Security of symmetric primitives and their implementations

Nicolas Bordes

Université Grenoble Alpes

THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

Spécialité : Mathématiques et Informatique

Arrêté ministériel : 25 mai 2016

Présentée par

Nicolas BORDES

Thèse dirigée par:
Jean-Guillaume DUMAS, Professeur, Université Grenoble Alpes
et co-encadrée par:
Pierre KARPMAN, Maître de Conférences, Université Grenoble Alpes
et **Paolo MAISTRI**, Chargé de Recherche, CNRS

préparée au sein du **Laboratoire Jean KUNTZMANN**
dans **l'École Doctorale Mathématiques, Sciences et Technologies de l'Information, Informatique**

Sécurité des primitives symétriques et de leurs implémentations

Security of symmetric primitives and their implementations

Thèse soutenue publiquement le **9 Décembre 2021**,
devant le jury présidé par **David MONNIAUX** et composé de :

Anne CANTEAUT
Directrice de Recherche, Inria de Paris, Rapporteur
Louis GOUBIN
Professeur, Université Paris-Saclay, Rapporteur
David MONNIAUX
Directeur de Recherche, CNRS Délégation Alpes, Président
Jean-Sébastien CORON
Professeur, Université du Luxembourg, Examinateur
Thomas ROCHE
Docteur en science, Ninjalab, Examinateur
Jean-Guillaume DUMAS
Professeur, Université Grenoble Alpes, Directeur de thèse
Pierre KARPMAN
Maître de Conférences, Université Grenoble Alpes, Invité, Co-encadrant de thèse
Paolo MAISTRI
Chargé de Recherche, CNRS Délégation Alpes, Invité, Co-encadrant de thèse

This page intentionally left blank.

Remerciements

Avant tout, je tiens à remercier mon directeur de thèse, Jean-Guillaume Dumas, ainsi que mes co-encadrants, Pierre Karpman et Paolo Maistri. Merci beaucoup pour votre confiance tout au long de ces trois dernières années, confiance qui a été cruciale dans les moments de doutes.

Plus particulièrement : merci à Paolo de m'avoir permis de faire le pont entre la théorie et la pratique ainsi que pour son soutien matériel indispensable ; merci à Jean-Guillaume pour son expérience, sa bienveillance, sa passion, son exigence et sa curiosité, qui se sont ressenties depuis les premiers cours auxquels j'ai pu assister jusqu'aux dernières minutes de mon doctorat ; enfin, Pierre, à qui je dois en grande partie mon éclosion scientifique, merci pour le chemin parcouru ensemble, collaboration quotidienne pour laquelle je me sens à la fois fier et extrêmement reconnaissant, et merci pour les discussions parfois enflammées, souvent agitées, mais toujours enrichissantes, divertissantes, et rafraichissantes.

Merci à l'ensemble des personnes composant le jury lors de la soutenance de ma thèse, Anne Canteaut, Louis Goubin, David Monniaux, Jean-Sébastien Coron, et Thomas Roche pour votre disponibilité, votre interêt et votre rigueur d'écoute. Je remercie tout particulièrement Anne Canteaut et Louis Goubin pour avoir accepté de rapporter mon manuscrit de thèse.

Je ne remercie pas le SARS-CoV2 ainsi que l'ensemble de ces variants : j'ai toujours aimé les challenges, mais celui de faire un doctorat en pleine pandémie, je m'en serai bien passé.

Un merci très chaleureux à mes parents ainsi qu'à mon frère et ma sœur, qui m'ont vu grandir et qui ont réussi à supporter mon caractère, mes questions et mes (parfois longs) exposés lors des repas. Merci aussi à Mamette et à Tatie qui, même depuis l'Ariège, ont eu sur moi un regard bienveillant et sans cesse encourageant.

Merci à Grenoble qui m'a accueilli depuis 2015, à ses montagnes qui semblent veiller sur moi malgré mon insistance à ne pas les visiter très souvent, ainsi qu'à ses bars, notamment le Dr D, qui réchauffent les corps et les cœurs lors des soirées d'hiver et les rafraichissent lors de celles d'été. Évidemment, merci à tous mes amis Grenoblois qui rendent ces lieux vivants et qui sont le rappel quotidien que l'échange enrichit l'existence, que ce soit lors de fêtes sous de curieux chapeaux, lorsque l'on débat et réfléchit à de meilleurs lendemains ou lorsque l'on s'envoie des boules de neige devant le Shannon. Merci tout particulièrement à Bruno qui lui aussi a décidé de partir dans cette aventure un peu folle qu'est le doctorat et à Thomas sur qui je peux toujours compter, notamment pour discuter échecs et maths.

Je tiens aussi à remercier tous les membres passés et présents de l'association Securimag, dans laquelle a pu librement se développer ma passion pour la sécurité informatique.

Merci aussi aux Zigottos qui font la démonstration, chaque année depuis la fin du lycée, qu'on ne mesure pas une amitié à la distance nous séparant mais à sa capacité à nous réunir régulièrement et à rire ensemble.

Merci à Antide : un colocataire, un acolyte, peut-être un jour un collègue, mais surtout un ami avec qui ça a été un plaisir de partager un appartement, la gestion de Securimag et l'organisation de GreHack, des discussions techniques et, dans tous ces contextes, un grand respect l'un envers l'autre et des moments de complicité.

Enfin, je tiens à exprimer ma plus grande reconnaissance à Manon, qui a bien plus compté que ce que je ne pourrais jamais réussir à imprimer et qui a, par sa présence, son amour et sa douceur, été un pilier essentiel de ma stabilité durant ces trois dernières années. Alors, pour les plaids et les balades, pour les théières et les goûters, pour Java, pour les mots et les silences, pour les sourires : merci.

This page intentionally left blank.

Aux jardins et à ces sentiers qui bifurquent.

Contents

This page intentionally left blank.

General introduction

Symmetric cryptography is a subfield of modern cryptography where all the legitimate participants share the same knowledge and secrets. Protocols based on symmetric cryptography are used in a wide range of applications, *e.g.* to protect the confidentiality of a message (using *symmetric encryption schemes*) or to protect its integrity (*Message Authentication Codes*, MAC). In the vast majority of cases, such cryptographic protocols rely on lower-level and specialised cryptographic components called *primitives*. This modular approach allows to study the security of the primitives independently of the security of the whole protocol, effectively producing complex symmetric cryptosystems from the careful composition of secure smaller ones. There exists many different examples of symmetric primitives, with each answering some specific needs. For example, *hash functions* are commonly used in the design of MACs whereas *block ciphers* or *permutations* are often employed inside symmetric encryption schemes.

In this thesis, we study the security of the design and implementation of some cryptographic primitives, from their design to their implementations in presence of adversaries.

First, we study some security properties of a specific kind of cryptographic primitives, *cryptographic permutations*. Permutations can be used in a variety of cryptographic protocols such as symmetric encryption schemes or can also appear as a building block of other primitives like hash functions. There exists multiple approaches to the underlying design of a permutation used in symmetric cryptography. One of them, popularized by the Advanced Encryption Standard (AES), consists in partitioning the bits of the state in non-trivial groups, *e.g.* in bytes, and consistently processing them in these groups. This approach can be seen as *aligned*, in a certain sense of alignment. One of the benefits of such an approach, is that it leads to structures that make it easier to reason about differential and linear properties (the main properties used by statistical attacks against primitives, namely differential and linear attacks). Using combinatorial arguments, one may thus be able to bound the applicability of those attacks. In contrast, an *unaligned* approach avoids any meaningful grouping of states bits in its design. The major drawback is that the same combinatorial arguments can no longer be used to describe the differential and linear properties of such permutations. However, it also means that there is much less structure to be exploited by an adversary.

The goal of the first part of this thesis is to define formally what it means for a permutation to be aligned and to study its concrete impact on the differential and linear properties. To do so, we analyse four different permutations that have in common that they are constructed with a "bottom-up" approach. In fact, they are all built from the successive application of two underlying components: a linear layer and a nonlinear layer. However, the four primitives are exponents of four different design strategies with three that are deemed to be aligned and one unaligned. Even if the nonlinear and linear layers have been thoroughly studied in the literature, their interaction has much less often been studied in a formal framework. We present a quantitative framework based on the construction of histograms to analyse these interactions and use it to compare the aligned approaches to the unaligned one. This part takes its source from a joint work with Joan Daemen, Daniël Kuijsters and Gilles Van Assche and the results have been published in the proceedings of CRYPTO 2021.

In the second part of this thesis we focus on another important aspect in the security of cryptographic primitives: the protection of their implementations. More specifically, we study a class of attacks, *side-channel attacks*, where an attacker may be able to extract the secrets of a cryptographic algorithm by only measuring physical leakages from the components computing

it. In the right conditions, often met when targeting embedded devices, these attacks can be devastatingly effective against a cryptosystem and its role in the overall security of the device on which it has been implemented. Nonetheless, this class of attacks can be counteracted, or at least their efficiency can be reduced, by artificially increasing the measurement noise which will affect the precision of the information gathered by an attacker. One such countermeasure, called *masking*, leverages secret-sharing schemes to split the sensitive data into uniformly random and individually independent shares. Doing so forces the attacker to be able to get access to multiple shares simultaneously in order to be able to extract secret information. The number of shares needed by an attacker, called the masking order, is thus a good estimator of the level of security provided by this countermeasure: as it grows, the attacker needs increasingly more measurement to reach their goal. However, the main challenge from the designer point-of-view is to be able to securely compute the target cryptographic primitive given not directly the inputs, but rather a sharing of them. One major obstacle to such secure implementations is that their cost rapidly increases in the masking order. The overhead that arises at high orders (that is, when the masking order is strictly greater than one) can sometimes be impracticable, especially for applications on embedded devices. Moreover, the verification that a masked implementation is indeed secure in a given security model can be, in itself, computationally intractable at such high orders and formal proofs are often not provided, at least not for arbitrary orders.

In this thesis, we show how to improve the performance of both the verification and execution cost of masked implementations at high orders. To do so, we develop a new algorithm to verify the security of high-order masked components in different security models faster than the state of the art. Additionally, by modifying already existing masked designs and by being able to efficiently assess their security using our verification algorithm, we present more efficient high-order masking schemes. An article on this subject, with Pierre Karpman as a co-author, has been published in the proceedings of EUROCRYPT 2021. Following these results, we also propose a fully masked version of the AES that is less costly than what is already done in the literature and we experimentally verify its robustness against side-channel attacks by following a leakage assessment methodology.

Part I

Alignment

Overview

In this part of the manuscript, we focus on a specific kind of symmetric primitives called cryptographic permutations, that are invertible functions from a finite space of states to itself. These permutations can be used to build more complex cryptographic algorithms as symmetric encryption schemes or hash functions. The use of permutations as primitive is conditioned on them having "good" properties and what is exactly meant by that is studied in a subfield of symmetric cryptography: *permutation-based cryptography*. One of the interesting properties a permutation must have in order to be useful in a cryptographic context is its resistance against differential and linear cryptanalysis. These attacks exploit the structure of the permutation to find relations between its input and output that are verified with a non-negligible probability.

The design of permutations is thus focused on having, among other properties, sufficiently good differential and linear properties while still being as efficient as possible with respect to different metrics of cost, such as the number of cycles to execute, the memory consumption or even the area taken by hardware implementations. In our work, we more specifically focus on an approach widely used consisting in designing permutations from smaller *round functions*, that are meant to be successively composed to build the full permutation. We also focus on round functions that are made from two distinct components: a linear one and a nonlinear one. Theses two components may have distinct but sometimes overlapping roles in the overall properties of the permutation.

In turn, the linear and nonlinear components of a round function can be designed in multiple ways. In this part, we propose a way to classify these designs in two categories: *aligned* and *unaligned*. This classification comes from the observation that for some permutations, the input bits are grouped together, *e.g.* in bytes, and are consistently processed using operations that never break these groups. Rijndael, a block cipher that later became the Advanced Encryption Standard (AES), popularized this *aligned* approach because it allows the use of convenient combinatorial arguments to reason on its differential and linear properties. However, some permutations have also been designed specifically with the goal of making sure that such groupings do not exist. Such *unaligned* permutations are constructed in the hope that it leads to better differential and linear properties, but at the same time it makes their analysis more difficult. To bring light on the compared behaviour of *aligned* and *unaligned* approaches, we study four permutations, Rijndael, Spongent, Saturnin and Xoodoo, with different design philosophy.

In Chapter 1, we introduce the concepts needed in the other chapters, formalize the notion of alignment and present in more details the four permutations studied as well as their alignment properties. Then, Chapter 2 defines metrics and associated histograms that are used to quantitatively compare the differential and linear properties of the four permutations. Finally, we study the approximations that are sometimes made in the description of the differential and linear properties in Chapter 3 and show that the confidence in their accuracy can be higher for unaligned permutations than for aligned ones.

This part takes it source from a joint work with Joan Daemen, Daniël Kuijsters and Gilles Van Assche and the results have been published in the proceedings of CRYPTO 2021 [BDK+21].

Chapter 1

Introduction to alignment

Contents

1.1 Notation

We use the following conventions and notation:

— We define $[\![1,\, k]\!] = \{i \in \mathbb{N} \mid 1 \leq i \leq k\}$.

— Given a set $\mathcal{S}$ and an equivalence relation $\sim$ on $\mathcal{S}$, we write $[\boldsymbol{a}]_\sim$ for the equivalence class of $\boldsymbol{a} \in \mathcal{S}$:

$$[\boldsymbol{a}]_\sim = \{\boldsymbol{a}' \in \mathcal{S} \mid \boldsymbol{a}' \sim \boldsymbol{a}\}\,.$$

— We denote the cardinality of $\mathcal{S}$ by $\#\mathcal{S}$.

— We use the term *state*, *difference* or *linear mask* for a vector of b bits, with b specified or clear from the context.

— Given a state $\boldsymbol{a} \in \mathbb{F}_2^b$, we refer to its i-th component as $\boldsymbol{a}_i$.

— We consider index sets $\mathrm{B}_i \subseteq [\![1,\, b]\!]$ that form a partition of the full index set $[\![1,\, b]\!]$.

— We write $P_i(\boldsymbol{a})\colon \mathbb{F}_2^b \to \mathbb{F}_2^{\#\,\mathrm{B}_i}$ for the projection onto the bits of $\boldsymbol{a}$ indexed by B_i.

— We write $\boldsymbol{e}_i^k$ for the i-th standard basis vector in $\mathbb{F}_2^k$, *i.e.* for $j \in [\![1,\, k]\!]$ we have that $\boldsymbol{e}_{ij}^k = 1$ if $i = j$ and 0 otherwise.

— Permutations of the index space are written as $\tau\colon [\![1,\, b]\!] \to [\![1,\, b]\!]$; By *shuffle (layer)*, we mean a linear transformation $\pi\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ given by $\pi(\boldsymbol{a}) = \mathrm{P}_\tau\, \boldsymbol{a}$, where P_τ is the permutation matrix associated with some τ, *i.e.* obtained by permuting the columns of the $(b \times b)$ identity matrix according to τ.

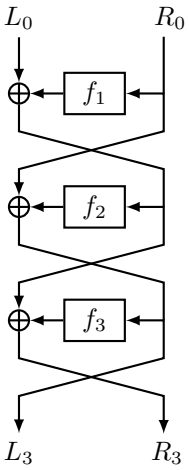
— Given a linear transformation $\mathrm{L}\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$, there exists a matrix $\boldsymbol{M} \in \mathbb{F}_2^{b\times b}$ such that $\mathrm{L}(\boldsymbol{a}) = \boldsymbol{M}\boldsymbol{a}$, with the state $\boldsymbol{a}$ seen as a column vector. We define its transpose $\mathrm{L}^t\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ by $\mathrm{L}^t(\boldsymbol{a}) = \boldsymbol{M}^t\boldsymbol{a}$.

1.2 Preliminaries

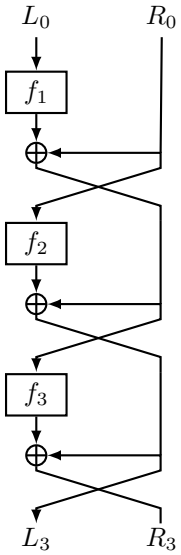
In this part we study bijective functions from $\{0,1\}^b$, the finite space of bit-strings of length b, to itself. These functions are called permutations of $\{0,1\}^b$. Permutations are symmetric primitives that can be used as modules to implement more complex cryptosystems such as symmetric encryption schemes (*e.g.* as in ChaCha [Ber08] or using the more generic duplex construction [BDP+11]) and hash functions (*e.g.* as in the SHA-3 finalist JH [Wu11] or the sponge construction [BDP+11]).

Often, a permutation f used as a building block for cryptographic primitives is designed as the composition of a number of lightweight *round functions*, *i.e.* $f = \mathrm{R}_r \circ \cdots \circ \mathrm{R}_2 \circ \mathrm{R}_1$ for some number of rounds $r \in \mathbb{N}^*$. We write $f[k] = \mathrm{R}_k \circ \cdots \circ \mathrm{R}_1$ the successive application of k rounds, and define $f[0] = \mathrm{id}$ with id the identity function. There are multiple approaches to the design of the round function itself. One can use a (generalized) Feistel network [Fei73] with the help of underlying smaller functions f_i. Three rounds of a two-branch Feistel network are illustrated in Figure 1.1a. Another approach can be to use a MISTY structure [Mat96] as shown in Figure 1.1b. However, in this part of the thesis we focus on yet another type of round function design often called a Substitution Permutation Network (SPN).

In an SPN, the round function R_i is further decomposed into *step functions*, *i.e.* $\mathrm{R}_i = \iota_i \circ \mathrm{L}_i \circ \mathrm{N}_i$, where N_i is a nonlinear map, L_i is a linear map, and ι_i is the addition of a round constant. Apart from the round constant addition, these step functions are often, but not always, identical. For this reason, we often simply write N or L, without referring to an index if the context allows it, and call them the *nonlinear layer* and *linear layer* respectively. When the nonlinear layer can be further decomposed as the parallel application of smaller nonlinear maps S_i, we call these



(a) Three rounds of a Feistel network (figure from [Jea15]).



(b) Three rounds of a MISTY network.

Figure 1.1: Two different approaches to the design of a round function.

maps S-Boxes, write n for the number of S-boxes of N and denote their size by m. In this context, the index set of a whole state $[\![1, b]\!]$ can often be naturally partitioned along the boundaries of S-Boxes with index sets $\mathrm{B}_i = [\![(i-1)m+1, im+1]\!]$. An example of such a construction can be found in Example 1.1.

Example 1.1: Toy round function R_t

Let R_t be a toy round function working on states of length $b_t = 12$ bits. It is built as $\mathrm{R}_t = \mathrm{L}_t \circ \mathrm{N}_t$ with:

— N_t, a nonlinear layer composed of the parallel application of $n_t = 4$ S-boxes S_t each of length $m_t = 3$ bits, where:

$$\mathrm{S}_t \ : \ \mathbb{F}_2^3 \to \mathbb{F}_2^3, (\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2) \mapsto ((\boldsymbol{a}_1 \oplus 1).\boldsymbol{a}_2, (\boldsymbol{a}_2 \oplus 1).\boldsymbol{a}_0, (\boldsymbol{a}_0 \oplus 1).\boldsymbol{a}_1)\,;$$

— $\mathrm{L}_t = \pi_t \circ \mathrm{M}_t$, a linear layer such that:

— M_t is a linear layer composed of the parallel application of two identical linear maps, each of length 6 bits, that can be seen as the left multiplication by the binary matrix

$$\boldsymbol{M}_t = \begin{pmatrix} 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 0 & 1 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \end{pmatrix};$$

— π_t a shuffle layer that swaps the first three bits $[\![1, 3]\!]$ with the three bits at position $[\![7, 9]\!]$, in the same order.

Three successive applications of this toy round function are shown in Figure 1.2.

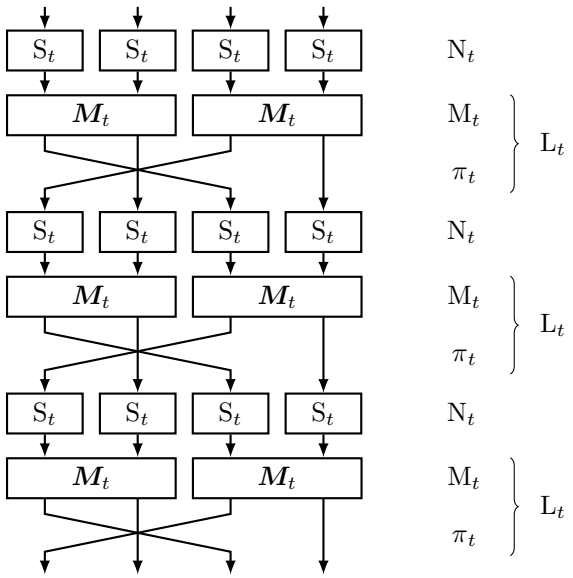


Figure 1.2: Three successive applications of the toy round function R_t.

A block cipher E is not a permutation but rather a family of permutations E_k indexed by a key k. In this work, we implicitly study the permutation obtained from a given block cipher by fixing a key, *e.g.* by considering the block cipher with an all-zero key $E_{\mathbf{0}}$.

1.3 Differential and linear cryptanalysis

1.3.1 Differential cryptanalysis

Differential cryptanalysis [BS90] is framework for chosen-plaintext attacks that exploits the non-uniformity of the distribution of differences at the output of a permutation when it is applied to pairs of inputs with a fixed difference. We call an ordered pair of an input and output difference $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) \in (\mathbb{F}_2^b)^2$ a differential. Given a differential, we are interested in its solution set:

Definition 1.3.1 (Solution set)**.** Let $f\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ be a permutation and define

$$U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) = \{\boldsymbol{x} \in \mathbb{F}_2^b \mid f(\boldsymbol{x}) + f(\boldsymbol{x} + \boldsymbol{\Delta}_{\text{in}}) = \boldsymbol{\Delta}_{\text{out}}\}\,.$$

We call $U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ the solution set of the differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$.

If there exists an ordered pair $(\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{\Delta}_{\text{in}})$ with $\boldsymbol{x} \in U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$, then it is said to follow the differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$. We then give the probability that a uniformly random state follow the given differential using its differential probability.

Definition 1.3.2 (Differential probability)**.** The *differential probability* (DP) of a differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ over the permutation $f\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ is defined as:

$$\mathrm{DP}_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) = \frac{\#U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})}{2^b}\,.$$

When a differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ is such that $U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) \neq \emptyset$ or equivalently that $\mathrm{DP}_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) \neq 0$, we say that the input difference $\boldsymbol{\Delta}_{\text{in}}$ is *compatible* with the output difference $\boldsymbol{\Delta}_{\text{out}}$ through f and call $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ a *valid* differential.

Example 1.2: Valid differential and differential probability

Let S_t be the S-box defined in Example 1.1. Let $\boldsymbol{\Delta}_{\text{in}} = (1\,0\,0)$ and $\boldsymbol{\Delta}_{\text{out}} = (1\,1\,0)$.

Since $\mathrm{S}_t((0\,1\,1)) = (0\,0\,1)$ and $\mathrm{S}_t((1\,1\,1)) = (1\,1\,1)$, we have that $\mathrm{S}_t((0\,1\,1)) + \mathrm{S}_t((0\,1\,1) + \boldsymbol{\Delta}_{\text{in}}) = (1\,1\,0) = \boldsymbol{\Delta}_{\text{out}}$. This implies that $(0\,1\,1) \in U_{\mathrm{S}_t}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ (and $(1\,1\,1) \in U_{\mathrm{S}_t}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$). Thus, $U_{\mathrm{S}_t}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) \neq \emptyset$ and the differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ is valid over an S-Box S_t.

There are only two vectors (namely $(0\,1\,1)$ and $(1\,1\,1)$) in $U_{\mathrm{S}_t}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$, thus the differential probability of $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ is $\mathrm{DP}_{\mathrm{S}_t}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) = 2/2^{m_t} = 2^{-2}$.

When studying permutations that can be seen as the application of multiple round functions, we are often interested in decomposing the differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ into a sequence of intermediate differences. Such a decomposition is called a differential trail (or characteristic).

Definition 1.3.3 (Differential trail)**.** A sequence $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)}) \in \left(\mathbb{F}_2^b\right)^{k+1}$ that satisfies $\mathrm{DP}_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}) > 0$ for $0 \leq i \leq k-1$ is called a k-round *differential trail*.

Sometimes we specify a trail as $Q = (\boldsymbol{b}_{-1}, \boldsymbol{a}_0, \boldsymbol{b}_0, \ldots, \boldsymbol{a}_k, \boldsymbol{b}_k)$ by giving the additional intermediate differences between the nonlinear layers N_i and the linear layers L_i of each round considered. Here, $\boldsymbol{b}_i = \mathrm{L}_i(\boldsymbol{a}_i) = \boldsymbol{q}_{i+1}$.

Example 1.3: Differential trail

Let $\mathrm{R}_t = \pi_t \circ \mathrm{M}_t \circ \mathrm{N}_t$ be as defined in Example 1.1. Let $Q = (\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\delta}^{(0)}, \boldsymbol{\delta}^{(1)}, \boldsymbol{\Delta}_{\text{int}}, \boldsymbol{\Delta}_{\text{out}})$ be a differential trail over one and a half round $f = \mathrm{N}_t \circ \mathrm{R}_t$ without the last linear layer $\pi_t \circ \mathrm{M}_t$ where each difference is defined before and after each step function.

As an example, we take $\boldsymbol{\Delta}_{\text{in}} = (0\,0\,0\,0\,0\,0\,0\,1\,0\,0\,1\,0\,0)$, $\boldsymbol{\Delta}_{\text{int}} = (1\,0\,1\,0\,0\,0\,0\,0\,0\,0\,1\,0\,1)$, and $\boldsymbol{\Delta}_{\text{out}} = (0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\,1)$ This differential trail is shown in Figure 1.3. Since M_t and π_t are linear, the values of $\boldsymbol{\delta}^{(0)}$ and $\boldsymbol{\delta}^{(1)}$ are fully determined by the one of $\boldsymbol{\Delta}_{\text{int}}$ but are shown to illustrate the difference propagation through each function.

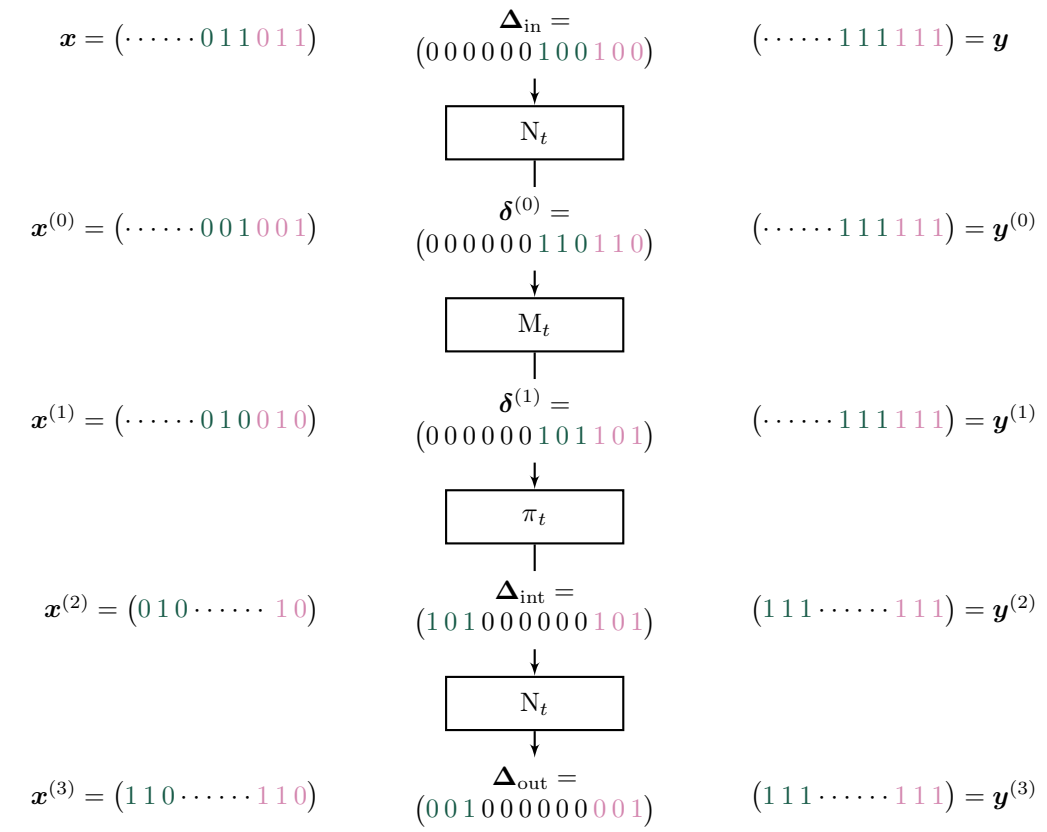


Figure 1.3: Example of a trail over $f = \mathrm{N}_t \circ \mathrm{R}_t$.

In Figure 1.3 we also give an example of a pair $(\boldsymbol{x}, \boldsymbol{y})$ following this differential trail and the values $\boldsymbol{x}^{(i)}$ and $\boldsymbol{y}^{(i)}$ they take at each step. Thanks to the inherent structure of the round function R, the six bits corresponding to the first half of the state before the shuffle π_t (denoted by "$.''$") can be chosen arbitrarily as long as bits at the same index in $\boldsymbol{x}$ and $\boldsymbol{y}$ are equal, that is $\forall i \in [\![1, 6]\!], \boldsymbol{x}_i = \boldsymbol{y}_i$. This means that the pair $(\boldsymbol{x}, \boldsymbol{y})$ can be seen as an equivalence class of pairs of size 2^6.

Given a valid differential, there may exist multiple differential trails:

Definition 1.3.4 (Enveloping differential, differential trail core and clustering of trails)**.** We write $\mathrm{DT}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ for the set of all differential trails with $\boldsymbol{q}^{(0)} = \boldsymbol{\Delta}_{\mathrm{in}}$ and $\boldsymbol{q}^{(k)} = \boldsymbol{\Delta}_{\mathrm{out}}$. We call $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ the *enveloping differential* of the trails in $\mathrm{DT}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$.

By deleting the initial difference $\boldsymbol{\Delta}_{\mathrm{in}}$ and final difference $\boldsymbol{\Delta}_{\mathrm{out}}$ of a differential trail $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k-1)}, \boldsymbol{\Delta}_{\mathrm{out}})$ we are left with a *differential trail core* $(\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k-1)})$.

If $\#\mathrm{DT}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) > 1$, then we say that trails *cluster* together in the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$.

A differential trail core obtained by deleting the initial $\boldsymbol{\Delta}_{\mathrm{in}}$ and final $\boldsymbol{\Delta}_{\mathrm{out}}$ difference of a differential trail is said to be *in* the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$.

We now define the differential probability of a differential trail.

Definition 1.3.5 (Differential trail probability)**.** Let $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)}) \in \left(\mathbb{F}_2^b\right)^{k+1}$ be a differential trail over k rounds. Each round differential $(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)})$ has a non-empty solution set $U_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)})$ because, by definition, we have $\mathrm{DP}_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}) > 0$ for $0 \leq i \leq k-1$. Consider the transformed sets of points $\mathcal{U}_i = f[i]^{-1}(U_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}))$ at the input of f. For an ordered pair $(\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{q}^{(0)})$ to follow the differential trail, it is required that $\boldsymbol{x} \in U_f(Q) = \bigcap_{i=0}^{k-1} \mathcal{U}_i$. The fraction of states $\boldsymbol{x}$ that satisfy this equation is the differential probability of the trail.

The differential probability DP of a differential trail is defined as:

$$\mathrm{DP}_f(Q) = \frac{\#U_f(Q)}{2^b}$$

Any given ordered pair $(\boldsymbol{x}, \boldsymbol{x} + \boldsymbol{\Delta}_{\mathrm{in}})$ follows exactly one differential trail. Hence, the differential probability of the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ is the sum of the differential probabilities of all differential trails with initial difference $\boldsymbol{\Delta}_{\mathrm{in}}$ and final difference $\boldsymbol{\Delta}_{\mathrm{out}}$:

$$\mathrm{DP}_f(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) = \sum_{Q \in \mathrm{DT}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})} \mathrm{DP}_f(Q)\,.$$

As a first approximation, one may consider that there are only few differential trails for the same differential which results in the differential probability being approximated to the differential trail. However, the more differential trails cluster together the furthest this approximation is from the real value of the differential probability.

While it is hard in general to compute the differential probability of a differential over a function f, the differential probability of any differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ over a round function $\mathrm{R} = \iota \circ \mathrm{L} \circ \mathrm{N}$ where N consists in the parallel application of n S-boxes S_i is easy to compute. We start by specifying the intermediate differences as a differential trail $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{\Delta}_{\mathrm{out}})$. Thanks to the linearity of L, we have $\boldsymbol{c} = \mathrm{L}(\boldsymbol{b})$ and due to the fact that a difference is invariant under addition of a constant, all valid such differential trails are of the form $(\boldsymbol{\Delta}_{\mathrm{in}}, \mathrm{L}^{-1}(\boldsymbol{\Delta}_{\mathrm{out}}), \boldsymbol{\Delta}_{\mathrm{out}}, \boldsymbol{\Delta}_{\mathrm{out}})$. Therefore, the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ contains only a single trail and its differential probability is the differential probability of the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \mathrm{L}^{-1}(\boldsymbol{\Delta}_{\mathrm{out}}))$ over the S-box layer N:

$$\mathrm{DP}_{\mathrm{R}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) = \prod_{1 \leq j \leq n} \mathrm{DP}_{\mathrm{S}_j}(P_j(\boldsymbol{\Delta}_{\mathrm{in}}), P_j(\mathrm{L}^{-1}(\boldsymbol{\Delta}_{\mathrm{out}}))) \tag{1.1}$$

where P_j is the projection over the j-th S-Box. Hence, the differential probability of a round differential is the product of the differential probabilities of its S-box differentials.

However, the differential probability of a differential trail may not be equal to the product of probability of each of its round differentials. The concept of independence of round differentials follows from this observation:

Definition 1.3.6 (Independence of round differentials)**.** The round differentials of a given differential trail $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)}) \in \left(\mathbb{F}_2^b\right)^{k+1}$ are said to be *independent* if and only if

$$\mathrm{DP}_f(Q) = \prod_{i=1}^{k} \mathrm{DP}_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)})\,.$$

Example 1.4: Differential trail clustering and round dependence

Example 1.3 shows a differential trail $Q = (\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{int}}, \boldsymbol{\Delta}_{\mathrm{out}})$ over $f = \mathrm{N}_t \circ \mathrm{R}_t$ as in Example 1.1. The differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ is the enveloping differential of Q. $\boldsymbol{\Delta}_{\mathrm{int}}$ is called the trail core of Q and $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ is the enveloping differential of Q.

The differential probability of each round differential can be computed using Equation (1.1). In our case, we have that each of the two round differentials is equal to 2^{-4}. Since it is computationally easy to exhaustively visit every possible input state of our toy permutation (there are only $2^{12} = 4096$ different states), the exact differential probability of Q can be computed: there are exactly $2 \times 2^6 = 2^7$ pairs following this differential trail, which implies that $\mathrm{DP}(Q) = 2^7/2^{b_t} = 2^{-5}$. This probability is not equal to the product of the probabilities of the round differentials, 2^{-8}. Thus, the round differentials are not independent in the sense of Definition 1.3.6.

However, Q is not the only trail in its enveloping differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$. In fact, there are three other trails in $\mathrm{DT}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$. These trails $Q' = (\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}'_{\mathrm{int}}, \boldsymbol{\Delta}_{\mathrm{out}})$, $Q'' = (\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}''_{\mathrm{int}}, \boldsymbol{\Delta}_{\mathrm{out}})$ and $Q''' = (\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}'''_{\mathrm{int}}, \boldsymbol{\Delta}_{\mathrm{out}})$ differ only in their trail core: $\boldsymbol{\Delta}'_{\mathrm{int}} = \left(0\,1\,1\,0\,0\,0\,0\,0\,0\,0\,1\,1\right)$, $\boldsymbol{\Delta}''_{\mathrm{int}} = \left(1\,1\,1\,0\,0\,0\,0\,0\,0\,1\,1\,1\right)$ and $\boldsymbol{\Delta}'''_{\mathrm{int}} = \left(0\,0\,1\,0\,0\,0\,0\,0\,0\,0\,0\,1\right)$. Thus, we say that Q, Q', Q'', Q''' cluster together. This effect is shown in Figure 1.4.

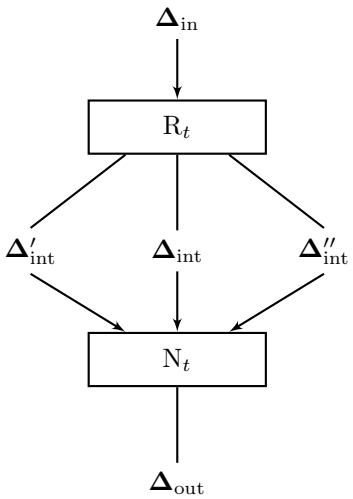


Figure 1.4: Illustration of trail clustering over $f = \mathrm{N}_t \circ \mathrm{R}_t$.

The number of pairs that follow Q' is also equal to 2×2^6. The same is true for Q'' and Q'''. The differential probability of $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ is the sum of the probability of all differential that cluster inside it, which is:

$$\mathrm{DP}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) = 4 \times 2^{-5} = 2^{-3}\,.$$

This is a non-negligible variation from the approximation $\mathrm{DP}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) \approx \mathrm{DP}(Q) = 2^{-5}$.

We now introduce the concept of restriction weight of a differential.

Definition 1.3.7 (Restriction weight of a differential)**.** The *restriction weight* of a differential $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ that satisfies $\text{DP}_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) > 0$ is defined as:

$$\text{w}_\text{r}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) = -\log_2 \text{DP}_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})\,.$$

To motivate the definition, consider the set $U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$. If we suppose that $U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ is a non-empty affine space, then we have $\text{w}_\text{r}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}}) = b - \dim_{\mathbb{F}_2} U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$. Hence, in this case the weight equals the number of independent affine equations describing $U_f(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$.

For a differential, its restriction weight is directly tied to its differential probability. However, for a differential trail, the restriction weight is instead defined as the sum of the restriction weights of every round differentials:

Definition 1.3.8 (Restriction weight of a differential trail)**.** The restriction weight of a differential trail $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)})$ is defined as

$$\text{w}_\text{r}(Q) = \sum_{i=0}^{k-1} \text{w}_\text{r}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)})\,.$$

In general, the weight of a differential trail is *easy* to compute since the weight of each round differential can be computed from its differential probability given by Equation (1.1), whereas the differential probability of a differential trail might be difficult to compute when the round differentials are not supposed independent in the sense of Definition 1.3.6. In the case where the round differentials are independent, or supposed independent for an approximation of the differential trail probability, then we have that $\text{DP}_f(Q) = 2^{-\text{w}_\text{r}(Q)}$. However, this approximation is always false when $\text{w}_\text{r}(Q) > b - 1$, since by definition of a differential trail $\text{DP}_f(Q) \geq 2^{1-b}$ because Q must be followed by at least two ordered pairs of states. Example 1.5 shows an example of the computation of the weight of a differential trail.

Example 1.5: Weight of a differential trail

Let $Q = (\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{int}}, \boldsymbol{\Delta}_{\text{out}})$ be a differential trail as in Examples 1.3 and 1.4 over $f = \text{N}_t \circ \text{R}_t$ as in Example 1.1.

Then, the weight of Q can be computed by adding the restriction weight of $Q_1 = (\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{int}})$ and $Q_2 = (\boldsymbol{\Delta}_{\text{int}}, \boldsymbol{\Delta}_{\text{out}})$. As explained before, the differential probability of Q_1 can be computed by multiplying the probability of the differential over each of its S-box, which gives:

$$\text{DP}(Q_1) = \text{DP}(((1\,0\,0), (1\,1\,0))) \times \text{DP}(((1\,0\,0), (1\,1\,0)) = 2^{-2} \times 2^{-2} = 2^{-4}\,.$$

See Example 1.2 for the computation of the differential probability over a single S-Box. The same computation can be made for the differential probability of Q_2: $\text{DP}(Q_2) = 2^{-4}$. The weight of Q is thus:

$$\text{w}_\text{r}(Q) = \text{w}_\text{r}(Q_1) + \text{w}_\text{r}(Q_2) = 4 + 4 = 8\,.$$

However, $\text{DP}(Q_1) \times \text{DP}(Q_2) = 2^{-8} \neq \text{DP}(Q) = 2^{-5}$ (as computed in Example 1.4) which means that, as already observed in Example 1.4, the round differentials Q_1 and Q_2 are not independent.

1.3.2 Linear cryptanalysis

Linear cryptanalysis [Mat93; TG91] is a known-plaintext attack. It exploits correlations between linear combinations of input bits and linear combinations of output bits of a permutation.

We first formally define the concept of correlation in the context of linear cryptanalysis:

Definition 1.3.9 (Correlation)**.** The (signed) *correlation* between the linear mask $\boldsymbol{u} \in \mathbb{F}_2^b$ at the input and the linear mask $\boldsymbol{v} \in \mathbb{F}_2^b$ at the output of a function $f\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ is defined as

$$\mathrm{C}_f(\boldsymbol{u}, \boldsymbol{v}) = \frac{1}{2^b} \sum_{\boldsymbol{x} \in \mathbb{F}_2^b} (-1)^{\boldsymbol{u}^t x + \boldsymbol{v}^t f(\boldsymbol{x})} .$$

We call the ordered pair of linear masks $(\boldsymbol{u}, \boldsymbol{v})$ a *linear approximation*. A positive (respectively, negative) correlation $\mathrm{C}_f(\boldsymbol{u}, \boldsymbol{v})$ indicates that a linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ is more often followed than not (respectively, more often not followed than it is) for the function f. If $\mathrm{C}_f(\boldsymbol{u}, \boldsymbol{v}) \neq 0$, then we say that $\boldsymbol{u}$ is compatible with $\boldsymbol{v}$ and that $(\boldsymbol{u}, \boldsymbol{v})$ is a valid linear approximation.

Example 1.6: Valid linear approximation

Let S_t be the S-box defined in Example 1.1. Let $\boldsymbol{u} = \begin{pmatrix}0\,0\,1\end{pmatrix}$ and $\boldsymbol{v} = \begin{pmatrix}1\,0\,1\end{pmatrix}$. We can then compute the correlation of the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ by going over each of the eight 3-bit states. By determining if $(\boldsymbol{x}, \mathrm{S}_t(\boldsymbol{x}))$ follows (repsectively does not follow) the linear approximation (u, v), we increment (respectively decrement) a counter. At the end, the value of this counter divided by 8, the total number of states, is exactly the correlation of $(\boldsymbol{u}, \boldsymbol{v})$ over S_t. This way, we find $\mathrm{C}_{\mathrm{S}_t}(\boldsymbol{u}, \boldsymbol{v}) = 2^{-1}$.

Since $\mathrm{C}_{\mathrm{S}_t}(\boldsymbol{u}, \boldsymbol{v}) \neq 0$, u is compatible with v and the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ is valid.

We note that in the literature (*e.g.* in the linear cryptanalysis attack by Matsui [Mat93]) the term linear approximation has several meanings. It should not be confused with what we call a linear trail, which is used to decompose a linear approximation into intermediate masks.

Definition 1.3.10 (Linear trail)**.** A sequence $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)}) \in \left(\mathbb{F}_2^b\right)^{k+1}$ that satisfies $\mathrm{C}_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}) \neq 0$ for $0 \leq i \leq k-1$ is called a *linear trail*.

Linear approximations and linear trails are the counterparts of differentials and differentials trails in differential cryptanalysis. As for the differential analysis, many linear trails may exists for the same linear approximation.

Definition 1.3.11 (Enveloping linear approximation, linear trail core and clustering of linear trails)**.** We write $\mathrm{LT}(\boldsymbol{u}, \boldsymbol{v})$ for the set of all linear trails in the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$, so with $\boldsymbol{q}^{(0)} = \boldsymbol{u}$ and $\boldsymbol{q}^{(k)} = \boldsymbol{v}$. We call $(\boldsymbol{u}, \boldsymbol{v})$ the *enveloping linear approximation* of the trails in $\mathrm{LT}(\boldsymbol{u}, \boldsymbol{v})$.

By deleting the initial linear mask $\boldsymbol{u}$ and final linear mask $\boldsymbol{v}$ of a linear trail $(\boldsymbol{u}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k-1)}, \boldsymbol{v})$ we are left with a *linear trail core* $(\boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k-1)})$.

If $\#\mathrm{LT}(\boldsymbol{u}, \boldsymbol{v}) > 1$, then we say that trails *cluster* together in the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$.

A linear trail core obtained by deleting the initial mask $\boldsymbol{u}$ and final mask $\boldsymbol{v}$ of a linear trail is said to be in the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$. Note that a linear trail core actually defines a set of linear trails with the same inner linear masks.

We now define what we call the correlation contribution of a linear trail Q in $\mathrm{LT}(\boldsymbol{u}, \boldsymbol{v})$:

Definition 1.3.12. The *correlation contribution* of a linear trail Q over f equals

$$\mathrm{C}_f(Q) = \prod_{i=1}^{k} \mathrm{C}_{\mathrm{R}_{i+1}}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}) .$$

From the theory of correlation matrices [Dae95], it follows that

$$\mathrm{C}_f(\boldsymbol{u}, \boldsymbol{v}) = \sum_{Q \in \mathrm{LT}(\boldsymbol{u}, \boldsymbol{v})} \mathrm{C}_f(Q) .$$

While it is hard in general to compute the correlation of a linear approximation of a function f, the correlation of any linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ of a round function $\mathrm{R} = \iota \circ \mathrm{L} \circ \mathrm{N}$ where N

consists in the parallel application of n S-boxes S_i is easy to compute. We start by specifying the intermediate linear masks as a linear trail $(\boldsymbol{u}, \boldsymbol{b}, \boldsymbol{c}, \boldsymbol{v})$. Thanks to the linearity of L, we have $\boldsymbol{b} = \mathrm{L}^t(\boldsymbol{c})$ and due to the fact that a linear mask is invariant under addition of a constant, all valid such linear trails are of the form $(\boldsymbol{u}, \mathrm{L}^t(\boldsymbol{v}), \boldsymbol{v}, \boldsymbol{v})$. Hence the linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ contains only a single trail and its correlation contribution is the correlation of the linear approximation $(\boldsymbol{u}, \mathrm{L}^t(\boldsymbol{v}))$ over the S-box layer, where the round constant addition affects only the sign:

$$\mathrm{C_R}(\boldsymbol{u}, \boldsymbol{v}) = (-1)^{\boldsymbol{v}^t \iota(0)} \prod_{1 \leq j \leq n} \mathrm{C_{S_j}}(P_j(\boldsymbol{u}), P_j(\mathrm{L}^t(\boldsymbol{v}))$$

where P_j is the projection over the j-th S-Box.

The squares of the correlations play an important role in linear cryptanalysis. Hence, it makes sense to give this quantity a name and we call it the linear potential.

Definition 1.3.13 (Linear potential)**.** The *linear potential* (LP) of a linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ is defined as:

$$\mathrm{LP}_f(\boldsymbol{u}, \boldsymbol{v}) = \mathrm{C}_f(\boldsymbol{u}, \boldsymbol{v})^2 \,.$$

Analogous to the differential cryptanalysis case, we define a weight metric.

Definition 1.3.14 (Correlation weight of a linear approximation)**.** The *correlation weight* of a linear approximation $(\boldsymbol{u}, \boldsymbol{v})$ with $\mathrm{LP}_f(\boldsymbol{u}, \boldsymbol{v}) \neq 0$ is given by:

$$\mathrm{w_c}(\boldsymbol{u}, \boldsymbol{v}) = -\log_2 \mathrm{LP}_f(\boldsymbol{u}, \boldsymbol{v}) \,.$$

For a linear approximation, the weight is directly tied to its linear potential. However, for a linear trail, the correlation weight is instead defined as the sum of the correlation weight of every round linear approximation:

Definition 1.3.15 (Correlation weight of a linear trail)**.** The correlation weight of a linear trail $Q = (\boldsymbol{q}^{(0)}, \boldsymbol{q}^{(1)}, \ldots, \boldsymbol{q}^{(k)})$ is defined as

$$\mathrm{w_c}(Q) = \sum_{i=0}^{k-1} \mathrm{w_c}(\boldsymbol{q}^{(i)}, \boldsymbol{q}^{(i+1)}) \,.$$

1.4 Box partitioning and alignment

In this section, we consider the partition of the index space defined by the nonlinear layer N of a given SPN-based permutation. We then define the *alignment* properties of the other step functions with respect to this partition.

The nonlinear layer N is described as the parallel application of n S-boxes of size m to disjoint parts of the state, indexed by B_i. Formally, this means that we can write N as $\mathrm{S}_0 \times \cdots \times \mathrm{S}_{n-1}$ and that it is characterized by

$$P_i \circ (\mathrm{S}_1 \times \cdots \times \mathrm{S}_n) = \mathrm{S}_i \circ P_i \text{ for } 1 \leq i \leq n \,.$$

Hence, N defines a unique *ordered* partition $\Pi_\mathrm{N} = (\mathrm{B}_1, \ldots, \mathrm{B}_n)$ of the index space $[\![1, b]\!]$. We call Π_N the *box partition* defined by N and the B_i N-boxes. If there is no ambiguity, we call the box partition Π and its members boxes.

Besides the box partition, it is clearly possible to define other partitions of the index space as well. We call a partition *non-trivial* if it has at least two members. Between any two partitions of the index space there may be a relation that we denote as *refinement*.

Definition 1.4.1 (Refinement of a partition)**.** We call Π a *refinement* of Π' and write $\Pi \leq \Pi'$ if for every $\mathrm{B}_i \in \Pi$ there exists a $\mathrm{B}'_j \in \Pi'$ such that $\mathrm{B}_i \subseteq \mathrm{B}'_j$.

Let Π be a partition of the index space consisting of k boxes, each of size l. We call a shuffle layer a Π-shuffle if the associated permutation matrix is an invertible block matrix with k blocks being the identity matrix $\boldsymbol{I}_l$ and all the others being $\mathbf{0}_l$, the all-zero matrix of dimension $(l \times l)$. If this is the case, then bit index permutations can be specified as a box index permutation.

Definition 1.4.2 (Alignment of a function to a given partition)**.** We say that $\phi\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ is *aligned* to Π if we can decompose it as

$$\phi_1 \times \cdots \times \phi_k \colon \bigtimes_{i=1}^{k} \mathbb{F}_2^l \to \bigtimes_{i=1}^{k} \mathbb{F}_2^l ,$$

with $P_i \circ \phi = \phi_i \circ P_i$ where the projections P_i are defined from the partition Π.

We are mostly interested in looking at the behaviour of the whole round function and the interaction of the linear components with the nonlinear one. Thus, we define what it is for a round function to be aligned in a definition that makes the partition implicitly defined by the nonlinear layer of the round function.

Definition 1.4.3 (Alignment of a round function)**.** Given a round function that is composed of the parallel application N of equally-sized S-boxes, a linear layer L, and the addition ι of a round constant, we say it is *aligned* if it is possible to decompose the linear layer L as $\mathrm{L} = \pi \circ \mathrm{M}$ in such a way that

— π is a Π_{N}-shuffle;

— M is aligned (in the sense of Definition 1.4.2) to a non-trivial partition Π_{M} that satisfies $\Pi_{\mathrm{N}} \leq \Pi_{\mathrm{M}}$.

We assume that the split between the linear and nonlinear layer is chosen so as to maximize the number of S-boxes in N.

Note that ι does not play a role in the alignment properties. If all of the round functions of a primitive are aligned, then we call the primitive aligned. If the primitive is *not* aligned, then we call it *unaligned*. We show in Example 1.7 that the toy permutation defined in Example 1.1 is aligned.

Example 1.7: Decomposition of an aligned round function

Let $\mathrm{R}_t = \pi_t \circ \mathrm{M}_t \circ \mathrm{N}_t$ be as in Example 1.1. Let $\Pi_{\mathrm{N}_t} = (\{1,2,3\},\{4,5,6\},\{7,8,9\},\{10,11,12\})$ and $\Pi_{\mathrm{M}_t} = (\{1,2,3,4,5,6\},\{7,8,9,10,11,12\})$ be the partition naturally obtained from N_t and M_t.

In Figure 1.5, the linear layer L_t is shown as the composition of a function M_t and a Π_{N_t}-shuffle π_t. The mixing layer M_t is aligned with the partition Π_{M_t} of which the partition Π_{N} is a refinement in the sense of Definition 1.4.1. Thus, R_t is aligned.

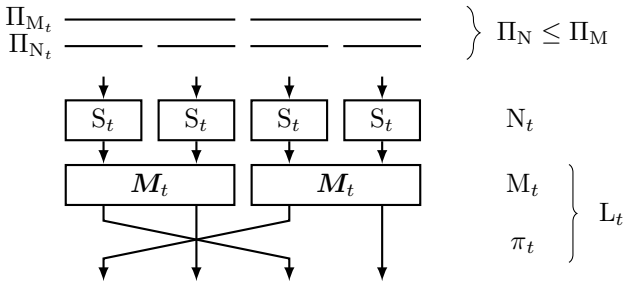


Figure 1.5: Alignment of R_t.

For some round functions, it is more natural to decompose the linear layer L as a composition of a shuffle π followed by a mixing layer M instead of a mixing layer M followed by a π as imposed by Definition 1.4.3. In fact, both description are equivalent.

Proposition 1.4.4. *If the linear layer* L *can be decomposed as* $\mathrm{M} \circ \pi$*, with* M *and* π *following the same properties as in Definition 1.4.3, then the round function is aligned.*

Proof of Proposition 1.4.4. Given a round function that is composed of the parallel application N of a layer of S-Boxes and a linear layer L, we suppose that we can decompose L as $\mathrm{M} \circ \pi$ with π a Π_{N}-shuffle and M aligned to Π_{M} such that $\Pi_{\mathrm{N}} \leq \Pi_{\mathrm{M}}$.

Let us define $\mathrm{M}' = \pi^{-1} \circ \mathrm{M} \circ \pi$. Then $\pi \circ \mathrm{M}' = \mathrm{M} \circ \pi = \mathrm{L}$. Since M is aligned to $\Pi_{\mathrm{M}} \leq \Pi_{\mathrm{N}}$ and π is a Π_{N}-shuffle, M' is itself aligned to a partition $\Pi_{\mathrm{M}'} \leq \Pi_{\mathrm{N}}$. The partition $\Pi_{\mathrm{M}'}$ can be seen as the shuffle of the partition Π_{M} by π^{-1}.

Thus, the round function is aligned and can be decomposed as $\pi \circ \mathrm{M}'$. □

The decomposition of L used (either $\mathrm{L} = \pi \circ \mathrm{M}$ or $\mathrm{L} = \mathrm{M} \circ \pi$) and the associated partition Π_{M} are chosen depending on the studied round function.

Superbox structure. Any aligned primitive has a *superbox* structure [PSC+02], that is helpful when investigating distributions and bounds on the differential probability of two-round differentials and the linear potential of two-round trails. We explain what this means. Consider a two-round structure: $\pi \circ \mathrm{M} \circ \mathrm{N} \circ \pi \circ \mathrm{M} \circ \mathrm{N}$. The final two linear steps π and M have no effect on the distributions studied, so we can simplify this expression to $\mathrm{N} \circ \pi \circ \mathrm{M} \circ \mathrm{N}$. Clearly, $\mathrm{N} \circ \pi = \pi \circ \mathrm{N}'$, with $\mathrm{N}' := \pi^{-1} \circ \mathrm{N} \circ \pi$. Hence, this is equivalent to $\pi \circ \mathrm{N}' \circ \mathrm{M} \circ \mathrm{N}$. Discarding the shuffle layer at the end gives $\mathrm{N}' \circ \mathrm{M} \circ \mathrm{N}$. Since $\Pi_{\mathrm{N}'} = \Pi_{\mathrm{N}} \leq \Pi_{\mathrm{M}}$, we can view this as the parallel application of a number of superboxes that are defined as the functions applied in parallel on each component of Π_{M} over two rounds. We call this a *superbox layer*.

In a sequence of two rounds, $\mathrm{N}' \circ \mathrm{M} \circ \mathrm{N}$ is a (composite) nonlinear layer and $\pi \circ \mathrm{M} \circ \pi$ is a (composite) linear layer. If the latter is aligned to a non-trivial partition Π such that $\Pi_{\mathrm{N}' \circ \mathrm{M} \circ \mathrm{N}} = \Pi_{\mathrm{M}} \leq \Pi$, then we can describe the sequence of two rounds as a single round that is itself aligned.

1.5 Permutations under study

In this section we describe the round functions of the permutations we investigate, their alignment properties, and compare their implementation cost.

1.5.1 Rijndael

RIJNDAEL [DR20] is a block cipher family supporting all block and key lengths of $b = 32k$ bits, with $4 \leq k \leq 8$, *i.e.* ranging from 128 up to and including 256 bits. The case $b = 128$ is of great importance as RIJNDAEL with that block length is the ubiquitous AES [oST01]. In this paper we investigate RIJNDAEL-256, the instance with $b = 256$, a width closer to those of the other permutations we investigate. In the remainder of this part we write RIJNDAEL for RIJNDAEL-256.

The RIJNDAEL round function consists of four steps: a nonlinear layer `SubBytes`, a box shuffle `ShiftRows`, a mixing layer `MixColumns`, and round key addition `AddRoundKey`. As the name of the nonlinear layer `SubBytes` suggests, $\Pi_{\texttt{SubBytes}}$ partitions the state in bytes. The mixing layer `MixColumns` is aligned to a non-trivial partition $\Pi_{\texttt{MixColumns}}$ such that $\Pi_{\texttt{SubBytes}} \leq \Pi_{\texttt{MixColumns}}$. By applying Proposition 1.4.4 we can conclude that RIJNDAEL is aligned.

Figure 1.6 shows a round of RIJNDAEL-128 (without the key addition) that is is easier to draw due to its dimensions, but the alignment properties for RIJNDAEL-256 are the same.

1.5.2 Saturnin

The SATURNIN [CDL+20] block cipher has a 256-bit key and block length. The state has several representations: three-dimensional, two-dimensional, and flat. In three dimensions, the 256-bit state is represented as a $4 \times 4 \times 4$ cube of 4-bit *nibbles*. Nibbles in the cube are indexed by triples (x, y, z). A *slice* is a subset of the nibbles with z constant. A *sheet* is a subset of the nibbles with x constant. A *column* is a subset of the nibbles with x and z constant. However, we focus on the

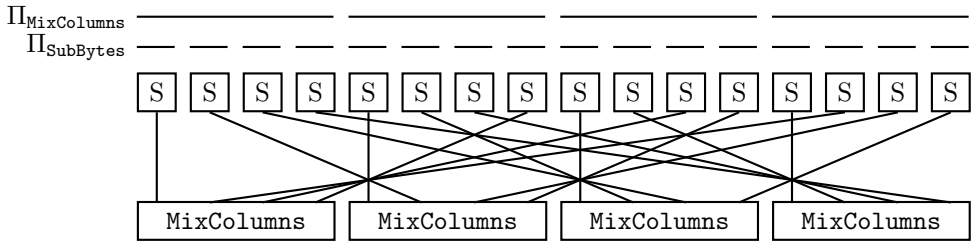


Figure 1.6: Alignment properties of Rijndael.

flat representation of Saturnin to draw the figure with the step functions and the partitions they define. A nibble with index (x, y, z) in the three-dimensional representation corresponds to a nibble with index $y + 4x + 16z$ in the flat representation.

The Saturnin permutation is composed of a number of so-called super-rounds, where a super-round consists of two consecutive rounds with indices $2r$ and $2r + 1$. Round $2r$ is composed as $\mathrm{MC} \circ \mathrm{S}$, where MC is a mixing layer and S is a nonlinear layer. There are two different rounds with odd indices. Round $4r + 1$ is composed as follows: $\mathrm{RC} \circ \mathrm{RK} \circ \mathrm{SR}_{\mathrm{slice}}^{-1} \circ \mathrm{MC} \circ \mathrm{SR}_{\mathrm{slice}} \circ \mathrm{S}$. Round $4r + 3$ consists of $\mathrm{RC} \circ \mathrm{RK} \circ \mathrm{SR}_{\mathrm{sheet}}^{-1} \circ \mathrm{MC} \circ \mathrm{SR}_{\mathrm{sheet}} \circ \mathrm{S}$. Here, RC denotes addition of a round constant, RK denotes addition of a round key, and $\mathrm{SR}_{\mathrm{slice}}$ and $\mathrm{SR}_{\mathrm{sheet}}$ shuffle nibbles. The partition Π_{S} divides the state into 64 nibbles. The shuffles $\mathrm{SR}_{\mathrm{slice}}$ and $\mathrm{SR}_{\mathrm{sheet}}$ are Π_{S}-shuffles. The mixing layer MC is aligned to a non-trivial partition Π_{MC} that divides the state into 16 columns, each consisting of 4 nibbles, and that satisfies $\Pi_{\mathrm{S}} \leq \Pi_{\mathrm{MC}}$. It follows that Saturnin is aligned. In a super-round we identify the sequence $\mathrm{S} \circ \mathrm{MC} \circ \mathrm{S}$ as a superbox layer with partition Π_{MC} and the linear layer of such a round is $\mathrm{SR}_{\mathrm{slice}}^{-1} \circ \mathrm{MC} \circ \mathrm{SR}_{\mathrm{slice}}$. This is a mixing layer that is aligned to a non-trivial partition Π_{slice} that divides the state into 4 slices, each containing 4 columns, and we have $\Pi_{\mathrm{MC}} \leq \Pi_{\mathrm{slice}}$. Similarly, for the other type of super-round, the mixing layer is aligned to a non-trivial partition Π_{sheet} that divides the state into 4 sheets, and we have $\Pi_{\mathrm{MC}} \leq \Pi_{\mathrm{sheet}}$. It follows that the super-rounds of Saturnin are aligned and hence have their own superboxes. These have width 64 bits and we call them *hyperboxes*. Figure 1.7 shows the alignment properties of the steps.

1.5.3 Spongent

Spongent [BKL+11] is a sponge-based hash function family that uses a Present-like permutation. The permutation is defined for any b that is a multiple of 4. In this paper, we only consider the case $b = 384$, to match the state size of the largest of the other permutations that we investigate, Xoodoo. The round function of Spongent consists of three steps: a round constant addition `lCounter`, a 4-bit S-box layer `sBoxLayer`, and a bit shuffle `pLayer`.

The index permutation of the bit shuffle `pLayer` is:

$$\texttt{pLayer}(j) = \begin{cases} 96j \bmod 383, & \text{if } j \in [\![0, 382]\!] \\ 383, & \text{if } j = 383 \end{cases}$$

As indicated by the Spongent designers in [BKL+11], we can decompose it into a mixing layer, followed by a box shuffle:

1. `SpongentMixLayer` applies the same mixing function `SpongentMix` in parallel to the 24 *subgroups* (following the terminology of [BKL+11]). It is a bit shuffle associated with the index permutation $\tau_{\mathrm{subgroup}} \colon [\![0, 15]\!] \to [\![0, 15]\!]$:

$$\tau_{\mathrm{subgroup}}(j) = \begin{cases} 4j \bmod 15, & \text{if } j \in [\![0, 14]\!] \\ 15, & \text{if } j = 15 \end{cases}$$

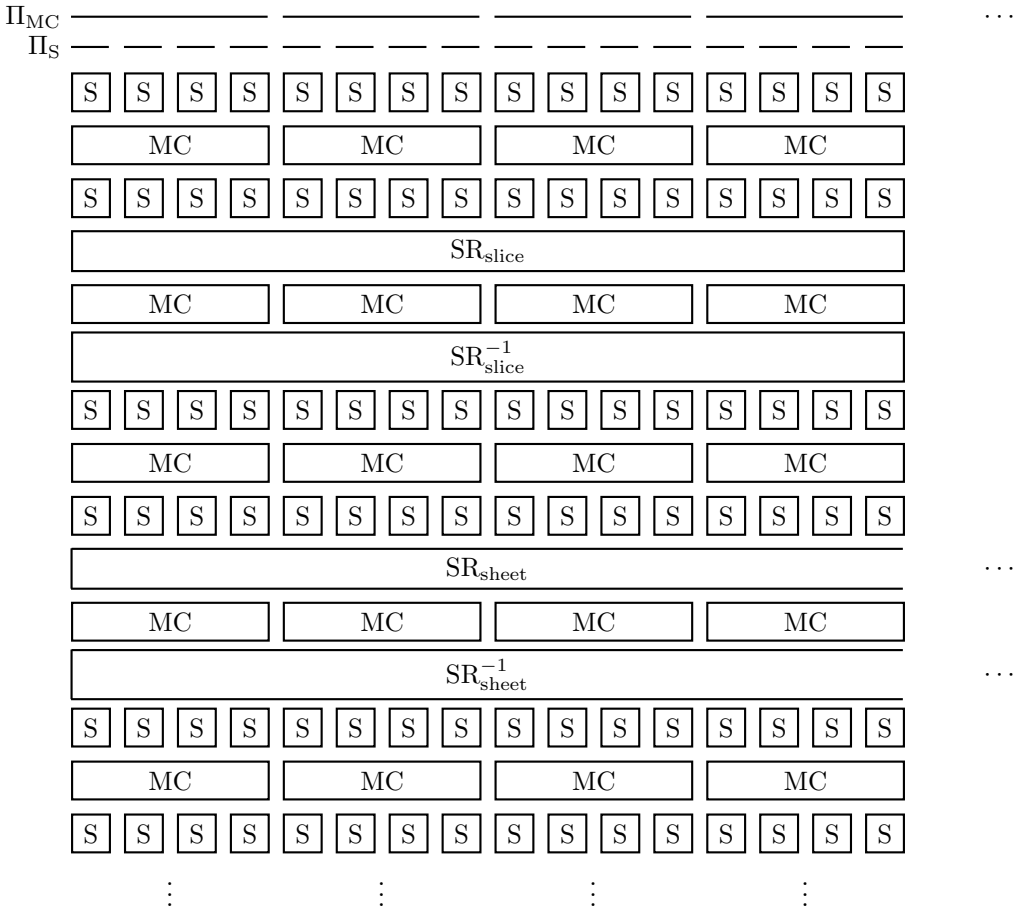


Figure 1.7: Alignment properties of SATURNIN.

2. `SpongentBoxShuffle` is a box shuffle that is associated with the box index permutation $\tau_{\text{box}} \colon [\![0, 95]\!] \to [\![0, 95]\!]$ defined by:

$$\tau_{\text{box}}(j) = \left\lfloor \frac{j}{4} \right\rfloor + 24(j \bmod 4)\,.$$

The `sBoxLayer` defines a box partition $\Pi_{\texttt{sBoxLayer}}$ corresponding to the 96 4-bit boxes. The box shuffle `SpongentBoxShuffle` is a $\Pi_{\texttt{sBoxLayer}}$-shuffle. The bit shuffle `SpongentMixLayer` is aligned to a non-trivial partition $\Pi_{\texttt{SpongentMixLayer}}$ that divides the state into 96 16-bit subgroups, each grouping four consecutive boxes, and we have $\Pi_{\texttt{sBoxLayer}} \leq \Pi_{\texttt{SpongentMixLayer}}$. It follows that SPONGENT is aligned. Figure 1.8 shows these steps and their alignment properties.

1.5.4 Xoodoo

XOODOO [DHV+18a] is a permutation with $b = 384$. The state consists of 3 equally sized horizontal *planes*, each one consisting of 4 parallel 32-bit *lanes*. Alternatively, the state can be seen as a set of 128 *columns* of 3 bits, arranged in a 4×32 array.

The round function of XOODOO consists of the following five steps: a mixing layer θ, a bit shuffle ρ_{east}, round constant addition ι, a nonlinear layer χ, and a bit shuffle ρ_{west}. The χ step applies the same 3-bit S-box to the columns of the state. The nonlinear layer χ defines a box partition Π_χ that corresponds to the 128 columns. The bit shuffles ρ_{east} and ρ_{west} perform translations of planes and neither ρ_{east} nor ρ_{west} is a Π_χ-shuffle. Additionally, Π_χ is not a

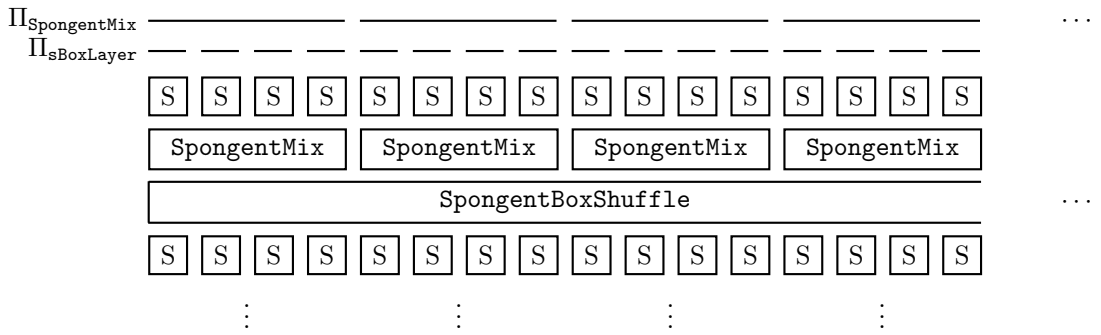


Figure 1.8: Alignment properties of Spongent.

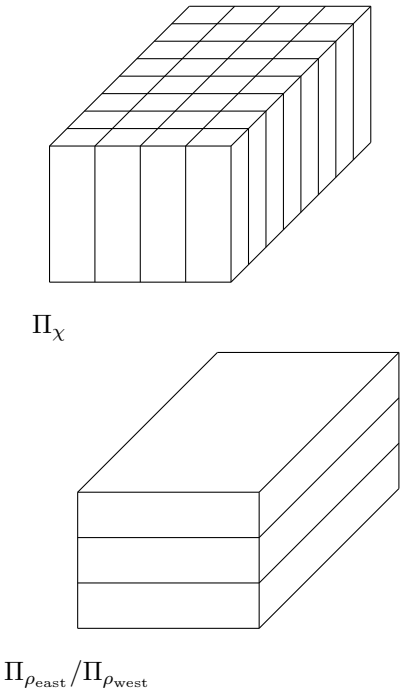


Figure 1.9: Alignment properties of Xoodoo.

refinement of neither $\Pi_{\rho_{\text{east}}}$ nor $\Pi_{\rho_{\text{west}}}$ The mixing layer θ defines no non-trivial box partition at all. Due to the properties of the ρ steps and θ it is impossible to split the linear layer in a column shuffle and a mixing layer that is aligned to a partition that Π_χ is a refinement of. In other words, Xoodoo is unaligned. Figure 1.9 shows the alignment properties of the steps.

We provide a computer-assisted proof that Xoodoo is unaligned.

Xoodoo is unaligned. Let us assume that we can factor the linear layer of Xoodoo into $\mathrm{L} = \pi \circ \mathrm{M}$ with M operating on non-trivial superboxes. We can identify the input bits of M that lie in the same superbox with the two following rules:

1. The output bits of L in the same box (column) depend on input bits from the same superbox;
2. Any two output bits that depend on the same input bit must also depend on input bits from the same superbox.

Therefore, we construct a bipartite graph with the 128 output boxes on one side and the 384

```
def buildGraph():
    G = Graph()
    for x in range(4):
        for z in range(32):
            G.add_vertex("out-{0}-{1}".format(x, z))
            for y in range(3):
                G.add_vertex("in-{0}-{1}-{2}".format(x, y, z))
    for x in range(4):
        for z in range(32):
            out = "out-{0}-{1}".format(x, z)
            G.add_edge(out, "in-{0}-{1}-{2}".format(x, 0, z))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 0, (z+27)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 0, (z+18)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 1, (z+26)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 1, (z+17)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+1)%4, 2, (z+19)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+1)%4, 2, (z+10)%32))

            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 1, (z+31)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+2)%4, 0, (z+27)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+2)%4, 0, (z+18)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+2)%4, 1, (z+26)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+2)%4, 1, (z+17)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+0)%4, 2, (z+19)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+0)%4, 2, (z+10)%32))

            G.add_edge(out, "in-{0}-{1}-{2}".format((x+2)%4, 2, (z+13)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 0, (z+16)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 0, (z+ 7)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 1, (z+15)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+3)%4, 1, (z+ 6)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+1)%4, 2, (z+ 8)%32))
            G.add_edge(out, "in-{0}-{1}-{2}".format((x+1)%4, 2, (z+31)%32))
    return G

G = buildGraph()
G.is_connected()
```

Figure 1.10: Sage code to construct the graph and to check its connectivity used to prove that Xoodoo is unaligned.

input bits on the other side, with edges connecting an output box to the input bits that it depends on. We explicitly construct this graph (see Figure 1.10) and check that it is connected. This contradicts the assumption that M operates on non-trivial superboxes.

Chapter 2

Weight histograms and huddling

Contents

Overview

The alignment properties of our four permutations, Rijndael, Saturnin, Spongent and Xoodoo, have been studied in Chapter 1 showing that both Rijndael, Saturnin and Spongent are aligned in the sense of our definition whereas Xoodoo is unaligned.

In this chapter, we explore how the alignment affects the effectiveness of their linear layer and also its impact on their differential and linear properties. First, we recall in Section 2.1 the definitions of the bit and box weight metrics. We use these metrics to build histograms depicting quantitatively the mixing power of the linear layer. We also compare how the histograms varies from bit to box weight because of an effect that we call *huddling*, which seems to have more impact on aligned designs. Then, we investigate in Section 2.2 how it translates to a first measure of their differential and linear properties by studying the weight of two-round trails. To do so, we define a third type of histograms, the *trail weight histogram*, that reports on the repartition of the trails by their weight and we construct this histogram for our four permutations.

2.1 Bit and box weight histograms and the huddling effect

2.1.1 Definitions of bit weight, box weight and their histograms

The weight of a two-round trail $(\boldsymbol{q}_{\text{in}}, \boldsymbol{a}, \boldsymbol{b}, \boldsymbol{q}_{\text{out}})$ over $\mathrm{N} \circ \mathrm{L} \circ \mathrm{N}$ is equal to the sum of the weight of $(\boldsymbol{q}_{\text{in}}, \boldsymbol{a})$ and the weight of $(\boldsymbol{b}, \boldsymbol{q}_{\text{out}})$. Since each of these differentials or linear approximations is over a layer of S-boxes, their weights directly depend on the product of the differential probabilities or linear potentials of the projections of $\boldsymbol{a}$ and $\boldsymbol{b}$ over each S-box. Knowing how many of these projections are non-zero can actually suffice to compute a lower bound of the total weight.

Additionally, the value of $\boldsymbol{a}$ fully determines the value of $\boldsymbol{b}$ as $\boldsymbol{b} = \mathrm{L}(\boldsymbol{a})$ in differential trails and $\boldsymbol{a} = \mathrm{L}^t(\boldsymbol{b})$ in linear trails. Thus, the distribution of the differences or linear masks depending on the number of non-zero projections before and after a linear layer L determines its *mixing power*. In this section, we define a metric for the number of these non-zero projections and formalize the corresponding distribution as histograms to be able to quantitatively describe the mixing power of L.

Bit weight and box weight. First, we formally define what it means for a bit to be active.

Definition 2.1.1 (Bit activity)**.** We call a bit i *active* in the state, difference or linear mask $\boldsymbol{a} \in \mathbb{F}_2^b$ if $\boldsymbol{a}_i = 1$ and *passive* otherwise. We call this property the *activity* of a bit.

The natural metric associated with bit activity is the bit weight, also called the Hamming weight:

Definition 2.1.2 (Bit weight)**.** Given a state, a difference or a linear mask $\boldsymbol{a}$, its *bit weight* $\mathrm{w}_2(\boldsymbol{a})$ is defined as:

$$\mathrm{w}_2(\boldsymbol{a}) = \#\{i \in [\![1, b]\!] \mid \boldsymbol{a}_i \neq 0\}\,.$$

These definitions can be naturally extended to describe a state, a difference or a linear mask $\boldsymbol{a}$ at the granularity of the S-boxes. To do so, we first define an indicator function with respect to a box partition.

Definition 2.1.3 (Indicator function with respect to a partition)**.** We call an *indicator function* with respect to a partition Π a function $1_i \colon \mathbb{F}_2^b \to \mathbb{F}_2$ such that

$$1_i(\boldsymbol{a}) = \begin{cases} 0, & \text{if } P_i(\boldsymbol{a}) = 0 \\ 1, & \text{otherwise.} \end{cases}$$

where projections P_i are taken along the partition Π.

We use this definition to define what is an active box.

Definition 2.1.4 (Box activity)**.** Let $\Pi = (\mathrm{B}_1, \ldots, \mathrm{B}_n)$ the box-partition naturally defined from the description of a nonlinear layer N that consists in the parallel application of n S-Boxes.

We call the box B_i *active* in the difference or linear mask $\boldsymbol{a} \in \mathbb{F}_2^b$ if $1_i(\boldsymbol{a}) = 1$ and *passive* otherwise. We call this property the *activity* of a box.

The natural metric associated with box activity is the box weight:

Definition 2.1.5 (Box weight)**.** Given a state, a difference or a linear mask $\boldsymbol{a}$, we define its *box weight* $\mathrm{w}_\Pi(\boldsymbol{a})$ with respect to a partition Π as:

$$\mathrm{w}_\Pi(\boldsymbol{a}) = \#\{i \in [\![1, n]\!] \mid 1_i(\boldsymbol{a}) \neq 0\}\,.$$

We can naturally encode the box activity of a difference or a linear mask as a vector:

Definition 2.1.6 (Activity pattern)**.** Given a state, a difference or a linear mask $\boldsymbol{a}$, its *activity pattern* $r_\Pi(\boldsymbol{a})$ with respect to a partition Π is defined as

$$r_\Pi(\boldsymbol{a}) = \sum_{i=1}^{n} 1_{\mathrm{B}_i}(\boldsymbol{a}) \boldsymbol{e}_i^n\,.$$

It is the vector whose i-th component is one if box B_i is active and zero otherwise.

We describe how the activity pattern is impacted by the application of the nonlinear layer in the following lemma.

Lemma 2.1.7 (Activity pattern preservation through N)**.** *Let $(\boldsymbol{\Delta}_{in}, \boldsymbol{\Delta}_{out})$ be a valid differential and $(\boldsymbol{u}, \boldsymbol{v})$ a valid linear approximation over a nonlinear layer* N *composed of n S-boxes* S *applied in parallel along the box partition* Π*. For the differential case, we need the additional hypothesis that* S *is injective.*

Then we have that:

$$r_\Pi(\boldsymbol{\Delta}_{in}) = r_\Pi(\boldsymbol{\Delta}_{out})$$
$$\textit{and}$$
$$r_\Pi(\boldsymbol{u}) = r_\Pi(\boldsymbol{v})$$

Proof. As seen in Section 1.3, the differential probability (respectively correlation) over N is the product of the differential probabilities (respectively correlations) of the differential (respectively linear approximation) over each S-Box. Additionally, over a single S-box S, a differential or a linear approximation of the form $(\boldsymbol{a}, 0)$ (or $(0, \boldsymbol{a})$) with $\boldsymbol{a} \neq 0$ have a differential probability of 0 when S is injective and a correlation of 0.

Thus in the valid differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$, for all $i \in [\![1, n]\!]$, $P_i(\boldsymbol{\Delta}_{\mathrm{in}}) = 0$ (respectively $P_i(\boldsymbol{\Delta}_{\mathrm{in}}) \neq 0$) implies $P_i(\boldsymbol{\Delta}_{\mathrm{out}}) = 0$ (respectively $P_i(\boldsymbol{\Delta}_{\mathrm{out}} \neq 0)$ where P_i is the projection over the i-th S-box. The same is true for the linear approximation. By definition of the activity pattern, this proves the lemma. □

Not surprisingly, the number of active bits is not a good estimator of the number of active boxes, as shown in Example 2.1.

Example 2.1: Bit weight and box weight

Let $\Pi = (\{1,2,3\}, \{4,5,6\}, \{7,8,9\}, \{10,11,12\})$ be the partition naturally defined from the layer of S-boxes N_t of the permutation defined in Example 1.1.

It follows that $r_\Pi(\left(1\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\right) = \left(1\,0\,0\,0\right)$ and $r_\Pi\left(1\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\right) = \left(1\,1\,1\,0\right)$. Thus we have, $\mathrm{w}_\Pi(\left(1\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\right)) = 1$ and $\mathrm{w}_2(\left(1\,1\,1\,0\,0\,0\,0\,0\,0\,0\,0\,0\right)) = 3$ whereas $\mathrm{w}_\Pi(\left(1\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\right)) = 3$ while still having $\mathrm{w}_2(\left(1\,0\,0\,1\,0\,0\,0\,0\,1\,0\,0\,0\right)) = 3$.

This example shows that a large number of active bits does not necessarily lead to a large

number of active boxes. This phenomenon is discussed more thoroughly in Subsection 2.1.3.

We now define two types of histograms to show the behaviour of the mixing layer at the bit level, at the box level and we comment on the link between these two behaviours.

Bit and box weight histograms. In order to quantify the "mixing power" of a linear transformation L, we consider the weight distribution of $(\boldsymbol{a} \,\|\, \mathrm{L}(\boldsymbol{a}))$ over all differences or linear masks $\boldsymbol{a} \in \mathbb{F}_2^b$ and embed it in a histogram. This is a well-known concept in coding theory, where weight distributions are embedded in so-called *weight enumerator polynomials* that classify the code [HP03]. For a given code C of length n and minimum distance d, its weight enumerator polynomial is the following bivariate formal polynomial $W_C = \sum_{k=0}^{n} A_k x^k y^{n-k}$ with A_k being the number of codewords in C of weight k.

Definition 2.1.8 (Weight histogram)**.** The *weight histogram* of a linear transformation $\mathrm{L}\colon \mathbb{F}_2^b \to \mathbb{F}_2^b$ is a function $\mathcal{N}_{\cdot,\mathrm{L}}\colon \mathbb{N} \to \mathbb{N}$ given by

$$\mathcal{N}_{\cdot,\mathrm{L}}(k) = \#\{\boldsymbol{a} \in \mathbb{F}_2^b \mid \mathrm{w}_\cdot(\boldsymbol{a}) + \mathrm{w}_\cdot(\mathrm{L}(\boldsymbol{a})) = k\}\,,$$

where $\cdot$ must be replaced by either 2 (bit weight histogram) or Π (box weight histogram).

Informally, for every integer k the weight histogram embeds the information of the number of different states, differences or linear masks such that the sum of their weight before and after applying the linear transformation L is equal to k.

The cumulative version on the same domain and codomain is given by

$$\mathcal{C}_{\cdot,\mathrm{L}}(k) = \sum_{l \le k} \mathcal{N}_{\mathrm{L}}(l)\,.$$

We call the *tail* of the histogram, the values that correspond to low weight. In the charts we present, the tail consists in their left-most parts.

Remarks on the branch number. We note that the *differential branch number* [Dae95] of the linear layer is simply the smallest non-zero entry of the bit or box weight histogram, *i.e.* $\min\{k > 0 \mid \mathcal{N}_{\cdot,\mathrm{L}}(k) > 0\}$. The linear branch number is the smallest non-zero entry in the corresponding histogram of L^t and can be different from its differential counterpart. This is not the case for the mappings in this part and we omit the qualifier in the remainder. A higher branch number typically implies higher mixing power. However, the box weight histogram is more informative than just the branch number as it gives the exact number of differences or linear masks meeting the branch number and it also gives a description of the repartition of differences and linear masks for higher weight values. In general, the weight histogram allows a more nuanced comparison of mixing layers than the branch number.

2.1.2 Computing the weight histograms for an aligned round function

Given an aligned round function R, it is sufficient to compute the weight histograms of only the mixing layer M to compute the one of the linear layer:

Lemma 2.1.9 (Box weight histogram of the linear layer of an aligned round function)**.** *Given an aligned round function* $\mathrm{R} = \pi \circ \mathrm{M} \circ \mathrm{N}$ *(or* $\mathrm{R} = \mathrm{M} \circ \pi \circ \mathrm{N}$*), the box weight histogram of* M *and* $\mathrm{L} = \pi \circ \mathrm{M}$ *(or* $\mathrm{L} = \mathrm{M} \circ \pi$*) are equal.*

Proof. By definition of an aligned round function, π is a Π_{N}-shuffle. Since π only permutes the boxes after (or before) the effect of M it does not impact on the box weight of the output (or input) of M.

We have that $\mathrm{w}_\Pi(\boldsymbol{a}) = \mathrm{w}_\Pi(\pi(\mathrm{M}(\boldsymbol{a})))$, thus the lemma is true for $\mathrm{L} = \pi \circ \mathrm{M}$.

If $\mathrm{L} = \mathrm{M} \circ \pi$, let $\mathcal{S}_{\mathrm{M}} = \{\boldsymbol{a} \in \mathbb{F}_2^b \mid \mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{M}(\boldsymbol{a})) = k\}$ and $\mathcal{S}_{\mathrm{L}} = \{\boldsymbol{a} \in \mathbb{F}_2^b \mid \mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{M}(\pi(\boldsymbol{a}))) = k\}$ for a given $k \in \mathbb{N}$. Since $\mathrm{w}_\Pi(\pi(\boldsymbol{a})) = \mathrm{w}_\Pi(\boldsymbol{a})$, π is a bijection from $\mathcal{S}_{\mathrm{L}}$ to $\mathcal{S}_{\mathrm{M}}$. Thus, both sets have the same cardinality for every $k \in \mathbb{N}$ and therefore the box weight histogram of M and L are equal. □

Hence, the box weight histogram of a box-aligned function equals that of the identity. The identity is special in that it has the least mixing power out of all permutations. In other words, a box-aligned function does not contribute to the mixing power. The same reasoning can be applied to the bit weight histogram of shuffles that only permute bits.

Additionally, the superbox structure of an aligned primitive makes it possible to use a divide-and-conquer approach to compute the box weight histograms. By definition of alignment the mixing layer M acts independently on each of the s superboxes defined by the partition Π_{M}. Thus, we can define M_i as the component of M that is applied to the i-th superbox. Additionally, we define for all $k \in \mathbb{N}$ $\mathcal{S}(k) = \{\boldsymbol{v} \in \mathbb{N}^s \mid \sum_{i=1}^{s} \boldsymbol{v}_i = k\}$. Finally, we can compute the weight histograms of M from the weight histograms of its superbox functions M_i:

$$\mathcal{N}_{\cdot,\mathrm{M}}(k) = \sum_{\boldsymbol{v}\in\mathcal{S}(k)} \prod_{i=1}^{s} \mathcal{N}_{\cdot,\mathrm{M}_i}(\boldsymbol{v}_i)\,. \tag{2.1}$$

The Maximum Distance Separable (MDS) case. First, we define what are Maximum Distance Separable linear codes:

Definition 2.1.10 (Maximum Distance Separable code)**.** A linear code of length n and dimension k is said to be *Maximum Distance Separable (MDS)* if its minimum distance d verifies: $d = n - k + 1$, that is if it reaches the Singleton bound.

From this definition, we can also define what are Maximum Distance Separable linear mappings:

Definition 2.1.11 (Maximum Distance Separable (MDS) function)**.** A function $f\colon \mathbb{F}_{2^m}^p \to \mathbb{F}_{2^m}^p$ is called a *Maximum Distance separable (MDS)* function if the set $\{(\boldsymbol{x} \,\|\, f(\boldsymbol{x})) \mid \boldsymbol{x} \in \mathbb{F}_{2^m}^p\} \subseteq \mathbb{F}_{2^m}^{2p}$ is an MDS code over $\mathbb{F}_{2^m}$ of minimum distance d.

When each function M_i is a Maximum Distance Separable (MDS) function, the box weight histogram is given from coding theory results on the underlying MDS code. It can be expressed as a combinatorial expressions of m, the box size, and n, the number of boxes. Since the round function studied is aligned, the ordered partition Π_{N} is a refinement of the ordered partition Π_{M} naturally defined by M and the M_is.

More precisely, we have the following theorem:

Theorem 2.1.12 (Special case of [HP03, Theorem 7.4.1])**.** *Let C be an $[2p, p, p+1]$ MDS code over $\mathbb{F}_q$. The coefficient A_i, $0 \le i \le 2p$ of the weight enumerator polynomial of C is given by:*

$$A_i = \begin{cases} 0 & \text{if } i \le p \\ \binom{2p}{i} \sum_{j=0}^{i-p-1} (-1)^j \binom{i}{j} (q^{i-p-j} - 1) & \text{if } i \ge p+1 \end{cases}$$

Thus, in the case of MDS mixing layers, computing the box weight of the linear layer is straightforward by combining Lemma 2.1.9, Theorem 2.1.12, and Equation (2.1).

General case. In general, we do not have the same numeric formula as in the MDS case and Equation (2.1) only applies to aligned permutations. The details on how the weight histograms are computed for Xoodoo should appear in Daniël Kuijsters' thesis [Kui].

2.1.3 Application to the four permutations and huddling

We discuss the cumulative bit and box weight histograms for the linear layers of our four permutations, given in Figure 2.1. We include the histogram for the identity function, assuming 3, 4 and 8-bit S-boxes for the box weight histogram to allow for comparison with the permutations.

Bit weight histogram. The bit weight histogram for Spongent coincides with that of the identity permutation. This is because its linear layer is a bit shuffle. As the identity permutation maps inputs to identical outputs, it has only non-zero entries for even bit weights. Its bit branch number is 2. In conclusion, its mixing power is the lowest possible.

The bit branch number of the mixing layer of Rijndael, `MixColumns`, is 6, that of Saturnin-MC is 5, and that of Xoodoo-θ is 4.

Similar to Spongent, the bit weight histograms of Rijndael and Xoodoo have only non-zero entries at even bit weights. This is because both Xoodoo-θ and Rijndael-`MixColumns` can be modeled as $\boldsymbol{a} \mapsto (\boldsymbol{I}_b + \boldsymbol{M})\boldsymbol{a}$ for some matrix $\boldsymbol{M} \in \mathbb{F}_2^{b\times b}$ with the property that the bit weight of $\boldsymbol{Ma}$ is even for all $\boldsymbol{a} \in \mathbb{F}_2^b$ and $\boldsymbol{I}_b$ the identity matrix of $\mathbb{F}_2^{b\times b}$. Saturnin-MC cannot be modeled in that way and does have non-zero entries at odd bit weights. The bit weight histograms of Rijndael and Saturnin are very close and that of Xoodoo is somewhat higher.

Box weight histogram. Both `MixColumns` in Rijndael and MC in Saturnin are mixing layers that are MDS function as defined in Definition 2.1.11. Thus, their box weight histogram can be computed using Theorem 2.1.12 combined with Equation (2.1). For Spongent, the box weight histogram is computed from the one of `SpongentMix`.

For Spongent the box branch number is 2, the same as the bit branch number. However, the box weight histogram of Spongent has a lower tail than the identity permutation. This shows that `SpongentMixLayer` has a higher mixing power than in the bit case.

The box branch number of the linear layers of Rijndael, `MixColumns`, and of Saturnin-MC are both 5, while for Xoodoo it is 4. Even if the branch numbers of `MixColumns` and MC are equal, Saturnin's mixing layer has a better mixing power than Rijndael's, which is indicated by a lower tail.

Bit huddling. The discrepancy between the bit and box weight histogram brings us to the qualitative notion of *bit huddling*: many active bits huddle together in few active boxes.

Huddling has an effect on the contribution of states $\boldsymbol{a}$ to the histogram, *i.e.*, by definition we have that $\mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{L}(\boldsymbol{a})) \leq \mathrm{w}_2(\boldsymbol{a}) + \mathrm{w}_2(\mathrm{L}(\boldsymbol{a}))$. In words, from bit to box weight, huddling moves states to the left in the histogram, thereby raising the tail. Huddling therefore results in the decay of mixing power at box level as compared to bit level. In the absence of huddling, the bit and box weight histogram would be equal. However, huddling cannot be avoided altogether as states do exist with multiple active bits in a box since $m \geq 2$.

Example 2.2: Bit huddling

If the mixing layer is a simple bit shuffle, then there are $n\binom{m}{2}$ states with only two active bits and where the two active bits are in the same box. These states have a total bit weight (before and after the mixing layer) of 4 since a bit shuffle has the same bit weight histogram as the identity permutation. However they have a total box weight (before and after the mixing layer) of either 2 if the bits are still in the same box after the bit shuffle or 3 otherwise.

Superbox huddling effect. We see Rijndael has *high* bit huddling. In moving from bit weights to box weights, the branch number decreases from 6 to 5 and the tail rises from being the lowest of the four to the highest. This is a direct consequence of the large width of the Rijndael S-boxes, namely 8, and the byte alignment. Indeed, `MixColumns` only mixes bits within the 32-bit columns. We call this the *superbox huddling effect*. Of course, there is a reason for these large S-boxes: they have low maximum differential probability and linear potential.

Saturnin, with its Rijndael-like structure also exhibits the superbox huddling effect, though less pronounced than Rijndael. From bits to boxes the branch number does not decrease and the tail rises less than for Rijndael. Clearly, its smaller S-box size, namely 4, allows for less bit huddling.

Due to its alignment, Spongent exhibits the superbox huddling effect, but less so than Saturnin. The reason for this is the already high tail in the bit weight histogram, due to the

absence of bit-level diffusion in the mixing layer. When designing a bit shuffle, one has a choice to minimize the amount of huddling. Such a bit shuffle would move bits of a small number of boxes at the input to a large number of boxes at the output (and vice versa). Unfortunately, the designers of Spongent have not followed such a design approach. The superbox structure of Spongent implies that the bits belonging to groups of four boxes are moved to bits of another group of four boxes. The result is higher huddling and as a consequence lower mixing power than what would have been possible.

Finally, Xoodoo has the *lowest* bit huddling of the four primitives studied. The huddling is mostly limited to what is unavoidable, resulting in entries with odd box weight and a slight change in the entries of even box weight due to states shifting to the left from higher weight or to lower weights. This is the consequence of two design choices: having very small S-boxes (3-bit) and the absence of alignment, avoiding the superbox huddling effect altogether.

Example 2.3: Quantitative indicator for huddling

An interesting quantitative indicator for huddling is the weight below which there are some given number of states. If we do the exercise with this number equal to 2^{30}, we see from bit to box weight that in Rijndael this value decreases from 20 to 8, in Saturnin from 20 to 12, in Spongent from 10 to 8 and in Xoodoo from 18 to 16. While starting out with better mixing at bit level, due to its alignment properties Saturnin loses more than Xoodoo when switching to boxes and ends up with worse mixing power than Xoodoo.

We see that the bit weight histograms are in line with the computational cost of the mix layers. From bit to box weight histograms we see the superbox huddling effect affecting Rijndael and Saturnin the most. Thanks to the absence of alignment in Xoodoo, its linear layer has the best box-level mixing despite the fact that its mixing layer has the lowest cost of these three.

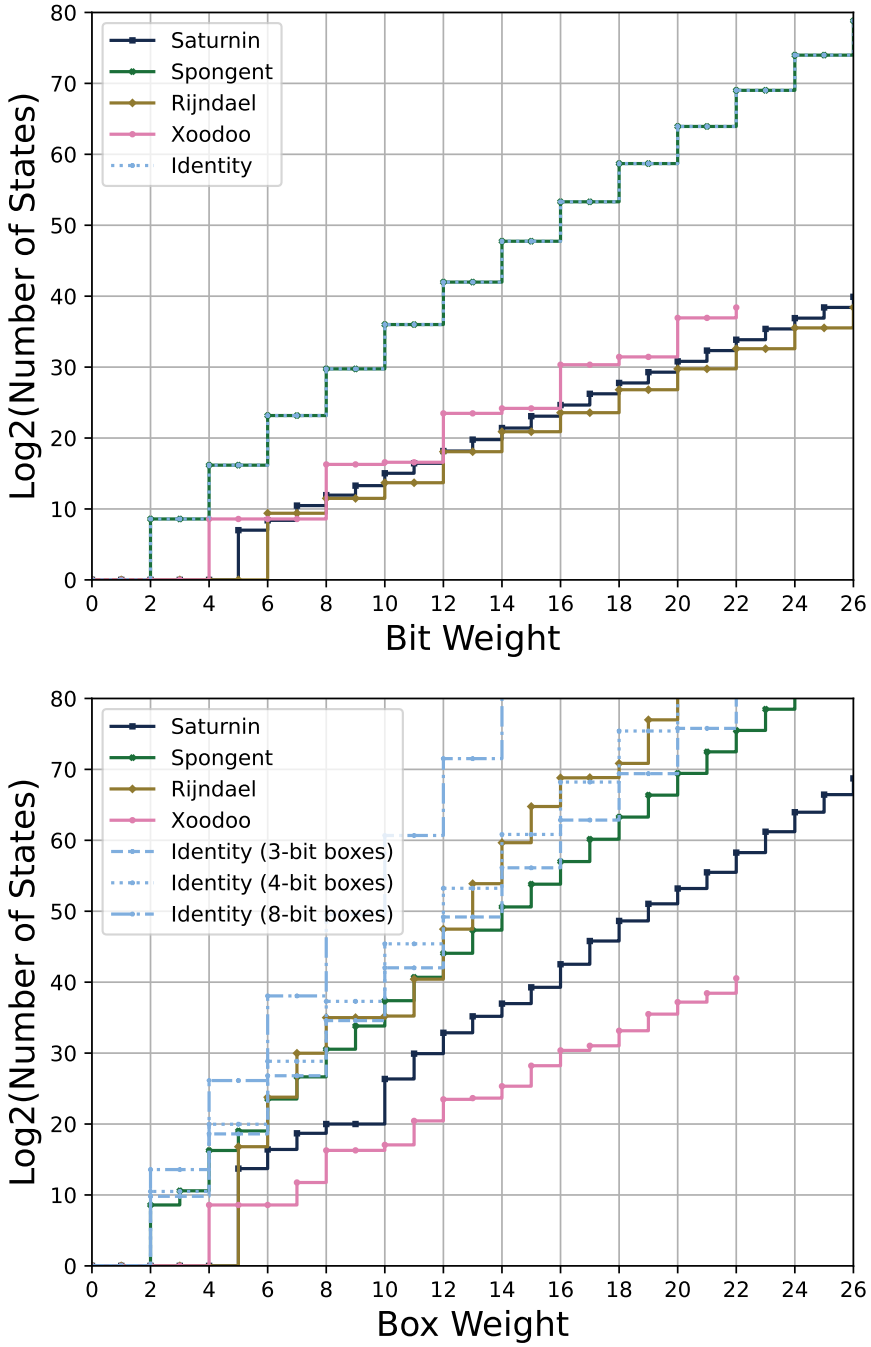


Figure 2.1: Cumulative bit weight and box weight histograms.

2.2 Trail weight histograms

2.2.1 Definition of trail weight histogram

In the previous section, we considered a mixing layer M with the box partition Π_S of the nonlinear layer taken into account to quantitatively describe its mixing power. In this section, we extend this view by taking the differential and linear properties of the S-box layer into account as well.

We define the trail weight histogram analogous to Definition 2.1.8:

Definition 2.2.1 (Trail weight histogram)**.** The *trail weight histogram* of a k-round transformation of the form $\mathrm{N} \circ \mathrm{L} \circ \cdots \circ \mathrm{L} \circ \mathrm{N}$ is a function $\mathcal{TH}_\cdot : \mathbb{N} \to \mathbb{N}$ given by

$$\mathcal{TH}_\cdot(k) = \#\left\{\text{trails } Q \mid \mathrm{w}_\cdot(Q) = k\right\} ,$$

where $\cdot$ must be replaced by either r (restriction weight of differential trails) or c (correlation weight of linear trails).

The cumulative version on the same domain and codomain is given by

$$\mathcal{CTH}_\cdot(k) = \sum_{k' \leq k} \mathcal{TH}_\cdot(k') .$$

As before, systematically lower values in the tail of one histogram compared to the other means the permutationassociated with the former performs better with respect to the metric.

2.2.2 Two-round trail weight histograms

Figure 2.2 reports on the distribution of the weight of two-round differential and linear trails of our four permutations.

We use a similar method as for weight histograms to compute the trail weight histograms of the aligned permutations, that is we exploit the superbox structures and compute the whole histogram from the one of a superbox using Equation (2.1).

While Rijndael performed the worst with respect to the box weight metric, we see that it performs the best with respect to the trail weights. The reasons are the low maximum differential probability and linear potential of its S-box as well as its high branch number. However, this have to be put in perspective with the implementation cost of these S-boxes, which is naturally higher than for smaller ones. The relative ranking of the other permutations does not change in moving from box weight to trail weights. Still, Xoodoo loses some terrain due to its more lightweight S-box layer.

Due to its bad mixing properties, Spongent does not perform well with respect to the trail weights either. Despite the difference in design approach, Xoodoo and Saturnin have quite similar two-round trail weight histograms.

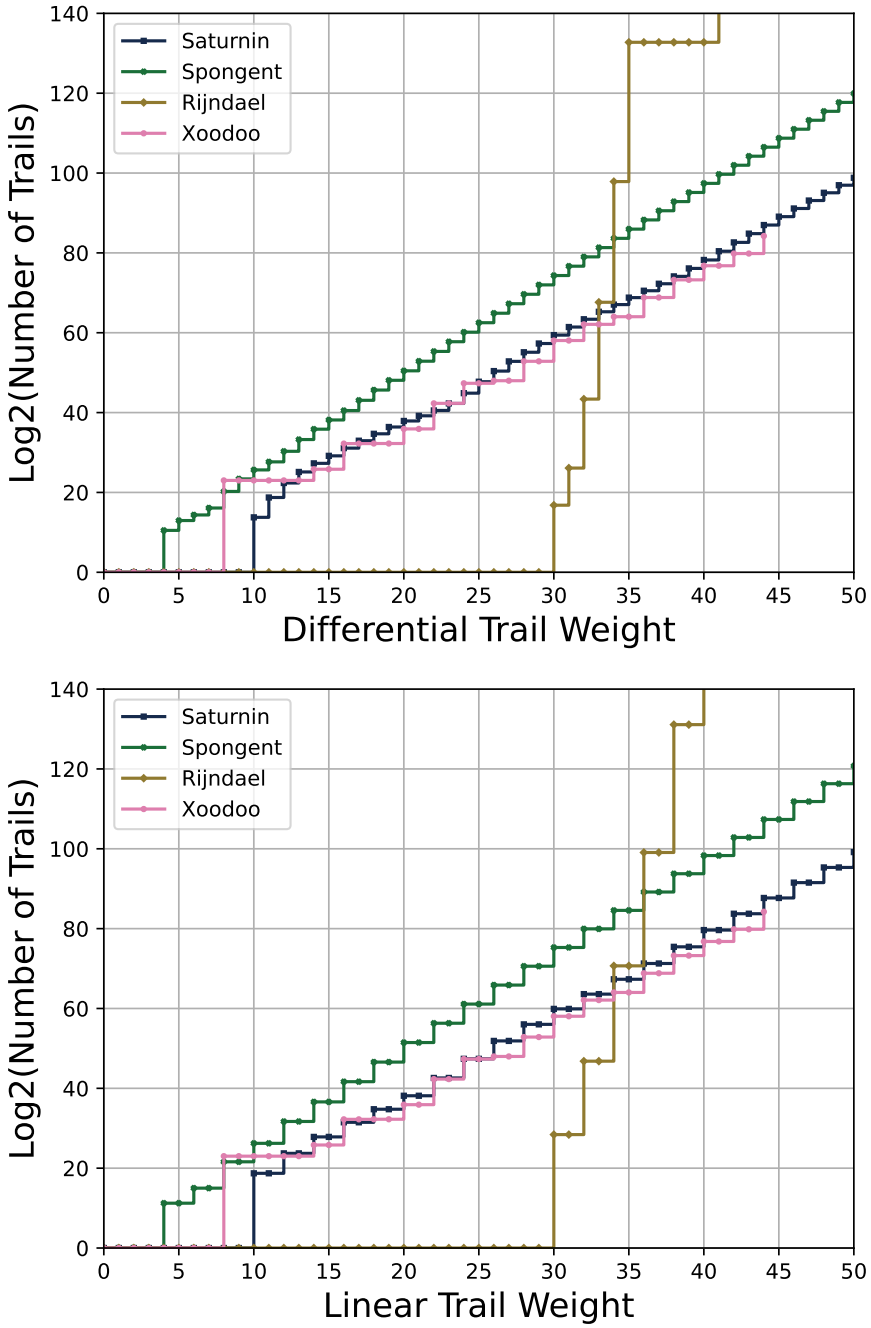


Figure 2.2: Two rounds: cumulative differential and linear trail weight histograms.

Chapter 3

Clustering and round differentials independence

Contents

Overview

The investigation of two-round trails and their weight conducted in Chapter 2 gives a first approximation of the differential and linear properties of the studied permutations but does not draw the full picture.

Indeed, as seen in Chapter 1, because of clustering effects this approximation is not always accurate. In fact, when trails cluster together in the same differential or linear approximation, it can increase its differential probability or linear potential compared to the one of the trail alone. We study the clustering of two-round trails for our four ciphers in Section 3.1 by introducing a fourth histogram, the *cluster histogram*. We show that this clustering behaviour is negligible in XOODOO while it is not the case for the other aligned permutations. Additionally, we present an algorithm searching for clusters of three-round trails. This algorithm is sufficiently efficient for XOODOO and is thus used to extend the analysis of clustering of trails for this permutation by showing that there exists no clustering of three-round trails up to weight 50.

Finally, in Section 3.2 we investigate another approximation seen in Chapter 1: the hypothesis that round differentials are independent, which is known to be false for RIJNDAEL for two-round trails [DR07]. This approximation can cause the differential probability of trails to be underestimated: this leads to an optimistic description of the differential and linear properties of the permutation under study. We focus on the three-round case and we present an algorithm searching for round differential dependence and independence. This algorithm is applied to XOODOO and show that every round differential of three-round differential trails up to weight 50 are independent.

3.1 Clustering

In this section, we investigate clustering of differential trails and of linear trails. The occurrence of such clustering in two-round differentials and linear approximations requires certain conditions to be satisfied. In particular, we define an equivalence relation of states with respect to a linear layer and a box partition that partitions the state space in candidate two-round trail cores and the size of its equivalence classes upper bounds the amount of possible trail clustering. This is the so-called cluster partition.

We present the partitions of our four permutations by means of their cluster histograms. For all four permutations, we report on two-round trail clustering and for XOODOO in particular we look at the three-round case. With its unaligned structure, we found little clustering in XOODOO. However, the effects of clustering are apparent in the aligned primitives RIJNDAEL, SATURNIN, and SPONGENT, with them being most noticeable in RIJNDAEL.

3.1.1 The cluster histogram

To define the cluster histogram we need to define two equivalence classes.

Definition 3.1.1 (Box-activity equivalence)**.** Two states are box-activity-equivalent if they have the same activity pattern with respect to a box partition Π:

$$\boldsymbol{a} \sim \boldsymbol{a}' \text{ if and only if } r_\Pi(\boldsymbol{a}) = r_\Pi(\boldsymbol{a}')\,.$$

We denote the set of states that are box-activity equivalent with $\boldsymbol{a}$ by $[\boldsymbol{a}]_\sim$ and call it the *box-activity class* of $\boldsymbol{a}$.

Box-activity equivalence has an application in the relation between trail cores and differentials or linear approximations.

We now state a lemma that motivates the use of this newly defined equivalence class.

Lemma 3.1.2. *Two trail cores* $(\boldsymbol{a}_0, \boldsymbol{b}_0, \cdots, \boldsymbol{a}_{r-2}, \boldsymbol{b}_{r-2})$ *and* $(\boldsymbol{a}_0^*, \boldsymbol{b}_0^*, \cdots, \boldsymbol{a}_{r-2}^*, \boldsymbol{b}_{r-2}^*)$ *over a function* $f = \mathrm{N}_{r-1} \circ \mathrm{L}_{r-2} \circ \mathrm{N}_{r-2} \circ \cdots \circ \mathrm{L}_0 \circ \mathrm{N}_0$ *that are in the same differential (or linear approximation) satisfy* $\boldsymbol{a}_0 \sim \boldsymbol{a}_0^*$ *and* $\boldsymbol{b}_{r-2} \sim \boldsymbol{b}_{r-2}^*$.

Proof. Let $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{\Delta}_{\text{out}})$ be the differential (or linear approximation) over f that the trail cores are in. Since N_0 and N_{r-2} preserve activity patterns (see Lemma 2.1.7), we have that $\boldsymbol{\Delta}_{\text{in}} \sim \boldsymbol{a}_0$, and $\boldsymbol{\Delta}_{\text{in}} \sim \boldsymbol{a}_0^*$, and $\boldsymbol{\Delta}_{\text{out}} \sim \boldsymbol{b}_{r-2}$, and $\boldsymbol{\Delta}_{\text{out}} \sim \boldsymbol{b}_{r-2}^*$. From the symmetry and transitivity of $\sim$ it follows that $\boldsymbol{a}_0 \sim \boldsymbol{a}_0^*$ and $\boldsymbol{b}_{r-2} \sim \boldsymbol{b}_{r-2}^*$. □

Considering the case $r = 2$ in Lemma 3.1.2 immediately gives rise to a refinement of box-activity equivalence.

Definition 3.1.3 (Cluster equivalence)**.** Two states are cluster-equivalent with respect to a linear mapping $\text{L} : \mathbb{F}_2^b \to \mathbb{F}_2^b$ and a box partition Π if they are box-activity equivalent before L and after it (See Figure 3.1):

$$\boldsymbol{a} \approx \boldsymbol{a}' \text{ if and only if } \boldsymbol{a} \sim \boldsymbol{a}' \text{ and } \text{L}(\boldsymbol{a}) \sim \text{L}(\boldsymbol{a}')\,.$$

We denote the set of states that are cluster-equivalent with $\boldsymbol{a}$ by $[\boldsymbol{a}]_\approx$ and call it the *cluster class* of $\boldsymbol{a}$. The partition of $\mathbb{F}_2^b$ according to these cluster classes is called the *cluster partition.*

Corollary 3.1.4 (Two-round trail)**.** *If two two-round trail cores* $(\boldsymbol{a}, \text{L}(\boldsymbol{a}))$ *and* $(\boldsymbol{a}^*, \text{L}(\boldsymbol{a}^*))$ *over* $f = \text{N} \circ \text{L} \circ \text{N}$ *are in the same differential, then* $\boldsymbol{a} \approx \boldsymbol{a}^*$.

Proof. If we apply Lemma 3.1.2 to the case $r = 2$, we have $\boldsymbol{a} \sim \boldsymbol{a}^*$ and $\text{L}(\boldsymbol{a}) \sim \text{L}(\boldsymbol{a}^*)$. It follows that $\boldsymbol{a} \approx \boldsymbol{a}^*$. □

Corollary 3.1.4 shows that the defining differences (or linear masks) of any two-round trail cores that cluster together are in the same cluster class. It follows that if these cluster classes are small, then there is little clustering.

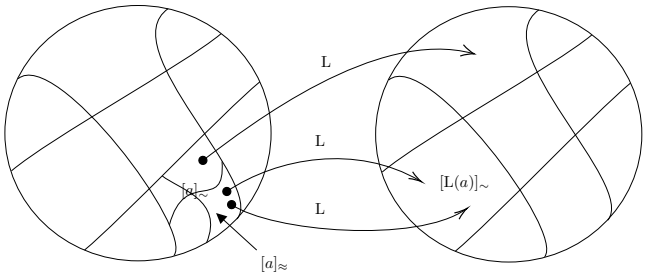


Figure 3.1: Partitions of $\mathbb{F}_2^b$ defined by $\sim$ and $\approx$.

For all $\boldsymbol{a}' \in [\boldsymbol{a}]_\approx$ the box weight $\text{w}_\Pi(\boldsymbol{a}') + \text{w}_\Pi(\text{L}(\boldsymbol{a}'))$ is the same since the activity patterns are equal by definition. Hence, this number is an invariant of the equivalence class. If we let $\widetilde{\text{w}} : \mathbb{F}_2^b/\!\approx \to \mathbb{N}$, given by $\widetilde{\text{w}}([\boldsymbol{a}]_\approx) = \text{w}_\Pi(\boldsymbol{a}) + \text{w}_\Pi(\text{L}(\boldsymbol{a}))$, be the function that maps an equivalence class to this number, then we see that $\widetilde{\text{w}}$ is well-defined.

In a similar way than for the bit and box weight histograms, we define an histogram from $\widetilde{\text{w}}$.

Definition 3.1.5 (Cluster histogram)**.** Let $\text{L} : \mathbb{F}_2^b \to \mathbb{F}_2^b$ be a linear transformation. Let $\approx$ be the equivalence relation given in Definition 3.1.3.

The *cluster histogram* $\text{N}_{\Pi,\text{L}} : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ of L with respect to the box partition Π is given by:

$$\text{N}_{\Pi,\text{L}}(k, c) = \#\{[\boldsymbol{a}]_\approx \in \mathbb{F}_2^b/\!\approx \mid \widetilde{\text{w}}([\boldsymbol{a}]_\approx) = k \wedge \#[\boldsymbol{a}]_\approx = c\}\,.$$

For a fixed box weight, the cluster histogram shows the distribution of the sizes of the cluster classes with that box weight. Ideally, for small box weights, the cluster classes are all very small. Large cluster classes of small weight may lead to two-round trails with a large differential probability or linear potential.

3.1.2 Computing the cluster histogram for an aligned round function

Given an aligned round function R we can exploit its superbox structure to compute the cluster histogram of its linear layer in the same fashion as in Subsection 2.1.2.

Lemma 3.1.6 (Cluster histogram of the linear layer of an aligned round function)**.** *Given an aligned round function* $\mathrm{R} = \pi \circ \mathrm{M} \circ \mathrm{N}$ *(or* $\mathrm{R} = \mathrm{M} \circ \pi \circ \mathrm{N}$*), the cluster histogram of* M *and* $\mathrm{L} = \pi \circ \mathrm{M}$ *(or* $\mathrm{L} = \mathrm{M} \circ \pi$*) are equal.*

Proof. We use the exact same reasoning as for the proof of Lemma 2.1.9. □

As seen before, the mixing layer M of an aligned round function can be decomposed as s functions M_i applied along a partition Π_{M} of which the box partition Π from the nonlinear layer N is a refinement. Let p be the number of m-bit boxes of Π_{N} in each of the s boxes of Π_{M}. As for the box weight histogram, the cluster histogram of the mixing layer M can be derived from the one of M_i using an equation similar to Equation (2.1).

The MDS case. We now focus on the computation of the cluster histogram of M_i when it is an MDS function as defined in Definition 2.1.11.

Since it is MDS, the branch number of M_i seen as a function from $\mathbb{F}_{2^m}^p$ to itself is $p+1$. Thus, $\mathrm{N}_{\Pi,\mathrm{M}_i}(k, c) = 0$ for any $c \in \mathbb{N}$ and $k < p + 1$.

We now prove the following lemma, for the case $k = p + 1$:

Lemma 3.1.7. *For $k = p + 1$, the only non-zero value in the cluster histogram $\mathrm{N}_{\Pi,\mathrm{M}_i}(k, \cdot)$ of M_i is for $c = \binom{2p}{k}$ and its value is $2^m - 1$.*

In other words, there are $\binom{2p}{k}$ cluster classes of weight $p + 1$, each of size $2^m - 1$.

Proof. Since M_i is an MDS function, there exists a $p \times p$ matrix $\boldsymbol{M}$ over $\mathbb{F}_{2^m}$ such that $\boldsymbol{H} = \begin{pmatrix}\boldsymbol{M}^t & \boldsymbol{I}_p\end{pmatrix}$ is a parity-check matrix of the MDS code associated with M_i of dimension p.

Let $\mathcal{S} \subseteq [\![1, 2p]\!]$ be a subset of the column index space of $\boldsymbol{H}$ with $\#\mathcal{S} = p$. The columns of $\boldsymbol{H}$ indexed by $\mathcal{S}$ form a $p \times p$ sub-matrix. Since $\boldsymbol{H}$ defines an MDS code, this sub-matrix is invertible. Let $\boldsymbol{H}'$ be the result of permuting the columns of $\boldsymbol{H}$ such that those originally indexed by $\mathcal{S}$ in $\boldsymbol{H}$ are now in position 1 to p in $\boldsymbol{H}'$ and let $\boldsymbol{H}''$ be the row reduced echelon form of $\boldsymbol{H}'$. Then, $\boldsymbol{H}'' = \begin{pmatrix}\boldsymbol{M}' & \boldsymbol{I}_p\end{pmatrix}$ is the parity-check matrix of a new code which defines a linear map $\mathrm{M}' : \mathbb{F}_{2^m}^p \to \mathbb{F}_{2^m}^p$ given by $\mathrm{M}'(\boldsymbol{a}) = \boldsymbol{M}'^t\boldsymbol{a}$.

Now, let $\boldsymbol{a} \in \mathbb{F}_2^{pm}$ be such that $\mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{M}_i(\boldsymbol{a})) = p+1$. Pick a subset $\mathcal{S}$ in such a way that it contains the index of one active box of $(\boldsymbol{a} \,\|\, \mathrm{M}_i(\boldsymbol{a}))$ and such that the other indices correspond to $p - 1$ passive boxes. The value of the remaining p active boxes are completely determined by the value in the chosen active box through the corresponding M'. There are $2^m - 1$ different possible values.

Since there are $\binom{2p}{p+1}$ different possible activity patterns of weight $p + 1$, there are $\binom{2p}{p+1}$ different cluster classes each of size $2^m - 1$. □

We extend Lemma 3.1.7 as the following recurrence relation:

Theorem 3.1.8. *For $p+1 \le k \le 2p$, all the $\binom{2p}{k}$ cluster classes of weight k have the same size. Let $C(k)$ be the size of these cluster classes.*

The following recurrence relation holds for $p + 1 \le k \le 2p$:

$$C(k) = (2^m - 1)^{k-p} - \sum_{1 \le i \le k-p-1} \binom{p}{i} C(k - i)$$

Moreover, $C(0) = 1$.

Proof. Let $\boldsymbol{a} \in \mathbb{F}_2^{pm}$ with $\mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{M}(\boldsymbol{a})) = k$. By the same argument as given in Lemma 3.1.7, pick $\mathcal{S}$ such that it contains the indices of $k - p$ active boxes.

There are $(2^m - 1)^{k-p}$ ways of choosing the vector $\boldsymbol{a}$ such that it is active in $k - p$ boxes. It follows that $p + 1 \leq \mathrm{w}_\Pi((\boldsymbol{a}, \mathrm{M}(\boldsymbol{a}))) \leq k$. We then subtract the number of vectors that lead to a box weight of $p + i$ for $1 \leq i \leq k - p - 1$ and obtain the result.

This is independent from the exact activity pattern of $\boldsymbol{a}$, thus every cluster class has the same size. □

General case. As for the bit and box weight histograms, there is no general formula to compute efficiently the cluster histogram of an unaligned permutation. The details on how the cluster histogram is computed for XOODOO should appear in Daniël Kuijsters' thesis [Kui].

3.1.3 The cluster histograms of our permutations

Next, we present the cluster histograms of the superboxes of RIJNDAEL, SATURNIN and of the SATURNIN hyperbox computed using Theorem 3.1.8 in Table 3.1a. We also present the cluster histogram of SPONGENT in Table 3.1b. Moreover, we present a partial cluster histogram of XOODOO in Table 3.1c.

Table 3.1: Cluster histograms of the four permutations studied

(a) The cluster histograms of RIJNDAEL and SATURNIN.

	$N \times C_{m,n}$		
$\widetilde{\mathrm{w}}$	RIJNDAEL superbox $m = 8, n = 4$	SATURNIN superbox $m = 4, n = 4$	SATURNIN hyperbox $m = 16, n = 4$
5	(56×255)	(56×15)	(56×65535)
6	$(28 \times 64005))$	(28×165)	(28×4294574085)
7	(8×16323825)	(8×2625)	$(8 \times 281444913315825)$
8	(1×4162570275)	(1×39075)	$(1 \times 18444492394151280675)$

(b) The cluster histogram of `SpongentMix` of SPONGENT.

$\widetilde{\mathrm{w}}$	$N \times C$
2	(16×1)
3	(48×1)
4	$(32 \times 1)\,(36 \times 7)$
5	$(8 \times 1)\,(48 \times 25)$
6	$(12 \times 79)\,(16 \times 265)$
7	(8×2161)
8	(1×41503)

(c) Partial cluster histogram (up to translation equivalence) of XOODOO.

$\widetilde{\mathrm{w}}$	$N \times C$
4	(3×1)
7	(24×1)
8	(600×1)
9	(2×1)
10	(442×1)
11	(10062×1)
12	(80218×1)
13	(11676×1)
14	$(228531 \times 1)\,(3 \times 2)$
15	$(2107864 \times 1)\,(90 \times 2)$
16	$(8447176 \times 1)\,(702 \times 2)$
⋮	⋮

In Table 3.1, C denotes the cardinality of a cluster class and N denotes the number of cluster classes with that cardinality.

Example 3.1: Reading the cluster histograms

In Table 3.1, an expression such as $(32 \times 1)\,(36 \times 7)$ means that there are 32 cluster classes of cardinality 1 and 36 classes of cardinality 7. Looking at $\widetilde{\mathrm{w}} = 8$ across the three tables, we see that RIJNDAEL, SATURNIN, and SPONGENT have only a single cluster class containing all the states with $\mathrm{w}_\Pi(\boldsymbol{a}) + \mathrm{w}_\Pi(\mathrm{L}(\boldsymbol{a})) = 8$. In contrast, for XOODOO, each state $\boldsymbol{a}$ sits in its own cluster class. This means that $\mathrm{L}(\boldsymbol{a})$ is in a different box activity class than $\mathrm{L}(\boldsymbol{b})$ for any $\boldsymbol{b} \in [\boldsymbol{a}]_\sim$ and $\boldsymbol{b} \neq \boldsymbol{a}$.

Table 3.1b gives the cluster histogram of SPONGENT's superbox. For weights above 4 we see large cluster equivalence classes. Hence, we expect to see clustering of differential and linear trails in differentials and linear approximations, respectively.

Now, consider the cluster histogram of XOODOO in Table 3.1c. We see that up to and including box weight 13, we have $\#[\boldsymbol{a}]_{\approx} = 1$. For box weight $14, 15$, and 16, we see that $\#[\boldsymbol{a}]_{\approx} \leq 2$. Due to its unaligned structure, it is less likely that equal activity patterns are propagated to equal activity patterns. Therefore, many cluster classes contain only a single state.

3.1.4 Two-round trail clustering

Two-round trail clustering in the keyed RIJNDAEL superbox was investigated in [DR06]. In that paper the *expected* differential probability of trails and differentials are studied, where expected means averaged over all keys. We see considerable clustering in differentials with 5 active S-boxes. For these, the maximum expected differential probability of differentials is more than a factor 3 higher than the maximum expected differential probability of two-round trails, with differentials containing up to 75 trails. For more active S-boxes the number of trails per differential is much higher and hence clustering is worse, but their individual contributions to the expected differential probability are much smaller and all differentials have expected differential probability very close to 2^{-32}. For fixed keys or in an unkeyed superbox these differentials and trails have a differential probability that is a multiple of 2^{-31}. This effect on differential trails was previously studied in [DR07]. In this section we report on our experiments on the other three permutations where we compare two-round differentials with differential trails and linear approximations with linear trails.

Figure 3.2 shows the number of two-round differentials and differential trails up to a given weight of the SATURNIN and the SPONGENT superboxes. For example, for SATURNIN there is a trail with 5 active S-boxes that has a weight of 10. The corresponding differential has a weight that is also approximately equal to 10.

In both cases, we see that for low weight the histograms are close and as the weight grows, these histograms diverge. For SATURNIN there are roughly 50 times more differentials with weight 15 or less than differential trails with weight 15 or less. For SPONGENT this ratio is roughly 20. This divergence is due to two reasons: differential trails of higher weight clustering to give a differential of lower weight, and what we call *clipping*.

Due to the large number of differential trails and the limited width of the superbox, the trails of high weight have to cluster in differentials of lower weight. For example in SATURNIN where superboxes have width 16, differentials of weight greater than 16 cannot exist. In fact since any differential over a superbox has an even number of ordered pairs, the minimum differential probability is 2^{-15}, yielding a maximum weight of 15. We call this effect clipping. In all generality, a trail over a k-bit superbox with weight $\mathrm{w} \leq k$ cannot have a differential probability equal to $2^{-\mathrm{w}}$ as this would imply a fractional number of pairs. Thus, the lowest differential probability of this trail is 2^{1-k}. This effect has been studied in RIJNDAEL and we refer to Section 3.2 for a discussion.

Figure 3.3 shows the weight histograms for two-round differentials and linear approximations. It can be compared with the weight histograms of two-round trails in Figure 2.2.

The full-state correlation weight histogram of SATURNIN was obtained from that of any of its columns by first rounding the correlation weights to the nearest integer to make integer arithmetic possible. The full-state correlation weight histogram of SPONGENT was obtained in a similar manner. The remainder of the histograms is exact. Table 3.1c shows that in XOODOO almost all differentials contain only a single trail. This means that the clustering is negligible. Therefore, there is no difference between Figure 2.2 and Figure 3.3 for XOODOO. For SATURNIN the clustering is the most striking: while there are about 2^{30} differential trails with weight 15 or less, there are more than 2^{35} such differentials: a factor 30 times more. For linear trails we observe a similar effect. For SPONGENT the effect of clustering is less clear due to the fact that the trail weight histogram is quite bad to start with.

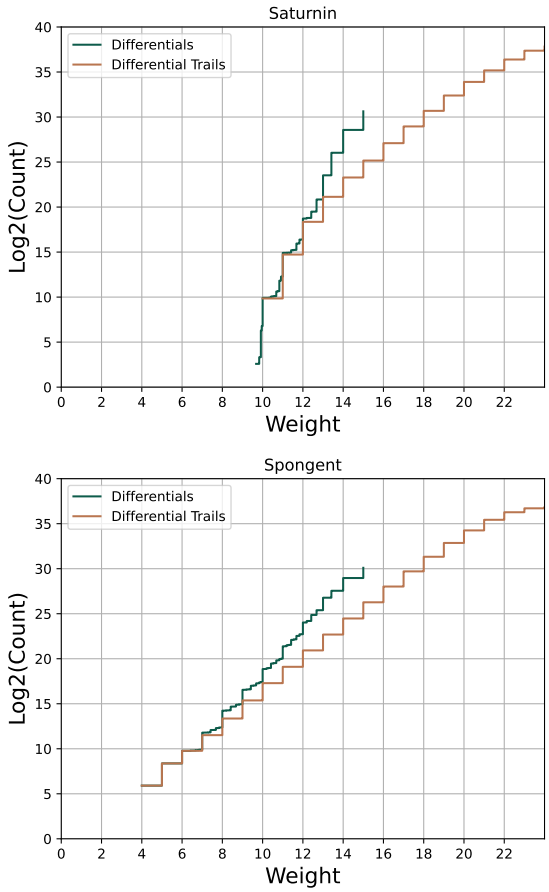


Figure 3.2: Cumulative count of two-round differentials and differential trails in the superboxes of Saturnin and Spongent.

The effect of clustering in four-round (or two super-round) Saturnin is interesting. Four-round Saturnin consists of the parallel application of four 64-bit hyperboxes. The consequence is that for a fixed key, the roughly $2^{127} \cdot 4$ differentials that are active in a single hyperbox and have non-zero differential probability, all have weight below 63. When computing expected differential probabilities averaging the differential probabilities over all round keys, this is closer to 64.

The cluster classes also determine the applicability of the very powerful *truncated differential attacks* [Knu94]. These attacks exploit sets of differentials that share the same box activity pattern in their input difference and the same box activity pattern in their output difference. Despite the fact that the individual trails in these truncated differentials may have very low differential probabilities, the joint probability can be significant due to the very high number of them. For two-round differentials the cluster classes are exactly the trail cores in a given truncated differential. In Table 3.1a we see that the cluster classes for the Rijndael superbox and Saturnin hyperbox are very large. This clustering leads to distinguishers for *e.g.* 4-round Rijndael and 8-round Saturnin. The latter can be modeled as 4 hyperboxes followed by an MDS mixing layer followed by 4 hyperboxes. An input difference with a single active hyperbox will have 4 active hyperboxes after 8 rounds, with probability 1. In contrast, if the cluster classes are small, as in the case of the unaligned Xoodoo permutation, it is very unlikely that truncated differential attacks would have an advantage over ordinary differential attacks.

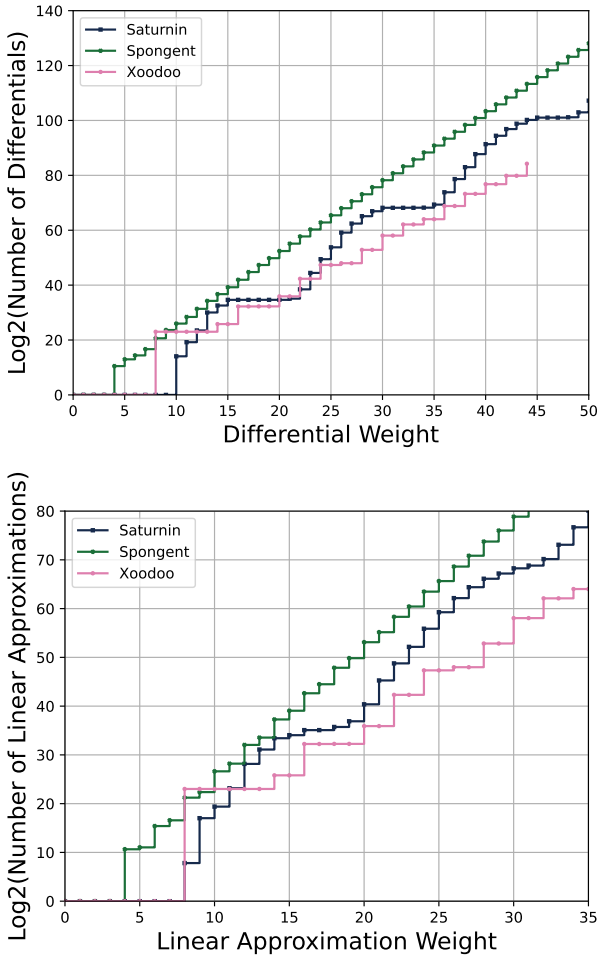


Figure 3.3: Two rounds: cumulative restriction and correlation weight histograms.

3.1.5 Three-round differential trail clustering in Xoodoo

According to [DHP+20], there exist no differential or linear trails over four rounds of Xoodoo with weight below 74. Additionally, Table 3.1c shows that trail clustering in two-round differentials in Xoodoo is negligible, as expected because of its unaligned design. We investigate the conjecture that Xoodoo's unaligned design also leads to negligible trail clustering for three rounds.

First, we present a generic technique that, given a three-round trail core, finds every trail that cluster with it. In other words, we find $\mathrm{DT}((\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}))$ for all enveloping trail $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ of the given trail core. We then apply the technique to known trail cores of Xoodoo, for which it is very efficient.

Given a trail core $(\boldsymbol{a}_1^*, \boldsymbol{b}_1^*, \boldsymbol{a}_2^*, \boldsymbol{b}_2^*)$, Lemma 3.1.2 shows that we can restrict our search to trails $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{\Delta}_{\mathrm{out}})$ such that $\boldsymbol{a}_1 \sim \boldsymbol{a}_1^*$ and $\boldsymbol{b}_2 \sim \boldsymbol{b}_2^*$. Figure 3.4 shows a differential trail over three rounds with the last linear layer omitted since the value of the difference after it deterministically depends on $\boldsymbol{\Delta}_{\mathrm{out}}$.

From the difference $\boldsymbol{a}_1^*$, we define the vector space A' of all the states in which a box is passive whenever it is passive in $\boldsymbol{a}_1^*$. In other words, if $\boldsymbol{a}_1 \in [\boldsymbol{a}_1^*]_\sim$, then $\boldsymbol{a}_1 \in A'$ but also, for it to be a vector space, A' includes states that are not in $[\boldsymbol{a}_1^*]_\sim$ but for which the activity pattern is *included* in the one of $\boldsymbol{a}_1^*$. We define a vector space B' from $\boldsymbol{b}_2^*$ in the exact same fashion. In other words, A' and B' contain the candidate for respectively $\boldsymbol{a}_1$ and $\boldsymbol{b}_2$ but are slightly augmented in

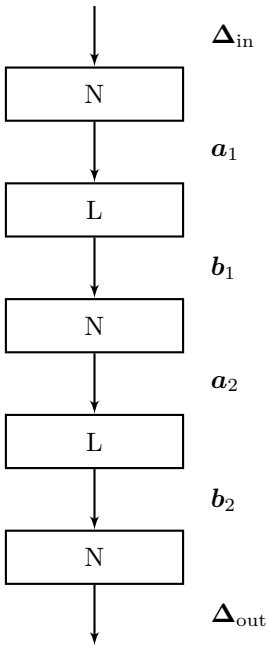


Figure 3.4: A differential trail over three-round (without the last linear layer)

order to be vector spaces of dimension respectively $m \times \mathrm{w}_\Pi(\boldsymbol{a}_1)$ and $m \times \mathrm{w}_\Pi(\boldsymbol{b}_2)$.

The vector space $B = \mathrm{L}(A')$ contains all possible candidate values for $\boldsymbol{b}_1$. Similarly, the vector space $A = \mathrm{L}^{-1}(B')$ contains all possible candidate values for $\boldsymbol{a}_2$. Since it preserves activity patterns (see Lemma 2.1.7), the central nonlinear layer N restricts the set of candidate values to those satisfying $\boldsymbol{b}_1 \sim \boldsymbol{a}_2$. Hence, we can limit the search to those $\boldsymbol{x} \in B$ and $\boldsymbol{y} \in A$ with $\boldsymbol{x} \sim \boldsymbol{y}$.

A naive exhaustive search can quickly be intractable, so we exploit the structure of the permutation to reduce the cost. To find all valid trails of the form $(\boldsymbol{\Delta}_{\text{in}}, a_1, b_1, a_2, b_2, \boldsymbol{\Delta}_{\text{out}})$, we first reduce the size of the space of all trail cores (a_1, b_1, a_2, b_2) using a necessary condition. When this space is small enough, we exhaustively search for a valid trail.

We write $\overline{\boldsymbol{B}}$ for a basis of B and $\overline{\boldsymbol{A}}$ for a basis of A. To reduce the dimension of the spaces, we define and apply an algorithm directly on their bases. Before doing so, we need the notion of *isolated active bit*.

Definition 3.1.9 (Isolated active bit)**.** A bit i of $\boldsymbol{b} \in \overline{\boldsymbol{B}}$ is said to be an *isolated active bit* if $\boldsymbol{b}_i = 1$ and $\boldsymbol{b}'_i = 0$ for all $\boldsymbol{b}' \in \overline{\boldsymbol{B}} \setminus \{\boldsymbol{b}\}$.

A basis vector having an isolated active bit determines the box activity of any linear combination that includes it as stated in the following lemma.

Lemma 3.1.10. *If $\boldsymbol{b} \in \overline{\boldsymbol{B}}$ has an isolated active bit at coordinate $i \in [\![1, b]\!]$, then any vector in the affine space $\boldsymbol{b} + \mathrm{span}(\overline{\boldsymbol{B}} \setminus \{\boldsymbol{b}\})$ has the corresponding box activated.*

Proof. If $\boldsymbol{b}$ has an isolated active bit at coordinate i, then the i-th bit of any vector in the affine space $\boldsymbol{b} + \mathrm{span}(\overline{\boldsymbol{B}} \setminus \{\boldsymbol{b}\})$ is active. As a result, the box in which lies this bit is active. □

Similarly to how an isolated active bit always activates the corresponding box, a box is never activated if no basis vector activates it:

Lemma 3.1.11. *If the i-th box is passive in every vector of $\overline{\boldsymbol{A}}$, then the i-th box is passive in all vectors of A. We say that box i is passive in $\overline{\boldsymbol{A}}$.*

To reduce the number of basis vectors of either base, we define a sufficient condition on basis vectors for it to be removed from the basis while ensuring that it does not exclude candidates that would lead to a valid differential over the central nonlinear layer.

Condition 3.1.12 (Reduction condition)**.** *We say that a basis vector $\boldsymbol{b} \in \overline{\boldsymbol{B}}$ satisfies the* reduction condition *if and only if it has an isolated active bit in a box that is passive in $\overline{\boldsymbol{A}}$. The same is true when swapping the role of $\overline{\boldsymbol{B}}$ and $\overline{\boldsymbol{A}}$.*

The following lemma shows that the reduction condition is sufficient to reduce the dimension of the vector space we consider.

Lemma 3.1.13. *If a basis vector $\boldsymbol{b} \in \overline{\boldsymbol{B}}$ satisfies Condition 3.1.12, then all valid differentials over the central nonlinear layer* N *are in* $\mathrm{span}(\overline{\boldsymbol{B}} \setminus \{\boldsymbol{b}\})$*. The same is true when swapping the role of $\overline{\boldsymbol{B}}$ and $\overline{\boldsymbol{A}}$.*

Proof. As a consequence of Lemma 3.1.10 and Lemma 3.1.11, a valid difference before the central nonlinear layer N cannot be constructed from $\boldsymbol{b}^{(i)}$ because it would contradict the fact that the activity pattern is preserved through N. □

We now define an algorithm that consists in repeatedly removing basis vectors from $\overline{\boldsymbol{B}}$ and $\overline{\boldsymbol{A}}$ that satisfy Condition 3.1.12 until this is no longer possible. Efficiency can be increased by searching for pivots for a Gaussian elimination among indices of vectors from $\overline{\boldsymbol{A'}}$ (respectively $\overline{\boldsymbol{B'}}$) that correspond to never activated boxes in $\overline{\boldsymbol{B'}}$ (respectively $\overline{\boldsymbol{A'}}$). Indeed, these pivots can be used to row-reduce the corresponding basis along them, thus revealing an isolated active bit.

To be part of a differential trail in any enveloping differential of the initial trail core, it is necessary and sufficient that a candidate differential $(\boldsymbol{b}_1, \boldsymbol{a}_2)$ meets all the following conditions:

— $(\boldsymbol{b}_1, \boldsymbol{a}_2)$ is a valid differential over N;

— There exists a $\boldsymbol{\Delta}_{\text{in}}$ such that both $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{a}_1^*)$ and $(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{a}_1)$ are valid differentials over N (where $\boldsymbol{a}_1 = \mathrm{L}^{-1}(\boldsymbol{b}_1)$);

— There exists a $\boldsymbol{\Delta}_{\text{out}}$ such that both $(\boldsymbol{b}_2^*, \boldsymbol{\Delta}_{\text{out}})$ and $(\boldsymbol{b}_2, \boldsymbol{\Delta}_{\text{out}})$ are valid differentials over N (where $\boldsymbol{b}_2 = \mathrm{L}(\boldsymbol{a}_2)$).

If the algorithm sufficiently decreases the dimensions, then we can exhaustively test all pairs $(\boldsymbol{b}_1, \boldsymbol{a}_2) \in B \times A$ remaining after reduction against these conditions. A cluster is found if, during this exhaustive search, we find at least one pair $(\boldsymbol{a}_1, \boldsymbol{b}_2)$ that is different from $(\boldsymbol{a}_1^*, \boldsymbol{b}_2^*)$.

Applying our method to all three-round trail cores of Xoodoo up to weight 50, given in [DHV+18b], shows that there exists no cluster for all these trails. The details of the search for three-round differential clusters in Xoodoo can be found in Table 3.2.

3.2 Independence of round differentials

In this section we study the dependence of round differentials in the sense of Definition 1.3.6. It has been found in [DR07] that the vast majority of trails over the Rijndael superbox have dependent round differentials. We expect that the dependence effects observed in Rijndael disappear in an unaligned permutation. As for the three-round clustering of trails, we present here a method to check for round differential dependence or independence given a three-round trail and we apply it to Xoodoo.

Table 3.2: Result of the search for three-round differential trails clustering in Xoodoo.

Trail weight	dim(B)	dim(A)	dim(B)	dim(A)	Valid trails
	Before reduction		After reduction		
36	18	18	3	3	1
36	6	42	1	2	1
38	15	33	5	8	1
46	12	45	2	2	1
46	24	30	1	1	1
46	6	57	3	10	1
46	15	45	1	2	1
46	12	45	2	3	1
48	12	48	2	2	1
48	9	51	2	4	1
48	12	36	4	8	1
48	18	36	3	4	1
48	21	39	1	2	1
48	21	39	4	7	1
48	12	48	2	3	1
48	18	36	3	3	1
48	6	60	1	3	1
50	21	39	2	3	1
50	12	51	2	3	1
50	18	39	3	4	1
50	12	54	4	10	1
50	21	39	2	2	1
50	15	51	5	9	1
50	24	36	5	7	1
50	6	63	1	3	1
50	24	36	2	2	1
50	15	42	3	5	1
50	12	54	2	3	1
50	15	51	4	7	1

3.2.1 Linear masks for differentials over nonlinear components

We note $V_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ the set of output states that follow the differential $(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ over N, *i.e.* $V_N(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}) = \mathrm{N}(U_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}}))$.

From [DR07, Lemma 4, 5, 6 and Property 2], we have that $U_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ and $V_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ are affine if $\#U_{\mathrm{S}_i}(P_i(\boldsymbol{\Delta}_{\mathrm{in}}), P_i(\boldsymbol{\Delta}_{\mathrm{out}})) \leq 4$ for each S-box. Since this assumption holds for our four permutations, in Xoodoo both $U_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ and $V_{\mathrm{N}}(\boldsymbol{\Delta}_{\mathrm{in}}, \boldsymbol{\Delta}_{\mathrm{out}})$ are affine. Thus, they

can be described by a system of affine equations on the bits of the input state $\boldsymbol{a}$. Each affine equation can be written as $\boldsymbol{u}^t\boldsymbol{a} + c$ with $\boldsymbol{u}$ a b-bit vector called mask and c a single bit..

Given a three-round differential trail $Q = (\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{a}_1, \boldsymbol{b}_1, \boldsymbol{a}_2, \boldsymbol{b}_2, \boldsymbol{\Delta}_{\text{out}})$ (see Figure 3.4), one can define four sets of masks:

— A_1, the masks that come from $V_{\text{N}}(\boldsymbol{\Delta}_{\text{in}}, \boldsymbol{a}_1)$, at the output of the first nonlinear layer;

— B_1, the masks that come from $U_{\text{N}}(\boldsymbol{b}_1, \boldsymbol{a}_2)$, at the input of the second nonlinear layer;

— A_2, the masks that come from $V_{\text{N}}(\boldsymbol{b}_1, \boldsymbol{a}_2)$, at the output of the second nonlinear layer;

— B_2, the masks that come from $U_{\text{N}}(\boldsymbol{b}_2, \boldsymbol{\Delta}_{\text{out}})$, at the input of the third and last nonlinear layer.

In this section, our goal is to prove that restrictions that come from three different rounds are *independent* in the sense that:

$$\#U_{\text{N}\circ\text{L}\circ\text{N}\circ\text{L}\circ\text{N}}(Q) = 2^{b-(\#A_1+\#B_1+\#B_2)} = 2^{b-(\#A_1+\#A_2+\#B_2)}\,.$$

which is, per Definition 1.3.6, equivalent to the independence of round differentials.

We first present a generic method for determining whether three-round trail masks are independent. Then we apply this method to Xoodoo for which the method is efficient.

3.2.2 Independence of masks over a nonlinear layer

Since L is linear, A_1 can be linearly propagated through it to obtain a set of masks A_1' at the input of the second nonlinear layer. Similarly, we can propagate B_2 through the inverse linear layer to obtain a set of masks B_2' at the output of the second nonlinear layer.

B_1 and A_1' form sets of masks at the input of the second nonlinear layer. If the rank of $C_1 = B_1 \cup A_1'$ is the sum of the ranks of B_1 and A_1', then C_1 contains independent masks. The same strategy can be used to test for dependence of masks in $C_2 = A_2 \cup B_2'$.

At this point, we suppose that the internal linear dependence of $C_1 = A_1' \cup B_1$ and $C_2 = A_1 \cup B_2'$ have been checked using the criterion of the rank. As for the independence of masks of the complete trail, we need to check for dependence between C_1 and B_2' or between A_1' and C_2. We apply an algorithm similar to the one we used in Subsection 3.1.5 to reduce bases. However, here we use it to reduce the cardinalities of the mask sets.

To do so, we will try to find the masks in either C_1 or B_2' which removal would cause the number of solutions to double. By iteratively removing them, we hope to reach the point where at least one of the two sets of masks is empty. In this case, since we have already checked that the remaining set is internally independent, we would have proven independence of all the starting masks. To find removable masks, we will use a condition that is analogous to Condition 3.1.12:

Condition 3.2.1. *Let $\boldsymbol{u}$ be a restriction in C_1. $\boldsymbol{u}$ is said to satisfy Condition 3.2.1 if and only if $\boldsymbol{u}$, seen as a linear mask, has an isolated bit that corresponds to an S-box that is not activated by any restrictions of B_2'. Condition 3.2.1 can be defined for restrictions in B_2' by swapping the role of C_1 and B_2'.*

Condition 3.2.1 is a sufficient condition for a restriction to be removed from the set of restrictions studied:

Lemma 3.2.2. *Let C_1 and B_2' be two sets of masks before and after a nonlinear layer composed of invertible S-boxes applied in parallel. If a mask $\boldsymbol{u}$ in C_1 satisfies Condition 3.2.1, then the number of states that satisfy the equations associated with the masks in both $C_1 \setminus \{\boldsymbol{u}\}$ and B_2' is exactly two times the number of solutions before removing $\boldsymbol{u}$. The same is true by swapping the role of C_1 and B_2'.*

Proof. Since $\boldsymbol{u}$ satisfies Condition 3.1.12, let i be the index of the isolated bit, j be the index of the corresponding S-Box and k the number of masks in B_2'.

No mask in B_2' is putting a constraint on any of the m bits of the j-th S-Box, thus the 2^{b-k} solutions can be seen as 2^{b-k-m} groups of 2^m different states that only differ in the m bits of the j-th S-box. Since the S-box is invertible, the application of the inverse of the nonlinear layer to a whole group of 2^m vectors results in a group of 2^m different states that, again, only differ on the value of the j-th S-box.

We can further divide those 2^{b-k-m} groups each into 2^{m-1} subgroups of 2 different states that only differ in the value of the i-th bit. By definition of an isolated bit, either both or none of the two states inside a subgroup satisfy all equations associated with the masks in $C_1 \setminus \{\boldsymbol{u}\}$.

Finally, inside a subgroup exactly one of the two states will satisfy the equation associated with mask $\boldsymbol{u}$. Thus, the number of solutions by removing $\boldsymbol{u}$ is multiplied by exactly two. □

Algorithm to check for (in)dependence. The whole algorithm to check for independence goes as follows: we first check for linear dependence inside C_1 by computing its associated rank. Then, we recursively check if some mask in either C_1 or B_2' satisfies Condition 3.2.1 and if it is the case we remove them from the sets of masks. We apply this process until one of the three following possible outcomes occurs:

— If C_1 is not full rank, we can conclude that masks in B_1 and A_1' are dependent;

— Else, if either set is empty, Lemma 3.2.2 applied backward at each step guarantees us that the number of states satisfying the equations associated with the masks in both C_1 and B_2' is equal to $2^{b-\left(\#C_1+\#B_2'\right)}$, that is to say the masks are independent;

— If none of the two outcomes above happened and no mask can be further removed, we cannot directly conclude about (in)dependence between remaining masks. However, we can try applying the same method to A_1 and C_2.

3.2.3 Application to Xoodoo

This process is used to check for independence in differential trails over three rounds of Xoodoo. It has been applied to the same differential trails as processed in Subsection 3.1.5. In all cases, the masks, and thus round differentials, were found to be independent. This was not obtained by sampling, but instead by counting the exact number of solutions, hence this independence is exact in the sense of Definition 1.3.6. As a result, the differential probability of each such trail is the product of the differential probabilities of its round differentials, which implies that $\mathrm{DP}(Q) = 2^{-\mathrm{w_r}(Q)}$.

Summary and future work

In this part of the thesis, we focused on the differential and linear properties of four permutations, namely Rijndael, Spongent, Saturnin and Xoodoo. After having proposed a formal definition in Chapter 1 for what alignment meant for cryptographic permutations, we investigated in Chapter 2 the decay of the mixing power of the linear layer between its analysis at bit-level and box-level. This effect, called huddling, is less visible for Xoodoo than for the aligned primitives under study. Also, we observed that the alignment property seems to have an impact on the repartition of trails by their weight, which is a more precise criterion than the branch number. Finally, we saw in Chapter 3 that alignment also has an impact on approximations made during the study of differential and linear properties and in particular that in the differential analysis of Xoodoo over three rounds, these approximations are correct.

One natural follow-up of this work would be to adapt the algorithm searching for clustering of differential trails to do the same for linear trails. The same adaptation could be done for the algorithm checking the (in)dependence of round differentials. Additionally, both algorithms could be applied to other permutations to have a clearer view on their behaviour over three rounds.

To continue the effort towards a more fine-grained understanding of the differential and linear properties of permutations, the formal framework we defined could be extended to study more than three rounds. More importantly, and in order to reach a stronger conclusion on the impact of alignment, this framework could be applied to other permutations following different design strategies.

Part II

High-order masking

Overview

In this part of the manuscript, we study another important aspect of the security of cryptographic applications: the robustness of their implementation against adversaries having physical access to the device on which the primitives are executing. These adversaries may be able to extract sensitive data or directly bypass cryptosystems only by making physical measurements on the device under attack. A physical quantity which variation depends on secret data and that is measurable by an attacker is called a *side-channel*. The attacks exploiting these side-channels are called *side-channel attacks*. In certain conditions, such as when the adversary can make precise measurements on the device, these attacks can be devastatingly effective.

In Chapter 4, we go over different kinds of countermeasures against side-channel attacks and present one of them in more details: *masking*. Masking consists in using a secret-sharing scheme to split the secret values into shares that are, when observed individually, independent from the secret. The number of shares that an attacker needs to observe to learn something on the underlying data is called the masking order and can be seen as a good indicator of the security level of a masked implementation. In Chapter 5, we give the definition of formal security models and describe how they can be used to build complex masked implementations from smaller components. We also explain how the cost of a masked implementation is linked to its masking order. Then, in Chapter 6, we show that, even for very formal and composable security models, verifying the security of a masked implementation can be too costly in itself. To address this issue, we introduce an algorithm based on the enumeration of potential attacks that has better verification performance than the state of the art. We implemented this algorithm as a tool and use it to design more efficient masking schemes. Finally, in Chapter 7 we implement a masked version of the Advanced Encrypted Standard (AES) at order 3 and 7 for ARM Cortex-M processors that is executing faster than the state of the art for these masking orders. Additionally, we experimentally assess its resistance against side-channel attacks.

Chapter 4

Introduction to masking

Contents

4.1 Side-channel attacks

As well-designed as a cryptographic algorithm may be, vulnerabilities may appear during its implementation and execution. Even if it is necessary to prove algorithms in high-level security models, one must not neglect security issues that arise when an algorithm becomes a program and when the program becomes a process. Software vulnerabilities through user data input or hardware tampering (*e.g.* fault injection, hardware Trojan, ...) are examples of attacks which goal is to deviate the process from its legitimate behaviour, allowing to bypass cryptographic protection. However, even when control-flow integrity is ensured and state-of-the-art cryptographic algorithms are used, there exists a whole range of passive attacks that can be used against a vulnerable implementation. *Side-channel attacks* are one of them.

Side-channel attacks is a class of attack that aims to exploit physical variations during the program execution. These physical variations are called *side-channels* and can be, for example, the time it takes for the program to execute [Koc96], the power consumption of the circuit [KJJ99] but also the electromagnetic radiations [QS01] or the acoustic waves [GST14] emitted during the computations. When these depend on secret internal data, an attacker may be able to extract information that would be not available to them in a model where they only have a "black box" access to the cryptographic function as it is the case in Known Plaintext Attacks (KPA), Chosen Plaintext Attacks (CPA) and Chosen Ciphertext Attacks (CCA). Even a partial knowledge of, for example, the secret key but also more broadly of the intermediate values of a program implementing cryptographic primitives can be devastating for its security. A recent example is given in January 2021 by Lomné and Roche when they published a practical side-channel attack against Titan security keys [LR21]. They exploited electromagnetic leakage during the computation of an ECDSA (Elliptic Curve Digital Signature Algorithm) signature to retrieve the full private key, allowing to generate new valid ECDSA signatures and thus to clone the security key.

Except for the special case of timing attacks where execution duration may be measured remotely, the attacker needs to have physical access to the device in order to do measurements. Thus, embedded devices are the most targeted devices. They can be smart cards, hardware wallets for cryptocurrencies, or embedded electronic components in a vehicle. All of these devices share the fact that they can be stolen, (partially) publicly reachable, or at least that they are made to be installed in or to be operated in a non-fully controlled environment. Additionally, those devices are often made to execute only a few features and rarely equipped with multiprocessors or other devices that can generate additional noise to measures. Thus, this fact along with the one that an attacker may be able to stand really close to it, induces low noise level during, *e.g.* electromagnetic, measurement which makes the attack easier on embedded devices than on any other setup.

4.2 Countermeasures against side-channel attacks

Embedded devices are the most susceptible to be targeted by side-channel attacks but they also have hard efficiency and production constraints. These constraints are even stronger than for other devices and they must be taken into account when designing and choosing countermeasures. These constraints may be stated in term of computation power, memory usage, physical size or manufacturing cost in general.

Most countermeasures against side-channel attacks aim at limiting the exploitability of the hypothetical signal an attacker could be able to measure from the device and to increase the cost (in time, technical expertise, hardware measurement tool, ...) of the attack. There are three main approaches for the design of countermeasures.

4.2.1 Reducing the signal strength

This first approach is the more obvious one: by reducing the signal strength, it is harder for an attacker to get useful information for it. To do so in practice turns out to be hard. Even if it is

possible to build an electromagnetic shield on the device keeping it from leaking electromagnetic radiations, it is far from perfect. First, this countermeasure requires to act very early in the production line and can hardly be applied to any device already manufactured. Also, in a context where the attacker has physical access to the device, this countermeasure can often be defeated by a direct intervention of the attacker voluntarily modifying or damaging the device shield to reduce its impact. Most importantly, this countermeasure is specific for a given side-channel: an electromagnetic shield will not protect against power analysis. Thus, protecting against a wide range of side-channels can be very costly without any guarantee that the attacker will not find a new side-channel to exploit.

4.2.2 Lowering measure reproducibility

The concrete measure of a side-channel over a period of time is classically called a *trace*. In practice, a single trace is rarely enough to successfully attack a device and the attacker often must gather additional traces in order to reduce measurement noise and improve signal quality. However to be really useful for an attack these traces must be synchronized, that is the attacker must be able to know which measurement points in the traces correspond to the same computation. This way, multiplying the number of traces actually increases the signal to noise ratio. Knowing that, countermeasures are designed to specifically make this synchronization difficult, for example by adding dummy code at random location during runtime [CCD00]. The goal is no longer to keep leakages from happening but instead to offset in a non-deterministic manner the point in time where the secret data are used by the program, making the analysis of traces much more difficult. Such a countermeasure has been proposed by Coron and Kizhvatov [CK09; CK10] at CHES in 2009 and 2010. However this approach has its limits: if only one trace is enough to attack the device [KPP20], desynchronization does not help; also, using signal processing and pattern recognition techniques, the points of leakages can be nonetheless identified and one may be able to counteract desynchronization attempts [DRS+12; CDP17].

4.2.3 Increasing the measurement noise

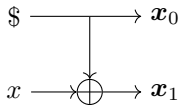
The last approach to keep attackers from exploiting side-channel leakages in practice is to create or amplify measurement noise. Doing so forces the attacker to record an increasingly higher number of traces until it is no longer possible to do so in practice. Noise can be generated leveraging architectural properties of the circuit [KBG09; GM11] but we will focus on a generic approach: masking.

4.3 Masking

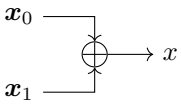
The concept at the root of masking is to use a secret sharing algorithm to split the sensitive data into multiple shares that are individually statistically independent from the original data. All computation are done on those shares such that what an attacker is observing looks like noise and do not directly depend on the sensitive data. This approach has been introduced simultaneously by Goubin *et al.* [GP99] and Chari *et al.* [CJR+99] in 1999. The word *masking* has first been used in this context by Messerges [Mes01] in 2001 where he applies this principle to AES finalists.

Most of the time an additive secret sharing scheme is used which consists in sharing a given value x into two or more shares. When using two shares $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$, their values is such that $\boldsymbol{x}_0$ is drawn uniformly at random and $x = \boldsymbol{x}_0 + \boldsymbol{x}_1$. This way, as for the one-time pad, both $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$ follow a uniform distribution. Figure 4.1 shows two boolean circuits: one realising the sharing and the other retrieving the original value from its sharing.

Masking is said to be at *order* d when the number of shares is equal to $d+1$. When using an additive secret sharing scheme, an order d masking is achieved by drawing uniformly at random the first d shares $\boldsymbol{x}_0, \cdots, \boldsymbol{x}_{d-1}$ and computing the last share $\boldsymbol{x}_d$ such that $x = \bigoplus_{i=0}^{d} \boldsymbol{x}_i$. This way, every subset of fewer than d shares is uniformly distributed. Figure 4.2 shows a generalized circuit for masking and unmasking at order d.

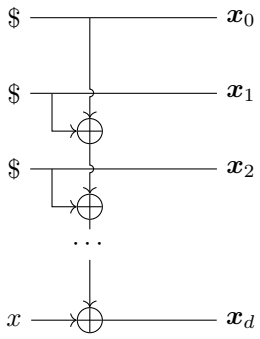


(a) A boolean masking circuit sharing x in $\boldsymbol{x} = (\boldsymbol{x}_0, \boldsymbol{x}_1)$ where \$ means drawing the value uniformly at random.

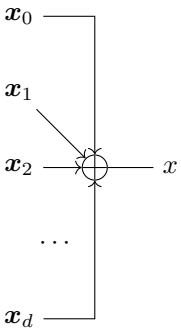


(b) A circuit taking two shares and retrieving the corresponding unmasked boolean value $x = \boldsymbol{x}_0 \oplus \boldsymbol{x}_1$.

Figure 4.1: Masking and unmasking circuits.



(a) A boolean masking circuit for x at order d.



(b) A circuit retrieving x from its sharing at order d.

Figure 4.2: Masking and unmasking circuits at order d.

However, it is not sufficient to have a way to securely share a secret. As in secure Multi-Party Computation, one also needs to be able to apply meaningful operations on the shared-secret without revealing the secret itself.

In Chapter 5 we first present the security models used to formally define the notion of security of masked circuits and show a generic method to design secure masked circuits from smaller ones. Then, we introduce in Chapter 6 a new algorithm to computationally verify the security of masked circuit. Finally, we implement a masked algorithm and assess its security by using experimental leakage assessment methods in Chapter 7.

Chapter 5

Designing masked circuit

Contents

Overview

We saw in Chapter 4 that masking techniques can be used to protect an implementation against side-channel attacks. In this chapter, we start be formally defining key masking concepts: masking gadgets and probes on these circuits. Then, we introduce the main security framework used for a formal analysis of masking gadgets which is the d-probing model. We next define concrete security models in this framework and more particularly we focus on compositional security models. These models are needed to adopt a generic and modular approach to the design of masked implementations based on the composition of smaller secure masked circuits. Finally, we discuss the cost of these smaller gadgets depending on the function they implement.

This chapter was in part published in an article at EUROCRYPT 2021 [BK21].

5.1 Formal security models: d-probing model and d-privacy

We start by recalling the needed definitions before giving the models definitions.

5.1.1 Gadgets and probes

To begin with, we give the definition of what we call a *masked gadget* implementing any function f:

Definition 5.1.1 (Gadgets)**.** Let $\mathbb{K}$ be an arbitrary finite field and let $f : \mathbb{K}^n \to \mathbb{K}^m$, $u, v \in \mathbb{N}$; a (u, v)-*gadget* for the function f is a randomised circuit C with output $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m) \in (\mathbb{K}^v)^m$ such that for every tuple $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{K}^u)^n$ and every set of random coins $\mathcal{R}$, $(\boldsymbol{y}_1, \ldots, \boldsymbol{y}_m) \leftarrow\!\shortmid C(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n; \mathcal{R})$ satisfies:

$$\left(\sum_{j=1}^{v} \boldsymbol{y}_{1,j}, \ldots, \sum_{j=1}^{v} \boldsymbol{y}_{m,j}\right) = f\left(\sum_{j=1}^{u} \boldsymbol{x}_{1,j}, \ldots, \sum_{j=1}^{u} \boldsymbol{x}_{m,j}\right).$$

We then use x_i to denote $\sum_{j=1}^{u} \boldsymbol{x}_{i,j}$, and similarly for y_i; $\boldsymbol{x}_{i,j}$ is called the jth *share* of x_i.

In this definition, a randomised circuit C is a directed acyclic graph whose vertices represent arithmetic operation *gates* (addition and multiplication) over $\mathbb{K}$ of arity two, or random gates of arity zero whose outputs are uniform over $\mathbb{K}$ and pairwise independent for every execution of the circuit, and recorded in the variable $\mathcal{R}$; the edges of the graph are *wires* that connect the input and output of the gates together so as to describe the full computation of a given function.

A *probe* on a circuit C is a map that for every execution $C(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n; \mathcal{R})$ returns the value propagated on one of the wires of C. One may further distinguish between *external* probes on the output wires (or output shares) $\boldsymbol{y}_{i,j}$ of C, and the remaining *internal* probes.

Example 5.1: Addition gadget with probes

A $(2, 2)$-gadget for the addition over $\mathbb{K} = \mathbb{F}_2$ is a circuit with four input wires: two shares for each of the two operands. The two output wires must be a valid sharing for the result of the addition, and that for all possible values produced by the random gates. Each input, intermediate and output wire can be probed.

We show in Figure 5.1 a $(2, 2)$-gadget for the addition in $\mathbb{F}_2$ with an external probe p_1 and two internal probes $\{p_2, p_3\}$.

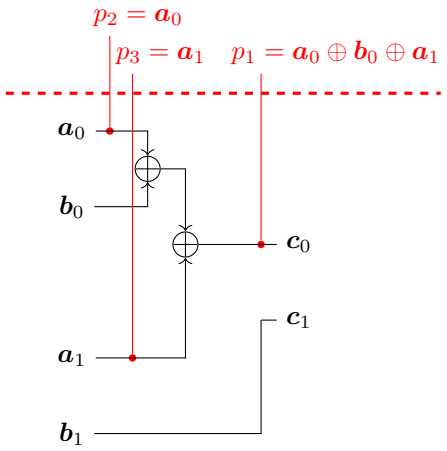


Figure 5.1: Toy addition gadget.

Example 5.2: Multiplication gadget

In Figure 5.2 we show an example of a $(2, 2)$-gadget for the multiplication over $\mathbb{F}_2$ along with a probe p. The $\otimes$ block computes every $a_i b_j$, $0 \leq i, j \leq d$, that is to say the tensor product of $\boldsymbol{a}$ and $\boldsymbol{b}$.

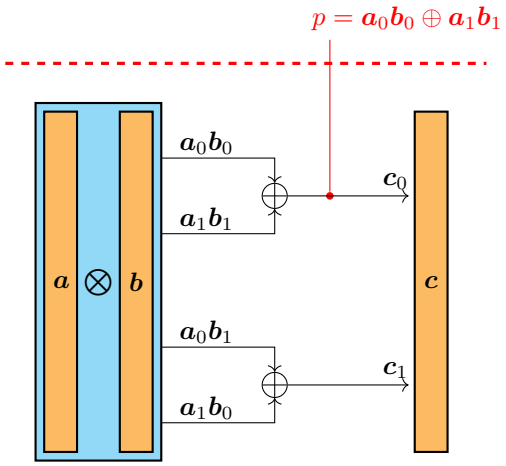


Figure 5.2: Insecure multiplication gadget.

In Example 5.3 we show that this gadget is insecure at order 1 in the security model described in Definition 5.1.2.

5.1.2 The d-probing attack model and the role of the order d

The *d-probing model* is an attack model where the attacker is given access to up to d probes on the target gadget. At first glance, this model can seem far from the real capabilities of an attacker: on one hand, an attacker is often not strictly limited by the number of probes, especially on a software implementation where the sequential execution can lead an attacker to observe the leakage of each operations over a whole period of time; on the other hand, an attacker observation always embeds measurement noise.

Another model, the *noisy leakage model*, has been introduced in the work of Chari *et al.* [CJR+99] and later extended by Prouff and Rivain [PR13]. In this model, the security is studied while considering noise perturbations on the shares and during the computations.

However, Duc *et al.* [DDF14] have proved that this more realistic model is, under reasonable assumptions, equivalent to the probing model.

Additionally, the masking order d plays an important role in the security because, for equivalent noise level, the number of measurements needed for a successful attack increases exponentially in d [DFS15]. Unfortunately, generic high-order schemes also come with a significant overhead as discussed in Section 5.4.

5.1.3 A first security model: d-privacy

To assess the security of a given circuit against side-channel attacks in a d-probing setting, we need to introduce security models. A first security model was proposed in 2003 by Ishai, Sahai and Wagner [ISW03] through the concept of d-privacy:

Definition 5.1.2 (d-privacy)**.** Let C be a (u, v)-gadget for $f : \mathbb{K}^n \to \mathbb{K}^n$. C is said to be d-private if for any set of d probes $\mathcal{P} = \{p_1, \ldots, p_d\}$ and for any $(x_1, \ldots, x_n), (x'_1, \ldots, x'_n) \in \mathbb{K}^n$ the two distributions

$$\{\mathcal{P}(x_1, \ldots, x_n)\}_{\mathcal{R}} \text{ and } \{\mathcal{P}(x'_1, \ldots, x'_n)\}_{\mathcal{R}}$$

are identical, where $\{\mathcal{P}(x_1, \ldots, x_n)\}_{\mathcal{R}}$ denotes the distribution over the random coins $\mathcal{R}$ of the tuple of values returned by the probes in $\mathcal{P}$ and where $\mathcal{R}$ is used for both the sharing of the $(x_1, \ldots, x_n)$ and the additional random coins needed by C.

This notion of d-privacy is rather intuitive to define the security of a gadget: the distribution of the values observed by an attacker must not depend on the concrete value on which the circuit is evaluating.

Example 5.3: d-privacy

The gadget shown in Example 5.1 is 1-private since there is no single probe whose distribution depends on either a or b. On the other hand, it is not 2-private because the distribution of $\{p_2, p_3\}$ depends on the value of a: $p_2 \oplus p_3 = a$.

The gadget shown in Example 5.2 is not 1-private. To see why, let us compute the conditional probability of $p = 0$ knowing $a = 0$ and $b = 0$, $\mathbb{P}[p = 0 \mid a = 0, b = 0]$. We have that:

$$a = 0, b = 0 \implies \boldsymbol{a}_0 = \boldsymbol{a}_1, \boldsymbol{b}_0 = \boldsymbol{b}_1 \implies p = \boldsymbol{a}_0\boldsymbol{b}_0 \oplus \boldsymbol{a}_1\boldsymbol{b}_1 = 0$$

Thus $\mathbb{P}[p = 0 \mid a = 0, b = 0] = 1 \neq \mathbb{P}[p = 0 \mid a = 1, b = 1] = 0.5$. The distribution of p depends on the input a and b, which implies that the gadget is not 1-private.

5.2 Compositional security models

It was shown in 2013 by Coron *et al.* [CPR+13] that the sequential composition of two d-private gadgets does not necessarily yield a d-private circuit. In order to be able to build bigger circuits by composing gadgets while having guarantees on their security, new security models have been introduced by Barthe *et al.* at CCS 2016 [BBD+16].

5.2.1 (Strong) Non-Interference

These new models are based on the following definition:

Definition 5.2.1 (t-Simulatability)**.** Let C be a (u, v)-gadget for $f : \mathbb{K}^n \to \mathbb{K}^n$, and ℓ, $t \in \mathbb{N}$. A set $\mathcal{P} = \{p_1, \ldots, p_\ell\}$ of probes on C is said to be *t-simulatable* if $\exists\, I_1, \ldots, I_n \subseteq [\![1, u]\!]$; $\#I_i \leq t$ and a randomised function $\pi : (\mathbb{K}^t)^n \to \mathbb{K}^\ell$ such that for any fixed $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{K}^u)^n$, $\{p_1, \ldots, p_\ell\}_{\mathcal{R}} \sim \{\pi(\{\boldsymbol{x}_{1,i},\, i \in I_1\}, \ldots, \{\boldsymbol{x}_{n,i},\, i \in I_n\})\}_{\mathcal{R}}$.

Less formally, a set $\mathcal{P}$ of probes on C is t-simulatable if there exists a randomised function that perfectly simulates the distribution of $\{p_1, \ldots, p_\ell\}$ while requiring at most t shares of every input to C to do so. It is important to remark here that the simulation is done w.r.t. a *fixed* input $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, regardless of the fact that one may randomise these inputs across many executions of C.

Intuitively, a set of probe that is t-simulatable does not give more informations than the knowledge of t shares for each input precisely because the probes can be simulated with those shares.

Example 5.4: t-simulatability

In Figure 5.3 we show an example of the use of a random gate generating the value labelled r in a multiplication gadget over $\mathbb{K} = \mathbb{F}_2$. r is drawn uniformly at random in $\mathbb{K}$ at each execution. Although it appears twice in the circuit it is in fact the output of a single random gate, duplicated for convenience.

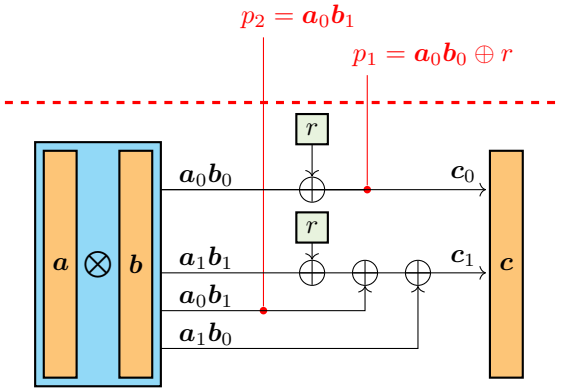


Figure 5.3: Generic scheme from Ishai, Sahai and Wagner [ISW03] instantiated at order $d = 1$.

The probe p_1 is 0-simulatable because, for fixed $\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{b}_0$, and $\boldsymbol{b}_1$, the distribution of the value taken by p is uniform thanks to r. It thus can be perfectly simulated without the knowledge of any input share.

The probe p_2 is not 0-simulatable but is 1-simulatable: one perfectly simulate its distribution given only one share of each input, namely $\boldsymbol{a}_0$ and $\boldsymbol{b}_1$, but cannot without them.

Definition 5.2.1 is only a property on a given set of probes but it can be used to define the following property characterizing the security of a gadget:

Definition 5.2.2 (d-Non-interference)**.** A (u, v)-gadget C for a function over $\mathbb{K}^n$ is *d-non-interfering* (d-NI) if and only if for any set $\mathcal{P}$ of at most d probes on C $\exists t \leq d$ such that $\mathcal{P}$ is t-simulatable.

This notion of d-Non-interference can be reformulated in a less formal way as follows: given any set of d or less probes on a d-NI circuit an attacker is gaining at most as much information as the knowledge of d shares on each input. The distribution of those d shares being uniform and independent of the value they mask, the whole gadget is d-private. Thus d-NI implies d-privacy, but the converse is not true.

Example 5.5: A d-private circuit that is not d-NI

A gadget can be d-private while having a set of probes that is not d-simulatable, meaning that it is not d-NI.

For example, let us look at the circuit presented in Example 5.1. It can be shown that it is 1-private because there is no single probe having a distribution that depends on either a, b or c. However, the probe $p_1 = \boldsymbol{a}_0 \oplus \boldsymbol{b}_0 \oplus \boldsymbol{a}_1$ cannot be simulated from only a single share of $\boldsymbol{a}$ and thus is an attack against the 1-NI property.

In some contexts, we need a slight variation of the d-Non-interference notion:

Definition 5.2.3 (d-Tight non-interference)**.** A (u, v)-gadget C for a function over $\mathbb{K}^n$ is *d-tight-non-interfering* (d-TNI) if and only if any set of $t \leq d$ probes on C is t-simulatable.

However, the notions of d-(tight)Non-interference are not sufficient to ensure composability and one last security notion is needed:

Definition 5.2.4 (d-Strong non-interference)**.** A (u, v)-gadget C for a function over $\mathbb{K}^n$ is *d-strong non-interfering* (d-SNI) if and only if for every set $\mathcal{P}_1$ of d_1 internal probes and every set $\mathcal{P}_2$ of d_2 external probes such that $d_1 + d_2 \leq d$, then $\mathcal{P}_1 \cup \mathcal{P}_2$ is d_1-simulatable.

To prove that a given set of probe is not an attack against the d-TNI notion one needs to simulate it using as many input shares as the cardinality of the set of probe, whereas in the d-SNI setting the external probes do not provide input shares for the simulation. It is thus harder to simulate a set of probe in the latter case than in the former. Thus, a d-SNI circuit is also d-TNI. Using the same reasoning, we see that d-TNI implies d-NI.

However, tight non-interference does not imply strong non-interference. Also, non-interference and tight non-interference are in fact equivalent [BBD+16] which in proofs allows to select the most convenient notion between d-TNI and d-NI.

Example 5.6: A d-NI circuit that is not d-SNI

In Figure 5.4 we show a multiplication (3,3)-gadget over $\mathbb{F}_2$ with two probes, one internal (p_1) and one external (p_2):

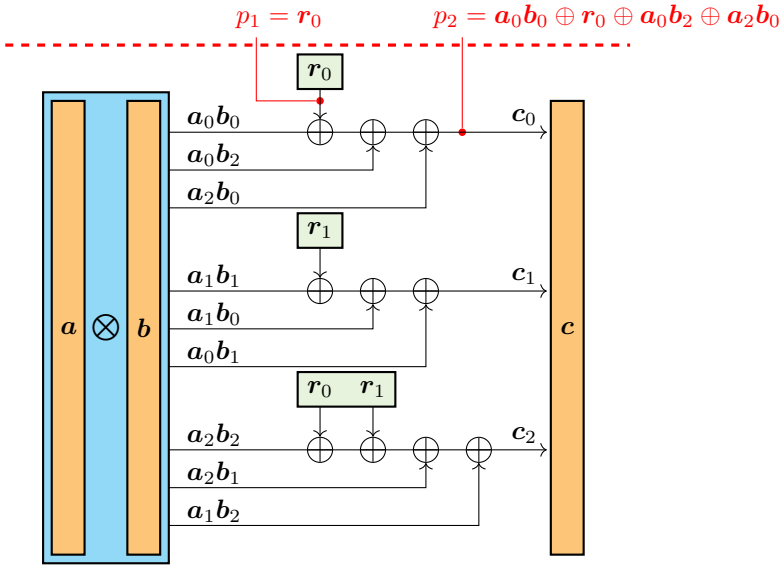


Figure 5.4: Scheme from Belaïd *et al.* [BBP+16, Algorithm 4].

The set $\{p_1, p_2\}$ can be perfectly simulated using two shares of each input, namely $\boldsymbol{a}_0$, $\boldsymbol{a}_2$, $\boldsymbol{b}_0$ and $\boldsymbol{b}_2$. For this gadget to be 2-SNI this set of probes must be simulated using only one share of each input since there is only one internal probe, p_1. However, this is not the case because $p_1 \oplus p_2 (= \boldsymbol{a}_0\boldsymbol{b}_0 \oplus \boldsymbol{a}_0\boldsymbol{b}_2 \oplus \boldsymbol{a}_2\boldsymbol{b}_0)$ requires at least the knowledge of $\boldsymbol{a}_0$, $\boldsymbol{a}_2$, $\boldsymbol{b}_0$ and $\boldsymbol{b}_2$ to be perfectly simulated.

Those security models are designed to ease the secure combination of gadgets. Belaïd *et al.* [BGR18] proved the following lemma, which is an implication of the d-SNI notion:

Lemma 5.2.5 ([BGR18, Lemma 1])**.** *Let C be a n-input $(d+1)$-shared d-SNI circuit for a function $f\colon \mathbb{K}^n \to \mathbb{K}$. Then for every uniform and independent input sharings $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \in (\mathbb{K}^{d+1})^n$, an evaluation of C on these inputs produces a sharing $\boldsymbol{y}$ which is uniform and mutually independent of $(\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$.*

Proof. See [BGR18, Appendix A]. □

Informally, Lemma 5.2.5 means that a d-SNI circuit produces a sharing independent of the inputs provided that the inputs are themselves independent. This directly implies that the composition of two d-SNI gadgets is also d-SNI which in turn implies means that it is d-private. This is not always true with the composition of d-private circuits. Additionally, under the right assumptions, the composition of a d-NI gadgets with a d-SNI one is itself d-SNI [BBD+16, Proposition 4]. This allows to construct more complex secure circuits by composing smaller gadgets that are individually proven to be d-NI or d-SNI.

5.2.2 Mask-refreshing gadgets

Mask-refreshing gadgets are d-SNI gadgets on a single masked input that return a single masked output without having any functional impact (both input and output are a masked representation of the same value). The purpose of such a gadget is to break the propagation of dependence because, by Lemma 5.2.5, its output is a independent masking of its input. As a direct corollary of [BBD+16, Proposition 4], a d-NI gadget can be turned into a d-SNI one by composing it with a d-SNI mask-refreshing gadget.

The initial designs for refreshing gadget were using the ISW multiplication gadget by replacing one of the two operands by $(1, 0, \ldots, 0)$, which is a correct additive sharing for 1 at the same order as the masked input. However, doing so has a big impact on performance each time a refreshing gadget is used.

There are three main approaches to reduce the cost induced by the use of refreshing gadgets on a masked implementation:

Lowering the cost of a single refreshing gadget. The first approach is to design less costly refreshing gadgets. The best current results comes from [BBD+18] who prove the SNI security at any order of a "block" refreshing gadget introduced in [BDF+17] (see Example 5.7 for a refreshing gadget at order 2), when iterated enough times. Together with [GPS+18], they also remark that it is possible to make significant improvements in practice at the cost of losing generic proofs, and they give cheaper alternatives verified secure up to order 16. In Chapter 6 and more precisely in Subsection 6.4.2, we discuss the verification of the security of refreshing gadgets and provide a slightly improved 7-SNI refreshing gadget.

Example 5.7: Mask-refreshing gadget at order 2

In Figure 5.5, we show an example of a mask-refreshing gadget at order 2 that is used to refresh the randomness of $\boldsymbol{a}$ and that is 2-SNI.

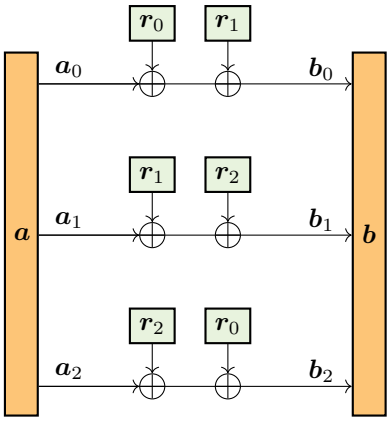


Figure 5.5: Mask-refreshing 2-SNI gadget from [BDF+17].

As expected, the value masked by $\boldsymbol{a}$ and $\boldsymbol{b}$ are the same because each of the three additional random masks (r_0, r_1 and r_2) appears twice and thus will cancel out when unmasking the value of $\boldsymbol{b}$.

Reducing the use of refreshing gadgets. The second approach to improve performance in practice is to use refreshing gadgets only where they are necessary to make the whole circuit secure. A first attempt at a tool that finds the exact places where it is needed to add refreshing gadget was published in 2016 by Barthe *et al.* [BBD+16] and is called `maskComp`. However, in some cases the tool is not able to find an attack but also cannot prove its security without a refreshing gadget. Because it then conservatively adds a refreshing gadget where they might not be needed, the tool is said not to be tight. To fill this gap, Belaïd, Goudarzi and Rivain [BGR18] proposed a tool named TightPROVE that is able to check in a tight manner the security of any circuit composed of only d-SNI multiplication gadgets, d-SNI refresh gadgets and sharewise addition gadgets.

Changing the compositional models. Finally, to deal with the problem of composing the trivial implementations of linear functions in a secure way, a new model called Probe Isolating Non-Interference has been proposed in 2020 by Cassiers and Standaert [CS20].

5.2.3 Probe Isolating Non-Interference

The Probe Isolating Non-Interference (PINI) framework comes from the need of directly composing trivial implementations of linear gadgets. This framework relies on the observation that in the trivial implementation of those gadgets, *e.g.* the gadget for the finite field addition in Example 5.1, an output share $\boldsymbol{c}_i$ only depends on the i-th share of both input. From this observation, Cassiers and Standaert derive the following definition of the PINI security model:

Definition 5.2.6 (d-Probe Isolating Non-Interference)**.** A (u, v)-gadget C for a function over $\mathbb{K}^n$ is *d-probe isolating non-interfering* (d-PINI) if and only if for every set $\mathcal{P}_1$ of d_1 internal probes and $\mathcal{I}_2$ a set of d_2 indices such that $d_1 + d_2 \leq d$, then there exists a set $\mathcal{I}_1$ of at most d_1 indices such that $\mathcal{P}_1 \cup \mathcal{P}_2$ is simulatable by giving access only to input shares of indices in $\mathcal{I}_1 \cup \mathcal{I}_2$, where $\mathcal{P}_2$ is the set of all external probes on output shares of indices in $\mathcal{I}_2$.

It is shown in [CS20] that the composition of only d-PINI gadgets is also d-PINI and that d-PINI implies d-privacy. Additionally, the trivial implementation of linear functions with $d+1$ shares is d-PINI. d-PINI multiplication gadgets can be designed artificially from d-SNI gadgets by using a mask-refreshing gadget systematically on one input of a d-SNI multiplication gadget. However, Cassiers and Standaert proposed a more efficient d-PINI multiplication gadget based on the ISW multiplication and using the same amount of additional random values [CS20, Section IV.B].

5.3 Accounting for physical or micro-architectural effects: the robust probing model

A limitation of the traditional probing model is that it does not capture interactions between intermediate values of a computation made possible by either physical or micro-architectural effects. For instance Gao *et al.* showed that some bitslicing implementation strategies of software masking schemes could exhibit unwanted bit-interactions, thereby violating typical independence assumptions from the probing model and resulting in unwanted leakage [GMP+20]. Similarly, Grégoire *et al.* had noticed that their 4-share vectorised implementation of a masked AES was subject to such an order reduction, without identifying the exact cause [GJR+18].

In the case of hardware implementations, additional violations to the probing model are typically witnessed and some of them are well-identified enough to be formally captured. For one such phenomenon known as *glitches*, a probe at an arithmetic gate (*i.e.* an addition or a multiplication) can leak more to the adversary than its sole output —something that is not taken into account in the basic model. In an effort to remedy this situation, Faust *et al.* recently proposed to extend probing security into a *robust probing model* [FGP+18], able to take several types of hardware defects into account.

Concretely, the robust probing model defines a leakage set $\mathfrak{L}(p)$ of possibly more than one value for every probe p at an arbitrary gate. A probe at an arithmetic gate leaks the union of the leakage sets of its two inputs. One consequence is that if two arithmetic gates are connected together, leakage at the first one also propagates to the second. To stop this propagation, one must then use a memory gate (a register): the leakage set of a memory gate is equal to the singleton of its output value.

Example 5.8: Masked circuit in the robust probing model

We show in Figure 5.6 a circuit implementing a multiplication over $\mathbb{F}_2$ using Ishai Sahai and Wagner scheme [ISW03] at order $d = 1$.

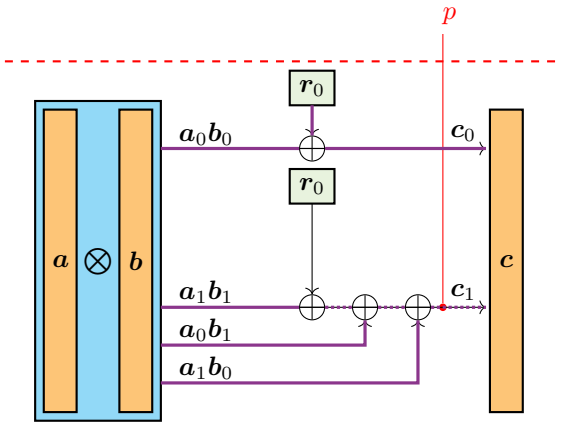


Figure 5.6: Multiplication gadget in presence of hardware glitches.

The propagation of the electric signal at a given moment in time is shown by the purple line. When the output of some intermediate gates may change before the end of the current execution because of propagation delays in the circuit, the line is dotted.

In the previous circuit, a delay occurring at the output of the random gate producing $\boldsymbol{r}_0$ leads to the propagation of a temporary state up to the probe p. This temporary state does not take the value of $\boldsymbol{r}_0$ into account since it has not propagated yet. The probe p can thus read the value $\boldsymbol{a}_1\boldsymbol{b}_1 \oplus \boldsymbol{a}_0\boldsymbol{b}_1 \oplus \boldsymbol{a}_1\boldsymbol{b}_0$, which is an attack against 1-Non-interference.

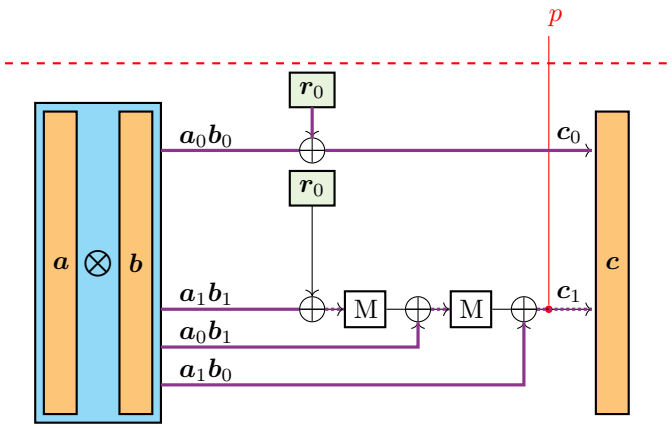


Figure 5.7: Multiplication gadget in presence of glitches with memory gates (M) added

Adding two memory gates at the output of the two XOR gates prevents the temporary signal from propagating all the way to the probe p, which only reads $\boldsymbol{a}_1\boldsymbol{b}_0$ in this example.

5.4 Generic approach to (high-order) masking and its cost

5.4.1 A generic approach to masking using compositional security

Given a function f that needs to be protected against side-channel attacks, one can try to design a gadget implementing it in a masked way. However for complex circuits like for cryptographic primitives and especially for high masking orders, which corresponds in practice to order above 2, it is often not practically possible to conceive the whole circuit at once while formally ensuring the absence of an attack.

Compositional security models are used in this context to allow the following generic and modular approach at masking complex circuits:

— decompose the complex circuit into elementary components, *e.g.* boolean gates for cryptographic primitives having a convenient bit-level description;

— design gadgets that are proven secure in compositional security models for the previously defined elementary components;

— replace every elementary component by its masked equivalent;

— determine where to add mask-refreshing gadgets.

The crucial part for the efficiency of masked implementation is to design, for every different elementary component, masking gadgets that are both efficient and proved secure in the composable models seen previously. We show now that the difficulty of this task is radically different between linear and non-linear components.

5.4.2 Designing secure and efficient generic linear gadgets at any order

Depending on the secret sharing scheme used to mask a value, what is deemed "linear" may differ. Here we look at the case of the most widely used secret sharing scheme: additive secret sharing.

Additions. The simplest linear operation that we want to implement is the addition. In this setting, a naive share-wise masked addition is trivially d-NI, as shown in the following theorem:

Theorem 5.4.1 (Security of the trivial addition gadget)**.** *Let $\boldsymbol{a} = (\boldsymbol{a}_0, \ldots, \boldsymbol{a}_d)$ be a sharing of $a \in \mathbb{K}$ and $\boldsymbol{b} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_d)$ be a sharing of $b \in \mathbb{K}$. Let C be the naive masking gadget producing the shared output $\boldsymbol{c} = (\boldsymbol{c}_0, \ldots, \boldsymbol{c}_d)$ such that $\boldsymbol{c}_i = \boldsymbol{a}_i + \boldsymbol{b}_i, 0 \leq i \leq d$.*

Then, C is a correct and d-Non-Interfering gadget for the addition in $\mathbb{K}$.

Proof. C is correct since:

$$\sum_{i=0}^{d} \boldsymbol{c}_i = \sum_{i=0}^{d} \boldsymbol{a}_i + \boldsymbol{b}_i = \sum_{i=0}^{d} \boldsymbol{a}_i + \sum_{i=0}^{d} \boldsymbol{b}_i = a + b$$

Now to prove its d-NI security, we must prove that any set $\mathcal{P}$ of $t \leq d$ probes can be perfectly simulated. Probes on C are either of the form, $p = \boldsymbol{a}_i$, $p = \boldsymbol{b}_i$ or $p = \boldsymbol{a}_i \oplus \boldsymbol{b}_i$. Since each probe is composed of at most one share of each input, each probe in $\mathcal{P}$ is constant and can be simulated by knowing exactly one probe of each input. Thus, the constant distribution of $\mathcal{P}$ can be perfectly simulated with the knownledge of the $t = \#\mathcal{P}$ corresponding shares of both $\boldsymbol{a}$ and $\boldsymbol{b}$. □

This naive masking scheme has a cost linear in the order d since it takes only $d+1$ additions (one for each share) to compute the masked addition. The case of the addition in $\mathbb{F}_2$ for a boolean additive masking is illustrated in Example 5.9.

Example 5.9: XOR gate masked at order 2

The trivial masked implementation of the addition in $\mathbb{F}_2$ uses 3 XOR gates and consists in simply doing the sharewise XOR between each of the 3 shares of the input to obtain each of the 3 shares of the output. The circuit is shown in Figure 5.8.

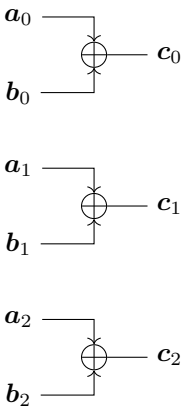


Figure 5.8: 2-NI addition gadget in $\mathbb{F}_2$, *i.e.* masked XOR gate.

Generic linear transformation. More generically, any linear transformation can naively be implemented by applying it independently on each tuple of shares of index i as stated by the following theorem:

Theorem 5.4.2 (Security of the generic gadget for a linear form)**.** *Let f be a linear form from $\mathbb{K}^n$ to $\mathbb{K}$. Let C be a circuit taking as input n additive sharings $\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(n)}$ of $x^{(1)}, \dots, x^{(n)}$ and computing an output $\boldsymbol{c} = (\boldsymbol{c}_0, \dots, \boldsymbol{c}_d)$ such that $\boldsymbol{c}_i = f(\boldsymbol{x}_i^{(1)}, \dots, \boldsymbol{x}_i^{(n)})$.*

Then C is a correct and d-Non-Interfering gadget for f.

Proof. C is correct since by linearity:

$$\sum_{i=0}^{d} \boldsymbol{c}_i = \sum_{i=0}^{d} f(\boldsymbol{x}_i^{(1)}, \dots, \boldsymbol{x}_i^{(n)}) = f(\sum_{i=0}^{d} \boldsymbol{x}_i^{(1)}, \dots, \sum_{i=0}^{d} \boldsymbol{x}_i^{(n)}) = f(x^{(1)}, \dots, x^{(n)})$$

To prove its security, the reasoning is the same as for the proof of Theorem 5.4.1: a probe on the circuit is the function of at most one share of each input; any set of fewer than d probes can be simulated with at most d shares of each input. Thus, C is d-NI. □

This theorem gives us a way to design secure masked gadgets at any order d and for any linear form. Additionally, any linear transformation from $\mathbb{K}^n$ to $\mathbb{K}^m$ can be seen as m linear form and Theorem 5.4.2 can thus be naturally extended to the generic masked gadget of any linear transformation. The cost of these masked gadgets is linear in the order d: the masked gadgets is repeating d times the computation, each time for a different share's index.

This can be seen in a Multi-Party Computation point-of-view where each participant locally computes the linear function on its own shares of the inputs, thus obtaining a share of the result without leaking anything to the other participants.

Generic affine transformation. Addition of a constant, that is computing $c = a + k$ for a constant k in $\mathbb{K}$ on a masked input $\boldsymbol{a}$, is also trivial to implement. In fact, adding a constant k can be done using any correct sharing of k and using the same gadget as for the addition. However, it can actually be done with a constant complexity by taking the following sharing of k: $\boldsymbol{k} = (k, 0, \dots, 0)$. Then, computing $c = a + k$ in a masked way costs only a single addition since for $1 \leq i \leq d$ no addition is required. The case of $\mathbb{K} = \mathbb{F}_2$ and $k = 1$ is interesting since the

circuit implementing $c = a \oplus 1$ in a masked way is a masked gadgets for the NOT boolean gate. This gadget at order 2 is shown in Example 5.10.

Example 5.10: NOT gate masked at order 2

By simply doing the XOR of the first share with 1, one can implement the NOT gate in a masked way with only one XOR gate. The circuit is shown in Figure 5.9.

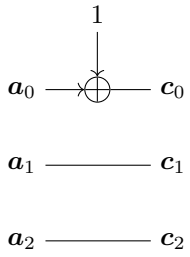


Figure 5.9: 2-NI masked NOT gate

5.4.3 Generic multiplication

As seen in Example 5.3, the design of multiplication gadget in $\mathbb{K}$ is not as trivial as for linear transformations.

Since the proposition of Ishai, Sahai and Wagner in 2003 [ISW03], no better generic design proven to be d-SNI for any order d has been proposed. The ISW multiplication uses $(d+1)^2$ multiplication and $2d(d+1)$ additions in $\mathbb{K}$ but more importantly requires, for each multiplication, the generation of $\frac{d(d+1)}{2}$ fresh random elements in $\mathbb{K}$. The quadratic complexity in fresh masks generation is not negligible since this generation can be an important bottleneck during concrete implementation of masking schemes: in 2017, Journault and Standaert report that between 68% and 92% of the time is spent in randomness generation during their 32-share implementations [JS17].

In 2016, Belaïd *et al.* [BBP+16, Algorithm 3] presented a new design to build, for any d, a multiplication gadget that is proven to be d-NI. This new design is less costly than the ISW multiplication since for even (respectively odd) d it uses $\frac{d^2}{4}+d$ (respectively $\frac{(d^2-1)}{4}+d$) additions, $(d+1)^2$ multiplications and $\frac{d(7d+10)}{4}$ (respectively $\frac{(d+1)(7d+1)}{4}$) fresh random elements in $\mathbb{K}$.

Thus, the overall cost of using masking as a protection against side-channel attacks for a complex function mostly resides in its non-linear components. Some block ciphers have been designed with this constraint in mind, such as PICARO [PRC12], `Zorro` [GGN+13], Fantomas/Robin [GLS+14b], (i)SCREAM [GLS+14a] or more recently `Pyjamask` [GJK+20]. They are all based on having a multiplicative complexity as low as possible to reduce the cost of masking. However, the approach proposed in the design of these block ciphers is not generic and cannot necessarily be used for other primitives. Additionally, drastically reducing the multiplicative complexity can make them less resistant to algebraic attacks, as shown by Leander, Minaud and Rønjom in an attack against (i)SCREAM, Robin and `Zorro` [LMR15]. Thus, being able to construct more efficient gadgets for the multiplication and especially reducing their randomness cost is still an active research topic in order to lower the cost of masking in a generic way.

Additionally, trying to prove the security of a given multiplication gadget by enumerating all possible subsets of probes is exponential in the masking order and can be quickly intractable in practice for high masking order. Thus, in absence of a formal proof, deciding whether a circuit is secure or not is not a trivial task and is the subject of the next chapter.

Chapter 6

Verifying masked circuit in characteristic two

Contents

Overview

Chapter 5 presented compositional security models used in a generic approach to the implementation of masked algorithm. We also discussed the cost of such implementations and saw that it increase quadratically in the masking order. However, the naive method to verify that a gadget is secure in the d-probing model has to go over every candidate set of probes and check that it is not an attack. This approach is quickly intractable at high masking orders since the number of such set of probes increase exponentially in this order.

In this chapter, we extend a matrix-based model introduced in 2017 by Belaïd *et al.* [BBP+17] so that it can be used on masking gadgets over small finite fields (*e.g.* $\mathbb{F}_2$) to prove their security in the compositional models from Chapter 5. We present a verification algorithm and implement it as a new tool which is publicly available[1]. This tool compares favourably against tools from state-of-the-art. In fact it allows to verify masking gadgets three orders of magnitude faster than existing tools and is able to check the security of masking schemes at orders never achieved before. Finally, the verification tool has been of great help to design slightly better multiplication gadgets, mask-refreshing gadgets and to disprove a conjecture from Barthe *et al.* [BDF+17]. The new multiplications gadgets will be used in a concrete implementation presented in Chapter 7.

Most parts of this chapter are originally published as an article at EUROCRYPT 2021 [BK21], with Pierre Karpman as a co-author.

6.1 Proving (strong) non-interference in small finite fields

When designing new masking gadgets for a function f, checking for their correctness is trivial: it can be done by checking that the output is a correct sharing of the result of the function f over the input ; this can be easily done by symbolically summing all the output shares and verifying the result. However, verifying their security is much harder, even in very formal security models as the one introduced in Chapter 5.

In this section, we first recall the matrix model introduced by Belaïd *et al.* [BBP+17] and then extend it to small finite fields.

6.1.1 Matrix model for non-interference revisited

We now recall Theorem 3.5 from Belaïd *et al.* [BBP+17], which defines a powerful matrix model to analyze the (T)NI property of a gadget over a sufficiently large field $\mathbb{K}$ for which all probes are *bilinear*. We then generalise it as Theorem 6.1.7 to work with schemes over any finite field (and $\mathbb{F}_2$ in particular), and to also analyse SNI security in Theorem 6.1.15.

In all of the following, we restrict our interest to gadgets for binary functions[2] $f : \mathbb{K}^2 \to \mathbb{K}$, and the inputs to f (respectively their sharings in a gadget C) will be denoted a and b (respectively $\boldsymbol{a} = (\boldsymbol{a}_0, \ldots, \boldsymbol{a}_{u-1})^t$, $\boldsymbol{b} = (\boldsymbol{b}_0, \ldots, \boldsymbol{b}_{u-1})^t$). We also write the elements of the set $\mathcal{R}$ of R random additional coins as a vector $\boldsymbol{r} = (\boldsymbol{r}_1, \ldots, \boldsymbol{r}_R)^t$

Definition 6.1.1 (Bilinear probe)**.** A probe p on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$ is called *bilinear* iff. it is an affine function in $\boldsymbol{a}_i$, $\boldsymbol{b}_j$, $\boldsymbol{a}_i\boldsymbol{b}_j$, $\boldsymbol{r}_k$; $0 \leq i, j \leq d$, $1 \leq k \leq R$. Equivalently, p is bilinear iff. $\exists\, \boldsymbol{M} \in \mathbb{K}^{(d+1)\times(d+1)}$, $\boldsymbol{\mu}, \boldsymbol{\nu} \in \mathbb{K}^{(d+1)}$, $\boldsymbol{\sigma} \in \mathbb{K}^R$ and $\tau \in \mathbb{K}$ such that $p = \boldsymbol{a}^t\boldsymbol{M}\boldsymbol{b} + \boldsymbol{a}^t\boldsymbol{\mu} + \boldsymbol{b}^t\boldsymbol{\nu} + \boldsymbol{r}^t\boldsymbol{\sigma} + \tau$.

By considering only such bilinear probes, we are implicitly restricting our analysis to gadgets using only additions and multiplications gates. Also, the multiplicative depth of those gadgets must not be more than one. While this may seems very restrictive, composing such gadgets is made possible thanks to the d-(S)NI properties defined in Subsection 5.2.1, thus allowing to build more complex circuits.

[1] https://github.com/NicsTr/binary_masking

[2] Results for unary functions can then easily be obtained by *e.g.* fixing one input.

Definition 6.1.2 (Functional dependence)**.** An expression $\mathrm{E}(x_1, \ldots, x_n)$ is said to *functionally depend* on x_n iff. $\exists\, c_1, \ldots, c_{n-1}$ such that the mapping $x_n \mapsto \mathrm{E}(c_1, \ldots, c_{n-1}, x_n)$ is not constant.

We now introduce the following condition which plays a central role in the security analysis of a gadget in the matrix model.

Condition 6.1.3 ([BBP+17, Condition 3.2])**.** *A set of bilinear probes $\mathcal{P} = \{p_1, \ldots, p_\ell\}$ on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$ satisfies Condition 6.1.3 iff. $\exists\, \boldsymbol{\lambda} \in \mathbb{K}^\ell$, $\boldsymbol{M} \in \mathbb{K}^{(d+1)\times(d+1)}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu} \in \mathbb{K}^{d+1}$, and $\tau \in \mathbb{K}$ such that $\sum_{i=1}^{\ell} \boldsymbol{\lambda}_i p_i = \boldsymbol{a}^t \boldsymbol{M} \boldsymbol{b} + \boldsymbol{a}^t \boldsymbol{\mu} + \boldsymbol{b}^t \boldsymbol{\nu} + \tau$ and all the rows of the block matrix $\begin{pmatrix} \boldsymbol{M} & \boldsymbol{\mu} \end{pmatrix}$ or all the columns of the block matrix $\begin{pmatrix} \boldsymbol{M} \\ \boldsymbol{\nu}^t \end{pmatrix}$ are non-zero.*

In other words, this condition states that there exists a linear combination of probes of $\mathcal{P}$ that does not functionally depend on any random scalar and that functionally depends on either all of the shares for a or all of the shares for b. Thus, $\mathcal{P}$ cannot be perfectly simulated using only d shares of a and d shares of b, effectively proving that the C is not d-NI. In such a case, we say that $\mathcal{P}$ is an attack against the d-NI property of C.

We are now ready to state the following theorem.

Theorem 6.1.4 ([BBP+17, Theorem 3.5])**.** *Let $\mathcal{P}$ be a set of bilinear probes on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$. If $\mathcal{P}$ satisfies Condition 6.1.3, then it is not d-simulatable. Furthermore, if $\mathcal{P}$ is not d-simulatable and $\#\mathbb{K} > d+1$, then it satisfies Condition 6.1.3.*

Example 6.1: Condition 6.1.3 $\implies$ d-NI attack

We reuse the circuit and probes defined in Example 5.1. The probe p_1 can be written as $p_1 = \boldsymbol{a}_0 \oplus \boldsymbol{b}_0 \oplus \boldsymbol{a}_1 = \boldsymbol{a}^t \boldsymbol{M} \boldsymbol{b} + \boldsymbol{a}^t \boldsymbol{\mu} + \boldsymbol{b}^t \boldsymbol{\nu}$, with $\boldsymbol{M} = \mathbf{0}_{2\times 2}$, $\boldsymbol{\mu} = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$ and $\boldsymbol{\nu} = \begin{pmatrix} 1 \\ 0 \end{pmatrix}$. Since the block matrix $\begin{pmatrix} \boldsymbol{M} & \boldsymbol{\mu} \end{pmatrix}$ has no zero row, $\{p_0\}$ satisfies Condition 6.1.3. This means that $\{p_0\}$ is not 1-simulatable and is thus an attack against the 1-NI property of the circuit.

The previous theorem immediately leads to the following corollary.

Corollary 6.1.5 ([BBP+17, Corollary 3.7])**.** *Let C be a $(d+1, v)$-gadget for a function $f : \mathbb{K}^2 \to \mathbb{K}$ for which all probes are bilinear. If C is d-NI, then there is no set of d probes on C satisfying Condition 6.1.3. Furthermore, if $\#\mathbb{K} > d+1$ and there is no set of d probes on C satisfying Condition 6.1.3, then C is d-NI.*

This corollary is more useful than the theorem in practice because, under the restriction that $\#\mathbb{K} > d+1$, it can be directly applied as an algorithm to determine if a given gadget is d-NI or not.

For the masking schemes of CRYPTO 2017 [BBP+17] the restriction $\#\mathbb{K} > d+1$ is never an issue, as they are defined over large fields; however, this condition means that one cannot directly apply Corollary 6.1.5 to prove the security of a scheme over a small field such as $\mathbb{F}_2$.

Example 6.2: Not d-simulatable $\not\implies$ Condition 6.1.3 (in a small field)

We show a counter-example to demonstrate the limitation of Condition 6.1.3 in a small field such as $\mathbb{F}_2$.

Let C be a circuit for a function $f\colon \mathbb{F}_2 \to \mathbb{F}_2$ with one of its input being $\boldsymbol{a} = (\boldsymbol{a}_0, \boldsymbol{a}_1, \boldsymbol{a}_2)$. Let $\mathcal{P} = \{p_1, p_2\}$ be a set of probes on this circuit with $p_1 = \boldsymbol{a}_0 \oplus \boldsymbol{a}_2$ and $p_2 = \boldsymbol{a}_0 \oplus \boldsymbol{a}_1$. We can write $p_1 = \boldsymbol{a}^t \boldsymbol{\mu}_1$ and $p_2 = \boldsymbol{a}^t \boldsymbol{\mu}_2$ with $\boldsymbol{\mu}_1 = \begin{pmatrix} 1 \\ 0 \\ 1 \end{pmatrix}$ and $\boldsymbol{\mu}_2 = \begin{pmatrix} 1 \\ 1 \\ 0 \end{pmatrix}$.

$\mathcal{P}$ cannot be simulated with only the knowledge of two shares of $\boldsymbol{a}$, thus is not 2-simulatable. However, all four linear combinations $\sum \boldsymbol{\lambda}_i p_i = \sum \boldsymbol{\lambda}_i \boldsymbol{a}^t \boldsymbol{\mu}_i$ for $\boldsymbol{\lambda} \in \mathbb{F}_2^2$ are such that at least one row is zero. Thus, even if the set of probes is not 2-simulatable, it does not satisfies Condition 6.1.3.

We now sketch a proof of the second statement of Theorem 6.1.4 as a preparation to extending it to any field.

Proof of Theorem 6.1.4 right to left, sketch. Let $\mathcal{P} = \{p_1, \dots, p_\ell\}$ be a set of bilinear probes that is not d-simulatable. We call $\boldsymbol{R}$ the block matrix $\begin{pmatrix}\boldsymbol{\sigma}_1 & \cdots & \boldsymbol{\sigma}_\ell\end{pmatrix}$, where $\boldsymbol{\sigma}_i$ denotes as in Definition 6.1.1 the vector of random scalars on which p_i depends. Up to a permutation of its rows and columns[3], the reduced column echelon form $\boldsymbol{R}'$ of $\boldsymbol{R}$ is of the shape $\begin{pmatrix}\boldsymbol{I}_t & \boldsymbol{0}_{t,\ell-t}\\ \boldsymbol{N} & \boldsymbol{0}_t\end{pmatrix}$, where $t \leq \ell$ is the rank of $\boldsymbol{R}$ and $\boldsymbol{N}$ is arbitrary. If we now consider the symbolic matrix $\boldsymbol{P} = \begin{pmatrix}p_1 & \cdots & p_\ell\end{pmatrix}^t$ and multiply it by the change-of-basis matrix from $\boldsymbol{R}$ to $\boldsymbol{R}'$, we obtain the matrix $\boldsymbol{P}' = \begin{pmatrix}\boldsymbol{P}'_r & \boldsymbol{P}'_d\end{pmatrix}$ where $\boldsymbol{P}'_r$ represents t linear combinations $\{p'_1, \dots, p'_t\}$ of probes that each depend on at least one random scalar which does not appear across any of the other linear combinations, and $\boldsymbol{P}'_d$ represents $\ell - t$ linearly independent linear combinations $\mathcal{P}' = \{p'_{t+1}, \dots, p'_\ell\}$ of probes that do not depend on any random scalar. All of the $\{p'_1, \dots, p'_t\}$ can then be simulated by independent uniform distributions without requiring the knowledge of any share, and as $\mathcal{P}$ is not d-simulatable, $\mathcal{P}'$ cannot be d-simulatable either. W.l.o.g., this means that for every share $\boldsymbol{a}_i$, there is at least one linear combination of probes in $\mathcal{P}'$ that depends on it. In other words, the matrix $\boldsymbol{D} = \begin{pmatrix}\boldsymbol{M}'_{t+1} & \boldsymbol{\mu}_{t+1} & \cdots \boldsymbol{M}'_\ell & \boldsymbol{\mu}_\ell\end{pmatrix}$ that records this dependence has no zero row. We now finally want to show that there is a linear combination $\begin{pmatrix}\boldsymbol{\lambda}_{t+1} & \cdots & \boldsymbol{\lambda}_\ell\end{pmatrix}^t$ of elements of $\mathcal{P}'$ that satisfies Condition 6.1.3. This can be done by showing that $\exists \boldsymbol{\Lambda} = \begin{pmatrix}\boldsymbol{\Lambda}_{t+1} & \cdots & \boldsymbol{\Lambda}_\ell\end{pmatrix}^t$ such that $\boldsymbol{D}\boldsymbol{\Lambda}$ has no zero row, where the $\boldsymbol{\Lambda}_i$'s are the $(d+2)\times(d+2)$ scalar matrices of multiplication by the $\boldsymbol{\lambda}_i$'s. By the Schwartz-Zippel-DeMillo-Lipton lemma this is always the case as soon as $\#\mathbb{K} > d+1$ [Sch80], and this last step is the only one that depends on $\mathbb{K}$. □

We now wish to extend Theorem 6.1.4 and its corollary to any finite field $\mathbb{K}$. We do this using the TNI notion rather than NI, and so first state an appropriate straightforward adaptation of Condition 6.1.3:

Condition 6.1.6. *A set of bilinear probes $\mathcal{P} = \{p_1, \dots, p_\ell\}$ on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$ satisfies Condition 6.1.6 iff. $\exists \boldsymbol{\lambda} \in \mathbb{K}^\ell$, $\boldsymbol{M} \in \mathbb{K}^{(d+1)\times(d+1)}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu} \in \mathbb{K}^{d+1}$, and $\tau \in \mathbb{K}$ such that $\sum_{i=1}^{\ell} \boldsymbol{\lambda}_i p_i = \boldsymbol{a}^t\boldsymbol{M}\boldsymbol{b} + \boldsymbol{a}^t\boldsymbol{\mu} + \boldsymbol{b}^t\boldsymbol{\nu} + \tau$ and the block matrix $\begin{pmatrix}\boldsymbol{M} & \boldsymbol{\mu}\end{pmatrix}$ has at least $\ell+1$ non-zero rows or the block matrix $\begin{pmatrix}\boldsymbol{M}\\ \boldsymbol{\nu}^t\end{pmatrix}$ has at least $\ell+1$ non-zero columns.*

In other words, Condition 6.1.6 states that the expression $\sum_{i=1}^{\ell} \boldsymbol{\lambda}_i p_i$, which involves ℓ probes, functionally depends on no random scalar and on at least $\ell+1$ shares of a or $\ell+1$ shares of b, and hence is an l-TNI attack. We will then show the following:

Theorem 6.1.7. *Let $\mathcal{P}$ be a set of at most d bilinear probes on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$. If $\mathcal{P}$, is not d-simulatable then $\exists \mathcal{P}' \subseteq \mathcal{P}$ such that $\mathcal{P}'$ satisfies Condition 6.1.6.*

Corollary 6.1.8 (Corollary of Theorems 6.1.4 and 6.1.7)**.** *Let C be a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$ for which all probes are bilinear. If C is d-NI, then there is no set of d probes on C satisfying Condition 6.1.3. Furthermore, if there is no set of $t \leq d$ probes on C satisfying Condition 6.1.6, then C is d-NI.*[4]

As for Corollary 6.1.5, this corollary will be particularly convenient to design an algorithm proving the (non)d-NI property of a gadget given its probe.

The proof of Theorem 6.1.7 essentially relies on the following lemmas, conveniently formulated with linear codes:[5]

[3] This permutation corresponds to renaming/reordering the random scalar $\boldsymbol{\sigma}_i$ and probes p_i

[4] As Condition 6.1.6 directly implies an attack, one could also formulate this corollary solely in terms of this condition.

[5] Recall that an $[n, k]$ linear code over a field $\mathbb{K}$ is a k-dimensional linear subspace of $\mathbb{K}^n$.

Lemma 6.1.9. *Let $\mathcal{C}_1$ (respectively $\mathcal{C}_2$) be an $[n_1, k]$ (respectively $[n_2, k]$, $n_2 > n_1$) linear code over a finite field $\mathbb{K}$. Let $\boldsymbol{G}_1 \in \mathbb{K}^{k\times n_1}$ and $\boldsymbol{G}_2 \in \mathbb{K}^{k\times n_2}$ be two generator matrices for $\mathcal{C}_1$ and $\mathcal{C}_2$ that have no zero column. Then the code $\mathcal{C}_{1,2}$ generated by $\boldsymbol{G}_{1,2} := \begin{pmatrix}\boldsymbol{G}_1 & \boldsymbol{G}_2\end{pmatrix}$ has the following property: $\exists\, \boldsymbol{c} \in \mathcal{C}_{1,2}$ such that $\mathrm{wt}_1(\boldsymbol{c}) < \mathrm{wt}_2(\boldsymbol{c})$, where $\mathrm{wt}_1(\cdot)$ (respectively $\mathrm{wt}_2(\cdot)$) denotes the Hamming weight function restricted to the first n_1 (respectively last n_2) coordinates of $\mathcal{C}_{1,2}$.*

One may remark that if $\#\mathbb{K}$ is sufficiently large with respect to the parameters of the codes, then by the Schwartz-Zippel-DeMillo-Lipton lemma there exists a word in $\mathcal{C}_{1,2}$ of maximal wt_2 weight, and the conclusion immediately follows; yet this argument does not hold over any field.

We first give the following two definitions that will be used in the proof of Lemma 6.1.9:

Definition 6.1.10 (Shortening of a linear code)**.** Let $\mathcal{C}$ be an $[n, k]$ linear code over $\mathbb{K}$ generated by $\boldsymbol{G} \in \mathbb{K}^{k\times n}$, the *shortened code* $\mathcal{C}'$ with respect to coordinate $i \in [\![1, n]\!]$ is the subcode made of all codewords of $\mathcal{C}$ that are zero at coordinate i, with this coordinate then being deleted.

Definition 6.1.11 (Isolated coordinate)**.** Let $\boldsymbol{M} \in \mathbb{K}^{m\times n}$, a coordinate $i \in [\![1, n]\!]$ is called *isolated* for the row $\boldsymbol{M}_j$ of $\boldsymbol{M}$, $j \in [\![1, m]\!]$, iff. $\boldsymbol{M}_{j,i} \neq 0$ and $\forall j' \neq j \in [\![1, m]\!]$, $\boldsymbol{M}_{j',i} = 0$.

In order to prove Lemma 6.1.9 we will define a procedure that aims to reduce the dimension of the starting code by shortening it in a specific way:

Procedure 6.1.12. *We reuse the notation of the statement of Lemma 6.1.9. This procedure is* applied on a row of $\boldsymbol{G}_{1,2}$ *by doing the following: denote $\mathcal{I}_1$ (respectively $\mathcal{I}_2$) the (possibly empty) set of isolated coordinates for this row on its first n_1 (respectively last n_2) columns; then if $\#\mathcal{I}_1 \geq \#\mathcal{I}_2$, shorten $\mathcal{C}_{1,2}$ with respect to all the coordinates in $\mathcal{I}_1 \cup \mathcal{I}_2$.*

Practically, shortening the code with respect to one or more isolated coordinates means deleting from $\boldsymbol{G}_{1,2}$ the row being processed and all the columns in $\mathcal{I}_1 \cup \mathcal{I}_2$. The row being processed is the one and only one having to be removed from the generator matrix because: 1) all other rows have a zero in the given coordinates, thus all codewords generated by them are zero in these coordinates; 2) any linear combination of rows that includes the row being processed with a non-zero coefficient will result in a non-zero value in the isolated coordinated.

This results in a code $\mathcal{C}'_{1,2}$ generated by $\begin{pmatrix}\boldsymbol{G}'_1 & \boldsymbol{G}'_2\end{pmatrix}$ where $\boldsymbol{G}'_1 \in \mathbb{K}^{(k-1)\times n'_1}$ (respectively $\boldsymbol{G}'_2 \in \mathbb{K}^{(k-1)\times n'_2}$) is a submatrix of $\boldsymbol{G}_1$ (respectively $\boldsymbol{G}_2$) and $n'_1 < n_1$, $n'_2 \leq n_2$, $n'_1 < n'_2$, and none of the columns of $\boldsymbol{G}'_{1,2}$ is zero. One may also remark that since $\boldsymbol{G}'_1$ is of rank $k-1$, we have $k - 1 \leq n'_1$.

Example 6.3: Procedure 6.1.12

We consider the following matrix that satisfies the premise of Lemma 6.1.9:

$$\boldsymbol{G}_{1,2} = \left(\begin{array}{cccccc|cccccc}
1 & & & 1 & 0 & 1 & 1 & & & 1 & 0 & 0 & 1\\
 & 1 & & 1 & 1 & 1 & & 1 & & 1 & 0 & 0 & 0\\
 & & 1 & 0 & 0 & 1 & & & 1 & 0 & 1 & 0 & 0\\
 & & & 1 & 1 & 0 & 0 & & & 1 & 0 & 0 & 1 & 0
\end{array}\right)$$

We apply Procedure 6.1.12 to its second row. To do so, we have to find the isolated coordinates. Here, $\mathcal{I}_1 = \{2, 6\}$ and $\mathcal{I}_2 = \{9\}$:

$$\boldsymbol{G}_{1,2} = \left(\begin{array}{cccccc|cccccc}
1 & & & 1 & 0 & 1 & 1 & & & 1 & 0 & 0 & 1\\
 & 1 & & 1 & 1 & 1 & & 1 & & 1 & 1 & 1 & 1\\
 & & 1 & 0 & 0 & 1 & & & 1 & 0 & 1 & 0 & 0\\
 & & & 1 & 1 & 0 & 0 & & & 1 & 0 & 0 & 1 & 0
\end{array}\right)$$

Since we have $\#\mathcal{I}_1 \geq \#\mathcal{I}_2$, applying Procedure 6.1.12 on the second row leads to the shortening of the code along the columns $2, 6$ and 9. This results in the following generator matrix

of the shortened code, with the isolated coordinates of each row being highlighted:

$$\boldsymbol{G}'_{1,2} = \left(\begin{array}{ccccc|ccccc} 1 & & & 1 & 1 & 1 & & & 1 & 1 \\ & 1 & & 0 & 1 & & 1 & & 0 & 0 \\ & & 1 & 1 & 0 & & & 1 & 0 & 0 \end{array}\right)$$

At this point and for each row, the number of isolated coordinates for the left part is strictly lower than for the right part. Thus, there is no row for which applying Procedure 6.1.12 results in a shortening of the code.

We are now ready to prove Lemma 6.1.9.

Proof of Lemma 6.1.9. We prove this lemma by induction using Procedure 6.1.12.

In a first step one applies Procedure 6.1.12 to every row of $\boldsymbol{G}_{1,2}$ one at a time and repeats this process again until either there is no row for which applying the procedure results in a shortening, or the dimension of the shortened code reaches 1.

In the latter case, this means that the only non-zero codeword in $\boldsymbol{G}'_{1,2} \in \mathbb{K}^{1\times(n'_1+n'_2)}$ is of full weight $n'_1+n'_2$ with $n'_1 < n'_2$ (since $\boldsymbol{G}'_{1,2}$ only has a single row and none of its columns is zero). This induces a codeword c of $\mathcal{C}$ such that $\mathrm{wt}_1(c) = n'_1$ and $\mathrm{wt}_2(c) = n'_2$, so we are done.

In the former case, one is left with a matrix $\boldsymbol{G}'_{1,2} \in \mathbb{K}^{k'\times(n'_1+n'_2)}$, $k' > 1$. One then computes the reduced row echelon form of $\boldsymbol{G}'_{1,2}$ (this does not introduce any zero column since the elementary row operations are invertible) and again iteratively applies Procedure 6.1.12 on the resulting matrix as done in the first step. Now either the application of Procedure 6.1.12 leads to a shortened code of dimension 1 and then we are done as above, or we are left with a matrix $\boldsymbol{G}''_{1,2} \in \mathbb{K}^{k''\times(n''_1+n''_2)}$ which can be of two forms:

1. $k'' \geq n''_1$. Up to permutation of its columns, $\boldsymbol{G}''_{1,2}$ can be written as:

$$\left(\begin{array}{c|ccc} \boldsymbol{I}_{n''_1} & \boldsymbol{I}_{n''_1} & \boldsymbol{I}_{n''_1} & * \\ \boldsymbol{0}_{(k''-n''_1)\times n''_1} & * & * & * \end{array}\right),$$

   where the $*$ are arbitrary and the bottom $(k''-n''_1)\times(n''_1+n''_2)$ block is possibly non-existent. The left $k''\times n''_1$ block is justified from $\boldsymbol{G}''_{1,2}$ being in reduced row echelon form and having none of its column equal to zero. The right $k''\times n''_2$ block is justified from the fact that every non-zero row of the left block has exactly one isolated coordinate; since no simplification can be done anymore to $\boldsymbol{G}''_{1,2}$ by applying Procedure 6.1.12, this means that those rows have at least two isolated coordinates on the right block. This is enough to conclude on the existence of a codeword of $\mathcal{C}$ satisfying the desired property.

2. $k'' < n''_1$. Up to a permutation of its columns, the rank-k'' matrix $\boldsymbol{G}''_{1,2}$ can be written as:

$$\left(\boldsymbol{I}_{k''} \quad *_L \;\middle|\; \boldsymbol{I}_{k''} \quad \boldsymbol{I}_{k''} \quad *_R\right),$$

   and it has no zero column. One then applies Lemma 6.1.9 inductively on the code generated by the submatrix $\boldsymbol{G}'''_{1,2} := \left(*_L \;\middle|\; \boldsymbol{I}_{k''} \quad *_R\right)$ which is of strictly smaller length. Let $c''' = \boldsymbol{\lambda}\boldsymbol{G}'''_{1,2}$ be a codeword of this latter code that satisfies the desired property, then $\boldsymbol{\lambda}\boldsymbol{G}''_{1,2}$ also satisfies it for $\mathcal{C}_{1,2}$, which concludes the proof.

□

Example 6.4: Application of Lemma 6.1.9

We study the case of the same matrix $\boldsymbol{G}'_{1,2}$ defined in Example 6.3. It is already in row reduced echelon form and cannot be shortened any more using Procedure 6.1.12. Since the number of rows $k'' = 3$ of $\boldsymbol{G}'_{1,2}$ is stricly less than the number of columns $n''_1 = 5$ of the first code, we follow the proof of Lemma 6.1.9 by studying the code of strictly smaller length generated by:

$$\boldsymbol{G}'''_{1,2} = \left(\begin{array}{cc|cccc} 1 & 1 & 1 & & & 1 \\ 0 & 1 & & 1 & & 0 \\ 1 & 0 & & & 1 & 0 \end{array}\right)$$

Applying Procedure 6.1.12 cannot lead to a shortening of the code since it was not the case for $\boldsymbol{G}''_{1,2}$. To continue, $\boldsymbol{G}'''_{1,2}$ is put into its row reduced echelon form:

$$\tilde{\boldsymbol{G}}_{1,2} = \left(\begin{array}{cc|cccc} 1 & 0 & 1 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

We can remark that the last row of the matrix is already a codeword such that

$$\mathrm{wt}_1(\begin{pmatrix} 0 & 0 & | & 1 & 1 & 1 & 1 \end{pmatrix}) = 0 < \mathrm{wt}_2(\begin{pmatrix} 0 & 0 & | & 1 & 1 & 1 & 1 \end{pmatrix}) = 4$$

which is what we want to find. The first row is also a codeword that satisfies the same property. Nonetheless, we will continue to follow the proof's algorithm by applying Procedure 6.1.12 to the first row, which leads to a shortened code:

$$\tilde{\boldsymbol{G}}'_{1,2} = \left(\begin{array}{c|cccc} 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 1 & 1 \end{array}\right)$$

And then, on the first row again:

$$\tilde{\boldsymbol{G}}'_{1,2} = \begin{pmatrix} | & 1 & 1 & 1 & 1 \end{pmatrix}$$

This new code is of dimension 1 and so the algorithm stops. The only codeword left satisfies the property:

$$\mathrm{wt}_1(\begin{pmatrix} | & 1 & 1 & 1 & 1 \end{pmatrix}) = 0 < \mathrm{wt}_2(\begin{pmatrix} | & 1 & 1 & 1 & 1 \end{pmatrix}) = 4$$

Up to a permutation of the columns, $\tilde{\boldsymbol{G}}'_{1,2}$ is the generator matrix of a shortening of the code described by the initial $\boldsymbol{G}_{1,2}$. Thus a codeword in $\tilde{\boldsymbol{G}}'_{1,2}$ can be zero-extended to obtain a codeword in $\boldsymbol{G}_{1,2}$, which means that there exists a codeword c in $\boldsymbol{G}_{1,2}$ such that $\mathrm{wt}_1(c) < \mathrm{wt}_2(c)$.

Shortening the code with respect to some coordinates during the application of Procedure 6.1.12 leads to a subcode where codewords have zeroes for those coordinates. Thus by keeping track of those coordinates and of the operations used during the row reduction steps, one can retrieve a codeword satisfying the desired property. In this example, the resulting codeword is: $\begin{pmatrix} 1 & 0 & 1 & 1 & 0 & 0 & 0 & | & 1 & 0 & 1 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$.

Now we are almost ready to prove Theorem 6.1.7, not by using Lemma 6.1.9 but using an extension of it:

Lemma 6.1.13. *The statement of Lemma 6.1.9 still holds if $\mathbb{K}$ is replaced by a matrix ring $\mathbb{K}'^{d\times d}$ and if $\boldsymbol{G}_1$ is defined over the subfield of the scalar matrices of $\mathbb{K}'^{d\times d}$.*

Proof of Lemma 6.1.13. The proof simply consists in remarking that all the steps of the proof of Lemma 6.1.9 can be carried out in the modified setting of Lemma 6.1.13. Mainly:

- Definitions 6.1.10 and 6.1.11 and Procedure 6.1.12 naturally generalise to matrices over rings, and the application of Procedure 6.1.12 is unchanged.
- Recall that by induction the left $k' \times n'_1$ submatrix is always of full rank k', which is also the rank of $\boldsymbol{G}'_{1,2}$. Since $\boldsymbol{G}_1$ is defined over scalar matrices, the row reduction can be computed as if over a field.

□

The proof of Theorem 6.1.7 then follows.

Proof of Theorem 6.1.7. We start similarly from the proof of Theorem 6.1.4, and use the same notation: let $\mathcal{P}'$ be a set of $\ell - t$ linearly-independent linear combinations of probes of $\mathcal{P}$ that do not depend on any random scalar, and let $\boldsymbol{D} = \begin{pmatrix} \boldsymbol{M}'_{t+1} & \boldsymbol{\mu}_{t+1} & \cdots \boldsymbol{M}'_{\ell} & \boldsymbol{\mu}_{\ell} \end{pmatrix}$ be the matrix that records the dependence of these probes on every share $\boldsymbol{a}_i$.We will show that $\exists \mathcal{P}'' \subseteq \mathcal{P}$ that satisfies Condition 6.1.6. To do this, we introduce two new indicator matrices:

— Let $\boldsymbol{\Pi} \in \mathbb{K}^{(d+2)\times(d+2)^{(\ell-t)\times\ell}}$ be such that for every $p' \in \mathcal{P}'$ it records in its rows its dependence on the probes of $\mathcal{P}$ as scalar matrices;[6] that is, $\boldsymbol{\Pi}$ is such that $p'_i = \sum_{j=1}^{\ell} \boldsymbol{\pi}_{i,j} p_j$ where $\boldsymbol{\pi}_{i,j}$ is the scalar on the diagonal of the scalar matrix $\boldsymbol{\Pi}_{i,j}$. Without loss of generality, we may assume that every probe of $\mathcal{P}$ appears at least once in a linear combination of $\mathcal{P}'$, otherwise it is simply discarded, so $\boldsymbol{\Pi}$ has no zero column.

— Let $\boldsymbol{\Delta} \in \mathbb{K}^{(d+2)\times(d+2)^{(\ell-t)\times(d+1)}}$ be the matrix that for every $p' \in \mathcal{P}'$ records in its rows its dependence on the shares $\boldsymbol{a}_i$s; that is if the bilinear probe p'_i can be written as $p'_i = \boldsymbol{a}^t\boldsymbol{M}'\boldsymbol{b} + \boldsymbol{a}^t\boldsymbol{\mu}' + \boldsymbol{b}^t\boldsymbol{\nu}' + \tau'$, then $\boldsymbol{\Delta}_{i,j}$ is set to the diagonal matrix of the j^{th} row of $\begin{pmatrix} \boldsymbol{M}' & \boldsymbol{\mu}' \end{pmatrix}$.[7] Note that since by assumption $\boldsymbol{D}$ has no zero row, $\boldsymbol{\Delta}$ has no zero *column*.

Now we invoke Lemma 6.1.13 with $\boldsymbol{\Pi}$ as $\boldsymbol{G}_1$ and $\boldsymbol{\Delta}$ as $\boldsymbol{G}_2$ the generator matrices for the code $\mathcal{C}_{1,2}$. Let $\boldsymbol{c} \in \mathcal{C}_{1,2}$ be a codeword that satisfies $\mathrm{wt}_1(\boldsymbol{c}) < \mathrm{wt}_2(\boldsymbol{c})$; this translates to a linear combination of $\ell'' := \mathrm{wt}_1(\boldsymbol{c})$ probes of $\mathcal{P}'' \subseteq \mathcal{P}$ that (as linear combinations of elements of $\mathcal{P}'$) does not depend on any randomness and such that the associated matrix $\begin{pmatrix} \boldsymbol{M}'' & \boldsymbol{\mu}'' \end{pmatrix}$ has $\mathrm{wt}_2(\boldsymbol{c}) \geq \ell'' + 1$ non-zero rows (by applying the inverse transformation from $\boldsymbol{\Delta}$ to $\boldsymbol{D}$), hence $\mathcal{P}''$ satisfies Condition 6.1.6. □

Finally, the proof of Corollary 6.1.8 is immediate from Theorems 6.1.4 and 6.1.7.

6.1.2 Matrix model for strong non-interference revisited

We now wish to adapt the approach of Theorems 6.1.4 and 6.1.7 to be able to prove that a scheme is SNI. This is in fact quite straightforward, and it mostly consists in defining a suitable variant of Condition 6.1.6 and in applying Lemma 6.1.13 to well-chosen matrices, to show again that there is a subset of probes that satisfies the condition whenever there is an attack.

Condition 6.1.14. *A set of $\ell = \ell_1 + \ell_2$ bilinear probes $\mathcal{P} = \{p_1, \ldots, p_\ell\}$ on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$, of which ℓ_1 are internal, satisfies Condition 6.1.14 iff. $\exists \boldsymbol{\lambda} \in \mathbb{K}^{\ell}$, $\boldsymbol{M} \in \mathbb{K}^{(d+1)\times(d+1)}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu} \in \mathbb{K}^{d+1}$, and $\tau \in \mathbb{K}$ such that $\sum_{i=1}^{\ell} \boldsymbol{\lambda}_i p_i = \boldsymbol{a}^t\boldsymbol{M}\boldsymbol{b} + \boldsymbol{a}^t\boldsymbol{\mu} + \boldsymbol{b}^t\boldsymbol{\nu} + \tau$ and the block matrix $\begin{pmatrix} \boldsymbol{M} & \boldsymbol{\mu} \end{pmatrix}$ (respectively the block matrix $\begin{pmatrix} \boldsymbol{M} \\ \boldsymbol{\nu}^t \end{pmatrix}$) has at least $\ell_1 + 1$ non-zero rows (respectively columns).*

Theorem 6.1.15. *Let $\mathcal{P}$ be a set of at most d bilinear probes on a $(d+1, v)$-gadget C for a function $f : \mathbb{K}^2 \to \mathbb{K}$, of which ℓ_1 are internal. If $\mathcal{P}$ is not ℓ_1-simulatable then $\exists \mathcal{P}' \subseteq \mathcal{P}$ such that $\mathcal{P}'$ satisfies Condition 6.1.14.*

Proof. We reuse the notation of Theorems 6.1.4 and 6.1.7. The proof is essentially the same as the one of Theorem 6.1.7, except that we only account for internal probes in $\boldsymbol{\Pi}$. Let $\mathcal{P}'$ be a set of $\ell - t$ linearly independent linear combinations of probes of $\mathcal{P}$ that do not depend on any random scalar, and let $\boldsymbol{D} = \begin{pmatrix} \boldsymbol{M}'_{t+1} & \boldsymbol{\mu}_{t+1} & \cdots \boldsymbol{M}'_{\ell} & \boldsymbol{\mu}_{\ell} \end{pmatrix}$ be the matrix that records the dependence of

[6] This use of scalar matrices is only so that $\boldsymbol{\Pi}$ is defined on the same base structure as $\boldsymbol{\Delta}$ below. As an example, taking $\ell = d = 2$ and considering two probes in $\mathcal{P}'$ as $p'_1 = p_1 + p_2$; $p'_2 = p_2$, then $\boldsymbol{\Pi} = \begin{pmatrix} \boldsymbol{I}_4 & \boldsymbol{I}_4 \\ \boldsymbol{0}_4 & \boldsymbol{I}_4 \end{pmatrix}$.

[7] This use of diagonal matrices allows to keep track of (the lack of) simplifications when combining several probes; for instance, if two probes depend on the same $\boldsymbol{a}_i$ as $\boldsymbol{a}_i\boldsymbol{b}_j$ and $\boldsymbol{a}_i\boldsymbol{b}_{j'}$ with $j \neq j'$, then the sum of those probes still depends on $\boldsymbol{a}_i$. Continuing the previous example and taking $p'_1 = \boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{a}_0\boldsymbol{b}_1 + \boldsymbol{a}_1\boldsymbol{b}_2 + \boldsymbol{a}_2$, then the first row of $\boldsymbol{\Delta}$ (whose entries are 4×4 matrices) is $\begin{pmatrix} 1 & 0 & 0 & \\ & 1 & 0 & 0 & \\ & & 0 & 1 & 0 & \\ & & & 0 & 0 & 1 \end{pmatrix}$.

these probes on every share $\boldsymbol{a}_i$. From the assumption that $\mathcal{P}$ is not ℓ_1-simulatable, we have that without loss of generality, $\boldsymbol{D}$ has at least ℓ_1+1 non-zero rows. We will show that $\exists\, \mathcal{P}'' \subseteq \mathcal{P}$ that satisfies Condition 6.1.14, using the following indicator matrices:

— Let $\boldsymbol{\Pi} \in \mathbb{K}^{(d+2)\times(d+2)^{(\ell-t)\times\ell_1}}$ be such that for every $p' \in \mathcal{P}'$ it records in its rows its dependence on the ℓ_1 *internal* probes (without loss of generality, $\{p_1, \ldots, p_{\ell_1}\}$) of $\mathcal{P}$ as scalar matrices; that is, $\boldsymbol{\Pi}$ is such that $p'_i = \sum_{j=1}^{\ell_1} \boldsymbol{\pi}_{i,j} p_j + \sum_{j=\ell_1+1}^{\ell} \alpha_j p_j$, where $\boldsymbol{\pi}_{i,j}$ is the scalar on the diagonal of the scalar matrix $\boldsymbol{\Pi}_{i,j}$ and the α_js are unimportant. Without loss of generality, we may assume that every internal probe of $\mathcal{P}$ appears at least once in a linear combination of $\mathcal{P}'$, otherwise it is simply discarded, so $\boldsymbol{\Pi}$ has no zero column.

— Let $\boldsymbol{\Delta} \in \mathbb{K}^{(d+2)\times(d+2)^{(\ell-t)\times d'}}$ be the matrix that for every $p' \in \mathcal{P}'$ records in its rows its dependence on the shares $\boldsymbol{a}_i$s. If a row of $\boldsymbol{D}$ is all zero, the corresponding column is not included in $\boldsymbol{\Delta}$, and since $\boldsymbol{D}$ has at least ℓ_1+1 non-zero rows, $\boldsymbol{\Delta}$ has at least $d' \geq \ell_1+1$ columns none of which are zero.

Now we invoke Lemma 6.1.13 with $\boldsymbol{\Pi}$ as $\boldsymbol{G}_1$ and $\boldsymbol{\Delta}$ as $\boldsymbol{G}_2$ the generator matrices for the code $\mathcal{C}_{1,2}$. Let $\boldsymbol{c} \in \mathcal{C}_{1,2}$ be a codeword that satisfies $\mathrm{wt}_1(\boldsymbol{c}) < \mathrm{wt}_2(\boldsymbol{c})$; this translates to a linear combination of $\ell'' := \mathrm{wt}_1(\boldsymbol{c})$ internal probes to which one can add a linear combination of up to ℓ_2 external probes such that it does not depend on any randomness and the associated matrix $\begin{pmatrix}\boldsymbol{M}'' & \boldsymbol{\mu}''\end{pmatrix}$ has $\mathrm{wt}_2(\boldsymbol{c}) \geq \ell''+1$ non-zero rows. The set $\mathcal{P}'' \subseteq \mathcal{P}$ of these internal and external probes thus satisfies Condition 6.1.14. □

And we then have the immediate corollary:

Corollary 6.1.16. *Let C be a $(d+1, v)$-gadget for a function $f : \mathbb{K}^2 \to \mathbb{K}$ for which all probes are bilinear, then C is d-SNI iff. there is not set of $t \leq d$ probes on C that satisfies Condition 6.1.14.*

Proof. From left to right, by contrapositive: a set of probe satisfying Condition 6.1.14 functionally depends on at least ℓ_1+1 shares of $\boldsymbol{a}$ or $\boldsymbol{b}$, without functionally depending on any $\boldsymbol{r}_i$; it cannot be simulated using ℓ_1 or fewer shares of either $\boldsymbol{a}$ or $\boldsymbol{b}$ and thus C is not d-SNI.

From right to left: it follows directly from Theorem 6.1.15. □

6.1.3 Security of binary schemes over finite fields of characteristic two

Let C be a d-NI or SNI gadget for a function defined over $\mathbb{F}_2$; a natural question is whether its security is preserved if it is lifted to an extension $\mathbb{F}_{2^n}$. Indeed, the probes available to the adversary are the same in the two cases, but the latter offers more possible linear combinations $\sum_{i=1}^{\ell} \boldsymbol{\lambda}_i p_i$, since the $\boldsymbol{\lambda}_i$s are no longer restricted to $\{0,1\}$. We answer this question positively, and give a simple proof based on Theorems 6.1.7 and 6.1.15.

Theorem 6.1.17. *Let C be a d-NI (respectively d-SNI) gadget for a function $f : \mathbb{F}_2^2 \to \mathbb{F}_2$, then for any n, the natural lifting $\widehat{C}$ of C to $\widehat{f} : \mathbb{F}_{2^n}^2 \to \mathbb{F}_{2^n}$ is also d-NI (respectively d-SNI).*

Proof. We only prove the d-NI case, the d-SNI one being similar. From Corollary 6.1.8, it is sufficient to show that if there is no set of probes $\mathcal{P}$ for C that satisfies Condition 6.1.6, then the same holds for $\widehat{C}$. We do this by showing the following contrapositive: if a set of probes $\mathcal{P}$ is not d-simulatable for $\widehat{C}$, then it is not d-simulatable either for C.

From the proofs of Theorems 6.1.4 and 6.1.7, if $\mathcal{P}$ is not d-simulatable for $\widehat{C}$, then there is a matrix $\widehat{\boldsymbol{D}}$ that leads to the existence of $\mathcal{P}'$ such that Condition 6.1.6 is satisfied. All we need to do is showing that a similar matrix $\boldsymbol{D}$ can also be found for C. Since C is defined over $\mathbb{F}_2$, the matrices $\boldsymbol{R}$ and $\boldsymbol{P}$, and thence $\widehat{\boldsymbol{R}}$ and $\widehat{\boldsymbol{P}}$ have all their coefficients in $\{0,1\}$. As 1 is its own inverse, the change-of-basis matrix from $\widehat{\boldsymbol{R}}$ to $\widehat{\boldsymbol{R}}'$ is also binary; equivalently, this means that the row elimination of $\widehat{\boldsymbol{R}}$ can be done in the subfield $\mathbb{F}_2$. Thus one only has to take $\boldsymbol{D} = \widehat{\boldsymbol{D}}$ to satisfy Condition 6.1.6 on C. □

This result is quite useful as it means that the security of a binary scheme only needs to be proven once in $\mathbb{F}_2$, even if it is eventually used in one or several extension fields. Proceeding thusly is in particular beneficial in terms of verification performance, since working over $\mathbb{F}_2$ limits the number of linear combinations to consider and may lead to some specific optimisations (cf. *e.g.* Sections 6.2 and 6.3).

Remark. This result was in fact already implicitly used (in a slight variant) by Barthe *et al.* in their masking compiler [BBD+15] and in maskVerif [BBC+19], since they use gadgets defined over an arbitrary structure $(\mathbb{K}, 0, 1, \oplus, \ominus, \odot)$. However we could not find a proof therein, which actually seems necessary to justify the correctness of this approach and of our algorithms of the next section.

6.2 Algorithm for checking (strong) non-interference

In this section, we present a new efficient algorithm to check if a scheme is (strong) non-interfering. This algorithm is a modification of the one presented by Belaïd *et al.* at EUROCRYPT 2016 [BBP+16, Section 8], and its correctness crucially relies on Theorems 6.1.7 and 6.1.15; it thus only applies to schemes for which all probes are bilinear, but this is not a hard restriction in practice since this condition is satisfied by most schemes of the literature.

In all of the following we assume that the field $\mathbb{K}$ over which the scheme is defined is equal to $\mathbb{F}_2$, which means that we simultaneously assess its security in that field and all its extensions (cf. Subsection 6.1.3). Some discussion of implementation in the NI case for schemes natively defined over larger fields (meaning that shares or random masks may be multiplied by constants not in $\{0, 1\}$) for which the new Theorem 6.1.7 is not needed can be found in [KR18].

We start by introducing some vocabulary and by recalling the algorithm from Belaïd *et al.*.

Definition 6.2.1 (Elementary probes)**.** A probe p is called *elementary* if it is of the form $p = \boldsymbol{a}_i \boldsymbol{b}_j$ (elementary *deterministic* probe) or $p = \boldsymbol{r}_i$ (elementary *random* probe).

Definition 6.2.2 (Shares indicator matrix)**.** Let p be a bilinear probe. We call *shares indicator matrix* and write $\boldsymbol{M}_p$ the matrix $\boldsymbol{M}$ from Definition 6.1.1.

Definition 6.2.3 (Randomness indicator matrix)**.** Let p be a bilinear probe. We call *randomness indicator matrix* and write $\boldsymbol{\sigma}_p$ the column matrix $\boldsymbol{\sigma}$ from Definition 6.1.1.

Example 6.5: Elementary probes and indicator matrices
We reuse the circuit and probes defined in Example 5.4.

The probe $p_2 = \boldsymbol{a}_0 \boldsymbol{b}_1$ is an example of an elementary deterministic probe on the circuit.

The share indicator matrix of $p_1 = \boldsymbol{a}_0 \boldsymbol{b}_0 \oplus r$ is $\boldsymbol{M}_{p_1} = \begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix}$ and the one of $p_2 = \boldsymbol{a}_0 \boldsymbol{b}_1$ is $\boldsymbol{M}_{p_2} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$. The randomness indicator matrix of p_1 is $\boldsymbol{\sigma}_{p_1} = \begin{pmatrix} 1 \end{pmatrix}$ and the one of p_2 is $\boldsymbol{\sigma}_{p_2} = \begin{pmatrix} 0 \end{pmatrix}$.

6.2.1 The algorithm from EUROCRYPT 2016

At EUROCRYPT 2016, Belaïd *et al.* presented an efficient probabilistic algorithm to find potential attacks against the d-privacy notion[8] for masking schemes for the multiplication over $\mathbb{F}_2$. By running the algorithm many times and not detecting any attack, one can also establish the security of a scheme up to some probability, but deriving a deterministic counterpart is less trivial. This algorithm works as follows.

[8] It can also be trivially modified to check attacks against NI security.

Consider a scheme on which all possible probes $\mathcal{P}$ are bilinear, and let $\boldsymbol{H}_{\mathcal{P}} := \left(\boldsymbol{\sigma}_p\right), p \in \mathcal{P}$ be the block matrix constructed from all the corresponding randomness indicator matrices. The algorithm of [BBP+16, Section 8] starts by finding a set of fewer than d probes whose sum[9] does not depend on any randomness. That is to say, it is looking for a vector $\boldsymbol{x}$ such that $\boldsymbol{H}_{\mathcal{P}} \cdot \boldsymbol{x} = \boldsymbol{0}$ and $\mathrm{wt}(\boldsymbol{x}) \leq d$. This can be immediately reformulated as a coding problem, as one is in fact searching for a codeword of weight less than d in the dual code of $\boldsymbol{H}_{\mathcal{P}}$. This search can then be performed using any information set decoding algorithm, and Belaïd *et al.* used the original one of Prange [Pra62].[10] Once such a set has been found, it is tested against [BBP+16, Condition 2] (which is similar to Condition 6.1.3) to determine if it is a valid attack against the d-NI notion, and [BBP+16, Condition 1] to determine if it is an attack for d-privacy. This procedure is then repeated until an attack is found or one has gained sufficient confidence in the security of the scheme.

Removing elementary *deterministic* probes. To make the above procedure more efficient, an important observation made by Belaïd *et al.* is that if the sum of every probe of a given set does not functionally depend on some $\boldsymbol{a}_i$ or $\boldsymbol{b}_j$, it is always possible to make it so by adding a corresponding elementary probe $\boldsymbol{a}_i\boldsymbol{b}_j$. This can be used to check, say, d-NI security by simply comparing the number of missing $\boldsymbol{a}_i$ or $\boldsymbol{b}_j$ to $d - \mathrm{wt}(\boldsymbol{x})$. This allows to reduce the number of probes that one has to include in $\mathcal{P}$ (and thus the dimension of $\boldsymbol{H}_{\mathcal{P}}$), making the algorithm more efficient.

6.2.2 A new algorithm based on enumeration

We now describe a new algorithm based on a partial enumeration of the power set $\wp(\mathcal{P})$ of $\mathcal{P}$. The idea is to simply consider every sum of fewer than d probes and check if it depends on all shares and no random masks, relying on Corollaries 6.1.8 and 6.1.16 for correctness. Since the cost of such an enumeration quickly grows with the size of $\mathcal{P}$, we then follow and extend the above observation by Belaïd *et al.* and only perform the enumeration on a reduced set. We first describe a simple extension of this "dimension reduction" strategy, before detailing the algorithms themselves. A more elaborate dimension reduction process is then described in Subsection 6.2.3, and we discuss implementation aspects in Section 6.3.

Removing elementary *random* probes. It is easy to adapt a deterministic enumeration so that one can completely remove elementary random probes; it suffices to remark that if the sum of every probe of a given set functionally depends on some $\boldsymbol{r}_i$, it is always possible to make it not so by adding the corresponding elementary probes.

Combining the two above observations, we may remove every elementary probe from the set $\mathcal{P}$.[11] This can be summarized by saying that in the enumeration, one is not restricted anymore to finding exactly a combination of fewer than d probes that depends on all shares and no random masks, as it is enough to find a combination of $\ell \leq d$ probes that depends on u shares and v masks as long as $d - \ell \geq (d + 1 - u) + v$, since the missing shares and extra masks can be dealt with elementary probes in a predictable way. This is in fact exactly the check that is performed in our implementation in the case of NI security, as is detailed and justified below.

6.2.2.1 Checking a scheme for non-interference

We now state the following:

[9] That is, the only non-trivial linear combination over $\mathbb{F}_2$ that depends on all the elements of the set.

[10] One may remark that since information set decoding relies on Gaussian elimination, the cost of one step of this algorithm increases more than linearly in the size of $\mathcal{P}$.

[11] Note that this means that one would not detect the existence of an attack that would use *only* elementary probes. However, it is easy to see from their definitions that ℓ such probes functionally depend on at most ℓ shares, and so can never lead to a non-trivial attack.

Proposition 6.2.4. *Let C be a $(d+1, v)$-gadget for a function $f : \mathbb{F}_2^2 \to \mathbb{F}_2$ for which all probes are bilinear, and $\mathcal{Q}_0$ be a set of n_0 non-elementary probes on C that functionally depends on n_a shares $\boldsymbol{a}_i$s, n_b shares $\boldsymbol{b}_j$s, and n_r random scalars $\boldsymbol{r}_i$s. Let $\mathcal{Q}_1$ be one of the smallest sets of elementary probes needed to complete $\mathcal{Q}_0$ such that $\mathcal{Q}_0 \cup \mathcal{Q}_1$ satisfies Condition 6.1.6 and functionally depends on all the $\boldsymbol{a}_i$s or all the $\boldsymbol{b}_i$s.*[12] *Then $n_1 := \#\mathcal{Q}_1 = n_r + (d+1-\max(n_a, n_b))$.*

Proof. An elementary probe functionally depends on either one $\boldsymbol{r}_i$ or one $\boldsymbol{a}_i$ and one $\boldsymbol{b}_j$, but not both. Thus, the minimum number of elementary probes needed to cancel every $\boldsymbol{r}_i$ and to add the $d+1-n_a$ (respectively $d+1-n_b$) missing $\boldsymbol{a}_i$s (respectively $\boldsymbol{b}_j$s) in $\mathcal{Q}_0$ is $n_r + (d+1-n_a)$ (respectively $n_r + (d+1-n_b)$). Thus, $\#\mathcal{Q}_1 = \min(n_r + d + 1 - n_a, n_r + d + 1 - n_b) = n_r + d + 1 - \max(n_a, n_b)$. □

This proposition can then be used in a straightforward way to check if a scheme is d-NI. To do so, one simply has to enumerate every set $\mathcal{Q}_0 \in \wp(\mathcal{P})$ of d non-elementary probes or fewer and check if $n_0 + n_1 \leq d$. By Corollary 6.1.8, if no such set $\mathcal{Q}_0$ can be completed as in Proposition 6.2.4 and still contain fewer than d probes, then the scheme is d-NI.

6.2.2.2 Checking a scheme for strong non-interference

We only need to adapt Proposition 6.2.4 to distinguish between internal and external probes:

Proposition 6.2.5. *Let C be a $(d+1, v)$-gadget for a function $f : \mathbb{F}_2^2 \to \mathbb{F}_2$ for which all probes are bilinear, and $\mathcal{Q}_0$ be a set of n_0 non-elementary probes on C that functionally depends on n_a shares $\boldsymbol{a}_i$s, n_b shares $\boldsymbol{b}_j$s, and n_r random scalars $\boldsymbol{r}_i$s. Let n_I denote the number of internal probes in $\mathcal{Q}_0$. Then there is a set $\mathcal{Q}_1$ of n_r elementary random probes such that $\mathcal{Q}_0 \cup \mathcal{Q}_1$ satisfies Condition 6.1.14 iff.* $\max(n_a, n_b) > n_I + n_r$.

Proof. Recall that all elementary probes are internal. If $\mathcal{Q}_0$ does not satisfy Condition 6.1.14, then adding an elementary deterministic probe increases by at most one the number of non-zero rows, while increasing by one the total number of probes, so this completed set does not satisfy $\mathcal{Q}_0$ either. It is thus enough to only consider random probes in $\mathcal{Q}_1$.

For $\mathcal{Q} = \mathcal{Q}_0 \cup \mathcal{Q}_1$ to satisfy Condition 6.1.14, it is necessary to cancel all the potential randomness $\boldsymbol{r}_i$s on which $\mathcal{Q}_0$ depends; so $\mathcal{Q}_1$ must be the (possibly empty) set of the n_r corresponding elementary random probes. Now $\mathcal{Q}$ contains $n_I + n_r$ internal probes and it functionally depends on n_a $\boldsymbol{a}_i$s and n_b $\boldsymbol{b}_j$s. Thus it satisfies Condition 6.1.14 iff. $\max(n_a, n_b) > n_I + n_r$. □

This proposition can then be used in a straightforward way to check if a scheme is d-SNI. To do so, one simply has to enumerate every set $\mathcal{Q}_0 \in \wp(\mathcal{P})$ of d non-elementary probes or fewer and check if $\max(n_a, n_b) > n_I + n_r$ and $n_0 + n_r \leq d$. If no such set satisfying this condition is found, then the scheme is d-SNI by Corollary 6.1.16.

6.2.3 Further dimension reduction

To further reduce the size of the space to explore during the verification, it may be possible to filter additional non-elementary probes from the set $\mathcal{P}$, in the case where they can be replaced by "better" ones. To do so while preserving the correctness of our verification algorithm, we first define the following:

Definition 6.2.6 (Reduced sets)**.** Let $\mathcal{P} := \cup_{k=0}^{v} \mathcal{P}_k$ and $\mathcal{P}' := \cup_{k=0}^{v} \mathcal{P}'_k$ be two sets of probes on a $(d+1, v)$-gadget C for a function $f : \mathbb{F}_2^2 \to \mathbb{F}_2$ for which all probes are bilinear, where $\mathcal{P}_k$ (respectively $\mathcal{P}'_k$) denotes the probes on the wires of C that are connected to the output share $\boldsymbol{c}_k$. Then $\mathcal{P}'$ is said to be a *reduced set* for $\mathcal{P}$ iff.:

— $\#\mathcal{P}' \leq \#\mathcal{P}$

[12] This additional constraint is not in itself necessary, but it simplifies the overall algorithm.

— For all output wires k, for every linear combination of probes of $\mathcal{P}_k$ there is a linear combination of equal or lower weight of probes of $\mathcal{P}'_k$ with: 1) exactly the same randomness dependence (reusing the notation of Definition 6.1.1 this means that both combinations have the same $\boldsymbol{\sigma}$ term); 2) the shares dependence of the combination from $\mathcal{P}'_k$ covers the one of the combination from $\mathcal{P}_k$ (*i.e.* the support of the $\boldsymbol{M}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu}$ terms of the former include the ones of the same terms of the latter).

We then have:

Lemma 6.2.7. *If two linear combinations of probes $\sum \boldsymbol{\lambda}_i p_i$ and $\sum \boldsymbol{\lambda}'_i p'_i$ functionally depend on disjoint sets of elementary probes and shares $\boldsymbol{a}_i\boldsymbol{b}_j$, $\boldsymbol{a}_i$ and $\boldsymbol{b}_j$, then their sum functionally depends on the union of those sets.*

Proof. Immediate, since using the notation of Definition 6.1.1, the supports of $\boldsymbol{M}$, $\boldsymbol{\mu}$, $\boldsymbol{\nu}$ are disjoint from the ones of $\boldsymbol{M}'$, $\boldsymbol{\mu}'$, $\boldsymbol{\nu}'$. □

Finally, we conclude with the following:

Proposition 6.2.8. *Let $\mathcal{P}'$ be a reduced set for a set of probes $\mathcal{P}$ on a $(d+1, v)$-gadget C for a function $f : \mathbb{F}_2^2 \to \mathbb{F}_2$ for which all probes are bilinear and for which all output shares functionally depend on pairwise disjoint sets of elementary probes and shares $\boldsymbol{a}_i\boldsymbol{b}_j$, $\boldsymbol{a}_i$ and $\boldsymbol{b}_j$. Then if $\mathcal{Q} \subseteq \mathcal{P}$ satisfies Condition 6.1.6, $\exists\, \mathcal{Q}' \subseteq \mathcal{P}'$, $\#\mathcal{Q}' \leq \#\mathcal{Q}$ that also satisfies Condition 6.1.6.*

Proof. Let us write $\mathcal{Q}$ as $\cup_{k=0}^{v} \mathcal{Q}_k$ (respectively $\mathcal{Q}'$ as $\cup_{k=0}^{v} \mathcal{Q}'_k$) where $\mathcal{Q}_k$ (respectively $\mathcal{Q}'_k$) denotes the probes on the wires of C that are connected to the output share $\boldsymbol{c}_k$. Let $\sum_{p_i \in \mathcal{Q}} \boldsymbol{\lambda}_i p_i$ denote one linear combination of elements of $\mathcal{Q}$ whose existence is guaranteed by its satisfying Condition 6.1.6, which we rewrite as: $\sum_k \sum_{p_i^{(k)} \in \mathcal{Q}_k} \boldsymbol{\lambda}_i^{(k)} p_i^{(k)}$. For each $\boldsymbol{\lambda}^{(k)}$, let $\boldsymbol{\lambda}'^{(k)}$ be the coefficients for one of the linear combination of elements of $\mathcal{Q}'_k$ whose existence is guaranteed by $\mathcal{P}'$ being a reduced set for $\mathcal{P}$.

Each $\sum_{p_i^{(k)} \in \mathcal{Q}_k} \boldsymbol{\lambda}_i^{(k)} p_i^{(k)}$ (respectively $\sum_{p_i^{(k)} \in \mathcal{Q}'_k} \boldsymbol{\lambda}_i'^{(k)} p_i^{(k)}$) satisfies the premise of Lemma 6.2.7 which can be applied successively on $\sum_{j=0}^{k-1} \sum_{p_i^{(j)} \in \mathcal{Q}_j} \boldsymbol{\lambda}_i^{(j)} p_i^{(j)}$ and $\sum_{p_i^{(k)} \in \mathcal{Q}_k} \boldsymbol{\lambda}_i^{(k)} p_i^{(k)}$ (respectively $\sum_{j=0}^{k-1} \sum_{p_i^{(j)} \in \mathcal{Q}'_j} \boldsymbol{\lambda}_i'^{(j)} p_i^{(j)}$ and $\sum_{p_i^{(k)} \in \mathcal{Q}'_k} \boldsymbol{\lambda}_i'^{(k)} p_i^{(k)}$).

It follows that both $\sum_k \sum_{p_i^{(k)} \in \mathcal{Q}_k} \boldsymbol{\lambda}_i^{(k)} p_i^{(k)}$ and $\sum_k \sum_{p_i^{(k)} \in \mathcal{Q}'_k} \boldsymbol{\lambda}_i'^{(k)} p_i^{(k)}$ do not functionally depend on any elementary random probe $\boldsymbol{r}_i$, and the elementary deterministic probes and shares on which the latter functionally depends is a superset of the ones on which depends the former; thus $\mathcal{Q}'$ satisfies Condition 6.1.6. □

Example 6.6: Reduced set of probes

Consider a set $\mathcal{P}$ of three probes $\boldsymbol{a}_0\boldsymbol{b}_1$, $\boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{r}_0 + \boldsymbol{a}_0\boldsymbol{b}_1$ and $\boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{r}_0 + \boldsymbol{a}_0\boldsymbol{b}_1 + \boldsymbol{a}_1\boldsymbol{b}_0$ on the same output share. Then, the set $\mathcal{P}' = \{\boldsymbol{a}_0\boldsymbol{b}_1, \boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{r}_0 + \boldsymbol{a}_0\boldsymbol{b}_1 + \boldsymbol{a}_1\boldsymbol{b}_0\}$ is a reduced set for $\mathcal{P}$ because for any linear combination of k probes in $\mathcal{P}$, a linear combination of $k' \leq k$ probes in $\mathcal{P}'$ can be computed such that it has exactly the same randomness dependence and has at least the same shares dependence.

On the other hand, a set containing two probes $\boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{r}_0 + \boldsymbol{a}_0\boldsymbol{b}_1 + \boldsymbol{a}_1\boldsymbol{b}_0$ and $\boldsymbol{a}_0\boldsymbol{b}_0 + \boldsymbol{r}_0 + \boldsymbol{a}_0\boldsymbol{b}_1 + \boldsymbol{a}_1\boldsymbol{b}_0 + \boldsymbol{r}_1$ cannot be simplified since the two probes do not include exactly the same random masks.

Algorithm 1 shows a summary of the algorithm to check the (strong) non-interference at order d of a gadget given the set of all possible probes on this gadget.

Section 6.4 shows how Proposition 6.2.8 can be used in practice to significantly improve verification performance. The nature of the probes that can be removed of course depends on the scheme under consideration, and we will later detail how to do this for some concrete gadgets.

Algorithm 1: Summary of our algorithm to check if a gadget is d-(S)NI.

Input : $\mathcal{P}$, the set of all possible probes on the gadget C
Input : The order d of the verification
Input : `SNI` a boolean indicating if we check the d-SNI or the d-NI property
Output: `True` if C is d-(S)NI else `False`

```
P ← {p ∈ P | p is not an elementary probe}        // Removing elementary probes
P ← Reduce(P)                                     // Using Proposition 6.2.8
for 1 ≤ n_0 ≤ d do
    for Q_0 ∈ {Q ∈ ℘(P) | #Q = n_0} do
        n_r ← number of r_i on which Q_0 functionally depends
        n_a ← number of a_i on which Q_0 functionally depends
        n_b ← number of b_i on which Q_0 functionally depends
        if SNI then
            n_I ← number of internal probes in Q_0
            if max(n_a, n_b) > n_i + n_r and n_0 + n_r ≤ d then
                return False          // Q_0 can be completed into a d-SNI attack
            end
        else
            n_1 ← n_r + (d + 1 − max(n_a, n_b))
            if n_0 + n_1 ≤ d then
                return False          // Q_0 can be completed into a d-NI attack
            end
        end
    end
end
return True
```

6.2.4 Adaptation to the robust probing model

The verification of a gadget is done in two steps: first iterate over all subsets $\mathcal{P}$ of d probes or fewer; then check that the subset $\mathcal{P}$ does not lead to an attack. In the case of the non-robust probing model, the second step can be done directly, as explained in Subsection 6.2.2, by considering the sum of the expression associated with each probe in $\mathcal{P}$.

However, when the implementation of the masking scheme is in a context where there are hardware defects to take into account, one may need to verify the target gadget in the robust probing model seen previously in Section 5.3. In this model each probe p can be associated with a leakage set $\mathfrak{L}(p)$ containing more than one expression. Since the value leaked for a given probe can be any binary linear combination of the expressions in its leakage set, there are in general $\prod_{p\in\mathcal{P}} \left(2^{\#\,\mathfrak{L}(p)} - 1\right)$ expressions for which we want to know if they satisfy the appropriate attack condition. We explain next in Section 6.3 how this can be done efficiently.

Related work. The maskVerif tool [BBC+19] also implements the robust probing model to check security in presence of glitches. More dedicated approaches are the ones of Bloem *et al.* [BGI+18] and of the SILVER tool [KSM20].

6.3 Implementing algorithm of Section 6.2 as a tool

We now describe an efficient `C` implementation of the algorithms of the previous section for $\mathbb{K} = \mathbb{F}_2$. Our software is publicly available at https://github.com/NicsTr/binary_masking.

6.3.1 Data structures

To evaluate if a set of probes $\mathcal{P}$ may lead to an attack, it is convenient to define the following:

Definition 6.3.1 (Attack matrix)**.** The *attack matrix* $\boldsymbol{A}_{\mathcal{P}}$ of a set of probes $\mathcal{P}$ is defined as the sum of the share indicator matrices of the probes in $\mathcal{P}$:

$$\boldsymbol{A}_{\mathcal{P}} = \sum_{p \in \mathcal{P}} \boldsymbol{M}_p.$$

Definition 6.3.2 (Noise matrix)**.** The *noise matrix* $\boldsymbol{B}_{\mathcal{P}}$ of a set of probes $\mathcal{P}$ is defined as the sum of the randomness indicator matrices of the probes in $\mathcal{P}$:

$$\boldsymbol{B}_{\mathcal{P}} = \sum_{p \in \mathcal{P}} \boldsymbol{\sigma}_p.$$

One can then simply compute the quantities n_a, n_b and n_r needed in Propositions 6.2.4 and 6.2.5 as the number of non-zero rows or columns of these two matrices, which we do using an efficient vectorised Hamming weight routine. To analyse a given scheme, one then just has to provide a full description of $\boldsymbol{M}_p$ and $\boldsymbol{\sigma}_p$ for every non-elementary probe. Additionally, since Proposition 6.2.5 requires to compute the number of internal probes n_I in a set, those have to be labelled as such.

We inline all data structures and store them in either standard or vector registers. $\boldsymbol{A}_{\mathcal{P}}$ is stored twice, once row-wise and once column-wise, in order to avoid the otherwise costly transposition needed to compute both its row and its column weight. For schemes at order $d \leq 15$, each row or column fits within a 16-bit words leading to a quite efficient vectorised Hamming weight computation, as shown in Listing 6.1. We also provide a slower implementation for schemes at higher order; in this case actually proving the security with our algorithm is likely to be intractable due to the combinatorial explosion of the number of sets to consider, yet a partial run may still be able to detect attacks, in the fashion of the original algorithm from EUROCRYPT 2016.

```
int popcount256_16(__m256i v)
{
        return __builtin_popcountl(_mm256_cmpgt_epi16_mask(v,
↪   _mm256_setzero_si256()));
}
```

Listing 6.1: Hamming weight computation of a vector of dimension 16 over 16-bit words using AVX512VL and AVX512BW; a variant with only a few more instructions can be used with only AVX2.

6.3.2 Amortised enumeration & parallelisation

Recall that to prove the security of a scheme at order d, the algorithm of Section 6.2 requires to enumerate all the $\sum_{i=1}^{d} \binom{n}{i}$ subsets of a (possibly filtered) set of probes $\mathcal{P}$ of size n. For a subset $\mathcal{P}' \subseteq \mathcal{P}$ of size ℓ, a naïve approach in computing $\boldsymbol{A}_{\mathcal{P}'}$ would use $\ell - 1$ additions, and this for every such $\mathcal{P}'$. However, a well-known optimisation for this kind of enumeration is instead to go through all the subsets of a fixed weight in a way that ensures that two consecutive sets $\mathcal{P}'$ and $\mathcal{P}''$ only differ by two elements. One can then compute, say, $\boldsymbol{A}_{\mathcal{P}''}$ efficiently by updating $\boldsymbol{A}_{\mathcal{P}'}$ with one addition and one subtraction. We do this in our implementation by using a so-called "revolving-door algorithm" described by Knuth [Knu11, Algorithm R] for the Nijenhuis-Wilf-Liu-Tang "combination Gray code"[NW78; LT73].

In practice, Algorithm 2 is used to compute both attack matrix $\boldsymbol{A}$ and noise matrix $\boldsymbol{B}$ for every set of k probes among the n probes $\mathcal{P}$ available on the circuit. Going from a matrix $\boldsymbol{A}_{\mathcal{P}'}$ to the next matrix $\boldsymbol{A}_{\mathcal{P}''}$ can always be done using only one addition and one subtraction:

Algorithm 2: Revolving-door combinations algorithm [Knu11, Algorithm R] going over the $\binom{n}{k}$ combinations $\boldsymbol{c}_k \dots \boldsymbol{c}_2 \boldsymbol{c}_1$.

R1. Set $\boldsymbol{c}_j \leftarrow j-1$ for $1 \leq j \leq k$ and $\boldsymbol{c}_{k+1} \leftarrow n$.
R2. The combination $\boldsymbol{c}_k \dots \boldsymbol{c}_2 \boldsymbol{c}_1$ is ready to be used.
R3. (Step depending on the parity of k)
If k is odd: If $\boldsymbol{c}_1 < \boldsymbol{c}_2$, increase $\boldsymbol{c}_1$ by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R4.
If k is even: If $\boldsymbol{c}_1 > 0$, decrease $\boldsymbol{c}_1$ by 1 and return to R2, otherwise set $j \leftarrow 2$ and go to R5.
R4. (At this point $\boldsymbol{c}_j = \boldsymbol{c}_{j-1} + 1$.)
If $\boldsymbol{c}_j \geq j$, set $\boldsymbol{c}_j \leftarrow \boldsymbol{c}_{j-1}$, $\boldsymbol{c}_{j-1} \leftarrow j-2$, and return to R2.
Otherwise increase j by 1.
R5. (At this point $\boldsymbol{c}_{j-1} = j-2$.)
If $\boldsymbol{c}_j + 1 < \boldsymbol{c}_{j+1}$, set $\boldsymbol{c}_{j-1} \leftarrow \boldsymbol{c}_j$, $\boldsymbol{c}_j \leftarrow \boldsymbol{c}_j + 1$, and return to R2.
Otherwise increase j by 1, and go to R4 if $j \leq k$.
The algorithm has reached its end if $j > k$.

— if the update is done in step R3, one needs to subtract $\boldsymbol{M}_{p_{\boldsymbol{c}_1}}$ and add, depending on the parity of k, either $\boldsymbol{M}_{p_{\boldsymbol{c}_1+1}}$ or $\boldsymbol{M}_{p_{\boldsymbol{c}_1-1}}$;

— if the update is done in step R4, one needs to subtract $\boldsymbol{M}_{p_{\boldsymbol{c}_j}}$ and add $\boldsymbol{M}_{p_{j-2}}$;

— if the update is done in step R5, one needs to subtract $\boldsymbol{M}_{p_{\boldsymbol{c}_{j-1}}}$ and add $\boldsymbol{M}_{p_{\boldsymbol{c}_j+1}}$.

The same applies to go from $\boldsymbol{B}_{\mathcal{P}'}$ to $\boldsymbol{B}_{\mathcal{P}''}$ by adding and subtracting the right matrices $\boldsymbol{\sigma}_p$.

In the robust probing model setting one may also need to enumerate more than one expression for a given set of probes; this can still be done efficiently using Gray codes. First one uses the same approach as described above to enumerate the sets of probes thanks to a combination Gray code. Then for each of these sets $\mathcal{P}$, checking if it leads to an attack or not requires one to go over the $\prod_{p \in \mathcal{P}} (2^{\#\,\mathfrak{L}(p)} - 1)$ linear combinations of the relevant leakage sets as explained in Subsection 6.2.4. This enumeration itself is done using two layers of Gray codes: an outer layer is composed of a mixed-radix Gray code of length $\#\mathcal{P}$, with the radix associated with probe p being equal to $2^{\#\,\mathfrak{L}(p)}$; this outer Gray code indicates at each step which probe needs to be "incremented" to obtain the next linear combination. Then this increment is itself implemented efficiently by using an inner ("standard") binary Gray code in dimension $\#\,\mathfrak{L}(p)$.

The entire enumeration process can also be easily parallelised, and the main challenge is to couple this with the above amortised approaches. This can in fact be done quite efficiently, as the combination Gray code that we use to enumerate the probe subsets possesses an efficient *unranking* map from the integers to arbitrary configurations [Wal, p.25-26] given in Algorithm 3.

Algorithm 3: Unranking algorithm from [Wal, pp.25-26].

Input : The rank r strictly lower than $\binom{n}{k}$
Output: The configuration $\boldsymbol{c}_k \dots \boldsymbol{c}_1$ of rank r
for $i \leftarrow k$ **to** 1 **do**
  $\boldsymbol{c}_i \leftarrow \min\{x \mid \binom{x}{i} \geq r\}$
  $r \leftarrow \binom{\boldsymbol{c}_i}{i} - r$
end

One can then easily divide a full enumeration of a total of n combinations into j jobs by starting each of them independently at one of the configurations given by the unranking of $i \times n/j$, $i \in [\![0, j[\![$.

6.3.3 From high-level representation to `C` description

We use a custom parser to convert a readable description of a masking scheme into a `C` description of its probes' indicator matrices.

Each line of the high-level description corresponds to an output share. The available symbols are:

— `sij` which represents a product $\boldsymbol{a}_i\boldsymbol{b}_j$;

— `ri` which represents a random mask $\boldsymbol{r}_i$;

— a space '␣', a binary operator which represents an addition (*i.e.* XOR) gate;

— parentheses, which allow explicit scheduling of the operations;

— `|`, a postfixed unary operator which represents the use of a register to store the expression that is *before* the symbol. This is only needed for an analysis in presence of glitches.

Additionally, the user needs to specify the order d of the scheme as well as the list of random masks used.

The scheduling of the operations needed to compute the output shares is important, as it determines the probes available to the adversary. In that respect, the parser uses by default an implicit left-to-right scheduling and addition gates have precedence over registers. As an example the scheme whose output shares are defined as:

$$\begin{aligned}\boldsymbol{c}_0 &= ((((\boldsymbol{a}_0\boldsymbol{b}_0 \oplus \boldsymbol{r}_0) \oplus \boldsymbol{a}_0\boldsymbol{b}_1) \oplus \boldsymbol{a}_1\boldsymbol{b}_0) \oplus \boldsymbol{r}_1)\\ \boldsymbol{c}_1 &= ((((\boldsymbol{a}_1\boldsymbol{b}_1 \oplus \boldsymbol{r}_1) \oplus \boldsymbol{a}_1\boldsymbol{b}_2) \oplus \boldsymbol{a}_2\boldsymbol{b}_1) \oplus \boldsymbol{r}_2)\\ \boldsymbol{c}_2 &= ((((\boldsymbol{a}_2\boldsymbol{b}_2 \oplus \boldsymbol{r}_2) \oplus \boldsymbol{a}_2\boldsymbol{b}_0) \oplus \boldsymbol{a}_0\boldsymbol{b}_2) \oplus \boldsymbol{r}_0)\end{aligned}$$

is described by the file:

```
ORDER = 2
MASKS = [r0, r1, r2]
s00 r0 s01 s10 r1
s11 r1 s12 s21 r2
s22 r2 s20 s02 r0
```

Another example is the following *DOM-indep* multiplication by Groß *et al.* [GMK16], which is NI at order two even in the presence of glitches:

```
ORDER = 2
MASKS = [r0, r1, r2]
 s00      (s01 r0|) (s02 r1|)
(s10 r0|)  s11      (s12 r2|)
(s20 r1|) (s21 r2|)  s22
```

6.4 Application of the verification tool

In this section we apply our fast implementation of the verification algorithm of Section 6.2 to various state-of-the-art masking gadgets and also propose new improved instances in medium order, including better SNI multiplication and refreshing gadgets for the practically-relevant case of 8 shares.

We analyse:

— In Subsection 6.4.1: NI and SNI multiplication gadgets originally from [BDF+17; GJR+18].

— In Subsection 6.4.2: SNI refreshing gadgets originally from [BDF+17; BBD+18].

— In Subsection 6.4.3: Glitch-resistant NI multiplication from [GMK16].

6.4.1 NI and SNI multiplication gadgets

We first study a family of multiplication gadgets that were introduced by Barthe *et al.* at EUROCRYPT 2017 [BDF+17] and used in the efficient masked AES implementation of Grégoire *et al.* [GPS+18] (who also propose improvements in the 4-share setting) and in the very high order implementations of Journault and Standaert [JS17].

Our motivations in doing so are the following: since there is no known security proof at arbitrary order for these schemes, it is natural to try to prove them computationally at the highest possible order. Barthe *et al.* originally did this up to order 7,[13] and we manage to reach order 11 both for NI and SNI security, which represents a significant improvement.[14] A second motivation is that the verification of multiplication gadgets quickly becomes intractable with increasing order, and such a task allows us to clearly demonstrate our performance gain over maskVerif. Finally, this improved verification efficiency is exploited in trying to find *ad hoc* gadget variants with lower cost.

On the negative side our verification shows that a conjecture from Barthe *et al.* on the security of a natural strategy to convert NI multiplication into SNI fails at order 10. More positively, we were able to find *ad hoc* conversions tuned to every NI multiplication we considered, which sometimes also bring a significant improvement in randomness cost over Barthe *et al.*'s strategy. For instance we are able to gain 17% for an 8-share, 7-SNI gadget similar to the one used in [GPS+18]. Finally using a slight variant of Barthe *et al.*'s gadget generation algorithm, we occasionally obtain some improvements also in the NI case, notably at order 5.

We give details of our improvements in Table 6.1 and the descriptions of all the gadgets at `https://github.com/NicsTr/binary_masking`. Note however that Belaïd *et al.* also propose optimized gadgets in [BBP+16] up to order 4, that ISW is also better than [BDF+17] at order 3 and that Grégoire *et al.* already proposed improvements at this same order in [GPS+18]. The main range of interest of Table 6.1 is thus at order 5 and beyond.

6.4.1.1 The NI multiplication gadget family of [BDF+17, Algorithm 3]

We give in Algorithm 4 a description of a slightly modified variant of [BDF+17, Algorithm 3], which occasionally gives better gadgets than the original. We also provide a small script to automatically generate a scheme at a given order at `https://github.com/NicsTr/binary_masking`.

This description relies on the following convenient definition:

Definition 6.4.1 (Pair of shares)**.** Let $(\boldsymbol{a}_i\boldsymbol{b}_j)$, $i, j \in [\![0, d]\!]$ be the input shares of a $(d+1, v)$ gadget. We define $\hat{\boldsymbol{\alpha}}_{i,j}$ as:

$$\hat{\boldsymbol{\alpha}}_{i,j} = \begin{cases} \boldsymbol{a}_i\boldsymbol{b}_j & \text{if } i = j \\ \boldsymbol{a}_i\boldsymbol{b}_j + \boldsymbol{a}_j\boldsymbol{b}_i & \text{otherwise} \end{cases}$$

6.4.1.2 Extension to SNI security

One can derive an SNI multiplication gadget from Algorithm 4 by doing the following: 1) proving NI security at some order d; 2) proving SNI security at the same order for a *refreshing gadget*; 3) composing the two gadgets.

This strategy can for instance be implemented with the refreshing gadgets also introduced in [BDF+17] that we discuss in the next Subsection 6.4.2, but Barthe *et al.* already remarked that it was in fact apparently not necessary to use full refreshing gadgets and that one could do better by using a degraded variant thereof: in a nutshell, one starts from a secure NI multiplication and simply masks every output share with a fresh random mask and then again with the mask of the following share in a circular fashion.

[13] We ourselves used the latest version of maskVerif to do so up to order 8.

[14] This however still cannot theoretically justify the use of this masked multiplication at order 31 as is done in [JS17].

Table 6.1: Explicit randomness cost of multiplication gadgets.

Order d		Defined *and* verified in [BDF+17]		Defined *or* verified in §6.4.1	
		Random masks	XOR gates	Random masks	XOR gates
2	SNI	3	12	=	=
3	NI	4	20	=	=
	SNI	8	28	5	24
4	NI	5	30	=	=
	SNI	10	40	9	38
5	NI	12	54	10	50
	SNI	18	66	12	54
6	NI	14	70	=	=
	SNI	21	84	18	78
7	NI	—	—	16	88
	SNI	24	104	20	96
8	NI	—	—	18	108
	SNI	—	—	27	126
9	NI	—	—	26	142
	SNI	—	—	30	150
10	NI	—	—	33	176
	SNI	—	—	39	188
11	NI	—	—	36	204
	SNI	—	—	42	216

Barthe *et al.* then conjecture in [BDF+17] that this transformation is always enough to convert an NI scheme into an SNI one. However we could check that this is not true for 11- and 12-share gadgets: the respective instantiations of Algorithm 4 are NI, but the transformation fails to provide SNI multiplications. Yet it is in fact still possible to derive an 11-share, 10-SNI multiplication gadget at no additional cost by simply rotating the last repeated masks by two positions instead of one, for a total cost of 44 random masks.

We explored several other transformation strategies, trying to exploit the special shape of the NI multiplication gadgets as much as possible. This almost always improved on the use of a new mask for every share (the current exception being the order-8 gadget), usually requiring only about half. For instance our best 11-share gadget in fact only requires 39 masks instead of the above 44 as shown in Figure 6.1, and we found a 7-SNI multiplication with only 20 masks shown in Figure 6.2, which is 4 less than [BDF+17]. While this latter improvement is somewhat moderate at about 17%, this 8-share case is quite relevant due to its use in the efficient vectorised masked AES implementation of Grégoire *et al.* [GPS+18]; using our new variant should then result in a noticeable decrease in randomness usage. In fact, an implementation using our variant is described in Chapter 7 and publicly available.

Algorithm 4: A conjectured d-NI $(d+1, d+1)$-gadget for multiplication over fields of characteristic two.

Input : $\mathcal{S} = \{\hat{\boldsymbol{\alpha}}_{i,j}, 0 \leq i \leq j \leq d\}$
Input : $\mathcal{R} = \{\boldsymbol{r}_i\}$, $i \in \mathbb{N}$
Output: $(\boldsymbol{c}_i)_{0 \leq i \leq d}$, such that $\sum_{i=0}^{d} \boldsymbol{c}_i = \sum_{i=0}^{d} \boldsymbol{a}_i \sum_{i=0}^{d} \boldsymbol{b}_i$

for $i \leftarrow 0$ **to** d **do**
  $\boldsymbol{c}_i \leftarrow \hat{\boldsymbol{\alpha}}_{i,i}$
  $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\hat{\boldsymbol{\alpha}}_{i,i}\}$
end
$\mathcal{R}' \leftarrow \{\}$
$j \leftarrow 1$
while $\mathcal{S} \neq \emptyset$ **do**
  for $i \leftarrow 0$ **to** d **do**
    if $j \equiv 1 \mod 2$ **then**
      $\boldsymbol{c}_i \leftarrow \boldsymbol{c}_i + \boldsymbol{r}_{\frac{(j-1)}{2}.(d+1)+i}$
      $\mathcal{R}' \leftarrow \mathcal{R}' \cup \left\{\boldsymbol{r}_{\frac{(j-1)}{2}.(d+1)+i}\right\}$
    else
      $\boldsymbol{c}_i \leftarrow \boldsymbol{c}_i + \boldsymbol{r}_{\frac{(j-2)}{2}.(d+1)+(i+1 \mod (d+1))}$
      $\mathcal{R}' \leftarrow \mathcal{R}' \setminus \left\{\boldsymbol{r}_{\frac{(j-2)}{2}.(d+1)+(i+1 \mod (d+1))}\right\}$
    end
    if $\mathcal{S} \neq \emptyset$ **then**
      $\boldsymbol{c}_i \leftarrow \boldsymbol{c}_i + \hat{\boldsymbol{\alpha}}_{i,((i+j) \mod (d+1))}$
      $\mathcal{S} \leftarrow \mathcal{S} \setminus \{\hat{\boldsymbol{\alpha}}_{i,((i+j) \mod (d+1))}\}$
    else
      break
    end
  end
  $j \leftarrow j + 1$
end
$k \leftarrow \#\mathcal{R}'$
for $i \leftarrow 0$ **to** d **do**
  $\boldsymbol{c}_i \leftarrow \boldsymbol{c}_i + \boldsymbol{r}_{\frac{(j-1)}{2}(d+1)+(i+1 \mod k)}$
end

```
s00 r00 s01 s10 r01 s02 s20 r11 s03 s30 r12 s04 s40 r22 s05 s50 r23 r40
s11 r01 s12 s21 r02 s13 s31 r12 s14 s41 r13 s15 s51 r23 s16 s61 r24 r41
s22 r02 s23 s32 r03 s24 s42 r13 s25 s52 r14 s26 s62 r24 s27 s72 r25 r42
s33 r03 s34 s43 r04 s35 s53 r14 s36 s63 r15 s37 s73 r25 s38 s83 r26 r43
s44 r04 s45 s54 r05 s46 s64 r15 s47 s74 r16 s48 s84 r26 s49 s94 r27 r44
s55 r05 s56 s65 r06 s57 s75 r16 s58 s85 r17 s59 s95 r27 s5a sa5 r28 r45
s66 r06 s67 s76 r07 s68 s86 r17 s69 s96 r18 s6a sa6 r28 s60 s06 r29 r40
s77 r07 s78 s87 r08 s79 s97 r18 s7a sa7 r19 s70 s07 r29 s71 s17 r30 r41
s88 r08 s89 s98 r09 s8a sa8 r19 s80 s08 r20 s81 s18 r30 s82 s28 r31 r42
s99 r09 s9a sa9 r10 s90 s09 r20 s91 s19 r21 s92 s29 r31 s93 s39 r32 r43
saa r45 sa0 s0a r00 sa1 s1a r21 sa2 s2a r11 sa3 s3a r32 sa4 s4a r22 r44 r10
```

Figure 6.1: 10-SNI gadget for multiplication, using 39 random masks.

```
s00 r00 s01 s10 r01 s02 s20 r08 s03 s30 r09 s04 r20
s11 r01 s12 s21 r02 s13 s31 r09 s14 s41 r10 s15 r21
s22 r02 s23 s32 r03 s24 s42 r10 s25 s52 r11 s26 r22
s33 r03 s34 s43 r04 s35 s53 r11 s36 s63 r12 s37 r23
s44 r04 s45 s54 r05 s46 s64 r12 s47 s74 r13 s40 r20
s55 r05 s56 s65 r06 s57 s75 r13 s50 s05 r14 s51 r21
s66 r06 s67 s76 r07 s60 s06 r14 s61 s16 r15 s62 r22
s77 r07 s70 s07 r00 s71 s17 r15 s72 s27 r08 s73 r23
```

Figure 6.2: 7-SNI gadget for multiplication, using 20 random masks.

6.4.1.3 Verification performance

We now analyse the performance of our verification software on these multiplication schemes, and compare it with the one of the latest version of maskVerif [BBC+19].[15]

Probes filtering. Following the results of Subsection 6.2.3, we use a filtering process to reduce the initial set of probes that one has to enumerate to prove security. For the gadgets of Algorithm 4 and their SNI counterparts, this means removing probes of the form: $\hat{\boldsymbol{\alpha}}_{*,*} + \sum(\boldsymbol{r}_* + \hat{\boldsymbol{\alpha}}_{*,*}) + \boldsymbol{r}_* + \boldsymbol{a}_*\boldsymbol{b}_*$,[16] and the fact that the filtered set really is a reduced set in the sense of Definition 6.2.6 is verified by an exhaustive check on the subsets corresponding to every output share; this filtering process was only partially automated since an initial human intervention was necessary to identify the probes that could be removed. Intuitively, the idea is that one can always replace in an attack a probe of the above form with one that includes one extra $\boldsymbol{a}_j\boldsymbol{b}_i$ term, *i.e.* one of the form $\hat{\boldsymbol{\alpha}}_{*,*} + \sum(\boldsymbol{r}_* + \hat{\boldsymbol{\alpha}}_{*,*}) + \boldsymbol{r}_* + \hat{\boldsymbol{\alpha}}_{*,*}$, since the latter only adds an additional functional dependence on the input shares "for free".

The concrete impact of filtering on the verification performance of our schemes can be seen in Table 7.1, where we give the size of the attack sets to enumerate before and after this filtering.

Performance. For order $d \leq 10$ (except the 10-SNI case) we have run our software on a single core of the `retourdest` server, which features a single Intel Xeon Gold 6126 at 2.60 GHz. The corresponding timings are given in Table 6.2. At peak performance, we are able to enumerate $\approx 2^{27.5}$ candidate attack sets per second for NI verification, while SNI performance is slightly worse.

Using filtered sets significantly improves verification time, especially at high order. For instance, the running times of 2 and 6 hours for NI and SNI multiplication at order 9 are an order of magnitude faster than the 3 and 6 days initially spent before we implemented filtering. This optimisation was also essential in allowing to check the security of 10-NI multiplication in less than one calendar day on a single machine (using parallelisation); it would otherwise have taken a rather costly 1 core-year.

We also tested a multi-threaded implementation of our software on schemes at order $8 \sim 10$, using all 12 physical cores of the same Xeon Gold 6126; the results are shown in the right column of Table 6.2. While we do not have many data points, the speed-up offered by the parallelisation seems to be close to linear, albeit slightly less for NI verification: the 9-SNI multi-threaded wall time is ≈ 11.7 times less than the single-threaded one, and multi-threading for 9- and 10-NI saves a factor ≈ 9.7.

The largest schemes that we verified are NI (resp. SNI) multiplication at order $d = 11$. We relied heavily on parallelisation to enumerate the $\approx 2^{52.72}$ (resp. $\approx 2^{54.48}$) possible attack sets,[17] using up to 40 nodes of the *Dahu* cluster.[18] Each node has two 16-core Intel Xeon Gold 6130 at

[15] Available at https://gitlab.com/benjgregoire/maskverif.
[16] This corresponds exactly to the probes made of an even number of $\boldsymbol{a}_*\boldsymbol{b}_*$ terms.
[17] This is after filtering of the initial $\approx 2^{59}$ (resp. $\approx 2^{59.76}$) sets.
[18] https://ciment.univ-grenoble-alpes.fr/wiki-pub/index.php/Hardware:Dahu

2.10 GHz, and when using hyperthreading allows to enumerate $\approx 2^{31.38}$ sets per second[19]. This cluster was also used to verify the best version of our 10-SNI gadget.

Comparison with maskVerif. We used the maskVerif tool from Barthe *et al.* [BBC+19] to check the security of the gadgets at order 6 to 8. Due to system constraints, we could not run the verification on `retourdest`, and instead defaulted to the older `hpac`, which features an Intel Xeon E5-4620 at 2.20 GHz. We compare this to our software on this machine using 4 threads —the same amount of parallelisation that maskVerif is able to exploit.

The running times are summarised in Table 6.3. Even though we cannot benefit from vectorisation due to the absence of AVX2 instructions on `hpac`, it is notable that our own software is faster by three orders of magnitude, for instance taking slightly more than two minutes to check 8-NI multiplication *versus* two days for maskVerif. Note that this comparison is done after filtering in our case, which saves us up to a factor ≈ 30 (*cf.* for instance the 8-NI case) as can be computed from Table 6.2.

Table 6.2: Running time of our verification software on `retourdest`.

Order d		$\log_2$(number of sets) Before/After filtering	Wall time (1 thread) Best (after filtering)	Wall time (12 threads) Best (after filtering)
1	NI	2.6/2.6	$<$ 0.01 sec.	—
	SNI	2.6/2.6	$<$ 0.01 sec.	—
2	NI	6.3/5.5	$<$ 0.01 sec.	—
	SNI	6.3/5.5	$<$ 0.01 sec.	—
3	NI	10.4/8.9	$<$ 0.01 sec.	—
	SNI	11.2/9.96	$<$ 0.01 sec.	—
4	NI	15.0/12.6	$<$ 0.01 sec.	—
	SNI	16.4/14.6	$<$ 0.01 sec.	—
5	NI	21.2/18.6	$<$ 0.01 sec.	—
	SNI	21.7/19.3	$<$ 0.01 sec.	—
6	NI	27.1/23.9	0.09 sec.	—
	SNI	28.0/25.3	0.28 sec.	—
7	NI	32.7/28.7	2.43 sec.	—
	SNI	33.6/30.6	11.70 sec.	—
8	NI	38.5/33.7	1 min. 17 sec.	7.43 sec.
	SNI	40.3/36.3	9 min. 28 sec.	47.0 sec
9	NI	45.6/40.5	2 h. 18 min.	14 min. 20 sec.
	SNI	46.3/41.6	6 h. 30 min.	33 min. 20 sec.
10	NI	52.6/47.1	9 days 3h.	22 h. 30 min.
	SNI	53.5/48.4	—	—

[19]This is somewhat slow compared to performance on the similar '6126. The reason is currently unclear, but might involve the different build environment and overall setup.

Table 6.3: Comparison with maskVerif [BBC+19] on `hpac`.

Order d		Wall time maskVerif (4 threads)	Wall time Our software (4 threads, filtered)
6	NI	2 min. 44 sec.	0.57 sec.
	SNI	8 min. 11 sec.	1.48 sec.
7	NI	1 h. 39 min.	4.13 sec.
	SNI	5 h. 54 min.	15.60 sec.
8	NI	2 days 10h.	2 min. 15 sec.
	SNI	13 days 6h.	14 min. 35 sec.

6.4.2 SNI refreshing gadgets

We used our software to verify the SNI security of some (variations of) refreshing gadgets introduced in [BDF+17], and subsequently improved in [GPS+18; BBD+18]. As explained in Subsection 5.2.2, such schemes are useful when designing large circuits based on gadgets satisfying composable security notions since they help in providing strong security for the overall design. However, refreshing also comes with a significant cost in terms of randomness while not performing any sort of useful computation, leading several prior work to try finding new low-cost gadgets.

Our contribution here is an 8-share, 7-SNI refreshing gadget shown in Figure 6.3 that only needs 13 masks, which improves slightly on the best gadget from [BBD+18], which requires 16. Since such gadgets are used in the implementation of [GPS+18], it could again lead to actual practical gains.

We also compared the verification time of our tool with the one of maskVerif on the largest "RefreshZero" instances of [BBD+18], and actually have worse performance. For instance, even using 24 threads on the 12-core `retourdest`, verifying $\mathsf{RefreshZero}^{14}_{[1,3]}$ took us about 3 hours 40 minutes, while [BBD+18] reports an "Order of Magnitude" of 1 hour 30 minutes. We suspect this to be caused by the fact that there is no obvious probe filtering to be done on this sort of gadget, whereas maskVerif is likely able to successfully exploit their structure to reduce the number of attack sets to consider.

```
s00 r00 r01 r10 r20
s11 r01 r02 r11 r20
s22 r02 r03 r12 r20
s33 r03 r04 r13 r20
s44 r04 r05 r10
s55 r05 r06 r11
s66 r06 r07 r12
s77 r07 r00 r13
```

Figure 6.3: 7-SNI refreshing gadget, using 13 random masks.

6.4.3 Glitch-resistant NI multiplication

We conclude with a brief application to the *DOM-indep* family of multiplication gadgets introduced by Groß *et al.* [GMK16]. While those schemes are not more efficient than the state-of-the-art in terms of randomness cost, their main advantage is their resistance to glitches. A description of an instantiation at order 2 can be found in Subsection 6.3.3, and at any order less than 5 at `https://github.com/NicsTr/binary_masking`.

These gadgets can be instantiated at an arbitrary order d but do not come with a generic security proof guaranteeing the security of the result. We then have used our implementation to verify that instantiations up to order 5 are NI in the robust probing model. The running times on `retourdest` are summarised in Table 6.4. At order 1 and 2 both verification are almost instantaneously. However, our software is able to also verify the gadget at order 3, 4 and 5 in less than a second while SILVER is taking 24 seconds at order 3 and does not provide proof for higher orders.

Table 6.4: Comparison with SILVER [KSM20] for the NI verification of *DOM-indep* [GMK16] schemes in the robust probing model.

Order d	Wall time SILVER [KSM20]	Wall time Our software (1 thread, on `retourdest`)
1	$<$ 0.01 sec.	$<$ 0.01 sec.
2	$<$ 0.01 sec.	$<$ 0.01 sec.
3	24.4 sec.	$<$ 0.01 sec.
4	—	0.12 sec.
5	—	2 min. 22 sec.

Chapter 7

Masked implementations in practice

Contents

Overview

In this chapter, we follow the generic masking approach presented in Chapter 5 to implement a masked version of the AES. To do so, we use the previously designed and proven multiplication gadgets of Chapter 6. We discuss the choices made during the implementation and review different ways to implement the nonlinear layer of the AES. This results in an implementation publicly available at https://github.com/NicsTr/better_cortexm. Finally, we apply experimental methodologies to assess the security of our implementation by making concrete leakage measurements during the execution on a development board.

7.1 Introduction to concrete implementations of masking schemes

7.1.1 Which hardware target?

As explained in Chapter 4, the devices that are the most vulnerable against side-channel attacks are embedded ones. In this context, the processor used often implements the ARM architecture. It is thus natural to consider implementing a masking scheme specifically for this architecture. Following Schwabe and Stoffelen's work [SS16], we will further focus on one of the most popular modern microprocessor family in this context: the ARM Cortex-M4.[1] This family of microprecessors implements the ARMv7E-M architecture which provides all the Thumb-1, Thumb-2 and Digital Signal Processing (DSP) instructions. Unlike the higher-end Cortex-A family of microprocessors (used for example in [GPS+18]), it does not implement the NEON Single Instruction Multiple Data (SIMD) instructions. This means that our implementation will not use these powerful vectorised instructions to instead target cheaper processors for cheaper applications. However, we will see in Section 7.2 that optimizations aimed at using as much as possible the 32-bit registers are strongly related to vectorization and are also possible on ARM Cortex-M4.

More specifically, our implementation's test target is an STM32L432 Nucleo development board[2] (see Figure 7.1) that embeds an ARM Cortex-M4 processor, 256 kilobytes of flash memory and 64 kilobytes of SRAM. The processor has an embedded Random Number Generator (RNG) using ring oscillators to generate 32-bit random output. Using ST environment, we are able to easily program the board using STLINK[3] through USB. Also, an open-source low-level library called `libopencm3`[4] is used to ease the development of bare-metal programs in C.

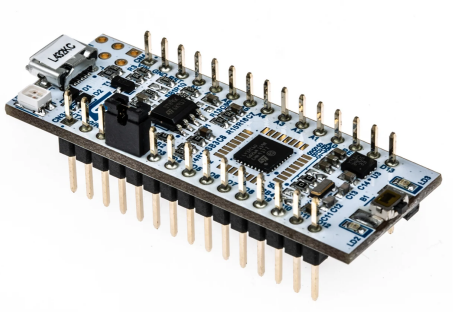


Figure 7.1: STM32L432 Nucleo development board

We explain in Section 7.4 how the board is instrumented in order to perform the experimental leakage assessment.

[1] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4
[2] https://www.st.com/en/evaluation-tools/nucleo-l432kc.html
[3] https://www.st.com/en/development-tools/st-link-v2.html
[4] https://libopencm3.org/

7.1.2 Which security models?

In the 2000's, many attempts have been made to design ad hoc masked implementations and "provably secure" ones [PGA06; SP06; RP10; BFG+12]. However, many were found to be vulnerable to side-channel attacks a few years after their publication [CGP+08; CPR07; CPR+13; PRR14].

In the light of those attempts, we will use the more recent compositional security models introduced by Barthe *et al.* [BBD+16]. As shown before, these models's goal is specifically turned toward the design of complex masked circuits given only elementary gadgets. The role of these models is to ensure that, under reasonable assumptions, the security of the composition is naturally derived from the security of the elementary gadgets and from compositional rules.

In order to have more confidence in the security of the implementation in practice, we explain in Section 7.4 how we performed an experimental leakage assessment using current state-of-the-art methodologies and we show the effect of masking on a concrete implementation. Since masking's ultimate goal is to amplify noise during the critical computations by introducing random values, the effect of masking in practice will be demonstrated by comparing the leakages while random gates in the masked circuit are implemented using an embedded TRNG or while they are implemented with deterministic values.

7.1.3 Which masking order?

Using the generic approach for masking a full algorithm, we need first to choose the elementary gadgets that will implement the elementary gates and especially the masked AND gate. The choice of the masking order at which these gates are instantiated is crucial for both security and performance since it has a quadratic impact on both. The need for implementation masked at order greater than one has been recently highlighted by the practical attack of Bronchain and Standaert [BS20] on an implementation of the AES at order one.

The most recent and optimized implementations using the same generic approach as ours [JS17; GJR+18] chose an order such that the number of shares is a power of two. Their choice was made because they are based on a specific implementation approach (described in Subsection 7.2.4) that provides the best performance when the size of the registers in bit is a multiple of the number of shares. To allow for a fair comparison between our implementation and the state of the art, we chose to also use an order where the number of shares is a power of two.

Even if there exists masked AND gadgets, that are formally proven to be secure in composable models at any order (such as ISW [ISW03]), we chose to use gadgets that give better performance while still being proved secure. To do so, we refer to the gadgets analysed by our tool presented in Chapter 6.

With these constraints in mind, we provide two masked implementations: one at order 3 (four shares) and one at order 7 (eight shares). During the compilation of the library, the user must choose between these two orders. The other orders are left as a future works but only require implementing the gadgets in assembly, while the rest of the code base is generic.

7.1.4 Which algorithm to implement?

Recently, symmetric block cipher proposals were made that were specifically designed to be masked, for example by reducing the nonlinear complexity of the algorithms. However, some of these new constructions were subject of cryptanalytic attacks. As of today, they are not yet widely used in concrete applications.

The Advanced Encryption Standard [oST01] is a block cipher standardized in 2001. It is based on Rijndael [DR98; DR20], which was designed in 1998 by Joan Daemen and Vincent Rijmen, with the additional constraints that blocks of the AES are 128-bit long and the key size must be either 128, 192 or 256 bits. It consists in the successive application of a round function composed of four operations: a round key addition, `AddRoundKey`; a layer of nonlinear components called S-boxes, `SubBytes`; a byte shuffle, `ShiftRows`; and finally a linear transformation on four bytes, `MixColumns`.

The AES is used in a wide range of applications, most commonly as a building block of symmetric encryption schemes. Implementations of the AES has been the target of heavy optimization efforts. These optimizations are often made to decrease its running time or its memory consumption without giving much attention to side-channel protections, which is relevant in many threat models where the physical access to the device is near impossible. However, for embedded devices, the threat model is different and side-channel attacks must not be overlooked. Some effort has been made toward masked implementation but mainly at a masking order of one.

Our goal is then to continue improving the efficiency of masked implementations of the AES, especially at higher orders, in the hope that it will help moving from ad hoc and specialized countermeasures against side-channel attacks toward formally secure and generic ones.

7.2 Different approaches to implement a masked AES S-Box

In both the original Rijndael reference [DR98] and its standardization as AES [DR20], the block state is described as an array of 16 elements in $\mathbb{F}_{2^8}$ stored as 16 bytes and the operations made on them use the same representation. Specifically, the AES S-Box is described for each byte as an inversion of the current element followed by a $\mathbb{F}_2$-linear transformation. Every other step in the AES being linear and are easily masked as shown in Section 5.4, the biggest differences between masked implementations are often found in the computation of the S-Box. There are mainly four popular approaches.

7.2.1 Table lookups

Table lookups is a popular way to implement the AES S-box, even in implementations that are not masked. This method consists in pre-computing tables to be used during the actual computation. In the case where there is no masking, this table contains the 256 different results of the S-Box applied to each of the 256 elements in $\mathbb{F}_{2^8}$. During the actual computation of the S-Box, the implementation just has to fetch the value at the offset corresponding to the byte for which we want to compute the S-Box. However, it may lead to cache-timing attacks when the implementation is running on a processor where a data cache is available [Ber05].

Since the table look-up depends on the value of the input, this cannot be done directly in a masked implementation even on a microprocessor without a cache: there would be a trivial single-probe side-channel attack because computing the right offset in the table is equivalent to unmasking the data. Thus, the precomputed tables must be built depending on the value of the first $d-1$ shares of the masked byte. However, doing so has a major drawback: it is very memory consuming when trying to protect against high-order attacks because it requires to pre-compute and store tables for every possible different values of the first $d-1$ shares.

7.2.2 Finite-field secure multiplication

The second approach is to consider the original description using an inversion in $\mathbb{F}_{2^8}$ and designing a masking gadget implementing this operation. In fact, since inverting an element in $\mathbb{F}_{2^8}$ is equivalent to raising it to the power 254, this can be done using multiplication and squaring gadgets, *e.g.* using the *square-and-multiply* algorithm. However, squaring in characteristic two is a linear operation and linear gadgets are much less costly than multiplication gadgets. Thus one aims to have the least possible number of multiplications which lead Rivain and Prouff to propose an algorithm [RP10, Algorithm 2] to do so in only four multiplications. Nonetheless, this computation must be done for each byte of the AES state independently and requires arithmetic in $\mathbb{F}_8$.

7.2.3 Bitslicing

In 1997 Biham introduced the idea of *bitslicing* as a new way to implement the Data Encryption Standard (DES) [Bih97]. In modern computers, including our target architecture, the instructions work on registers having a bit-width of more than 8. Instructions as `and` (bitwise AND) and `eor` (bitwise XOR) are actually computing in parallel as much binary operations as the bit-width of the registers considered. The main idea of Biham is to *"view a 64-bit processor as a SIMD computer with 64 one-bit processors"* [Bih97].

Instead of applying a (potentially complex) operation on one of the 16 bytes at a time, the idea is to slice each byte and see them as 8 16-bit words instead of 16 8-bit words, by gathering all the i-th bits in the i-th 16-bit word. The main difficulty is then to represent the operation as a series of bitwise operations such that each operation between the i-th and j-th bit will be made by executing only one instruction between the i-th and j-th word, thus doing the operation on the 16 bytes in parallel.

Using circuit minimization techniques, Boyar and Peralta proposed a binary circuit to compute the AES S-Box using 32 AND gates and 83 XOR/XNOR gates [BP10]. It was further improved to 32 AND gates and 81 XOR/XNOR gates [Per20]. For a masked implementation of the AES, it means that instead of using a masked circuit for the inversion in $\mathbb{F}_{2^8}$ and applying it to the 16 bytes of the state, masking gadgets for the binary AND are used to build the full circuit of the S-Box from Boyar and Peralta which will be executed on the 16 bytes in parallel thanks to bitslicing.

7.2.4 Shareslicing

Another approach close to bitslicing exists and has been introduced as a way to parallelize the computation of the multiplication gadgets defined by Barthe *et al.* in 2017 [BDF+17]. It consists in regrouping the shares of the same value inside the same register. As in the case of bitslicing, this allows to parallelize the computation of the masked gadget by using only the most common instructions. In Example 7.1, we show an example of how shareslicing is able to increase performance in the case of a multiplication gadget at order 3.

Example 7.1: Shareslicing

Figure 7.2 shows an example of a multiplication gadget from Barthe *et al.* at order 3.

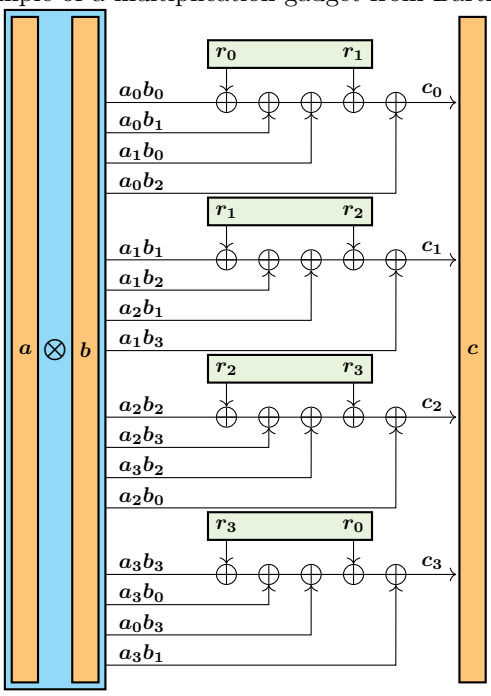


Figure 7.2: 3-NI multiplication scheme by Barthe *et al.* [BDF+17]

Given $\boldsymbol{a}$ and $\boldsymbol{b}$ the vectors representing the shares of a and the shares of b, and $\boldsymbol{r} = (r_0, r_1, r_2, r_3)$ the vector of the four additional random values, the naive approach would be to compute directly all the $a_i b_j$ and add them sequentially following the scheme. However, one can implement this scheme following Algorithm 5.

Algorithm 5: Sharesliced algorithm for 3-NI multiplication scheme by Barthe *et al.* [BDF+17]

Input : $\boldsymbol{a}, \boldsymbol{b}, \boldsymbol{r}$
Output: $\boldsymbol{c}$
$\boldsymbol{c} \leftarrow \boldsymbol{a} \wedge \boldsymbol{b}$
$\boldsymbol{c} \leftarrow \boldsymbol{c} \oplus \boldsymbol{r}$
$\boldsymbol{c} \leftarrow \boldsymbol{c} \oplus (\boldsymbol{a} \wedge \mathrm{ROR}(\boldsymbol{b}, 1))$
$\boldsymbol{c} \leftarrow \boldsymbol{c} \oplus (\mathrm{ROR}(\boldsymbol{a}, 1) \wedge \boldsymbol{b})$
$\boldsymbol{c} \leftarrow \boldsymbol{c} \oplus \mathrm{ROR}(\boldsymbol{r}, 1)$
$\boldsymbol{c} \leftarrow \boldsymbol{c} \oplus (\boldsymbol{a} \wedge \mathrm{ROR}(\boldsymbol{b}, 2))$

In this algorithm, $\mathrm{ROR}(\cdot, n)$ means rotating right the vector by n positions. Most modern processors have an instruction to perform these rotations (*e.g.*, `ror` on ARM or x86 processors). And, given that both $\boldsymbol{a}$, $\boldsymbol{b}$ and $\boldsymbol{r}$ are stored in the same register, the XOR and AND operations can also be made on the whole vectors with a single instruction as in the bitsliced approach. Thus, this scheme can be implemented on most processors using only 13 instructions.

The example seen in Example 7.1 can be easily extended to all the multiplication and mask-refreshing gadgets proposed by Barthe *et al.* [BDF+17]. This allows to gain significantly in efficiency as shown by Goudarzi *et al.* [GJR+18].

Furthermore, there can be additional performance gain by using architecture specific capabilities as the "flexible second operand" feature on Cortex-M4 processors. This feature allows to do a rotation of the second operand of some instructions for free. It uses a barrel shifter, which is a component made from multiple multiplexers stages. This component is inserted on the datapath before the component that actually computes the operation asked, *i.e.* the Arithmetic Logic Unit (ALU). The barrel shifter action occurs during the same cycle as the instruction itself and thus does not have an impact on the number of execution cycles. This feature has been used together with shareslicing by Journault and Standaert [JS17] in their implementation.

7.3 Implementation choices and performance

7.3.1 Masking the nonlinear layer of the AES

We first considered both shareslicing and bitslicing to implement the nonlinear layer of the AES. However, the security of shareslicing heavily relies on the assumption that there is no interaction between the bits of a register during the execution. In 2020, Gao *et al.* [GMP+20] have shown that this assumption does not always hold and they show an example with the effect of the barrel shifter on Cortex-M0 and Cortex-M3 processors. In light of the work of Gao *et al.* [GMP+20], since our target uses a Cortex-M4 processor and to avoid relying on this assumption, we choose of not using shareslicing techniques for our implementation. Additionally, to avoid unwanted effects due to bit interactions inside registers, we make sure that the shares of a given value are always stored in different registers throughout the execution.

Details on the bitsliced implementation. Our implementation uses the bitsliced approach to implement all 16 S-boxes in parallel as in the implementation by Goudarzi and Rivain [GR17]. Goudarzi and Rivain are using the 83-XOR/XNOR version of the circuit by Boyar and Peralta and mask-refreshing gadgets are regularly inserted to ensure correct composition of the masked gadgets. However, the work of Belaïd *et al.* [BGR18] has shown that the circuit remains probing

secure without them. For our implementation, we use the more recent 81-XOR/XNOR version of the circuit and to ensure that it is still secure from a compositional point of view, we used the same tool called TightPROVE.

In non-masked implementations, bitslicing techniques are often used together with n-bit, $n \geq 16$, registers to compute the AES encryption on $\frac{n}{16}$ blocks in parallel. In our case, $n = 32$ on Cortex-M4. However, during the computation of a masked implementation, each one of the 8 16-bit words of the bitsliced state is masked which means that each word is in fact a masked state of $d + 1$ 16-bit words. Each of the 32 AND gates of the Boyar-Peralta circuit must be replaced by a masked AND gadget working on 16 masked bits in parallel. Nonetheless, since our target is a microprocessor with 32-bit registers we still want to maximize the use of our registers. To do so, Goudarzi and Rivain [GR17] proposed to group the 32 AND gates by pairs allowing to replace each pair by a masked AND gadget working on 32 masked bits in parallel such that each share is 32-bit wide. For this to work properly, we must ensure that the output of one gate in the pair is not needed for the input of the other gate.

The exact circuit used in our implementation is thus a slightly-modified version of the 81-XOR/XNOR circuit as shown in Figure 7.3 where highlighted pairs of gates can be computed in parallel.

Masked gadgets for the AND gates. The exact gadget used for the AND gates depends on the order chosen at compile time for the implementation: at order 3, the 3-SNI gadget of [BDF+17] is used; at order 7, the new 7-SNI gadget from Subsection 6.4.1. The implementation of the parallel version of these masked gadgets is done directly in ARM assembly. To limit performance overhead due to memory access, we used the instruction scheduler and register allocator from the work of Schwabe and Stoffel [SS16; Sto16]. This tool helps reducing the number of store and load instructions used.

7.3.2 Masking the linear components of the AES

As seen in Section 5.4, thanks to the linearity of the additive secret-sharing used for masking the data the transformations of the linear layer can be applied share-wise which introduces much less overhead than for the gadgets for the nonlinear layers.

Since we are using a bitsliced approach to the computation of the AES S-Box, the inputs and outputs of the nonlinear layers are in a bitsliced representation. To avoid having to pay costly bit manipulations that are needed to go from and into a more standard representation where each byte of the block state is stored in a single byte in memory, the linear components (`MixColumns`, `ShiftRows` and `AddRoundKey`) are implemented using the same bitsliced representation. Our implementation reuses already existing bitsliced version of `MixColumns` [KS09; SS16] (see Figure 7.4) and `Shiftrows`[5].

7.3.3 Round function composition

During the design of the masked nonlinear layer, TightPROVE [BGR18] is used to ensure that the circuit for the S-Box is d-private. However, and as seen in Chapter 5, this is not sufficient for the composition of rounds to be d-private. Thus, the security of the full AES circuit is not formally proven, but only the nonlinear layer is.

We conjecture that the composition is nonetheless secure in the d-probing model but it is left as future work to prove it. We believe that it can be done, for example, by using TightPROVE on the circuit that includes every linear operations after the output of the last AND gates of the previous nonlinear layer and the circuit of the S-box itself.

Another approach to ensure that the full circuit is secure is to either: add mask-refreshing gadgets on one of the input for every masked AND as done by Goudarzi and Rivain [GR17] (there are 16×32 masked bits to refresh for each S-box layer); or refresh the whole AES state before every nonlinear layer (there are 16×8 masked bits to refresh for each S-box layer) such that

[5] Proposed in a GitHub Issue by Peter Dettman to improve the implementations presented in [SS16; AP21]

$$
\begin{array}{rrrr}
y_{14} = U_3 \oplus U_5 & y_4 = y_1 \oplus U_3 & y_{15} = t_1 \oplus U_5 & y_{17} = y_{10} \oplus y_{11} \\
y_{13} = U_0 \oplus U_6 & y_{12} = y_{13} \oplus y_{14} & y_{20} = t_1 \oplus U_1 & y_{19} = y_{10} \oplus y_8 \\
y_9 = U_0 \oplus U_3 & y_2 = y_1 \oplus U_0 & y_6 = y_{15} \oplus U_7 & y_{16} = t_0 \oplus y_{11} \\
y_8 = U_0 \oplus U_5 & y_5 = y_1 \oplus U_6 & y_{10} = y_{15} \oplus t_0 & y_{21} = y_{13} \oplus y_{16} \\
t_0 = U_1 \oplus U_2 & y_3 = y_5 \oplus y_8 & y_{11} = y_{20} \oplus y_9 & y_{18} = U_0 \oplus y_{16} \\
y_1 = t_0 \oplus U_7 & t_1 = U_4 \oplus y_{12} & y_7 = U_7 \oplus y_{11} &
\end{array}
$$

(a) Top linear transformation, U_i being the i-th bit of the input byte

$$
\begin{array}{llll}
t_2 = y_{12} \wedge y_{15} & t_{23} = t_{19} \oplus y_{21} & t_{34} = t_{23} \oplus t_{33} & z_4 = t_{40} \wedge y_1 \\
t_3 = y_3 \wedge y_6 & t_{26} = t_{21} \wedge t_{23} & t_{35} = t_{27} \oplus t_{33} & z_5 = t_{29} \wedge y_7 \\
t_4 = t_3 \oplus t_2 & t_{15} = y_8 \wedge y_{10} & z_2 = t_{33} \wedge U_7 & z_6 = t_{42} \wedge y_{11} \\
t_7 = y_{13} \wedge y_{16} & t_{16} = t_{15} \oplus t_{12} & t_{36} = t_{24} \wedge t_{35} & z_7 = t_{45} \wedge y_{17} \\
t_5 = y_4 \wedge U_7 & t_{18} = t_6 \oplus t_{16} & t_{37} = t_{36} \oplus t_{34} & z_8 = t_{41} \wedge y_{10} \\
t_6 = t_5 \oplus t_2 & t_{20} = t_{11} \oplus t_{16} & t_{38} = t_{27} \oplus t_{36} & z_9 = t_{44} \wedge y_{12} \\
t_{10} = y_2 \wedge y_7 & t_{22} = t_{18} \oplus y_{19} & t_{44} = t_{33} \oplus t_{37} & z_{10} = t_{37} \wedge y_3 \\
t_8 = y_5 \wedge y_1 & t_{24} = t_{20} \oplus y_{18} & z_0 = t_{44} \wedge y_{15} & z_{11} = t_{33} \wedge y_4 \\
t_9 = t_8 \oplus t_7 & t_{25} = t_{21} \oplus t_{22} & t_{39} = t_{29} \wedge t_{38} & z_{12} = t_{43} \wedge y_{13} \\
t_{11} = t_{10} \oplus t_7 & t_{27} = t_{24} \oplus t_{26} & t_{40} = t_{25} \oplus t_{39} & z_{13} = t_{40} \wedge y_5 \\
t_{12} = y_9 \wedge y_{11} & t_{30} = t_{23} \oplus t_{24} & t_{41} = t_{40} \oplus t_{37} & z_{14} = t_{29} \wedge y_2 \\
t_{13} = y_{14} \wedge y_{17} & t_{31} = t_{22} \oplus t_{26} & t_{42} = t_{29} \oplus t_{33} & z_{15} = t_{42} \wedge y_9 \\
t_{14} = t_{13} \oplus t_{12} & t_{32} = t_{31} \wedge t_{30} & t_{43} = t_{29} \oplus t_{40} & z_{16} = t_{45} \wedge y_{14} \\
t_{17} = t_4 \oplus y_{20} & t_{28} = t_{25} \wedge t_{27} & t_{45} = t_{42} \oplus t_{41} & z_{17} = t_{41} \wedge y_8 \\
t_{19} = t_9 \oplus t_{14} & t_{29} = t_{28} \oplus t_{22} & z_1 = t_{37} \wedge y_6 & \\
t_{21} = t_{17} \oplus t_{14} & t_{33} = t_{32} \oplus t_{24} & z_3 = t_{43} \wedge y_{16} &
\end{array}
$$

(b) Middle nonlinear transformation with pairs of AND gates highlighted

$$
\begin{array}{rrrr}
tc_1 = z_{15} \oplus z_{16} & tc_8 = z_7 \oplus tc_6 & S_3 = tc_3 \oplus tc_{11} & S_0 = tc_3 \oplus tc_{16} \\
tc_2 = z_{10} \oplus tc_1 & tc_9 = z_8 \oplus tc_7 & tc_{16} = z_6 \oplus tc_8 & S_6 = \neg(tc_{10} \oplus tc_{18}) \\
tc_3 = z_9 \oplus tc_2 & tc_{10} = tc_8 \oplus tc_9 & tc_{17} = z_{14} \oplus tc_{10} & S_4 = tc_{14} \oplus S_3 \\
tc_4 = z_0 \oplus z_2 & tc_{11} = tc_6 \oplus tc_5 & tc_{18} = tc_{13} \oplus tc_{14} & S_1 = \neg(S_3 \oplus tc_{16}) \\
tc_5 = z_1 \oplus z_0 & tc_{12} = z_3 \oplus z_5 & S_7 = \neg(z_{12} \oplus tc_{18}) & tc_{26} = tc_{17} \oplus tc_{20} \\
tc_6 = z_3 \oplus z_4 & tc_{13} = z_{13} \oplus tc_1 & tc_{20} = z_{15} \oplus tc_{16} & S_2 = \neg(tc_{26} \oplus z_{17}) \\
tc_7 = z_{12} \oplus tc_4 & tc_{14} = tc_4 \oplus tc_{12} & tc_{21} = tc_2 \oplus z_{11} & S_5 = tc_{21} \oplus tc_{17}
\end{array}
$$

(c) Bottom linear transformation, S_i being the i-th bit of the output byte

Figure 7.3: AES S-Box binary circuit from Boyar and Peralta [BP10; Per20] with slight modifications of the order of operations to regroup AND gates by pair. 113 gates are needed, 32 AND ($\wedge$) and 81 XOR ($\oplus$) and 4 NOT ($\neg$).

$$
\begin{aligned}
R'_1 &= \left(R_2 \oplus R_2^{\gg 4}\right) \oplus R_1^{\gg 4} \oplus \left(R_1 \oplus R_1^{\gg 4}\right)^{\gg 8} \\
R'_2 &= \left(R_3 \oplus R_3^{\gg 4}\right) \oplus R_2^{\gg 4} \oplus \left(R_2 \oplus R_2^{\gg 4}\right)^{\gg 8} \\
R'_3 &= \left(R_4 \oplus R_4^{\gg 4}\right) \oplus R_3^{\gg 4} \oplus \left(R_3 \oplus R_3^{\gg 4}\right)^{\gg 8} \\
R'_4 &= \left(R_5 \oplus R_5^{\gg 4}\right) \oplus R_4^{\gg 4} \oplus \left(R_4 \oplus R_4^{\gg 4}\right)^{\gg 8} \oplus \left(R_1 \oplus R_1^{\gg 4}\right) \\
R'_5 &= \left(R_6 \oplus R_6^{\gg 4}\right) \oplus R_5^{\gg 4} \oplus \left(R_5 \oplus R_5^{\gg 4}\right)^{\gg 8} \oplus \left(R_1 \oplus R_1^{\gg 4}\right) \\
R'_6 &= \left(R_7 \oplus R_7^{\gg 4}\right) \oplus R_6^{\gg 4} \oplus \left(R_6 \oplus R_6^{\gg 4}\right)^{\gg 8} \\
R'_7 &= \left(R_8 \oplus R_8^{\gg 4}\right) \oplus R_7^{\gg 4} \oplus \left(R_7 \oplus R_7^{\gg 4}\right)^{\gg 8} \oplus \left(R_1 \oplus R_1^{\gg 4}\right) \\
R'_8 &= \left(R_1 \oplus R_1^{\gg 4}\right) \oplus R_8^{\gg 4} \oplus \left(R_8 \oplus R_8^{\gg 4}\right)^{\gg 8}
\end{aligned}
$$

Figure 7.4: Bitsliced `MixColumns` implementation from [KS09]. R_i is the 16-bit register containing the i-th bit of every byte and $R_i^{\gg j}$ refers to a rotation of j bits to the right.

every masked inputs are uniform and mutually independent, thus matching the requirement of TightPROVE's inputs. Both strategies would have a negative impact on the overall performance of the implementation but the new mask-refreshing gadgets presented in Chapter 6 could be used to reduce this overhead.

7.3.4 Code structure

For each masking order, we provide a C header file and a C static library exposing the following main functions:

— `mask_bitslice_state`, that is used to convert a 128 bits AES state block into its masked and bitsliced representation;

— `unmask_unbitslice_state`, that is used to convert a masked and bitsliced AES state block into 128-bit non-bitsliced unmasked representation;

— `masked_aes_keyschedule128`, that implements the AES-128 keyschedule for a masked and bitsliced 128 bits key;

— `masked_aes_encrypt128`, that implements the AES-128 encryption for a masked and bitsliced 128 bits state block given the masked and bitsliced round key from the keyschedule.

These main functions are using auxiliary functions. To improve performance by taking into account the fact that the target architecture is known and by leveraging on ARM architecture's specific features, the most critical components are written directly in assembly. Also, this approach gives more control on the final executable code, as there is no interpretation by a compiler. These components are written such that they are exposing one function and they are respecting the calling convention of ARM 32-bit. This way each assembly-written component can be seen as independent of the rest of the project and a change in the implementation of a component (*e.g.* to fix bugs, to improve performance, to refactor or to change the masking order) can be made without worrying about its impact on the rest of the project.

7.3.5 Randomness generation

Each gadget for the masked AND gate at order 3 (respectively 7) requires the generation of 5 (respectively 20) random masks. In our implementation, the functions `masked_aes_encrypt128`

and `masked_aes_keyschedule128` take as last parameter a function pointer to `rng_fill`, a function that is used to fill a buffer of arbitrary size with random values. This function must be provided by the user of the library.

For our specific environment, `rng_fill` is implemented using the Random Number Generator (RNG) embedded in the STM32L432. This RNG is based on ring oscillators and allows to generate 32-bit words. A specific control register in memory is updated by the RNG module to tell whether or not the random value is ready to be read for the data register in memory. It takes approximately 64 cycles to generate each 32-bit word. The performance impact of the RNG on the overall implementation is discussed in Subsection 7.3.6.

The embedded RNG is working asynchronously with the rest of the chip and thus it may induce unexpected and non-deterministic delays while waiting for new random values to be drawn. Thus, during the leakage assessment presented in Section 7.4, all needed random values are drawn beforehand and stored in a buffer. This buffer contains as many random values as needed by the function on which the assessment is conducted. This impacts negatively on the memory footprint of the implementation but has the major advantage that two consecutive runs of the same function are taking exactly the same number of cycle to execute, allowing the analysis of multiple traces without risking desynchronization between them. This approach to random values generation is thus advantageous for an attacker and detrimental to its performance (since the RNG is a device running parallel to the processor). It is done only for leakage assessment and not in the final implementation.

The experimental leakage assessment done in Section 7.4 uses this RNG (through `rng_fill`) but also uses a function `fake_rng_fill` that fills the buffer with deterministic values. As explained in Section 7.1 and further discussed in Subsection 7.4.4.3, this allows to observe almost directly the impact of masking by comparing the effect of the RNG on the detection of leakages.

7.3.6 Performance

The performance of our implementation in the number of cycles by block encryption is presented in Table 7.1.

Table 7.1: Number of cycles to compute AES-128 keyschedule, encryption and S-Box, using either the embedded RNG or deterministic values.

Order d	3		7	
	With RNG	Without RNG	With RNG	Without RNG
AND gate	491	145	1771	497
S-Box	9353	3929	31025	10721
Keyschedule	100644	46393	323758	120716
Encryption	102442	48203	327383	124343

As a comparison with an unprotected AES optimized for the same platform, Schwabe and Stoffelen report that their implementation is computing a single block encryption in 661.7 cycles [SS16]. In the same paper, they proposed an implementation masked at order 1 that takes 7422 cycles by using an RNG that outputs a 32-bit every 40 cycles while the RNG we are using for our implementation is slower (64 cycles for each 32-bit words).

For masked implementations at higher orders, we do not directly compare to the one by Journault and Standaert [JS17] since they use masking schemes only at order 32. Also, we do not compare to the implementation by Grégoire *et al.* [GPS+18] because it is targeted at ARM Cortex-A, a higher end family of processors with vectorized instructions. Instead, we compare our implementation to the one of Goudarzi and Rivain [GR17] which has the same target (ARM Cortex-M) and the same range of masking order. They report that their implementation uses $3280(d+1)^2+14075(d+1)+12192$ cycles to compute an AES encryption where d is the masking

order, including the cost of randomness generation. However, they use a RNG that is even faster than the one used by Schwabe and Stoffelen: it is able to produce a 32-bit random value every 10 cycles.

Taking RNG performance into account. As seen in Table 7.1 and also reported in almost every implementations [SS16; GR17; JS17; GJR+18], the performance of the RNG itself is crucial to be taken into account. In fact, in our implementation the cost of randomness generation is taking around 53% (respectively 62%) of the total number of cycles needed for a block encryption at order 3 (respectively order 7). Indeed, this impact strongly depends on the efficiency of the RNG used. Thus, to achieve a fair comparison, we approximate the cost of generating the needed random values parametrized by n_{RNG}, the number of clock cycles taken by the RNG to output a 32-bit random word.

At order 3, each S-Box is using 32 AND gates and each bitsliced AND gate on 16-bit operands uses 16×5 random bits. At order 7, each bitsliced AND gates on 16-bit operands uses 16×20 random bits. For each nonlinear layer of the AES, 32 AND gates on 16-bit operands are computed. Thus, the overall randomness usage of an S-Box computation is $32 \times 16 \times 5$ bits, that is 80 32-bit words at order 3 and $32 \times 16 \times 20$ bits, that is 320 32-bit words at order 7. In AES-128, there are 10 nonlinear layers to apply to the state, which means that at order 3 (respectively 7) our implementation needs $80 \times 10 = 800$ (respectively $320 \times 10 = 3200$) 32-bit words of randomness. Then, the cost of randomness generation for a full block encryption takes approximately $800n_{\text{RNG}}$ at order 3 and $3200n_{\text{RNG}}$ at order 7.

The implementation of Schwabe and Stoffelen [SS16] at order 1 is using $n_{\text{RNG}} = 40$. With the same value for n_{RNG}, our implementation would take approximately $48203 + 800n_{\text{RNG}} = 80203$ cycles to run at order 3 and $124343 + 3200n_{\text{RNG}} = 252343$ at order 7.

The implementation of Goudarzi and Rivain [GR17] is using $n_{\text{RNG}} = 10$ and takes 120972 cycles to run at order 3 and 334712 cycles at order 7. With the same value for n_{RNG}, our implementation would take approximately $48203 + 800n_{\text{RNG}} = 56203$ cycles at order 3 and $124343 + 3200n_{\text{RNG}} = 156343$ at order 7. This is an improvement of around 53% for both order 3 and 7.

7.4 Experimental leakage assessment

In this section, we explore different experimental methodologies and apply them to test the security of our implementation against side-channel attacks.

7.4.1 The Test Vector Leakage Assessment (TVLA)

The Test Vector Leakage Assessment (TVLA) is a procedure that consists in trying to detect statistically meaningful differences between two datasets containing traces measured during the computation on the device under test.

These two datasets may be, for example, one containing the traces measured during the encryption of a constant plaintext with a constant key while the second dataset contains the traces measured during the encryption of a random plaintext with the same key. This is called a *fixed vs. random* test. The two datasets may also be filled with measurements of traces during encryption of two fixed different plaintexts, thus performing a *fixed vs. fixed* test. Being able to statistically distinguish the two sets of traces would mean that what is leaked during the execution depends on the plaintext on which the encryption is performed. This can be seen as a necessary condition to the presence of attacks against the implementation but, the number of total traces analysed is crucial in experimental evaluation. It is not uncommon to start observing significant differences between the two datasets only after several hundred thousands traces measured when the environmental measurement noise is high or when some counter-measures are present.

Finally, to determine if the two datasets are distinguishable, the TVLA relies on an underlying statistical test at a chosen (Type I) error rate α, which is defined as the probability of concluding

that the datasets are statistically distinguishable and being wrong. In practice, the result of those statistical test is a so-called p-value which can then be compared to α: if $p < \alpha$ then the datasets are said to be distinguishable. In the literature, α is often equal to 10^{-5}.

Welch's t-test The underlying statistical test commonly used is a Welch's t-test done for each point in time. The common approach to aggregate every trace inside each dataset is to compute their mean and to try to distinguish this mean from the mean of the other dataset.

The Welch's t-test is producing a statistic t:

$$t = \frac{\mu_1 - \mu_2}{\sqrt{\frac{{s_1}^2}{n_1} + \frac{{s_2}^2}{n_2}}} ,$$

with μ_i the mean of the i-th dataset, s_i its variance and n_i its size. The greater the absolute value of t is, the more statistically significant the difference between the two datasets is. The associated p-value which can then be compared to the target error rate α, can be computed using the statistic t along with its associated *degree of freedom* ν defined as:

$$\nu = \frac{\left(\frac{{s_1}^2}{n_1} + \frac{{s_2}^2}{n_2}\right)^2}{\frac{\left(\frac{{s_1}^2}{n_1}\right)}{n_1 - 1} + \frac{\left(\frac{{s_2}^2}{n_2}\right)}{n_2 - 1}} .$$

In most of the concrete cases ν is big enough to allow an approximation of p such as ν is not needed anymore during the computation. Using this approximation, we have:

$$p = 2 \times \left(1 - \mathrm{CDF}_{\mathcal{N}(0,1)}(|t|)\right) ,$$

with $\mathrm{CDF}_{\mathcal{N}(0,1)}$ being the cumulative distribution function of the normal distribution of mean 0 and standard deviation 1.

In the original proposal of the TVLA, the two datasets are compared using a common threshold directly on the value of t. This threshold is fixed to ± 4.5 for every point in time. This means that the device under test is said to successfully pass the test (*i.e.* no leakage is found) if the maximum absolute value of t over all points in time is lower than 4.5. For a single point in time, this t-value of 4.5 corresponds to a p-value of $p \approx 7 \times 10^{-6}$. Thus, having a threshold at 4.5 is almost equivalent to an error rate of $\alpha = 10^{-5}$.

However, since this test is done independently once for each point in time, the effect of repeating the t-test must be taken into account for long traces. Thus, it has been proposed [DZD+17; BGG+14] to adapt this threshold to the actual length of the traces. At a given target error rate α, the corresponding target error rate at each point in time is given by

$$\alpha_{\mathrm{TH}} = 1 - (1 - \alpha)^{1/n_L} , \tag{7.1}$$

with and n_L the number of time samples. The threshold for the t-value is thus:

$$\mathrm{TH} = \mathrm{CDF}^{-1}_{\mathcal{N}(0,1)}\left(1 - \alpha_{\mathrm{TH}}/2\right) . \tag{7.2}$$

For instance in a setting with 10 000 time samples, the same threshold for the t-value as before, $\max(|t|) < 4.5$, corresponds in fact to an error rate of only $\alpha \approx 0.06$. To have an error rate of $\alpha = 10^{-5}$, one need to put the threshold at: $\max(|t|) < \mathrm{TH} = 6.1$.

Higher-order t-test At EUROCRYPT 2012, Moradi [Mor12] proposed to try to distinguish these datasets by using moments of higher order (variance, skewness, kurtosis, ...) instead of the variances. In 2015, Schneider and Moradi [SM15] introduced formulas to compute the moments of higher-order incrementally allowing to use each trace to update each moment before discarding them. This helps reducing the memory footprint of the analysis, especially when hundreds of thousand traces are used.

In 2017, Reparaz *et al.* [RGV17] claimed to have designed a new algorithm to compute the t-statistic of the Welch's t-test even faster than the approach by Schneider and Moradi. Their implementation as a C library called `libfastld` is not publicly available.

χ^2-test. Replacing the Welch's t-test by Pearson's χ^2-test has been proposed in 2015 as a complementary approach [MRS+18].

As for the t-test, this test aims at trying to distinguish the two datasets. However, it does not consists in comparing the means (or higher-order moments) of the observations but instead it works directly on the whole distribution. Nonetheless, this test also work at the granularity level of a single point in time.

First, the $n_1 + n_2$ observations are stored in a matrix $\boldsymbol{F}$ with two rows, one for each dataset, and where each of the c columns corresponds to a possible observation (or more generally an interval). From this matrix $\boldsymbol{F}$, one can compute an expected frequencies matrix $\boldsymbol{E}$ as

$$\boldsymbol{E}_{1,j} = \frac{(\boldsymbol{F}_{1,j} + \boldsymbol{F}_{2,j}) \sum_{i=1}^{c} \boldsymbol{F}_{1,k}}{n_1 + n_2}$$
$$\boldsymbol{E}_{2,j} = \frac{(\boldsymbol{F}_{1,j} + \boldsymbol{F}_{2,j}) \sum_{i=1}^{c} \boldsymbol{F}_{2,k}}{n_1 + n_2} .$$

Then the statistic x and degree of freedom ν associated with the χ^2-test are given by

$$x = \sum_{j=1}^{c} \frac{(\boldsymbol{F}_{1,j} - \boldsymbol{E}_{1,j})^2}{\boldsymbol{E}_{1,j}} + \sum_{j=1}^{c} \frac{(\boldsymbol{F}_{2,j} - \boldsymbol{E}_{2,j})^2}{\boldsymbol{E}_{2,j}} , \nu = c - 1 .$$

Finally, the p-value is given using the cumulative distribution function of the χ^2 distribution:

$$p = 1 - \mathrm{CDF}_{\chi^2}(x)$$

In the χ^2-test setting, there is no concept of moment of higher order as for the t-test since it does not use moments. In some case the χ^2-test can directly detect leakages that otherwise would need a higher-order analysis in the case of the t-test. We use this test as it was intended and described in the work of Moradi *et al.* [MRS+18], that is as a complementary approach to the t-test.

7.4.2 Multivariate analysis

In the TVLA methodology, both the t-test and the χ^2-test are conducted on each point in time independently. This is sometimes called a univariate (or vertical) analysis, in opposition with a multivariate (or horizontal) analysis where $n > 1$ points in time are considered simultaneously. In a multivariate setting, the distribution of subsets of n points are compared in order to distinguished between the fixed and the random dataset.

An univariate leakage assessment is more adapted to an implementation where multiple shares are manipulated in parallel as in hardware implementations of masked circuit. However, in software implementations the computation are done sequentially and different shares of the same value are manipulated in different clock cycle, except when the value in memory or in a register is overwritten with another value. Thus, a multivariate analysis is often needed to successfully achieve a side-channel attack on masked software implementation.

The major drawback of a multivariate analysis is its efficiency: when the number of datapoint of a single trace increases, the number of different subsets of n points grows exponentially. The goal of an attacker is then to find the so-called *Points of Interest* (PoI), which are a reduced set of points in time that are carrying the most information and are thus more susceptible to lead to an attack. Detecting those points is often the most challenging part, while the Welch's t-test or the χ^2-test can be adapted to the multivariate case by carefully choosing an aggregation function that is applied on the subsets of n points. In Subsection 7.4.4.3, we conduct a bivariate analysis where points of interest are found based on the fact that we have full control over the device under test.

7.4.3 Template attacks

Template attacks (sometimes called *profiling attacks*) are a family of side-channel attacks where the target device is attacked in two steps: in the profiling step, the attacker has total control over the device and makes side-channel measurements during the execution of the program on chosen plaintexts with chosen keys; in the attack step, the device is running on known plaintexts with an unknown key. Template attacks are based on building a reliable profile of the device thanks to controlled executions and to use this profile to match the behaviour of the device during the execution with the unknown key. The profiling step requires to have full access and control on a device which is as close as possible to the target device in term of leaking behaviour. For example, to attack an RFID tag using a template attack, the attacker can first the same tag model to build a reliable profile before applying the attack step to the target one. The number of traces needed in the first step is often much higher than in the second step. Thus, when differences between target and profiling devices are small enough that accurate profiles can be computed, the attack against an instance of the target device can require only a devastatingly low number of traces.

Recently, Bronchain and Standaert [BS21] published an article attacking the most recent masked implementations of the AES on Cortex-M processors using a combination of a profiling attack and a multivariate analysis. Their tool, SCALib, is published as a Python interface with Rust libraries and can be found at https://github.com/simple-crypto/SCALib.

7.4.4 Results

We assess experimentally the security of our implementation using the methodology and statistical tests seen in Subsection 7.4.1 before conducting a bivariate analysis. We choose our target error rate to be equal to $\alpha = 10^{-5}$. This error rate is indeed arbitrary but is chosen to match the one found in the literature. We use Equations (7.1) and (7.2) to compute the thresholds needed for the different statistical test.

7.4.4.1 Hardware bench

We monitor the target board and acquire traces using a ChipWhisperer Lite Capture board[6]. To do so, we configure and connect six of the 30 pins available on the STM32L432. Table 7.2 gives the configuration for each connected pin between the target board and the capture board.

Table 7.2: Connections between the capture board and the target board. Each line gives the function and the corresponding pin for each side. When relevant, the alternate function (AF) to configure is also given.

ChipWhisperer Lite capture board		STM32L432 Nucleo target board
Ground voltage (GND)	⟷	Ground voltage (GND)
Clock Input (HS1/I)	⟷	System clock output (PA8 with AF0)
Reset target trigger (nRST)	⟷	Self reset trigger (NRST)
Serial receiver (IO2)	⟷	Serial transmitter (PA9 with AF7)
Serial transmitter (IO1)	⟷	Serial receiver (PA10 with AF7)
Trigger capture (IO4)	⟷	Trigger capture (PA12)

Since masking is a generic counter-measure that is side-channel agnostic, we chose to measure leakages from the processor using an electromagnetic directional probe from Langer EMV-

[6] https://store.newae.com/chipwhisperer-lite-cw1173-two-part-version/

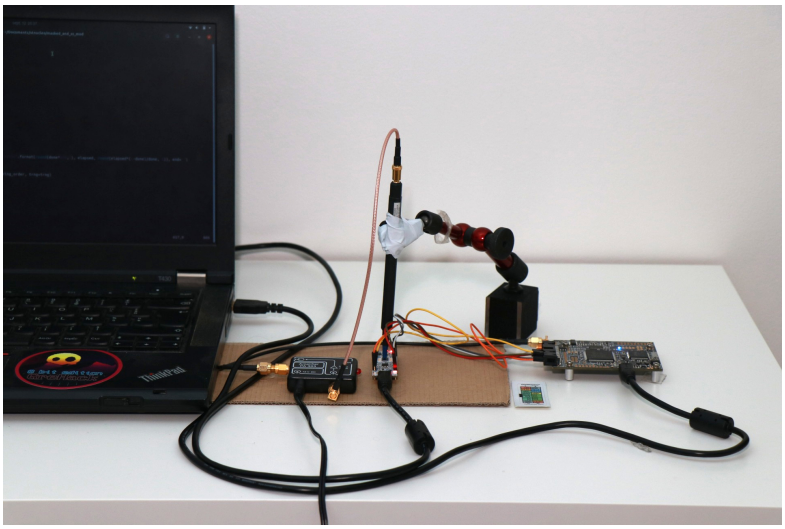


Figure 7.5: Picture of our hardware bench.

Technik[7]. This probe is connected to a pre-amplifier in order to amplify the signal before it reaches the capture board. This setup is illustrated in Figure 7.5.

Every experiment will be done twice: once using deterministic values in lieu of random masks (RNG off); and a second time using the integrated RNG of the STM32L432 described in Subsection 7.3.5 (RNG on) to generate these random masks. Doing so allows to directly compare the effect of masking on the leakages. It also helps distinguishing between points in time that are not depending on the processed data (*e.g.*, loop counter increment or unconditional jump) with points in time actually leaking information to the attacker. Thus, this can be used to get rid of many points before doing a multivariate analysis by keeping only the most information-carrying points, reducing the overall cost of the assessment. The exact steps used to reduce the cost of the multivariate analysis are explained in Subsection 7.4.2

The capture board used in our setup is limited to 24 000 time samples at once. This means that we are not able to make a measurement for each cycle during the full masked encryption for neither order 3 nor order 7, which takes respectively 48 203 and 124 343 cycles to execute. Instead, we assess the leakages that occurs during the computation of the S-boxes. As explained before in Subsection 7.3.5, we generate the additional random values needed in the function `masked_aes_sbox` (and its underlying component) beforehand. This allows to run the implementation at both order 3 (in less than 4000 cycles) and order 7 (in less than 11000 cycles).

7.4.4.2 TVLA application and bivariate analysis results

Welch's t-test We conduct a first order t-test on the application of the S-box layer protected with a masking scheme at order 3 and 7. To do so, we use the TVLA methodology by splitting traces in two datasets (fixed versus random) as explained in Subsection 7.4.1. We apply the test in the two settings: with deterministic or random masks. The threshold for the t-value is set to ± 6 for 4000 time samples and ± 6.1 for 11000 time samples.

When deterministic values are used in lieu of randomly drawn ones, we use 100 000 traces for each dataset which is sufficient to see clear leakage points. However, since we did not find any leakage when using random values, we stopped the assessment after 1 000 000 traces for each dataset which takes approximately 15h in our setup.

The result is shown in Figures 7.6 and 7.7. We can clearly see the impact of the Random Number Generator on the security of the implementation. When using deterministic values, the

[7] https://www.langer-emv.de/en/product/lf-passive-100-khz-up-to-50-mhz/36/lf-b-3-h-field-probe-100-khz-up-to-50-mhz/3

t-test is able to find clear differences between the computation on fixed plaintexts compared to the computation of random plaintexts. These leakages are not found when using the embedded RNG, even after recording 1 000 000 traces for each dataset.

χ^2-test We use the same TVLA but instead of using a t-test to distinguish the two datasets, we apply a χ^2-test as explained in Subsection 7.4.1. This test is applied on the same traces as the t-test previously shown, with the same rationale behind the choice of the total number of traces and the use of deterministic values or randomly generated ones as masks during the computation. The threshold for the p-value at each point in time is set to 2.5×10^{-9} for 4000 time samples and to 9×10^{-10} for 11000 time samples.

The result is shown in Figures 7.8 and 7.9 where the y axis is $-\log(p)$ for convenience. As for the t-test, the role of the RNG is visibly crucial. Compared with the t-test, the χ^2-test seems to be less sensitive but more precise: there are fewer points in time going above the threshold (183 against 879 for the t-test) but with higher contrast between non-leaking and leaking points.

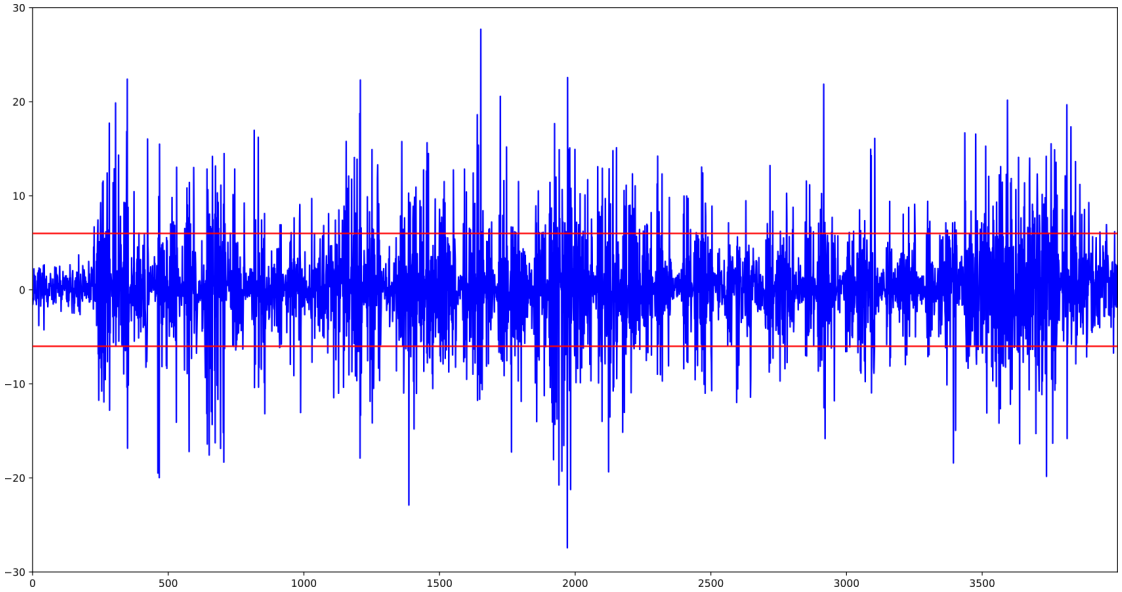
7.4.4.3 Bivariate analysis

We conduct a bivariate analysis using the χ^2-test on our implementation by following the methodology of Moradi *et al.* [MRS+18].

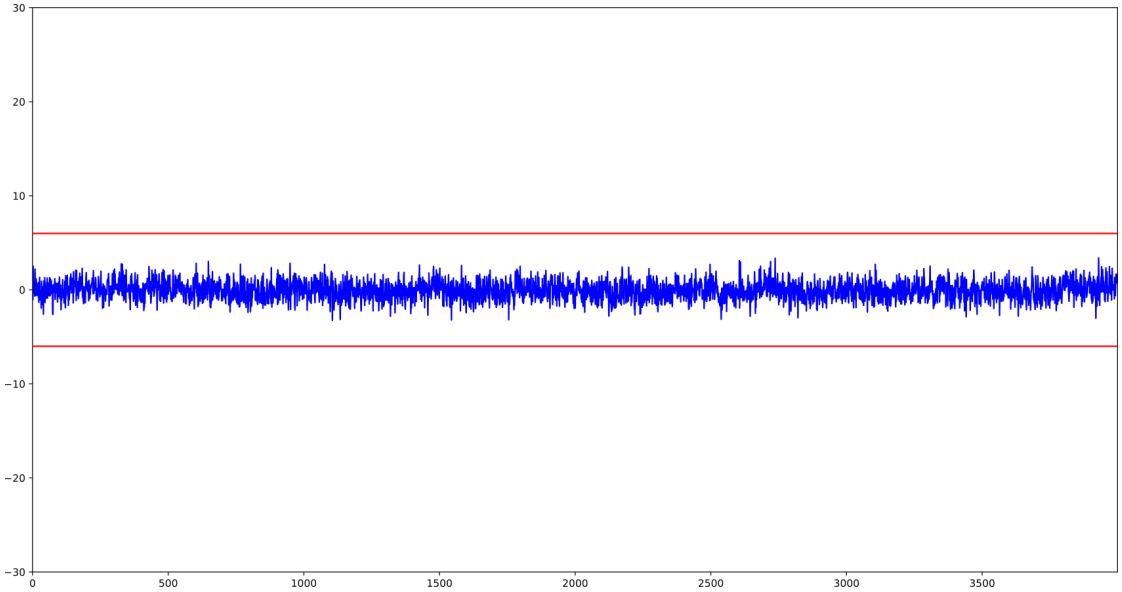
In order to limit the time and memory complexity of a bivariate analysis, we keep only the points in time that actually carry information. We do so by filtering out the ones that do not go above the threshold during the univariate χ^2 analysis on the implementation using deterministic masks. Thanks to this step, we are able to reduce the number of points from 4000 to 183 at order 3 and from 11000 to 585 at order 7.

The threshold for the p-value at each point in time is set to 5.9×10^{-10} at order 3 (for a total of $183(183+1)/2 = 16836$ points of analysis) and to 5.8×10^{-11} at order 7 (for a total of $585(585+1)/2 = 171405$ points of analysis).

The result of the bivariate analysis is shown in Figures 7.10 and 7.11. The colour of each pixel represents the p-value of the combination of the two points in time at the corresponding coordinate. The diagonal thus corresponds to the univariate analysis. Since our bivariate analysis relies on a symmetric combination function of each point, the resulting graph is also symmetric. p-values below the threshold are shown in levels of red and the ones that are below the threshold are in plain white. When deterministic masks are used, only 30 000 traces for each dataset are sufficient to detect clear differences between the two datasets whereas after 1 000 000 traces for each dataset, no bivariate leakage has been found for the implementation at both order 3 and 7 when the embedded RNG is used to generate the masks.

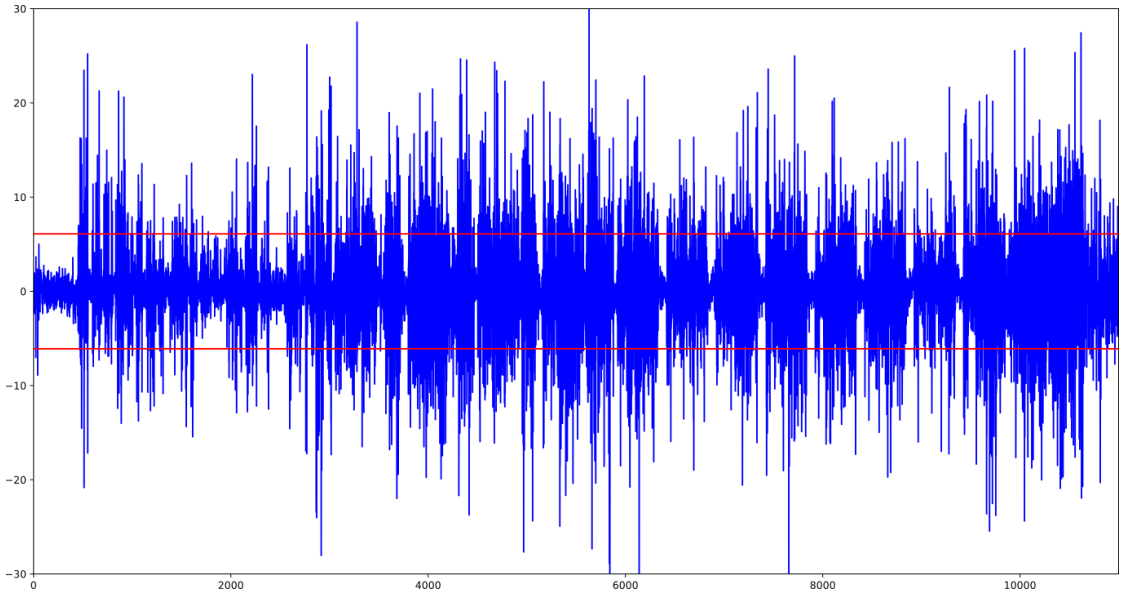


(a) t-test on $2 \times 100\,000$ traces (fixed vs. random) with deterministic masks on the implementation at order 3.

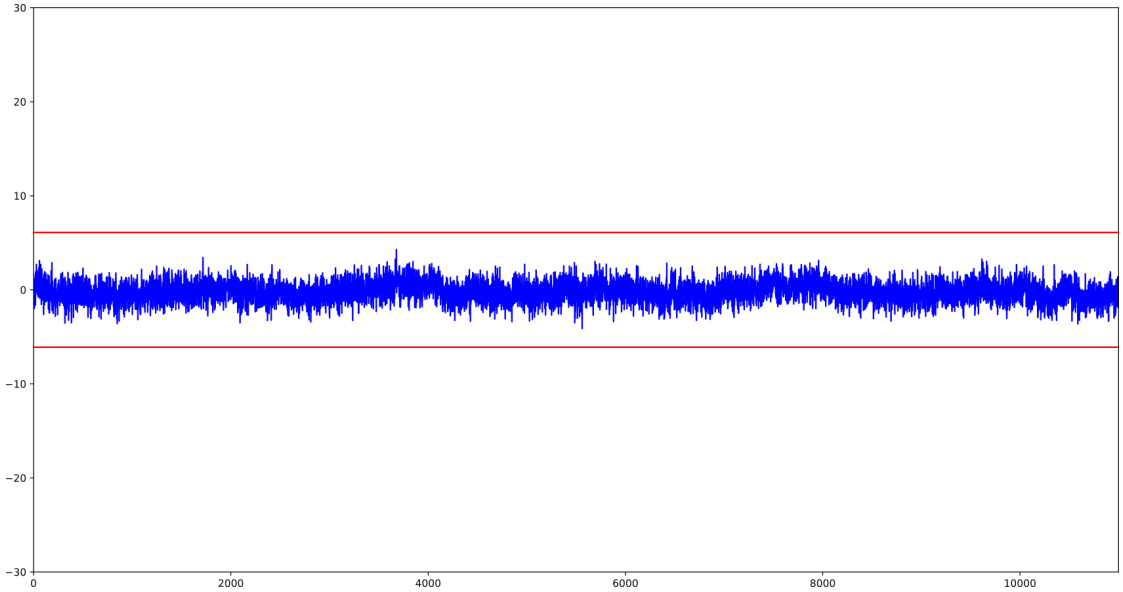


(b) t-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks on the implementation at order 3.

Figure 7.6: TVLA with Welch's t-test on the S-box layer masked at order 3.

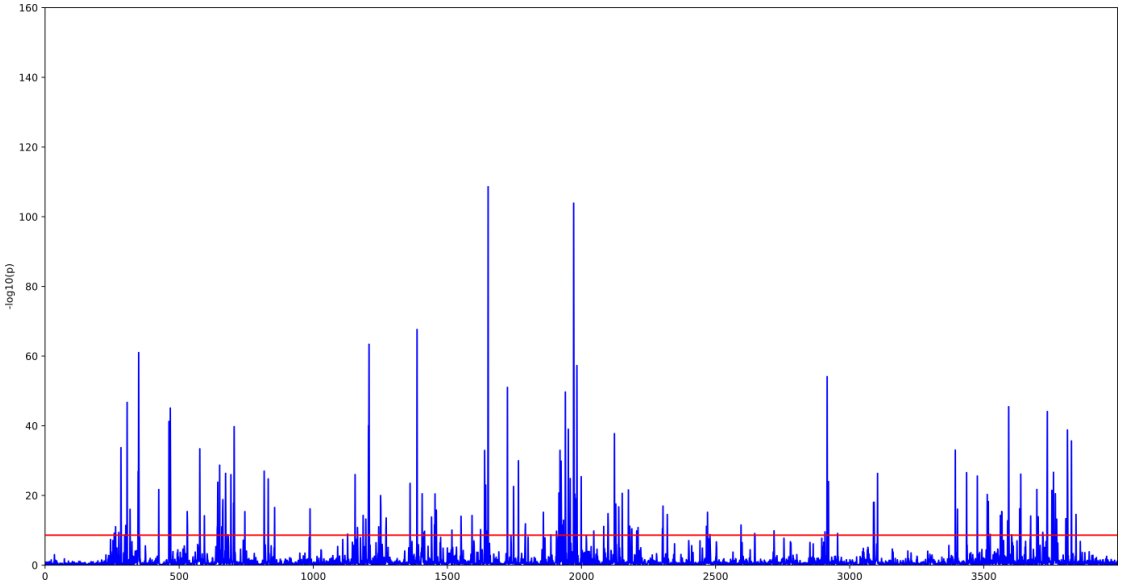


(a) t-test on $2 \times 100\,000$ traces (fixed vs. random) with deterministic masks on the implementation at order 7.

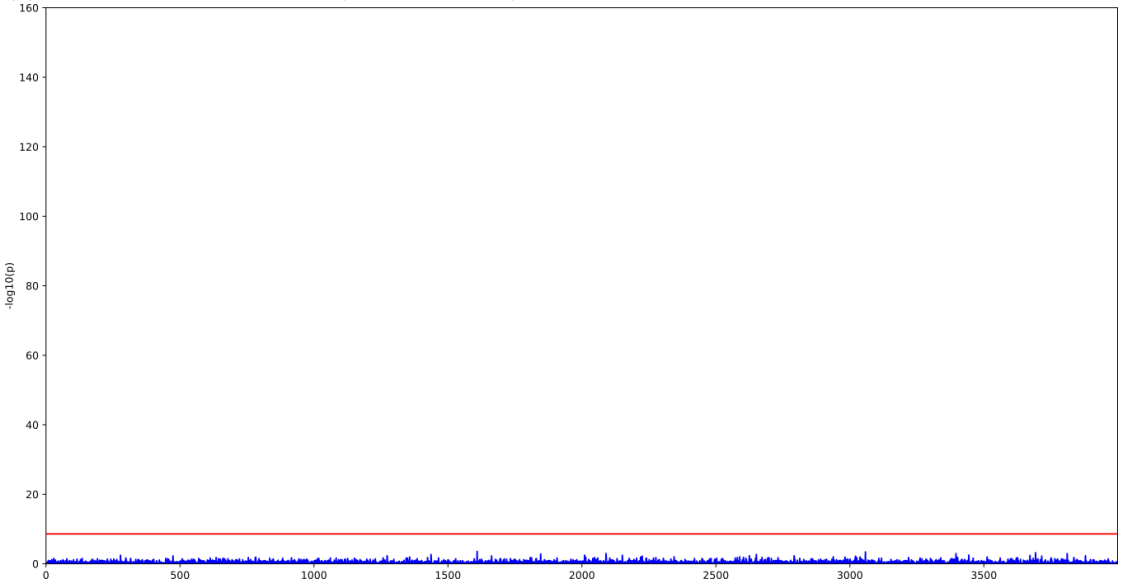


(b) t-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks on the implementation at order 7.

Figure 7.7: TVLA with Welch's t-test on the S-box layer masked at order 7.

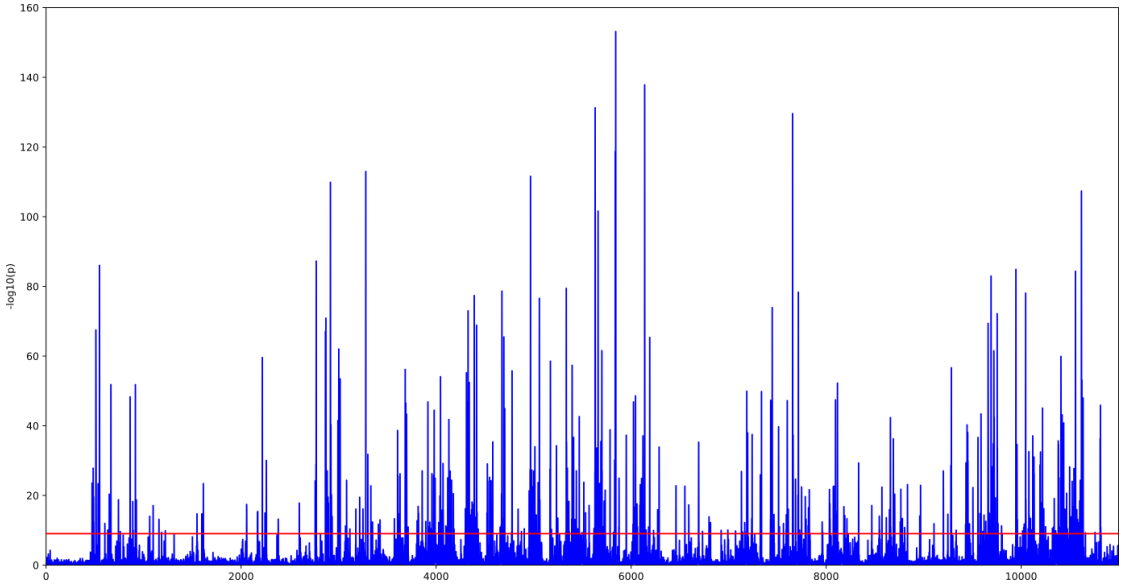


(a) χ^2-test on $2 \times 100\,000$ traces (fixed vs. random) with deterministic masks on the implementation at order 3.

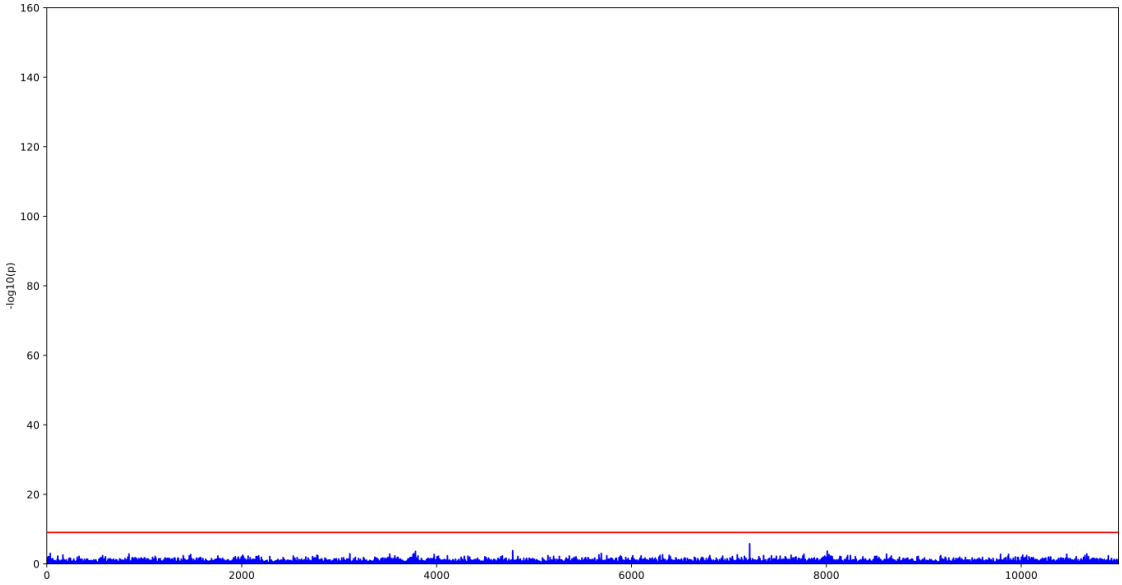


(b) χ^2-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks on the implementation at order 3.

Figure 7.8: TVLA with χ^2-test on the S-box layer masked at order 3.

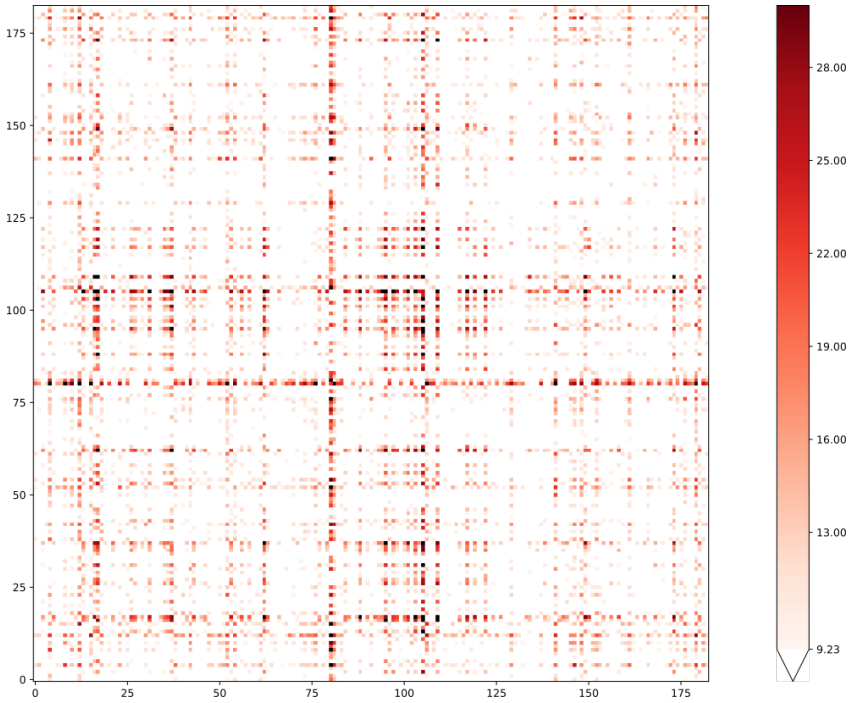


(a) χ^2-test on $2 \times 100\,000$ traces (fixed vs. random) with deterministic masks on the implementation at order 7.

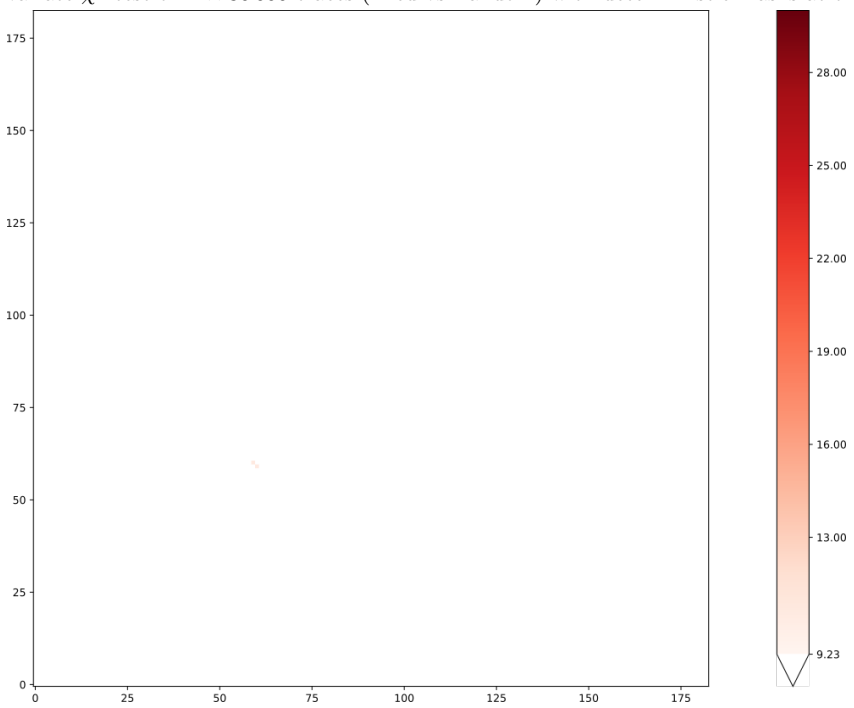


(b) χ^2-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks on the implementation at order 7.

Figure 7.9: TVLA with χ^2-test on the S-box layer masked at order 7.

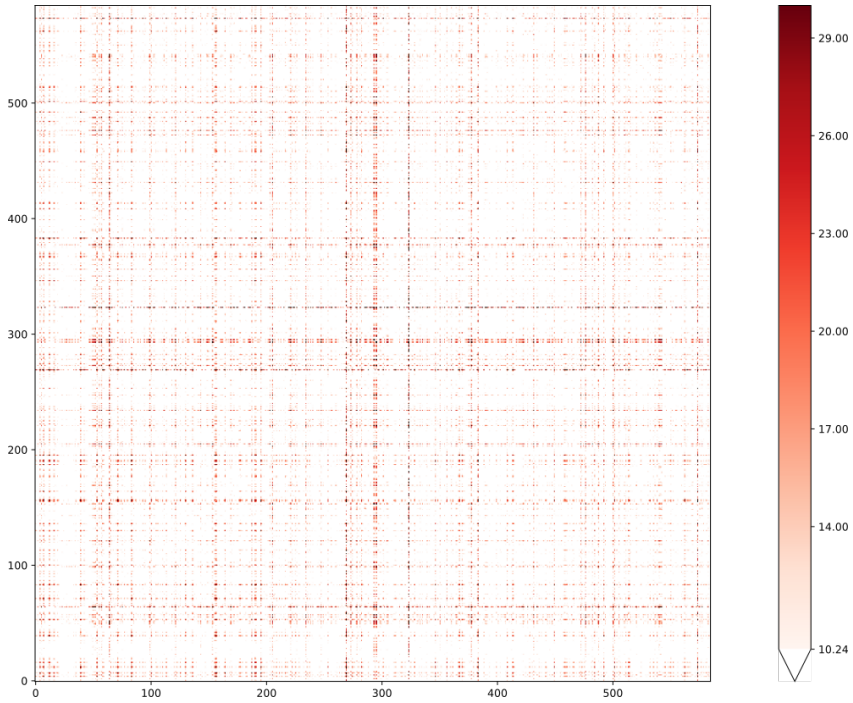


(a) Bivariate χ^2-test on $2 \times 30\,000$ traces (fixed vs. random) with deterministic masks at order 3.

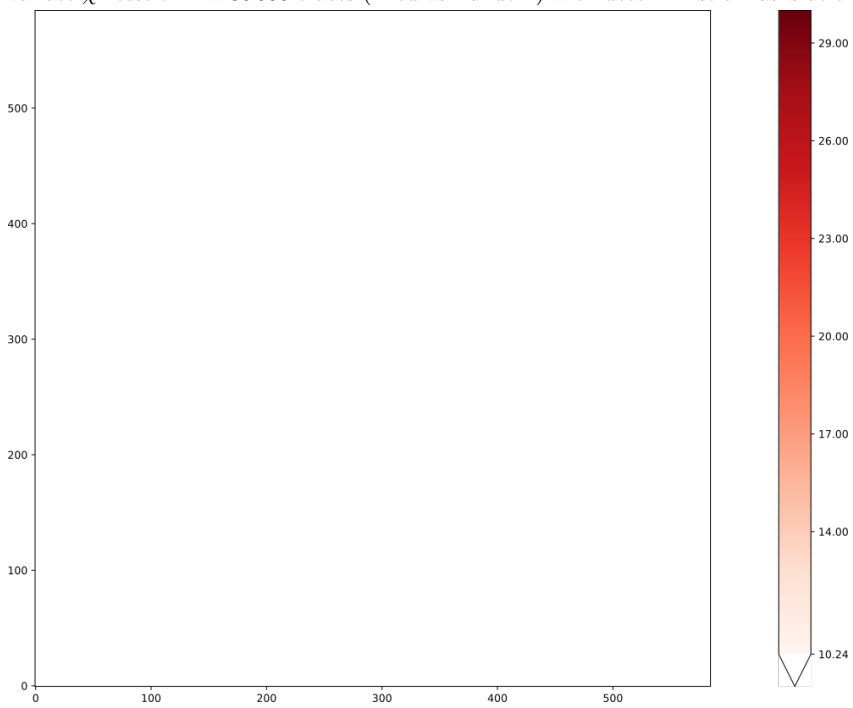


(b) Bivariate χ^2-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks at order 3.

Figure 7.10: Bivariate χ^2-test on the S-box layer masked at order 3 with 183 points of interest. p-values below the threshold are shown in levels of red, below threshold in white.



(a) Bivariate χ^2-test on $2 \times 30\,000$ traces (fixed vs. random) with deterministic masks at order 7.



(b) Bivariate χ^2-test on $2 \times 1\,000\,000$ traces (fixed vs. random) with random masks at order 7.

Figure 7.11: Bivariate χ^2-test on the S-box layer masked at order 7 with 585 points of interest. p-values below the threshold are shown in levels of red, below threshold in white.

Summary and future work

In this part of the thesis, we focused on a class of attacks that target specifically the implementations of cryptographic primitives: side-channel attacks. We described how masking can be used as a countermeasure against such attacks. After having presented different security models and how to design masked implementations in Chapter 5, we introduced in Chapter 6 a new algorithm to verify the security of masking schemes. This algorithm was implemented as a tool which is publicly available, and was used to design more efficient masking gadgets. Finally, we presented in Chapter 7 a secure implementation of the AES, also publicly available, and conducted an experimental assessment of the leakage that occurs when executing it giving more confidence in its security.

As future work, this experimental assessment could be made more complete by using different frameworks. For example, the recent tool by Bronchain and Standaert [BS21] called SCALib could be used to verify its security against more complex attacks, namely template and multivariate attacks. Additionally, due to hardware restrictions on the number of time samples, our analysis focused only on the security of the most critical component: the nonlinear layer. The use of a higher-end hardware such as the ChipWhisperer Pro could allow us to assess the full execution of the AES.

As pointed out in Section 7.3, only the nonlinear layer of this implementation is formally proven to be d-probing secure. To prove the conjecture that the full masked circuit is also secure, we could either make TightPROVE verify the full circuit or improve it to prove the security of circuits in composable models. We could also add mask-refreshing gadgets to our implementation between the application of two rounds.

Finally, our verification tool could be improved by adding support to the verification of gadgets in the PINI security model presented in Section 5.1. Even if d-PINI gadgets are in general costlier than d-SNI ones, we could use them to make an alternative version of our masked implementation of the AES, solving the problem of rounds composition altogether.

This page intentionally left blank.

List of Figures

List of examples

This page intentionally left blank.

Bibliography

[AP21] Alexandre Adomnicai and Thomas Peyrin. Fixslicing aes-like ciphers new bitsliced AES speed records on arm-cortex M and RISC-V. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(1):402–425, 2021 (page 93).

[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 Kazue Sako, Steve A. Schneider, and Peter Y. A. Ryan, editors, *Computer Security - ESORICS 2019 - 24th European Symposium on Research in Computer Security, Luxembourg, September 23-27, 2019, Proceedings, Part I*, volume 11735 of *Lecture Notes in Computer Science*, pages 300–318. Springer, 2019 (pages 72, 76, 83–85).

[BBD+15] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, and Benjamin Grégoire. Compositional Verification of Higher-Order Masking: Application to a Verifying Masking Compiler. *IACR Cryptology ePrint Archive*, 2015:506, 2015 (page 72).

[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 Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 116–129. ACM, 2016. ISBN: 978-1-4503-4139-4 (pages 54, 56–58, 89).

[BBD+18] Gilles Barthe, Sonia Belaïd, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, François-Xavier Standaert, and Pierre-Yves Strub. Improved Parallel Mask Refreshing Algorithms: Generic Solutions with Parametrized Non-Interference & Automated Optimizations. *IACR Cryptology ePrint Archive*, 2018:505, 2018 (pages 57, 79, 85).

[BBP+16] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Randomness complexity of private circuits for multiplication. In Marc Fischlin and Jean-Sébastien Coron, editors, *Advances in Cryptology - EUROCRYPT 2016 - 35th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Vienna, Austria, May 8-12, 2016, Proceedings, Part II*, volume 9666 of *Lecture Notes in Computer Science*, pages 616–648. Springer, 2016 (pages 56, 62, 72, 73, 80).

[BBP+17] Sonia Belaïd, Fabrice Benhamouda, Alain Passelègue, Emmanuel Prouff, Adrian Thillard, and Damien Vergnaud. Private multiplication over finite fields. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology - CRYPTO 2017 - 37th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 20-24, 2017, Proceedings, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 397–426. Springer, 2017 (pages 64, 65).

[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 Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 535–566, 2017 (pages 57, 64, 79–81, 85, 91–93).

[BDK+21] Nicolas Bordes, Joan Daemen, Daniël Kuijsters, and Gilles Van Assche. Thinking outside the superbox. In Tal Malkin and Chris Peikert, editors, *Advances in Cryptology - CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16-20, 2021, Proceedings, Part III*, volume 12827 of *Lecture Notes in Computer Science*, pages 337–367. Springer, 2021 (page 4).

[BDP+11] G. Bertoni, J. Daemen, M. Peeters, and G. Van Assche. Cryptographic sponge functions. Submission to NIST (Round 3), 2011 (page 6).

[Ber05] Daniel J. Bernstein. Cache-timing attacks on aes. April 2005. URL: https://cr.yp.to/antiforgery/cachetiming-20050414.pdf (page 90).

[Ber08] Daniel J. Bernstein. Chacha, a variant of salsa20. January 2008. URL: https://cr.yp.to/chacha/chacha-20080120.pdf (page 6).

[BFG+12] Josep Balasch, Sebastian Faust, Benedikt Gierlichs, and Ingrid Verbauwhede. Theory and practice of a leakage resilient masking scheme. In Xiaoyun Wang and Kazue Sako, editors, *Advances in Cryptology - ASIACRYPT 2012 - 18th International Conference on the Theory and Application of Cryptology and Information Security, Beijing, China, December 2-6, 2012. Proceedings*, volume 7658 of *Lecture Notes in Computer Science*, pages 758–775. Springer, 2012 (page 89).

[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 Marc Joye and Amir Moradi, editors, *Smart Card Research and Advanced Applications - 13th International Conference, CARDIS 2014, Paris, France, November 5-7, 2014. Revised Selected Papers*, volume 8968 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2014 (page 98).

[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 Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018*, volume 10821 of *Lecture Notes in Computer Science*, pages 321–353. Springer, 2018 (page 76).

[BGR18] Sonia Belaïd, Dahmun Goudarzi, and Matthieu Rivain. Tight private circuits: achieving probing security with the least refreshing. In Thomas Peyrin and Steven D. Galbraith, editors, *Advances in Cryptology - ASIACRYPT 2018 - 24th International Conference on the Theory and Application of Cryptology and Information Security, Brisbane, QLD, Australia, December 2-6, 2018, Proceedings, Part II*, volume 11273 of *Lecture Notes in Computer Science*, pages 343–372. Springer, 2018 (pages 56, 58, 92, 93).

[Bih97] Eli Biham. A fast new des implementation in software. In Eli Biham, editor, *Fast Software Encryption*, pages 260–272, Berlin, Heidelberg. Springer Berlin Heidelberg, 1997. ISBN: 978-3-540-69243-0 (page 91).

[BK21] Nicolas Bordes and Pierre Karpman. Fast verification of masking schemes in characteristic two. In Anne Canteaut and François-Xavier Standaert, editors, *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part II*, volume 12697 of *Lecture Notes in Computer Science*, pages 283–312. Springer, 2021 (pages 52, 64).

[BKL+11] Andrey Bogdanov, Miroslav Knezevic, Gregor Leander, Deniz Toz, Kerem Varici, and Ingrid Verbauwhede. SPONGENT: the design space of lightweight cryptographic hashing. *IACR Cryptol. ePrint Arch.*, 2011:697, 2011 (page 17).

[BP10] Joan Boyar and René Peralta. A new combinational logic minimization technique with applications to cryptology. In Paola Festa, editor, *Experimental Algorithms, 9th International Symposium, SEA 2010, Ischia Island, Naples, Italy, May 20-22, 2010. Proceedings*, volume 6049 of *Lecture Notes in Computer Science*, pages 178–189. Springer, 2010 (pages 91, 94).

[BS20] Olivier Bronchain and François-Xavier Standaert. Side-channel countermeasures' dissection and the limits of closed source security evaluations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(2):1–25, 2020 (page 89).

[BS21] Olivier Bronchain and François-Xavier Standaert. Breaking masked implementations with many shares on 32-bit software platforms or when the security order does not matter. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021(3):202–234, 2021 (pages 100, 109).

[BS90] Eli Biham and Adi Shamir. Differential cryptanalysis of des-like cryptosystems. In Alfred Menezes and Scott A. Vanstone, editors, *Advances in Cryptology - CRYPTO '90, 10th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1990, Proceedings*, volume 537 of *Lecture Notes in Computer Science*, pages 2–21. Springer, 1990 (page 8).

[CCD00] Christophe Clavier, Jean-Sébastien Coron, and Nora Dabbous. Differential Power Analysis in the Presence of Hardware Countermeasures. In *CHES 2000*, pages 252–263, Worcester, United States, August 2000 (page 49).

[CDL+20] Anne Canteaut, Sébastien Duval, Gaëtan Leurent, María Naya-Plasencia, Léo Perrin, Thomas Pornin, and André Schrottenloher. Saturnin: a suite of lightweight symmetric algorithms for post-quantum security. *IACR Transactions on Symmetric Cryptology*, 2020(S1):160–207, June 2020 (page 16).

[CDP17] Eleonora Cagli, Cécile Dumas, and Emmanuel Prouff. Convolutional neural networks with data augmentation against jitter-based countermeasures - profiling attacks without pre-processing. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 45–68. Springer, 2017 (page 49).

[CGP+08] Jean-Sébastien Coron, Christophe Giraud, Emmanuel Prouff, and Matthieu Rivain. Attack and improvement of a secure s-box calculation based on the fourier transform. In Elisabeth Oswald and Pankaj Rohatgi, editors, *Cryptographic Hardware and Embedded Systems - CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10-13, 2008. Proceedings*, volume 5154 of *Lecture Notes in Computer Science*, pages 1–14. Springer, 2008 (page 89).

[CJR+99] Suresh Chari, Charanjit S. Jutla, Josyula R. Rao, and Pankaj Rohatgi. Towards sound approaches to counteract power-analysis attacks. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 398–412. Springer, 1999 (pages 49, 53).

[CK09] Jean-Sébastien Coron and Ilya Kizhvatov. An efficient method for random delay generation in embedded software. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 156–170. Springer, 2009 (page 49).

[CK10] Jean-Sébastien Coron and Ilya Kizhvatov. Analysis and improvement of the random delay countermeasure of CHES 2009. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 95–109. Springer, 2010 (page 49).

[CPR+13] Jean-Sébastien Coron, Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. Higher-order side channel security and mask refreshing. In Shiho Moriai, editor, *Fast Software Encryption - 20th International Workshop, FSE 2013, Singapore, March 11-13, 2013. Revised Selected Papers*, volume 8424 of *Lecture Notes in Computer Science*, pages 410–424. Springer, 2013 (pages 54, 89).

[CPR07] Jean-Sébastien Coron, Emmanuel Prouff, and Matthieu Rivain. Side channel cryptanalysis of a higher order masking scheme. In Pascal Paillier and Ingrid Verbauwhede, editors, *Cryptographic Hardware and Embedded Systems - CHES 2007, 9th International Workshop, Vienna, Austria, September 10-13, 2007, Proceedings*, volume 4727 of *Lecture Notes in Computer Science*, pages 28–44. Springer, 2007 (page 89).

[CS20] Gaëtan Cassiers and François-Xavier Standaert. Trivially and efficiently composing masked gadgets with probe isolating non-interference. *IEEE Trans. Inf. Forensics Secur.*, 15:2542–2555, 2020 (page 58).

[Dae95] Joan Daemen. *Cipher and hash function design, strategies based on linear and differential cryptanalysis, PhD Thesis*. K.U.Leuven, 1995 (pages 13, 24).

[DDF14] Alexandre Duc, Stefan Dziembowski, and Sebastian Faust. Unifying leakage models: from probing attacks to noisy leakage. In Phong Q. Nguyen and Elisabeth Oswald, editors, *Advances in Cryptology - EUROCRYPT 2014 - 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings*, volume 8441 of *Lecture Notes in Computer Science*, pages 423–440. Springer, 2014 (page 54).

[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 Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015*, volume 9056 of *Lecture Notes in Computer Science*, pages 401–429. Springer, 2015. ISBN: 978-3-662-46799-2 (page 54).

[DHP+20] Joan Daemen, Seth Hoffert, Michaël Peeters, Gilles Van Assche, and Ronny Van Keer. Xoodyak, a lightweight cryptographic scheme. *IACR Trans. Symmetric Cryptol.*, 2020(S1):60–87, 2020 (page 38).

[DHV+18a] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. The design of Xoodoo and Xoofff. *IACR Trans. Symmetric Cryptol.*, 2018(4):1–38, 2018 (page 18).

[DHV+18b] Joan Daemen, Seth Hoffert, Gilles Van Assche, and Ronny Van Keer. XooTools. https://github.com/KeccakTeam/Xoodoo/tree/master/XooTools, 2018. Accessed: 02 October 2020 (page 40).

[DR06] Joan Daemen and Vincent Rijmen. Understanding two-round differentials in AES. In Roberto De Prisco and Moti Yung, editors, *Security and Cryptography for Networks, 5th International Conference, SCN 2006, Proceedings*, volume 4116 of *Lecture Notes in Computer Science*, pages 78–94. Springer, 2006 (page 36).

[DR07] Joan Daemen and Vincent Rijmen. Plateau characteristics. *IET Information Security*, 1(1):11–17, 2007 (pages 32, 36, 40, 41).

[DR20] Joan Daemen and Vincent Rijmen. *The Design of Rijndael - The Advanced Encryption Standard (AES), Second Edition*. Information Security and Cryptography. Springer, 2020. ISBN: 978-3-662-60768-8 (pages 16, 89, 90).

[DR98] Joan Daemen and Vincent Rijmen. The block cipher rijndael. In Jean-Jacques Quisquater and Bruce Schneier, editors, *Smart Card Research and Applications, This International Conference, CARDIS '98, Louvain-la-Neuve, Belgium, September 14-16, 1998, Proceedings*, volume 1820 of *Lecture Notes in Computer Science*, pages 277–284. Springer, 1998 (pages 89, 90).

[DRS+12] François Durvaux, Mathieu Renauld, François-Xavier Standaert, Loïc van Oldeneel tot Oldenzeel, and Nicolas Veyrat-Charvillon. Efficient removal of random delays from embedded software implementations using hidden markov models. In Stefan Mangard, editor, *Smart Card Research and Advanced Applications - 11th International Conference, CARDIS 2012, Graz, Austria, November 28-30, 2012, Revised Selected Papers*, volume 7771 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2012 (page 49).

[DZD+17] A. Adam Ding, Liwei Zhang, François Durvaux, François-Xavier Standaert, and Yunsi Fei. Towards sound and optimal leakage detection procedure. In Thomas Eisenbarth and Yannick Teglia, editors, *Smart Card Research and Advanced Applications - 16th International Conference, CARDIS 2017, Lugano, Switzerland, November 13-15, 2017, Revised Selected Papers*, volume 10728 of *Lecture Notes in Computer Science*, pages 105–122. Springer, 2017 (page 98).

[Fei73] Horst Feistel. Cryptography and computer privacy. *Scientific American*, 228(5):15–23, 1973. ISSN: 00368733, 19467087 (page 6).

[FG18] Junfeng Fan and Benedikt Gierlichs, editors. *Constructive Side-Channel Analysis and Secure Design — COSADE 2018*, volume 10815 of *Lecture Notes in Computer Science*, 2018. Springer. ISBN: 978-3-319-89640-3.

[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. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(3):89–120, 2018 (page 58).

[GGN+13] Benoît Gérard, Vincent Grosso, María Naya-Plasencia, and François-Xavier Standaert. Block ciphers that are easier to mask: how far can we go? In Guido Bertoni and Jean-Sébastien Coron, editors, *Cryptographic Hardware and Embedded Systems - CHES 2013 - 15th International Workshop, Santa Barbara, CA, USA, August 20-23, 2013. Proceedings*, volume 8086 of *Lecture Notes in Computer Science*, pages 383–399. Springer, 2013 (page 62).

[GJK+20] Dahmun Goudarzi, Jérémy Jean, Stefan Kölbl, Thomas Peyrin, Matthieu Rivain, Yu Sasaki, and Siang Meng Sim. Pyjamask: block cipher and authenticated encryption with highly efficient masked implementation. *IACR Trans. Symmetric Cryptol.*, 2020(S1):31–59, 2020 (page 62).

[GJR+18] Dahmun Goudarzi, Anthony Journault, Matthieu Rivain, and François-Xavier Standaert. Secure Multiplication for Bitslice Higher-Order Masking: Optimisation and Comparison. In [FG18], pages 3–22 (pages 58, 79, 89, 92, 97).

[GLS+14a] Vincent Grosso, Gaetan Leurent, Francois-Xavier Standaert, Kerem Varici, Francois Durvaux, Lubos Gaspar, and Stephanie Kerckhof. Scream & iscream. Entry in the CAESAR competition, available at `http://competitions.cr.yp.to/round1/screamv1.pdf`, March 2014 (page 62).

[GLS+14b] Vincent Grosso, Gaëtan Leurent, François-Xavier Standaert, and Kerem Varici. Ls-designs: bitslice encryption for efficient masked software implementations. In Carlos Cid and Christian Rechberger, editors, *Fast Software Encryption - 21st International Workshop, FSE 2014, London, UK, March 3-5, 2014. Revised Selected Papers*, volume 8540 of *Lecture Notes in Computer Science*, pages 18–37. Springer, 2014 (page 62).

[GM11] Tim Güneysu and Amir Moradi. Generic side-channel countermeasures for reconfigurable devices. In Bart Preneel and Tsuyoshi Takagi, editors, *Cryptographic Hardware and Embedded Systems – CHES 2011*, pages 33–48, Berlin, Heidelberg. Springer Berlin Heidelberg, 2011. ISBN: 978-3-642-23951-9 (page 49).

[GMK16] Hannes Groß, Stefan Mangard, and Thomas Korak. Domain-Oriented Masking: Compact Masked Hardware Implementations with Arbitrary Protection Order. In Begül Bilgin, Svetla Nikova, and Vincent Rijmen, editors, *ACM TIS@CCS 2016*, page 3. ACM, 2016. ISBN: 978-1-4503-4575-0 (pages 79, 85, 86).

[GMP+20] Si Gao, Ben Marshall, Dan Page, and Elisabeth Oswald. Share-slicing: Friend or Foe? *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(1):152–174, 2020 (pages 58, 92).

[GP99] Louis Goubin and Jacques Patarin. DES and differential power analysis (the "duplication" method). In Çetin Kaya Koç and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems, First International Workshop, CHES'99, Worcester, MA, USA, August 12-13, 1999, Proceedings*, volume 1717 of *Lecture Notes in Computer Science*, pages 158–172. Springer, 1999 (page 49).

[GPS+18] Benjamin Grégoire, Kostas Papagiannopoulos, Peter Schwabe, and Ko Stoffelen. Vectorizing Higher-Order Masking. In [FG18], pages 23–43 (pages 57, 80, 81, 85, 88, 96).

[GR17] Dahmun Goudarzi and Matthieu Rivain. How fast can higher-order masking be in software? In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology - EUROCRYPT 2017 - 36th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part I*, volume 10210 of *Lecture Notes in Computer Science*, pages 567–597, 2017 (pages 92, 93, 96, 97).

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part I*, volume 8616 of *Lecture Notes in Computer Science*, pages 444–461. Springer, 2014 (page 48).

[HP03] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2003. ISBN: 978-0-51180707-7 (pages 24, 25).

[ISW03] Yuval Ishai, Amit Sahai, and David Wagner. Private circuits: securing hardware against probing attacks. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003*, pages 463–481, Berlin, Heidelberg. Springer Berlin Heidelberg, 2003. ISBN: 978-3-540-45146-4 (pages 54, 55, 59, 62, 89).

[Jea15] Jérémy Jean. TikZ for Cryptographers. https://www.iacr.org/authors/tikz/, 2015 (page 7).

[JS17] Anthony Journault and François-Xavier Standaert. Very high order masking: efficient implementation and security evaluation. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 623–643. Springer, 2017 (pages 62, 80, 89, 92, 96, 97).

[KBG09] Najeh Masmoudi Kamoun, Lilian Bossuet, and Adel Ghazel. Correlated Power Noise Generator as a Low Costs DPA Countermeasures to Secure Hardware AES Cipher. In *Proceeding of the 3rd IEEE International Conference on Signals, Circuits and Systems, SCS 2009, pp. 1-6, Djerba, Tunisa, November 2009.* Pages 1–6, Tunisia, November 2009 (page 49).

[KJJ99] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999 (page 48).

[Knu11] Donald E. Knuth. *Combinatorial Algorithms, Part 1*, volume 4A of *The Art of Computer Programming*. Addison Wesley, 2011 (pages 77, 78).

[Knu94] Lars R. Knudsen. Truncated and higher order differentials. In Bart Preneel, editor, *FSE 1994*, volume 1008 of *Lecture Notes in Computer Science*, pages 196–211. Springer, 1994 (page 37).

[Koc96] Paul C. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996 (page 48).

[KPP20] Matthias J. Kannwischer, Peter Pessl, and Robert Primas. Single-trace attacks on keccak. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2020(3):243–268, 2020 (page 49).

[KR18] Pierre Karpman and Daniel S. Roche. New Instantiations of the CRYPTO 2017 Masking Schemes. In Thomas Peyrin and Steven D. Galbraith, editors, *ASIACRYPT 2018*, volume 11273 of *Lecture Notes in Computer Science*, pages 285–314. Springer, 2018. ISBN: 978-3-030-03328-6 (page 72).

[KS09] Emilia Käsper and Peter Schwabe. Faster and timing-attack resistant AES-GCM. In Christophe Clavier and Kris Gaj, editors, *Cryptographic Hardware and Embedded Systems - CHES 2009, 11th International Workshop, Lausanne, Switzerland, September 6-9, 2009, Proceedings*, volume 5747 of *Lecture Notes in Computer Science*, pages 1–17. Springer, 2009 (pages 93, 95).

[KSM20] David Knichel, Pascal Sasdrich, and Amir Moradi. SILVER - statistical independence and leakage verification. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Daejeon, South Korea, December 7-11, 2020, Proceedings, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 787–816. Springer, 2020 (pages 76, 86).

[Kui] Daniël Kuijsters. *Work in progress*. PhD thesis (pages 25, 35).

[LMR15] Gregor Leander, Brice Minaud, and Sondre Rønjom. A generic approach to invariant subspace attacks: cryptanalysis of robin, iscream and zorro. In Elisabeth Oswald and Marc Fischlin, editors, *Advances in Cryptology - EUROCRYPT 2015 - 34th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Sofia, Bulgaria, April 26-30, 2015, Proceedings, Part I*, volume 9056 of *Lecture Notes in Computer Science*, pages 254–283. Springer, 2015 (page 62).

[LR21] Victor Lomne and Thomas Roche. A Side Journey to Titan. https://ninjalab.io/wp-content/uploads/2021/01/a_side_journey_to_titan.pdf, 2021 (page 48).

[LT73] C. N. Liu and Donald T. Tang. Enumerating Combinations of m Out of n Objects [G6] (Algorithm 452). *Commun. ACM*, 16(8):485, 1973 (page 77).

[Mat93] Mitsuru Matsui. Linear cryptanalysis method for DES cipher. In Tor Helleseth, editor, *Advances in Cryptology - EUROCRYPT '93, Proceedings*, volume 765 of *Lecture Notes in Computer Science*, pages 386–397. Springer, 1993 (pages 12, 13).

[Mat96] Mitsuru Matsui. New structure of block ciphers with provable security against differential and linear cryptanalysis. In Dieter Gollmann, editor, *Fast Software Encryption, Third International Workshop, Cambridge, UK, February 21-23, 1996, Proceedings*, volume 1039 of *Lecture Notes in Computer Science*, pages 205–218. Springer, 1996 (page 6).

[Mes01] Thomas S. Messerges. Securing the aes finalists against power analysis attacks. In Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, and Bruce Schneier, editors, *Fast Software Encryption*, pages 150–164, Berlin, Heidelberg. Springer Berlin Heidelberg, 2001. ISBN: 978-3-540-44706-1 (page 49).

[Mor12] Amir Moradi. Statistical tools flavor side-channel collision attacks. In David Pointcheval and Thomas Johansson, editors, *Advances in Cryptology - EUROCRYPT 2012 - 31st Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cambridge, UK, April 15-19, 2012. Proceedings*, volume 7237 of *Lecture Notes in Computer Science*, pages 428–445. Springer, 2012 (page 98).

[MRS+18] Amir Moradi, Bastian Richter, Tobias Schneider, and François-Xavier Standaert. Leakage detection with the x2-test. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2018(1):209–237, 2018 (pages 99, 102).

[NW78] Albert Nijenhuis and Herbert S. Wilf. *Combinatorial algorithms for computers and calculators*. Academic Press, second edition, 1978 (page 77).

[oST01] National Institute of Standards and Technology. Advanced encryption standard. *NIST FIPS PUB 197*, 2001 (pages 16, 89).

[Per20] René Peralta. Circuit minimization work. January 2020. URL: https://www.cs.yale.edu/homes/peralta/CircuitStuff/CMT.html (visited on 07/21/2021) (pages 91, 94).

[PGA06] Emmanuel Prouff, Christophe Giraud, and Sébastien Aumônier. Provably secure s-box implementation based on fourier transform. In Louis Goubin and Mitsuru Matsui, editors, *Cryptographic Hardware and Embedded Systems - CHES 2006, 8th International Workshop, Yokohama, Japan, October 10-13, 2006, Proceedings*, volume 4249 of *Lecture Notes in Computer Science*, pages 216–230. Springer, 2006 (page 89).

[PR13] Emmanuel Prouff and Matthieu Rivain. Masking against side-channel attacks: A formal security proof. In Thomas Johansson and Phong Q. Nguyen, editors, *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*, volume 7881 of *Lecture Notes in Computer Science*, pages 142–159. Springer, 2013 (page 53).

[Pra62] Eugene Prange. The use of information sets in decoding cyclic codes. *IRE Trans. Information Theory*, 8(5):5–9, 1962 (page 73).

[PRC12] Gilles Piret, Thomas Roche, and Claude Carlet. PICARO - A block cipher allowing efficient higher-order side-channel resistance. In Feng Bao, Pierangela Samarati, and Jianying Zhou, editors, *Applied Cryptography and Network Security - 10th International Conference, ACNS 2012, Singapore, June 26-29, 2012. Proceedings*, volume 7341 of *Lecture Notes in Computer Science*, pages 311–328. Springer, 2012 (page 62).

[PRR14] Emmanuel Prouff, Matthieu Rivain, and Thomas Roche. On the practical security of a leakage resilient masking scheme. In Josh Benaloh, editor, *Topics in Cryptology - CT-RSA 2014 - The Cryptographer's Track at the RSA Conference 2014, San Francisco, CA, USA, February 25-28, 2014. Proceedings*, volume 8366 of *Lecture Notes in Computer Science*, pages 169–182. Springer, 2014 (page 89).

[PSC+02] Sangwoo Park, Soo Hak Sung, Seongtaek Chee, E-Joong Yoon, and Jongin Lim. On the security of rijndael-like structures against differential and linear cryptanalysis. In Yuliang Zheng, editor, *Advances in Cryptology - ASIACRYPT 2002, Proceedings*, volume 2501 of *Lecture Notes in Computer Science*, pages 176–191. Springer, 2002 (page 16).

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and counter-measures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001 (page 48).

[RGV17] Oscar Reparaz, Benedikt Gierlichs, and Ingrid Verbauwhede. Fast leakage assessment. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 387–399. Springer, 2017 (page 98).

[RP10] Matthieu Rivain and Emmanuel Prouff. Provably secure higher-order masking of AES. In Stefan Mangard and François-Xavier Standaert, editors, *Cryptographic Hardware and Embedded Systems, CHES 2010, 12th International Workshop, Santa Barbara, CA, USA, August 17-20, 2010. Proceedings*, volume 6225 of *Lecture Notes in Computer Science*, pages 413–427. Springer, 2010 (pages 89, 90).

[Sch80] Jacob T. Schwartz. Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, 27(4):701–717, 1980 (page 66).

[SM15] Tobias Schneider and Amir Moradi. Leakage assessment methodology - A clear roadmap for side-channel evaluations. In Tim Güneysu and Helena Handschuh, editors, *Cryptographic Hardware and Embedded Systems - CHES 2015 - 17th International Workshop, Saint-Malo, France, September 13-16, 2015, Proceedings*, volume 9293 of *Lecture Notes in Computer Science*, pages 495–513. Springer, 2015 (page 98).

[SP06] Kai Schramm and Christof Paar. Higher order masking of the AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 208–225. Springer, 2006 (page 89).

[SS16] Peter Schwabe and Ko Stoffelen. All the AES you need on cortex-m3 and M4. In Roberto Avanzi and Howard M. Heys, editors, *Selected Areas in Cryptography - SAC 2016 - 23rd International Conference, St. John's, NL, Canada, August 10-12, 2016, Revised Selected Papers*, volume 10532 of *Lecture Notes in Computer Science*, pages 180–194. Springer, 2016 (pages 88, 93, 96, 97).

[Sto16] Ko Stoffelen. Instruction scheduling and register allocation on arm cortex- m. In *Software performance enhancement for encryption and decryption, and benchmarking - SPEED-B*, 2016 (page 93).

[TG91] Anne Tardy-Corfdir and Henri Gilbert. A known plaintext attack of FEAL-4 and FEAL-6. In Joan Feigenbaum, editor, *Advances in Cryptology - CRYPTO '91, 11th Annual International Cryptology Conference, Santa Barbara, California, USA, August 11-15, 1991, Proceedings*, volume 576 of *Lecture Notes in Computer Science*, pages 172–181. Springer, 1991 (page 12).

[Wal] Timothy R. Walsh. A simple sequencing and ranking method that works on almost all gray codes. Unpublished research report. Available at: https://www.labunix.uqam.ca/~walsh_t/papers/sequencing_and_ranking.pdf (page 78).

[Wu11] Hongjun Wu. The hash function jh. Submission to NIST (round 3), 2011 (page 6).

Abstract / Résumé

The first part of this thesis is concerned with the study of some properties of cryptographic permutations. It takes its source from a joint work with Joan Daemen, Daniël Kuijsters and Gilles Van Assche published in the proceedings of CRYPTO 2021. These symmetric primitives can be designed using multiple approaches. One of them, popularized by the Advanced Encryption Standard (AES), consists in grouping the bits, *e.g.* in bytes, and consistently processing them in these groups. This *aligned* approach leads to structures that make it possible to reason about the differential and linear propagation properties using combinatorial arguments. In contrast, an *unaligned* approach avoids any such grouping in its design, which however complexifies the analysis of the same properties. In this thesis, we define formally what it means for a permutation to be aligned and study its impact on the differential and linear properties of four primitives adopting different design strategies.

The second part of this thesis focuses on the secure implementation of symmetric primitives. Specifically, we study a class of attacks, *side-channel attacks*, where an attacker may be able to extract the secrets of a cryptographic algorithm by only measuring physical leakages from the components computing it. One countermeasure against these attacks, called *masking*, leverages secret-sharing schemes to split the sensitive data into random shares while allowing to securely compute using this sharing. However, verifying that a masked implementation is indeed secure and the countermeasure itself are both costly. We improve the performance on this two aspects in an article published in the proceedings of EUROCRYPT 2021 with Pierre Karpman as a co-author. Following these results, we also propose a new masked version of the AES and we experimentally verify its robustness against side-channel attacks.

La première partie de cette thèse décrit certaines propriétés des permutations cryptographiques. Ce travail est issue d'une collaboration avec Joan Daemen, Daniël Kuijsters and Gilles Van Assche et est publié dans les actes de CRYPTO 2021. Ces primitives symétriques peuvent être construites en adoptant différentes approches. Une d'entre elles, popularisée par l'*Advanced Encryption Standard* (AES) consiste à regrouper les bits, par exemple en octets, et à procéder à des opérations uniquement à cette granularité. Cette approche, dite *alignée*, fait émerger une structure permettant de décrire les propriétés de propagation différentielle et linéaire en utilisant des arguments combinatoriaux. Au contraire, il est possible de concevoir des permutations de telle manière qu'un tel alignement n'existe pas, rendant néanmoins l'analyse différentielle et linéaire plus complexe. Dans cette thèse, nous définissons formellement cette propriété d'alignement et étudions son impact sur les propriétés différentielles et linéaires de quatre permutations ayant des constructions différentes.

La seconde partie de cette thèse se concentre sur l'implémentation sécurisée des primitives symétriques. Plus particulièrement, elle étudie une classe d'attaques, les *attaques par canaux auxiliaires*, permettant dans certaines conditions à un attaquant d'extraire les secrets manipulés par un algorithme cryptographique en mesurant les variations de grandeurs physiques lors de l'exécution de celui-ci. Un exemple de contre-mesure contre ce type d'attaques, le *masquage*, utilise un partage de secret pour répartir l'information à protéger en des parties individuellement indépendantes tout en permettant d'effectuer les calculs de manière sécurisée en utilisant ce partage. Néanmoins, vérifier qu'une implémentation est sécurité ainsi que la contremesure elle-même peuvent être très couteux. Nous améliorons les performances de l'état de l'art sur ces deux aspects dans un article co-écrit avec Pierre Karpman et publié dans les actes de EUROCRYPT 2021. Finalement, nous proposons une nouvelle version masquée de l'AES et nous évaluons sa robustesse contre les attaques par canaux auxiliaires.