of a matrix A at row i and column j is denoted by $A_{i,j}$. If $B = A^\mathsf{T}$, we have $B_{i,j} = A_{j,i}$. The trace of a square matrix is the linear function that simply takes the sum of its diagonal elements. It is denoted by $\mathrm{tr}(A)$, so $\mathrm{tr}(A) = \sum_i A_{i,i}$.

5.2.2 Definition of Column-Parity Mixers

We consider linear mappings θ that operate on arrays with m rows and n columns of elements of a finite ring, but in this section just $\mathbb{F}_2$.

Definition 9. The *column parity* of a matrix A is a (row) vector defined as $\mathbf{1}_m^\mathsf{T} A$.

In a matrix A, a column x is called even (odd) if the component with index x in $\mathbf{1}_m^\mathsf{T} A$ is zero (one).

Definition 10. The *expanded column parity* of A is a matrix with m rows all equal to the column parity of A, and it is given by $\mathbf{1}_m^m A$.

A column-parity mixer (CPM) makes use of a linear transformation operating on the column parity of a matrix, called its *parity-folding transformation*. We denote the parity-folding transformation by multiplying the column parity with a square matrix Z at the right. We call the $n \times n$ matrix Z the *parity-folding matrix* of θ. We are now ready to define the θ-effect of a matrix A.

Definition 11. The *θ-effect* of A with respect to Z is a row vector, denoted as $\mathbf{e}_\mathbf{Z}(A)$ (or just $\mathbf{e}(A)$ if Z is clear from the context) and is defined by $\mathbf{e}_\mathbf{Z}(A) = \mathbf{1}_m^\mathsf{T} A Z$.

For a given input A and parity-folding matrix Z, a column x is called *unaffected* (affected) if the component with index x in $\mathbf{e}_\mathbf{Z}(A)$ is zero (nonzero). Whether a column is affected or not is fully determined by the column parity of A and the column x of the parity-folding matrix Z.

Definition 12. The *expanded θ-effect* of A with respect to Z is a matrix with m rows all equal to the CPM effect, namely, $\mathbf{E}_\mathbf{Z}(A) = \mathbf{1}_m^m A Z$.

A column-parity mixer θ simply consists in computing the expanded θ-effect of a matrix A and adding it to A.

Definition 13. The *column-parity mixer θ* using parity-folding matrix Z is defined as

$$\theta(A) = A + \mathbf{E}_\mathbf{Z}(A) = A + \mathbf{1}_m^m A Z\,.$$

Note that a column-parity mixer is fully defined by a parity-folding matrix Z and m.

Example 1. Keccak [BDPV11b] uses a three-dimensional structure, so, for the sake of this example, let us first *flatten* the state by looking at a single sheet. With Keccak-f[200], this array would have $m = 5$ rows and $n = 8$ columns. Consider the following state A:

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}.$$

Then the *column parity* of A is $[1, 1, 0, 0, 0, 1, 0, 0]$, so the first two columns and the sixth column are *odd*, while the others are *even*. The θ step in Keccak-f affects the adjacent sheets, but one can modify it slightly such that the operation is performed within the same sheet. To the reader who is familiar with the Keccak specification, we change $x - 1 \bmod 5$ and $x + 1 \bmod 5$ to $x \bmod 5$ in the computation of $D[x, z]$ given $C[x, z]$. This means that we can express the *parity-folding matrix* Z as follows:

$$Z = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \end{bmatrix}.$$

This yields a *θ-effect* of $\mathbf{e}(A) = [0, 1, 0, 0, 1, 1, 0, 1]$, so the second, fifth, sixth, and eighth column are *affected*, the others are *unaffected*. The result of the *column-parity* *mixer* defined by Z and m on A is then

$$\theta(A) = \begin{bmatrix} 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} .$$

5.2.3 Group Properties

The composition of two column-parity mixers is again a column-parity mixer.

Lemma 2. *Let $\alpha = \theta' \circ \theta$ with θ and θ' column-parity mixers with parity-folding matrices Z and Z' respectively. Then α is a column-parity mixer. If m is even, the parity-folding matrix of α is $Z + Z'$. If m is odd, its parity-folding matrix is $(Z' + \mathbf{I})(Z + \mathbf{I}) + \mathbf{I}$.*

Proof. We have

$$\begin{aligned} \theta'(\theta(A)) &= A + \mathbf{1}_m^m AZ + \mathbf{1}_m^m \left(A + \mathbf{1}_m^m AZ\right) Z' \\ &= A + \mathbf{1}_m^m AZ + \mathbf{1}_m^m AZ' + (\mathbf{1}_m^m)^2 AZZ' . \end{aligned}$$

If m is even, we have $(\mathbf{1}_m^m)^2 = \mathbf{0}$ and this reduces to $A + \mathbf{1}_m^m A(Z + Z')$. If m is odd, we have $(\mathbf{1}_m^m)^2 = \mathbf{1}_m^m$ and we have

$$\begin{aligned} \theta'(\theta(A)) &= A + \mathbf{1}_m^m A(Z + Z' + ZZ') \\ &= A + \mathbf{1}_m^m A((Z + \mathbf{I})(Z' + \mathbf{I}) + \mathbf{I}) . \end{aligned}$$

□

This implies the following corollary:

Corollary 1. *The set of all column-parity mixers for given dimensions $m \times n$, with m even, together with composition form a group that is isomorphic to the abelian group $(\mathbb{Z}_2^{n^2}, +)$.*

Proof. Lemma 2 says that the composition of two CPMs with parity-folding matrices Z and Z' is the CPM with parity-folding matrix $Z + Z'$. It follows that the set of all CPMs of given dimensions $m \times n$ and m even, is isomorphic to the set of binary $n \times n$ matrices with addition. This addition is closed and inherits the associativity and commutativity from the addition in $\mathbb{F}_2$. Its neutral element is $\mathbf{0}$ and each element is its own inverse. This is $(\mathbb{Z}_2^{n^2}, +)$. $\square$

It follows that column-parity mixers operating on matrices with an even number m of rows are involutions.

For m odd, not all column-parity mixers are invertible. We have the following:

Corollary 2. *The set of column-parity mixers for given dimensions $m \times n$, with m odd and $Z + \mathbf{I}$ nonsingular, form a group with composition that is isomorphic to the group of binary invertible $n \times n$ matrices with multiplication.*

Proof. Lemma 2 says that the composition of two CPMs with parity-folding matrices Z and Z' is the CPM with parity-folding matrix $(Z + \mathbf{I})(Z' + \mathbf{I}) + \mathbf{I}$. Let us call $Z + \mathbf{I}$ the associated matrix of a CPM. Then the associated matrix of the composition of two CPMs is the product of their associated matrices. It follows that the set of all CPMs of given dimensions $m \times n$ and m odd, is isomorphic to the set of invertible binary $n \times n$ matrices with multiplication. This is the general linear group $GL(n, 2)$ [DF03]. $\square$

For m odd, the order of a CPM is the multiplicative order of its associated matrix $Z + \mathbf{I}$. The associated matrix of the inverse of an invertible CPM with parity-folding matrix Z is $(Z + \mathbf{I})^{-1}$.

5.2.4 The Special Case of Circulant Parity-Folding Matrices

Z is a circulant matrix if its elements satisfy $Z_{i+j \bmod n,\, j} = Z_{i,0}$ for all i, j. Circulant matrices have become a popular building block in many ciphers, including AES [DR02] and Keccak [BDPV11b]. As the product and sum of two circulant matrices is a circulant matrix and both $\mathbf{I}$ and $\mathbf{0}$ are circulant matrices, it follows that both for m even and odd the set of invertible circulant CPMs form subgroups of the set of invertible CPMs, for given dimensions.

We can express multiplication by a circulant matrix as multiplication by a polynomial. For that purpose, we express a matrix A as a bivariate polynomial, where the element in row i and column j corresponds to the coefficient of monomial $x^i y^j$. Computing the column parity of A corresponds to multiplication by the polynomial $\sum_{i=0}^{n-1} y^i \bmod 1 + y^n$. This polynomial can also be expressed as $\frac{1+y^n}{1+y}$. Multiplication by Z then corresponds to multiplication by a polynomial $z(x) \bmod 1 + x^n$. So for our column-parity mixer θ, we have

$$\theta(a(x, y)) = a(x, y) + \frac{1 + y^m}{1 + y} z(x)a(x, y) \bmod (1 + x^n)(1 + y^m)\,.$$

We call $z(x)$ the parity-folding polynomial.

Circulant CPMs with even m are involutions. A circulant CPM with odd m is invertible if its associated polynomial $1 + z(x)$ is coprime to $1 + x^n$ and its inverse has the associated polynomial $y(x) = (z(x) + 1)^{-1} \bmod (1 + x^n)$. A necessary condition for invertibility is that $z(x)$ has an even number of nonzero terms. If not, $1 + x$ divides both $1 + z(x)$ and $1 + x^n$.

The transpose of θ is determined by the parity-folding polynomial $z(x^{-1}) \bmod 1 + x^n$, i. e., the polynomial $z(x)$ where the terms x^i are replaced by x^{n-i}.

5.2.5 Computational Cost

Computing the column parities costs $m - 1$ additions in $\mathbb{F}_2$ per column totalling to $(m - 1)n$ additions. Adding the effect to the matrix costs one addition per bit totalling to mn binary additions. So the total computational cost is $(2m - 1)n$ plus the computational cost of applying Z to the parity.

For circulant mixers with h the Hamming weight of $z(x)$, a straightforward parity-folding implementation would cost $(h - 1)n$, totalling to $(2m + h - 2)n$. Dividing by the total number of bits in the state, this results in a computational cost per bit of $2 + (h - 2)/m$. Remarkably, for parity-folding polynomials with two nonzero terms, the cost is exactly 2 additions per bit.

The number of additions gives a good idea of the circuit complexity in dedicated hardware and the number of binary `XOR` gates in bitslice-oriented software implementations. In the latter, the efficiency additionally depends on the way the state can be mapped onto CPU words and computing θ may involve additional (cyclic) shift operations. Section 5.6.5 shows the exact cost for a concrete example.

5.3 Propagation of Linear Masks

After having explained what CPMs are, this section discusses their properties related to linear cryptanalysis [Mat94]. We first provide a brief overview of linear propagation for generic iterated permutations, before discussing CPM-specific details.

5.3.1 Linear Propagation in Iterated Permutations

Linear cryptanalysis (LC) exploits large correlations across a cryptographic primitive and resistance against it is one of the main criteria of modern cryptographic design. It can be described in different ways and we adopt the notation and formalism used in [DR02].

In LC we consider a sum (in $\mathbb{F}_2$) of bits (usually called *parities*) at the output of a mapping and try to find sums of bits at the input of the mapping that have a high correlation with the sum at the output. Which bits are included in the sums is described by *masks*. Masks have the same dimensions and shape as the state and indicate the bits included in a sum by having a 1 in the corresponding positions and 0 in all other positions. While the correlations are between sums of input bits and sums of output bits, we will indicate these by the term mask to make the descriptions more readable. Masks play a role in LC similar to that of *differences* in differential cryptanalysis (DC).

In iterated permutations or block ciphers, a correlation between an output mask v and an input mask u can be split into a number of *linear trails*. The value of the correlation is the sum of the *correlation contributions* of these individual trails. Note that correlations and correlation contributions have a sign, so there can be destructive interference. A linear trail Q over an n-round permutation (or block cipher) consists of a sequence of $n + 1$ masks: a mask at the input of each round q_i and a mask q_n at the output of the last round. A pair of consecutive masks q_i, q_{i+1} has a certain correlation over round i, denoted as $C(q_i, q_{i+1})$. The correlation contribution $C(Q)$ of a linear trail Q is simply the product of the correlations over all its rounds: $\prod_{i=0}^{n-1} C(q_i, q_{i+1})$.

The value of an input-output correlation $C(u, v)$ is given by:

$$C(u,v) = \sum_{Q \text{ with } q_0=u, q_n=v} C(Q)\,.$$

Large input-output correlations can hence occur if there are linear trails with large correlation contributions (e. g., in the block cipher DES [Mat94]) but they can in principle also occur when there is systematic clustering of huge numbers of trails with very small correlation contributions. An artificial example is a permutation P that consists of a permutation P' followed by its inverse P'^{-1}. The permutation P is the identity and hence has many large input-output correlations but no high-correlation linear trails if P' has none.

Modern ciphers should be designed taking LC into account and hence should not have linear trails with high correlation contributions. A way to achieve this with a relatively small number of rounds is called the *wide-trail strategy* [DR02], underlying many modern designs including AES and Keccak. In this strategy, the mixing layer in the round function plays an important role. To study the correlation contribution of linear trails, we need to study correlations over the round function. The round function typically consists of a nonlinear layer and a linear layer. The description of the correlation properties of nonlinear S-box layers can be efficiently computed using the Walsh-Hadamard transform. Linear layers have the property that every output mask is correlated to exactly one input mask and that the correlation is one. In other words, there is a deterministic function that maps masks v at the output of a linear layer to masks u at its input. In the following subsection, we will derive an expression for this function $u = f(v)$.

5.3.2 Mask Propagation in Column-Parity Mixers

We want to express this function in our matrix notation. As an intermediate step, we first need a way to express a sum of specific bits of a matrix, for which we use the trace function.

The trace function has a number of interesting properties that we will exploit in our derivation:

- Linearity: $\mathrm{tr}(A + B) = \mathrm{tr}(A) + \mathrm{tr}(B)$.
- Transpose-invariance: $\mathrm{tr}(A^\mathsf{T}) = \mathrm{tr}(A)$.
- Commute-invariance: for A and B with compatible dimensions and AB a square matrix, it is easy to show that $\mathrm{tr}(AB) = \mathrm{tr}(BA)$ [LLM16].
- Multiplication-cyclic invariance: for any sequence of matrices $A, B, C, \ldots, Z$ with compatible dimensions and with $AB \cdots Z$ a square matrix: $\mathrm{tr}(AB \cdots Z) = \mathrm{tr}(BC \cdots ZA) = \mathrm{tr}(CD \cdots ZAB) = \ldots$.

A mask V has the same dimensions as the matrix A and it specifies the sum (in $\mathbb{F}_2$) of the elements of A in positions where V has a one, i. e., $\sum_{i,j} V_{i,j} A_{i,j}$. This cannot be expressed as a matrix multiplication, but we can express it with the trace function.

Lemma 3. *The sum of the elements of A selected by the mask V can be expressed as* $\mathrm{tr}(V^\mathsf{T} A)$*.*

Proof. Let $B = V^\mathsf{T} A$, then $B_{j,j} = \sum_i V_{i,j} A_{i,j}$. As $\mathrm{tr}(B) = \sum_j B_{j,j}$, it follows that $\mathrm{tr}(V^\mathsf{T} A) = \sum_{i,j} V_{i,j} A_{i,j}$. ☐

Lemma 4. *A sum of bits at the input of θ defined by a mask U equals a sum of bits at the output of θ defined by mask V if and only if*

$$U = V + \mathbf{1}_m^m V Z^\mathsf{T} .$$

Proof. Let B be the image of A through the column-parity matrix. Then given $\operatorname{tr}(U^\mathsf{T} A) = \operatorname{tr}(V^\mathsf{T} B)$, we want to derive the relation between U and V. In particular, given the mask V at the output of θ, we can compute the mask U at its input by filling in the expression for B:

$$\begin{aligned}
\operatorname{tr}(U^\mathsf{T} A) &= \operatorname{tr}(V^\mathsf{T}(A + \mathbf{1}_m^m A Z)) \\
&= \operatorname{tr}(V^\mathsf{T} A + V^\mathsf{T}\mathbf{1}_m^m A Z) \\
&= \operatorname{tr}(V^\mathsf{T} A) + \operatorname{tr}(V^\mathsf{T}\mathbf{1}_m^m A Z) .
\end{aligned}$$

For the rightmost term, using multiplication-cyclic invariance gives $\operatorname{tr}(V^\mathsf{T}\mathbf{1}_m^m A Z) = \operatorname{tr}(Z V^\mathsf{T}\mathbf{1}_m^m A)$. Moreover, using $Z V^\mathsf{T}\mathbf{1}_m^m = (\mathbf{1}_m^{m\,\mathsf{T}} V Z^\mathsf{T})^\mathsf{T}$ with $\mathbf{1}_m^{m\,\mathsf{T}} = \mathbf{1}_m^m$ yields

$$\begin{aligned}
\operatorname{tr}(U^\mathsf{T} A) &= \operatorname{tr}(V^\mathsf{T} A) + \operatorname{tr}((\mathbf{1}_m^m V Z^\mathsf{T})^\mathsf{T} A) \\
&= \operatorname{tr}(V^\mathsf{T} A + (\mathbf{1}_m^m V Z^\mathsf{T})^\mathsf{T} A) \\
&= \operatorname{tr}((V + \mathbf{1}_m^m V Z^\mathsf{T})^\mathsf{T} A) .
\end{aligned}$$

□

We call the mapping from V to U specified in Lemma 4 the transpose of θ and denote it as θ^T. The expression of Lemma 4 is essential when searching for linear trails, as reported on in Section 5.5.3.

5.4 Diffusion Properties

The diffusion properties of an MDS matrix are typically summarized by its differential and linear branch numbers [DR02] that are both the same and equal to the dimension of the matrix plus 1. This dimension is also the number of elements a single-element difference propagates to. The study of the diffusion properties of an MDS matrix is largely independent of the other steps in the round function. The proof of the fact that every 4-round trail in AES has 25 active S-boxes requires from the MixColumns matrix only that it is MDS. Of course, that proof also requires a property of ShiftRows.

Diffusion in column-parity mixers is more subtle: their properties only become meaningful in the context of a design approach, where interaction with other steps of the round function must be taken into account. In this section we discuss some concepts that are shared by all column-parity mixers.

5.4.1 The Column-Parity Kernel

The set of states A with column parity equal to zero form a vector space with dimension $n(m-1)$. Following [BDPV11b], we call this the *(column-parity) kernel*. For states A in the kernel the parity is zero and consequently θ reduces to the identity mapping.

There are states in the kernel with Hamming weight 2, namely all states with a pair of active bits in the same column. Again following [BDPV11b], we call this an *orbital*. States in the kernel have an even number of active bits per column and as observed in [MDA17], they can be seen as a collection of orbitals. Due to the existence of single-orbital states, the (Hamming) branch number, both differentially and linearly, of every column-parity mixer is at most 4. We prove that for all reasonable choices of m, n and Z, the branch number is 4.

Lemma 5. *An invertible CPM θ with $m \geq 2$ and where Z has no all-zero rows or columns, has differential and linear branch numbers 4, with the only exception of the case where $m = n = 2$ and $Z = \mathbf{I}$.*

Proof. The branch number is at most 4 as θ will map a single-orbital state to a single-orbital state.

Let us first look at the differential branch number. The differential branch number can only become smaller than 4 if there is a single-bit state that is mapped to a state with less than 3 bits by θ or θ^{-1}.

Let us first look at a single-bit state at the input of θ. A single-bit state A leads to a single-bit parity. Let the number of affected columns in the θ-effect be x. We know that $x \geq 1$ as Z has at least one 1 per row. If these affected columns overlap with the column with the single 1 in A, then the Hamming weight of $\theta(A)$ is $xm - 1$. If not, it is $xm + 1$. Let v denote the sum of the Hamming weights of A and $\theta(A)$. It follows that $v = xm$ and $v = xm + 2$, respectively. We now distinguish between the cases where m is even and where m is odd.

- Even m: if there was no overlap, $v = xm + 2 \geq 4$. If there was overlap, $v = xm = 2$ only if $x = 1$ and $m = 2$. So this is the case of a single affected column overlapping with the odd column, but this is exactly what is excluded in the lemma. As for even m we have $\theta^{-1} = \theta$, that case is proven too.
- Odd m: if $x = 1$, i. e., there is one affected column, then it cannot overlap. Namely, if that is the case, it implies a single 1 in the corresponding row that is on the main diagonal. However, for θ to be invertible, $Z + \mathbf{I}$ must be invertible and if Z contains a row with a single 1 and that is on the main diagonal, that row is 0 in $Z + \mathbf{I}$ and hence it is not invertible. The same reasoning applies to θ^{-1} as $Z' + \mathbf{I}$ with Z' the parity-folding matrix of θ^{-1} must be invertible.

For the linear branch number it suffices to replace θ by θ^{T} and rows of Z become columns of Z. □

Example 2. The simplest possible CPM that satisfies the conditions of Lemma 5 operates on two rows and two columns and has

$$Z = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} .$$

If the 4 bits of the state are arranged in a 4-bit column vector, θ can be expressed as multiplying it with the following familiar matrix:

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} .$$

This matrix was first used in the block cipher MMB [DGV93] and is still used in many modern lightweight ciphers (sometimes modulo some row and/or column permutations), where it is often referred to as a near-MDS matrix. See for instance Minalpher [STA+15], L_0 and L_3 in PRIDE [ADK+14], or Midori [BBI+15]. It has an implementation cost of 1.5 additions per bit. To see that the CPM with the given Z and the matrix are very similar, let

$$A = \begin{bmatrix} a & b \\ c & d \end{bmatrix} .$$

Then, using a CPM:

$$\theta(A) = A + \mathbf{1}_2^2 AZ = \begin{bmatrix} a & b \\ c & d \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} a+b+d & a+b+c \\ b+c+d & a+c+d \end{bmatrix} .$$

Similarly:

$$\begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \\ 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a+b+d \\ a+b+c \\ b+c+d \\ a+c+d \end{bmatrix} .$$

We now consider the size of the column-parity kernel by counting the number of states with a given number of orbitals. The number of single-orbital states, i. e., states that have in total 4 active bits before and after θ, is given in following lemma.

Lemma 6. *The number of single-orbital states is* $n\binom{m}{2}$.

Proof. The orbital can be in one of n columns and the number of possible positions of the bits of the orbital in the column is $\binom{m}{2}$. □

If we divide the number of 2-bit states in the kernel by the state size, we obtain $\frac{m-1}{2}$. In this respect, it is advantageous for a given state size to have a small number of rows and a large number of columns.

We can similarly compute the number of states in the kernel with 2, 3, 4, etc. orbitals. The number of such states increases exponentially with the number of active bits. For example, we have the following lemma on the number of two-orbital states.

Lemma 7. *The number of two-orbital states is* $\binom{n}{2}\binom{m}{2}^2 + n\binom{m}{4}$*, where the second term disappears for* $m < 4$*.*

Proof. We can partition the set of states with two orbitals into two subsets: those where the orbitals are in different columns and those where they are in the same column. For the first set of states, one can position the two orbitals in the n columns in $\binom{n}{2}$ ways and each of the two orbitals can take $\binom{m}{2}$ positions. Their product accounts for the first term. For the second set of states, the four active bits can be in one of n columns and the number of possible positions among the m bits of that column is $\binom{m}{4}$. If $m < 4$, of course no four bits can be stuffed in a column and the second term disappears. □

5.4.2 Propagation of Isolated Bits

Often one tries to determine the minimum number of rounds such that each output bit depends on each input bit, typically called *full diffusion*. One way of measuring full diffusion is propagating single-bit input differences or a single-bit output mask through the rounds. In this context it is interesting to see how single-bit differences propagate.

A single-bit difference at the input of θ propagates to on average $1 + |Z|m$ bits, with $|Z|$ the Hamming weight of Z divided by its dimension. For the circulant case, this is exactly $1 + |z(x)|m$ with $|z(x)|$ the Hamming weight of the parity-folding polynomial, assuming its constant term is 0. The same holds for propagation of single-bit masks at the output of θ to the input. For a single-bit difference at the output (or single-bit mask at the input) of θ we have to distinguish between two cases. If m is even, the column-parity mixer is an involution and the same properties hold as in the forward direction. For odd m, we have to consider the column-parity mixer defined by $Z' = (Z + \mathbf{I})^{-1} + \mathbf{I}$. The matrix Z' may have much higher Hamming weight than Z, as is illustrated by θ in Keccak [BDPV11b] and even for large states, full diffusion in the backward direction can be immediate.

5.4.3 Comparison to Other Mixing Layers

Mixing layers can be compared by their cost, where cost can mean the number of additions, but also the number of cycles on a specific CPU microarchitecture or other quantities. Here we consider the number of binary additions or two-input `XOR` gates, as that is independent from CPUs or standard cell libraries.

Mixing layers should also be compared by their *mixing power*. This is best quantified by bounds on linear and differential trails, but that requires to study a full permutation. We will do so in Section 5.6.3, but for the moment we restrict ourselves to only consider the mixing layer itself. This implies that we are limited to use the linear and differential branch numbers that should be high relative to the dimensions of the state.

Ever since SHARK [RDP+96] and AES [DR02], MDS matrices have been a common choice for a mixing layer as their branch numbers are optimal. The past few years have seen a lot of work on searching for more efficient MDS matrices. A noteworthy development is that of serialization of the mixing layer, first put forward by PHOTON [GPP11] and LED [GPPR11]. This decreases the hardware area that is required to implement matrix multiplication. Another line of work focussed more directly on searching MDS matrices that require fewer additions [BKL16; LS16; LW16; LW17].

While there has been a lot of improvement, it was always assumed that there was a fixed cost of $n(n-1)m$ additions [BKL16], derived from accumulating all the multiplication results. Recently, it has been shown that this lower bound does not really hold when global optimizations are taken into account [KLSW17]. In fact, there exist MDS matrices that can be implemented with fewer additions.

Other ciphers such as PRIDE [ADK+14], Midori [BBI+15], Minalpher [STA+15], and SKINNY [BJK+16] have dropped the MDS requirement to achieve a better trade-off between performance and security.

Table 5.1 provides a comparison of mixing layers. Most cost numbers are computed using the results of [KLSW17]. We list the number of additions per bit, in order to normalize for the different dimensions. While more lightweight mixing layers exist, CPMs offer a good trade-off. It should be noted, however, that bounds over multiple rounds for a concrete scheme are much more meaningful than just a pair of branch numbers.

5.5 A General Design Strategy

In this section, we present a design strategy for the round function of nibble- or byte-oriented iterative block ciphers or permutations, viewed as a substitution-permutation network (SPN), where a CPM is embedded as its mixing layer. We also outline a search strategy for truncated linear and differential trails. In Section 5.6, we instantiate this design strategy with a specific permutation and give security bounds and implementation benchmarks.

5.5.1 Structure of the Round Function

We take the b-bit state to be a rectangle with m rows and n columns. The state consists of mn cells of size ℓ, so $b = \ell mn$. The cells are typically nibbles ($\ell = 4$)

Table 5.1: Comparison of mixing layers.

Cipher/permutation	Type	Dimensions	Additions per bit	Branch number
AES	MDS	$4 \times 4, \mathbb{F}_{2^8}$	3.03	5
Joltik [JNP15]	MDS	$4 \times 4, \mathbb{F}_{2^4}$	3	5
PHOTON [GPP11]	MDS	$6 \times 6, \mathbb{F}_{2^4}$[*]	5[†]	7
Prøst [KLL+14]	MDS	$16 \times 16, \mathbb{F}_2$	4.5[†]	5
Midori [BBI+15]	Not MDS[‡]	$4 \times 4, \mathbb{F}_{2^4}$ or $\mathbb{F}_{2^8}$	1.5	4
Minalpher [STA+15]	Not MDS[‡]	$4 \times 4, \mathbb{F}_{2^4}$	1.5	4
Prince [BCG+12]	Not MDS	$64 \times 64, \mathbb{F}_2$	1.5	4
SKINNY [BJK+16]	Not MDS	$4 \times 4, \mathbb{F}_{2^4}$ or $\mathbb{F}_{2^8}$	0.75	2
Keccak-f [BDPV11b]	CPM	$5 \times 5 \times w$[§]$, \mathbb{F}_2$	2	4
Circulant CPM	CPM	$m \times n$	$2 + \frac{h-2}{m}$[‖]	4

[*] Dimensions are for P_{100}, P_{144}, and P_{196}.

[†] Unknown whether it can be computed with fewer additions.

[‡] Can also be considered to be a CPM, following Example 2.

[§] Where $w \in \{8, 16, 32, 64\}$, depending on which variant of Keccak-f.

[‖] Where h is the Hamming weight of $z(x)$. Additions/bit are between $2 - 1/m$ and $2 + (n - 2)/m$ (inclusively).

or bytes ($\ell = 8$) for software-performance reasons, although other values are also possible. The cells can be seen as elements of $\mathbb{F}_{2^\ell}$.

The round function consists of four layers:

- A nonlinear S-box layer γ, that applies the same invertible S-box to every cell.
- A column-parity mixer θ. Although it operates on elements of $\mathbb{F}_{2^\ell}$, the parity-folding matrix only contains 0 and 1.

- A transposition layer. This consists of two steps: one that permutes the rows called π and one that performs cyclic shift operations (rotations) on the rows, called ρ.
- The addition of round constants.

In the case of a block cipher, round keys can be added between the rounds that are derived from a cipher key (and a possible tweak in the case of tweakable block ciphers) by means of a key schedule. In this chapter we consider the design of key schedules out of scope.

The roles of the different layers are the same as in the well-known wide-trail strategy.

The role of the round constants is to remove invariants of the round function:

- Invariance of the round function with respect to simple transformations of the mappings of the state, such as rotations. Attacks relevant in this context are rotational cryptanalysis [KN10].
- Invariance of the round function across the different rounds. Relevant in this context are slide attacks [BW99].
- Existence of sets that are invariant under the round function. Relevant attacks are here invariant subspace attacks [LAAZ11].

5.5.2 Outline of the Steps in our Design Approach

We assume the designers are faced with the request to design a permutation or block cipher with some given width b, that must be efficient on some given set of platforms. They also know whether side-channel attacks are a concern leading to the requirement of constant-time code or the ability to mask efficiently, and whether the inverse of the mapping must be efficient.

The first step in the design process consists of choosing ℓ, m, and n with the constraint $\ell m n = b$. As we have showed in Section 5.4.1, choosing m

large compared to n implies that there will be more states in the kernel of θ. Diffusion of isolated bits, however, benefits from choosing m large, as shown in Section 5.4.2.

Once ℓ is determined, one can design the γ S-box quasi-independently of the other components of the round function. Important metrics are its algebraic degree, the maximum input-output correlation, the maximum differential probability, but also its implementation cost. For the latter, one should take into account which are the main implementation targets, such as ASICs, bitsliced software, or software using table lookups. This is not different from most design strategies.

The design of θ, π, and ρ can be performed in two phases, taking advantage of the kernel property of CPMs.

In a first phase, propagation is investigated of difference patterns and masks inside the kernel and that are *truncated* [Knu95]. The latter means that at cell level we do not consider the actual values but only whether the cell in the pattern is passive (all-zero) or active (nonzero). For such patterns, θ acts as the identity function, as we impose that they are in the kernel. Moreover, γ acts as the identity, because the S-box is invertible and hence a nonzero input difference leads to a nonzero output difference, and any sum of bits at the output is balanced and hence a nonzero mask at the output cannot propagate to an all-zero mask at the input. It follows that the propagation of such patterns is fully determined by the transposition steps π and ρ. As these steps just move cells around and do not mix them, this propagation is fully *deterministic*. Moreover, transpositions are their own transpose and hence masks and difference patterns propagate in the same way. Hence one can take a pattern A_0 in the kernel and propagate it forward through $\rho \circ \pi$ resulting in A_1. If the resulting pattern can be in the kernel, one can propagate one more round and so on. A pattern that has at least one column with a single active cell cannot be in the kernel, all other patterns can. In a column with two active cells, these must have the same value (2^ℓ possibilities). In a column with three active cells, if two of the cells have value x and y, the third has value $x + y$ and hence it follows that $x \neq y$ ($2^\ell(2^\ell - 1)$ possibilities). As the number of active cells grows, more cases can be distinguished. In any case, a pattern can be propagated through the rounds until it has a column with a single active cell.

This first phase can be used to select the rotation distances of ρ and the π permutation. This allows to reduce all possible values to a reasonably-sized set of candidate parameter values. For instance, if ρ_0 and ρ_1 are two rotation distances and $\rho_0 = \rho_1$, it makes it easier to remain longer in the kernel: a single orbital in some column will, after $\rho \circ \pi$, still be a single orbital. Hence this should be avoided. Similarly, when $\rho_i \neq \rho_j$ but $\rho_i = c\rho_j$ for all $i, j \in [0, m-1], i \neq j$ where c is some integer, there exist patterns with multiple orbitals that remain in the kernel after applying the transposition layer.

The obtained bounds on trails in the kernel can then be used in the second phase to decide on the amount of work one wants to do in the parity-folding matrix. A further selection of Z candidates can be made by means of a full trail search and diffusion criteria such as the number of rounds to reach full diffusion or to meet the (strict) avalanche criterion.

Finally, the round constants can be designed quasi-independently from the other parts of the round function, such that various invariant attacks are not a concern [BCLR17].

5.5.3 Searching Linear and Differential Trails

Given a candidate design that follows the outlined approach, its resistance against truncated linear and differential cryptanalysis should be quantified by scanning a large search space of trails. For most ciphers nowadays, this search is performed using Mixed Integer Linear Programming (MILP). However, recently a very different technique was used to find trails for Keccak, which appeared to be quite promising [MDA17]. There, a unique decomposition of a difference pattern is defined, allowing a tree-based generation of so-called two-round trail cores, where many branches can be pruned and symmetry can be used to skip over large parts in the search space. We will explain this in more detail.

We use techniques that are based on that work to find trails for our design. However, to make it work for our design, we have to make a number of modifications. In this section we will also highlight the most interesting differences, that are due to the different dimensions of the states and the choice of a bit-oriented design versus a design with an almost arbitrary cell size ℓ.

Trails. A trail Q is a sequence over r rounds. In the case of linear cryptanalysis, it is a sequence of masks. In the case of differential cryptanalysis, it is a sequence of difference patterns. This section mostly uses the neutral term *state pattern*. A cell in a state pattern is called *passive* when it is 0 and *active* when it is any nonzero value.

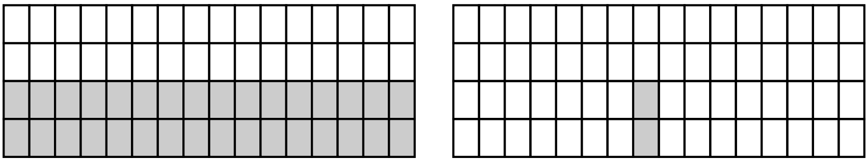
We barely distinguish between linear and differential cryptanalysis, because we have shown in Section 5.3 that masks propagate through θ^{T} in the same way that differences propagate through θ. Furthermore, the γ step acts as the identity function as it does not alter the activity of cells: what is active remains active and what is passive remains passive. The same holds for the addition of round constants. Linear masks also propagate the same as differences through row permutations and rotations. This means that the only difference between searching truncated differential and truncated linear trails is that in differential trails, forward propagation through θ is governed by Z, and in linear trails backward propagation through θ is governed by Z^{T}.

Because we only consider truncated trails, we only look at cells and not at individual bits. The *weight* W of a trail Q over r rounds is defined as $W = \sum_{i=0}^{r-1} |q_i|$, where $|q_i|$ denotes the number of active cells in q_i.

Trivial trails in the kernel. The possibility to remain in the kernel leads to some trivial trails. One example of a starting state pattern is given in Fig. 5.1a, for dimensions $m = 4$ and $n = 16$, with two full rows of active cells. In general, for all dimensions, if the column parity is the all-0 vector, θ will just be the identity function. Furthermore, a transposition layer consisting of row permutations and rotations will be unable to move the pattern out of the kernel. This leads to a trivial trail over an arbitrary number of rounds. However, this trail has a high weight of $W = 2nr$ active cells.

When Z is circulant, when there is only one full row of active cells and when the column parity is therefore the all-1 vector, θ is the identity function when the number of affected columns for a single active cell is even. Then there is a trail of weight $W = nr$ active cells. Recall that a column is affected when its bit in the θ-effect is one. When the number of affected columns is odd, θ will be the complement function. The trail (with even length) will then have weight $W = n\frac{r}{2} + (m-1)n\frac{r}{2} = mnr/2$ active cells.

For reasonable dimensions, n is high enough such that these do not pose a problem. Other state patterns with fewer active cells might be able to stay in the kernel for one or two rounds, but the transposition layer should always move them out of the kernel soon. An example is shown in Fig. 5.1b. This means that the property that a CPM has a kernel will not pose a serious threat.



(a) Remains in the kernel indefinitely. **(b)** Should quickly move out of the kernel.

Figure 5.1: Examples of state patterns in the kernel.

Generating two-round trails. In an r-round trail with weight W the average weight per round differential is $\frac{W}{r}$ and hence it always contains a round differential with weight $L \leq \left\lfloor \frac{W}{r} \right\rfloor$ active cells. When one wants to find all r-round trails up to weight W, one can therefore generate round differentials up to weight L and extend all of them forward and backward $r-1$ rounds. However, it was observed in [MDA17] that there are fewer two-round trail cores with weight below $2L$ than differentials of weight below L. We therefore generate two-round trail cores up to weight $2L + 1$ active cells and extend them $r - 2$ rounds forward and backward.

Similarly to [MDA17], generating the two-round trails is modeled as the traversal of a tree, where each node represents a pair of state patterns at the input and output of θ. The root is formed by the empty state patterns. The idea is that the weight monotonically increases when visiting a node's children. It is then possible to prune the branches of which it is known that all descendants will have a weight higher than $2L$.

As with Keccak-f, we can uniquely decompose every state pattern into a *parity-bare state* and a list of *free orbitals*. A parity-bare state is a state pattern that does not have free orbitals. A free orbital is a pair of two active cells in an unaffected column, as was already mentioned in Section 5.4.1. Removing a free orbital from a state pattern also removes two active cells after θ, so adding a free orbital to a state pattern will always increase the weight of a trail by 4 active cells. This decomposition reduces the problem to that of generating all parity-bare states in a weight-increasing manner, because we can construct all states up to some weight $2L + 1$ by generating all parity-bare states up to that weight and adding free orbitals to those where there is still budget.

A parity-bare state consists of a list of odd columns, i. e., columns with nonzero parity. Initially, at the root node, this list is empty. Then one odd column is added to the list. This means that the children of the root node are all the state patterns with a single odd column. At every level of the tree, new columns are added.

On top of the tree of parity-bare states, all the nodes also form the root of their own tree, now maintaining a list of free orbitals, such that at each level free orbitals are added up to the limit weight. Together, this guarantees that all two-round trails will be found.

Moreover, the rotation-symmetric nature of the state in the horizontal direction and of all operations allow us to prune many more branches of the trees. There is only need to consider *canonical* state patterns, where a state pattern is defined to be canonical if and only if it is minimal under all horizontal translations given a total ordering over state patterns. In this case, the total ordering is just the lexicographical ordering on $[z, y]$, where z is the horizontal axis and y the vertical axis, to keep the notation similar to [MDA17]. The lower left has coordinates $(z, y) = (0, 0)$.

Propagation of cells and trail extension. After having generated two-round trail cores, they need to be extended forward and backward by $r - 2$ rounds. With Keccak-f, extension implies that one needs to check which patterns are compatible through the nonlinear χ step. Given a nonzero state pattern at the end of a trail, there are multiple possibilities after χ and each case needs to be explored. This branching behaviour causes an exponential increase of the search space when extending to more rounds.

However, in our case we are considering truncated trails and γ has no effect on state patterns. This is very different from the situation in [MDA17]. Instead, this branching behaviour is now caused by θ.

To see this, consider a state pattern before and after θ. Assume a column has ≥ 2 active cells before θ. To apply θ, we start by computing the parity of that column. This means that we sum over nonzero differences (in the case of differential cryptanalysis). It is possible that this results in a nonzero difference (i. e., the column is odd), but it is also possible that the nonzero differences cancel each other out and the column becomes even. Both cases should be explored and this causes the search space to branch.

There is another source of branching. We distinguish between four types of columns: even affected, even unaffected, odd affected, and odd unaffected.

When a column of the state pattern is unaffected, θ will not add anything to the cells in that column. Therefore, cells that were active before θ will remain active after θ. Passive cells will also remain passive. We therefore only need to consider affected columns. In affected columns, every cell that is passive before θ becomes active after θ, as a nonzero difference is added to a difference of zero in the case of differential trails. However, if a cell is active before θ, it is unknown whether it will remain active or will cancel out after θ, as some nonzero difference is added to some nonzero difference. A search for truncated trails would need to consider both cases for all active cells in affected columns.

However, one can do slightly better. How an active cell propagates depends on the number of active cells in that column. As an example, let $\#_x$ denote the number of active cells in column x (and let $m \geq 4$). We will make a case distinction on whether x is even or odd, and on $\#_x$, up to $\#_x = 4$.

$\#_x = 0$**, column** x **even.** Nothing can cancel out. Of course, all cells become active (with the same difference value) after θ.

$\#_x = 0$**, column** x **odd.** Impossible.

$\#_x = 1$**, column** x **even.** Impossible.

$\#_x = 1$**, column** x **odd.** That active cell may cancel out or not after θ.

$\#_x = 2$**, column** x **even.** This can only happen when the two active cells have the same difference value. When x is affected, the two active cells either both cancel out after θ, or none of them, but never only one.

$\#_x = 2$**, column** x **odd.** This can only happen when the two active cells have a different difference value. Therefore, either no active cells cancel out after θ, or one of them, but never both.

$\#_x = 3$**, column** x **even.** This can only happen when all three active cells are different. At most one of the active cells cancels out after θ.

$\#_x = 3$**, column** x **odd.** Now there are more possibilities. The active cells can be all the same, all different, or can consist of one pair and one other value. This implies that any in $\{0, 1, 2, 3\}$ active cells cancel out after θ.

$\#_x = 4$, column x even. The active cells are all the same, all different, or consist of two pairs. So any in $\{0, 1, 2, 4\}$ active cells cancel out after θ.

$\#_x = 4$, column x odd. All combinations are possible, except that all active cells have the same difference value. Therefore any in $\{0, 1, 2, 3\}$ active cells may cancel out after θ.

This can be extended further to higher values for $\#_x$, but the relative amount of cases that can be skipped will then decrease. It is clear that the search space of differential trails branches heavily with a CPM.

We wrote dedicated software to traverse the search space of truncated linear and differential trails for designs following our approach. The software is freely available at `https://github.com/Ko-/cpm`.

5.6 The Mixifer Permutation

5.6.1 Design Goals

To show an instance of this design strategy, in this section we introduce a concrete permutation and study its security and efficiency. We aim for a suitable permutation for lightweight applications, such as in the Internet of Things. In this case, with "lightweight" we mean that it should be fast in constant-time code for microprocessors such as the ARM Cortex-M series and that it should require little area in ASICs.

This could be used in a simple (single-key) Even-Mansour construction [EM93] to build a block cipher or in a construction such as XPX [Men16] to build a tweakable block cipher. Based on this, one could even build other primitives such as authenticated encryption. Depending on the mode, we may or may not need the inverse permutation. To make the permutation more flexible, we therefore require that the inverse also needs to be efficient.

For efficient full-width processing in software, we would like a state size that is a power of two. A potential problem with a state size b of 128 bits is that this gives a birthday bound of only 64 bits, which may or may not be an issue depending on the mode in which this permutation is to be used. Again, we design this permutation to be useful in many scenarios, which is why we choose our state size to be equal to the next power of two, $b = 256$ bits. This gives a birthday bound of 128 bits, which should be more than enough.

To achieve fast constant-time code, we aim for a round function that can be efficiently implemented with bitwise Boolean instructions and cyclic shifts. This has implications for the way we arrange the bits of the cells in CPU words.

5.6.2 The Construction

First we have to select ℓ, m and n. To have an efficient inverse of θ, we choose n to be even, such that θ becomes an involution. For an efficient bitsliced implementation of γ, we choose $\ell = 4$. Now both 8 rows and 8 columns, and 4 rows and 16 columns are viable options. We set $m = 4$ and $n = 16$, based on the trivial in-kernel trails discussed in Section 5.5.3.

For γ, applying a 4-bit S-box to every nibble separately can quickly become expensive. We make sure that, in software, the S-box can be applied efficiently on a full row, without using lookup tables or other operations that might lead to timing attacks. We suggest a bitsliced state representation and explore one such possibility in Section 5.6.5. This works when the S-box is rotation-symmetric.

An ℓ-bit S-box $S : \mathbb{F}_{2^\ell} \to \mathbb{F}_{2^\ell}$ is *rotation-symmetric* if and only if $S(a \ggg d) = S(a) \ggg d$ for all $a \in \mathbb{F}_{2^\ell}$, $d \in [0, \ell - 1]$, where $\ggg$ denotes bitwise rotation. Properties of rotation-symmetric S-boxes (RSSBs) have been explored in, e. g., [RBG08] and [Kav12]. Rotation-symmetric S-boxes are determined by a single coordinate function. There are therefore only 2^{2^4} 4-bit rotation-symmetric functions, which makes them efficiently enumerable. We search exhaustively through these candidate S-boxes to find an example with nice cryptographic properties and that also has an efficient implementation. Out of the 65 536 functions, 1536 are invertible. Of course, many are linearly and/or affinely equivalent to each other [LP07]. Out of these 1536 candidates, 512 are *optimal* S-boxes, as defined in [LP07]. This means that they have a maximum differential probability of 1/4 and a maximum input-output correlation of 1/2. All of these have 3 as their algebraic degree. We then selected the S-box based on the number of binary Boolean operations that are required to compute it and its inverse. It can be defined in algebraic normal form by the coordinate function $b_0 = a_1 + a_2 + a_0a_2 + a_1a_2 + a_1a_2a_3$. More details and representations can be found in the appendix of the original publication [SD18].

For the first phase of selecting θ, π, and ρ, we consider trails in the kernel of θ. Because of all the symmetry, we can arbitrarily set one rotation distance to zero, shaving off a few more operations. After some experiments, we set π to be the permutation that rotates all rows down. ρ rotates rows nibble-wise to the right by the distances 14, 3, 10, and 0, respectively, from top to bottom. This yielded a best trail with a weight of 52 active cells over 4 rounds. For implementation efficiency, we also make ρ compute an intra-nibble rotation to the left for all nibbles that are wrapped to the other side. In Section 5.6.5 it will become clear why this is actually more efficient and not less. Note that this does not affect our truncated-trail search.

For the second phase, the parity-folding matrix Z should be chosen such that the truncated-trail search yields bounds that are not much better than the in-kernel trails. We end up with a circulant matrix where a column can affect three other columns. Over 4 rounds, the best differential trail then has a weight of 46 active cells

The steps of the round function have high symmetry. All four γ, θ, π, and ρ are invariant under rotation along the rows and rotation of bits within the cells. Moreover, γ and θ are even invariant under rotation along the columns. We therefore add round constants, that should achieve the following goals:

1. Applying a few rounds to a state that has some symmetry $A \oplus (A \ggg d) = 0$ shall result in a state B where $B \oplus (B \ggg d)$ has high Hamming weight, for all values of d.

2. Applying a few rounds to two states A and A^* with $A' = A \oplus A^*$ and $A' \oplus (A' \ggg d) = 0$ shall result in a difference B' where $B' \oplus (B' \ggg d)$ has high Hamming weight, for all values of d.

3. There shall be no invariant subspaces in the linear part of the round function, including the addition of the round constant.

4. Round-constant addition shall be cheap.

The minimum cost of round-constant addition is a single bitwise addition per round. Therefore, we have round constants that are nonzero in a single word. We choose the word that contains the nibbles of the top row in even positions. We use a simple scheme to achieve asymmetry: all round constants are obtained by performing a (noncyclic) shift of a single master round constant, where the shift offset is simply the round index. We believe criteria 1 and 2 listed above are satisfied for the following reasons. First, the round constants are outside the kernel and hence their influence will spread very quickly. Second, they do not have symmetry along the rows or within the nibbles.

As for criterion 3, we investigated the algebraic properties of the linear part of the round function, as explained in [BCLR17]. It can be seen as 4 identical mappings, each operating on the bits in a specific position of the nibbles. So each one operates on a matrix of 4 rows and 16 columns. The characteristic polynomial of this mapping is $1 + x^{64} = (1 + x)^{64}$. This suggests that it has nontrivial invariant subspaces of dimension 1, 2, 4, 8, 16, 32. These are easy to find, knowing the symmetry. The ones of dimension above 2 simply consist of states that have periodic rows. The subspace of dimension 32 is the set of all states with period 8, dimension 16 with period 4, dimension 8 with period 2 and dimension 4 with period 1 (rows are all-1 or all-0). The subspace with dimension 2 are the 4 states with an even number of all-1 states and an even number of all-0 rows. Finally, the subspace with dimension 1 are the all-0 and the all-1 state.

The 32 active bits in the round constants are spread evenly over these 4 mappings. It would be problematic if the differences between the round constants were all periodic. We made sure that this is not the case.

The master round constant has the following value expressed as a bit sequence: `11000110111010100001001011001111`, starting with the least-significant bit. This has been generated with the 5-bit LFSR with feedback polynomial $1 + x^3 + x^5$. While there are linear recurrences among bits due to a short LFSR, this has no relation with the symmetry in the steps of the round function and we therefore believe that this is not a problem.

The round constant for round i is this sequence shifted to the left by i bits.

Summary. To summarize this and to make this more precise, we introduce a 256-bit iterated permutation called Mixifer. Mixifer consists of 16 rounds. The 256-bit state is arranged as $m = 4$ rows with $n = 16$ columns of 4-bit nibbles. Every round consists of applying $\iota \circ \rho \circ \pi \circ \theta \circ \gamma$.

- γ is a nonlinear mapping, the S-box. It is a 4-bit S-box that is applied to each nibble. We choose the rotational-symmetric S-box defined by the following equation (over $\mathbb{F}_2$) for the first coordinate: $b_0 = a_1 + a_2 + a_0a_2 + a_1a_2 + a_1a_2a_3$.
- θ is our column-parity mixer, defined by dimensions $m = 4$, $n = 16$, and by the following circulant parity-folding matrix Z:

$$Z = \begin{bmatrix} 0 & 1 & 1 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 1 \\ 1 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \end{bmatrix}.$$

Alternatively, the parity-folding polynomial is $z(x) = x + x^2 + x^5$.

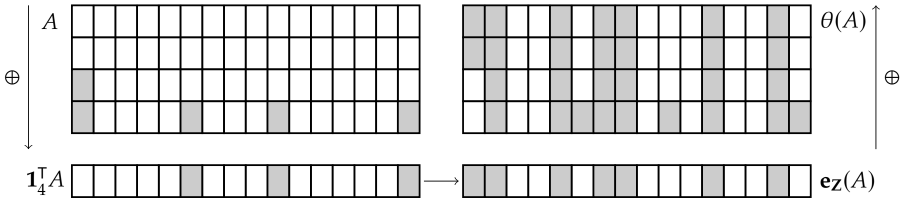


Figure 5.2: The diffusion layer θ.

- π permutes the rows as follows: $0 \mapsto 1, 1 \mapsto 2, 2 \mapsto 3, 3 \mapsto 0$, where $\{0, 1, 2, 3\}$ are row indices, starting from the top.

- ρ rotates the 4 rows nibble-wise to the right by the distances 14, 3, 10, and 0, respectively, from top to bottom. Additionally, all the nibbles that are wrapped to the other side are rotated to the left by 1 bit within that nibble.

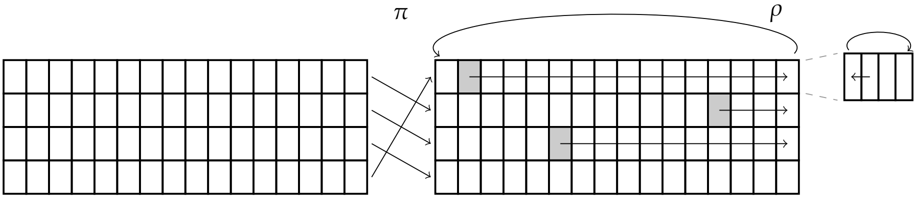


Figure 5.3: The transposition layer $\rho \circ \pi$.

- ι adds a round constant into the even cells of the top row, starting to count at 0, as highlighted in Fig. 5.4. In round i this is `0xF3485763` shifted to the right by i, again starting to count at 0.

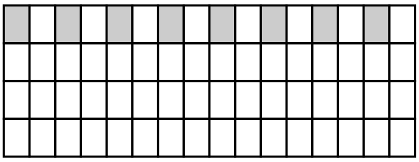


Figure 5.4: Locations where ι adds the round constant.

5.6.3 Evaluation

Avalanche effect. We first study the diffusion of our CPM by considering the avalanche effect for Mixifer. Good diffusion implies that a single bit will quickly affect as many other bits as possible. Avalanche testing is a quick and easy test to quantify how quickly diffusion takes place.

For each of the 256 bits, we generate 10 000 random states. A round-reduced Mixifer is then applied to that state and to that state where one bit is flipped. Each time, we look at the differences between the outputs. We consider the amount of bits that are flipped, but also in how many different cells bits are flipped. It can be seen in Table 5.2 that a single bit flip in a random state has very rapid diffusion. Already after 3 rounds, each bit of the output is flipped with 50.0% probability. This is also known as the strict avalanche criterion.

Table 5.2: Avalanche test flipping a single bit.

Rounds	1	2	3
Average probability bit flip	10.2%	47.1%	50.0%
Average probability cell change	20.2%	88.83%	93.8%
Worst-case bits flipped	13	44	89
Worst-case cells changed	13	32	48

Another way to test the avalanche effect is by starting with a state where only a single bit is set, and to measure how many rounds it takes before every bit in the state is touched. Typically, operations can be changed to `OR`s and it can be checked when all bits in the state are set. The results are listed in Table 5.3. After 5 rounds, 'full' diffusion is achieved from every starting bit.

Table 5.3: Avalanche test starting with single bit set.

Rounds	1	2	3	4	5
Worst-case bits set	71	196	243	255	256
Worst-case cells set	25	58	63	64	64

Linear and differential cryptanalysis. A strategy for searching truncated linear and differential trails has been outlined in Section 5.5.3. Here, we only summarize the trails and bounds that we computed for Mixifer.

First it should be verified that the kernel of the CPM does not pose a serious concern and that it is infeasible to stay in the kernel for many consecutive rounds, except for trivial trails where two full rows are active, as was discussed in Section 5.5.3. We generated trails where the state pattern remains in the kernel until the last round. The number of active cells per state pattern in the trail is therefore constant. The in-kernel bounds are summarized in Table 5.4 and the actual figures are in Figs. 5.5 and 5.6.

Table 5.4: Minimum weight for trails in the kernel until the last round.

Number of rounds r	2	3	4
Minimum W active cells	4	18	52

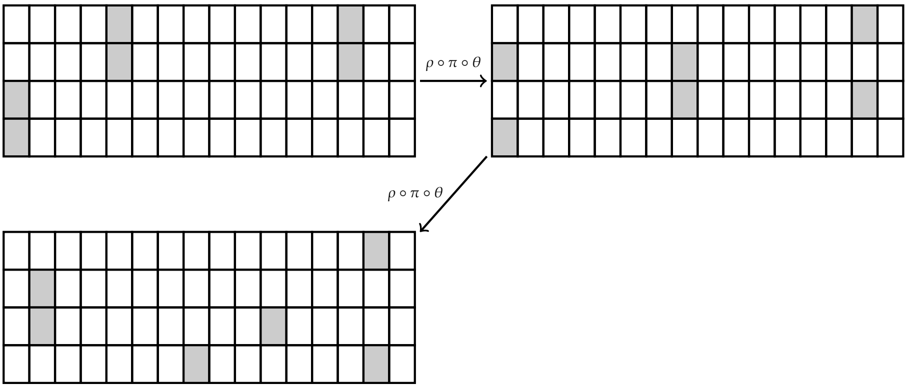


Figure 5.5: Trail in the kernel until the third round with $W = 18$.

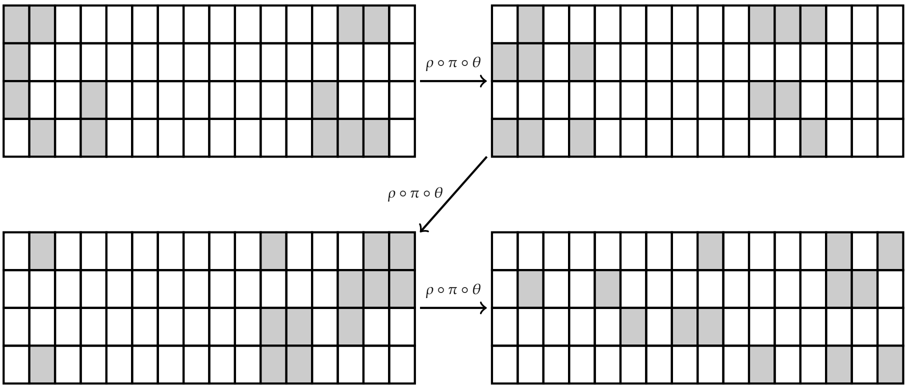


Figure 5.6: Trail in the kernel until the fourth round with $W = 52$.

Outside of the kernel, we first use our software to generate two-round trails. After extending these two-round trails, we find the minimum-weight trails for three and four rounds, both for linear and for differential trails. For three rounds, both the best linear and the best differential trail have a weight of 27 active cells. The minimum-weight trails can be found in Figs. 5.7 through 5.10.

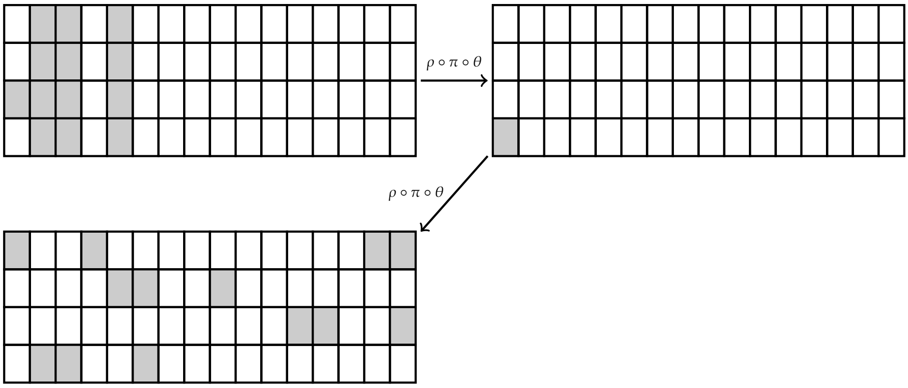


Figure 5.7: Differential trail over 3 rounds with $W = 27$.

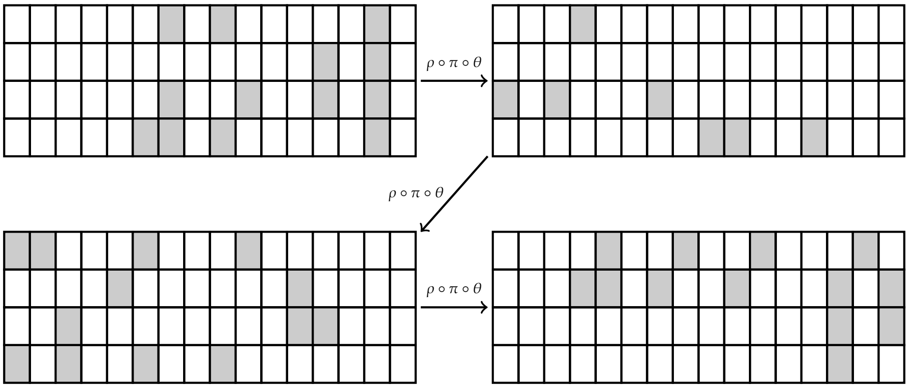


Figure 5.8: Differential trail over 4 rounds with $W = 46$.

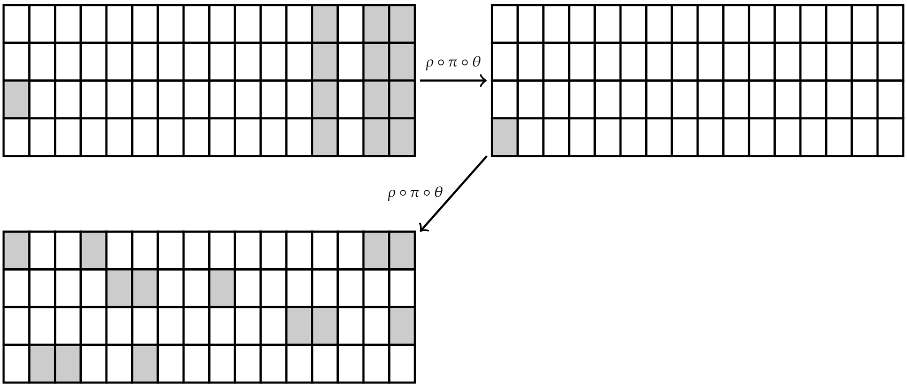


Figure 5.9: Linear trail over 3 rounds with $W = 27$.

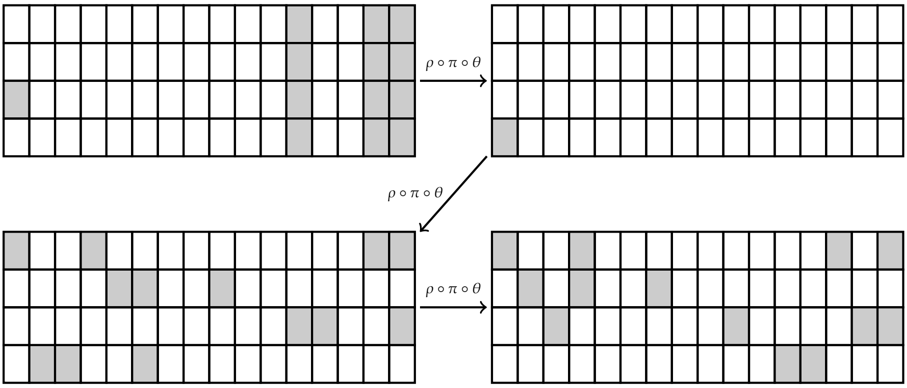


Figure 5.10: Linear trail over 4 rounds with $W = 40$.

For this experiment, Fig. 5.11 shows the number of two-round differential trails that needed to be considered and their weights. The plot for linear trails would look similar.

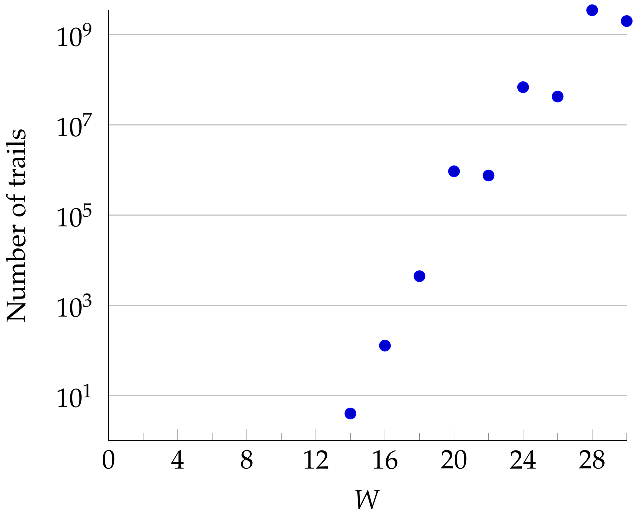


Figure 5.11: Number of active cells of two-round differential trails.

Table 5.5 shows the overall minimum weights, combining trails in the kernel and outside the kernel. It can be seen that after four rounds, the trails in the kernel are no longer the minimum-weight trails. We expect the weight to again increase significantly for subsequent rounds, but we were not able to cover the full search space anymore.

Table 5.5: Minimum weight in number of active cells for trails.

Number of rounds r	1	2	3	4
Minimum W differential	1	4	18	46
Minimum W linear	1	4	18	40

Clustering of differential trails. So far we have considered truncated trails. However, as the round function is cell-oriented, there may be significant cases of trail clustering. In particular, there may be multi-round differentials with many differential trails in the same truncated differential trail.

For two-round AES, this was investigated in [DR06] and it turned out that the differential over two rounds has a maximum differential probability (DP) of 13.25×2^{-32}, while the best two-round trail has DP 4×2^{-32}. The differential is the result of 75 trails, all in the same truncated trail. These investigations were done on an interesting sub-structure of AES, the so-called *AES super-box*. This structure is a permutation operating on a 4-byte array and consists of the SubBytes S-box layer, MixColumns, round key addition, and again a SubBytes S-box layer, all restricted to a 4-byte array. The best differentials over that structure have in total 5 active S-boxes over the two SubBytes layers and those are the ones that are investigated.

For Mixifer, we analyze an analogous case, namely the best differential trails over two rounds. Those are the ones with two active cells that form an orbital in θ of the first round. So we have two nibbles in the same column with difference values d_0 and d_1 at the input of γ, that maps them both to the same difference d_Δ. The subsequent θ acts as the identity and ρ and π move them to different positions before they arrive at the next γ layer. There, the two active nibbles map to output differences d_2 and d_3. The remainder of the second round does not modify the differential probability. So the super-box structure we consider consists of two γ S-boxes, followed by two γ S-boxes. There is no mapping in between, only the requirement that in the intermediate difference the two active cells are equal. So the trails look like this: $(d_0, d_1) \xrightarrow{\gamma} (d_\Delta, d_\Delta) \xrightarrow{\gamma} (d_2, d_3)$. This means that their differential probability is $\mathrm{DP}(d_0, d_\Delta)\mathrm{DP}(d_1, d_\Delta)\mathrm{DP}(d_\Delta, d_2)\mathrm{DP}(d_\Delta, d_3)$. Assuming independent rounds, they contribute to the super-box differential in the following way:

$$\mathrm{DP}((d_0, d_1) \to (d_2, d_3)) = \sum_{d_\Delta} \mathrm{DP}(d_0, d_\Delta)\mathrm{DP}(d_1, d_\Delta)\mathrm{DP}(d_\Delta, d_2)\mathrm{DP}(d_\Delta, d_3)\,.$$

A single trail can have DP at most 2^{-8}, as the maximum DP of our S-box is 2^{-2}. We have exhaustively checked for all 2^{16} possible combinations of (d_0, d_1, d_2, d_3), and the differential with the highest DP turned out to be the one where all d_i are all-1 (so `F` in hexadecimal). It has DP $2^{-8} + 6 \times 2^{-12}$ that is the contribution of 7 trails, one dominant with DP 2^{-8} and 6 with DP each 2^{-12}. This gives a DP for the differential that is only a factor 1.375 higher than that of its dominating trail.

As a preliminary conclusion, it appears that the effect of clustering appears to be smaller than for AES.

Impossible-differential cryptanalysis. It is well-known that every permutation has many impossible differentials [BBS05]. For any input difference, there are exactly 2^{b-1} pairs and hence there can be only 2^{b-1} output differences. However, to the best of our knowledge no attacks have been reported that exploit that aspect. Impossible-differential attacks exploit large classes of impossible differentials, i. e., differentials with zero probability. Here we will discuss whether this can be applied to Mixifer.

Let us consider a single-cell difference at the input of a round (called *first round*):

- First round: θ maps the single active cell to a difference with three additional affected columns and the subsequent ρ moves the active cells to different columns.

- Second round: γ transforms the difference values of the 13 active cells to possibly different values. The subsequent θ adds affected columns. We found that at this stage, *all* columns can be affected. The subsequent ρ and π steps just move the active cells around.

- Third round: γ transforms the difference values of the active cells to possibly different values. Within each column, all possibilities can occur: in the kernel or out of the kernel, all four the same 4, three the same $3 + 1$, $2 + 2$, $2 + 1 + 1$ and all four different $1 + 1 + 1 + 1$. The subsequent θ computes

> column parities over all columns and adds three affected columns for each one. This leads to a state where each cell can be active or passive, with the exception of an all-passive state. This remains to be the case after ρ and π.

So a difference with a single active cell may lead to any (nonzero) truncated difference after three rounds. If we consider fully specified patterns taking into account the cell-difference values, we have verified that, when we apply a difference a to the serial composition of two S-boxes with the addition of an offset in between, that this may lead to all possible nonzero differences for any nonzero value a. So, even though for any stage of the computation there will certainly exist (nontruncated) difference patterns that cannot occur, after γ of the fourth round, it will be very hard to exploit them.

We treated the case of a difference pattern with a single active cell. If there is more than one active cell, one may try to slow down diffusion by having the pattern at the input of θ of the first round in the kernel. In that case, one would apply a pattern at the input consisting of one or more orbitals. However, after γ of the first round, the difference values of the active cells belonging to the same orbital depend on the absolute value of the state, and hence they are only equal in a subset of all cases. So for the other cases that may occur and are hence not impossible, there are actually more active cells present after θ and hence diffusion is even faster.

We see that Mixifer has no exploitable impossible differentials that start from a single-cell difference over more than 3 rounds. This is actually as good as AES despite the larger number of cells, and we attribute it to the large average diffusion of the column-parity mixer that we use.

Invariant attacks. Under invariant attacks, we consider rotational cryptanalysis [KN10], slide attacks [BW99], invariant subspace attacks [LAAZ11], and nonlinear invariant attacks [TLS16]. One property that these attacks share, is that they can be defeated by appropriately choosing round constants [BCLR17].

We explicitly designed our round constants to fulfill all the relevant criteria, which is why we conclude that there are no exploitable invariant attacks against Mixifer.

5.6.4 The Number of Rounds

Recall that we aim for use cases such as XPX [Men16], where an adversary can apply difference patterns across the complete input and can do inverse permutation queries, and that we target a security level of 128 bits. Note that this is different than for block ciphers where by default a security level equal to the block length is claimed, leading to the absurd situation that attacks requiring an adversary to query almost the full codebook can "break" a cipher, at least from an academic perspective. Our security claim is compatible with the fact that most modes do not achieve security above the birthday bound, that is at a comfortable 128 bits for our permutation width.

Based on our investigations, we fix the number of rounds to 16 and believe that this gives a comfortable safety margin for the following reasons:

- The best differential trails over 4 rounds have DP 2^{-92}, so the best ones over 12 rounds have an approximated DP of 2^{-276}. In our best 2-round differential trail, the DP is 2^{-8}, while the best 2-round differential has DP $1.375 \times 2^{-8} \approx 2^{-7.54}$. So clustering may lead to some loss, but even dividing the exponent of the best trail DP by two would give 2^{-138}, an unexploitable value.
- For linear trails we have a maximum linear potential (square of the correlation) over 4 rounds of 2^{-80}, resulting in 2^{-240} over 12 rounds. This is slightly less comfortable than the case of differential trails, but it is still very large, even if a huge degeneration due to clustering would occur.
- The number of rounds for which structural attacks, such as integral cryptanalysis and impossible differentials, still work, is strongly correlated

with the number of rounds that it takes to achieve full diffusion. We have illustrated this explicitly with impossible differentials. Specifically, we claim that despite its larger state and more lightweight round function, Mixifer achieves full diffusion about as fast as AES.

- The algebraic degree of the round function is 3, as is that of its inverse. This already starts saturating after 5 rounds, so we do not expect there to be any problems.

Of course every concrete cipher can only be considered secure after intensive public scrutiny so we invite all members of the cryptographic community to attack Mixifer.

5.6.5 Implementation Cost

Mixifer is designed for efficient constant-time implementations in hardware and in software on 32-bit and 64-bit architectures. We provide two reference implementations in C and an optimized assembly implementation for the (32-bit) ARM Cortex-M3 and M4 microprocessors.

The first C implementation uses 4 `uint64_ts` to represent the state and implements the round function in a very straightforward way. In a naive software implementation like this, γ can be relatively computationally expensive. The S-box needs to be computed for each cell individually. Even when one applies the S-box to an entire row in parallel, bitmasks are typically required to separate the four bits of a cell over separate registers. By selecting a rotation-symmetric S-box and assuming a bitsliced state, this can be made more efficient, especially on architectures such as those by ARM where shifts and rotations are cheap. After loading our state, we take it to be organized as in Fig. 5.12.

The ith byte of a 32-bit word stores the ith bit for 8 cells, as the S-box can then be computed in just 4 instructions per register. We also choose to interleave adjacent cells over two words, as a word-rotation by one nibble is now only a bytewise rotation by one bit and a swap of registers, which is free, instead of having to do this bytewise rotation on two words. The second C implementation uses this approach and represents the state with 8 `uint32_ts`.

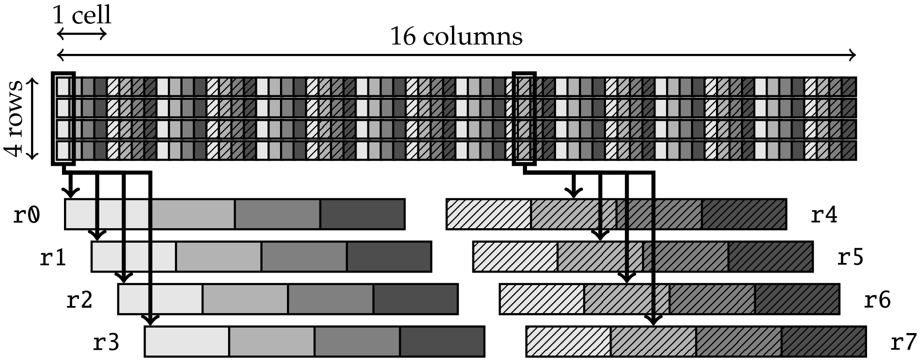


Figure 5.12: Bitsliced representation for 32-bit architectures. At the top is the 256-bit state, where every small rectangle is a bit. At the bottom are 8 32-bit registers labeled `r0` to `r7`, where every colored rectangle is a byte.

Chapter 6 provides more details on the microarchitectures, but on the ARM Cortex-M3 or M4, we can compute γ with this representation in 32 single-cycle instructions. On a 64-bit architecture, this could naturally be twice as efficient. Computing θ takes 6 cycles for computing the parity, 17 cycles for calculating the effect, and 8 cycles for adding it back to the state. π can be implemented by just renaming registers, and ρ takes us 2 cycles per row, except for the last row which is not rotated, so only 6 cycles in total. ι takes a single cycle.

All combined, our optimized assembly implementations needs 70 cycles for a single round. Together with some overhead for a function call and for loading and storing data, the full unrolled 16-round permutation runs in 1174 cycles, or 36.69 cycles per byte on the ARM Cortex-M4, as measured on an STM32F407 microcontroller. For ARM Cortex-M3, we measure 1175 cycles on an STM32L100 microcontroller. The implementation uses only 44 bytes on the stack, excluding the input. They are mostly to store callee-save registers.

Notably, the inverse permutation has the same memory requirements and even a slightly higher speed, measuring 1126 cycles on the STM32F407.

All source code is put into the public domain and available at `https://github.com/Ko-/mixifer`.

5.6.6 Comparing to Other Ciphers

To put our bounds and benchmarks into context, we compare the results to some other permutations and ciphers. We aim to make this comparison fair by only considering schemes for which there is an optimized implementation for the ARM Cortex-M3 or M4. Aside from the usual speed and size numbers, we would also like to somehow incorporate the *quality* of a round function into the comparison.

Some schemes have many lightweight rounds while others have more computationally demanding rounds, but need fewer of them to reach a certain level of security. Some schemes are also more conservative than others with regard to the number of rounds compared to the best known attacks. Moreover, schemes target different security levels. All of that makes it hard to quantify the notion of the quality of a round function in a meaningful way. Intuitively, however, we believe that it would be interesting to look at the amount of *security* that is gained per amount of *work* that has to be done.

Table 5.6 lists some of the results. Regarding security, we consider the weight that a single round adds to the best known differential trail. In the table, the weight is expressed as the $-\log_2$ of the maximum differential probability (DP). *Work* is then quantified as the number of cycles per byte per round on these Cortex-M microarchitectures. Of course these numbers do not reflect security against different types of attacks, performance on different architectures, the security margin, and so on.

AES [DR02] is a prime example of a cipher using an MDS matrix and with strong bounds. For fair comparison, we also mention performance numbers for a bitsliced implementation that does not use any secret-data dependent lookup tables and is not vulnerable to cache attacks. Gimli [BKL+17] is a 384-bit permutation with a much lighter round function which shows in its speed, but also in the best known trails. We compare to Salsa20/20 [Ber08b] because it is known to be very fast in software. The speed that is mentioned, is actually that of ChaCha20 [Ber08a], which has comparable speed. However, to the best of our knowledge, no bounds are known for ChaCha20 and Salsa20/20 does not have an optimized implementation for the Cortex-M3 and M4. We compare to Keccak-$f[400]$ and Keccak-$f[800]$ [BDPV11b] because they are known to have very strong security guarantees against linear and differential cryptanalysis in the form of a good bound on the weight of trails [MDA17]. The big gap in performance is because Keccak-$f[800]$ is the smallest Keccak-f that can fully use 32-bit registers.

It is clear that Mixifer may not be as fast as Salsa20/20 or Gimli, but their currently available bounds leave something to be desired, although they may improve in the future. When one considers the amount of *security* that one gets per amount of *work* that needs to be done, Mixifer stands out.

5.7 Conclusions and Future Work

We have generalized the mixing layer of the permutation of Keccak and showed that column-parity mixers are an interesting alternative to MDS matrices. They can be very lightweight mixing layers and they lend themselves well to bitsliced implementations, even for nibble-oriented ciphers and permutations. Additionally, we formulated a strategy to obtain strong bounds with respect to truncated linear and differential cryptanalysis and demonstrated its effectiveness by an exemplary permutation called Mixifer.

Table 5.6: Comparison of performance on the Cortex-M3/M4 and bounds.

Name	r	Speed (cycles/byte)		Bound trails	
		Full	$/r$	$-\log_2(\max DP)$	$/r$
AES bitsliced	10	50.52 [SS16c]	5.05	150 over 4 [DR02]	37.5
AES tables		39.97 [SS16c]	4.00		
Gimli	24	21.81[BKL+17]	0.91	52 over 8[BKL+17]	6.5
Keccak-f[400]	20	106 [BDH+]	5.3	92 over 6 [MDA17]	15.3
Keccak-f[800]	22	48.02 [BDH+]	2.18	92 over 6 [MDA17]	15.3
Salsa20/20	20	13.88 [HRS16]	0.69	18 over 3 [MP13]	6
Mixifer	16	36.69	2.33	92 over 4	23

CPMs look promising, but there are still some interesting aspects that are worth investigating. For instance, intuitively it makes sense that a sparser parity-folding matrix means less diffusion and a cheaper implementation, but it is less obvious what would be an ideal trade-off and how this relates to the trail search space that needs to be covered. One can imagine that after some point, there is not so much to be gained by making a parity-folding matrix denser, especially when considering trails over many rounds.

For circulant parity-folding matrices, there exist interesting patterns after one selects which columns should be affected given a single isolated active cell. For example, consider the case where $z(x) = x + x^2 + x^3 + x^4$. Then a single active cell leads to four affected columns. However, with two active cells in adjacent columns, there would be some destructive interference in the θ-effect leading to only two affected columns. In general, something similar can happen with other parity-folding matrices and multiple odd columns. It is worth investigating what is an optimal choice.

As another example, it would be interesting to know whether there exist better transposition layers than a combination of row permutations and rotations. Or, to learn what is the optimal ratio between ℓ, m, and n.

For Mixifer, we considered truncated trails and while we touched upon the subject of trail clustering, more can be done. It would be interesting to investigate this behaviour for more rounds, both for AES and for MDS-based ciphers in general, as for Mixifer. We consider further research on clustering of trails to be future work.

We are looking forward to see any future work on the subject of column-parity mixers.

PART II

Optimized Implementations

Chapter 6

ARM Cortex-M

The first chapter of Part II describes how (cryptographic) software, and in particular AES, can be optimized for the ARM Cortex-M line of embedded microprocessors. We also provide a tool to assist with instruction scheduling and register allocation. The original publication claimed that the masked implementation presented there is secure against first-order power analysis [SS16c]. This claim has been removed as I no longer believe that it holds. Instead, the implementation only provides first-order security in the probing model [ISW03]. The implementation is left in as it serves as a baseline for Chapter 9. For most implementations, this chapter presents slightly better results than what was in the original publication.

6.1 Introduction

AES was published as Rijndael in 1998 and standardized in FIPS PUB 197 in 2001. Highly optimized implementations have been written for most common architectures, ranging from 8-bit AVR microcontrollers to x86-64 and NVIDIA GPUs. See, for example, [BS08; KS09; OBSC10]. Implementing optimized AES on any of these architectures essentially requires to start from scratch to find out which implementation approach is going to be the most efficient. The past decades have seen a large shift toward ARM architectures and while we have seen efficient AES implementations for high-end processors used in modern smartphones [BS12] and for older microprocessors used in smart cards [ABM04; BBF+03], there is little to choose from for modern low-end embedded devices and Internet-of-Things applications.

Sometimes an embedded device contains a coprocessor that can perform AES encryption in hardware, but such a coprocessor is not always available. It makes a device more expensive and it can increase the power consumption of a device. Simply compiling an existing implementation written in, for example, the C programming language, is unlikely to produce optimal performance. Even worse, embedded systems are typical targets for timing attacks, power-analysis attacks, and other forms of side-channel attacks, so software for those devices typically needs to include adequate protection against such attacks.

We fill some of these gaps by providing highly-optimized software implementations of AES for two of the most popular modern microprocessors for constrained embedded devices, the ARM Cortex-M3 and the Cortex-M4. Our implementations of AES-$\{128, 192, 256\}$-CTR are more than twice as fast as previous implementations. We also provide a single-block AES-128 implementation, a constant-time AES-128-CTR implementation and a masked implementation with two shares. All of them are the fastest of their kind. They are put into the public domain and available at `https://github.com/Ko-/aes-armcortexm`.

The results of this chapter are not only interesting for *stand-alone* AES encryption. In the CAESAR competition for authenticated encryption schemes[1], 14 out of the 29 second-round candidates that were remaining at the time of the original publication [SS16c], are based on AES or the AES round function. Our implementations will be helpful to speed up those candidates on embedded ARM microprocessors.

Organization of the chapter. In Section 6.2, we first discuss AES and give an outline of the different implementation approaches that have been used so far. We also provide an overview of the target architecture and what features we can benefit from when optimizing software for speed. Section 6.3 then discusses our fastest AES implementations, based on the T-tables approach, while Section 6.4 considers our constant-time bitsliced implementation. We report performance benchmarks and provide a comparison to related work at the end of Sections 6.3 and 6.4.

---

[1] `https://competitions.cr.yp.to/caesar.html`

6.2 Preliminaries

6.2.1 Implementing AES

AES [DR02] is an iterated block cipher that operates on 128-bit blocks. Key sizes of 128, 192, and 256 bits are supported. Depending on the key size, the cipher has 10, 12, or 14 rounds, respectively. The nonlinear substitution layer consists of the SubBytes step, where an 8-bit S-box is applied to each byte of the state. The linear layer consists of ShiftRows and MixColumns, to provide diffusion. In the beginning, between all rounds, and at the end, the AddRoundKey step `XOR`s the state with round keys that are derived from the main key during a key schedule. MixColumns is omitted in the final round. In software, there are four main implementation approaches:

- **Traditional.** All steps are implemented *as is*; typically SubBytes is implemented through a 256-byte lookup table.

- **T-tables.** SubBytes, ShiftRows, and MixColumns are combined in 4 1024-byte lookup tables. Each AES round then consists of 16 masks, 16 loads from the lookup tables and 4 loads from the round keys, and 16 `XOR`s. This leads to very efficient implementations on platforms with a word size of at least 32 bits. At the cost of extra rotations, only 1 lookup table is required. This strategy was already suggested in the original Rijndael proposal [DR99]. Our fastest implementations in Section 6.3 are based on this approach.

- **Vector permute.** The disadvantage of the T-tables approach is that key- and data-dependent lookups open the door for timing attacks on architectures with caches. See, for example, [Ber05a; OST06; TOS10]. Another approach to implementing AES, that avoids such data-dependent lookups, uses vector-permute instructions [Ham09]. However, such instructions are unavailable on our target platform, which is why we do not go into more detail on this strategy.

- **Bitslicing.** Another approach that does not require lookup tables is bitslicing, originally introduced for DES by Biham [Bih97]. The core idea is that data is split over multiple registers, but that other blocks are used to fill the registers. Multiple blocks can then be processed in parallel in a single-instruction-multiple-data (SIMD) fashion. This approach is especially beneficial for architectures with large registers. For AES, the 128-bit state is usually sliced over 8 registers, as this allows for an efficient linear layer. Various papers describe bitsliced implementations of AES on Intel processors [Kön08; Mat06; MN07]. At the time of the original publication [SS16c], the implementation by Käsper and Schwabe from 2009 held the speed record [KS09], but meanwhile this has been further improved by Park and Lee [PL18]. Our implementations in Sections 6.4 and 6.5 also use bitslicing.

6.2.2 ARM Cortex-M

The Cortex-M is a family of 32-bit processors by ARM meant for use in embedded microcontrollers. They are designed to be cheap and to be energy efficient, while still powerful enough to offer adequate performance in applications such as automotive systems, medical instruments, the Internet of Things, or other consumer products. As of 2015, over 10 billion of these processors have been shipped [ARM15].

The Cortex-M3 was announced in 2004, while the Cortex-M4 is from 2010. Both microprocessors have 16 32-bit registers, of which three are reserved for program counter, stack pointer, and link register. The link pointer can be pushed to the stack to free another register. Both microprocessors support the ARMv7-M architecture and the Thumb-2 technology, but the Cortex-M4 supports additional instructions for digital signal processing, i. e., the ARMv7E-M architecture. However, we do not use these extensions.

Bitwise and arithmetic instructions take one cycle on these architectures, except for divisions or writes to the program counter. Branches, loads, and stores may take more cycles, which is why they can easily bottleneck the performance. A distinguishing feature of the ARM architecture is the availability of barrel-shifting registers. This means that we can do arithmetic on rotated or shifted registers, without any additional cost for the rotation or shift.

We used the STM32L100C and STM32F407VG development boards. The first comes with 256 KB of flash memory, 16 KB of RAM, and 4 KB of EEPROM. It can run a Cortex-M3 core at up to 32 MHz. The second is more powerful and has a 168 MHz Cortex-M4 core, 1024 KB of flash memory, 192 KB of RAM, and a true-random-number generator.

6.2.3 Accelerating Memory Access

Memory access can be expensive in terms of CPU cycles. Additionally, there are a lot of ways to introduce penalty cycles. Carefully optimized software therefore avoids as many potential delays as possible. Here we list a number of generic strategies related to memory access to reduce the cycle count of programs running on the Cortex-M3 and M4. A siginifcant portion of our speedups of AES stems from a combination of these strategies.

Flash. The instructions and tables are typically stored in flash memory. Accessing flash can introduce a number of wait states, depending on the relative clock frequency of the microprocessor and the memory chip. For our development boards, the STM32L100C and STM32F407VG, STMicroelectronics describes in its documentation when it is possible to have zero wait states [STM20, p. 59, tbl. 13][STM19, p. 80, tbl. 10]. For example, on the STM32L100C, the CPU clock can only run at 16 MHz for a supply voltage of 3.3 V. To be able to compare the performance of implementations across different devices or boards, it is important to be in this scenario.

RAM. Something similar holds for accessing RAM, where the stack is stored. On the STM32F407VG, four different regions of RAM are available: `SRAM1`, `SRAM2`, `SRAM3`, and `CCM`. In our case it turned out to be fastest to use only `SRAM1`.

Alignment. The Cortex-M3 and M4 support Thumb-2 technology, which means that 16-bit and 32-bit encodings of instructions can freely be mixed. However, consider the case that a 16-bit instruction starts at a word-aligned address, followed by one or more 32-bit instructions. The 32-bit instructions are then no longer word-aligned, which may cause penalty cycles for the instruction fetcher, which fetches multiple instructions at a time. In this case, forcing the use of a 32-bit encoding for the first instruction by adding a `.w` suffix can improve the instruction alignment and reduce the cycle count. Our implementations take this into consideration. Penalty cycles may also be introduced when branching to addresses that are not word-aligned, when loading from memory at addresses that are not word-aligned or when not loading full words from memory. Implementers need to take care of the alignment themselves. Our implementations carefully avoid these penalty cycles.

Pipelining loads. Most `str` instructions take 1 cycle, because of the availability of a write buffer, but `ldr` instructions generally take at least 2 cycles. However, n `ldr` instructions can be pipelined together to be executed in $n + 1$ cycles if there are no address dependencies and the program counter remains untouched. An instruction such as `ldm` pipelines all of its loads together, but when it is followed by an `ldr`, those will not be pipelined together. For our implementations, we pipeline as many loads as possible.

Caches and prefetch buffers. The Cortex-M3 and M4 by themselves do not have any caches. However, caches can be added in embedded devices or development boards to boost the performance. For example, the STM32F407VG contains 64 128-bit lines of instruction cache memory and 8 128-bit lines of data cache memory [STM19, p. 73]. It also contains an instruction prefetch buffer to reduce the experienced number of wait states when a microprocessor running at a high clock frequency accesses flash memory to fetch 128 bits of instructions [STM19, p. 82]. The STM32L100C supports a similar prefetch buffer when 64-bit flash access is enabled [STM20, p. 59].

Data location. When one wants to read data that is stored in flash memory, one first needs to load the address of the data block before one can load the data itself. However, when data is located within 4096 bytes of the value of the program counter, the first load instruction can be replaced by an `adr` pseudo-instruction that is really an addition or subtraction of the program counter, which may save one cycle, depending on whether the load could be pipelined. It is therefore useful to store data close to where the data is being used.

6.3 Making AES Fast

Ever since Rijndael was standardized as AES, a lot of effort has been put into fast and secure software implementations for a large range of platforms and architectures. Numerous optimization tricks have been suggested to improve the performance. For T-table-based implementations, the majority is summarized in [BS08]. In this section we discuss which strategies are useful to apply on the Cortex-M3 and M4.

Using the T-table-based approach, AES-128-CTR can typically be implemented in 720 instructions: 208 loads, 4 stores, 160 shifts, 176 masks, 168 `XOR`s and 4 others [BS08]. Thanks to ARM's barrel-shifting registers, we can do combined shifts and masks, saving 160 instructions. [BS08] also mentions *scaled-index loads* and *second-byte instructions*. A scaled-index load is the option to shift the offset of a load instruction for free, while a second-byte instruction allows for extracting the second byte of a register in one instruction. Both features are supported by our architecture, but as all shifts are already fully subsumed, these optimizations no longer yield any additional advantage.

Byte loads and *two-byte loads* could save another 8 instructions by not requiring an additional mask, but loads that are not word-aligned cause a penalty cycle, so for speed these optimizations are of little use. Other potential optimization strategies, such as combining *masks and inserts* or *loads and `XOR`s*, are not possible in a single instruction on these platforms. Being able to do *byte extraction via loads* allows to exchange arithmetic instructions for load instructions, but loads are either as fast or slower, so this strategy gives no advantage either.

With *round-key recomputation*, only one out of four round-key words is stored for all rounds except the first. During encryption, the other parts of the round keys can be recomputed on the fly, exchanging 30 loads for 30 `XOR`s. However, in our case the loads can be fully pipelined and the round keys from the previous round would not fit into registers anymore, so this would also not reduce the total number of cycles. *Round-key caching*, where all round keys are kept in registers when encrypting multiple blocks, would require even more registers. Another technique called *padded registers* exists, where a 32-bit value is stored in a 64-bit register in such a way that combining shifts and masks can be done a bit more cleverly. However, our registers are too small to use anything like this.

However, *counter-mode caching* helps to save another 81 instructions in the main loop. In counter mode, for 256 consecutive blocks, only 1 byte of the input changes. This means that through the first and second AES round, computations that do not depend on this one byte can be cached and reused. Starting from the third round, everything will depend on all input bytes. While there is some additional overhead involved in storing and retrieving the cached values, this trick already leads to a speedup when only 2 blocks are processed.

6.3.1 Our Implementations

Our implementations of AES-128 encryption, AES-128-CTR, AES-192-CTR, and AES-256-CTR use one 1024-byte lookup table. The extra rotates that this would normally cause come for free thanks to ARM's barrel shifting registers. Using four tables would save another 40 1-cycle instructions in the key schedule, and 16 1-cycle instructions in the final round for encryption, but as there is typically little memory available on microcontrollers and the improvement in speed is only marginal, we decided that this trade-off was not worth it. AES-128 decryption needs two 1024-byte lookup tables. On the other hand, the 16 mask instructions in the final round are no longer required.

Key expansion is performed separately, as the round keys can be reused for multiple blocks. In our implementation of counter mode, there is a 32-bit counter and a 96-bit nonce. The reason is that then we do not have to deal with a carry from the counter and a conditional `add` for the second counter word, which gives another small speedup. We consider a 32-bit counter, providing a maximum stream length of $2^{32} \cdot 16 = 68\,719\,476\,736$ bytes, to be large enough in a typical microcontroller environment.

The performance of our speed-optimized implementations is summarized in Table 6.1. All results are averages over 10000 runs with random keys, inputs, and, if applicable, nonces. For encryption in counter mode, the number of cycles reflects the average number of cycles per block when processing 256 blocks, or 4096 bytes. Loops are fully unrolled, so the code size can be reduced drastically with only a small performance penalty. Note that data in ROM is typically shared by key expansion and encryption/decryption, so it has to be in memory only once. Under RAM usage, I/O refers to the amount of RAM that is required to store the input and output for the functions, e. g., $192 + 2n$ means that we require 4 bytes for the counter, 12 for the nonce, 176 for all round keys, n for our n-byte input, and n for the n-byte output. Again, I/O data is typically shared by key expansion and encryption/decryption and the same stack space can be reused for the encryption/decryption function call. It turns out that the same code usually runs in slightly more cycles on the Cortex-M3, which we cannot really explain.

Table 6.1: Performance of unprotected AES on Cortex-M.

Algorithm	Speed (cycles)		ROM (bytes)		RAM (bytes)	
	M3	M4	Code	Data	I/O	Stack
AES-128 key exp. enc.	249.8	244.9	742	1024	176	32
AES-128 key exp. dec.	1031.6	971.2	2978	2048	176	176
AES-128 one block enc.	637.5	634.7	1970	1024	$176 + 2n$	40
AES-128 one block dec.	640.5	636.3	1974	2048	$176 + 2n$	40
AES-128-CTR	531.8	527.9	2128	1024	$192 + 2n$	68
AES-192 key exp.	232.9	232.2	682	1024	208	32
AES-192-CTR	651.0	644.0	2512	1024	$224 + 2n$	68
AES-256 key exp.	310.8	309.9	958	1024	240	28
AES-256-CTR	767.0	760.7	2896	1024	$256 + 2n$	68

6.3.2 Comparison to Previous Implementations

There are few publicly available AES implementations optimized for the Cortex-M3 and M4:

- In the SharkSSL crypto library v2.4, a speed of 1066.7 cycles per block is claimed for AES-128-ECB on the Cortex-M3.[2] CTR mode is unavailable.
- A company called Cryptovia sells an implementation that does AES-128 on a single block in 1463 cycles, also on the Cortex-M3.[3]

---

[2] `https://realtimelogic.com/products/sharkssl/Cortex-M3/`

[3] `http://cryptovia.com/cryptographic-libraries-for-arm-cpu/`

- The latest version of mbed TLS,[4] formerly known as PolarSSL, contains a table-based AES-128-CTR implementation that takes 1247.4 cycles per block on the M3, while AES-128 key expansion takes 41 545 cycles.[5]
- NXP hosts the AN11241 AES library,[6] but its implementation is very slow. AES-128-ECB runs in 4179.1 cycles per block on the M3, while the AES-128 key expansion takes 1089 cycles.[5]
- The fastest implementation currently listed by the FELICS benchmarking framework [DCK+15] encrypts a single block with AES-128 in 1816 cycles on a Cortex-M3. The fastest key scheduling takes 724 cycles.[7]

We therefore claim that our CTR-mode implementations are about twice as fast as previous implementations. We also require fewer cycles than optimized implementations for older yet similar ARM architectures [ABM04], even though in [ABM04] heavy use is made of the fact that the full lookup tables fit in the data cache on a StrongARM-1110, which does not hold for our platforms.

6.3.3 Benchmarking with FELICS

The FELICS framework [DCK+15] has been proposed as an open system to benchmark the performance of implementations of lightweight cryptographic systems on three different microprocessors, one of them being the ARM Cortex-M3. Cycle counts and memory usage are measured for three usage scenarios. Scenario 0 deals with single-block encryption, where the round keys are stored in RAM. In scenario 1, 128 bytes are encrypted in CBC mode. In scenario 2, 128 bits are encrypted in CTR mode.

---

[4] `https://tls.mbed.org/`

[5] We used `gcc -O3 -funroll-loops -fno-schedule-insns` with GCC 6.1.1 for these benchmarks, the best set of compiler flags we could find, based on all sets that are tried in the SUPERCOP benchmarking framework.

[6] `https://www.nxp.com/docs/en/application-note/AN11241.zip`

[7] `AES_128_128_V06` in scenario 0 with `-Os` and with `-O3`, respectively.

This choice of scenarios means that our implementation needs to be adapted to fit in the framework. In particular, counter-mode caching can no longer be used and needs to be removed, which impacts the performance. Furthermore, the decryption algorithm and decryption key expansion are now required as well in scenarios 0 and 1. What is even more significant is that the FELICS framework does not set the number of wait states, which means that a load from memory will cost more than 2 cycles at the maximum clock frequency, greatly inflating the total cycle counts. The reported cycle counts are therefore biased toward implementations with fewer load instructions.

The framework reports 1641 cycles for our encryption in scenario 0 and 578 cycles for our key schedule. Although this is still faster than previous results, the margin is smaller. This also holds for scenarios 1 and 2.

6.4 Protecting against Timing Attacks

While the availability of caches allows for speedups on platforms with relatively slow memory, it also makes table-based AES implementations vulnerable to cache-timing attacks [Ber05a; Koc96]. On our target platforms, it is possible to simply disable the caches when performing cryptographic operations, but it is useful to have a constant-time implementation that does not depend on this ability. Moreover, this implementation serves as a step toward the masked implementation. A popular technique for writing a constant-time AES implementation that is still reasonably fast, is applying bitslicing.

Bitslicing is often explained as a technique where every bit of the state is stored in a separate register, such that we can do operations on the bits independently and such that we can process 32 blocks in parallel on 32-bit machines. However, in the case of AES this is not the fastest way to bitslice, as most operations are byte-oriented. Full bitslicing would also increase the amount of registers needed to store the state by a factor of 32. There are very few architectures that have enough registers to keep the bitsliced state in registers, so there would be a lot of overhead in storing and loading data to other types of memory.

Könighofer suggested in [Kön08] to *byteslice* and to process 4 blocks in parallel on an architecture with 64-bit registers. Note that this only benefits a mode of operation that allows for parallel encryption, such as CTR mode. Käsper and Schwabe were able to process 8 blocks in parallel using 128-bit registers [KS09]. Unfortunately, the Cortex-M3 and M4 only have 32-bit registers, so we can only process 2 blocks in parallel while still retaining an efficient implementation of the linear layer.

6.4.1 Our Implementation

After key expansion, the round keys are stored in their bitsliced representation. To transform to bitsliced representation, we require 12 SWAPMOVE operations [MPC00].

```
SWAPMOVE(a,b,n,n) {
  t = ((a ≫ n) ⊕ b) & m
  b = b ⊕ t
  a = a ⊕ (t ≪ n)
}
```

Due to ARM's barrel shifter, we can implement SWAPMOVE in just 4 1-cycle instructions, which gives a transformation overhead of 48 cycles.

```
eor t, b, a, lsl #n
and t, m
eor b, t
eor a, a, t, lsr #n
```

During encryption, the AES state is first transformed to bitsliced representation. AddRoundKey is then again just a matter of `XOR`ing the bitsliced round keys with the bitsliced state.

For SubBytes, a lot of research has been done on an efficient hardware implementation of the AES S-box [Can05]. These results are also very useful for bitsliced software implementations. Boyar and Peralta found a circuit with only 115 gates [BP10], which was later improved to 113: 32 `AND` gates, 77 `XOR` gates, and 4 `XNOR` gates. At the time of the original publication [SS16c], this was the smallest known implementation, which is why we used it as a basis for our implementation. With only 14 available registers, it is impossible to implement the S-box directly in 113 instructions. We need more instructions to deal with storing values on the stack or with recomputation of values. We wrote an ad-hoc combined instruction scheduler and register allocator that is tailored to our microprocessors. Although recently even smaller circuits have been found [ME19; RTA18], we have examined these and they do not lead to faster results on Cortex-M microcontrollers, because of the use of multiplexer gates and the requirement for more loads and stores.

Scheduling. Both instruction scheduling and register allocation are hard problems, as is the combined problem. Compilers usually implement a graph coloring algorithm and/or linear-scan allocation. They aim to schedule well on average, but do not necessarily generate the most efficient assembly for a specific part of code.

Existing compilers do not provide a lot of options to play with different scheduling and allocation strategies, which is why we decided to write an ARM-specific instruction scheduler and register allocator. This allows us to focus on ARM's three-operand instructions and to try several approaches. We aim to minimize the number of loads and stores and the usage of the stack. We first reschedule instructions to reduce the size of the active data set, by pushing instructions down based on their left-hand side and by pushing instructions up based on their right-hand side. Then we allocate registers in a greedy fashion, where we insert loads and stores when necessary and try to leave the output in registers. A more thorough overview of the tool is provided in [Sto16a], including a comparison against the compilers GCC, Clang, and the ARM Compiler.

Our tool is nondeterministic because of hash randomization in Python, so we try several scheduling strategies multiple times and only use the best result. With our scheduler we are able to compute the AES S-box in 145 instructions: the 113 original operations, 16 loads and 16 stores. It is unknown whether this is optimal.

ShiftRows on a bitsliced state can be computed very efficiently on modern Intel CPUs using 8 SSSE3 byte-shuffling instructions [KS09]. However, something like this is unavailable on the Cortex-M3 and M4. We use the `ubfx` and `uxtb` bitfield instructions, together with `eor` on shifted registers, to compute ShiftRows in $8 \cdot 13 = 104$ 1-cycle instructions. The code below performs ShiftRows on `r9`, while `r12` and `r5` are used as temporary registers.

```
uxtb r12, r9
ubfx r5, r9, #14, #2
eor r12, r12, r5, lsl #8
ubfx r5, r9, #8, #6
eor r12, r12, r5, lsl #10
ubfx r5, r9, #20, #4
eor r12, r12, r5, lsl #16
ubfx r5, r9, #16, #4
eor r12, r12, r5, lsl #20
ubfx r5, r9, #26, #6
eor r12, r12, r5, lsl #24
ubfx r5, r9, #24, #2
eor r9, r12, r5, lsl #30
```

In contrast, the barrel shifters allow us to compute MixColumns in just 27 `eor` instructions on shifted registers, which is even more efficient than in [KS09].

To update the counter for the next blocks, one can either store the bitsliced representation and operate on this, or one can use the original representation and transform this to bitsliced representation every two blocks. While the first may appear to be faster, we implemented both and it turned out that the latter is in fact more efficient. This is due to overhead caused by the limited way in which you can do conditional execution with `IT`-blocks on these microprocessors.

Table 6.2 contains performance benchmarks of our implementation. Again, speed is measured as the average number of cycles per block when encrypting 256 consecutive blocks, which explains the decimal for the encryption. The amount of cycles is exactly equal for all 10 000 combinations of random nonces, keys, and inputs that we tried. We see a slowdown of roughly a factor 3 compared to our previous implementation. Note, however, that when one can disable the caches during the AES execution or when caches are not available at all, our previous faster implementations are also constant-time and should be favored. We verified that after disabling caches, the cycle counts are exactly equal for random combinations of inputs and keys. There is little related work that would make a fair comparison.

Table 6.2: Performance of constant-time AES on Cortex-M.

Algorithm	Speed (cycles)		ROM (bytes)		RAM (bytes)	
	M3	M4	Code	Data	I/O	Stack
AES-128 bitsliced key expansion	1024.8	1021.9	3434	1036	368	184
AES-128-CTR bitsliced constant-time	1618.6	1616.6	11 806	12	$368 + 2n$	104

6.5 Protecting against Side-Channel Attacks

Microcontrollers are typical targets for side-channel attacks such as differential power analysis or differential electromagnetic analysis. A well-known countermeasure against first-order side-channel attacks that is used in practice is Boolean masking, where a secret intermediate value a is split into two statistically independent shares, i.e., r_a and $\bar{a} = (a \oplus r_a)$, where r_a is called a random mask. Linear operations can be computed on both shares independently. After a linear operation, the shares can be `XOR`ed together to unmask the result. Nonlinear operations are more difficult to mask securely. Trichina suggested the following provably secure method to mask $a \cdot b$ [Tri03], where $\bar{a} = (a \oplus r_a)$, $\bar{b} = (b \oplus r_b)$, and r_a, r_b, r are random masks:

$$((\bar{a} \cdot \bar{b}) \oplus ((r_a \cdot \bar{b}) \oplus ((r_a \cdot r_b) \oplus r))) \oplus (r_b \cdot \bar{a}).$$

This means that every `AND` operation requires 4 `AND` operations, 4 `XOR` operations, and 1 load (of r) to mask.

We added first-order Boolean masking using Trichina gates to our constant-time bitsliced implementations to find out how much this additional security would cost on common microprocessors.

6.5.1 Our Implementation

To generate the masks, we need a source of randomness. The STM32F407VG contains a random number generator (RNG) that guarantees a new 32-bit random word every 40 periods of the RNG clock. In the case of AES, 8 random words are required to mask the input, as two blocks are processed in parallel, and 320 random words are required for a single encryption, as SubBytes contains 32 `AND` operations and is executed in all 10 rounds. While interleaving randomness generation and executing instructions can decrease the waiting time, the performance of the implementation will greatly depend on the performance of the RNG and the relative clock frequency between the core and the RNG.

All other operations are linear, so at least a factor of 2 slowdown can be expected there. However, because the size of the active data set doubles and will not fit in 14 registers anymore, a lot of overhead is created by additional loads and stores. Our scheduler manages to generate a securely masked bitsliced SubBytes implementation in $2 \cdot 83 + 4 \cdot 32 = 294$ `XOR`s, $4 \cdot 32 = 128$ `AND`s, 99 stores and 167 loads, that are pipelined as much as possible. Once more, the speed is measured as the average number of cycles per block when encrypting 256 consecutive blocks.

Table 6.3: Performance of masked constant-time AES on Cortex-M.

Algorithm	Speed (cycles)		ROM (bytes)		RAM (bytes)	
	M3	M4	Code	Data	I/O	Stack
AES-128-CTR masked constant-time	N/A	8727.6	39 224	12	$368 + 2n$	1584

The performance of the final implementation is summarized in Table 6.3. Note that of these 8727.6 cycles per block, 3439.5 are spent on generating random words and pushing them to the stack, while all the rest takes 5288.1 cycles per block. A faster RNG could significantly boost the total speed. Of the 1584 bytes on the stack, 1312 are taken by the 328 random words.

6.5.2 Comparison to Previous Implementations

Balasch, Gierlichs, Reparaz, and Verbauwhede [BGRV15] showed at CHES 2015 that adding first-order Boolean masking with Trichina gates slows the implementation down by roughly a factor of 5 on the Cortex-A8. On the Cortex-M4, we see something similar compared to the unmasked bitsliced implementation, with a factor 6.0, although a faster RNG could reduce this to almost a factor of 3.5. Furthermore, we require less randomness because we based ourselves on the 113-gate SubBytes implementation.

Goudarzi and Rivain [GR17] investigated the performance of different approaches to higher-order masking based on the ISW masking scheme [ISW03] by implementing masked versions of AES and PRESENT on the ARM7TDMI-S microprocessor, a somewhat older architecture from 2001 that is still widely deployed. For first-order masking, their fastest implementation takes 49 329 cycles [GR17, tbl. 16, standard AES with parallel Kim-Hong-Lim S-box, 2 shares], which is a factor 5.6 more than ours, but that comparison is not entirely fair as we do not support higher-order masking. However, instruction timings appear to be similar between the two architectures.

6.6 Conclusion and Outlook

This chapter presented various speed-optimized AES software implementations for multiple use cases, including timing-attack protections, for the ARM Cortex-M3 and M4. All of them are the fastest of their kind. Additionally, we provide an ARM-specific instruction scheduler and register allocator that is of independent interest to optimize other software for these platforms. All software is put into the public domain, which also may benefit the performance of (AES-based) CAESAR candidates on modern embedded microcontrollers.

Chapter 7

RISC-V

This chapter describes how cryptographic software can be optimized for RISC-V, a popular open instruction-set architecture. In particular, the focus lies on AES, ChaCha, Keccak-f, and arbitrary-precision arithmetic on the RV32I microarchitecture. The results are then compared to results on an ARM Cortex-M4. In comparison to the original publication [Sto19], changes are mostly related to formatting and to better reflect the current status of the ongoing RISC-V project.

7.1 Introduction

The RISC-V project started out in 2010 as a research project at the University of California, Berkeley. The goal was to design an open-source reduced instruction set that was free and practical to use by academics and industry. Today, it comprises an association with hundreds of member organizations, including major industry partners such as Google, Qualcomm, and Samsung. The fact that many large companies are joining this efforts indicates that RISC-V might await a bright future. In particular, no longer having to pay any license fees makes it an attractive alternative and a serious competitor to ARM-based microcontrollers.

Together, the association's members developed a specification for the RISC-V instruction-set architecture [RIS19]. RISC-V targets both embedded 32-bit devices and larger 64-bit, and even 128-bit devices. While some parts of the specification are still in development, the most important parts have been frozen such that hardware and software could be implemented. Compilers, debuggers, and software libraries with RISC-V support have been around for several years.[1]

---

[1] `https://riscv.org/software-status`

Boards with fully functional RISC-V SoCs have been commercially available since 2016.[2]

There exist several open-source RISC-V CPU designs designed to be easily extensible. This makes the platform an ideal candidate for software-hardware co-design, as was exemplified by a recent implementation of the hash-based signature scheme XMSS [WJW+18]. The underlying hash function, SHA-256 [NIS15a], was implemented in hardware to increase the performance of the full signature scheme. However, it is not always possible to "simply" add a hardware coprocessor of a required cryptographic primitive. In practice, one may have to deal with whatever hardware is available or a developer might lack the capabilities to modify a hardware implementation. More importantly, adding a coprocessor to an ASIC will most likely increase the production cost of that chip. In order to make any trade-off decision for software-hardware co-design meaningful, some numbers need to exist to have an idea about the cost of software implementations. To the best of our knowledge, we are the first to provide such numbers for cryptographic primitives.

We explain how AES-128, ChaCha20, and Keccak-$f[1600]$ can be implemented efficiently on RISC-V and we optimize 32-bit RISC-V assembly implementations. We also study the speed of arbitrary-precision addition, schoolbook multiplication, and Karatsuba multiplication for unique and redundant or reduced-radix integer representations. We then draw a parallel to the ARM Cortex-M line of microprocessors and we show how architectural features such as the availability of native rotation instructions, a carry flag, and the number of available registers impact the performance of these primitives. We continue by estimating what the performance would be if a RISC-V core were to be extended with these features.

In Section 7.2 we first explain details about the RISC-V instruction set and our benchmarking platform. Sections 7.3 through 7.5 cover implementation strategies that are specific to AES, ChaCha, and Keccak, respectively. Arbitrary-

---

[2] `https://www.sifive.com/boards/hifive1`

precision integer arithmetic is discussed in Section 7.6. Finally, in Section 7.7 we compare the relative performance of cryptographic primitives to that on the ARM Cortex-M4 and estimate what the performance would be with RISC-V extensions for several architectural features.

Our software implementations are open-source and placed into the public domain. They are available at `https://github.com/Ko-/riscvcrypto`.

7.2 The RISC-V Architecture

The specification of the RISC-V instruction-set architecture (ISA) is split into a user-level ISA and a privileged ISA. The privileged ISA specifies instructions and registers that are useful when creating, for example, operating systems, but for our purpose we only need to consider the user-level ISA. The user-level ISA is divided in a base ISA and in several standardized extensions that are discussed in Section 7.2.2. At the time of writing, the base ISAs for 32-bit and 64-bit machines, called RV32I and RV64I respectively, have been frozen at version 2.1. A base ISA for 128-bit machines (RV128I) and a smaller 32-bit variant with fewer registers (RV32E) still have draft status. In this work we focus on the 32-bit RV32I instruction set.

7.2.1 The RV32I Base Instruction Set

RV32I specifies 32 32-bit registers named `x0` to `x31`. However, not all of them can be used freely. The registers have aliases that make their purpose more clear. For example, `x0` is also known as `zero`: writes to it are ignored and it always reads as the value 0. The others are: `ra` (return address, `x1`), `sp` (stack pointer, `x2`), `gp` (global pointer, `x3`), `tp` (thread pointer, `x4`), `a0`-`a7` (function arguments and return value), `s0`-`s11` (saved registers), and `t0`-`t6` (temporary registers). That means that 27 registers can be used without complications and maybe a few more depending on the environment. Only `sp` and `s0`-`s11` are callee-saved.

As a true RISC, the number of available instructions is fairly limited. We therefore include a concise but complete overview in this section. All instructions are described in more detail in the official specification [RIS19].

Arithmetic and bitwise instructions have three register operands, or two register operands and a sign-extended 12-bit immediate, denoted by the `I` suffix. The following self-explanatory instructions are available: `ADD`, `ADDI`, `SUB`, `AND`, `ANDI`, `OR`, `ORI`, `XOR`, and `XORI`. There is no `SUBI`, because that is just an `ADDI` with a negative immediate. Similarly, there is no real `NOT` instruction, because it can be implemented with `XORI` and -1 as immediate. `NOT` is recognized as a pseudo-instruction by assemblers.

Regarding shifts, the following instructions exist: `SLL`, `SLLI`, `SRL`, `SRLI`, `SRA`, and `SRAI`. The naming convention that is used here is `S`hift (Left or `R`ight) (`L`ogical or `A`rithmetic) (`I`mmediate). Note that the base ISA does not specify a rotation instruction.

To load a value from memory, `LW`, `LH`, `LHU`, `LB`, and `LBU` can be used. The `W` stands for word (32 bits), the `H` for half-word (16 bits), and the `B` for byte (8 bits). With `LH` and `LB`, the value is assumed to be signed and will therefore be sign-extended to a 32-bit register. `LHU` and `LBU` are their unsigned counterparts that perform zero-extension instead of sign-extension. To store a register value to memory, one can use `SW`, `SH`, and `SB`. For all load and store instructions, the base address needs to be in a register. An immediate offset can be specified in the instruction. For example, `LW a1, 4(a0)` loads a word from `a0` + 4 in `a1`. It is not possible to specify the offset in a register or to automatically increment/decrement the address.

The `JAL` and `JALR` instructions specify unconditional jumps. The target address can be specified relative to the program counter (`JAL`) or as an absolute address in a register (`JALR`). On the other hand, `BEQ`, `BNE`, `BLT`, `BLTU`, `BGE`, and `BGEU` denote conditional jumps based on a comparison. Their first two operands are registers of which the values are compared. The `U` suffix denotes that the operands are interpreted as unsigned values for the comparison. The third operand specifies the destination address relative to the program counter.

It is also possible to compare without branching. The `SLT`, `SLTU`, `SLTI`, and `SLTIU` instructions set a destination register to one if the second operand (a register) is less than (signed or unsigned) the third operand (either a register or an immediate). Otherwise, the destination register is set to zero.

The `LUI` (load upper immediate) and `AUIPC` (add upper immediate to program counter) instructions can be used to set values larger than 12 bits in a register.

Finally, for the sake of completeness, there are specialized instructions to deal with synchronization (`FENCE`), to call an operating system (`ECALL`), to signal debuggers (`EBREAK`) and to hint the microarchitecture (`HINT`). We will not use them.

7.2.2 Standardized Extensions

A RISC-V core has to implement a base ISA, and optionally it can implement one or several standardized extensions to the instruction set. Most extensions are denoted by a single letter. The extensions with a frozen specification are *M* (with instructions for integer multiplication/division), *A* (atomic instructions), *F* (single-precision floating point), *D* (double-precision floating point), *Q* (quad-precision floating point), *C* (compressed instructions), *Zicsr* (control and status registers), *Zifencei* (instruction-fetch fence), and *Ztso* (total store ordering).

Other extensions, such as those for bit manipulation, vector instructions, and user-level interrupts still have draft status. To the best of our knowledge the extensions in draft status have not yet been implemented by any commercially available core.[3]

---

[3] `https://riscv.org/risc-v-cores`

7.2.3 Benchmarking Platform

We use a HiFive1 development board as our benchmarking platform, as they are relatively easily available. This contains the FE310-G000 SoC [SiF17] with an E31 core [SiF18]. The core implements the RV32IMAC instruction set, i. e., the RV32I base ISA with the extensions for multiplication/division, atomic instructions, and compressed instructions. Of these, only the M extension is relevant to us.

The RISC-V specification does not specify how long instructions take to execute or what kinds of memory are available. This is left open to the hardware core implementer. Benchmarks across different RISC-V cores therefore need to be compared with caution. To provide more insight, we briefly describe some characteristics of this particular RISC-V core.

The E31 is designed as a 5-stage single-issue in-order pipelined CPU that runs at 320+ MHz, although the PLL clock generator has an output of at most 384 MHz. The core has support for up to 64 KiB of DTIM memory that is used as RAM, but the HiFive1 only has 16 KiB. Outside of the core, there is another 16 MB of QSPI flash memory. To accelerate instruction fetches from the flash memory, the E31 comes with 16 KiB of 2-way instruction cache.

Most instructions have a result latency of a single cycle. There are a few exceptions. For example, word-loads have a result latency of 2 cycles with a cache hit. With a cache miss, it highly depends on the relative clock frequency of the flash controller compared to the core. Half-word-loads and byte-loads have a result latency of 3 cycles in the event of a cache hit. Misaligned DTIM accesses are not allowed and result in a trap signal.

The E31 has an elaborate branch predictor, consisting of a branch target buffer, a branch history buffer, and a return address stack. Correctly predicted branches should suffer no penalty, while wrong guesses receive a penalty of 3 cycles.

The RISC-V specification describes a 64-bit increasing cycle counter that is accessible through two CSR registers. This can be used for accurate benchmarking of code. We aim to unroll the code as much as possible as long as the code still fits in the instruction cache. Tables and constants are stored in the DTIM memory. This way, we manage to get very consistent measurements. Occasionally, a measurement ends up taking much longer than expected. These outliers are ignored.

7.3 AES

32-bit software implementations of AES usually fall into two categories, depending on whether it it safe to use table lookups or not. The fastest encryption implementations *for a single block* use the idea that the various steps of the round function can be combined in large lookup tables, usually called T-tables [DR02]. However, this type of implementation is known to be vulnerable to cache-based timing attacks [Ber05a; OST06]. A CPU cache can leak information about which memory address has been accessed during a computation. When this memory address depends on a secret intermediate value as is the case with the T-table approach, it can be used to extract secret information.

When *multiple blocks* can be processed in parallel (e. g., in CTR or GCM mode) and the CPU registers are large enough to accommodate multiple blocks, bitsliced implementations can be more efficient [Kön08; KS09]. This type of AES implementation has the additional advantage that lookup tables are easily avoidable, allowing a careful implementer to make it resistant against timing attacks.

Our particular benchmarking platform does not have a data cache. We also do not identify other potential sources of timing leakage. Therefore, it *should* be safe to use a table-based AES implementation on this device. However, this might not be the case on other RISC-V platforms. Table-based implementations might also demand an unreasonable amount of memory on small embedded RISC-V-based devices. This is why we treat both implementation categories.

7.3.1 Table-based Implementations

At Indocrypt 2008, Bernstein and Schwabe explained how to optimize table-based AES implementations for a variety of CPU architectures [BS08]. They describe a baseline of 16 shift instructions, 16 mask instructions, 16 load instructions for table lookups, 4 load instructions for round keys, and 16 `XOR` instructions per AES round, plus 16 additional mask instructions in the last round and 4 additional round-key loads and 4 `XOR` instructions for the initial AddRoundKey. This baseline excludes the cost of loading the input into registers, writing the output back to memory, and some overhead such as setting the address of the lookup table in a register and storing callee-save registers on the stack when necessary. They then continue by listing various architecture-dependent optimizations.

On RISC-V, very few of these techniques are possible, which is no surprise given that the instruction set is intentionally kept very simple. The `LBU` byte load instruction allows to save 4 mask instructions in the final round. On the other hand, the baseline count assumes that it is possible to load from an address specified by a base value in one register and an offset in another register. While this holds for many architectures, it is not true for RISC-V. Instead, the full address needs to be explicitly computed each time. This means that we require 16 extra `ADD` instructions per round.

With round-key recomputation, only 14 round-key words have to be stored and loaded instead of 44. This saves 30 `SW` instructions in the key expansion, but more importantly, it allows to swap 30 `LW` instructions for 30 `XOR` instructions at the cost of using 4 extra registers of which their values need to be saved on the stack. We expected this to improve performance for encryption on our platform. However, it turned out that this was not the case so we did not employ this technique.

There is more that can be done with the free registers that are available. Some of the round keys could also be cached in registers such that they do not have to be loaded for every block when encrypting multiple blocks. However, to keep the implementation as versatile as possible, we decided not to do this and to encrypt just a single block. This makes it possible to straightforwardly build any mode around it.

Result. We implemented and optimized the AES-128 key expansion and encryption algorithms. Both use the same 4 KiB lookup table. Key expansion finishes in 340 cycles and requires no stack memory. Encryption of a single 16-byte block is performed in 912 clock cycles. This uses 24 bytes on the stack to store callee-save registers.

7.3.2 Bitsliced Implementations

With bitsliced AES implementations, the internal parallelism in the SubBytes step usually means that the AES state is represented in such a way that a register is made to contain the ith bit of every byte of the state. This means that 8 registers are needed to represent the AES state, but then only 16 bits in the register are used, which is suboptimal. However, when multiple AES blocks can be processed in parallel, they can be stored in the same registers in order to process them simultaneously. Especially when the registers are large, this yields very high throughputs [KS09].

We implement an optimized bitsliced implementation of AES-128 in CTR mode. With 32-bit registers, only 2 blocks can be processed in parallel. The implementation is inspired by an earlier implementation optimized for the ARM Cortex-M4 architecture [SS16c] that formed the basis of Chapter 6.

For the most expensive operation, SubBytes, we use the smallest known circuit by Boyar and Peralta of 113 gates [BP10]. On the Cortex-M4, this could not be implemented directly because there were not enough registers available. With RV32I, carefully rearranging the instructions permits not having to spill any intermediate value to the stack. We can therefore implement SubBytes in exactly 113 single-cycle bitwise instructions.

ShiftRows with a regular state representation uses rotations over the full rows of the AES state that are stored in registers. The equivalent for the bitsliced state representation requires to do rotations within a byte of a register, which is trickier to implement. The RV32I base ISA does not offer convenient instructions to do this or to extract bits from a register. It therefore has to be implemented by simply masking out a group of bits, shifting them to their correct position and inserting them in a result register. This takes 6 `OR` instructions, 7 `AND(I)` instructions and 6 shift instructions per state register. There are 8 state register, so this has to be done 8 times for one AES round.

On the Cortex-M4, MixColumns could be implemented with just 27 `XOR` instructions, heavily using the fact that one operand can be rotated for free. RISC-V, however, does not have a native rotation instruction in the base ISA at all. Therefore the rotation has to be implemented with two shifts and an `OR` instruction. We study the impact of rotation instructions in more detail in Section 7.7.2. In total, our MixColumns implementation uses 27 `XOR` instructions and 16 rotations.

The other parts of the implementation are straightforward or are very similar to the Cortex-M implementation of Chapter 6.

Result. Key expansion and conversion of all round keys to the bitsliced format takes 1239 clock cycles and 16 stack bytes. For benchmarking encryption, we selected a fixed plaintext size of 4096 bytes. This can be encrypted or decrypted with AES-128-CTR in 509 622 cycles, or at 124.4 cycles per byte. 60 stack bytes are used to store callee-save registers and copies of a few other values.

7.4 ChaCha

ChaCha is a family of stream ciphers based on Salsa20 [Ber08a]. It is known for its high speed in software and together with a message authentication code called Poly1305 it is used in TLS and OpenSSH [Ber05b; LCM+16; NL18].

ChaCha starts by loading constants, a 256-bit key, a 96-bit nonce, and a 32-bit counter into a 512-bit state. With RV32I, there are enough registers to keep the full state in registers during the whole computation. ChaCha20 is the most commonly used ChaCha variant that performs 20 rounds. Every round contains 4 quarter-rounds and every quarter-round consist of 4 additions, 4 `XOR`s, and 4 rotations of 32-bit words. Because the RV32I base ISA lacks rotation instructions, every rotation has to be replaced by 2 shift instructions and an `OR` instruction. In total we require 20 single-cycle instructions to implement the ChaCha quarter-round.

The other parts are straightforward. As long as the input to the stream cipher is longer than 64 bytes, we generate the key-stream and `XOR` it with the input in blocks of 64 bytes. If the input length is not divisible by 64 bytes, there will be some bytes remaining that still need to be encrypted. For those, another 64 bytes of key-stream is generated. These are `XOR`ed with the input first per word (4 bytes) and finally per byte.

7.4.1 Result

Our implementation of the complete Chacha20 stream cipher requires 32 bytes in the DTIM memory to store constants and another 40 bytes on the stack to store callee-save registers. We benchmark speed with the same fixed input size of 4096 bytes as we used for the bitsliced AES-128-CTR implementation. This can be encrypted or decrypted in 114 365 clock cycles, or at 27.9 cycles per byte.

7.5 Keccak

The Keccak-f family of permutations was designed in the course of the SHA-3 competition [BDPV11b]. The Keccak-f[1600] instance is now at the core of the SHA-3 hash functions and the SHAKE extendable output functions standardized by NIST [NIS15b]. It it also used in various other cryptographic functions. An optimized implementation of the Keccak-f[1600] permutation therefore benefits all those schemes. In the *Keccak implementation overview* a number of implementation techniques are discussed, including those relevant to 32-bit software implementations [BDP+12].

7.5.1 Efficient Scheduling

The permutation operates on a relatively large state of 1600 bits. Having the RV32I architecture in mind, this state is clearly too large to fit into registers. It is therefore required to swap parts between memory and registers during the computation. Loads from memory and stores to memory are relatively expensive, so for an efficient implementation it is important to keep the number of loads and stores at a manageable level.

The permutation iterates a round function consisting of the steps θ, ρ, π, χ, and ι. The first four steps each process the full state. Computing them one by one would therefore use many loads and stores. The designers described a technique to merge the computation of these steps such that only two passes over the full state are required per round. This is explained in detail in the implementation overview document [BDP+12]. We follow the same approach for our RISC-V implementation.

7.5.2 Bit Interleaving

The state is structured as 5×5 64-bit lanes. On a 32-bit architecture, one could simply split the lanes into two halves that are stored in separate registers, but for the permutation itself, it is more efficient to interleave the bits. The bits with an *even* index are then stored in one register and those with an *odd* index in another. The lane-wise translations in θ and in ρ then become 32-bit rotations. It has been mentioned before that the RV32I base ISA does not contain rotation instructions.

In fact, with both approaches a lane-wise translation costs 6 single-cycle instructions. The difference is that with the interleaved representation, for translation offsets of 1 or -1 only a single register has to be rotated. Those then only cost 3 single-cycle instructions. Because this is the case for 6 out of 29 lane translations per round, bit interleaving still provides a nice improvement if one only considers Keccak-f.

7.5.3 Lane Complementing

The χ step computes 5 `XOR`, 5 `AND`, and 5 `NOT` (64-bit) operations on the lanes of every plane of the state. There are 5 such planes and we only have 32-bit instructions, so in total χ requires 50 `XOR` instructions, 50 `AND` instructions, and 50 `XORI` instructions with -1 as immediate per round. The number of `XORI` instructions can be reduced to 10 by representing certain lanes by their complement and by changing some `AND` instructions into `OR` instructions. This comes at the cost of applying a mask at the beginning and at the output of Keccak-f. This technique is also described in more detail in the implementation overview document [BDP+12]. This is a useful technique on the RISC-V, because there is no instruction that combines an `AND` with a `NOT` of one of its operands, as is the case on some other architectures.

7.5.4 Result

Our RISC-V implementation is inspired by the fastest Cortex-M3/M4 implementation known to us, which is the `KeccakP-1600-inplace-32bi-armv7m-le` implementation in the eXtended Keccak Code Package.[4] The main differences are that we add lane complementing and that we keep more variables in registers instead of having to store them on the stack.

Memory-wise our implementation requires 192 bytes in the DTIM memory for the round constants and 20 bytes on the stack. To benchmark speed, we measure a single execution of the permutation from the instruction cache. This takes 13 774 clock cyles, or 68.9 cycles per byte.

7.6 Arbitrary-Precision Arithmetic

Arbitrary-precision arithmetic on integers, also called big-integer arithmetic, is a core component of public-key cryptographic systems such as RSA and elliptic-curve cryptography. We consider addition and two multiplication algorithms, schoolbook and Karatsuba multiplication. The multiplication algorithms make heavy use of the RISC-V M extension. This provides a 32×32-bit multiplier and the `MUL` and `MULHU` instructions, among some others that we will not use. `MUL` gives the lower 32 bits of the 64-bit multiplication result, `MULHU` the higher 32 bits, interpreting its operands both as unsigned values. On the E31, they each have a result latency of 2 clock cycles.

7.6.1 Carries and Reduced-Radix Representations

An arbitrarily large integer is usually represented as a vector of CPU words. The part of the integer that fits in a single CPU word is called a *limb*. Arithmetic on arbitrary-precision integers then translates to an algorithm that performs arithmetic with the limbs, as those are the only units that a CPU can work with.

The addition of two limbs may result in an overflow. On most CPU architectures, whether an overflow occurred is stored in a carry flag. This can then subsequently be used in an add-with-carry operation.

---

[4] `https://github.com/XKCP/XKCP`

RISC-V, however, does not specify the existence of a carry flag. Instead, the carry needs to be explicitly computed every time. The `SLTU` instruction (set less than unsigned) is very useful for this. Let $r = a + b$, where r, a, and b are unsigned 32-bit values. The addition produces a carry c when $r < a$ (or $r < b$). In assembly, this can be implemented with `ADD r, a, b; SLTU c, r, a`.

This explicit carry handling can be the cause of a significant overhead. One way to avoid this is by guaranteeing that a carry will not occur. This is possible by using a reduced-radix representation, also known as a redundant integer representation. Instead of the full 32 bits, one can use the least-significant k bits of every limb, such that the most-significant $32 - k$ bits are zero at the start. This *radix-*2^k *representation* requires more limbs to store an integer of the same bit length, but the advantage is that one can do one or even many additions without producing an overflow. The carries are accumulated in the most-significant $32 - k$ bits of the same limb. Only in the end they may need to be added to the next limb to get back to a unique integer representation.

What is more efficient is highly application-dependent, as that determines how many and which operations are computed on the integers. We aim to keep this neutral by studying the performance of both types of addition and multiplication algorithms for an arbitrary number of limbs, without specifying a precise radix.

7.6.2 Addition

Arbitrary-precision addition is a simple operation that consists of a carry chain for full-limb (radix-2^{32}) integer representations. The operands are added limbwise, where every such addition may result in an overflow that has to be carried to the next limb.

Figure 7.1 shows how both reduced and full representations compare. It appears that carry handling is a significant part of the computational effort. A reduced-radix representation is approximately 37% faster than a representation that is not redundant. However, one should note that with a reduced-radix representation, more limbs will be required. For example, it is fairer to compare the reduced-radix representation with 12 limbs to the full-radix representation with 10 limbs, when only 27 bits are used in every limb, i. e., in radix 2^{27}. The cost of carrying at the end to get back to a unique representation also needs to be taken into account.

Still, it appears that reduced-radix representations can be beneficial when multiple additions have to be computed.

Figure 7.1 also shows the estimated cost of full-limb addition if there were a carry flag and add-with-carry operation. This is discussed in Section 7.7.4.

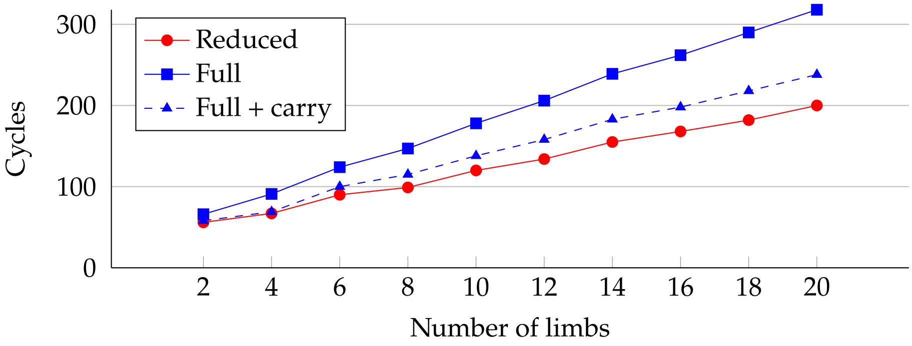


Figure 7.1: Performance of arbitrary-precision addition.

7.6.3 Schoolbook Multiplication

Many algorithms exist to implement arbitrary-precision multiplication. One of the simplest ones is called schoolbook multiplication. With the schoolbook multiplication method multiplying two n-limb integers takes n^2 single-limb (in our case: 32×32-bit) multiplications.

A representation that is not reduced still has to perform some carry handling, but the cost of this is much less significant with multiplication compared to addition, as can be seen in Fig. 7.2. Schoolbook multiplication with reduced-radix representations is only 8% faster than multiplication with representations that are not reduced. And because more limbs will be required, there is actually very little advantage to using a reduced-radix representation.

This can be explained by the fact that the `LW`, `SW`, `MUL`, and `MULHU` instructions take more CPU cycles compared to the simpler bitwise and arithmetic instructions. A reduced-radix representation does not avoid this more significant part of the cost of the inner loop of the algorithm.

7.6.4 Karatsuba Multiplication

The Karatsuba algorithm was the first multiplication algorithm that was discovered that has a lower asymptotic time complexity than $\mathcal{O}(n^2)$ [KO63]. Instead, it can recursively multiply arbitrary-precision integers in $\mathcal{O}(n^{\log_2 3})$. It succeeds in this by effectively trading an n-limb multiplication for 3 $\frac{n}{2}$-limb multiplications and several additions.

The details of the Karatsuba multiplication algorithm have been extensively covered in other works. It is used in many implementations of cryptographic schemes, most notably for RSA [SV93] and elliptic-curve cryptography [BCL14; DHH+15; FA19], but also for more recent lattice-based [KRS19] and isogeny-based [SLLH18] post-quantum cryptography.

We implement a single level of subtractive Karatsuba that multiplies two equal-length operands with an even number of limbs. This restriction is only there to simplify the performance analysis by being able to omit a few implementation details for dealing with special cases. The case of equal-length operands with an even number of limbs is also in fact the most common scenario in cryptography, which is why it is not even necessarily a relevant restriction.

Figure 7.2 shows that even for a very small number of limbs, the Karatsuba multiplication algorithm is already faster than schoolbook multiplication. This is not obvious, as the cost of the extra additions and constants in the complexity typically imply a certain threshold where Karatsuba starts to perform better.

The gap between reduced-radix representations and nonreduced or full-limb representations is slightly larger than with schoolbook multiplication, which can be partially explained by the extra additions that need to be computed. Its difference is now approximately 21%. Whether this suffices to make a reduced-radix representation more efficient in practice is hard to conclude from this data. It will depend on the specific application.

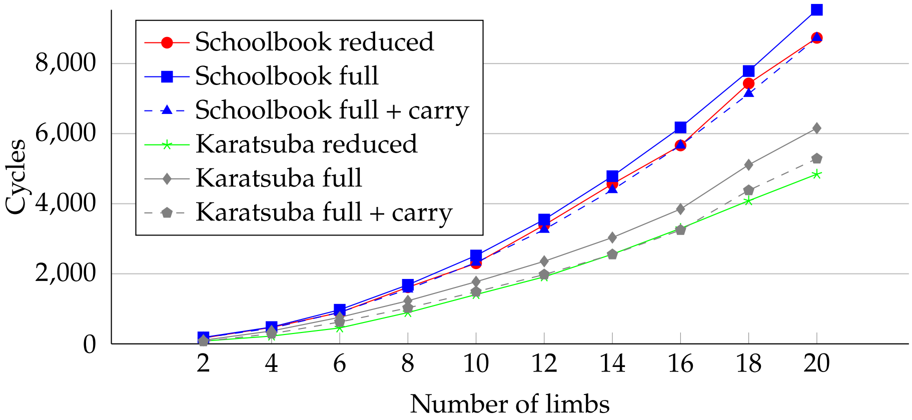


Figure 7.2: Performance of arbitrary-precision multiplication.

7.7 Extending RISC-V and Discussion

7.7.1 Speed Comparison with ARM Cortex-M4

The RISC-V platform that we used has similarities with ARM Cortex-M microprocessors. Both have 32-bit architectures and are designed for cheap embedded applications. The main difference is that ARM microprocessors have a richer (proprietary) instruction set. For example, rotations are always available in ARMv7-M and can even be combined with arithmetic instructions in a single CPU cycle. The architecture also provides nicer bit-extraction instructions, a carry flag and a single-cycle add-with-carry. On the other hand, RV32I comes with more registers, which may benefit cryptographic primitives that have a larger state. This can save a lot of overhead of having to spill values to the stack.

At first sight, it is unclear which weighs more heavily. We therefore compare the relative performance of our optimized cryptographic primitives with their counterpart on the Cortex-M4. There already exist AES, ChaCha20, and Keccak-$f[1600]$ assembly implementations optimized for that platform.

Table 7.1: Comparison between the E31 (RV32IMAC) and the Cortex-M4.

Scheme	Cortex-M4		E31/RV32IMAC	
	Cycles	Cyc./byte	Cycles	Cyc./byte
Table AES-128 key exp.	244.9 [SS16c]		340	
Table AES-128	634.7 [SS16c]	39.8	912	57.0
Bitsliced AES-128 key exp.	1021.9 [SS16c]		1239	
Bitsliced AES-128-CTR	413 849.6*	101.0 [SS16c]	509 622*	124.4
ChaCha20 encrypt	56 934.4*	13.9 [HRS16]	114 365*	27.9
Keccak-$f[1600]$ permute	12 969†	64.8	13 774	68.9

* When encrypting 4096 bytes.

† We benchmarked the `KeccakP1600_Permute_24rounds` function from `https://github.com/XKCP/XKCP/blob/master/lib/low/KeccakP-1600/ARM/KeccakP-1600-inplace-32bi-armv7m-le-gcc.s` on a board with an STM32F407 microcontroller.

Table 7.1 provides the exact numbers, while Fig. 7.3 visualizes their relative speed. It can be seen that all schemes require more cycles with the RV32I architecture. Of course, this does not directly relate to speed in practice, as we do not take the different CPU clock frequencies into account. It shows that all schemes use instructions that can be computed in a single cycle on the Cortex-M4, but not with RV32I. Relatively, it appears that ChaCha20 has the largest disadvantage because of this. For this scheme, the lack of rotation instructions seems to outweigh the possibility to keep the full state in registers without spilling to the stack, something that is necessary on the Cortex-M4.

7.7.2 The RISC-V B Extension

The RISC-V foundation reserved the *B* extension for bit manipulation instructions. In 2017 there was an active working group that would develop a specification for the B extension. However, the working group dissolved in November 2017 for bureaucratic reasons.[5] An independent fork was developed outside of the RISC-V foundation, which was merged back and made official in March 2019.

The latest V0.92 draft specification adds 58 new instructions.[6] While it is unknown which will be used in the end, it is likely that this will include some type of rotation, byte shuffle, and bit-extraction instructions. The current specification also includes an and-with-complement instruction. This would imply that lane complementing would no longer be advantageous for Keccak-f[1600].

We estimate the impact that this extension will have, focussing on rotations. For each scheme, we counted all instruction sequences that could be replaced by a rotation instruction. Our table-based AES does not use rotations, while the bitsliced AES implementations uses 144 of them. ChaCha20 uses 320 rotation instructions and Keccak-f[1600] 1248.

Assuming that the rotation would be done in a single cycle, we calculated how many CPU cycles would be saved by having this instruction. The results

---

[5] `https://groups.google.com/forum/#!forum/riscv-xbitmanip`

[6] `https://github.com/riscv/riscv-bitmanip`

can be seen in Table 7.2 and Fig. 7.3. For Keccak and especially for ChaCha20, rotations are a significant part of their computational cost. From Fig. 7.3 it is clear that with rotations, the Keccak-f permutation can be computed in fewer cycles than on the Cortex-M4. This is because more registers are available.

Table 7.2: Estimated improvement with a rotation instruction.

Scheme	Rotations	Improvement	Cycles/byte
Table-based AES	0	0.0%	57.0
Bitsliced AES	144	7.0%	115.7
ChaCha20	320	35.8%	17.9
Keccak-f[1600]	1248	18.1%	56.4

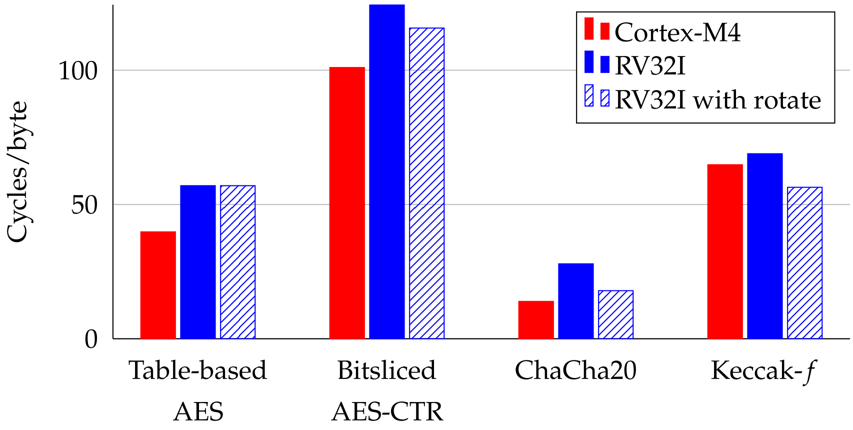


Figure 7.3: Speed of cryptographic primitives.

7.7.3 Number of Registers

We already discussed some consequences of the large number of registers that are available on the performance of these implementations. Especially ChaCha and Keccak, but also the bitsliced AES implementation, benefit from having to spill fewer intermediate values to the stack. It is noteworthy to mention that the RV32E instruction set, which is nearing its completion and which is intended to target embedded devices, will most likely decrease the number of registers from 32 to 16 [RIS19]. This will set back the performance of aforementioned schemes, but this may be compensated by supporting the B extension with a rotation instruction.

7.7.4 Carry Flag

In Section 7.6 we studied the performance of arbitrary-precision addition and multiplication with and without reduced-radix integer representations. We now estimate how full-limb representations would perform if an RV32I core was extended with a carry flag and an add-with-carry instruction. We assume that this instruction would have a result latency of a single CPU clock cycle, similar to a regular addition instruction.

For addition, 4 cycles per limb would be saved in our implementation. We then subtracted $4n$ cycles from the full-radix addition results, where n is the number of limbs. The result can be seen in Fig. 7.1. As is to be expected, addition with this instruction is almost as fast as reduced-radix representation, the only difference being the top (most-significant) limb that gets set.

With schoolbook multiplication $2n^2$ cycles are subtracted, as we can save 2 cycles in the inner loop with the add-with-carry instruction, which is executed n^2 times. For Karatsuba multiplication we computed that the add-with-carry instruction would save $27\frac{n}{2} + 6\left(\frac{n}{2}\right)^2$ cycles. The quadratic term comes from the cycles that are saved with the schoolbook multiplications and the linear part from the cycles that are saved with additions. Figure 7.2 contains plots for both estimates. With an add-with-carry instruction both schoolbook and Karatsuba multiplication would be approximately as fast as their reduced-radix counterparts. The reduced-radix implementations use more limbs and still need to carry at the end, so it is appears that an add-with-carry instruction completely compensates for any advantage that a reduced-radix implementation gives.

7.8 Conclusion

We showed how AES, ChaCha, and Keccak-f can be implemented efficiently on the 32-bit variant of the promising open-source RISC-V architecture. We also showed how arbitrary-precision addition and multiplication can be implemented and studied the performance of all these primitives. As the RISC-V is an open design intended to be extensible, we showed for several features, such as a rotation instruction and an add-with-carry instruction, how much improvement exactly could be gained by adding these features. These numbers are essential for making reasonable trade-offs in software-hardware co-design and we hope that they will be found useful by a wide audience.

PART III

Side-Channel Countermeasures

Chapter 8

Vectorization

Part III introduces techniques to optimize specific countermeasures against side-channel attacks. Masking is one of these countermeasures. This chapter is about how vector registers in CPUs can be exploited in the implementation of parallel masking schemes. In comparison to the original publication [GPSS18], the appendix has been merged into the text and minor textual edits have been made.

8.1 Introduction

There is a long history of protecting AES [DR02] implementations against side-channel analysis (SCA) or side-channel attacks. Side-channel attacks exploit physical information, such as power consumption or electromagnetic radiation of devices running some cryptographic primitive, to learn information about secret data, typically cryptographic keys. Higher-order masking is a well-studied countermeasure against such attacks [CJRR99; GP99]; unfortunately, it comes at a rather high cost in terms of performance. This is a reason why in practice, well-protected implementations are not as ubiquitous as one would hope. In software, higher-order masked implementations are typically orders of magnitude slower compared to unprotected implementations, as was explored at Eurocrypt 2017 by Goudarzi and Rivain [GR17].

Simultaneously at Eurocrypt 2017, a theoretical model was proposed to study the security of *parallel implementations* of masking schemes, called the bounded-moment leakage model [BDF+17]. As parallelization can be a very powerful tool to increase performance, this model gives the foundation for faster protected implementations. One common way to parallelize software implementations is through vectorization. In a vectorized implementation, a single instruction operates on multiple data elements inside one vector register at the same time. For vectorization to be useful, data parallelism is required, which in the case of higher-order masking is trivially provided by the necessity to compute on multiple shares.

Precisely this approach of vectorization with data-level parallelism coming from multiple shares was used in a CHES 2017 paper by Journault and Standaert [JS17]. That paper studies a parallel bitsliced (i. e., vectorized with 1-bit vector elements) implementation using 32 shares on the ARM Cortex-M4. The reason for using 32 shares was the fact that the Cortex-M4 has 32-bit registers and bitslicing is then very efficient at 32× data-level parallelism. Empirical tests described in that paper confirmed that the bounded-moment model is also useful in practice. Specifically, these tests showed that a 4-share version of their implementation yielded no leakage of order less than 4. It is of course still possible that the actual security order is lower, but it can at least be viewed as an encouraging result. They conclude their evaluation by performing an information-theoretic analysis of the leakage in order to bound the attack complexity for the 32-share implementation.

In this chapter we study how the powerful NEON vector unit on larger ARM Cortex-A processors can be used to obtain efficient masked AES implementations. Straightforwardly adapting the approach from [JS17] to obtain data-level parallelism would result in implementations with 64 or 128 shares (for 64-bit or 128-bit vector registers), which would be a security overkill and result in terrible performance. Instead we follow the approach of the bitsliced AES implementations presented in [Kön08; KS09], which exploit the data-level parallelism of 16 independent S-box computations. As a result, we present implementations using 4 and 8 shares, which in theory offer security up to the 3rd and 7th order. We use refreshing and multiplication algorithms that are based on the algorithms in [BDF+17] and even slightly improve on some of them by requiring less randomness. They are proven secure in the bounded-moment model and also proven to satisfy strong non-interference [BBD+16]. We provide a concrete evaluation of our implementations on a BeagleBone Black, which has been used successfully before to perform differential electromagnetic analysis at 1 GHz [BGRV15]. Using nearly the same setup, we employ the popular TVLA methodology [CDG+13] in conjunction with leakage certification [DSP16] and we show that there is actually some leakage in the 3rd order of our 4-share implementation, but not in the 2nd order. We then continue to bound the measurement complexity of the 8-share implementation using an information-theoretic approach [DFS15].

To summarize, the contributions of this chapter are that

- we provide the first vectorized instantiation of the bounded-moment leakage model published at Eurocrypt 2017 [BDF+17] with strong non-interference [BBD+16];
- we provide the fastest publicly available higher-order masked AES implementations with 4 and 8 shares for the ARM Cortex-A8; and that
- we perform a practical side-channel evaluation of the 4-share AES implementation and derive security bounds for the 8-share implementation.

Source code. The source code of our implementations is available in the public domain. It can be downloaded at `https://github.com/Ko-/aes-masked-neon`.

8.2 Preliminaries

8.2.1 Higher-Order Masking of AES

Implementations of cryptographic primitives such as block ciphers are typically vulnerable to attacks that use SCA. Information about physical characteristics, such as the electromagnetic radiation, of a device that executes a block cipher can be used to recover the secret key [GMO01; Koc96].

A well-studied countermeasure against this class of attacks is (higher-order) masking. It works by splitting each secret variable x into d shares x_i that satisfy

$x_0 \oplus x_1 \oplus x_2 \oplus \cdots \oplus x_{d-1} = x$. When $\oplus$ denotes the Boolean `XOR` operation, this is called Boolean masking. Any $d-1$ of these shares should be statistically independent of x and should be uniformly randomly distributed. If this is the case, then this masking scheme provides privacy in the $(d-1)$-probing model, as put forward by Ishai, Sahai, and Wagner [ISW03]. The idea is that an attacker applying $d-1$ probes to learn intermediate values of the computation will not be able to learn anything about the secret value. The value $d-1$ is called the *order* of the masking scheme.

When masking is applied, operations on x are to be performed on its shares. For linear operations f, those that satisfy $f(x+y) = f(x)+f(y)$ and $f(ax) = af(x)$, it holds that they can trivially be computed on the shares of x individually. For nonlinear operations, several algorithms have been suggested to retrieve the correct result. In [ISW03] it was shown how to compute a masked `AND` gate and, together with the linear `NOT`, this is functionally complete.

AES [DR02] in particular has received a lot of attention when it comes to protected implementations. The round function of AES consists of AddRoundKey, SubBytes, ShiftRows, and MixColums. AddRoundKey, ShiftRows, and MixColumns are all linear. SubBytes is not. Much research has therefore been aimed at finding efficient representations of a masked variant of the AES S-box [CB08; GR17; KHL11; RP10].

8.2.2 Strong Non-interference

Strong non-interference (SNI) is a security notion, formalized in [BBD+16], that is slightly stronger than probing security. It currently seems to be the right security notion when considering practical security. The problem with probing security is that, given two algorithms that are secure at order $d-1$ in the probing model, the composition of these algorithms is not necessarily secure at order $d-1$. SNI, on the other hand, means that an algorithm is *composable*, guaranteeing that one can verify the security of the composition of multiple secure algorithms.

As an example to see why SNI is desirable, consider the provably secure masking scheme by Rivain and Prouff from CHES 2010 [RP10]. Three years later, an attack was found against the composition of the refreshing of masks and the masked multiplication [CPRR14]. The scheme was fixed subsequently. It was shown in [BBD+16] that the main difference between the original and the fixed algorithms is exactly this notion of strong non-interference.

Automated verification tools exist to formally prove strong non-interference. This gives stronger guarantees on the theoretical security of a masking scheme.

8.2.3 Bounded-Moment Leakage Model

The probing model and its variants are not always straightforward to interpret. The fact that $d - 1$ shares should be statistically independent is based on the idea that an attacker can inspect the leakage of intermediate computations on the shares separately. In software, it therefore applies better to serial implementations. When computations are performed on multiple shares in parallel, it is not immediately clear what the relation with the probing security model is.

To handle this, the bounded-moment model has been proposed in [BDF+17]. It is more targeted toward parallel implementations and can deal with the concept that multiple shares are manipulated simultaneously. Barthe, Dupressoir, Faust, Grégoire, Standaert, and Strub proved that probing security of a serial implementation implies bounded-moment security for its parallel counterpart. It is a weaker security notion than the noisy leakage model [CJRR99; PR13].

Security in the bounded-moment model is defined using *leakage vectors* and *mixed moments*. For every clock cycle c, there is a leakage vector L_c. The leakage vector is a random variable that is computed as the sum of a deterministic part that depends on the shares that are manipulated, and on the noise R_c. The mixed moment of a set $\{Y_1, \ldots, Y_r\}$ of r random variables at orders $o_1, \ldots, o_r$ can be defined as $\mathrm{E}\left[\prod_{i=1}^{r} Y_i\right]$, where E denotes the expected value. Now, consider an N-cycle cryptographic implementation that manipulates a secret variable x. This results in a set $\{L_1, \ldots, L_N\}$ of N leakage vectors. The implementation is said to be secure at order o in the bounded-moment model if all the mixed moments of order $\leq o$ of $\{L_1, \ldots, L_N\}$ are statistically independent of x.

8.2.4 Vectorization with NEON

The ARM Cortex-A8 is a 32-bit processor that implements the ARMv7-A microarchitecture. It is used in smartphones, digital TVs, and printers, among others. It was first introduced in 2005 and is currently widely deployed. Its main core can run at 1 GHz and implements features such as superscalar execution, an advanced branch prediction unit, and a 13-stage pipeline. There are 16 32-bit `r` registers, of which 14 are generally available to the programmer.

The Cortex-A8 comes with the so-called Advanced SIMD extension, better known as NEON, that adds another 16 128-bit `q` registers. These vector registers can also be viewed as 32 64-bit `d` registers. For example, `q0` consists of `d0` and `d1`, `q1` consists of `d2` and `d3`, etc. Operations can typically be performed on 8-bit, 16-bit, or 32-bit elements in a SIMD fashion. While 128-bit registers are supported, the data path of the Cortex-A8 is actually only 64 bits wide, which means that operations on 128-bit registers will be performed in two steps. NEON has a separate 10-stage pipeline. In particular, it has a load/store unit that runs next to an arithmetic unit. This means that an aligned load and an arithmetic instruction can be executed in the same cycle.

NEON has been used successfully in the past to vectorize and optimize implementations of cryptographic primitives [BS12], but its power has to the best of our knowledge not yet been exploited for higher-order masking in the way that we propose here.

8.3 Vectorizing Masking of AES

8.3.1 Representing the Masked State

The AES state [DR02] is usually pictured as a square matrix of 4 by 4 byte-sized elements. This representation leads to efficient software implementations when SubBytes is implemented using lookup tables. However, such implementations are also prone to cache-timing attacks [Ber05a], as the memory location of the value that is looked up depends on some secret intermediate value. An alternative bitsliced representation avoids these attacks. In this bitsliced representation, all the first bits of every byte are put in one register, all the second bits in the next register, etc. For SubBytes, one can now compute the S-box on the individual bits and do that for all 16 bytes in parallel. The S-box parallelism of AES for bitslicing was first exploited by Könighofer in [Kön08] and it was also used in the speed-record-setting AES implementation targeting Intel Core 2 processors by Käsper and Schwabe [KS09]. At a small cost, the other (linear) operations of AES are modified to operate on this bitsliced representation as well.

However, on most devices registers are longer than 16 bits, so it would be a waste to not utilize this. AES implementations without side-channel protections choose to process multiple blocks in parallel, by simply concatenating multiple 16-bit chunks from independent blocks in one register. For example, the AES implementation of [KS09] processes 8 blocks in parallel in a 128-bit XMM register. When the vector registers become larger, this trivially leads to higher throughputs for parallel modes of operation.

In this section we present three implementations that, instead of multiple blocks, process multiple shares in parallel. The first implementation fills a 64-bit `d` register with 4 shares. The second has 8 shares, that are used to fill a 128-bit `q` register. The third combines 2 blocks with each 4 shares, and also utilizes the 128-bit `q` registers. It interleaves the shares of the 2 blocks for efficiency reasons. Note that this third implementation requires a parallel mode of operation.

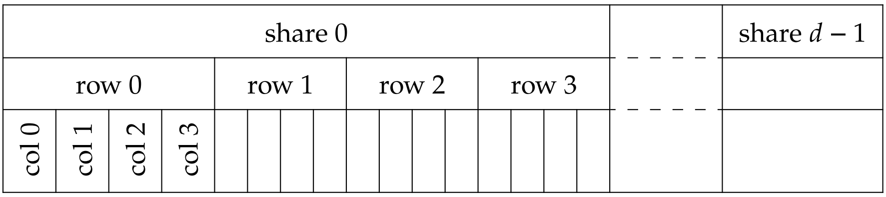


Figure 8.1: Register layout for the single-block implementations. There are 8 of these $16d$-bit vector registers. Rectangles on the bottom left represent individual bits.

8.3.2 Parallel Multiplication and Refreshing

In [BDF+17], new algorithms for parallel multiplication (including the `AND` operation) and parallel refreshing were proposed. They are proven to be secure in the bounded-moment model and proven to satisfy strong non-interference using techniques from automated program verification [BBD+15]. Correct implementations of these algorithms are critical for the security of our implementations. We suggest slightly improved algorithms for $d = 4$ and $d = 8$ that require less randomness, but we could not generalize them to an improvement for all orders. As with the original algorithms, they are proven secure using the same automatic verification tools.

Refreshing – 4 shares. Refreshing can be necessary to make sure that values in registers are again statistically independent. The refreshing algorithm in [BDF+17] requires $2d$ bytes of fresh uniform randomness. Let $\mathbf{x}$ (in boldface) denote a vector register that contains $[x_0, \ldots, x_{d-1}]$, where $\bigoplus_{i=0}^{d-1} x_i = x$, and let $\mathbf{r}$ be a vector of the same length that contains uniformly random values. For AES, a single share takes 2 bytes in a register, so a randomness vector $\mathbf{r}$ will be $2d$ bytes.

Then $\mathbf{x}' = \mathbf{r} \oplus \mathrm{rot}(\mathbf{r}, 1) \oplus \mathbf{x}$ is a secure way to refresh x, where $\mathrm{rot}(\mathbf{a}, n)$ rotates $\mathbf{a}$ to either left or right by n shares. In the case of AES, this is equal to applying a rotation by $2n$ bytes. For 4 shares, this algorithm additionally achieves SNI.

Listing 8.1: NEON implementation of refreshing with 4 shares.

```
//param rand is r register with address of randomness
//param a is d register to refresh
//param tmp is d register that gets overwritten
.macro refresh rand a tmp
  vld1.64 {\tmp}, [\rand]! //get 8 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #1
  veor \a, \tmp
.endm
```

Refreshing – 8 shares. While the previous refreshing algorithm generalizes to 8 shares, it no longer achieves SNI at 8 shares. To reach this, in [BDF+17] it turned out to be necessary to iterate the refreshing algorithm 3 times. In other words, one would need to compute

$$\mathbf{r} \oplus \mathrm{rot}(\mathbf{r}, 1) \oplus \mathbf{r}' \oplus \mathrm{rot}(\mathbf{r}', 1) \oplus \mathbf{r}'' \oplus \mathrm{rot}(\mathbf{r}'', 1) \oplus \mathbf{x}$$

to achieve SNI at order 7. This requires 3 vectors of uniform randomness, or 48 bytes with AES. We improve this algorithm by computing

$$\mathbf{r} \oplus \mathrm{rot}(\mathbf{r}, 1) \oplus \mathbf{r}' \oplus \mathrm{rot}(\mathbf{r}', 2) \oplus \mathbf{x}\,.$$

We verified with the current version of the tool of [BBD+15] that this also achieves SNI at order 7. Moreover, it requires one less randomness vector. In the case of AES, we now require 32 bytes of uniform randomness.

Listing 8.2: NEON implementation of refreshing with 8 shares.

```
//param rand is r register with address of randomness
//param a is q register to refresh
//param tmp is q register that gets overwritten
.macro refresh rand a tmp
  vld1.64 {\tmp}, [\rand:128]! //get 16 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #1
  veor \a, \tmp
  vld1.64 {\tmp}, [\rand:128]! //get 16 bytes of randomness
  veor \a, \tmp
  vext.16 \tmp, \tmp, #2
  veor \a, \tmp
.endm
```

Multiplication – 4 shares. Multiplication in a finite field, or an `AND` gate in the case of $\mathbb{F}_2$, is trickier to perform in a secure way. Consider the case where one wants to compute $z = x \cdot y$. Let $\mathbf{r}$ and $\mathbf{r}'$ be uniformly random vectors. Then, with 4 shares, the algorithm suggested in [BDF+17] computes the following to achieve SNI at order 3:

$$\begin{aligned}\mathbf{z} = \mathbf{x} \cdot \mathbf{y} \oplus \mathbf{r} \oplus \mathbf{x} \cdot \mathrm{rot}(\mathbf{y}, 1) \oplus \mathrm{rot}(\mathbf{x}, 1) \cdot \mathbf{y} \oplus \mathrm{rot}(\mathbf{r}, 1) \\ \oplus \mathbf{x} \cdot \mathrm{rot}(\mathbf{y}, 2) \oplus \mathbf{r}' \oplus \mathrm{rot}(\mathbf{r}', 1)\,.\end{aligned}$$

However, we can again improve this slightly such that less randomness will be necessary. Let r_4 be a uniformly random value. Then we proved using the tool of [BBD+15] that the following is also 3rd-order SNI-secure. For AES, this requires 10 fresh uniformly random bytes (8 for $\mathbf{r}$ and 2 for r_4) instead of 16:

$$\begin{aligned}\mathbf{z} = \mathbf{x} \cdot \mathbf{y} \oplus \mathbf{r} \oplus \mathbf{x} \cdot \mathrm{rot}(\mathbf{y}, 1) \oplus \mathrm{rot}(\mathbf{x}, 1) \cdot \mathbf{y} \oplus \mathrm{rot}(\mathbf{r}, 1) \\ \oplus \mathbf{x} \cdot \mathrm{rot}(\mathbf{y}, 2) \oplus [r_4, r_4, r_4, r_4]\,.\end{aligned}$$

Listing 8.3: NEON implementation of multiplication with 4 shares.

```
//param rand is r register with address of randomness
//param c is d register where result gets stored
//param a and b are d registers to and, remain unchanged
//param tmp and tmpr are d registers that get overwritten
.macro masked_and rand c a b tmp tmpr
  vand \c, \a, \b //z = x.y
  vld1.64 {\tmpr}, [\rand]! //get 8 bytes of randomness
  vext.16 \tmp, \b, \b, #1
  veor \c, \tmpr  // + r
  vand \tmp, \a
  veor \c, \tmp   // + x.(rot y 1)
  vext.16 \tmp, \a, \a, #1
  vand \tmp, \b
  veor \c, \tmp   // + (rot x 1).y
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot r 1)
  vext.16 \tmp, \b, \b, #2
  vand \tmp, \a
  veor \c, \tmp   // + x.(rot y 2)
  vld1.16 {\tmp[]}, [\rand]! //get 2 bytes of randomness
  veor \c, \tmp   // + (r4,r4,r4,r4)
.endm
```

Multiplication – 8 shares. With 8 shares, we use the original algorithm of [BDF+17] that is SNI at order 7. This requires 3 randomness vectors, which in the case of AES amounts to 48 bytes:

$$\begin{aligned}\mathbf{z} = \mathbf{x}\cdot\mathbf{y} &\oplus \mathbf{r} \oplus \mathbf{x}\cdot\mathrm{rot}(\mathbf{y},1) \oplus \mathrm{rot}(\mathbf{x},1)\cdot\mathbf{y} \oplus \mathrm{rot}(\mathbf{r},1)\\ &\oplus \mathbf{x}\cdot\mathrm{rot}(\mathbf{y},2) \oplus \mathrm{rot}(\mathbf{x},2)\cdot\mathbf{y} \oplus \mathbf{r}'\\ &\oplus \mathbf{x}\cdot\mathrm{rot}(\mathbf{y},3) \oplus \mathrm{rot}(\mathbf{x},3)\cdot\mathbf{y} \oplus \mathrm{rot}(\mathbf{r}',1)\\ &\oplus \mathbf{x}\cdot\mathrm{rot}(\mathbf{y},4) \oplus \mathbf{r}'' \oplus \mathrm{rot}(\mathbf{r}'',1)\,.\end{aligned}$$

We attempted to reduce this by replacing the last randomness vector by a vector with a single random value, as in the algorithm for 4 shares, but we found that this does not achieve SNI at order 7.

Listing 8.4: NEON implementation of multiplication with 8 shares.

```
//param rand is r register with address of randomness
//param c is q register where result gets stored
//param a and b are q registers to and, remain unchanged
//param tmp and tmpr are q registers that get overwritten
.macro masked_and rand c a b tmp tmpr
  vand \c, \a, \b //K = A.B
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  vext.16 \tmp, \b, \b, #1
  veor \c, \tmpr  // + R
  vand \tmp, \a
  veor \c, \tmp   // + A.(rot B 1)
  vext.16 \tmp, \a, \a, #1
  vand \tmp, \b
  veor \c, \tmp   // + (rot A 1).B
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R 1)
  vext.16 \tmp, \b, \b, #2
  vand \tmp, \a
  veor \c, \tmp   // + A.(rot B 2)
  vext.16 \tmp, \a, \a, #2
  vand \tmp, \b
  veor \c, \tmp   // + (rot A 2).B
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  vext.16 \tmp, \b, \b, #3
  veor \c, \tmpr  // + R'
  vand \tmp, \a
  veor \c, \tmp   // + A.(rot B 3)
  vext.16 \tmp, \a, \a, #3
  vand \tmp, \b
  veor \c, \tmp   // + (rot A 3).B
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R' 1)
  vext.16 \tmp, \b, \b, #4
  vand \tmp, \a
  veor \c, \tmp   // + A.(rot B 4)
  vld1.64 {\tmpr}, [\rand:128]! //get 16 bytes of randomness
  veor \c, \tmpr  // + R''
  vext.16 \tmpr, \tmpr, #1
  veor \c, \tmpr  // + (rot R'' 1)
.endm
```

Randomness. Implementations that are protected using higher-order masking require a lot of randomness. To be able to prove statistical independence, this randomness should be fresh and uniformly distributed. For resisting attacks in practice, it is not so clear whether the exact requirements are this strict. For instance, it might also be fine to expand a random seed using a pseudo-random number generator, or even to re-use randomness [BGG+14]. We consider this discussion to be out of scope of this work. However, because the impact on the performance can be very significant, we consider various approaches that occur in the literature. The first is to read all the randomness that we require from `/dev/urandom` using `fread`, like in [BGRV15]. This is the most conservative approach, but it is rather slow. Second, we also consider the case where all required randomness is already in a file that needs to be read into memory. The third approach assumes that there exists a fast true random-number generator and only considers the cost of a normal load instruction (`vld1`), like in [GR17].

The AES implementation with 4 shares requires 8 bytes per refresh and 10 bytes per masked `AND`. In the next section we will see that this amounts to $10 \cdot 32 \cdot (8 + 10) = 5760$ random bytes in total for the full AES, excluding the randomness used to do the initial masking of the input and the round keys. Naturally, the implementation that computes two blocks in parallel requires double the amount of random bytes. For 8 shares, refreshing takes 32 bytes and a masked `AND` uses 48 bytes, which makes the total $10 \cdot 32 \cdot (32 + 48) = 25\,600$ bytes.

8.3.3 SubBytes

Using the masked `AND` and refreshing algorithms, we can build our bitsliced SubBytes. Several papers have presented optimized bitsliced representations of the AES S-box. The smallest known to us at the time of the original publication [GPSS18] was by Boyar and Peralta [BP10]. It uses 83 `XOR`s/`XNOR`s and 32 `AND`s, which was shortly after improved to 81 `XOR`s/`XNOR`s and 32 `AND`s. The few `NOT`s can be moved into the key expansion, so we only need to consider `XOR`s and `AND`s. We used this implementation as our starting point, as this is also the implementation with the smallest number of binary `AND`s, and an `AND` will be much slower to compute than a `XOR`. More recent improvements to the AES S-box have not reduced the number of nonlinear operations [ME19; RTA18].

We have used the compiler provided in [BBD+16] to generate a first masked implementation of SubBytes. This tells us when it is necessary to refresh a value, making sure that we do not refresh more often than strictly necessary. For our version of SubBytes, however, the compiler adds a refresh on one of the inputs for every `AND`. Then we implement an `XOR` on multiple shares in parallel with a `veor` instruction. For an `AND`, we use the algorithms of the previous section. Finally, the code has been manually optimized to limit pipeline stalls.

The S-box implementation has many intermediate variables. With 4 shares and a single block, the `d` registers are used. There are 32 of them and this turns out to be sufficient to store all the intermediate values. With two blocks or with 8 shares, however, we can use only 16 `q` registers. This implies that values have to be spilled to the stack. Of course, we want to minimize the overhead caused by this. In [SS16c], an instruction scheduler and register allocator for the ARM Cortex-M4 was used to optimize the number of pushes to the stack. We modified this tool to handle the NEON instructions that we need, and use it to obtain an implementation with 18 push instructions and 18 loads.

According to a cycle-count simulator,[1] our SubBytes implementation takes 1035 cycles with one block and 4 shares and 2127 cycles with 8 shares.

8.3.4 Linear Layer

We now discuss the linear operations of AES. We manually optimized them using the same cycle-count simulator to hide as many latencies as possible.

---

[1] `http://pulsar.webshaker.net/ccc/index.php?lng=us`. Note that the subdomain no longer resolves. Add `185.16.44.200 pulsar.webshaker.net` to your hosts file to visit this.

AddRoundKey. AddRoundKey loads the round key with the `vld1` instruction and adds it to the state using `veor`. The loads and arithmetic instructions can be interleaved. This helps because they go into separate NEON pipelines. An arithmetic instruction can then be executed in parallel with the load of the next part of the round key. For the loads, we make sure that they are aligned to at least 64 bits. AddRoundKey then only takes 10 cycles.

ShiftRows. With ShiftRows, rotations by fixed distances over 16 bits need to be computed. This can be implemented using `vand`, `vsra`, `vshl`, and `vorr` instructions. The arithmetic pipeline is now clearly the bottleneck. According to the simulator, our ShiftRows takes 150 cycles.

MixColumns. MixColumns requires more rotations by 4 or by 12 over 16 bits. This takes 106 cycles as measured by the simulator.

8.3.5 Performance

We benchmark our implementations on the BeagleBone Black with the clock frequency fixed at 1 GHz. In other words, we disabled frequency scaling. For the rest, we did not apply any changes to a standard Debian Linux 9 installation. In particular, we did not disable background processes and did not give our process special priority or CPU core affinity. The implementations are run 10 000 times and the median cycle counts are given in Table 8.1.

When using `/dev/urandom`, more than 99% of the time is spent on waiting for randomness, which is delivered at a rate of only 369 cycles per byte in the 8-share case. With a faster RNG, it becomes clear that our implementations are very fast and practical. We reach 474 cycles/byte with 4 shares and 1 476 cycles/byte with 8 shares with pre-loaded randomness. Note that all implementations are fully unrolled, so the code size can trivially be decreased to roughly a tenth when this is a concern. However, we do not expect this to be an issue for devices with a Cortex-A8 or similar microprocessors, as they are relatively high-end.

	4 shares 1 block	4 shares 2 blocks	8 shares 1 block
Clock cycles (randomness from `/dev/urandom`)	1 598 133	4 738 024	9 470 743
Clock cycles (randomness from normal file)	14 488	17 586	26 601
Clock cycles (pre-loaded randomness)	12 385	15 194	23 616
Random bytes	5 760	11 520	25 600
Stack usage in bytes	12	300	300
Code size in bytes	39 748	44 004	70 188

Table 8.1: Performance of our masked AES implementations.

Comparison to related work. In the following we discuss how our implementation compares to related work. We note that one should be cautious when it comes to comparing cycle counts, in particular when benchmarks were obtained on different microarchitectures or from simulators.

Goudarzi and Rivain [GR17] compared the performance of different higher-order masking approaches on ARM architectures. A simplified model is assumed for the number of cycles that specific instructions take, without referring to a specific microarchitecture. Private communication made clear that they are derived from the Keil simulator based on an ARM7TDMI-S. Their fastest bitsliced implementation is claimed to take 120 972 cycles with 4 shares and 334 712 cycles with 8 shares. To achieve this performance, the presence of a fast TRNG is assumed that delivers fresh randomness at 2.5 cycles per byte. Only the cost of a normal `ldr` instruction it taken into account, which corresponds to our performance with pre-loaded randomness. Despite the differences between ARMv4T and ARMv7-A, it is clear that there is quite a performance gap.

Wang, Vadnala, Großschädl, and Xu [WVGX15] presented a masked AES implementation for NEON that appears to run in 14 855 cycles with 4 shares and 77 820 with 8 shares on a Cortex-A15 simulator. This uses a cheap LFSR-based PRNG to provide randomness of which the authors already say that it should be replaced by a better source of randomness. We require less randomness due to a different masking scheme and apply bitslicing instead of computing SubBytes with tower-field arithmetic. The Cortex-A15 is more modern and powerful than the Cortex-A8. It can decode 3 instructions instead of 2, has out-of-order execution, and its NEON unit has a 128-bit wide datapath instead of 64-bit. However, it has longer pipelines which means that the penalty for, for instance, wrong branch predictions will be higher. We ran their code on our Cortex-A8-based benchmarking device and measured 34 662 cycles for the 4-share implementation and 158 330 cycles for the 8-share implementation, but we cannot fully explain the difference due to the amount of possible causes and the unavailability of more detailed information.

Balasch, Gierlichs, Reparaz, and Verbauwhede [BGRV15] do use the same microarchitecture, but not the NEON SIMD processor. They do not mention the performance of their implementation. They explicitly say that they focus on the security evaluation and do not aim to achieve a high-throughput implementation.

Finally, Journault and Standaert [JS17] consider a bitsliced AES implementation with up to 32 shares on an ARM Cortex-M4. They exploit the parallelism of the shares, but not of AES itself as there are only 32-bit registers. An on-board TRNG is used to provide randomness at a reported speed of 20 cycles per byte. They use the refreshing and multiplication algorithms of [BDF+17] and almost the same S-box baseline implementation. Eventually, they report that 2 783 510 cycles are required to compute AES with 32 shares, of which 73% are spent on generating randomness. While this is certainly a very interesting idea, we show how the parallelism in SubBytes can additionally be exploited on a higher-end CPU with vector registers when using fewer shares might be sufficient.

Compared to unmasked implementations, there is of course still a noticeable performance penalty for adding side-channel protections. The unmasked bitsliced AES implementation of Bernstein and Schwabe [BS12] also exploits NEON to run at 19.12 cycles per byte (i. e., 306 cycles per block) in CTR mode, but that uses counter-mode caching and processes 8 blocks in parallel.

8.4 Side-Channel Evaluation

8.4.1 Measurement Setup

Balasch, Gierlichs, Reparaz, and Verbauwhede [BGRV15] described in detail how they performed DPA attacks on a BeagleBone Black running at 1 GHz. Our experimental setup and measuring environment follow their approach. The board is running Debian Jessie and has several processes running in the background. We power the board using a standard AC adapter and connect it to the measurement PC over Ethernet. A few lines of Python on the BeagleBone open a TCP socket and spawn a new AES process for every input that it receives. The measurement PC connects to the socket and sends inputs over Ethernet.

We use a LeCroy WaveRunner 8404M-MS oscilloscope with a bandwidth of 4 GHz, operating at a sampling rate of 2.5 GSamples/sec. The AES process sets a GPIO port high before the execution of AES and sets it low after AES is finished, so that it can be used as the trigger signal. We place a magnetic field probe from Langer, model RF-B 0.3-3, with a small tip on the back of the BeagleBone board, near capacitor 66. The probe is connected to a Langer amplifier, model PA 303 SMA. The acquired traces were post-processed in order to perform signal alignment. We note that OS-related interrupts in conjunction with time-variant cache behavior result in a fairly unstable acquisition process. Thus, the evaluator has to either discard a large portion of the acquired trace set or resort to more sophisticated alignment techniques such as elastic alignment [vWB11].

8.4.2 Security Order Evaluation

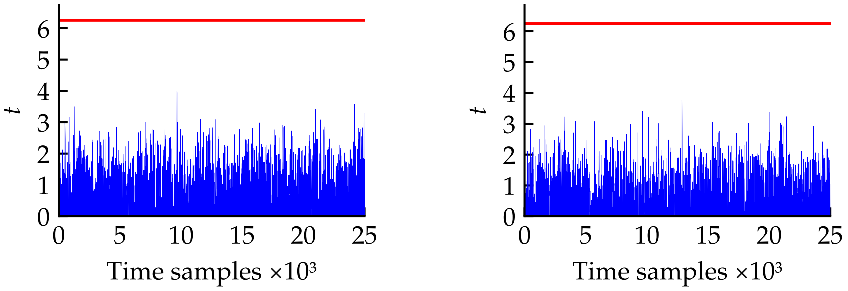
Since our implementation uses SNI gadgets, it maintains theoretical security against probing attacks of order $d - 1$ or less. The natural starting point of our side-channel evaluation is to identify any discrepancy between the theoretical and the actual security order, i. e., to determine the real-world effectiveness of the masking scheme. To achieve that goal, we need to assess whether the shares leak independently or whether the leakage function recombines them. Such recombinations can be captured by evaluating the security order in the bounded-moment model [BDF+17] using, e. g., the leakage-detection methodology [CDG+13; SM15; ZDD+18].

Several lines of work have observed divergence between the theoretical order of a masking scheme and its real-world counterpart. Initially, Balasch, Gierlichs, Grosso, Reparaz, and Standaert [BGG+14] put forward the issue of distance-based leakages, which can result in the order reduction of a scheme. Specifically, if a $(d - 1)$th-order scheme is implemented on a device that exhibits distance-based leakages, its actual order will reduce to $\lfloor (d - 1)/2 \rfloor$, damaging its effectiveness w.r.t. noise amplification. Such effects have been observed in numerous architectures such as AVR, 8051 [BGG+14], ARM Cortex-M4 [GPP+17], and FPGAs [CBG+17] and stem from both architectural choices and physical phenomena. To some extent, they can be mitigated by either increasing the order of the scheme or by hardening the implementation against effects that breach the independence of shares [PV17].

We evaluate the security order using the leakage-detection methodology known as TVLA [CDG+13], which emphasizes detection over exploitation in order to speed-up the procedure. To make the evaluation feasible w.r.t. data complexity, we focus on the first round of our single-block 4-share implementation and employ the random vs. fixed Welch t-test, which uses random and fixed plaintexts acquired in a nondeterministic and randomly interleaved manner. Consecutively, we perform univariate t-tests of orders 1 through 4 using the incremental, one-pass formulas of Schneider and Moradi [SM15] at a level of significance $\alpha = 0.00001$. The results are plotted in Figure 8.2. Note that the number of samples per trace is fairly high due to the lengthy computation of the 4-share masked AES round and due to the high sampling rate dictated by the clock frequency (1 GHz) and the Nyquist theorem. As a result, the t-test used in TVLA faces the issue of multiple comparisons and we need to control the familywise error rate using the Šidák correction $\alpha_{SID} = 1 - (1 - \alpha)^{1/\#\text{samples}}$ [Šid67]. The leakage-detection threshold is then computed using the formula $CDF^{-1}_{\mathcal{N}(0,1)}(1 - \alpha_{SID}/2)$, which equals to 6.25 when testing 25k samples per trace [ZDD+18].

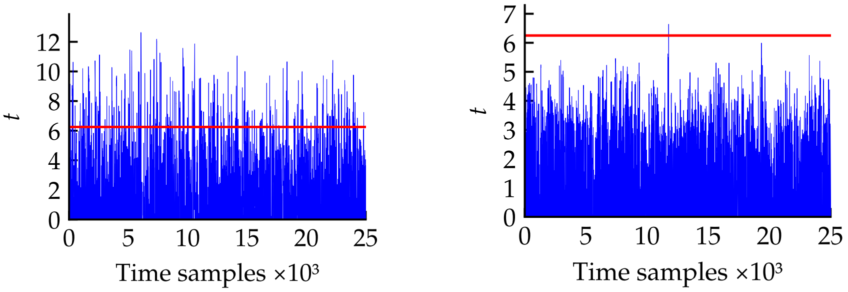
In Figure 8.2 we observe that for orders 1 and 2, a 1M random vs. 1M fixed t-test does not reject the null hypothesis, thus no leakage is detected in the first two statistical moments. The situation is different for higher orders: both the 3rd and the 4th-order univariate t-tests are able to detect leakage. This demonstrates that the actual security order of the implementation is less than the theoretical one and detecting the presence of 3rd-order leakage is in fact easier than detecting 4th-order leakage. Interestingly, the experimental results are not in direct accordance with the order reduction suggested by [BGG+14], i. e., our 3rd-order (4-share) implementation achieves practical order of 2, while the theorized reduction suggests $\lfloor 3/2 \rfloor = 1$st-order security.

An additional way to approach the order reduction issue is to phrase it as a leakage certification problem [DSP16; DSV14]. The leakage certification procedure allows us to assess the quality of a leakage model w.r.t. estimation and assumption errors. Gauging the effect of estimation errors, i. e., those that arise from insufficient profiling, is straightforward and can be carried out via cross-validation techniques [ET93]. Assumption errors are more difficult to assess, since they arise from incorrect modeling choices and would ideally require the comparison between the chosen model and an unknown perfect model. To tackle this, the indirect approach of Durvaux, Standaert, and Veyrat-Charvillon [DSV14] observes the relation between estimation and assumption



(a) 1st order

(b) 2nd order



(c) 3rd order

(d) 4th order

Figure 8.2: Univariate leakage detection, 1M random vs. 1M fixed.

errors and if the latter are negligible in comparison, they conclude that the chosen model is adequate.

In our approach, we use the t-test-based certification toolset of Durvaux, Standaert, and Del Pozo [DSP16], which focuses on the assumption and estimation errors for each statistical moment. Initially, we start with an erroneous model for our 4-share implementation: we assume that the leakage is sufficiently captured by a Gaussian template, i. e., a normal distribution that is fully described by the first two statistical moments. The results are visible in the upper part of Figure 8.3, using a trace set of size 900 000. In particular, we plot the p-value of a t-test that compares an actual statistical moment (estimated from the trace set) with a simulated statistical moment (estimated by sampling the profiled model). A high p-value (i. e., a mostly white image) indicates that estimation errors overwhelm assumption errors and that the chosen model is adequate. A small p-value indicates that assumption errors are larger than estimation errors, thus the chosen model is erroneous. The process is repeated for all first four statistical moments (mean, variance, skewness, kurtosis) using cross-validation.

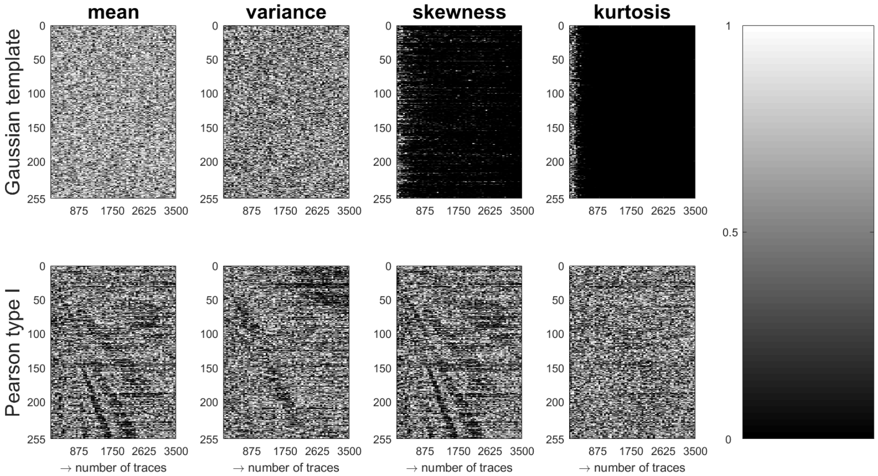


Figure 8.3: Leakage certification p-values for Gaussian templates and Pearson type I distributions.

In the first two images of Fig. 8.3 (upper part, mean and variance), the high p-values indicate that these moments are well-captured by the model. Naturally, the fourth image (upper part, kurtosis) is black, indicating that the model disregards the 4th moment of a parallel 4-share implementation which should (in theory) contain useful information. Interestingly, the third image (skewness) is also black, penalizing any model that does not include the 3rd statistical moment, although in a perfect scheme it should not convey any information. We continue this approach with a more adequate model for the 4-share implementation: we assume that the leakage is captured by a Pearson type I distribution [SMSG16], i. e., a 4-moment Beta distribution. The results are visible in the lower part of Fig. 8.3 and show that the assumption errors in the 3rd and 4th moments tend to be smaller than the corresponding estimation errors.

As demonstrated by both the t-test methodology and the leakage certification process, the NEON-based implementations on ARM Cortex-A8 are likely to be subject to order reduction and may require further hardening to prevent dependencies between shares. The potential causes of the order reduction remain unexplored since they may stem from bus/register/memory transitions, pipelined data processing or even electrical coupling effects. Pinpointing the origin of the security reduction remains an open problem in the side-channel field since it essentially requires the countermeasure designer to access/modify the hardware architecture and chip layout, a task that is not possible with proprietary designs.

8.4.3 Information-Theoretic Evaluation

Having investigated the security order of the single-block 4-share AES implementation, we turn to the evaluation of its 8-share counterpart. The core feature of a masking scheme is the noise amplification stage. Assuming sufficient noise, it has been shown that the number of traces required for a successful attack grows exponentially w.r.t. the order $d-1$ [CJRR99]. As a result, the evaluation of the proposed 8-share implementation can be beyond the measurement capability of most evaluators. To tackle this issue, we will rely on an information-theoretic approach used by Standaert *et al.* and Journault *et al.* [JS17; SMY09; SVO+10], assisted by the bound-oriented works of Prouff and Rivain [PR13], Duc, Faust, and Standaert [DFS15], and Grosso and Standaert [GS18].

Analytically, we start with an unprotected (single-share) AES implementation and estimate the device/setup signal-to-noise ratio (SNR). We define the random variable S to correspond to the sensitive (key-dependent) intermediate values that we try to recover. Likewise, we define the random variable L to correspond to the time sample that exhibits high leakage (heuristically chosen as the sample with the highest t-test value). Subsequently, we profile Gaussian templates for all sensitive values s that are instances of variable S. In other words, we estimate $\hat{\Pr}[L|s] \sim \mathcal{N}(\hat{\mu}_s, \hat{\sigma}_s^2)$ for all s. Using the estimated moments, we compute the SNR as the ratio $\mathrm{Var}(\hat{\mu}_s)/\mathrm{E}(\hat{\sigma}_s^2)$, resulting in SNR ≈ 0.004. We continue to compute the Hypothetical Information (HI) which shows the amount of information leaked if the leakage is adequately represented by the estimated model $\hat{\Pr}$.

$$\mathrm{HI}(S;L) = \mathrm{H}(S) + \sum_{s\in\mathcal{S}} \Pr[s] \cdot \sum_{l\in L} \hat{\Pr}[l|s] \cdot \log_2 \hat{\Pr}[s|l],$$

$$\text{where } \hat{\Pr}[s|l] = \frac{\hat{\Pr}[l|s]}{\sum_{s^*\in\mathcal{S}} \hat{\Pr}[l|s^*]}.$$

To simplify the evaluation process, we employ the independent shares' leakage assumption so as to extrapolate the information of a single share to the information of a d-tuple of shares. Thus, in order to obtain the HI bounds for security orders 3 and 7, we raise $\mathrm{HI}(S;L)$ to the security order. In addition, the evaluator should take special consideration w.r.t. horizontal exploitation [BCPZ16; VGS14], which can be particularly hazardous, e. g., in the context of lengthy masked multiplications. To showcase such a scenario, we employ the bound of Prouff and Rivain [PR13], stating that the multiplication leakage is roughly $1.72d + 2.72$ times the leakage of a d-tuple of shares. The results of the information-theoretic evaluation are visible in Fig. 8.4.

Figure 8.4 assesses the performance of the proposed 8-share AES implementation, using information-theoretic bounds. The solid line shows the ideal masking performance, while the dashed line shows a conservative masking

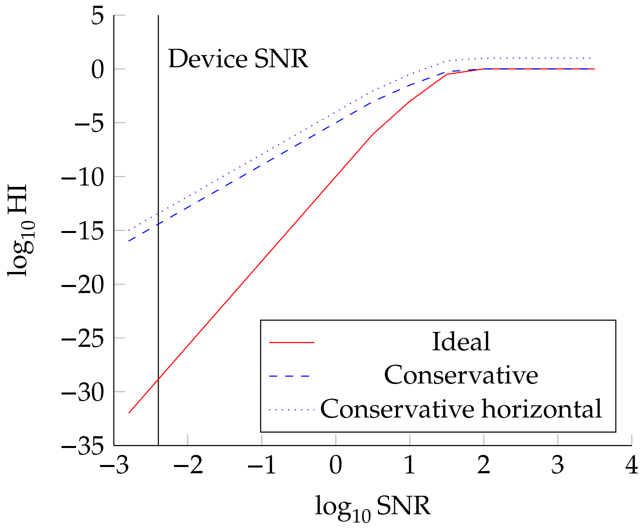


Figure 8.4: Information-theoretic evaluation of the 8-share implementation.

performance due to order reduction from order 7 to order 3. Last, the dotted line demonstrates the scenario where the adversary exploits the order-reduced (conservative) version in a horizontal fashion, i. e., (s)he incorporates all intermediate values computed during a masked AES multiplication. For the current SNR of the device, the measurement complexity is bounded by approximately 2^{91} measurements (ideal case), 2^{45} (conservative case) and 2^{42} (conservative horizontal case) [DFS15].

8.5 Conclusion and Outlook

We have shown how higher-order masking of AES can be sped up using NEON vector registers. With a good randomness source, such implementations are very fast and practical. We also performed a side-channel evaluation to study the security order of the single-block 4-share implementation and an information-theoretic analysis to bound the measurement complexity w.r.t. the 8-share implementation.

Future SCA work can delve deeper into order-reduction issues, in conjunction with multivariate and horizontal exploitation. For instance, with our high-order univariate methodology, it is implicitly assumed that all the shares are manipulated in parallel. While this appears to hold when looking at the NEON assembly specifications, full parallelism may not be enforced on a hardware level. A deeper inspection of the circuitry could potentially clarify the actual parallelism and identify the underlying issues behind order reduction. Moving toward multivariate exploitation, practical horizontal attacks such as soft-analytical attacks need to be carried out such that we can gauge in practice the detrimental effects of lengthy leaky computations and establish a fairer evaluation procedure.

Chapter 9
Reusing Randomness

Another approach to optimize masking countermeasures is to reduce the number of random bits that are required. This chapter explores what is possible with just two random bits that are carefully reused. In comparison to the original publication [GSD+19], the appendix has been merged into the text and minor textual edits have been made.

9.1 Introduction

Ever since the findings of Kocher, Jaffe, and Jun [KJJ99] on differential power analysis and Quisquater and Samyde [QS01] on electromagnetic emanation analysis, the efficient protection against so-called side-channel attacks has been eagerly studied. Over the years, masking has proven to be a countermeasure with high and well-understood security guarantees [DDF14; ISW03] as well as good scalability [CJRR99]. Despite its popularity, the research on more efficient approaches to mask security-critical implementations does not seem to come to an end soon [BGN+14; GIB18; GM17; GMK16; NRR06; RBN+15].

The lion's share of works on masking operate in the so-called t-probing model by Ishai, Sahai, and Wagner [ISW03]. In this model, an adversary is allowed to probe up to t intermediate values in an implementation. One has security against such an adversary if those t wires reveal no secret information. Despite the fact that this model has been shown to be insufficient in practice by several works [BGG+14; FGP+18; PV17], it remains the foundation for many new masking schemes.

One important drawback of most types of masking is their implementation costs, not least because of their high demand for fresh randomness. Since the creation of large amounts of fresh random bits requires additional time, chip area, energy, etc., a lot of research has been done on more randomness-efficient masking [BBP+16; BBP+17; BDCU17; BDF+17; FPS17; GD17; GM18; GMK16; Sug18]. Most of the existing work, however, focuses on the randomness optimization for specific masking gadgets, like masked `AND` gates, and do not consider the minimization of the overall randomness costs. An interesting result from prior work is the proof by Faust *et al.* [FPS17] that first-order masking with only one bit of randomness is impossible. They also demonstrated the theoretical possibility of masking with constant randomness cost.

Even more of the masking implementation papers only consider the so-called online randomness costs spent on producing fresh randomness to secure the computation once the initial sharing of the input data, e. g., plaintext, ciphertext, or data and key material, has been performed. There is, to the best of our knowledge, no paper that considers the minimization of randomness costs when taking the masking of the input data into account or that tries to minimize the overall randomness costs.

Our contribution. We start off this work in Section 9.2 by taking a step away from the modern sharing-based perspective of masking back to the classical Boolean masking perspective. From this masking point of view, we then demonstrate using linear transformations that first-order masking is theoretically possible with only two random one-bit masks. We then discuss what properties need to be fulfilled such that this approach also works for masked nonlinear transformations and show that existing approaches of masked `AND` gates do not fulfill these criteria. As a first practical contribution, we design a masked `AND` gate that allows reusing randomness from its inputs safely.

Based on our findings, we introduce in Section 9.3 a simple rule-based system. These rules can be encoded in SMT2 statements and they are then used to automatically check whether the masking approach is directly applicable to an unprotected implementation or if modifications (mask changes) are required.

Upon acceptance, our tool synthesizes a securely masked implementation for a given set of additional constraints like the used mask encoding.

We then show how our approach can be applied to larger implementations (Section 9.4) and demonstrate its feasibility and its impact on a full AES-128 encryption-only implementation in Section 9.4.3. With our approach, we successfully designed the first formally verified AES S-box design that requires only two random bits for the initial sharing of its inputs and requires no online randomness to achieve first-order security in the probing model. Even when going for a full AES implementation, the randomness requirements do not increase further. However, since existing formal tools are not yet efficient enough to digest a fully unrolled AES implementation, we instead verify each building block of our design using the maskVerif tool of Barthe *et al.* [BBC+19] for a predefined mask encoding of its inputs and outputs. Ensuring the same mask encoding for each input and output allows us to argue about the security when putting the components together in the full AES implementation. Details on the formal verification are given in Section 9.6.1. Finally, we discuss the limitations of the t-probing model for security in practice, as exemplified by our construction, in Section 9.6.3.

As a final contribution, we make our tool as well as our masking examples publicly available such that our findings are verifiable and future works can build upon them. They can be found at `https://github.com/LaurenDM/TwoRandomBits`.

9.2 Masking without Online Randomness

The goal of masking is to make the power consumption (and other side-channels related to the power consumption) independent of security-sensitive information. For this purpose, the security-sensitive information is first combined with uniformly random sampled data in an invertible masking function, such that the representation of the data itself becomes uniformly random distributed. In the case of Boolean masking, the sensitive information s, for instance, is combined with a random mask m by using the Boolean exclusive-or (`XOR`) operation. The resulting masked value $s_0 = s \oplus m_0$ thus becomes statistically independent of s, i. e., the mutual information between s and s_0 becomes zero. For this reason, any computation on s_0 trivially results in power consumption that is statistically independent of s as long as m_0 is not recombined with s_0.

Adversary model. The security of masked implementations is often expressed in the so-called t-probing model [ISW03] which assumes that an attacker can make up to t observations in the implementation (place up to t probes on the circuit). It has been verified in the past that this formal model also implies security against an attacker who employs differential side-channel analysis and who has access to noisy side-channel leakage traces [DDF14]. In the following we assume a first-order attacker, i. e., an attacker who can place a single probe on the device.

Sharing vs. masking. Often in the present literature, the relation between the masked data s_0 and the mask(s) m_0 is expressed using a sharing-based notation. For first-order masking (i. e., only one mask is used to protect s) the secret value is assumed to be split into two shares (e. g., s_0 and s_1) such that again the additive relation $s = s_0 \oplus s_1$ is fulfilled. While it is trivial to convert from a masking representation to a sharing representation by setting $s_0 = s \oplus m$ and $s_1 = m$, the sharing representation inherently hides the relation between secret information and masks.

For brevity reasons, we use the sharing-based representation in most parts of this chapter. Since in this work, we are particularly interested in the relation between secrets (or shares) and masks, we often switch to the masking form. To make the used notation clearer, we always use the prefix m for masks followed by a number in the subscript. Any other variable name with a suffix subscript number denotes a specific share of the variable. Most of the time we just use

0 or 1 in the subscript (e. g., a_0 or a_1) to refer to the first or second share of a first-order masked variable a, respectively. Without any subscript notation we always refer to the plain secret variable $(a, b, q, \dots)$.

9.2.1 Computation on Masked Data

To realize computations that are not only secure against side-channel analysis but also correct, the computed masked function needs to take the mask into account but in a way that does not unmask the data. For example, when calculating the `XOR` of two sensitive variables as $q = a \oplus b$, where a is shared in the two shares a_0 and a_1 and b is shared as b_0 and b_1, the correct and securely masked realization is trivial:

$$\begin{aligned} q_0 &= a_0 \oplus b_0 \\ q_1 &= a_1 \oplus b_1 \end{aligned} \tag{9.1}$$

With independent masks. When observing the masked representation of this equation with $a_0 = a \oplus m_0$, $a_1 = m_0$ and $b_0 = b \oplus m_1$, $b_1 = m_1$ the correctness can be easily observed when considering the addition of the shares of q, because both shares added together result in the desired operation in the sensitive variables a and b.

$$\begin{aligned} q &= q_0 \oplus q_1 \\ &= (a \oplus m_0) \oplus (b \oplus m_1) \oplus m_0 \oplus m_1 \\ &= a \oplus b \end{aligned}$$

To demonstrate the first-order security of the masked realization of the `XOR` in Eq. (9.1), it needs to be shown that each intermediate value (in this case only the output shares q_0 and q_1) is statistically independent of a and b. Statistical independence is given because we assume that each of the two masks m_0 and

m_1 is uniformly random and statistically independent of each other. By looking at the truth table for both shares of q in Table 9.1, one can observe that, when subdividing the truth table into the four possible combinations of values for a and b, the count of "1" appearances (or equivalently the Hamming weight of the truth table) for q_0 and q_1 in each case are equal.

Table 9.1: Truth table of the masked `XOR` from Eq. (9.1).

Shares				Secrets			TT	
a_0	a_1	b_0	b_1	a	b	$a \oplus b$	q_0	q_1
0	0	0	0	0	0	0	0	0
0	0	1	1				1	1
1	1	0	0				1	1
1	1	1	1				0	0
Hamming weight							**2**	**2**
0	0	0	1	0	1	1	0	1
0	0	1	0				1	0
1	1	0	1				1	0
1	1	1	0				0	1
Hamming weight							**2**	**2**
0	1	0	0	1	0	1	0	0
0	1	1	1				1	1
1	0	0	0				0	0
1	0	1	1				1	1
Hamming weight							**2**	**2**
0	1	0	1	1	1	0	0	1
0	1	1	0				1	0
1	0	0	1				0	1
1	0	1	0				1	0
Hamming weight							**2**	**2**

This equal distribution for each possible combination of secrets, results in power consumption that is on average equal for all cases of a and b. We note that an attacker with the ability to probe more signals could observe differences by combining multiple probed signals. Higher-order leakages, however, are more difficult to exploit than the average power consumption (exponentially more observations are required [CJRR99]) and are not considered in this chapter.

With equal masks. The situation changes when assuming that both masked variables use the same mask $m_0 = m_1$, which trivially reveals a and b in the equation of q_0.

$$q_0 = a_0 \oplus b_0 = (a \oplus m_0) \oplus (b \oplus m_0) = a \oplus b$$

Most state-of-the-art masking works assume that shares are produced using independent random masks which helps to avoid such situations. So when multiple `XOR` operations are chained together (e. g., $a \oplus b \oplus c \oplus \cdots \oplus z$) a lot of random masks are accumulated.

$$\begin{aligned}
q_0 &= a_0 \oplus b_0 \oplus c_0 \oplus \cdots \oplus z_0 \\
&= (a \oplus m_0) \oplus (b \oplus m_1) \oplus (c \oplus m_2) \oplus \ldots (z \oplus m_{25}) \\
q_1 &= a_1 \oplus b_1 \oplus c_1 \oplus \cdots \oplus z_1 \\
&= m_0 \oplus m_1 \oplus m_2 \oplus \cdots \oplus m_{25}
\end{aligned}$$

Please note that we assume here and in the remainder of this chapter that the masked equations are evaluated from left to right, and parentheses indicate atomic operations that do not produce further intermediate results (often to indicate the result of the evaluation of a sharing function or initial sharings). Our first and admittedly rather trivial observation is that the amount of accumulated randomness is unnecessarily high. One can realize the same function in a secure and correct shared way by simply alternating two random masks m_0 and m_1 in such a way that at no time an intermediate result is formed that depends on the secret value without a mask. One possible realization is to use m_0 to mask a and use m_1 for the remaining variables:

$$\begin{aligned} q_0 &= (a \oplus m_0) \oplus (b \oplus m_1) \oplus (c \oplus m_1) \oplus \ldots (z \oplus m_1) \\ &= a \oplus b \oplus c \oplus \cdots \oplus z \oplus m' \\ q_1 &= m_0 \oplus m_1 \oplus m_1 \oplus \cdots \oplus m_1 \\ &= m' \end{aligned}$$

where $m' = m_0$ if the number of inputs is odd (and thus the number of m_1 masks is even) and else $m' = m_0 \oplus m_1$.

This is only one example and there exist many other possible and secure realizations for this function. Depending on the mask assignments to the inputs, the resulting mask of the output can be either m_0 or m_1 or their combination $m_0 \oplus m_1$. With these findings, we can secure any linear function likewise. However, extending this to nonlinear functions is not straightforward.

9.2.2 Application to Nonlinear Gates

There exists a vast variety of first-order masked `AND` gates in the literature which form the simplest class of nonlinear functions and are used to construct more complex functions. These realizations of masked `AND` gates usually vary regarding online randomness requirements and the number of used input and output shares. The underlying functionality is of course always the same and, in the case of a realization with two shares, it requires the secure evaluation of four multiplication terms (where $\wedge$ represents a single `AND` operation):

$$\begin{aligned} q = a \wedge b &= (a_0 \oplus a_1) \wedge (b_0 \oplus b_1) \\ &= a_0 \wedge b_0 \oplus a_0 \wedge b_1 \oplus a_1 \wedge b_0 \oplus a_1 \wedge b_1 \end{aligned} \tag{9.2}$$

Any direct combination of either two multiplications terms (e. g., $a_0 \wedge b_0 \oplus a_0 \wedge b_1$) is insecure because it leads to a function that statistically depends on the secret a or b. Most of the existing masked `AND` gadgets thus use fresh random masks to realize the secure evaluation, like m_2 is used in the following example.

$$\begin{aligned} q_0 &= a_0 \wedge b_0 \oplus m_2 \oplus a_0 \wedge b_1 \\ q_1 &= a_1 \wedge b_0 \oplus m_2 \oplus a_1 \wedge b_1 \end{aligned} \tag{9.3}$$

This masked `AND` gate is indeed secure as long as the order of execution is from left to right and the masks including the ones used for sharing a and b are statistically independent and uniformly distributed. Another advantage of this realization is that it inherently refreshes the sharing which makes the result independent of a and b. Any linear or nonlinear combination of q with the sharing of a or b is thus still possible, as long as the transformation itself is secure under the assumption of independently shared inputs.

Without fresh randomness. There also exist realizations of a masked `AND` gate that do not require any fresh randomness. As an example, we consider the following equations from Biryukov *et al.* [BDCU17] where $\vee$ is the OR operation:

$$\begin{aligned} q_0 &= a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1) \\ q_1 &= a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1) \end{aligned} \tag{9.4}$$

A closer look at the properties of this realization from Biryukov *et al.* in Table 9.2 reveals that, while the masking itself is secure, a further (linear) combination with shares or combinations of shares from a and b (barring $a_0 \oplus a_1$) can make the sharing insecure again. Because this masked `AND` gate is insensitive to combinations with a single share from a (cf. column $q_0 \oplus a_0$ in Table 9.2), one could assume that q is similarly protected as an `XOR` gate is protected by the mask m_1 of b. The problem is that this masked `AND` gate behaves entirely different than the masked `XOR` gate from Eq. (9.1) or the masked `AND` from Eq. (9.3). For the output of a masked `XOR` gate where $q_0 = a \oplus b \oplus m_1$, we may assume that an `XOR` with m_0 followed by the addition of m_1 would result in a secure sharing masked by m_0, since $(a \oplus b \oplus m_1) \oplus m_0 \oplus m_1$ results in $a \oplus b \oplus m_0$. However, in

Table 9.2: Truth table for q_0 of Biryukov *et al.*'s masked `AND`.

Shares			Secrets	TT		
a_0	b_0	b_1	b	q_0	$q_0 \oplus a_0$	$(q_0 \oplus a_0) \oplus b_0$
0	0	0	0	1	1	1
0	1	1		0	0	1
1	0	0		1	0	0
1	1	1		0	1	0
Hamming weight				**2**	**2**	$\underline{\mathbf{2}}$
0	0	1	1	0	0	0
0	1	0		1	1	0
1	0	1		1	0	0
1	1	0		0	1	1
Hamming weight				**2**	**2**	$\underline{\mathbf{0}}$

case of the masked `AND` gate from Eq. (9.4), the `XOR` combination of the output q_0 with m_0 followed by another `XOR` with m_1 results in an insecure sharing (see different truth table Hamming weights for different cases of b in Table 9.2). Chaining of masked `AND` operations by carefully selecting (or changing) between two different masks is thus not possible with this masked `AND` gate.

9.2.3 Construction of a New Masked `AND`

We first transform the secure equations of Biryukov *et al.* such that we can directly observe what happens to the multiplication terms.

$$\begin{aligned}
q_0 &= a_0 \wedge b_0 \oplus (a_0 \vee \neg b_1) \\
&= a_0 \wedge b_0 \oplus \neg(\neg a_0 \wedge b_1) \\
&= a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1) \oplus 1 \\
q_1 &= a_1 \wedge b_0 \oplus (a_1 \vee \neg b_1) \\
&= a_1 \wedge b_0 \oplus \neg(\neg a_1 \wedge b_1) \\
&= a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1) \oplus 1
\end{aligned} \tag{9.5}$$

It can be verified that the terms $a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)$ from q_0 and $a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1)$ from q_1, considered separately, are securely masked by b_1 ($= m_1$, in the masking representation). Consider also the similarity with Eq. (9.3), but with m_2 replaced by b_1 in a similar fashion as so-called *correction terms* are used in threshold implementations [NRS11]. It is also interesting to note that these expressions correspond to multiplexer formulas: $a_0 \wedge b_0 \oplus (\neg a_0) \oplus b_1 = b_0$ if $a_0 = 0$, else b_1.

New construction. The design idea to ensure that the resulting sharing behaves similarly to the masked `XOR` gate is to securely combine all multiplication terms (Eq. (9.2)) in a single share of q, together with a single mask. However, adding q_0 and q_1 from Eq. (9.5) directly together is insecure because this results in $a \wedge b$ without any mask. We therefore first add a_1 ($= m_0$) to the second term (q_1) and then, both terms can be added without leaking information. The result (our new q_0) is only masked with a single mask m_0. To achieve correctness the second share (the new q_1) is set to m_0 (or equivalently a_1). This then results in the following masked `AND` gate:

$$\begin{aligned}
q_0 &= (a_0 \wedge b_0 \oplus (a_0 \wedge b_1 \oplus b_1)) \oplus ((a_1 \wedge b_0 \oplus (a_1 \wedge b_1 \oplus b_1)) \oplus a_1) \\
&= (a \wedge b) \oplus m_0 \\
q_1 &= a_1 = m_0
\end{aligned} \tag{9.6}$$

Further optimization. By closer observation of Eq. (9.6), we find that under given circumstances (possible mask configurations associated with the input shares), another optimization is possible. The truth table of the term $(a_1 \wedge b_1 \oplus b_1)) \oplus a_1$ of Eq. (9.6) is depicted in Table 9.3, which shows that it corresponds to a simple logical OR of the two input shares.

Table 9.3: Truth table of the equation $(a_1 \wedge b_1 \oplus b_1)) \oplus a_1$.

a_1	b_1	$= a_1 \vee b_1$
0	0	0
0	1	1
1	0	1
1	1	1

However, what is even more remarkable, is that when going through all possible valid mask configurations for the input shares (see Table 9.4), the term becomes a common constant $(m_0 \vee m_1)$ for all masked `AND` gates using the same masks. Please note that we define the second share of any variable (e. g., a_1, b_1) to carry only the mask information and never the secrets (a or b) in combination with a mask.

We thus write $m_0 \vee m_1$ as $[m_0 \vee m_1]$ in the resulting equation to denote that this is a term that only needs to be calculated once. The pratical implications become more evident in the implementation sections. With this optimization, Eq. (9.6) simplifies to Eq. (9.7) which saves one `AND` gate (for multiple occurences of masked `AND`s) and two `XOR`s for each masked `AND` gate.

$$\begin{aligned} q_0 &= \overbrace{(\underbrace{a_0 \wedge b_0}_{t_1} \oplus \underbrace{(a_0 \wedge b_1 \oplus b_1))}_{t_2}}^{t_3} \oplus \overbrace{(\underbrace{a_1 \wedge b_0}_{t_4} \oplus [m_0 \vee m_1])}^{t_5} \\ q_1 &= a_1 \end{aligned} \tag{9.7}$$

Table 9.4: Possible mask configurations for the input shares a_1 and b_1.

a_1	b_1	$a_1 \vee b_1$
m_0	m_0	invalid
m_0	m_1	$= m_0 \vee m_1$
m_0	$(m_0 \oplus m_1)$	$= m_0 \vee m_1$
m_1	m_0	$= m_0 \vee m_1$
m_1	m_1	invalid
m_1	$(m_0 \oplus m_1)$	$= m_0 \vee m_1$
$(m_0 \oplus m_1)$	m_0	$= m_0 \vee m_1$
$(m_0 \oplus m_1)$	m_1	$= m_0 \vee m_1$
$(m_0 \oplus m_1)$	$(m_0 \oplus m_1)$	invalid

Security. The security of the masked `AND` gate can be easily verified by hand as shown in Table 9.5 where t_i values denote intermediate results. We again record all possible input share combinations in a truth table and sort them by the unshared secrets a and b. For each possible intermediate (t_1 to t_5, and q_0), we count the number of ones in the truth tables per secret value for a and b (Hamming weight). If the truth table Hamming weights of t_i (resp. q_0) are identical for each secret, then the probability distribution of t_i (resp. q_0) is independent of the secret. Table 9.5 clearly shows that this is the case; Hence, a first-order attacker does not gain any sensitive information by probing either one of the intermediates.

In addition to the manual inspection of the masked `AND` gate, we also performed a formal verification by using the tools by Bloem *et al.* [BGI+18] and Barthe *et al.* [BBC+19] which gave us the same results. Furthermore, we did the same verification for the composition of the `AND` gate with an `XOR` ($q \oplus b$) and with another `AND` ($q \wedge b$) as can be seen in Tables 9.6 and 9.7. The code that can be used as input for maskVerif [BBC+19] can be found

Table 9.5: Security of the masked `AND` from Eq. (9.7).

Shares				Secrets			TT					
a_0	a_1	b_0	b_1	a	b	$a \wedge b$	t_1	t_2	t_3	t_4	t_5	q_0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1				0	1	1	0	1	0
1	1	0	0				0	0	0	0	1	1
1	1	1	1				1	0	1	1	0	1
Hamming weight							**1**	**1**	**2**	**1**	**2**	**2**
0	0	0	1	0	1	0	0	1	1	0	1	0
0	0	1	0				0	0	0	0	0	0
1	1	0	1				0	0	0	0	1	1
1	1	1	0				1	0	1	1	0	1
Hamming weight							**1**	**1**	**2**	**1**	**2**	**2**
0	1	0	0	1	0	0	0	0	0	0	1	1
0	1	1	1				0	1	1	1	0	1
1	0	0	0				0	0	0	0	0	0
1	0	1	1				1	0	1	0	1	0
Hamming weight							**1**	**1**	**2**	**1**	**2**	**2**
0	1	0	1	1	1	1	0	1	1	0	1	0
0	1	1	0				0	0	0	1	0	0
1	0	0	1				0	0	0	0	1	1
1	0	1	0				1	0	1	0	0	1
Hamming weight							**1**	**1**	**2**	**1**	**2**	**2**

at `https://github.com/LaurenDM/TwoRandomBits`. We note that for secure composition with the other input (a), the roles of a and b should be switched in Eq. (9.7).

Note that in Table 9.6 there is an imbalance when the output share q_0 is combined with input share a_0, but not when first `XOR`ed with b_0. This is normal, as the `AND` gate treats inputs a and b asymmetrically. The `AND` gate in Eq. (9.7) reuses the mask of a (m_0) and can therefore not be freely composed with a. By switching the roles of a and b in Eq. (9.7), one obtains the `AND` gate that

Table 9.6: Security of the masked `AND` from Eq. (9.7) composed with `XOR`.

Shares				Secrets			TT		
a_0	a_1	b_0	b_1	a	b	$a \wedge b$	$q_0 \oplus a_0$	$q_0 \oplus b_0$	$(q_0 \oplus b_0) \oplus a_0$
0	0	0	0	0	0	0	0	0	0
0	0	1	1				0	1	1
1	1	0	0				0	1	0
1	1	1	1				0	0	1
Hamming weight							**0**	**2**	**2**
0	0	0	1	0	1	0	0	0	0
0	0	1	0				0	1	1
1	1	0	1				0	1	0
1	1	1	0				0	0	1
Hamming weight							**4**	**2**	**2**
0	1	0	0	1	0	0	1	1	1
0	1	1	1				1	0	0
1	0	0	0				1	0	1
1	0	1	1				1	1	0
Hamming weight							**0**	**2**	**2**
0	1	0	1	1	1	1	0	0	0
0	1	1	0				0	1	1
1	0	0	1				0	1	0
1	0	1	0				0	0	1
Hamming weight							**0**	**2**	**2**

can be composed with a but not with b. These composition rules may seem complicated, but the tool of Section 9.3 automatically creates circuits that satisfy them.

By combining the findings for the `XOR` and the `AND` gates we can mask arbitrary implementations, and as we will show in the next section, we can also derive simple rules to synthesize securely masked implementations from unprotected ones.

Table 9.7: Security of the masked `AND` from Eq. (9.7) composed with `AND` as in Eq. (9.8).

Shares						Secrets				TT					
a_0	a_1	b_0	b_1	q_0	q_1	a	b	q	$q \wedge b$	t'_1	t'_2	t'_3	t'_4	t'_5	q'_0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0					0	1	1	0	1	0
1	1	0	0	1	1					0	0	0	0	1	1
1	1	1	1	1	1					1	0	1	1	0	1
Hamming weight										**1**	**1**	**2**	**1**	**2**	**2**
0	0	0	1	0	0	0	1	0	0	0	1	1	0	1	0
0	0	1	0	0	0					0	0	0	0	0	0
1	1	0	1	1	1					0	0	0	0	1	1
1	1	1	0	1	1					1	0	1	1	0	1
Hamming weight										**1**	**1**	**2**	**1**	**2**	**2**
0	1	0	0	1	1	1	0	0	0	0	0	0	0	1	1
0	1	1	1	1	1					1	0	1	1	0	1
1	0	0	0	0	0					0	0	0	0	0	0
1	0	1	1	0	0					0	1	1	0	1	0
Hamming weight										**1**	**1**	**2**	**1**	**2**	**2**
0	1	0	1	0	1	1	1	1	1	0	1	1	0	1	0
0	1	1	0	0	1					0	0	0	1	0	0
1	0	0	1	1	0					0	0	0	0	1	1
1	0	1	0	1	0					1	0	1	0	0	1
Hamming weight										**1**	**1**	**2**	**1**	**2**	**2**

$$
\begin{aligned}
q'_0 &= \overbrace{(\underbrace{q_0 \wedge b_0}_{t'_1} \oplus \underbrace{(q_0 \wedge b_1 \oplus b_1)}_{t'_2})}^{t'_3} \oplus \overbrace{(\underbrace{q_1 \wedge b_0}_{t'_4} \oplus [m_0 \vee m_1])}^{t'_5} \\
q'_1 &= q_1
\end{aligned}
\tag{9.8}
$$

9.3 Synthesis of First-Order Secure Implementations

Manually tracking the masks as they propagate through the implementations quickly becomes a very complex task as the implementation size increases. We thus decided to develop an automated approach to create a masked implementation when possible, or to indicate which signals need to be changed otherwise. As a first step, the tool reads the description of a Boolean program in static single assignment (SSA) form in Verilog syntax such that each instruction is either a one-bit signal assignment or a two-bit `XOR`, `XNOR`, or `AND` gate. The Boolean circuit is then represented as an SMT problem which is fed to the Z3 [MB08] theorem prover. Z3 searches for a possible solution for the mask encoding of the input signals so that for each gate the inputs have different masks. Furthermore, it allows ensuring a desired mask encoding for the input and output signals. We now give a more detailed description of how the implementation is encoded in SMT2 and which steps are necessary.

Each implementation takes two masks m_0 and m_1. As a result, there are three possible mask combinations and thus three possible encodings for the input signals: $1 = m_0$, $2 = m_1$, $3 = m_0 \oplus m_1$. With the following SMT2 code snippet, the input signal a is mapped to any of the three masking combinations. We adjust the assertions accordingly, depending on whether we target a specific encoding or we let the theorem prover decide on the encoding.

```
; Input encoding definition and constraints
(declare-const a Int)
(assert (> a 0))
(assert (< a 4))
```

The same rules are also applied to every output of a gate to restrict the output mask encoding to these three possibilities.

For each of the four possible instruction classes (assignment, `XOR`, `XNOR`, and `AND`) of the SSA-encoded input file, we create specific rules for deciding which masks can appear in the output q for the given input combination. In the example encoding it is always assumed that the signals a and optionally b form the operands. The encoding of the signal assignment $q = a$ just results in a copy of the mask encoding in the SMT2 rules.

```
; Signal assignment rule
(assert (= q a))
```

To encode the output of the `XOR` (and `XNOR`) instructions, we utilize the fact that for different input encodings of a and b, the output encoding (calculated by an `XOR`) is always different from the input encodings and equal to the third unused one.

$$\begin{aligned} m_0 \oplus m_1 &= (m_0 \oplus m_1) \\ m_0 \oplus (m_0 \oplus m_1) &= m_1 \\ m_1 \oplus (m_0 \oplus m_1) &= m_0 \\ &\dots \end{aligned}$$

When neglecting the case that both inputs could have the same mask encoding, which is covered by the safety rules for the gates in the next step, the following SMT2 encoding can be used.

```
; XOR/XNOR gate rule
(assert (not (or (= q a) (= q b))))
```

Note that the negation of the output in case of the `XNOR` has no influence on the encoding because it is a simple addition of a constant value 1.

Finally, for the `AND` gate, the mask encoding can be the same as either operand since the operands can be simply swapped. We thus let the theorem prover decide which signal is used as the first operand and this defines the mask encoding of the output (see Eq. (9.7)). The information of which masks appear in the output is later on taken into account when the masked implementation is created to decide on the first operand.

```
; AND gate rule
(assert (or (= q a) (= q b)))
```

Again the `AND` gate rule does not cover the cases of both operands having the same mask encoding.

For each two-input gate, we additionally define that both operands are required to have a different mask encoding which otherwise would create a flaw in the masked implementation.

```
; Safety rule for two input gates
(assert (not (= a b)))
```

To make the design and verification of separate modules easier, we decide to use the same input and output mask encoding on byte-level for all our modules. We can restrict the output encoding by setting the input and output signals equal, for example.

```
; Equal input and output byte-encoding
(assert (= o0 i0))
(assert (= o1 i1))
(assert (= o2 i2))
...
(assert (= o7 i7))
```

When the Z3 theorem solver finds a secure model that fulfills our constraints, it constructs the mask assignments for a masked implementation. The translation of the unprotected scheme to a secure masked implementation is then rather straightforward. At first, we duplicate all input and output ports of the module and additionally add the two masks m_0 and m_1 as input signals. For each instruction of the SSA input file we replace the original code by its masked variant according to the masked gates introduced in Section 9.2. As a further optimization, the second share of each instruction is (optionally) replaced by the resulting mask of the output signal which helps to save unnecessary instructions that would result in one of the three mask encodings anyway.

We do not give a more detailed description of our tool at this point since the rest of the functionality follows from the description of the masked gates above and is mostly engineering work.

9.4 Masking AES

To demonstrate the practicality of our approach, we target AES-128 (forward direction only) as an example. Since none of the existing formal verification tools are yet powerful enough to verify a full AES encryption, we decide to use a modular implementation and verification approach. To justify the security of the overall design when bringing the modules together, we restrict the mask encoding for each input and output byte of every function to be equal.

Our software implementation is partially based on earlier work described in Chapter 6 and in [SS16c]. There we describe various optimized assembly implementations targeting the 32-bit ARM Cortex-M3 and Cortex-M4 microprocessors. One implementation is masked using 2 Boolean shares. This is a bitsliced implementation of AES-128 in CTR mode, such that two consecutive AES blocks can be efficiently processed in parallel. When 256 blocks (or 4 kilobyte) of data are encrypted, we measured that encryption on average takes 8727 cycles per block (or 545 cycles per byte). We also noted that 3440 cycles of these are spent on generating all required 10 496 random bits using the onboard hardware RNG of an STM32F407 board, which is over 39% of the total cycle count.

Our implementation uses the same hardware RNG of the same board, but we only generate a single 32-bit fresh random word per 2 AES blocks. The architecture dictates a multiple of 32 bits, so 28 of these are ignored, 2 are used for the first AES block, and 2 for the second AES block.

9.4.1 SubBytes

The most complicated part of the AES is its SubBytes layer which can be implemented as 16 instances of S-box modules. Most of the masked AES designs published over the last years are based on the S-box construction of Canright [Can05]. A more suitable design for our bit-wise approach, however, is the design of Boyar and Peralta [BP12] which is already constructed in SSA form. There are follow-up works [BMP13; VSP18] that further reduce the size of the implementation in terms of gates/instructions. Various unmasked S-box implementations can be found online.[1] For hardware implementations, we would recommend the S-box implementation which aims at minimizing the logic depth (16). This S-box consists of 128 SSA instructions. In total there are 34 `AND`, 90 `XOR` and 4 `XNOR` instructions for the unmasked implementation. Each instruction takes two one-bit variables as input. For our case (a software implementation), the logic depth is not of such importance, so we choose the S-box with the smallest gate count (113). This S-box has 32 `AND`, 77 `XOR` and 4 `XNOR` instructions and has logic depth 27.

After running our synthesis tool on this S-box design without any further optimizations, the resulting masked design consists of 96 `AND` gates, 228 `XOR` gates, and 4 `NOT` gates (because `XNOR`s are decomposed to one `XOR` followed by a `NOT` gate in Yosys' ILANG). The 96 `AND` gates result from the fact that the masked `AND` triples the number of `AND` gates compared to the unmasked design. Also, each masked `AND` gate introduces 4 `XOR` gates which in total results in 128 additional `XOR` gates. The masking of the `XOR` and `XNOR` gates, on the other hand, does not introduce additional circuitry since the second output share can simply be assigned to the third mask (i. e., unused by the inputs). Some additional `XOR` gates are required because at some points we need to change the masking of a signal by introducing additional `XOR` instructions to receive a satisfiable Z3

---

[1] `http://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html`

model and thus a securely synthesizable implementation, and to ensure that the input and output mask encoding is equal.

After running an optimization pass in Yosys, which maps gates implementing the same function to a single gate and thus eliminates duplications, the number of gates could be reduced to 86 `AND` gates, 1 `OR` gate, 225 `XOR` gates and 4 `NOT` gates. We rerun the verification after this optimization to ensure that the implementation remains secure. The `NOT` gates can be moved to the key schedule such that they are not executed for every encryption/decryption call with the same key. The total overhead for the masking of the S-box is thus about a factor 2.79 excluding loads and stores.

From the design of the S-box, we also gathered a byte encoding to be used for the rest of the AES modules to ensure security and correctness of the full encryption. The byte encoding is $\{2, 3, 3, 1, 1, 2, 1, 2\}$ in the SMT encoding, corresponding to $\{m_1, m_0 \oplus m_1, m_0 \oplus m_1, m_0, m_0, m_1, m_0, m_1\}$ as the mask encoding for the S-box input bits i_0 to i_7.

The targeted microarchitecture has only 14 registers that can be freely used, which means that many store and load instructions need to be inserted. On this platform, loads from memory are relatively expensive. Most arithmetic instructions execute in a single cycle, while a load instruction will take at least two cycles, although when N loads can be pipelined they can often execute in $N + 1$ cycles. In earlier work [Sto16a], a tool was created that automatically reschedules instructions and allocates registers in order to minimize the overhead caused by spilling values to the stack. We use the same tool to schedule the 60 load instructions and 51 store instructions on top of the arithmetic instructions.

9.4.2 Linear Components

ShiftRows. Since the ShiftRows transformation only changes the order of the bytes in the state rows, no special modifications are required for the round transformation compared to an unprotected design. Furthermore, all of the masked AES modules (in the final design) do not explicitly carry the second share of each signal but instead just assume the byte mask encoding as used by the S-box for inputs and outputs as the second share. ShiftRows can be implemented with only wiring in hardware, but in software we need instructions to actually move the bits around. We use the same ShiftRows implementation as that of Chapter 6, which requires 104 single-cycle instructions.

MixColumns. We start with the unmasked MixColumns implementation from Chapter 6, that uses 27 32-bit `XOR`s for the four-column operation. However, to translate this to a masked implementation we now need to be careful that no values with the same mask get combined and ensure the correct byte mask encoding of the inputs and outputs. We therefore need to remask various values and the masks need to be loaded from the stack. In total we need to add 12 `XOR` and 6 load instructions to the original 27 `XOR`s. A program was used to verify that this was done correctly.

AddRoundKey. In AddRoundKey the round key is added to the state (or the plaintext in the first round). Since we enforce the same byte encoding for all bytes in our design, the key byte first needs to be remasked before it can be added to the state or plaintext byte and the result of the `XOR` also requires remasking. Instead of 8 32-bit `XOR`s as in the unprotected bitsliced design, we thus require 32 `XOR` instructions. The number of `XOR`s can be reduced to only sixteen if the state and key bytes are shared such that their sum again results in the assumed byte sharing for the S-box. Due to the lack of the possibility to formally verify the whole implementation, we decide on keeping the byte encoding for both key and state bytes the same. This makes arguing of the security of the overall implementation for individually verified modules easier because there are fewer issues to oversee. In addition to the arithmetic instructions, we use 11 load instructions and 1 store.

9.4.3 Results

For the entire AES encryption, we measure on average 3387.6 cycles per block (or 211.7 cycles per byte) under the exact same test conditions as in Chapter 6. This is a speed improvement of roughly 61%. Moreover, the stack requirements are lowered from 1584 bytes to only 188 bytes, a decrease of over 88%.

In Chapter 6 we also described an unmasked bitsliced AES-128-CTR implementation as an intermediate step. This took only 1616.6 cycles per block (or 101 cycles per byte) on the Cortex-M4. The overhead cost of adding first-order masking is therefore still almost a factor 2.1.

The most computation effort in each round is spent on the S-box calculations with 68.7% of the instructions. The AddRoundKey and MixColumns operations consume respectively 7.1% and 7.3% of the round instructions. The remaining instructions are mostly spent on the ShiftRows transformation with about 16.9%.

Table 9.8 summarizes the costs for the individual transformations that are required to implement the full AES.

9.5 Discussion

9.5.1 Comparison to Previous Work

A comprehensive comparison of our results with previous work is very difficult. On the one hand, masked AES implementations have been created for many different platforms. On the other hand, a number of works only report the speed which means that comparing based on other aspects such as memory usage is hard, especially when source code is not easily available. Moreover, the reported speeds were measured on different CPU (micro)architectures, which makes them harder to compare. For masked implementations, rather than comparing absolute execution times, it makes more sense to compare the overhead factor over an unprotected implementation, as was done in [WVGX15]. However, we were not able to find implementation results for unmasked implementations

Table 9.8: AES-128-CTR implementation results in terms of instruction counts.

Module	#	`AND`	`XOR`	Bit op*	Load	Store
AES†		870	3 433	560	773	520
PreRound	1	-	32	-	8	1
► AddRoundKey‡	1	-	32	-	8	1
Round	9	87	344	56	77	52
► SubBytes	1	87	225	-	60	51
► ShiftRows	1	-	48	56	-	-
► MixColumns	1	-	39	-	6	-
► AddRoundKey‡	1	-	32	-	11	1
LastRound	1	87	305	56	72	51
► SubBytes	1	87	225	-	60	51
► ShiftRows	1	-	48	56	-	-
► AddRoundKey‡	1	-	32	-	12	-

[*] This includes `ubfx` and `uxtb` instructions.

[†] This excludes the function prologue and epilogue, reading a random word and changing it to the right format, bitslicing the input, unbitslicing the output, loading the input, storing the output, increasing the CTR-mode counter, the initial masking of the input, the final unmasking of the output, `XOR`ing the keystream with the plaintext, keeping track of the remaining length, and managing some pointers and addresses. Of course all of this is included in the speed measurement.

[‡] Loads and stores slightly differ due to values that are already in registers or no longer necessary.

in all works. Furthermore, none of the previous works consider the cost of the initial masking of plaintext and key. Finally, it should also be noted that a lot of these works also present higher-order masking schemes, which again makes comparison with our optimized first-order AES unreasonable. Nevertheless, we gathered some data in Table 9.9 for completeness.

Performance. The only previous work to which a comparison is justified is the work underlying Chapter 6 [SS16c], since we used the same platform. When comparing to that work, we can conclude that our dramatic decrease of randomness does not imply sacrifices when it comes to speed or memory.

When it comes to speed, we also look at the work of Wang *et al.* [WVGX15], since it was able to create a significant improvement over previous works. By comparing the overhead factors over an unprotected implementation, we can conclude that the speed of our implementation is competitive.

It should be noted that like the implementations by Goudarzi *et al.* [GR17], we do not include a masked key schedule but instead store the precomputed round keys in memory in shared form.

Finally, we point out that we compare the average encryption speed per AES block, but as our implementation always processes two blocks in parallel using CTR mode, one cannot reach this speed when encrypting only a single AES block. Moreover, the implementation is fully unrolled, which results in very high speed and a large ROM requirement. If the ROM size is deemed too high for a particular application, it would be trivial to drastically reduce it at the cost of only a few extra CPU cycles.

Randomness. When it comes to total randomness consumption, our implementation clearly outshines most of the previous works. Only Faust *et al.* [FPS17] had previously reported a first-order AES with constant randomness. In fact, they were the first to show its feasibility. However, their complete AES implementation is not discussed in detail. For example, it is not clear if the 8 S-box input bits also use the same masks and if this is the case, which encoding was used. Although they use the same development board (including the hardware RNG), it should be noted that their implementation was written in C and more meant as a proof of concept than as a carefully optimized implementation.

Security. Apart from differences in speed, memory usage and randomness consumption, the implementations in Table 9.9 naturally also differ in level of practical security. A low fresh randomness consumption goes hand-in-hand with higher signal-to-noise ratio, which can benefit the adversary. It has also been noted that the reuse of randomness leads to dangerous transition leakage [WM18b]. On the other hand, transition leakages have also been shown to be a problem in the more conventional randomness-expensive masked implementations [PV17], which means our implementation is not necessarily the only one vulnerable to this. Moreover, even if we double our latency to account for reset cycles against transition leakage, our performance is very competitive with previous works.

9.5.2 Randomness in Perspective

Offline. Our design requires no online randomness and only two random bits for the initial sharing. For other implementations in the literature at least 128 bits for the sharing of the key and 128 bits for the sharing of the plaintext are required for a two-share implementation, or 2×256 bits for a three-share threshold implementation, respectively. From this perspective, our implementation saves at least 254 random bits for the initial sharing alone.

Online. In the current state-of-the-art on first-order masking, each multiplication and refreshing block requires one unit of fresh randomness. In the case of the AES S-box, each unit is one byte and the S-box can be constructed using four multiplications and two refreshings [RP10], which brings the total randomness

Table 9.9: Comparing performance results for AES-128.

	Platform	Speed (cycles)	Overhead factor	ROM (bytes)	RAM (bytes)	Random (bits)
		Comparable Platform				
This work	Cortex-M4	3 387.6	2.1	25.2k	188	2
Chapter 6	Cortex-M4	8 727.6	5.4	39.9k	2.0k	10.5k
[FPS17]	Cortex-M4	73 650	-	-	-	2/16
		Different Platforms				
[RP10]	8-bit 8051	129 000	64.5	3.2k	73	9.6k
[BFG+17]	8-bit AVR	157 196	-	2.8k (in total)		13.1k
[BFG+17]	8-bit AVR	73 769	-	1.8k (in total)		11.5k
[GR17]	ARM7TDMI	53 462	-	7.5k	-	30.8k
[GR17]	ARM7TDMI	49 329	-	4.8k	-	26.9k
[GR17]	ARM7TDMI	56 199	-	12.4k	-	19.2k
[WVGX15]	Cortex-A15 simulator	4 869	4.3	-	-	19.2k

cost per S-box evaluation to 48 bits. One SubBytes transformation consists of 16 S-boxes and thus requires 768 random bits. One encryption round including key schedule requires 960 bits. In total, the amount of online randomness for one AES-128 encryption is thus 9.6 kbits.

For the sake of completeness, we note that in hardware masking, there exist more online randomness efficient S-box implementations which, however, require an increased amount of input shares for the S-box (e. g., the four-share S-boxes of Ghoshal *et al.* [GD17] and Wegener *et al.* [WM18a]). There is no full AES implementation or estimation given in [GD17], so further comparison is difficult. Moreover, a flaw in their design was detected and reported by Wegener and Moradi [WM18a]. The S-box of [WM18a] also has four input and output shares and exploits the changing of the guards trick by Daemen [Dae17] to obtain zero online randomness consumption. Their full design uses the dynamic conversion approach, but avoids extra online randomness by a clever recycling of independent state bytes. They thus only require 256 bits for the initial sharing of the plaintext and key and 24 additional bits for the initial guards. More recently, Sugawara [Sug18] presented the first three-share AES S-box without online randomness. However, their entire AES implementation still uses 776 initial bits of randomness, which exceeds that of [WM18a].

Summary. Comparing our design with others is quite difficult because most of the existing implementations do not consider the amount of required initial randomness for sharing the key and plaintext data. However, we have at least shown that also the performance can be competitive with state-of-the-art implementations, even if we double the latency with reset cycles against transitional leakages. Requiring only two bits of randomness for each masked encryption could thus make the difference between deciding on requiring an additional PRNG or using an already on-board TRNG, and could thus make first-order masking cheap enough to be used for highly constrained devices like low-cost RFID tags.

9.5.3 Hardware

For a SCA-resistant implementation in hardware, security in the probing model with glitches needs to be ensured. The security of our approach critically depends on the correct order in which the signals are combined. For this reason, registers are required after t_1, t_2, t_3, t_4 and t_5.

The above has been formally verified in the presence of glitches using maskVerif [BBC+19].[2] Hence, our methodology is also applicable to hardware masking.

Efficiency. In practice however, the method is more amenable for application in software. The need for many registers means we pay for the randomness reduction by an increased amount of latency. The unmasked Boyar-Peralta S-box has a maximum logic depth of 16 and an `AND` depth of 4. Every masked `AND` gate requires three cycles to securely calculate the result. Accordingly, the total latency of the S-box is 12 cycles ($16 - 4$ `XOR` layers) plus 12 (4 `AND` layers $\times 3$) which in total amounts to 24 cycles. In software, the impact of our masking method on the latency and throughput is less dramatic, both because of the absence of glitches and because of the possibility of bitslicing.

Security. Moreover, most hardware masked AES implementations are round-based with a serial S-box calculation. As was shown by Wegener and Moradi [WM18b], these serialized designs can introduce dangerous transition leakages. So, in contrast with an unrolled implementation, these designs require register reset cycles or precharge logic, which would essentially again increase the latency to its double. For these reasons, we do not investigate a hardware implementation in further detail. We further discuss the security issues in Section 9.6.3.

To conclude, we have demonstrated that two random bits are enough to achieve theoretical first-order security in the probing model even in the presence of glitches. Its implementation in practice however incurs a high penalty in latency and requires extra care to not introduce new leakages.

---

[2] The results can also be found at `https://github.com/LaurenDM/TwoRandomBits`.

9.6 Security Analysis

9.6.1 Formal Verification in the t-Probing Model

For the verification of the side-channel security of our approach, we used the formal verification tool maskVerif of Barthe *et al.* [BBC+19] on the synthesized modules. Since maskVerif is originally designed to verify sharing-based implementations, the outcome of our synthesis tool creates a verification wrapper that is later on modified to represent the correct masking for the input signals of the actual masked implementation. The verification wrapper thus takes two shares per input of the masked module and creates the correct masking by first adding the mask as defined by the mask encoding and subsequently the second share of the input.

The input bits $\{i_0, i_1, \dots i_7\}$ of the masked AES S-box module for example, uses the same SMT mask encoding $\{2, 3, 3, 1, 1, 2, 1, 2\}$ (where 1 denotes m_0, 2 denotes m_1, and 3 denotes $m_0 \oplus m_1$) as any other module for both inputs and outputs. We take the input shares of the wrapper (indicated by the suffix "_0" for the first share or "_1" for the second share) and create the actual masking as follows.

```
module verification_wrapper(
    input   i0_0, i1_0, ..., i7_0,
    input   i0_1, i1_1, ..., i7_1,
    input   m0, m1,
    output  o0, o1, ..., o7);

    // Mask encoding
    assign i0 = (i0_0 ^ m1)      ^ i0_1; // 2
    assign i1 = (i1_0 ^ m0 ^ m1) ^ i1_1; // 3
    assign i2 = (i2_0 ^ m0 ^ m1) ^ i2_1; // 3
    assign i3 = (i3_0 ^ m0)      ^ i3_1; // 1
    assign i4 = (i4_0 ^ m0)      ^ i4_1; // 1
    assign i5 = (i5_0 ^ m1)      ^ i5_1; // 2
    assign i6 = (i6_0 ^ m0)      ^ i6_1; // 1
```

```
    assign i7 = (i7_0 ^ m1)       ^ i7_1; // 2

    // DUV
    aes_sbox sbox_inst(i0, i1, i2, ..., i7, ...);
endmodule
```

For the input in the maskVerif tool, the implementation is read by the Yosys [Wol] open synthesis tool. The circuit is then mapped to Yosys' internal gate representation (ILANG) and subsequently flattened such that a single module is created that contains all gates. The resulting circuit is then returned in ILANG format for which input, output and mask signals are annotated before it is fed into maskVerif. The implementations are validated for the probing model of Ishai *et al*. [ISW03] without glitches.

Table 9.10: SCA resistance verification results.

Module	Number of 1-uples	Verification time	Result
AddByte	95	16 ms	probing secure
MixColumns	315	108 ms	probing secure
SubByte	429	22 s	probing secure

As the results in Table 9.10 show, all of the modules on which our entire AES-128 encryption depends, are probing secure as intended. ShiftRows is only rewiring (readdressing) in hardware and just a bit permutation in software, which does not influence the probing security. With the input and output constraints for our synthesis tool, we also ensure that the mask encoding for each byte is the same, and we can thus safely compose these modules without creating flaws in the probing model for first-orders. However, we note that this composition argument is only true for first-order implementations for which a probing attacker is restricted to a single probe. This means that multivariate probes are of no concern and thus probes occur only in a single submodule. Tables 9.6 and 9.7 show that the reuse of randomness has no influence on the output distributions of cascaded gates, as long as the mask encoding is done with precision. Our synthesis tool creates implementations which, by construction, ensure that the mask encoding is fixed at the inputs and outputs of submodules. Our submodules have been formally verified for these encodings. Therefore, combined with the fact that probes can only placed on a single submodule, this ensures that the entire AES implementation is first-order secure.

We have proven the security of our scheme using formal verification tools and demonstrated that randomness can almost completely be eliminated for first-order security within the t-probing model. Apart from pushing the boundaries in terms of randomness cost, we are therefore also testing the limits of this model, which has become the standard adversary model for countermeasures against DPA. In the next two subsections, we demonstrate with our new masking scheme where the t-probing model is lacking.

9.6.2 Horizontal Attacks

With our scheme that fixes the mask encoding of the state across the rounds of an encryption, we need to be careful not to create a vulnerability to horizontal attacks. A horizontal side-channel attack considers correlations between multiple samples within a single trace, as opposed to the more common vertical side-channel attacks, which consider the same time sample across multiple traces. For this investigation, we cannot rely on established evaluation methods (e. g., TVLA) or verification tools since the state-of-the-art on horizontal attacks against symmetric primitives is quite limited. We will first consider the attacks from previous works and explain why they cannot be applied to our new scheme. Next, we use simulated traces to investigate the success probability of a trivial horizontal attack to recover the masks, followed by a classical CPA attack.

Previous works. The literature on horizontal attacks against public-key primitives is abundant [BJPW13; BJPW14; HKT15]. However, these attacks are typically based on the assumption that the secret determines the presence or absence of some collision(s) (e. g., between subsequent iterations in an exponentiation algorithm). Since this situation clearly differs from ours, we will not further discuss it.

A recent work investigates horizontal attacks against a very common building block in masked implementations, i. e., the ISW multiplication [BCPZ16]. However, we will not consider such an attack in this work, since it targets the ISW implementations when the number of shares n satisfies $n > c \cdot \sigma$, with σ the measurements noise. As an example, they attack a 21-share implementation. It is thus unlikely to be applicable to our 2-share implementation.

One type of horizontal attack against masked symmetric primitives targets implementations that compute the S-box by means of a masked table lookup [PdL09]. They exploit the fact that for each S-box input (i. e., one plaintext byte and one key byte), the table lookup is performed with multiple masks. Hence, a trace of a single encryption consists of multiple subtraces of S-box calculations that use the same key byte. By doing a CPA attack *horizontally* on these subtraces, the key byte can be recovered. In our scheme, each key byte is only used once per encryption. This makes a similar horizontal attack on a single trace impossible.

Experiments. Nevertheless, our implementation does use the same mask for every single state byte and in every single round of AES. A normal CPA attack relies on the ability of the attacker to make hypotheses on intermediates. In the case of AES, these hypotheses typically target the output of an S-box $S(x_i \oplus k)$, where x_i is variable and known (in the first/last round) and k is constant and unknown and hence, the target of the attack. In our case, the S-box output is protected with a mask. The intermediate in the traces would be $S(x_i \oplus k) \oplus m_i$ when considered vertically, with i the index of the trace/encryption. In this case, x_i is variable and known (the plaintext), k is constant and unknown and m_i is variable and unknown, since each encryption uses a new mask. The key cannot be extracted unless the mask for each encryption m_i is known. When we consider a trace horizontally, the S-box outputs are $S(x_j \oplus k_j) \oplus m$ with x_j variable and unknown (state bytes) in all rounds except the first and last, k_j variable and unknown (different round keys) and m constant and unknown. It is clear that the key cannot be extracted using a horizontal CPA attack. However, if the constant and unknown mask m can be extracted horizontally from each single trace, a classic vertical CPA attack using hypotheses $S(x_i \oplus k) \oplus m_i$ can extract the key bytes.

To attack the mask within a single trace, we need to choose an intermediate that combines the mask with variable and known data x_j, i. e., the plaintext or ciphertext bytes. This should thus give 16 or 32 subtraces to do CPA over with the hypotheses $x_j \oplus m$.

We create simulated traces to test this attack, consisting of the Hamming weight of each state byte after each intermediate operation. While our actual implementation is bitsliced, our simulated traces consider bytes for simplicity. As shown in [BGRV15], the signal-to-noise ratio (SNR) of a bitsliced implementation is lower, but it does not prevent SCA.

For each individual trace, we consider 16 subtraces corresponding to the 16 byte-`XOR`s in the last AddRoundKey stage. Across these subtraces, we perform CPA to recover the mask of that trace. At least in simulation, this attack is very successful. Table 9.11 shows that even for high noise levels, only 8% of the masks are guessed incorrectly. However, note that attacking a linear operation in practice is very difficult.

Table 9.11: Percentage wrongly guessed masks after horizontal CPA on 16 subtraces.

SNR	100	10	1	0.1	0.01
% wrong masks	0	0	1.885	7.411	7.501

Once the masks have been guessed, we perform a normal vertical CPA, using the (mostly correct) knowledge of the masks. We repeat the experiment for various signal-to-noise ratios (SNRs) and measure the success by the average rank of the correct key byte. Figure 9.1 shows that for very high SNR, the attack succeeds with only a few thousand traces. For a more realistic SNR = 1.0, the average key rank never becomes 0 with up to 10 000 traces. For low SNR levels, the attack never succeeds.

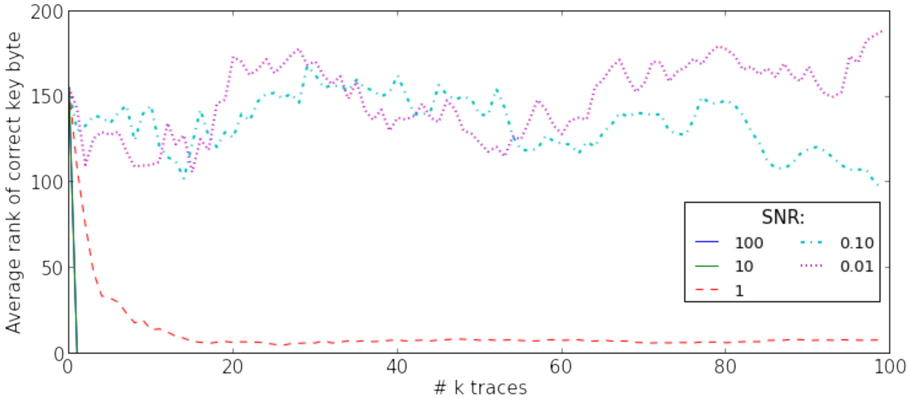


Figure 9.1: Average correct key rank as a function of the number of traces in a CPA attack followed by a horizontal mask recovery for different SNRs.

We performed a small investigation of the success of horizontal attacks against our implementation with extreme mask reuse. We used a fairly simplistic attack because we cannot rely on state-of-the-art analysis tools. We assumed a quite powerful adversary who can attack linear operations and showed that the reuse of masks does not trivially introduce vulnerabilities against horizontal attacks at realistic noise levels.

9.6.3 Beyond the t-Probing Model

It was already noted by Wegener and Moradi [WM18b] that the t-probing model does not incorporate transition leakages, which in our case of extreme randomness reuse, are dangerous. On the one hand, having the same masks for each bit of the state leads to transition leakages if the S-box is implemented in a serial way. Furthermore, the reuse of the same masks in each round, has the same effect in a round-based implementation. This example alone already shows the huge gap between theory and practice. However, we have shown that our solution can at least obtain very competitive performance compared to previous work, even if we double the latency with reset cycles against transition leakage.

While our scheme is an extreme example of how theoretical security may be insufficient in practice, similar conclusions can be made for previous works that target security in the t-probing model, even those in which randomness is used as described in [ISW03] and never reused. Effects such as transition leakages have been especially well studied for software by Balash *et al.* [BGG+14] and more recently by Papagiannopoulos *et al.* [PV17] among others. The resetting and clearing of registers is a popular solution proposed both in [WM18b] and [PV17], but incurs a very high penalty on the latency. The authors of [BGG+14] propose to use a theoretically $2t$-secure scheme when targeting t-order security.

To conclude, we have shown that eliminating randomness (apart from 2 bits) is possible. We however also noted that the models currently used are not prohibitive enough to guarantee security in practice and that theoretically secure solutions should be superposed with additional expensive countermeasures to achieve the desired protection. An interesting question for future work is whether the models can be adapted such that schemes are practically secure from the start.

9.7 Conclusions and Future Work

In this chapter, we have demonstrated that first-order masking in theory does not require more than two bits of randomness in both software and hardware. These two bits of randomness include the initial randomness for masking of secret data as well as the so-called online randomness that is usually required by other masking approaches to keep the first-order probing security. We thus throw away the distinction of randomness spent on masking the input data and the randomness spent on keeping this independence during the computation, since it is not very meaningful for applications in practice.

We have also shown that our approach not only leads to first-order probing secure implementations (which we verified using formal tools as well as manual verification) but also that this approach can be automated easily.

The downside of our approach, which is more noticeable in hardware, is an increased latency behavior due to the required control of the order in which operations are performed. However, we want to emphasize that the main idea of this work was to demonstrate that two bits of randomness not only pose the intuitive theoretical lower bound for first-order masking but that this bound is achievable in theory.

Our findings not only give answers to intriguing research questions but also lead the way to some follow-up questions.

- We demonstrated that when sacrificing latency in hardware, a lot of random bits can be saved and therefore the costs involved with the production of randomness. At the same time, there exists work like the low-latency masking approach of Gross *et al.* [GIB18], that show that arbitrary functions can be calculated in a securely masked way and in a single cycle when randomness considerations are not taken into account. A consequent next step is thus to research concepts to design masked implementations which achieve a better trade-off regarding latency and area for a give randomness budget.

- Another open question is if and how the introduced concepts can be extended to higher-order masking. For first-order masking, an attacker is limited to a single observation and thus masks can be reused in the same form and combination at different points in the implementation. For higher-order masking, the same combination of masks at different positions automatically lead to a violation of the probing security. This does not mean that mask reuse is not possible but only that more aspects need to be taken into account like the encoding of the masks at multiple positions.

- While two random bits are enough to achieve first-order probing security, it does not mean that it suffices to protect against first-order DPA in practice. Using less randomness may provide a larger attack surface to horizontal attacks and most likely also increases the signal-to-noise ratio for a DPA attacker. Also, transition leakages become more prominent when randomness recycling is used in extremis. The gap between theory and practice has never been more clear and also unoptimized schemes that have been designed in the t-probing model are vulnerable in practice. Naturally, our scheme's almost complete elimination of randomness means that its actual security level (e. g., in terms of required leakage traces to extract a secret) is not the same as for an implementation that uses a lot of randomness on mask refreshing of intermediate values. However, what is less obvious is the question whether or not the saved randomness could be more effectively used, e. g., for additional hiding countermeasures that lower the signal-to-noise ration by a higher extent than by spending more randomness on masking itself. There are two different approaches to take in the future. On the one hand, we can keep designing masking schemes in the theoretical t-probing model, pushing the limits and reaching new boundaries. However, further research is needed into the implementation cost of existing schemes when combined with the extra countermeasures necessary to take them beyond the t-probing model into practice. An alternative route is to adapt our models and design schemes which in themselves provide the needed practical security. The next challenge is then to push the limits in those models.

Chapter 10

Conclusions and Outlook

Throughout the previous chapters we proposed and discussed ideas for the optimization of various aspects of symmetric cryptography. The technique based on SAT solvers introduced in Chapter 3 works well for small S-boxes for which implementations with few operations exist. Unfortunately, the technique scales poorly to larger S-boxes. It would be interesting to see if this method can be combined with advances in the fields of logic synthesis and particularly exact synthesis, which may be less known in the symmetric-cryptography community.

Chapter 4 discussed global optimization of implementations of MDS matrices. We showed how a metric used previously in literature yields less-than-optimal results and how existing techniques for solving shortest linear straight-line programs apply to the optimization of MDS matrices. It makes one wonder how much of research is really obsoleted by existing results that are simply poorly available, phrased differently, forgotten through history, written in a different language, or known only in other research communities. Research on improving the heuristic algorithms may provide improvements to many results as the problems are very generic.

The final cryptographic building block that we looked at extensively was the column-parity mixer (CPM) of Chapter 5. We studied many of its properties and even designed a full permutation to showcase how CPMs can be integrated in designs of cryptographic schemes. One idea already put forward in the chapter is that it would be interesting to learn more about the relation between the density of a parity-folding matrix, the quality of the diffusion, and the implementation cost. There might be some optimal trade-off. Another direction that could be taken is to see whether there are better transposition layers that combine well with CPMs.

In Chapter 6 we showed speed-optimized assembly implementation of AES for the ARM Cortex-M3 and Cortex-M4. Despite the fact that none of the techniques are particularly new, the implementation did improve over the previous state of the art by quite a lot. This is partially due to a custom instruction scheduler and register allocator used for SubBytes, but also due to meticulously taking microarchitectural properties, such as alignment and pipelining, into account. Especially for embedded applications it holds that cryptography can be a performance bottleneck for an application, so we hope that our effort contributes to alleviating these costs.

Chapter 7 considers speed-optimized implementations of various cryptographic primitives on the RISC-V architecture and is somewhat more experimental in nature, in the sense that not a lot of optimization work had been done on this architecture before. One of the outcomes is that comparisons between RISC-V implementations will be painful due to the large number of optional instruction-set extensions. An extension with some extra instructions can have a large impact on what is the optimal implementation strategy for a particular cryptographic algorithm. It would be useful if, while the RISC-V project matures and more extension specifications are frozen, the supported extensions converge to a subset that is commonly available and can be used in comparisons.

In the final part we looked at countermeasures against side-channel analysis. Chapter 8 proposed a parallel implementation strategy for higher-order masking of AES. We showed how 4-share and 8-share implementations of AES can be done efficiently on an ARM Cortex-A8 with parallelism provided by NEON using gadgets that are proven to be SNI. We also performed an extensive side-channel evaluation. It would be interesting to find out why the order reduction happens as we observed and to find more efficient SNI gadgets that require less randomness.

Requiring less randomness is taken to the extreme in Chapter 9, where we showed how AES can be masked with two shares with just two random bits and no 'online' randomness whatsoever. We formally verified that our approach achieves first-order security in the probing model. While the implementation will not protect from a DPA attack in practice, it is an interesting experiment that raises many questions. One of them has been open for a while and that is if something meaningful can be said about the amount and quality of randomness that is needed to thwart DPA attacks in practice. On the more theoretical side, it is still unclear how to extend a masking scheme like the one presented here to higher orders.

These are just some examples of possible future directions based on the results presented in this thesis on optimizations in symmetric cryptography. Naturally, many more questions can be raised that may inspire new research. Scientific research progresses exactly because of this mechanic: an everlasting cycle of questions and attempts at answers. While the work presented in this thesis feels like a big deal to me personally as it captures years of work, it remains but a speck of dust in the grander scheme of science. As long as there are people that find my answers useful or inspiring, then at least it was a time well spent.

Bibliography

[ABB+14] Elena Andreeva, Begül Bilgin, Andrey Bogdanov, Atul Luykx, Florian Mendel, Bart Mennink, Nicky Mouha, Qingju Wang, and Kan Yasuda. *PRIMATEs v1.02*. CAESAR submission. Sept. 2014. URL: `https://competitions.cr.yp.to/round2/primatesv102.pdf`.

[ABM04] Kubilay Atasu, Luca Breveglieri, and Marco Macchetti. "Efficient AES Implementations for ARM Based Platforms". In: *Proceedings of the 2004 ACM Symposium on Applied Computing*. SAC '04. ACM, Mar. 2004, pp. 841–845. DOI: `10.1145/967900.968073`.

[ADK+14] Martin R. Albrecht, Benedikt Driessen, Elif Bilge Kavun, Gregor Leander, Christof Paar, and Tolga Yalçin. "Block Ciphers - Focus on the Linear Layer (feat. PRIDE)". In: *Advances in Cryptology – CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2014, pp. 57–76. DOI: `10.1007/978-3-662-44371-2_4`.

[ARM15] ARM Holdings plc. *ARM's Cortex-M and Cortex-R Embedded Processors*. 2015. URL: `https://www.arm.com/zh/files/event/2_2015_ARM_Embedded_Seminar_Ian_Johnson.pdf`.

[ARS+15] Martin R. Albrecht, Christian Rechberger, Thomas Schneider, Tyge Tiessen, and Michael Zohner. "Ciphers for MPC and FHE". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2015, pp. 430–454. DOI: `10.1007/978-3-662-46800-5_17`.

[Ava17] Roberto Avanzi. "The QARMA Block Cipher Family". In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017), pp. 4–44. ISSN: 2519-173X. DOI: `10.13154/tosc.v2017.i1.4-44`.

[BBC+19] Gilles Barthe, Sonia Belaïd, Gaëtan Cassiers, Pierre-Alain Fouque, Benjamin Grégoire, and François-Xavier Standaert. "maskVerif: Automated Verification of Higher-Order Masking in Presence of Physical Defaults". In: *ESORICS 2019: 24th European Symposium on Research in Computer Security, Part I*. Ed. by Kazue Sako, Steve Schneider, and Peter Y. A. Ryan. Vol. 11735. Lecture Notes in

Computer Science. Springer, Heidelberg, Sept. 2019, pp. 300–318. DOI: `10.1007/978-3-030-29959-0_15`.

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Pierre-Yves Strub. "Verified Proofs of Higher-Order Masking". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2015, pp. 457–485. DOI: `10.1007/978-3-662-46800-5_18`.

[BBD+16] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, Pierre-Yves Strub, and Rébecca Zucchini. "Strong Non-Interference and Type-Directed Higher-Order Masking". In: *ACM CCS 2016: 23rd Conference on Computer and Communications Security*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM Press, Oct. 2016, pp. 116–129. DOI: `10.1145/2976749.2978427`.

[BBF+03] Guido Bertoni, Luca Breveglieri, Pasqualina Fragneto, Marco Macchetti, and Stefano Marchesin. "Efficient Software Implementation of AES on 32-Bit Platforms". In: *Cryptographic Hardware and Embedded Systems – CHES 2002*. Ed. by Burton S. Kaliski Jr., Çetin Kaya Koç, and Christof Paar. Vol. 2523. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2003, pp. 159–171. DOI: `10.1007/3-540-36400-5_13`.

[BBI+15] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. "Midori: A Block Cipher for Low Energy". In: *Advances in Cryptology – ASIACRYPT 2015, Part II*. Ed. by Tetsu Iwata and Jung Hee Cheon. Vol. 9453. Lecture Notes in Computer Science. Springer, Heidelberg, Nov. 2015, pp. 411–436. DOI: `10.1007/978-3-662-48800-3_17`.

[BBK+13] Begül Bilgin, Andrey Bogdanov, Miroslav Knežević, Florian Mendel, and Qingju Wang. "Fides: Lightweight Authenticated Cipher with Side-Channel Resistance for Constrained Hardware". In: *Cryptographic Hardware and Embedded Systems – CHES 2013*. Ed. by Guido Bertoni and Jean-Sébastien Coron. Vol. 8086. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2013, pp. 142–158. DOI: `10.1007/978-3-642-40349-1_9`.

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Randomness Complexity of Private Circuits for Multiplication". In: *Advances in Cryptology – EUROCRYPT 2016, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. Lecture Notes in Computer Science. Springer, Heidelberg, May 2016, pp. 616–648. DOI: `10.1007/978-3-662-49896-5_22`.

[BBP+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. "Private Multiplication over Finite Fields". In: *Advances in Cryptology – CRYPTO 2017, Part III*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10403. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2017, pp. 397–426. DOI: `10.1007/978-3-319-63697-9_14`.

[BBR16a] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. *Atomic-AES v2.0*. Cryptology ePrint Archive, Report 2016/1005. 2016. URL: `https://eprint.iacr.org/2016/1005`.

[BBR16b] Subhadeep Banik, Andrey Bogdanov, and Francesco Regazzoni. "Atomic-AES: A Compact Implementation of the AES Encryption/Decryption Core". In: *Progress in Cryptology - INDOCRYPT 2016: 17th International Conference in Cryptology in India*. Ed. by Orr Dunkelman and Somitra Kumar Sanadhya. Vol. 10095. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2016, pp. 173–190. DOI: `10.1007/978-3-319-49890-4_10`.

[BBS05] Eli Biham, Alex Biryukov, and Adi Shamir. "Cryptanalysis of Skipjack Reduced to 31 Rounds Using Impossible Differentials". In: *Journal of Cryptology* 18.4 (Sept. 2005), pp. 291–311. DOI: `10.1007/s00145-005-0129-3`.

[BC14] Daniel J. Bernstein and Tung Chou. "Faster Binary-Field Multiplication and Faster Binary-Field MACs". In: *SAC 2014: 21st Annual International Workshop on Selected Areas in Cryptography*. Ed. by Antoine Joux and Amr M. Youssef. Vol. 8781. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2014, pp. 92–111. DOI: `10.1007/978-3-319-13051-4_6`.

[BCG+12] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knežević, Lars R. Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, Peter Rombouts, Søren S. Thomsen, and Tolga Yalçin. "PRINCE - A Low-Latency Block

Cipher for Pervasive Computing Applications - Extended Abstract". In: *Advances in Cryptology – ASIACRYPT 2012*. Ed. by Xiaoyun Wang and Kazue Sako. Vol. 7658. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2012, pp. 208–225. DOI: `10.1007/978-3-642-34961-4_14`.

[BCJ07] Gregory V. Bard, Nicolas T. Courtois, and Chris Jefferson. *Efficient Methods for Conversion and Solution of Sparse Systems of Low-Degree Multivariate Polynomials over GF(2) via SAT-Solvers*. Cryptology ePrint Archive, Report 2007/024. 2007. URL: `https://eprint.iacr.org/2007/024`.

[BCL+04] Alex Biryukov, Christophe De Cannière, Joseph Lano, Siddika Berna Ors, and Bart Preneel. *Security and Performance Analysis of ARIA*. Jan. 2004. URL: `https://www.esat.kuleuven.be/cosic/publications/article-500.pdf`.

[BCL14] Daniel J. Bernstein, Chitchanok Chuengsatiansup, and Tanja Lange. "Curve41417: Karatsuba Revisited". In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2014, pp. 316–334. DOI: `10.1007/978-3-662-44709-3_18`.

[BCLR17] Christof Beierle, Anne Canteaut, Gregor Leander, and Yann Rotella. "Proving Resistance Against Invariant Attacks: How to Choose the Round Constants". In: *Advances in Cryptology – CRYPTO 2017, Part II*. Ed. by Jonathan Katz and Hovav Shacham. Vol. 10402. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2017, pp. 647–678. DOI: `10.1007/978-3-319-63715-0_22`.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems – CHES 2004*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2004, pp. 16–29. DOI: `10.1007/978-3-540-28632-5_2`.

[BCPZ16] Alberto Battistello, Jean-Sébastien Coron, Emmanuel Prouff, and Rina Zeitoun. "Horizontal Side-Channel Attacks and Countermeasures on the ISW Masking Scheme". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in Computer Science.

Springer, Heidelberg, Aug. 2016, pp. 23–39. DOI: `10.1007/978-3-662-53140-2_2`.

[BDCU17] Alex Biryukov, Daniel Dinu, Yann Le Corre, and Aleksei Udovenko. "Optimal First-Order Boolean Masking for Embedded IoT Devices". In: *Smart Card Research and Advanced Applications – CARDIS 2017*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, Heidelberg, Jan. 2017, pp. 22–41. DOI: `10.1007/978-3-319-75208-2_2`.

[BDF+17] Gilles Barthe, François Dupressoir, Sebastian Faust, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. "Parallel Implementations of Masking Schemes and the Bounded Moment Leakage Model". In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2017, pp. 535–566. DOI: `10.1007/978-3-319-56620-7_19`.

[BDH+] Guido Bertoni, Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *eXtended Keccak Code Package*. URL: `https://github.com/XKCP/XKCP`.

[BDP+12] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keccak implementation overview*. Version 3.2. May 2012. URL: `https://keccak.team/files/Keccak-implementation-3.2.pdf`.

[BDP+16a] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Ketje v2*. CAESAR submission. Sept. 2016. URL: `https://keccak.team/files/Ketjev2-doc2.0.pdf`.

[BDP+16b] Guido Bertoni, Joan Daemen, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. *Keyak v2*. CAESAR submission. Sept. 2016. URL: `https://keccak.team/files/Keyakv2-doc2.2.pdf`.

[BDPV11a] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *Cryptographic sponge functions*. Jan. 2011. URL: `https://keccak.team/files/CSF-0.1.pdf`.

[BDPV11b] Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. *The Keccak reference*. Jan. 2011. URL: `https://keccak.team/files/Keccak-reference-3.0.pdf`.

[Bea75] Kenneth G. Beauchamp. *Walsh Functions and their Applications*. Academic Press, 1975. ISBN: 0-12-084050-2.

[Ber05a] Daniel J. Bernstein. *Cache-timing attacks on AES*. Apr. 2005. URL: `https://cr.yp.to/antiforgery/cachetiming-20050414.pdf`.

[Ber05b] Daniel J. Bernstein. "The Poly1305-AES Message-Authentication Code". In: *Fast Software Encryption – FSE 2005*. Ed. by Henri Gilbert and Helena Handschuh. Vol. 3557. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2005, pp. 32–49. DOI: `10.1007/11502760_3`.

[Ber08a] Daniel J. Bernstein. *ChaCha, a variant of Salsa20*. Jan. 2008. URL: `https://cr.yp.to/chacha/chacha-20080120.pdf`.

[Ber08b] Daniel J. Bernstein. "The Salsa20 Family of Stream Ciphers". In: *New Stream Cipher Designs: The eSTREAM Finalists*. Ed. by Matthew Robshaw and Olivier Billet. Vol. 4986. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 84–97. ISBN: 978-3-540-68351-3. DOI: `10.1007/978-3-540-68351-3_8`.

[Ber09] Daniel J. Bernstein. *Optimizing linear maps modulo 2*. Workshop Record of SPEED-CC – Software Performance Enhancement for Encryption and Decryption and Cryptographic Compilers. Aug. 2009. URL: `https://binary.cr.yp.to/linearmod2-20090830.pdf`.

[BFG+17] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, Clara Paglialonga, and François-Xavier Standaert. "Consolidating Inner Product Masking". In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2017, pp. 724–754. DOI: `10.1007/978-3-319-70694-8_25`.

[BFP19] Joan Boyar, Magnus Gausdal Find, and René Peralta. "Small low-depth circuits for cryptographic applications". In: *Cryptography and Communications* 11.1 (Jan. 2019), pp. 109–127. ISSN: 1936-2455. DOI: `10.1007/s12095-018-0296-3`.

[BGG+14] Josep Balasch, Benedikt Gierlichs, Vincent Grosso, Oscar Reparaz, and François-Xavier Standaert. "On the Cost of Lazy Engineering for Masked Software Implementations". In: *Smart Card Research and Advanced Applications – CARDIS 2014*. Ed. by Marc Joye and Amir Moradi. Vol. 8968. Lecture Notes in Computer Science. Springer, Heidelberg, 2014, pp. 64–81. ISBN: 978-3-319-16763-3. DOI: `10.1007/978-3-319-16763-3_5`.

[BGI+18] Roderick Bloem, Hannes Groß, Rinat Iusupov, Bettina Könighofer, Stefan Mangard, and Johannes Winter. "Formal Verification of Masked Hardware Implementations in the Presence of Glitches". In: *Advances in Cryptology – EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 321–353. DOI: `10.1007/978-3-319-78375-8_11`.

[BGN+14] Begül Bilgin, Benedikt Gierlichs, Svetla Nikova, Ventzislav Nikov, and Vincent Rijmen. "A More Efficient AES Threshold Implementation". In: *AFRICACRYPT 14: 7th International Conference on Cryptology in Africa*. Ed. by David Pointcheval and Damien Vergnaud. Vol. 8469. Lecture Notes in Computer Science. Springer, Heidelberg, May 2014, pp. 267–284. DOI: `10.1007/978-3-319-06734-6_17`.

[BGRV15] Josep Balasch, Benedikt Gierlichs, Oscar Reparaz, and Ingrid Verbauwhede. "DPA, Bitslicing and Masking at 1 GHz". In: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2015, pp. 599–619. DOI: `10.1007/978-3-662-48324-4_30`.

[Bih97] Eli Biham. "A Fast New DES Implementation in Software". In: *Fast Software Encryption – FSE'97*. Ed. by Eli Biham. Vol. 1267. Lecture Notes in Computer Science. Springer, Heidelberg, Jan. 1997, pp. 260–272. DOI: `10.1007/BFb0052352`.

[BJK+16] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. "The SKINNY Family of Block Ciphers and Its Low-Latency Variant MANTIS". In: *Advances in Cryptology – CRYPTO 2016, Part II*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9815. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 123–153. DOI: `10.1007/978-3-662-53008-5_5`.

[BJPW13] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. "Horizontal and Vertical Side-Channel Attacks against Secure RSA Implementations". In: *Topics in Cryptology – CT-RSA 2013*. Ed. by Ed Dawson. Vol. 7779. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2013, pp. 1–17. DOI: 10.1007/978-3-642-36095-4_1.

[BJPW14] Aurélie Bauer, Éliane Jaulmes, Emmanuel Prouff, and Justine Wild. "Horizontal Collision Correlation Attack on Elliptic Curves". In: *SAC 2013: 20th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Tanja Lange, Kristin Lauter, and Petr Lisonek. Vol. 8282. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2014, pp. 553–570. DOI: 10.1007/978-3-662-43414-7_28.

[BKL+07] Andrey Bogdanov, Lars R. Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew J. B. Robshaw, Yannick Seurin, and C. Vikkelsoe. "PRESENT: An Ultra-Lightweight Block Cipher". In: *Cryptographic Hardware and Embedded Systems – CHES 2007*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2007, pp. 450–466. DOI: 10.1007/978-3-540-74735-2_31.

[BKL+17] Daniel J. Bernstein, Stefan Kölbl, Stefan Lucks, Pedro Maat Costa Massolino, Florian Mendel, Kashif Nawaz, Tobias Schneider, Peter Schwabe, François-Xavier Standaert, Yosuke Todo, and Benoît Viguier. "Gimli : A Cross-Platform Permutation". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2017, pp. 299–320. DOI: 10.1007/978-3-319-66787-4_15.

[BKL16] Christof Beierle, Thorsten Kranz, and Gregor Leander. "Lightweight Multiplication in GF(2^n) with Applications to MDS Matrices". In: *Advances in Cryptology – CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 625–653. DOI: 10.1007/978-3-662-53018-4_23.

[BMP08] Joan Boyar, Philip Matthews, and René Peralta. "On the Shortest Linear Straight-Line Program for Computing Linear Forms". In: *Mathematical Foundations of Computer Science 2008*. Ed. by Edward Ochmanski and Jerzy Tyszkiewicz. Vol. 5162. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 168–179. ISBN: 978-3-540-85237-7. DOI: 10.1007/978-3-540-85238-4_13.

[BMP13] Joan Boyar, Philip Matthews, and René Peralta. "Logic Minimization Techniques with Applications to Cryptology". In: *Journal of Cryptology* 26.2 (Apr. 2013), pp. 280–312. DOI: `10.1007/s00145-012-9124-7`.

[BNN+10] Paulo S. L. M. Barreto, Ventzislav Nikov, Svetla Nikova, Vincent Rijmen, and Elmar Tischhauser. "Whirlwind: a new cryptographic hash function". In: *Designs, Codes and Cryptography* 56.2–3 (Aug. 2010), pp. 141–162. ISSN: 1573-7586. DOI: `10.1007/s10623-010-9391-y`.

[BP10] Joan Boyar and René Peralta. "A New Combinational Logic Minimization Technique with Applications to Cryptology". In: *Experimental Algorithms*. Ed. by Paola Festa. Vol. 6049. Lecture Notes in Computer Science. Springer, Heidelberg, 2010, pp. 178–189. ISBN: 978-3-642-13192-9. DOI: `10.1007/978-3-642-13193-6_16`.

[BP12] Joan Boyar and René Peralta. "A Small Depth-16 Circuit for the AES S-Box". In: *Information Security and Privacy Research – SEC 2010*. Vol. 376. IFIP Advances in Information and Communication Technology. Springer, Heidelberg, 2012, pp. 287–298. DOI: `10.1007/978-3-642-30436-1_24`.

[BPP00] Joan Boyar, René Peralta, and Denis Pochuev. "On the multiplicative complexity of Boolean functions over the basis $(\wedge, \oplus, 1)$". In: *Theoretical Computer Science* 235.1 (2000), pp. 43–57. ISSN: 0304-3975. DOI: `10.1016/S0304-3975(99)00182-6`.

[BR00a] Paulo S. L. M. Barreto and Vincent Rijmen. *The ANUBIS Block Cipher*. First Open NESSIE Workshop. 2000.

[BR00b] Paulo S. L. M. Barreto and Vincent Rijmen. *The Khazad legacy-level Block Cipher*. First Open NESSIE Workshop. 2000.

[BR00c] Paulo S. L. M. Barreto and Vincent Rijmen. *The WHIRLPOOL Hashing Function*. First Open NESSIE Workshop. 2000.

[BS08] Daniel J. Bernstein and Peter Schwabe. "New AES Software Speed Records". In: *Progress in Cryptology - INDOCRYPT 2008: 9th International Conference in Cryptology in India*. Ed. by Dipanwita Roy Chowdhury, Vincent Rijmen, and Abhijit Das. Vol. 5365. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2008, pp. 322–336.

[BS12] Daniel J. Bernstein and Peter Schwabe. “NEON Crypto”. In: *Cryptographic Hardware and Embedded Systems – CHES 2012*. Ed. by Emmanuel Prouff and Patrick Schaumont. Vol. 7428. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2012, pp. 320–339. DOI: `10.1007/978-3-642-33027-8_19`.

[BS91] Eli Biham and Adi Shamir. “Differential Cryptanalysis of DES-like Cryptosystems”. In: *Journal of Cryptology* 4.1 (Jan. 1991), pp. 3–72. DOI: `10.1007/BF00630563`.

[BU08] David Buchfuhrer and Christopher Umans. “The Complexity of Boolean Formula Minimization”. In: *Automata, Languages and Programming*. Ed. by Luca Aceto, Ivan Damgård, Leslie Ann Goldberg, Magnús M. Halldórsson, Anna Ingólfsdóttir, and Igor Walukiewicz. Vol. 5125. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 24–35. ISBN: 978-3-540-70574-1. DOI: `10.1007/978-3-540-70575-8_3`.

[BW99] Alex Biryukov and David Wagner. “Slide Attacks”. In: *Fast Software Encryption – FSE’99*. Ed. by Lars R. Knudsen. Vol. 1636. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 1999, pp. 245–259. DOI: `10.1007/3-540-48519-8_18`.

[Can05] David Canright. “A Very Compact S-Box for AES”. In: *Cryptographic Hardware and Embedded Systems – CHES 2005*. Ed. by Josyula R. Rao and Berk Sunar. Vol. 3659. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2005, pp. 441–455. DOI: `10.1007/11545262_32`.

[CB08] David Canright and Lejla Batina. “A Very Compact “Perfectly Masked” S-Box for AES”. In: *ACNS 08: 6th International Conference on Applied Cryptography and Network Security*. Ed. by Steven M. Bellovin, Rosario Gennaro, Angelos D. Keromytis, and Moti Yung. Vol. 5037. Lecture Notes in Computer Science. Springer, Heidelberg, June 2008, pp. 446–459. DOI: `10.1007/978-3-540-68914-0_27`.

[CBG+17] Thomas De Cnudde, Begül Bilgin, Benedikt Gierlichs, Ventzislav Nikov, Svetla Nikova, and Vincent Rijmen. “Does Coupling Affect the Security of Masked Implementations?” In: *COSADE 2017: 8th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2017, pp. 1–18. DOI: `10.1007/978-3-319-64647-3_1`.

[CDG+13] Jeremy Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, and Pankaj Rohatgi. *Test Vector Leakage Assessment (TVLA) methodology in practice*. 2013.

[CHM11] Nicolas T. Courtois, Daniel Hulme, and Theodosis Mourouzis. *Solving Circuit Optimisation Problems in Cryptography and Cryptanalysis*. Cryptology ePrint Archive, Report 2011/475. 2011. URL: `https://eprint.iacr.org/2011/475`.

[CJRR99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. "Towards Sound Approaches to Counteract Power-Analysis Attacks". In: *Advances in Cryptology – CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1999, pp. 398–412. DOI: `10.1007/3-540-48405-1_26`.

[CMH13] Nicolas Courtois, Theodosis Mourouzis, and Daniel Hulme. "Exact Logic Minimization and Multiplicative Complexity of Concrete Algebraic and Cryptographic Circuits". In: *International Journal On Advances in Intelligent Systems* 6.3&4 (2013), pp. 165–176. ISSN: 1942-2679.

[CMR05] Carlos Cid, Sean Murphy, and Matthew J. B. Robshaw. "Small Scale Variants of the AES". In: *Fast Software Encryption – FSE 2005*. Ed. by Henri Gilbert and Helena Handschuh. Vol. 3557. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2005, pp. 145–162. DOI: `10.1007/11502760_10`.

[CMV09] Benjamin Chambers, Panagiotis Manolios, and Daron Vroon. "Faster SAT Solving with Better CNF Generation". In: *Proceedings of the Conference on Design, Automation and Test in Europe*. DATE '09. European Design and Automation Association, Apr. 2009, pp. 1590–1595. ISBN: 978-3-9810801-5-5. URL: `https://dl.acm.org/citation.cfm?id=1874620.1875002`.

[CPRR14] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. "Higher-Order Side Channel Security and Mask Refreshing". In: *Fast Software Encryption – FSE 2013*. Ed. by Shiho Moriai. Vol. 8424. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2014, pp. 410–424. DOI: `10.1007/978-3-662-43933-3_21`.

[CYK+12] Jiali Choy, Huihui Yap, Khoongming Khoo, Jian Guo, Thomas Peyrin, Axel Poschmann, and Chik How Tan. "SPN-Hash: Improving the Provable Resistance against Differential Collision Attacks". In: *AFRICACRYPT 12: 5th International Conference on Cryptology in Africa*. Ed. by Aikaterini Mitrokotsa and Serge Vaudenay. Vol. 7374. Lecture Notes in Computer Science. Springer, Heidelberg, July 2012, pp. 270–286. DOI: `10.1007/978-3-642-31410-0_17`.

[Dae17] Joan Daemen. "Changing of the Guards: A Simple and Efficient Method for Achieving Uniformity in Threshold Sharing". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2017, pp. 137–153. DOI: `10.1007/978-3-319-66787-4_7`.

[Dae95] Joan Daemen. "Cipher and hash function design strategies based on linear and differential cryptanalysis". PhD thesis. KU Leuven, Mar. 1995.

[DCK+15] Daniel Dinu, Yann Le Corre, Dmitry Khovratovich, Léo Perrin, Johann Großschädl, and Alex Biryukov. *Triathlon of Lightweight Block Ciphers for the Internet of Things*. Cryptology ePrint Archive, Report 2015/209. 2015. URL: `https://eprint.iacr.org/2015/209`.

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. "Unifying Leakage Models: From Probing Attacks to Noisy Leakage". In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, Heidelberg, May 2014, pp. 423–440. DOI: `10.1007/978-3-642-55220-5_24`.

[DEMS16] Christoph Dobraunig, Maria Eichlseder, Florian Mendel, and Martin Schläffer. *Ascon v1.2*. CAESAR submission. Sept. 2016. URL: `https://ascon.iaik.tugraz.at/files/asconv12.pdf`.

[DES77] *Data Encryption Standard*. National Bureau of Standards, NBS FIPS PUB 46, U.S. Department of Commerce. Jan. 1977.

[DF03] David S. Dummit and Richard M. Foote. *Abstract Algebra*. 3rd ed. John Wiley and Sons, Inc., 2003. ISBN: 978-0-471-43334-7.

[DFS15] Alexandre Duc, Sebastian Faust, and François-Xavier Standaert. "Making Masking Security Proofs Concrete - Or How to Evaluate the Security of Any Leaking Device". In: *Advances in Cryptology – EUROCRYPT 2015, Part I*. Ed. by Elisabeth Oswald and Marc Fischlin. Vol. 9056. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2015, pp. 401–429. DOI: `10.1007/978-3-662-46800-5_16`.

[DGV93] Joan Daemen, René Govaerts, and Joos Vandewalle. "Block Ciphers Based on Modular Arithmetic". In: *Proceedings of the 3rd Symposium on the State and Progress of Research in Cryptography*. Ed. by William Wolfowicz. Feb. 1993, pp. 80–89.

[DHH+15] Michael Düll, Björn Haase, Gesine Hinterwälder, Michael Hutter, Christof Paar, Ana Helena Sánchez, and Peter Schwabe. "High-speed Curve25519 on 8-bit, 16-bit, and 32-bit microcontrollers". In: *Designs, Codes and Cryptography* 77.2–3 (Dec. 2015), pp. 493–514. ISSN: 1573-7586. DOI: `10.1007/s10623-015-0087-1`.

[DKR97] Joan Daemen, Lars R. Knudsen, and Vincent Rijmen. "The Block Cipher Square". In: *Fast Software Encryption – FSE'97*. Ed. by Eli Biham. Vol. 1267. Lecture Notes in Computer Science. Springer, Heidelberg, Jan. 1997, pp. 149–165. DOI: `10.1007/BFb0052343`.

[DR02] Joan Daemen and Vincent Rijmen. *The Design of Rijndael: AES - The Advanced Encryption Standard*. Information Security and Cryptography. Springer, Heidelberg, 2002. ISBN: 3-540-42580-2. DOI: `10.1007/978-3-662-04722-4`.

[DR06] Joan Daemen and Vincent Rijmen. "Understanding Two-Round Differentials in AES". In: *SCN 06: 5th International Conference on Security in Communication Networks*. Ed. by Roberto De Prisco and Moti Yung. Vol. 4116. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2006, pp. 78–94. DOI: `10.1007/11832072_6`.

[DR99] Joan Daemen and Vincent Rijmen. *AES proposal: Rijndael, version 2*. Sept. 1999. URL: `https://csrc.nist.gov/archive/aes/rijndael/Rijndael-ammended.pdf`.

[DSP16] François Durvaux, François-Xavier Standaert, and Santos Merino Del Pozo. "Towards Easy Leakage Certification". In: *Cryptographic Hardware and Embedded Systems – CHES 2016*. Ed. by Benedikt Gierlichs and Axel Y. Poschmann. Vol. 9813. Lecture Notes in

Computer Science. Springer, Heidelberg, Aug. 2016, pp. 40–60. DOI: `10.1007/978-3-662-53140-2_3`.

[DSV14] François Durvaux, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. "How to Certify the Leakage of a Chip?" In: *Advances in Cryptology – EUROCRYPT 2014*. Ed. by Phong Q. Nguyen and Elisabeth Oswald. Vol. 8441. Lecture Notes in Computer Science. Springer, Heidelberg, May 2014, pp. 459–476. DOI: `10.1007/978-3-642-55220-5_26`.

[EM93] Shimon Even and Yishay Mansour. "A Construction of a Cipher From a Single Pseudorandom Permutation". In: *Advances in Cryptology – ASIACRYPT'91*. Ed. by Hideki Imai, Ronald L. Rivest, and Tsutomu Matsumoto. Vol. 739. Lecture Notes in Computer Science. Springer, Heidelberg, Nov. 1993, pp. 210–224. DOI: `10.1007/3-540-57332-1_17`.

[ET93] Bradley Efron and Robert J. Tibshirani. *An Introduction to the Bootstrap*. Chapman and Hall/CRC, 1993. ISBN: 978-0412042317.

[FA19] Hayato Fujii and Diego F. Aranha. "Curve25519 for the Cortex-M4 and beyond". In: *LATINCRYPT 2017: 5th International Conference on Cryptology and Information Security in Latin America*. Ed. by Tanja Lange and Orr Dunkelman. Vol. 11368. Lecture Notes in Computer Science. Springer, Heidelberg, July 2019, pp. 109–127. DOI: `10.1007/978-3-030-25283-0_6`.

[FBR06] Décio Luiz Gazzoni Filho, Paulo S. L. M. Barreto, and Vincent Rijmen. "The MAELSTROM-0 Hash Function". In: *Anais do VI Simpósio Brasileiro em Segurança da Informaçã e de Sistemas Computacionais*. 2006.

[FGP+18] Sebastian Faust, Vincent Grosso, Santos Merino Del Pozo, Clara Paglialonga, and François-Xavier Standaert. "Composable Masking Schemes in the Presence of Physical Defaults & the Robust Probing Model". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018). `https://tches.iacr.org/index.php/TCHES/article/view/7270`, pp. 89–120. ISSN: 2569-2925. DOI: `10.13154/tches.v2018.i3.89-120`.

[FPS17] Sebastian Faust, Clara Paglialonga, and Tobias Schneider. "Amortizing Randomness Complexity in Private Circuits". In: *Advances in Cryptology – ASIACRYPT 2017, Part I*. Ed. by Tsuyoshi Takagi and Thomas Peyrin. Vol. 10624. Lecture Notes in Computer Science.

Springer, Heidelberg, Dec. 2017, pp. 781–810. DOI: `10.1007/978-3-319-70694-8_27`.

[FS10] Carsten Fuhs and Peter Schneider-Kamp. "Synthesizing Shortest Linear Straight-Line Programs over GF(2) Using SAT". In: *Theory and Applications of Satisfiability Testing – SAT 2010*. Ed. by Ofer Strichman and Stefan Szeider. Vol. 6175. Lecture Notes in Computer Science. Springer, Heidelberg, 2010, pp. 71–84. ISBN: 978-3-642-14185-0. DOI: `10.1007/978-3-642-14186-7_8`.

[FS12] Carsten Fuhs and Peter Schneider-Kamp. "Optimizing the AES S-Box using SAT". In: *IWIL 2010. The 8th International Workshop on the Implementation of Logics*. Ed. by Geoff Sutcliffe, Stephan Schulz, and Eugenia Ternovska. Vol. 2. EPiC Series in Computing. EasyChair, 2012, pp. 64–70. DOI: `10.29007/h5s4`.

[GD17] Ashrujit Ghoshal and Thomas De Cnudde. "Several Masked Implementations of the Boyar-Peralta AES S-Box". In: *Progress in Cryptology - INDOCRYPT 2017: 18th International Conference in Cryptology in India*. Ed. by Arpita Patra and Nigel P. Smart. Vol. 10698. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2017, pp. 384–402.

[GIB18] Hannes Gross, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018). `https://tches.iacr.org/index.php/TCHES/article/view/871`, pp. 1–21. ISSN: 2569-2925. DOI: `10.13154/tches.v2018.i2.1-21`.

[GKM+] Praveen Gauravaram, Lars R. Knudsen, Krystian Matusiewicz, Florian Mendel, Christian Rechberger, Martin Schläffer, and Søren S. Thomsen. *Grøstl – a SHA-3 candidate*. Submitted to SHA-3.

[GM17] Hannes Groß and Stefan Mangard. "Reconciling $d + 1$ Masking in Hardware and Software". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2017, pp. 115–136. DOI: `10.1007/978-3-319-66787-4_6`.

[GM18] Hannes Groß and Stefan Mangard. "A unified masking approach". In: *Journal of Cryptographic Engineering* 8.2 (June 2018), pp. 109–124. DOI: `10.1007/s13389-018-0184-y`.

[GMK16] Hannes Gross, Stefan Mangard, and Thomas Korak. *Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order*. Cryptology ePrint Archive, Report 2016/486. 2016. URL: `https://eprint.iacr.org/2016/486`.

[GMO01] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. "Electromagnetic Analysis: Concrete Results". In: *Cryptographic Hardware and Embedded Systems – CHES 2001*. Ed. by Çetin Kaya Koç, David Naccache, and Christof Paar. Vol. 2162. Lecture Notes in Computer Science. Springer, Heidelberg, May 2001, pp. 251–261. DOI: `10.1007/3-540-44709-1_21`.

[GP99] Louis Goubin and Jacques Patarin. "DES and Differential Power Analysis (The "Duplication" Method)". In: *Cryptographic Hardware and Embedded Systems – CHES'99*. Ed. by Çetin Kaya Koç and Christof Paar. Vol. 1717. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1999, pp. 158–172. DOI: `10.1007/3-540-48059-5_15`.

[GPP+17] Wouter de Groot, Kostas Papagiannopoulos, Antonio de la Piedra, Erik Schneider, and Lejla Batina. "Bitsliced Masking and ARM: Friends or Foes?" In: *LightSec 2016: 5th International Workshop on Lightweight Cryptography for Security and Privacy*. Ed. by Andrey Bogdanov. Vol. 10098. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2017, pp. 91–109. DOI: `10.1007/978-3-319-55714-4_7`.

[GPP11] Jian Guo, Thomas Peyrin, and Axel Poschmann. "The PHOTON Family of Lightweight Hash Functions". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2011, pp. 222–239. DOI: `10.1007/978-3-642-22792-9_13`.

[GPPR11] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matthew J. B. Robshaw. "The LED Block Cipher". In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2011, pp. 326–341. DOI: `10.1007/978-3-642-23951-9_22`.

[GPSS18] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. "Vectorizing Higher-Order Masking". In: *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analy-*

sis and Secure Design. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 23–43. DOI: `10.1007/978-3-319-89641-0_2`.

[GR13] Kishan Chand Gupta and Indranil Ghosh Ray. "On Constructions of Involutory MDS Matrices". In: *AFRICACRYPT 13: 6th International Conference on Cryptology in Africa*. Ed. by Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien. Vol. 7918. Lecture Notes in Computer Science. Springer, Heidelberg, June 2013, pp. 43–60. DOI: `10.1007/978-3-642-38553-7_3`.

[GR17] Dahmun Goudarzi and Matthieu Rivain. "How Fast Can Higher-Order Masking Be in Software?" In: *Advances in Cryptology – EUROCRYPT 2017, Part I*. Ed. by Jean-Sébastien Coron and Jesper Buus Nielsen. Vol. 10210. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2017, pp. 567–597. DOI: `10.1007/978-3-319-56620-7_20`.

[GS18] Vincent Grosso and François-Xavier Standaert. "Masking Proofs Are Tight and How to Exploit it in Security Evaluations". In: *Advances in Cryptology – EUROCRYPT 2018, Part II*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10821. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 385–412. DOI: `10.1007/978-3-319-78375-8_13`.

[GSD+19] Hannes Gross, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. "First-Order Masking with Only Two Random Bits". In: *Proceedings of ACM Workshop on Theory of Implementation Security*. Ed. by Begül Bilgin, Svetla Petkova-Nikova, and Vincent Rijmen. TIS'19. ACM, 2019, pp. 10–23. DOI: `10.1145/3338467.3358950`.

[Ham09] Mike Hamburg. "Accelerating AES with Vector Permute Instructions". In: *Cryptographic Hardware and Embedded Systems – CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2009, pp. 18–32. DOI: `10.1007/978-3-642-04138-9_2`.

[HKT15] Neil Hanley, HeeSeok Kim, and Michael Tunstall. "Exploiting Collisions in Addition Chain-Based Exponentiation Algorithms Using a Single Trace". In: *Topics in Cryptology – CT-RSA 2015*. Ed. by Kaisa Nyberg. Vol. 9048. Lecture Notes in Computer Science.

Springer, Heidelberg, Apr. 2015, pp. 431–448. DOI: `10.1007/978-3-319-16715-2_23`.

[HRS16] Andreas Hülsing, Joost Rijneveld, and Peter Schwabe. "ARMed SPHINCS - Computing a 41 KB Signature in 16 KB of RAM". In: *PKC 2016: 19th International Conference on Theory and Practice of Public Key Cryptography, Part I*. Ed. by Chen-Mou Cheng, Kai-Min Chung, Giuseppe Persiano, and Bo-Yin Yang. Vol. 9614. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 446–470. DOI: `10.1007/978-3-662-49384-7_17`.

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology – CRYPTO 2003*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2003, pp. 463–481. DOI: `10.1007/978-3-540-45146-4_27`.

[JA09] Jorge Nakahara Jr. and Élcio Abrahão. "A New Involutory MDS Matrix for the AES". In: *International Journal of Network Security* 9.2 (Sept. 2009), pp. 109–116. ISSN: 1816-353X. URL: `http://ijns.femto.com.tw/contents/ijns-v9-n2/ijns-2009-v9-n2-p109-116.pdf`.

[JMPS17] Jérémy Jean, Amir Moradi, Thomas Peyrin, and Pascal Sasdrich. "Bit-Sliding: A Generic Technique for Bit-Serial Implementations of SPN-based Primitives - Applications to AES, PRESENT and SKINNY". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2017, pp. 687–707. DOI: `10.1007/978-3-319-66787-4_33`.

[JNP15] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. *Joltik v1.3*. CAESAR submission. Aug. 2015. URL: `https://competitions.cr.yp.to/round2/joltikv13.pdf`.

[JPST17] Jérémy Jean, Thomas Peyrin, Siang Meng Sim, and Jade Tourteaux. "Optimizing Implementations of Lightweight Building Blocks". In: *IACR Transactions on Symmetric Cryptology* 2017.4 (2017), pp. 130–168. ISSN: 2519-173X. DOI: `10.13154/tosc.v2017.i4.130-168`.

[JS17] Anthony Journault and François-Xavier Standaert. "Very High Order Masking: Efficient Implementation and Security Evaluation". In: *Cryptographic Hardware and Embedded Systems – CHES 2017*. Ed. by Wieland Fischer and Naofumi Homma. Vol. 10529. Lecture

Notes in Computer Science. Springer, Heidelberg, Sept. 2017, pp. 623–643. DOI: `10.1007/978-3-319-66787-4_30`.

[JV04] Pascal Junod and Serge Vaudenay. “FOX: A New Family of Block Ciphers”. In: *SAC 2004: 11th Annual International Workshop on Selected Areas in Cryptography*. Ed. by Helena Handschuh and Anwar Hasan. Vol. 3357. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2004, pp. 114–129. DOI: `10.1007/978-3-540-30564-4_8`.

[Kav12] Selçuk Kavut. “Results on rotation-symmetric S-boxes”. In: *Information Sciences* 201 (Oct. 2012), pp. 93–113. DOI: `10.1016/j.ins.2012.02.030`.

[KHL11] HeeSeok Kim, Seokhie Hong, and Jongin Lim. “A Fast and Provably Secure Higher-Order Masking of AES S-Box”. In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2011, pp. 95–107. DOI: `10.1007/978-3-642-23951-9_7`.

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. “Differential Power Analysis”. In: *Advances in Cryptology – CRYPTO’99*. Ed. by Michael J. Wiener. Vol. 1666. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1999, pp. 388–397. DOI: `10.1007/3-540-48405-1_25`.

[KKP+04] Daesung Kwon, Jaesung Kim, Sangwoo Park, Soo Hak Sung, Yaekwon Sohn, Jung Hwan Song, Yongjin Yeom, E-Joong Yoon, Sangjin Lee, Jaewon Lee, Seongtaek Chee, Daewan Han, and Jin Hong. “New Block Cipher: ARIA”. In: *ICISC 03: 6th International Conference on Information Security and Cryptology*. Ed. by Jong In Lim and Dong Hoon Lee. Vol. 2971. Lecture Notes in Computer Science. Springer, Heidelberg, Nov. 2004, pp. 432–445.

[KLL+14] Elif Bilge Kavun, Martin M. Lauridsen, Gregor Leander, Christian Rechberger, Peter Schwabe, and Tolga Yalçın. *Prøst v1.1*. CAESAR submission. June 2014. URL: `https://competitions.cr.yp.to/round1/proestv11.pdf`.

[KLSW17] Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. “Shorter Linear Straight-Line Programs for MDS Matrices”. In: *IACR Transactions on Symmetric Cryptology* 2017.4 (2017), pp. 188–211. ISSN: 2519-173X. DOI: `10.13154/tosc.v2017.i4.188-211`.

[KN10] Dmitry Khovratovich and Ivica Nikolic. "Rotational Cryptanalysis of ARX". In: *Fast Software Encryption – FSE 2010*. Ed. by Seokhie Hong and Tetsu Iwata. Vol. 6147. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2010, pp. 333–346. DOI: `10.1007/978-3-642-13858-4_19`.

[Knu95] Lars R. Knudsen. "Truncated and Higher Order Differentials". In: *Fast Software Encryption – FSE'94*. Ed. by Bart Preneel. Vol. 1008. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 1995, pp. 196–211. DOI: `10.1007/3-540-60590-8_16`.

[KO63] Anatolii Karatsuba and Yuri Ofman. "Multiplication of multidigit numbers on automata". In: *Soviet Physics Doklady* 7 (1963). Translated from Doklady Akademii Nauk SSSR, Vol. 145, No. 2, pp. 293–294, July 1962, pp. 595–596.

[Koc96] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems". In: *Advances in Cryptology – CRYPTO'96*. Ed. by Neal Koblitz. Vol. 1109. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 1996, pp. 104–113. DOI: `10.1007/3-540-68697-5_9`.

[Kön08] Robert Könighofer. "A Fast and Cache-Timing Resistant Implementation of the AES". In: *Topics in Cryptology – CT-RSA 2008*. Ed. by Tal Malkin. Vol. 4964. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2008, pp. 187–202. DOI: `10.1007/978-3-540-79263-5_12`.

[KPPY14] Khoongming Khoo, Thomas Peyrin, Axel York Poschmann, and Huihui Yap. "FOAM: Searching for Hardware-Optimal SPN Structures and Components with a Fair Comparison". In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2014, pp. 433–450. DOI: `10.1007/978-3-662-44709-3_24`.

[KRS19] Matthias J. Kannwischer, Joost Rijneveld, and Peter Schwabe. "Faster Multiplication in $\mathbb{Z}_{2^m}[x]$ on Cortex-M4 to Speed up NIST PQC Candidates". In: *ACNS 19: 17th International Conference on Applied Cryptography and Network Security*. Ed. by Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung. Vol. 11464.

Lecture Notes in Computer Science. Springer, Heidelberg, June 2019, pp. 281–301. DOI: 10.1007/978-3-030-21568-2_14.

[KRSS19] Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. "pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4". In: *Second NIST PQC Standardization Conference*. 2019.

[KS09] Emilia Käsper and Peter Schwabe. "Faster and Timing-Attack Resistant AES-GCM". In: *Cryptographic Hardware and Embedded Systems – CHES 2009*. Ed. by Christophe Clavier and Kris Gaj. Vol. 5747. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2009, pp. 1–17. DOI: 10.1007/978-3-642-04138-9_1.

[LAAZ11] Gregor Leander, Mohamed Ahmed Abdelraheem, Hoda AlKhzaimi, and Erik Zenner. "A Cryptanalysis of PRINTcipher: The Invariant Subspace Attack". In: *Advances in Cryptology – CRYPTO 2011*. Ed. by Phillip Rogaway. Vol. 6841. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2011, pp. 206–221. DOI: 10.1007/978-3-642-22792-9_12.

[LCM+16] Adam Langley, Wan-Teh Chang, Nikos Mavrogiannopoulos, Joachim Strombergson, and Simon Josefsson. *RFC 7905: ChaCha20-Poly1305 Cipher Suites for Transport Layer Security (TLS)*. Internet Engineering Task Force. June 2016. URL: https://tools.ietf.org/html/rfc7905.

[LF04] Jérôme Lacan and Jérôme Fimes. "Systematic MDS erasure codes based on Vandermonde matrices". In: *IEEE Communications Letters* 8.9 (2004), pp. 570–572. DOI: 10.1109/LCOMM.2004.833807.

[LLM16] David C. Lay, Steven R. Lay, and Judi J. McDonald. *Linear Algebra and Its Applications*. 5th ed. Pearson, 2016. ISBN: 978-0-321-98238-4.

[LN97] Rudolf Lidl and Harald Niederreiter. *Finite Fields*. EBL-Schweitzer. Cambridge University Press, 1997. ISBN: 9780521392310.

[LP07] Gregor Leander and Axel Poschmann. "On the Classification of 4 Bit S-Boxes". In: *Arithmetic of Finite Fields*. Ed. by Claude Carlet and Berk Sunar. Vol. 4547. Lecture Notes in Computer Science. Springer, Heidelberg, 2007, pp. 159–176. DOI: 10.1007/978-3-540-73074-3_13.

[LS16] Meicheng Liu and Siang Meng Sim. “Lightweight MDS Generalized Circulant Matrices”. In: *Fast Software Encryption – FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 101–120. DOI: `10.1007/978-3-662-52993-5_6`.

[LW16] Yongqiang Li and Mingsheng Wang. “On the Construction of Lightweight Circulant Involutory MDS Matrices”. In: *Fast Software Encryption – FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 121–139. DOI: `10.1007/978-3-662-52993-5_7`.

[LW17] Chaoyun Li and Qingju Wang. “Design of Lightweight Linear Diffusion Layers from Near-MDS Matrices”. In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017), pp. 129–155. ISSN: 2519-173X. DOI: `10.13154/tosc.v2017.i1.129-155`.

[Mat06] Mitsuru Matsui. “How Far Can We Go on the x64 Processors?” In: *Fast Software Encryption – FSE 2006*. Ed. by Matthew J. B. Robshaw. Vol. 4047. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2006, pp. 341–358. DOI: `10.1007/11799313_22`.

[Mat94] Mitsuru Matsui. “Linear Cryptanalysis Method for DES Cipher”. In: *Advances in Cryptology – EUROCRYPT’93*. Ed. by Tor Helleseth. Vol. 765. Lecture Notes in Computer Science. Springer, Heidelberg, May 1994, pp. 386–397. DOI: `10.1007/3-540-48285-7_33`.

[Max19] Alexander Maximov. *AES MixColumn with 92 XOR gates*. Cryptology ePrint Archive, Report 2019/833. 2019. URL: `https://eprint.iacr.org/2019/833`.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems – TACAS 2008*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. Lecture Notes in Computer Science. Springer, Heidelberg, 2008, pp. 337–340. DOI: `10.1007/978-3-540-78800-3_24`.

[MDA17] Silvia Mella, Joan Daemen, and Gilles Van Assche. “New techniques for trail bounds and application to differential trails in Keccak”. In: *IACR Transactions on Symmetric Cryptology* 2017.1 (2017), pp. 329–357. ISSN: 2519-173X. DOI: `10.13154/tosc.v2017.i1.329-357`.

[ME19] Alexander Maximov and Patrik Ekdahl. "New Circuit Minimization Techniques for Smaller and Faster AES SBoxes". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.4 (2019). `https://tches.iacr.org/index.php/TCHES/article/view/8346`, pp. 91–125. ISSN: 2569-2925. DOI: `10.13154/tches.v2019.i4.91-125`.

[Men16] Bart Mennink. "XPX: Generalized Tweakable Even-Mansour with Improved Security Guarantees". In: *Advances in Cryptology – CRYPTO 2016, Part I*. Ed. by Matthew Robshaw and Jonathan Katz. Vol. 9814. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 64–94. DOI: `10.1007/978-3-662-53018-4_3`.

[MGH+14] Pawel Morawiecki, Kris Gaj, Ekawat Homsirikamol, Krystian Matusiewicz, Josef Pieprzyk, Marcin Rogawski, Marian Srebrny, and Marcin Wójcik. "ICEPOLE: High-Speed, Hardware-Oriented Authenticated Encryption". In: *Cryptographic Hardware and Embedded Systems – CHES 2014*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2014, pp. 392–413. DOI: `10.1007/978-3-662-44709-3_22`.

[MN07] Mitsuru Matsui and Junko Nakajima. "On the Power of Bitslice Implementation on Intel Core2 Processor". In: *Cryptographic Hardware and Embedded Systems – CHES 2007*. Ed. by Pascal Paillier and Ingrid Verbauwhede. Vol. 4727. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2007, pp. 121–134. DOI: `10.1007/978-3-540-74735-2_9`.

[Mou15] Theodosis Mourouzis. "Optimizations in Algebraic and Differential Cryptanalysis". PhD thesis. University College London, 2015.

[MP13] Nicky Mouha and Bart Preneel. *Towards Finding Optimal Differential Characteristics for ARX: Application to Salsa20*. Cryptology ePrint Archive, Report 2013/328. 2013. URL: `https://eprint.iacr.org/2013/328`.

[MPC00] Lauren May, Lyta Penna, and Andrew J. Clark. "An Implementation of Bitsliced DES on the Pentium MMX^{TM} Processor". In: *ACISP 00: 5th Australasian Conference on Information Security and Privacy*. Ed. by Ed Dawson, Andrew Clark, and Colin Boyd. Vol. 1841. Lecture Notes in Computer Science. Springer, Heidelberg, July 2000, pp. 112–122. DOI: `10.1007/10718964_10`.

[MS77] Florence Jessie MacWilliams and Neil James Alexander Sloane. *The theory of Error-Correcting Codes*. North-Holland Publishing Company, 1977.

[NIS15a] NIST. *Secure Hash Standard (SHS)*. FIPS 180-4. Aug. 2015. DOI: `10.6028/NIST.FIPS.180-4`.

[NIS15b] NIST. *SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions*. FIPS 202. Aug. 2015. DOI: `10.6028/NIST.FIPS.202`.

[NL18] Yoav Nir and Adam Langley. *RFC 8439: ChaCha20 and Poly1305 for IETF Protocols*. Internet Research Task Force. June 2018. URL: `https://tools.ietf.org/html/rfc8439`.

[NRR06] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. "Threshold Implementations Against Side-Channel Attacks and Glitches". In: *ICICS 06: 8th International Conference on Information and Communication Security*. Ed. by Peng Ning, Sihan Qing, and Ninghui Li. Vol. 4307. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2006, pp. 529–545.

[NRS11] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. "Secure Hardware Implementation of Nonlinear Functions in the Presence of Glitches". In: *Journal of Cryptology* 24.2 (Apr. 2011), pp. 292–321. DOI: `10.1007/s00145-010-9085-7`.

[OBSC10] Dag Arne Osvik, Joppe W. Bos, Deian Stefan, and David Canright. "Fast Software AES Encryption". In: *Fast Software Encryption – FSE 2010*. Ed. by Seokhie Hong and Tetsu Iwata. Vol. 6147. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2010, pp. 75–93. DOI: `10.1007/978-3-642-13858-4_5`.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. "Cache Attacks and Countermeasures: The Case of AES". In: *Topics in Cryptology – CT-RSA 2006*. Ed. by David Pointcheval. Vol. 3860. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2006, pp. 1–20. DOI: `10.1007/11605805_1`.

[Paa97] Christof Paar. "Optimized Arithmetic for Reed-Solomon Encoders". In: *IEEE International Symposium on Information Theory*. IEEE, June 1997. ISBN: 0-7803-3956-8. DOI: `10.1109/ISIT.1997.613165`.

[PdL09] Jing Pan, Jerry den Hartog, and Jiqiang Lu. "You Cannot Hide behind the Mask: Power Analysis on a Provably Secure S-Box Implementation". In: *WISA 09: 10th International Workshop on Information Security Applications*. Ed. by Heung Youl Youm and Moti Yung. Vol. 5932. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2009, pp. 178–192. DOI: `10.1007/978-3-642-10838-9_14`.

[PL18] Jin Hyung Park and Dong Hoon Lee. "FACE: Fast AES CTR mode Encryption Techniques based on the Reuse of Repetitive Data". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018). `https://tches.iacr.org/index.php/TCHES/article/view/7283`, pp. 469–499. ISSN: 2569-2925. DOI: `10.13154/tches.v2018.i3.469-499`.

[PR13] Emmanuel Prouff and Matthieu Rivain. "Masking against Side-Channel Attacks: A Formal Security Proof". In: *Advances in Cryptology – EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, Heidelberg, May 2013, pp. 142–159. DOI: `10.1007/978-3-642-38348-9_9`.

[PV17] Kostas Papagiannopoulos and Nikita Veshchikov. "Mind the Gap: Towards Secure 1st-Order Masking in Software". In: *COSADE 2017: 8th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Sylvain Guilley. Vol. 10348. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2017, pp. 282–297. DOI: `10.1007/978-3-319-64647-3_17`.

[QS01] Jean-Jacques Quisquater and David Samyde. "ElectroMagnetic Analysis (EMA): Measures and Counter-Measures for Smart Cards". In: *Smart Card Programming and Security – E-smart 2001*. Ed. by Isabelle Attali and Thomas Jensen. Vol. 2140. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2001, pp. 200–210. DOI: `10.1007/3-540-45418-7_17`.

[RBG08] Vincent Rijmen, Paulo S. L. M. Barreto, and Décio Luiz Gazzoni Filho. "Rotation symmetry in algebraically generated cryptographic substitution tables". In: *Information Processing Letters* 106.6 (June 2008), pp. 246–250. DOI: `10.1016/j.ipl.2007.09.012`.

[RBN+15] Oscar Reparaz, Begül Bilgin, Svetla Nikova, Benedikt Gierlichs, and Ingrid Verbauwhede. "Consolidating Masking Schemes". In: *Advances in Cryptology – CRYPTO 2015, Part I*. Ed. by Rosario

Gennaro and Matthew J. B. Robshaw. Vol. 9215. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2015, pp. 764–783. DOI: `10.1007/978-3-662-47989-6_37`.

[RDP+96] Vincent Rijmen, Joan Daemen, Bart Preneel, Anton Bossalaers, and Erik De Win. "The Cipher SHARK". In: *Fast Software Encryption – FSE'96*. Ed. by Dieter Gollmann. Vol. 1039. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 1996, pp. 99–111. DOI: `10.1007/3-540-60865-6_47`.

[RIS19] RISC-V Foundation. *The RISC-V Instruction Set Manual, Volume 1: User-Level ISA, Document Version 20191213*. Ed. by Andrew Waterman and Krste Asanović. Version 20191213. Dec. 2019. URL: `https://content.riscv.org/wp-content/uploads/2019/12/riscv-spec-20191213.pdf`.

[RP10] Matthieu Rivain and Emmanuel Prouff. "Provably Secure Higher-Order Masking of AES". In: *Cryptographic Hardware and Embedded Systems – CHES 2010*. Ed. by Stefan Mangard and François-Xavier Standaert. Vol. 6225. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2010, pp. 413–427. DOI: `10.1007/978-3-642-15031-9_28`.

[RTA18] Arash Reyhani-Masoleh, Mostafa Taha, and Doaa Ashmawy. "Smashing the Implementation Records of AES S-box". In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.2 (2018). `https://tches.iacr.org/index.php/TCHES/article/view/884`, pp. 298–336. ISSN: 2569-2925. DOI: `10.13154/tches.v2018.i2.298-336`.

[SD18] Ko Stoffelen and Joan Daemen. "Column Parity Mixers". In: *IACR Transactions on Symmetric Cryptology* 2018.1 (2018), pp. 126–159. ISSN: 2519-173X. DOI: `10.13154/tosc.v2018.i1.126-159`.

[SDMO12] Mahdi Sajadieh, Mohammad Dakhilalian, Hamid Mala, and Behnaz Omoomi. "On construction of involutory MDS matrices from Vandermonde Matrices in GF(2^q)". In: *Designs, Codes and Cryptography* 64.3 (Sept. 2012), pp. 287–308. ISSN: 1573-7586. DOI: `10.1007/s10623-011-9578-x`.

[Seg55] Beniamino Segre. "Curve razionali normali e k-archi negli spazi finiti". In: *Annali di Matematica Pura ed Applicata* 39.1 (Dec. 1955), pp. 357–379. ISSN: 1618-1891. DOI: `10.1007/BF02410779`.

[Šid67] Zbyněk Šidák. “Rectangular Confidence Regions for the Means of Multivariate Normal Distributions”. In: *Journal of the American Statistical Association* 62.318 (1967), pp. 626–633. ISSN: 01621459. DOI: `10.2307/2283989`.

[SiF17] SiFive, Inc. *SiFive FE310-G000 Manual, v2p3*. Version v2p3. Oct. 2017. URL: `https://sifive.cdn.prismic.io/sifive/4d063bf8-3ae6-4db6-9843-ee9076ebadf7_fe310-g000.pdf`.

[SiF18] SiFive, Inc. *SiFive E31 Core Complex Manual, v2p0*. Version v2p0. June 2018. URL: `https://sifive.cdn.prismic.io/sifive/b06a2d11-19ea-44ec-bf53-3e4c497c7997_sifive-e31-manual-v2p0.pdf`.

[SIH+11] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. “Piccolo: An Ultra-Lightweight Blockcipher”. In: *Cryptographic Hardware and Embedded Systems – CHES 2011*. Ed. by Bart Preneel and Tsuyoshi Takagi. Vol. 6917. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2011, pp. 342–357. DOI: `10.1007/978-3-642-23951-9_23`.

[SKOP15] Siang Meng Sim, Khoongming Khoo, Frédérique E. Oggier, and Thomas Peyrin. “Lightweight MDS Involution Matrices”. In: *Fast Software Encryption – FSE 2015*. Ed. by Gregor Leander. Vol. 9054. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2015, pp. 471–493. DOI: `10.1007/978-3-662-48116-5_23`.

[SKW+98] Bruce Schneier, John Kelsey, Doug Whiting, David Wagner, Chris Hall, and Niels Ferguson. *Twofish: A 128-Bit Block Cipher*. 1998.

[SLLH18] Hwajeong Seo, Zhe Liu, Patrick Longa, and Zhi Hu. “SIDH on ARM: Faster Modular Multiplications for Faster Post-Quantum Supersingular Isogeny Key Exchange”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2018.3 (2018), pp. 1–20. ISSN: 2569-2925. DOI: `10.13154/tches.v2018.i3.1-20`. URL: `https://tches.iacr.org/index.php/TCHES/article/view/7266`.

[SM15] Tobias Schneider and Amir Moradi. “Leakage Assessment Methodology - A Clear Roadmap for Side-Channel Evaluations”. In: *Cryptographic Hardware and Embedded Systems – CHES 2015*. Ed. by Tim Güneysu and Helena Handschuh. Vol. 9293. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2015, pp. 495–513. DOI: `10.1007/978-3-662-48324-4_25`.

[SMSG16] Tobias Schneider, Amir Moradi, François-Xavier Standaert, and Tim Güneysu. "Bridging the Gap: Advanced Tools for Side-Channel Leakage Estimation Beyond Gaussian Templates and Histograms". In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 58–78. DOI: `10.1007/978-3-319-69453-5_4`.

[SMTM01] Akashi Satoh, Sumio Morioka, Kohji Takano, and Seiji Munetoh. "A Compact Rijndael Hardware Architecture with S-Box Optimization". In: *Advances in Cryptology – ASIACRYPT 2001*. Ed. by Colin Boyd. Vol. 2248. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2001, pp. 239–254. DOI: `10.1007/3-540-45682-1_15`.

[SMY09] François-Xavier Standaert, Tal Malkin, and Moti Yung. "A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks". In: *Advances in Cryptology – EUROCRYPT 2009*. Ed. by Antoine Joux. Vol. 5479. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2009, pp. 443–461. DOI: `10.1007/978-3-642-01001-9_26`.

[SS16a] Sumanta Sarkar and Siang Meng Sim. "A Deeper Understanding of the XOR Count Distribution in the Context of Lightweight Cryptography". In: *AFRICACRYPT 16: 8th International Conference on Cryptology in Africa*. Ed. by David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi. Vol. 9646. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2016, pp. 167–182. DOI: `10.1007/978-3-319-31517-1_9`.

[SS16b] Sumanta Sarkar and Habeeb Syed. "Lightweight Diffusion Layer: Importance of Toeplitz Matrices". In: *IACR Transactions on Symmetric Cryptology* 2016.1 (2016), pp. 95–113. ISSN: 2519-173X. DOI: `10.13154/tosc.v2016.i1.95-113`. URL: `https://tosc.iacr.org/index.php/ToSC/article/view/537`.

[SS16c] Peter Schwabe and Ko Stoffelen. "All the AES You Need on Cortex-M3 and M4". In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Ed. by Roberto Avanzi and Howard M. Heys. Vol. 10532. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 180–194. DOI: `10.1007/978-3-319-69453-5_10`.

[SS17] Sumanta Sarkar and Habeeb Syed. “Analysis of Toeplitz MDS Matrices”. In: *ACISP 17: 22nd Australasian Conference on Information Security and Privacy, Part II*. Ed. by Josef Pieprzyk and Suriadi Suriadi. Vol. 10343. Lecture Notes in Computer Science. Springer, Heidelberg, July 2017, pp. 3–18. DOI: `10.1007/978-3-319-59870-3_1`.

[SSA+07] Taizo Shirai, Kyoji Shibutani, Toru Akishita, Shiho Moriai, and Tetsu Iwata. “The 128-Bit Blockcipher CLEFIA (Extended Abstract)”. In: *Fast Software Encryption – FSE 2007*. Ed. by Alex Biryukov. Vol. 4593. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2007, pp. 181–195. DOI: `10.1007/978-3-540-74619-5_12`.

[STA+15] Yu Sasaki, Yosuke Todo, Kazumaro Aoki, Yusuke Naito, Takeshi Sugawara, Yumiko Murakami, Mitsuru Matsui, and Shoichi Hirose. *Minalpher v1.1*. CAESAR submission. Aug. 2015. URL: `https://competitions.cr.yp.to/round2/minalpherv11.pdf`.

[STM19] STMicroelectronics. *RM0090 reference manual*. Feb. 2019. URL: `https://www.st.com/resource/en/reference_manual/dm00031020-stm32f405415-stm32f407417-stm32f427437-and-stm32f429439-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[STM20] STMicroelectronics. *RM0038 reference manual*. Jan. 2020. URL: `https://www.st.com/resource/en/reference_manual/cd00240193-stm32l100xx-stm32l151xx-stm32l152xx-and-stm32l162xx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf`.

[Sto15] Ko Stoffelen. “Intrinsic Side-Channel Analysis Resistance and Efficient Masking”. MA thesis. Radboud University, Aug. 2015.

[Sto16a] Ko Stoffelen. “Instruction Scheduling and Register Allocation on ARM Cortex-M”. In: *Software performance enhancement for encryption and decryption, and benchmarking – SPEED-B*. Oct. 2016. URL: `https://ccccspeed.win.tue.nl/papers/armscheduler-final.pdf`.

[Sto16b] Ko Stoffelen. “Optimizing S-Box Implementations for Several Criteria Using SAT Solvers”. In: *Fast Software Encryption – FSE 2016*. Ed. by Thomas Peyrin. Vol. 9783. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 140–160. DOI: `10.1007/978-3-662-52993-5_8`.

[Sto19] Ko Stoffelen. “Efficient Cryptography on the RISC-V Architecture”. In: *LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*. Ed. by Peter Schwabe and Nicolas Thériault. Vol. 11774. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2019, pp. 323–340. DOI: `10.1007/978-3-030-30530-7_16`.

[Sug18] Takeshi Sugawara. “3-Share Threshold Implementation of AES S-box without Fresh Randomness”. In: *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019.1 (2018). `https://tches.iacr.org/index.php/TCHES/article/view/7336`, pp. 123–145. ISSN: 2569-2925. DOI: `10.13154/tches.v2019.i1.123-145`.

[SV93] Mark Shand and Jean Vuillemin. “Fast implementations of RSA cryptography”. In: *Proceedings of IEEE 11th Symposium on Computer Arithmetic*. June 1993, pp. 252–259. DOI: `10.1109/ARITH.1993.378085`.

[SVO+10] François-Xavier Standaert, Nicolas Veyrat-Charvillon, Elisabeth Oswald, Benedikt Gierlichs, Marcel Medwed, Markus Kasper, and Stefan Mangard. “The World Is Not Enough: Another Look on Second-Order DPA”. In: *Advances in Cryptology – ASIACRYPT 2010*. Ed. by Masayuki Abe. Vol. 6477. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2010, pp. 112–129. DOI: `10.1007/978-3-642-17373-8_7`.

[TLS16] Yosuke Todo, Gregor Leander, and Yu Sasaki. “Nonlinear Invariant Attack - Practical Attack on Full SCREAM, iSCREAM, and Midori64”. In: *Advances in Cryptology – ASIACRYPT 2016, Part II*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10032. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2016, pp. 3–33. DOI: `10.1007/978-3-662-53890-6_1`.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. “Efficient Cache Attacks on AES, and Countermeasures”. In: *Journal of Cryptology* 23.1 (Jan. 2010), pp. 62–74. DOI: `10.1007/s00145-009-9049-y`.

[Tri03] Elena Trichina. *Combinational Logic Design for AES SubByte Transformation on Masked Data*. Cryptology ePrint Archive, Report 2003/236. 2003. URL: `https://eprint.iacr.org/2003/236`.

[TW15] Biaoshuai Tao and Hongjun Wu. "Improving the Biclique Cryptanalysis of AES". In: *ACISP 15: 20th Australasian Conference on Information Security and Privacy*. Ed. by Ernest Foo and Douglas Stebila. Vol. 9144. Lecture Notes in Computer Science. Springer, Heidelberg, June 2015, pp. 39–56. DOI: `10.1007/978-3-319-19962-7_3`.

[VGS14] Nicolas Veyrat-Charvillon, Benoît Gérard, and François-Xavier Standaert. "Soft Analytical Side-Channel Attacks". In: *Advances in Cryptology – ASIACRYPT 2014, Part I*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8873. Lecture Notes in Computer Science. Springer, Heidelberg, Dec. 2014, pp. 282–296. DOI: `10.1007/978-3-662-45611-8_15`.

[VSP17] Andrea Visconti, Chiara Valentina Schiavo, and René Peralta. *Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF(2)*. Cryptology ePrint Archive, Report 2017/194. 2017. URL: `https://eprint.iacr.org/2017/194`.

[VSP18] Andrea Visconti, Chiara Valentina Schiavo, and René Peralta. "Improved upper bounds for the expected circuit complexity of dense systems of linear equations over GF(2)". In: *Information Processing Letters* 137 (Sept. 2018), pp. 1–5. DOI: `10.1016/j.ipl.2018.04.010`.

[vWB11] Jasper G. J. van Woudenberg, Marc F. Witteman, and Bram Bakker. "Improving Differential Power Analysis by Elastic Alignment". In: *Topics in Cryptology – CT-RSA 2011*. Ed. by Aggelos Kiayias. Vol. 6558. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2011, pp. 104–119. DOI: `10.1007/978-3-642-19074-2_8`.

[War94] William P. Wardlaw. "Matrix Representation of Finite Fields". In: *Mathematics Magazine* 67.4 (1994), pp. 289–293. ISSN: 0025-570X.

[WFY+02] Dai Watanabe, Soichi Furuya, Hirotaka Yoshida, Kazuo Takaragi, and Bart Preneel. "A New Keystream Generator MUGI". In: *Fast Software Encryption – FSE 2002*. Ed. by Joan Daemen and Vincent Rijmen. Vol. 2365. Lecture Notes in Computer Science. Springer, Heidelberg, Feb. 2002, pp. 179–194. DOI: `10.1007/3-540-45661-9_14`.

[WJW+18] Wen Wang, Bernhard Jungk, Julian Wälde, Shuwen Deng, Naina Gupta, Jakub Szefer, and Ruben Niederhagen. *XMSS and Embedded Systems – XMSS Hardware Accelerators for RISC-V*. Cryptology ePrint Archive, Report 2018/1225. 2018. URL: `https://eprint.iacr.org/2018/1225`.

[WM18a] Felix Wegener and Amir Moradi. "A First-Order SCA Resistant AES Without Fresh Randomness". In: *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Ed. by Junfeng Fan and Benedikt Gierlichs. Vol. 10815. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 245–262. DOI: `10.1007/978-3-319-89641-0_14`.

[WM18b] Felix Wegener and Amir Moradi. *A Note on Transitional Leakage When Masking AES with Only Two Bits of Randomness*. Cryptology ePrint Archive, Report 2018/1117. 2018. URL: `https://eprint.iacr.org/2018/1117`.

[Wol] Clifford Wolf. *Yosys Open SYnthesis Suite*. URL: `http://www.clifford.at/yosys/`.

[WVGX15] Junwei Wang, Praveen Kumar Vadnala, Johann Großschädl, and Qiuliang Xu. "Higher-Order Masking in Practice: A Vector Implementation of Masked AES for ARM NEON". In: *Topics in Cryptology – CT-RSA 2015*. Ed. by Kaisa Nyberg. Vol. 9048. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2015, pp. 181–198. DOI: `10.1007/978-3-319-16715-2_10`.

[ZBL+14] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. *RECTANGLE: A Bit-slice Lightweight Block Cipher Suitable for Multiple Platforms*. Cryptology ePrint Archive, Report 2014/084. 2014. URL: `https://eprint.iacr.org/2014/084`.

[ZDD+18] Liwei Zhang, A. Adam Ding, François Durvaux, François-Xavier Standaert, and Yunsi Fei. "Towards Sound and Optimal Leakage Detection Procedure (Extended Version)". In: *Smart Card Research and Advanced Applications – CARDIS 2017*. Ed. by Thomas Eisenbarth and Yannick Teglia. Vol. 10728. Lecture Notes in Computer Science. Springer, Heidelberg, Jan. 2018, pp. 105–122. ISBN: 978-3-319-75208-2. DOI: `10.1007/978-3-319-75208-2_7`.

[ZWS17] Lijing Zhou, Licheng Wang, and Yiru Sun. *On the Construction of Lightweight Orthogonal MDS Matrices*. Cryptology ePrint Archive, Report 2017/371. 2017. URL: `https://eprint.iacr.org/2017/371`.

[ZWW+14] Lei Zhang, Wenling Wu, Yanfeng Wang, Shengbao Wu, and Jian Zhang. *LAC: A Lightweight Authenticated Encryption Cipher*. CAESAR submission. Mar. 2014. URL: `https://competitions.cr.yp.to/round1/lacv1.pdf`.

[ZWZZ16] Ruoxin Zhao, Baofeng Wu, Rui Zhang, and Qian Zhang. *Designing Optimal Implementations of Linear Layers (Full Version)*. Cryptology ePrint Archive, Report 2016/1118. 2016. URL: `https://eprint.iacr.org/2016/1118`.

Summary

This thesis contains contributions that optimize various aspects of the design, implementation, and protection of symmetric cryptographic primitives. Chapters 1 and 2 introduce the topic and provide the reader with some background. The main body of this thesis is divided into three parts.

I Cryptographic Building Blocks

Chapter 3 is about how to optimize S-boxes, the nonlinear building block of many symmetric ciphers. We define four optimization criteria and we provide a method based on generic SAT solvers to automatically find efficient S-box implementations for these criteria. This method also allows to prove statements about the optimal number of operations. We then apply the method to the S-boxes of the CAESAR candidates and find many efficient implementations.

Chapter 4 tackles the optimization of MDS matrices, a linear building block of many symmetric ciphers. In the search for efficiently implementable MDS matrices, researchers used an approach that we show only performs local optimization. Existing methods to find shortest linear straight-line programs can be used in the search for MDS matrices and outperform the approach that only does local optimization. With these methods we study the XOR counts of many MDS matrices used in ciphers and of a number of generic constructions.

Chapter 5 defines the column-parity mixer (CPM), a linear building block for symmetric ciphers that can be used instead of an MDS matrix. We first study its algebraic, diffusion, and mask propagation properties and show its efficiency compared to other mixing layers. Then we outline a strategy on how to incorporate a CPM in a larger design and exemplify this be with a new permutation called Mixifer. We perform a security analysis of Mixifer and showcase that the design leads to an efficient software implementation.

II Optimized Implementations

Chapter 6 is about optimized assembly implementations of AES for the ARM Cortex-M3 and M4 that outperformed the then existing implementations. We wrote a custom instruction scheduler and register allocator for minimizing load instructions in SubBytes. Next to table-based implementations, we also show a constant-time bitsliced implementation and a masked implementation with two shares.

Chapter 7 deals with the RISC-V architecture and comes with optimized assembly implementations of AES, ChaCha, and the Keccak-f permutation. We also study the performance of arbitrary-precision arithmetic for asymmetric cryptography. Finally, we compare the relative performance of our implementations to those on the ARM Cortex-M4 and we discuss the impact of potential ISA extensions.

III Side-Channel Countermeasures

Chapter 8 describes a method to speed up implementations of higher-order masking by exploiting vector registers. We implement AES with 4 shares and with 8 shares and we use the NEON unit of an ARM Cortex-A8 to efficiently compute on the shares in parallel. We conclude with a security evaluation against side-channel attacks.

Chapter 9 introduces a way to mask AES with only two random bits in total by coming up with a new masked `AND` gate and by carefully reusing randomness. We formally verify that this approach is (first-order) secure in the probing model. We also implement it in assembly to show its efficiency on an ARM Cortex-M4 and we perform experiments to discuss its security.

Samenvatting

Dit proefschrift bevat bijdragen die verschillende aspecten optimaliseren van het ontwerp, de implementatie en de bescherming van symmetrische cryptografische primitieven. Hoofdstuk 1 en 2 leiden het onderwerp in en geven de lezer enige achtergrond. De kern van dit proefschrift is opgedeeld in drie delen.

I Cryptografische Bouwstenen

Hoofdstuk 3 gaat over optimalisatie van S-boxes, een niet-lineaire bouwsteen van veel symmetrische vercijferingen. We definiëren vier optimalisatiecriteria en we geven een methode gebaseerd op generieke SAT-solvers om automatisch efficiënte S-box-implementaties te vinden. Deze methode maakt het ook mogelijk om stellingen over het minimum te bewijzen. We passen de methode toe op S-boxes van CAESAR-kandidaten en vinden veel efficiënte implementaties.

Hoofdstuk 4 beschouwt optimalisatie van MDS-matrices, een lineaire bouwsteen van veel symmetriche vercijferingen. Voor het zoeken naar efficiënte MDS-matrices gebruikten onderzoekers een methode waarvan wij aantonen dat deze slechts aan lokale optimalisatie doet. Bestaande methoden om *shortest linear straight-line programs* te vinden kunnen hiervoor ook gebruikt worden en presteren beter. Met deze methoden bestuderen we het aantal `XOR`-operaties benodigd voor veel MDS-matrices en ook een aantal generieke constructies.

Hoofdstuk 5 definieert de *column-parity mixer* (CPM), een lineaire bouwsteen voor symmetrische vercijferingen die in plaats van een MDS-matrix gebruikt kan worden. We bestuderen o. a. algebraïsche en differentiële eigenschappen en tonen aan hoe efficiënt het is vergeleken met andere typen mixlagen. Daarna geven we een manier om een CPM in een ontwerp te gebruiken, wat resulteert in een permutatie genaamd Mixifer. We voeren een veiligheidsanalyse uit en laten zien dat het ontwerp leidt tot een efficiënte implementatie.

II Geoptimaliseerde Implementaties

Hoofdstuk 6 gaat over geoptimaliseerde assembly-implementaties van AES voor de ARM Cortex-M3 en M4 die sneller waren dan bestaande implementaties. We schreven een eigen instructieplanner en registertoewijzer om het aantal load-instructies in SubBytes te minimaliseren. Naast implementaties gebaseerd op tabellen laten we ook een *bitsliced* implementatie zien en een tot de eerste orde gemaskeerde implementatie.

Hoofdstuk 7 beschouwt geoptimaliseerde assembly-implementaties van AES, ChaCha en de Keccak-f permutatie voor de RISC-V architectuur. We kijken ook naar de prestaties van willekeurige-precisie-rekenkunde voor asymmetrische cryptografie. Ten slotte vergelijken we de relatieve prestaties van onze implementaties met die op een ARM Cortex-M4 en bespreken we de impact van mogelijke instructiesetuitbreidingen.

III Maatregelen tegen Nevenkanaalaanvallen

Hoofdstuk 8 beschrijft een methode om tot hogere orden gemaskeerde implementaties te versnellen met het gebruik van vectorregisters. We implementeren AES met 4 delen en met 8 delen en we gebruiken de NEON-eenheid van een ARM Cortex-A8 om efficiënt geparallelliseerd op de delen te rekenen. We sluiten af met een veiligheidsevaluatie tegen nevenkanaalaanvallen.

Hoofdstuk 9 introduceert een manier om AES te maskeren met in totaal slechts twee willekeurige bits door het bedenken van een nieuwe gemaskeerde `AND`-operatie en door zorgvuldig hergebruik van willekeurigheid. Met formele verificatie tonen we aan dat deze aanpak veilig is (tot de eerste orde) in het sondemodel. We implementeren het ook in assembly op een ARM Cortex-M4 om aan te tonen hoe efficiënt het is en we voeren veiligheidsexperimenten uit.

Curriculum Vitae

Ko completed gymnasium at Maurick College in Vught in 2009. He then moved to Nijmegen for a bachelor's degree in computing science at Radboud University, which was awarded cum laude in 2013. He pursued a master's degree in computer security at the Kerckhoffs Institute, a joint programme formerly offered by Eindhoven University of Technology, University of Twente, and Radboud University. Radboud University awarded the degree cum laude in 2015, after he wrote his master thesis under supervision of prof. dr. Lejla Batina. During his master's degree he additionally completed the honours programme 'Reflections on Science'.

In 2015, Ko started as a PhD candidate under supervision of prof. dr. Peter Schwabe in the Digital Security group, part of the Institute for Computing and Information Sciences at Radboud University. The position was partially funded by the EU Horizon 2020 project PQCRYPTO. This thesis is the result of that PhD.

In 2018 the research was interleaved for a few months by an internship in London as a cryptography engineer in the crypto team of Nick Sullivan at Cloudflare. Last but not least, Ko is an honorary member of both Studievereniging Thalia and NSAV 't Haasje.

List of Publications

1. Hannes Gross, Ko Stoffelen, Lauren De Meyer, Martin Krenn, and Stefan Mangard. "First-Order Masking with Only Two Random Bits". In: *Proceedings of ACM Workshop on Theory of Implementation Security*. TIS'19. ACM, 2019, pp. 10–23

2. Matthias J. Kannwischer, Joost Rijneveld, Peter Schwabe, and Ko Stoffelen. "pqm4: Testing and Benchmarking NIST PQC on ARM Cortex-M4". In: *Second NIST PQC Standardization Conference*. 2019

3. Ko Stoffelen. “Efficient Cryptography on the RISC-V Architecture”. In: *LATINCRYPT 2019: 6th International Conference on Cryptology and Information Security in Latin America*. Vol. 11774. Lecture Notes in Computer Science. Springer, Heidelberg, Sept. 2019, pp. 323–340
4. Ko Stoffelen and Joan Daemen. “Column Parity Mixers”. In: *IACR Transactions on Symmetric Cryptology* 2018.1 (2018), pp. 126–159
5. Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. “Vectorizing Higher-Order Masking”. In: *COSADE 2018: 9th International Workshop on Constructive Side-Channel Analysis and Secure Design*. Vol. 10815. Lecture Notes in Computer Science. Springer, Heidelberg, Apr. 2018, pp. 23–43
6. Thorsten Kranz, Gregor Leander, Ko Stoffelen, and Friedrich Wiemer. “Shorter Linear Straight-Line Programs for MDS Matrices”. In: *IACR Transactions on Symmetric Cryptology* 2017.4 (2017), pp. 188–211
7. Ko Stoffelen. “Instruction Scheduling and Register Allocation on ARM Cortex-M”. in: *Software performance enhancement for encryption and decryption, and benchmarking – SPEED-B*. Oct. 2016
8. Peter Schwabe and Ko Stoffelen. “All the AES You Need on Cortex-M3 and M4”. In: *SAC 2016: 23rd Annual International Workshop on Selected Areas in Cryptography*. Vol. 10532. Lecture Notes in Computer Science. Springer, Heidelberg, Aug. 2016, pp. 180–194
9. Ko Stoffelen. “Optimizing S-Box Implementations for Several Criteria Using SAT Solvers”. In: *Fast Software Encryption – FSE 2016*. Vol. 9783. Lecture Notes in Computer Science. Springer, Heidelberg, Mar. 2016, pp. 140–160