Secure Implementation of Block Ciphers against Physical Attacks

Dahmun Goudarzi

THÈSE DE DOCTORAT

de l'Université de recherche Paris Sciences et Lettres
PSL Research University

Préparée à l'École normale supérieure

Secure Implementation of Block Ciphers against Physical Attacks

École doctorale n°386
Sciences Mathématiques de Paris Centre

Spécialité Informatique

Soutenue par
Dahmun Goudarzi
le 21 septembre 2018

Dirigée par
Matthieu Rivain et
Damien Vergnaud

COMPOSITION DU JURY

Mme. Canteaut Anne
INRIA Paris
Examinatrice

M. Fouque Pierre-Alain
Université de Rennes 1
Rapporteur

Mme. Handschuh Helena
Rambus
Examinatrice

M. Pointcheval David
CNRS, École normale supérieure
Examinateur

M. Renault Guénaël
ANSSI
Examinateur

M. Rivain Matthieu
CryptoExperts
Directeur de thèse

M. Schwabe Peter
Radboud University
Rapporteur

M. Vergnaud Damien
UPMC
Directeur de thèse

Secure Implementation of Block Ciphers against Physical Attacks

Dahmun Goudarzi
Thèse de doctorat dirigée par
Matthieu Rivain et Damien Vergnaud

Résumé

Depuis leur introduction à la fin des années 1990, les attaques par canaux auxiliaires sont considérées comme une menace majeure contre les implémentations cryptographiques. Parmi les stratégies de protection existantes, une des plus utilisées est le masquage d'ordre supérieur. Elle consiste à séparer chaque variable interne du calcul cryptographique en plusieurs variables aléatoires. Néanmoins, l'utilisation de cette protection entraîne des pertes d'efficacité considérables, la rendant souvent impraticable pour des produits industriels.
Cette thèse a pour objectif de réduire l'écart entre les solutions théoriques, prouvées sûres, et les implémentations efficaces déployables sur des systèmes embarqués. Plus particulièrement, nous nous intéressons à la protection des algorithmes de chiffrement par bloc tel que l'AES, dont l'enjeu principal revient à protéger les boîtes-s avec un surcoût minimal.
Nous essayons tout d'abord de trouver des représentations mathématiques optimales pour l'évaluation des boîtes-s en minimisant le nombre de multiplications (un paramètre déterminant pour l'efficacité du masquage, mais aussi pour le chiffrement homomorphe). Pour cela, nous définissons une méthode générique pour décomposer n'importe quelle boîte-s sur un corps fini avec une complexité multiplicative faible. Ces représentations peuvent alors être évaluées efficacement avec du masquage d'ordre supérieur. La flexibilité de la méthode de décomposition permet également de l'ajuster facilement selon les nécessités du développeur.
Nous proposons ensuite une méthode formelle pour déterminer la sécurité d'un circuit évaluant des schémas de masquages. Cette technique permet notamment de déterminer de manière exacte si une attaque est possible sur un circuit protégé ou non. Par rapport aux autres outils existants, son temps de réponse n'explose pas en la taille du circuit et permet d'obtenir une preuve de sécurité quelque soit l'ordre de masquage employé. De plus, elle permet de diminuer de manière stricte l'emploi d'outils coûteux en aléas, requis pour renforcer la sécurité des opérations de masquages.
Enfin, nous présentons des résultats d'implémentation en proposant des optimisations tant sur le plan algorithmique que sur celui de la programmation. Nous utilisons notamment une stratégie d'implémentation *bitslice* pour évaluer les boîtes-s en parallèle. Cette stratégie nous permet d'atteindre des records de rapidité pour des implémentations d'ordres élevés. Les différents codes sont développés et optimisés en assembleur ARM, un des langages les plus répandus dans les systèmes embarqués tels que les cartes à puces et les téléphones mobiles. Ces implémentations sont, en outre, disponibles en ligne pour une utilisation publique.

Abstract

Since their introduction at the end of the 1990s, side-channel attacks are considered to be a major threat to cryptographic implementations. Higher-order masking is considered to be one the most popular existing protection strategies against such attacks. It consists in separating each internal variable in the cryptographic computation into several random variables. However, the use of this type of protection entails a considerable efficiency loss, making it unusable for industrial solutions.
The goal of this thesis is to reduce the gap between theoretical solutions, proven secure, and efficient implementations that can be deployed on embedded systems. More precisely, I analyzed the protection of block ciphers such as the AES encryption scheme, where the main issue is to protect the s-boxes with minimal overhead in costs.
I have tried, first, to find optimal mathematical representations in order to evaluate the s-boxes while minimizing the number of multiplications (an important parameter for masking schemes, but also for homomorphic encryption). For this purpose, I have defined a generic method to decompose any s-box on any finite field with a low multiplicative complexity. These representations can then be efficiently evaluated with higher-order masking. The flexibility of the decomposition technique further allows the developer to easily adapt it to its needs.
Secondly, I have proposed a formal method for measuring the security of circuits evaluating masking schemes. This technique allows to define with exact precision whether an attack on a protected circuit is feasible or not. Unlike other tools, its computation time is not exponential in the circuit size, making it possible to obtain a security proof regardless of the masking order used. Furthermore, this method can strictly reduce the use of costly tools in randomness required for reinforcing the security of masking operations.
Finally, I present some implementation results with optimizations at both algorithmic and programming levels. I particularly employ a *bitslice* implementation strategy for evaluating the s-boxes in parallel. This strategy leads to speed record for implementations protected at high orders. The different codes are developed and optimized in ARM assembly, one of the most popular programming language in embedded systems such as smart cards and mobile phones. These implementations are also available online for public use.

Acknowledgments

Je tiens tout d'abord à remercier Matthieu Rivain qui m'a donné l'opportunité de rejoindre la famille CryptoExperts, dans un premier temps en stage, puis en me renouvelant sa confiance en encadrant ma thèse pour une aventure qui aura duré 3 ans. Matthieu (a.k.a Patrooonnn) m'a transmis sa passion de la recherche (avec ses problèmes à 1 cocktail), du dévoleppement (avec ce duel sur l'AES en ARM, ou les challenges sur la multiplication) et il m'a permis de mûrir tant sur le plan cryptographique que sur le plan humain. Sans lui, je n'aurai probablement pas poussé mes études jusqu'à un doctorat. Nos nombreuses nuits blances (notamment celle du CHES 2016) resteront parmi mes meilleurs souvenirs de jeune chercheur. Je suis fier et honoré d'avoir pu être ton "premier".

Je tiens également à remercier Damien Vergnaud d'avoir accepté d'être mon directeur de thèse. Damien aurait pu faire le choix de regarder passer cette thèse passivement, comme cela peut arriver dans certaines CIFRE. Au contraire, Damien s'est investi dans ma thèse et sa culture cryptographique et mathématiques ont été précieuses. Nos nombreuses heures passées au tableau, à faire des démos ou à déchiffrer des papiers, parfois très obscurs, ont été fondamentales dans mon apprentissage. J'ai hâte de continuer nos collaborations en cours dans le futur. Je te remercie également d'avoir élargi ma collection rétro grâce à ton site magique!

Je remercie Anne Canteaut, Pierre-Alain Fouque, Helena Handschuh, David Pointcheval, Guénael Renault et Peter Scwhabe de me faire l'honneur de constituer mon jury de thèse. I particulary thank Pierre-Alain and Peter for taking the time of reviewing my PhD manuscript during their holidays!

Je suis également reconnaissant envers Guenael Renault d'avoir été un des meilleurs professeurs que j'ai croisé durant ma scolarité. Guenael m'a sorti d'une période obscure où je comptais arrêter mes études pour m'accueillir dans son master et me mettre sur la chemin de la cryptographie. Sa présence, son amitié, et ses conseils ont été primordiaux à l'aboutissement de mon parcours. Et puis sans lui, je n'aurai jamais pu implémenter des pivots de Gauss tous les ans sans discontinuité de la L3 au doctorat ou être dans le staff de "Le Ches"!

J'adresse également mes remerciements aux coauteurs et collaborateurs avec qui j'ai eu la chance de travailler ces dernières années. Merci donc à Srinivas Vivek, Anthony Journault, François-Xavier Standaert, Antoine Joux, Sonia Belaid, Victor Lomné, Jérémy Jean, Thomas Prest, Alain Passelegue et Ange Martinelli. Je tiens également à remercier André Chailloux, Thomas Debris, Jean-Pierre Tillich et Nicolas Sendrier de m'avoir donné l'opportunité de travailler pendant 4 mois à l'INRIA avec eux sur une proposition de signature. La une confiance totale qu'ils m'ont accordée pour le developpement de leur schéma m'a permis de franchir un cap professionnel important.

Cette thèse n'aura pas pu être faite sans l'environnement et la famille de CryptoExperts. J'aimerais donc remercier les membres (présents ou passés) qui la composent: Pascal Paillier (a.k.a Big Boss), Thomas Baignères, Sonia Belaid, Cécile Delerablée, Matthieu Finiasz,

Louis Goubin, Antoine Joux, Tancrède Lepoint, Chantal Lundvel, Michele Minelli, Matthieu Rivain et Junwei Wang. Mes conjugaisons de franglais, mon fin palet, les heures à faire du theory crafting, les jeux mobiles débiles, les jeux mobiles moins débiles, les organisations de cryptoparty, ma fameuse salade diet ou les sandwich Dahmun, tightProve, les high-five post acceptation, les brainstorms jeux de mots pourris pour à peu près tout et n'importe quoi, les 36000 propositions de logos (dont celles de Pascal), les paris pour l'Euro, les discutions sur le transhumanisme ou l'organisation d'un Cartathon, les courses metro, le slack, Ibizacrypt, les glaces (ah ces traîtresses), les riiiiiibsssssss, Keil (foutu Keil, un jour je te reconstruirai from scratch), la présence systématique de Big Boss tard le soir les semaines de deadline, la brigade du JouxPointFi et son intransigeance, les débats politiques, les pistaches, les vidéos de foot, Mathematica, New-York, Santa-Barbara, Vienne, Saint-Malo, agoranov, le linksys qui crame, la dégustation de caviar iranien, le champagne pour presque toutes les occasions possibles, le tapis de souris italien et le coq français, les courses carrefour, la tentative ratée de CM, les points synchros, LA partie d'echec (qui n'est pas finie!) mais surtout ce phénoménal "pot" (dans tous les sens du terme) de départ. Merci à vous pour tous ces moments partagés et pour le support et l'aide que vous m'avez fournis. Stylé de ouf, tu peux pas test !

J'ai eu la chance de pouvoir effectuer ma thèse dans l'équipe Crypto de l'ENS auprès de collègues mais surtout amis: Michel Abdalla, Balthazar Bauer, Sonia Belaid, Fabrice Benhamouda, Raphael Bost, Florian Bourse, Celine Chevalier, Jeremy Chotard, Simon Cogliani, Mario Cornejo, Geoffroy Couteau, Edouard Dufour Sans, Rafael Del Pino, Aurelien Dupin, Pierre-Alain Dupont, Pooya Farshim, Houda Ferradi, Georg Fuchsbauer, Romain Gay, Chloe Hebant, Julia Hesse, Louiza Khati, Pierrick Meaux, Thierry Mefenza, Michele Minelli, Anca Nitulescu, Michele Orru, Alain Passelegue, David Pointcheval, Thomas Prest, Razvan Rosie, Melissa Rossi, Adrian Thillard, Bogdan Ursu, Damien Vergnaud, and Hoeteck Wee. Merci pour ces escapes, soirées quizz, soirées jeux de sociétés, la finale du TI, ces horreurs d'aquariums, mon mug, secret santa, cryptnic, les fins de journées à la Montagne et surtout ces moments en conférence. J'aimerais aussi remercier l'équipe administrative de l'ENS et les membres du SPI: Jacques Beigbeder, Lise-Marie Bivard, Isabelle Delais, Nathalie Gaudechoux, Joelle Isnard, Valerie Mongiat, Ludovic Ricardou et Sophie Jaudon.

Je tiens également à exprimer ma gratitude aux personnes qui ont pris le temps de relire les versions préliminaires de ce mémoire: Ange, Aurélien, Laetitia, Matthieu F., Michele, Nima, Pierrick, Sonia, Thomas, Titâm et Mam.

Mes amis ont constitué une part importante de cette thèse. Leur expliquer ce que je faisais n'a pas toujours été facile mais ils ont toujours répondu présent quand j'avais besoin de me détacher de la cryptographie, qu'il s'agisse faire du sport, des escapes games, sauver new-york d'une épidémie ou Azeroth de la légion ardente. Les énumérer serait fastidieux mais ils se reconnaitront dans cette liste (et aussi, probablement, car il me reste 10 minutes avant d'aller à l'imprimerie). J'aimerais donc remercier la team h4, la team fac, la team magic, la team des Iraniens Z'ailés (retrouver nos "nombreux buts" sur Dailymotion!), la team CGI, la team ASP6, la colloque des fous malades, la colloque des bouchers Dunkerque, la team Capull, la team Panthéon et Montpar', la team Escape, les Quizzlords et la team Vegas Babyy.

Enfin, j'aimerai remercier ma famille d'avoir été présente depuis le début. Mon père et ma mère pour tout ce qu'ils m'ont inculqué et qui constitue ce que je suis aujourd'hui mais surtout pour leur soutien inconditionel, notamment lors de mes nombreux changements de parcours. Mon frère Tiam, pour m'avoir transmis toutes les passions que j'ai. Dès le plus jeune âge,

il m'a appris à regarder les mathématiques comme un jeu, m'a initié à l'informatique lors des premiers balbutiements d'Internet, mais aussi faut le dire, ses blagues vraiment nulles et souvent embarassantes ! Daye, mon oncle, mon modèle, merci pour tes passages fréquents à Paris où sur un coup de tête on allait au Parc. Amouh et Ameh, pour m'avoir offert mon premier travail comme caissier et laveur de voitures, pour les après-midis aux casinos, les duels épiques de backgammon ou les virées en camping-car. Mon cousin Freever qui à 13 ans m'a mis un bouquin de C dans les mains et m'a donné 5 ans pour que je l'assimile et qu'on monte notre boîte de jeux vidéos. La dernière partie attend toujours, mais je ne perds pas espoir ! Mes cousins Ashley, Jo, Nima et Sheyda, nos soirées parisiennes, new-yorkaises ou californiennes. Merci à Olivier, pour ta patience, ta mémoire fléchissante, mais surtout d'avoir veillé sur moi durant une période charnière. Tes tentatives de traits d'humour n'ont rien à envier à celles de Tiam! Et à mon neveu Nilai pour tout ce qu'il apporte à notre famille, sa joie, sa malice lorsqu'il me laisse lui acheter du Coca en douce ou monter sur mes épaules, sa passion pour les "bikinis", c'est-à-dire les croquemonsieurs, pour les néophytes.

Contents

Chapter 1

Introduction

Cryptography is the science of secret, being both an ancient art from the antiquity and a new science for which scientific research has grown popular since the seventies. From ancient history, its use was essential to allow different military entities to communicate about vital information (army positions or numbers, resource disposal, etc.) without risking any leakage in case enemies intercept a message. Its ground goal is hence to guarantee the confidentiality of the information, namely the fact that only the authorized parties (namely that possess a certain cryptographic key) can understand encrypted information. With the rise of new technologies and the Internet, protecting communications and sensible data has become a crucial issue. Modern cryptography has then transitioned to answer the needs of the society as much as in democratising itself to no longer only serve military purposes but also to protect anyone's privacy, as well as in diversifying its utility. Nowadays, cryptography can provide confidentiality (only the sender and the receiver can understand the communication), authenticity (the receiver can verify from who he received the communication) and data integrity (the communication has not been altered during its transfer).

Since the 1970's, we distinguish two types of cryptography: secret-key cryptography and public-key cryptography. The former allows to establish secure communication when the two representatives share a secret private key that is used for both encryption and decryption (hence its other designation of symmetric cryptography), whereas the latter uses a pair of different keys, a public key and a private key, where the public key is used for encryption and the private key for decryption (hence its other designation of asymmetric cryptography). In this thesis, we will mainly focus on symmetric cryptography, and more particularly on cryptographic schemes called block ciphers. One of the most prevailing and deployed block ciphers is the Advanced Encryption Standard (AES) proposed by Daemen and Rijmen in 1998 and standardized by the National Institute of Standards and Technology (NIST) in 2001. Based on a substitution-permutation network, this encryption scheme is very efficient both in hardware and software. The security of this scheme, as for most symmetric cryptographic solutions, does not rely on formal security proofs but on the fact that since its release in the public domain 20 years ago no attack that performs better than an exhaustive search of the secret key has been found.

In order to render cryptographic applications easy and transparent for day to day usage, industry opted in the middle of the 1980's for a hardware solution that could fit inside a pocket: the smart card. Those cards are tiny computers that store secret keys and can perform cryptographic operations such as encryption, personal identification or authentication. The most famous ones are credit cards, the French Carte Vitale or SIM cards in our mobile phones. This solution represents a difficult cryptographic challenge as once deployed in the wild, it should guarantee the protection of the stored secrets from malicious adversaries that

can have a total control of the hardware.

In the late 1980's, Kocher introduced the first physical attack against cryptographic hardware such as embedded systems. This kind of attack differs from the one in the classical model, namely the black-box model, where the adversary can only access the input and the output of the cryptosystem to recover its secret key. Through physical access to the device evaluating cryptographic operations, the adversary can obtain information on intermediate computations that directly depends on the secret key, and can hence break the underlying cryptosystem. In particular, the adversary can exploit side channels of the hardware such as the computation time, the power consumption of the device or its electro-magnetic emanation during the computation. Those attacks are particularly efficient and disastrous since one can easily get access to the target device such as smart cards, and utilize those side-channels.

Among the existing countermeasures against physical attacks to protect block cipher implementations, the most widely used is (higher-order) masking. It consists in splitting every sensitive variable in the cryptographic computation into several random values, denoted shares, so that at the end of the computation the sum of all the shares is equal to the expected result. This is a sound countermeasure as it has been formally proven that the higher the number of shares, the more difficult for the attacker to recover the secret key. However, applying higher-order masking to protect a cryptographic implementation implies a strong overhead in performance which can make its deployment in an industrial context unrealistic. In fact, the performance losses come from the fact that elementary arithmetic operations are replaced with tools that perform secure operations on encodings of sensitive variables. The linear operations, such as addition, have a computation time that is linear in the number of masks, whereas the computation time of non-linear operations such as multiplication is quadratic in the number of masks. Hence, the bottleneck of protecting encryption schemes with higher-order masking is to efficiently compute those non-linear operations, which in a standard block cipher are the substitution boxes (s-boxes). Many theoretical schemes have been proposed to tackle the issue of efficient masking schemes. However most of them are still considered impracticable as soon as the masking order grows.

In the last couple of years, a first step towards efficient implementation of higher-order masking has been made. The solution is to look at the s-box as a polynomial over a finite field and to find an efficient way to represent it so that the number of multiplications required for its evaluation is minimized. Two main approaches have been studied in the literature. The first one targets specific s-boxes (such as the one of the AES) to exploit the particularity of their algebraic structure and get very efficient representation in terms of number of multiplications involved. The second one tackles this issue on any kind of functions and aims to get a generic decomposition technique. For both approaches, a new problem quickly arose. In fact, once the representation of the s-box has been found, its evaluation requires several compositions of masked operations which can introduce security flaws due to multiple manipulations of a same encoding. To thwart such security flaws, the solution is to generate new fresh encodings of the intermediate variables at carefully chosen positions in the masking scheme evaluation. However, this requirement introduces new overheads in efficiency as generating fresh encodings requires a large amount of randomness. Hence, an other issue of masking schemes is to reduce the randomness requirements.

This is why this thesis will be focusing on producing efficient secure implementations of block ciphers against physical attacks and bridging the gap between theoretical masking schemes and concrete practical implementations for which the overhead in efficiency is lowered such that the underlying implementation can be deployed in industrial solutions. To solve

this problem, I have first started doing an implementation case study of all existing masking schemes in ARM assembly. This study resulted in the first concrete comparisons between all the polynomial solutions proposed in the state-of-the-art. I have also investigated an alternative to polynomial methods for specific block ciphers such as the AES which is based on bitslicing at the s-box level. This led to significantly faster implementations compared to polynomial solutions. Following this line of work, I studied the use of bitslicing for random s-boxes and proposed a method to get a representation of Boolean circuit with a minimalized multiplicative complexity. This new method led to implementation that are asymptotically faster than the classical polynomial approach but less efficient for small masking orders. Hence, I proposed a more generic framework that allows to decompose any polynomial over any finite field of characteristic two and encapsulating the previous methods (the polynomial approach and the Boolean approach). In all produced implementations, and more specifically for the AES s-box with a bitslice strategy, I have chosen a very conservative approach of inserting a refresh gadget after each multiplication to guarantee the security of the implementation. This conservative choice was backed up by the state-of-the-art tools that analyze the security of a circuit. However, these tools happen to produce false negatives and can hence lead to overkill in security gadgets. Therefore, I proposed a formal verification tool that can exactly check the security of an s-box implementation for any masking order. This allows to reduce by a factor of two the randomness requirement for the implementation of the AES s-box.

1.1 Personal Contributions

The detailed list of my personal publications with full names, authors, and conferences is given at the end of the introduction. Here I briefly describe the goal of each of these papers, as well as the results they achieve.

1.1.1 Contributions in this Thesis

[CHES:GouRiv16] In this paper, we present a generic method to find a Boolean representation of an s-box with efficient bitsliced higher-order masking. Specifically, we propose a method to construct a circuit with low multiplicative complexity. Compared to previous works on this subject, our method can be applied to any s-box of common size and not necessarily to small s-boxes. We use it to derive higher-order masked s-box implementations that achieve important performance gain compared to optimized state-of-the-art implementations.

[EC:GouRiv17] In this paper, we investigate efficient higher-order masking techniques by conducting a case study on ARM architectures (the most widespread architecture in embedded systems). We follow a bottom-up approach by first investigating the implementation of the base field multiplication at the assembly level. Then we describe optimized low-level implementations of the ISW scheme and its variant (CPRR) due to Coron *et al.* (FSE 2013). Finally we present improved state-of-the-art polynomial decomposition methods for s-boxes with custom parameters and various implementation-level optimizations. We also investigate an alternative to these methods which is based on bitslicing at the s-box level. We describe new masked bitslice implementations of the AES and PRESENT ciphers. These implementations happen to be significantly faster than (optimized) state-of-the-art polynomial methods.

[CHES:GRVV17] In this paper, we propose a generalized decomposition method for s-boxes that encompasses several previously proposed methods while providing new trade-offs. It allows to evaluate $n\lambda$-bit to $m\lambda$-bit s-boxes for any integers $n, m, \lambda \geq 1$ by seeing it a sequence of m n-variate polynomials over $\mathbb{F}_{2^\lambda}$ and by trying to minimize the number of multiplications over $\mathbb{F}_{2^\lambda}$.

[COSADE:GJRS18] In this paper, we optimize the performance and compare several recent masking schemes in bitslice representation on 32-bit ARM devices, with a focus on multiplication. Our main conclusion is that efficiency (or randomness) gains always come at a cost, either in terms of composability or in terms of resistance against horizontal attacks. Our evaluations should therefore allow a designer to select a masking scheme based on implementation constraints and security requirements. They also highlight the increasing feasibility of (very) high-order masking that are offered by increasingly powerful embedded devices, with new opportunities of high-security devices in various contexts.

[AC:BelGouRiv18] In this paper, we exhibit the first method able to clearly state whether a shared circuit composed of standard gadgets (addition, multiplication and refresh) is t-probing secure or not. Given such a composition, our method either produces a probing-security proof (valid at any order) or exhibits a security flaw that directly imply a probing attack at a given order. Compared to the state-of-the-art tool `maskComp`, our method can drastically reduce the number of required refresh gadgets to get a probing security proof, and thus the randomness requirement for some secure shared circuits. We apply our method to a recent AES implementation secured with higher-order masking in bitslice and we show that we can save all the refresh gadgets involved in the s-box layer, which results in a significant performance gain.

1.1.2 Other contributions

[SAC:GouRivVer16] In this paper, we extend existing attacks that exploit a few bits of the nonce to fully recover the secret key of elliptic-curve signature implementations protected with common countermeasures. Specifically, we extend the famous Howgrave-Graham and Smart lattice attack when the nonces are blinded by the addition of a random multiple of the elliptic-curve group order or by a random Euclidean splitting. We then assume that noisy information on the blinded nonce can be obtained through a template attack targeting the underlying scalar multiplication and we show how to characterize the obtained likelihood scores under a realistic leakage assumption. To deal with this scenario, we introduce a filtering method which given a set of signatures and associated likelihood scores maximizes the success probability of the lattice attack. Our approach is backed up with attack simulation results for several signal-to-noise ratio of the exploited leakage.

[AC:GouJouRiv18] In this paper, we show how to compute in the presence of noisy leakage with a leakage rate up to $\tilde{O}(1)$ in complexity $\tilde{O}(n)$. We use a polynomial encoding allowing quasilinear multiplication based on the fast Number Theoretic Transform (NTT). We first show that our scheme is secure in the random-probing model with leakage rate $O(1/\log n)$. Using the reduction by Duc *et al.* this result can be translated in the noisy leakage model with a $O(1/|\mathcal{F}|^2 \log n)$ leakage rate. However, as in the

work of Andrychowicz *et al.* [**EC:AndDziFau16**], our construction also requires $|\mathcal{F}| = O(n)$. In order to bypass this issue, we refine the granularity of our computation by considering the noisy leakage model on logical instructions that work on constant-size machine words. We provide a generic security reduction from the noisy leakage model at the logical-instruction level to the random-probing model at the arithmetic level. This reduction allows us to prove the security of our construction in the noisy leakage model with leakage rate $\tilde{O}(1)$.

1.2 Organization of this Thesis

In Chapter **??**, I first detail all of the state-of-the-art solutions for masking schemes and some overview of the software choices we made throughout this thesis. In Chapter **??**, I propose a generic decomposition method for any s-boxes to obtain efficient implementations. In Chapter **??**, I formally define how the security of a circuit protected with higher-order masking can be reduced to problem of linear algebra. Then, I propose a formal verification tool that given a circuit outputs if an attack exists or not for any masking order. In Chapter **??**, I detail optimisations at the algorithmic and assembly levels and give implementation results for the state-of-the-art methods as for the different schemes proposed in this thesis. Finally, I conclude in Chapter **??** with a brief summary of what I achieved and open questions that remain to be answered after this work.

Chapter 2

Preliminaries

2.1 Higher-Order Masking

Among the existing protection strategies against physical attacks, one of the most widely used relies on applying *secret sharing* at the implementation level, which is known as (*higher-order*) *masking*. This strategy achieves provable security in the so-called *probing security model* [**C:IshSahWag03**] and *noisy leakage model* [**EC:ProRiv13**; **EC:DucDziFau14**], which makes it a prevailing way to obtain secure implementations against side-channel attacks.

2.1.1 Description

Higher-order masking consists in sharing each internal variable x of a cryptographic computation into d random variables x_1, x_2, ..., x_d, called *the shares* and satisfying

$$x_1 + x_2 + \cdots + x_d = x, \tag{2.1}$$

for some group operation $+$, such that any set of $d-1$ shares is randomly distributed and independent of x. In this manuscript, we will consider the prevailing *Boolean masking* which is based on the bitwise addition of the shares. It has been formally demonstrated that in the noisy leakage model, where the attacker gets noisy information on each share, the complexity of recovering information on x grows exponentially with the number of shares [**C:CJRR99**; **EC:ProRiv13**]. This number d, called *the masking order*, is hence a sound security parameter for the resistance of a masked implementation.

When dth-order masking is involved to protect a block cipher, a so-called dth-order masking scheme must be designed to enable the computation on masked data. To be sound, a dth order masking scheme must satisfy the two following properties:

- *completeness:* at the end of the encryption/decryption, the sum of the d shares must give the expected result;
- *probing security:* every tuple of $d-1$ or less intermediate variables must be independent of any sensitive variable.

2.1.2 Masking Block Ciphers

Most block cipher structures are composed of one or several linear transformation(s), and a non-linear function, the *s-box* (where the linearity is considered w.r.t. the bitwise addition). Computing a linear transformation $x \mapsto \ell(x)$ in the masking world can be done in $O(d)$ complexity by applying ℓ to each share independently. This clearly maintains the probing security and the completeness holds by linearity since we have $\ell(x_1)+\ell(x_2)+\cdots+\ell(x_d) = \ell(x)$. On the other hand, the non-linear operations (such as s-boxes) are more tricky to compute on the shares while ensuring completeness and probing security.

The state-of-the-art solutions to perform a multiplication between two masked variable x and y are all based on the following rationale:

1. Compute the d^2 cross products $x_i y_j$, for $1 \leq i, j \leq d$;
2. Combine the cross products to get d output shares (in order to keep completeness);
3. Interleave fresh randomness in the previous step to avoid security flaws (in order to keep the probing security).

The main differences between all the solution in the literature are on how to implement this three different stages (see for instance [**C:IshSahWag03**; **EC:BBPPTV16**; **CHES:BCPZ16**; **EC:BDFGSS17**]).

2.1.3 Implementation Transformation

We now describe how to transform the implementation of a block cipher into a secure implementation with higher-order masking.

2.1.3.1 Encoding

Let us first detail how to encode every input of the block cipher into masked variables. Let d be the masking order. A random d-sharing of an element $x \in \mathbb{F}_{2^n}$ is defined as follows:

Definition 2.1.1. *Let $x \in \mathbb{F}_{2^n}$ and $d \in \mathbb{N}$. A d-sharing of x is a tuple $(x_i)_{i=1}^d \in \mathbb{F}_{2^n}^d$ satisfying $\sum_{i=1}^d x_i = x$.*

A d-sharing is said to be *uniform* if, for a given x, it is uniformly distributed over the subspace of tuples satisfying $x = \sum_{i=1}^d x_i$. A uniform sharing of x is such that any m-tuple of its shares x_i is uniformly distributed over $\mathbb{F}_2^m$ for any $m \leq d$. We further denote by Enc the probabilistic algorithm that maps an element $x \in \mathbb{F}_{2^n}$ to a random uniform d-sharing of x:

$$\mathsf{Enc}(x) \mapsto (x_1, x_2, \ldots, x_d) \ . \tag{2.2}$$

The corresponding decoding function Dec is defined as:

$$\mathsf{Dec}(x_1, x_2, \ldots, x_d) := \sum_{i=1}^{d} x_i \tag{2.3}$$

It is easy to check that we have $\Pr(\mathsf{Dec}(\mathsf{Enc}(x)) = x) = 1$ for every $x \in \mathbb{F}_{2^n}$.

2.1.3.2 Gadgets

To perform operations on masked variables, we define new building blocks called *gadgets* that perform a given operation on d-sharings. For instance, for some two-input operation $*$, a $*$-gadget takes two input sharings $\mathsf{Enc}(x_1)$ and $\mathsf{Enc}(x_2)$ and it outputs a sharing $\mathsf{Enc}(y)$ such that $y = x_1 * x_2$. In this thesis, we will only consider gadgets that take one input (refresh, squaring, or evaluation of a quadratic function) or two inputs (addition, subtraction, multiplication). Specifically, a gadget performs several elementary operation on the shares of the inputs in order to get the desired outputs. We give in the following some examples of the most used gadgets in this thesis.

- **Addition gadget.** Let $(x_i)_{i=1}^d$ be a d-sharing of x and $(y_i)_{i=1}^d$ be a d-sharing of y. To compute a d-sharing $(z_i)_{i=1}^d$ of $z = x + y$, we simply compute:

$$(z_1, z_2, \ldots, z_d) \leftarrow (x_1 + y_1, x_2 + y_2, \ldots, x_d + y_d)$$

- **Field squaring gadget.** Let $(x_i)_{i=1}^d$ be a d-sharing of x. To compute a d-sharing $(z_i)_{i=1}^d$ of $z = x^2$, we simply compute:

$$(z_1, z_2, \ldots, z_d) \leftarrow (x_1^2, x_2^2, \ldots, x_d^2)$$

- **Field multiplication gadget.** Let $(x_i)_{i=1}^d$ be a d-sharing of x and $(y_i)_{i=1}^d$ be a d-encoding of y. To compute a d-sharing $(z_i)_{i=1}^d$ of $z = x \cdot y$, we simply compute:

$$(z_1, z_2, \ldots, z_d) \leftarrow \mathsf{IswMult}((x_1, x_2, \ldots, x_d), (y_1, y_2, \ldots, y_d))$$

  where IswMult denotes the secure multiplication proposed by Ishai, Sahai and Wager and described in the following section (Section **??**). Other algorithms for this operation are described in Chapter **??**.

Finally, another type of gadget exists that does not perform an operation on d-sharing but from an input d-sharing of a secret variable x, produces a fresh random uniform d-sharing of this variable. This type of gadgets is called *refresh* gadget and works as follows:

- **Refresh gadget.** Let $(x_i)_{i=1}^d$ be a d-sharing of x. To compute a d-sharing $(z_i)_{i=1}^d$ of $z = x$, we simply compute:

$$(z_1, z_2, \ldots, z_d) \leftarrow \mathsf{Refresh}(x_1, x_2, \ldots, x_d)$$

  The details on how to compute soundly the Refresh function are detailed hereafter. Basically, the idea is to perform a secure multiplication between an encoding of x and an encoding of 1, namely the d-sharing $(1, 0, \ldots, 0)$.

It is easy to see that to remain efficient, the goal is to find a secure implementation of the block cipher that minimizes the number of field multiplication, quadratic evaluation and refresh gadgets, as they are the ones with a quadratic complexity in the masking order. In this manuscript, we will equally use the denomination of gadgets and secure operations (namely secure multiplication, secure evaluation, etc...).

2.1.3.3 Transformation

Now that we have defined encodings of sensitive variables and gadgets that perform simple operations on encodings, we can describe how to transform an implementation of a block cipher into a secure implementation protected with higher-order masking. To do so, every elementary operation in the implementation of the block cipher is transformed into the corresponding gadget working on d-sharings. Hence, the secure implementation takes as inputs d-sharings corresponding to each input of the block cipher thanks to the encoding function Enc. At the end of the computation, the output d-sharings of the computation are transfomed into the outputs of the block-ciphers with the help of the decoding function Dec. At carefully chosen positions in the secure implementation, refresh gadgets are inserted in order to ensure the security of the implementation (see Chapter **??** for more details).

2.1.4 Probing security

The probing security was first introduced in the seminal work of Ishai, Sahai, and Wager at Crypto 2003 [**C:IshSahWag03**]. It has been used to prove the security of several of the masking schemes in the literature. The definition of the probing security is defined as follows:

Definition 2.1.2 (from [**C:IshSahWag03**])**.** *Let d be the masking order and $1 \leq t \leq d$ an integer. A algorithm is t-probing secure if and only if any set of at most t intermediate variables is independent from the secret.*

Informally speaking, in this model, we suppose that the adversary can probe up to t intermediate computations in the secure implementation. Namely, she can place up to t probes in any of the elementary operations performed by the gadgets. As each of the elementary operations in a gadget are performed on shares, the attacker recovers the knowledge of the one or two shares processed by the operation. This can be seen as probing one software instruction where one probe allows to recover information on either the source of the instruction (namely one or two operands) or the destination (namely the output of the instruction). Then with the information provided by her probes, the adversary tries to find some dependence with the secret. If such dependence does not exist, then the secure implementation is said to be t-probing secure. One of the strongest advantages of this model is that it allows to do elegant and easy proofs to ensure the security of the secure implementation.

This model is also practically relevant and realistic as it catches what an adversary performs with an higher-order side-channel attack. In fact, in a practical scenario the attacker tries to find points of interest in the leakage trace corresponding to the evaluation of the implementation. Then he tries to recombine these point of interests to find a statistical correlation with the secret. Hence, the point of interests can be seen as placing probes in the secure implementation.

2.2 Secure Non-Linear Operations

We now review some (widely used) constructions from the literature to compute masking gadgets for non-linear operations. The two operands $x, y \in \mathbb{F}_{2^n}$ of the non-linear operation gadget are represented as random d-sharings $(x_1, x_2, \ldots, x_d)$ and $(y_1, y_2, \ldots, y_d)$. Specifically, we focus on the scheme proposed by Ishai, Sahai, and Wagner (ISW scheme) for the secure multiplication [**C:IshSahWag03**], and its extension by Coron, Prouff, Rivain and Roche (CPRR scheme) to secure any quadratic function [**FSE:CPRR13**; **C:CPRR15**].

2.2.1 Ishai-Sahai-Wagner Multiplication

From two d-sharings $(x_1, x_2, \ldots, x_d)$ and $(y_1, y_2, \ldots, y_d)$, the ISW scheme computes an output d-sharing $(z_1, z_2, \ldots, z_d)$ as follows:

1. for every $1 \leq i < j \leq d$, sample a random value $r_{i,j}$ over $\mathbb{F}_{2^n}$;
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} + x_i \cdot y_j) + x_j \cdot y_i$;
3. for every $\leq i \leq d$, compute $z_i = x_i \cdot y_i + \sum_{j \neq i} r_{i,j}$.

One can check that the output $(z_1, z_2, \ldots, z_d)$ is well a d-sharing of the product $z = x \cdot y$. We indeed have $\sum_i z_i = \sum_{i,j} x_i \cdot y_j = (\sum_i x_i)(\sum_j y_j)$ since every random value $r_{i,j}$ appears exactly twice in the sum and hence vanishes.

2.2.2 Mask refreshing

The ISW multiplication was originally proved probing secure at the order $t = \lfloor (d-1)/2 \rfloor$ (and not $d-1$ as one would expect with masking order d). The security proof was later made tight under the condition that the input d-sharings are based on independent randomness [**CHES:RivPro10**]. In some situations, this independence property is not satisfied. For

instance, one might have to multiply two values x and y where $x = \ell(y)$ for some linear operation ℓ. In that case, the shares of x are usually derived as $x_i = \ell(y_i)$, which clearly breaches the required independence of input shares. To deal with this issue, one must refresh the sharing of x. However, one must be careful doing so since a bad refreshing procedure might introduce a flaw [**FSE:CPRR13**]. A sound method for mask-refreshing consists in applying an ISW multiplication between the sharing of x and the tuple $(1, 0, 0, \ldots, 0)$ [**EC:DucDziFau14**; **EC:BBDFGS15**]. This gives the following procedure:

1. for every $1 \leq i < j \leq d$, randomly sample $r_{i,j}$ over $\mathbb{F}_{2^n}$ and set $r_{j,i} = r_{i,j}$;
2. for every $1 \leq i \leq d$, compute $x'_i = x_i + \sum_{j \neq i} r_{i,j}$.

It is not hard to see that the output sharing $(x'_1, x'_2, \ldots, x'_d)$ well encodes x. One might think that such a refreshing implies a strong overhead in performance (almost as performing two multiplications) but this is still better than doubling the number of shares (which roughly quadruples the multiplication time). Moreover, we show in Chapter **??** that the implementation of such a refreshing procedure can be very efficient in practice compared to the ISW multiplication.

2.2.3 Coron-Prouff-Rivain-Roche evaluation

The CPRR scheme was initially proposed in [**FSE:CPRR13**] as a variant of ISW to securely compute multiplications of the form $x \mapsto x \cdot \ell(x)$ where ℓ is linear, without requiring refreshing. It was then shown in [**C:CPRR15**] that this scheme (in a slightly modified version) could actually be used to securely evaluate any quadratic function f over $\mathbb{F}_{2^n}$. The method is based on the following equation

$$f(x_1 + x_2 + \cdots + x_d) = \sum_{1 \leq i < j \leq d} f(x_i + x_j + s_{i,j}) + f(x_j + s_{i,j}) + f(x_i + s_{i,j}) + f(s_{i,j}) + \sum_{i=1}^{d} f(x_i) + (d + 1 \bmod 2) \cdot f(0) \quad (2.4)$$

which holds for every $(x_i)_i \in (\mathbb{F}_{2^n})^d$, every $(s_{i,j})_{1 \leq i < j \leq d} \in (\mathbb{F}_{2^n})^{d(d-1)/2}$, and every quadratic function f over $\mathbb{F}_{2^n}$.

From a d-sharing $(x_1, x_2, \ldots, x_d)$, the CPRR scheme computes an output d-sharing $(y_1, y_2, \ldots, y_d)$ as follows:

1. for every $1 \leq i < j \leq d$, sample two random values $r_{i,j}$ and $s_{i,j}$ over $\mathbb{F}_{2^n}$,
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = r_{i,j} + f(x_i + s_{i,j}) + f(x_j + s_{i,j}) + f((x_i + s_{i,j}) + x_j) + f(s_{i,j})$,
3. for every $1 \leq i \leq d$, compute $y_i = f(x_i) + \sum_{j \neq i} r_{i,j}$,
4. if d is even, set $y_0 = y_0 + f(0)$.

According to Equation (**??**), we then have $\sum_{i=1}^{d} y_i = f\big(\sum_{i=1}^{d} x_i\big)$, which shows that the output sharing $(y_1, y_2, \ldots, y_d)$ well encodes $y = f(x)$.

In [**FSE:CPRR13**; **C:CPRR15**] it is argued that in the gap where the field multiplication cannot be fully tabulated (2^{2n} elements is too much) while a function $f : \mathbb{F}_{2^n} \to \mathbb{F}_{2^n}$ can

be tabulated (2^n elements fit), the CPRR scheme is (likely to be) more efficient than the ISW scheme. This is because it essentially replaces (costly) field multiplications by simple look-ups.

2.3 Polynomial Methods for S-boxes

An approach to reduce the number of multiplication involved in the evaluation of the s-box in masking schemes is to consider the s-box as a polynomial over a finite field. This way, the problem of efficiently evaluating the non-linear part becomes equivalent to studying fast evaluation of polynomials over $\mathbb{F}_{2^n}$. In [**FSE:CGPQR12**], Carlet, Goubin, Prouff, Quisquater and Rivain, were the first to give a countermeasure based on polynomial representation of the s-box and proposed two methods to evaluate it. Then in [**CHES:CorRoyViv14**], Coron, Roy and Vivek extend their approach to propose a new generic method for fast evaluation of polynomials over binary fields.

In [**FSE:CGPQR12**] the notion of cyclotomic class is introduced, which will be used for the CRV method.

Definition 2.3.1. *[Cyclotomic class]* Let n be the number of input bits of the s-box and let $\alpha \in [0; 2^n - 2]$. The cyclotomic class of α is the set C_α defined by:

$$C_\alpha = \{\alpha.2^i \mod 2^n - 1 \ ; \ i \in [0; n-1]\}$$

A cyclotomic class allows to get efficient masked operations: a power x^α can be computed from x^β without any non-linear multiplications if α and β are in the same cyclotomic class since it only requires squarings.

2.3.1 Coron-Roy-Vivek method

The CRV method was proposed by Coron, Roy and Vivek in [**CHES:CorRoyViv14**]. It consists in representing an s-box $S(x)$ over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ as

$$S(x) = \sum_{i=1}^{t-1} p_i(x) \cdot q_i(x) + p_t(x) \ , \tag{2.5}$$

where $p_i(x)$ and $q_i(x)$ are polynomials with monomials in x^L for some set of cyclotomic classes $L = C_{\alpha_1=0} \cup C_{\alpha_2=1} \cup C_{\alpha_3} \cup \ldots \cup C_{\alpha_\ell}$ such that for every $i \geq 3$, $\alpha_i = \alpha_j + \alpha_k \bmod 2^n - 1$ for some $j, k < i$ (or more generally $\alpha_i = 2^w \cdot \alpha_j + \alpha_k \bmod 2^n - 1$ with $k \in [\![0, n-1]\!]$). Such polynomials can be written as:

$$p_i(x) = \sum_{j=2}^{\ell} l_{i,j}(x^{\alpha_j}) + c_{i,0} \quad \text{and} \quad q_i(x) = \sum_{j=2}^{\ell} l'_{i,j}(x^{\alpha_j}) + c'_{i,0} \ , \tag{2.6}$$

where the $l_{i,j}, l'_{i,j}$ are linearized polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ and where the $c_{i,0}, c'_{i,0}$ are constants in $\mathbb{F}_{2^n}$.

In [**CHES:CorRoyViv14**], the authors explain how to find such a representation. In a nutshell, one randomly picks the q_i's and searches for p_i's satisfying Equation (**??**). This amounts to solving a linear system with 2^n equations and $t \cdot |L|$ unknowns (the coefficients of the p_i's). Note that when the choice of the classes and the q_i's leads to a solvable system,

then it can be used with any s-box (since the s-box is the target vector of the linear system). We then have two necessary (non sufficient) conditions for such a system to be solvable, and one additional for the sake of efficiency:

1. the set L of cyclotomic classes is such that $t \cdot |L| \geq 2^n$
2. all the monomials can be reached by multiplying two monomials from x^L, that is $\{x^i \cdot x^j \bmod (x^{2^n} - x)\ ;\ i, j \in L\} = x^{[\![0,2^n-1]\!]}$
3. every class (but $C_0 = \{0\}$) have the maximal cardinality of n. Under this additional constraint, Condition **??** leads to the following inequality: $t \cdot (1 + n \cdot (\ell - 1)) \geq 2^n$.

Minimizing the number of nonlinear multiplications while satisfying this constraint leads to parameters $t \approx \sqrt{2^n/n}$ and $\ell \approx \sqrt{2^n/n}$.

Based on the above representation, the s-box can be evaluated using $(\ell - 2) + (t - 1)$ nonlinear multiplications (plus some linear operations). In a first phase, one generates the monomials corresponding to the cyclotomic classes in L. Each x^{α_i} can be obtained by multiplying two previous x^{α_j} and x^{α_k} (where x^{α_j} might be squared w times if necessary). In the masking world, each of these multiplications is performed with a call to ISW. The polynomials $p_i(x)$ and $q_i(x)$ can then be computed according to Equation (**??**). In practice the linearized polynomials are tabulated so that at masked computation, applying a $l_{i,j}$ simply consists in performing a look-up on each share of the corresponding x^{α_j}. In the second phase, one simply evaluates Equation (**??**), which takes $t - 1$ nonlinear multiplications plus some additions. We recall that in the masking world, linear operation such as additions or linearized polynomial evaluations can be applied on each share independently yielding an $O(d)$ complexity, whereas nonlinear multiplications are computed by calling ISW with an $O(d^2)$ complexity. The performance of the CRV method is hence dominated by the $\ell + t - 3$ calls to ISW.

2.3.2 Algebraic decomposition method

The algebraic decomposition method was proposed by Carlet, Prouff, Rivain and Roche in [**C:CPRR15**]. It consists in using a basis of polynomials $(g_1, g_2, \ldots, g_r)$ that are constructed by composing polynomials f_i as follows

$$\begin{cases} g_1(x) = f_1(x) \\ g_i(x) = f_i(g_{i-1}(x)) \end{cases} \tag{2.7}$$

The f_i's are of given algebraic degree s. In our context, we consider the algebraic decomposition method for $s = 2$, where the f_i's are (algebraically) quadratic polynomials. The method then consists in representing an s-box $S(x)$ over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$ as

$$S(x) = \sum_{i=1}^{t} p_i(q_i(x)) + \sum_{i=1}^{r} \ell_i(g_i(x)) + \ell_0(x)\ , \tag{2.8}$$

with

$$q_i(x) = \sum_{j=1}^{r} \ell_{i,j}(g_j(x)) + \ell_{i,0}(x)\ , \tag{2.9}$$

where the p_i's are quadratic polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$, and where the ℓ_i's and the $\ell_{i,j}$'s are linearized polynomials over $\mathbb{F}_{2^n}[x]/(x^{2^n} - x)$.

As explain in [**C:CPRR15**], such a representation can be obtained by randomly picking some f_i's and some $\ell_{i,j}$'s (which fixes the q_i's) and then searching for p_i's and ℓ_i's satisfying Equation (**??**). As for the CRV method, this amounts to solving a linear system with 2^n equations where the unknowns are the coefficients of the p_i's and the ℓ_i's. Without loss of generality, we can assume that only ℓ_0 has a constant term. In that case, each p_i is composed of $\frac{1}{2}n(n-1)$ monomials, and each ℓ_i is composed of n monomials (plus a constant term for ℓ_0). This makes a total of $\frac{1}{2}\,n\,(n-1)\cdot t + n\cdot(r+1)$ unknown coefficients. In order to get a solvable system we hence have the following condition: (1) $\frac{1}{2}\,n\,(n-1)\cdot t + n\cdot(r+1) \geq 2^n$. A second condition is (2) $2^{r+1} \geq n$, otherwise there exists some s-box with algebraic degree greater than 2^{r+1} that cannot be achieved with the above decomposition *i.e.* the obtained system is not solvable for every target S.

Based on the above representation, the s-box can be evaluated using $r + t$ evaluations of quadratic polynomials (the f_i's and the q_i's). In the masking world, this is done with the help of CPRR evaluations. The rest of the computation are additions and (tabulated) linearized polynomials which are applied to each share independently with a complexity linear in d. The cost of the algebraic decomposition method is then dominated by the $r + t$ calls to CPRR.

2.3.3 Specific Methods for AES and PRESENT

Specific methods that target the algebraic structure of a particular s-box have also been proposed in the literature. In this thesis, we mainly focus on the s-box of the AES and of PRESENT as they are popular block-ciphers, the former beeing the standard for encryption and the latter one of the most efficient lightweight block-ciphers.

2.3.3.1 Rivain-Prouf Decomposition

Many works have studied masking schemes for the AES s-box and most of them are based on its peculiar algebraic structure. This s-box is defined as the composition of the *inverse function* $x \mapsto x^{254}$ over $\mathbb{F}_{2^8}$ and an affine function: $S(x) = \mathrm{Aff}(x^{254})$. The affine function being straightforward to mask with linear complexity, the main issue is to design an efficient masking scheme for the inverse function. In [**CHES:RivPro10**], Rivain and Prouff introduced the approach of using an efficient addition chain for the inverse function that can be implemented with a minimal number of ISW multiplications. They show that the exponentiation to the power of 254 can be performed with 4 nonlinear multiplications plus some (linear) squarings, resulting in a scheme with 4 ISW multiplications. In [**FSE:CPRR13**], Coron *et al.* propose a variant where two of these multiplications are replaced CPRR evaluations (of the functions $x \mapsto x^3$ and $x \mapsto x^5$).[1] This was further improved by Grosso *et al.* in [**AFRICACRYPT:GroProSta14**] who proposed the following addition chain leading to 3 CPRR evaluations and one ISW multiplications: $x^{254} = (x^2 \cdot ((x^5)^5)^5)^2$. This addition chain has the advantage of requiring a single function $x \mapsto x^5$ for the CPRR evaluation (hence a single LUT for masked implementation). Moreover it can be easily checked by exhaustive

[1] The original version of the RP scheme [**CHES:RivPro10**] actually involved a weak mask refreshing procedure which was exploited in [**FSE:CPRR13**] to exhibit a flaw in the s-box processing. The CPRR variant of ISW was originally meant to patch this flaw but the authors observed that using their scheme can also improve the performances. The security of the obtained variant of the RP scheme was recently verified up to masking order 4 using program verification techniques [**EC:BBDFGS15**].

search that no addition chain exists that trades the last ISW multiplication for a CPRR evaluation.

2.3.3.2 Kim-Hong-Lim (KHL) method

This method was proposed in [**CHES:KimHonLim11**] as an improvement of the RP scheme. The main idea is to use the tower field representation of the AES s-box [**AC:SMTM01**] in order to descend from $\mathbb{F}_{2^8}$ to $\mathbb{F}_{2^4}$ where the multiplications can be fully tabulated. Let δ denote the isomorphism mapping $\mathbb{F}_{2^8}$ to $(\mathbb{F}_{2^4})^2$ with $\mathbb{F}_{2^8} \equiv \mathbb{F}_{2^4}[x]/p(x)$, and let $\gamma \in \mathbb{F}_{2^8}$ and $\lambda \in \mathbb{F}_{2^4}$ such that $p(x) = x^2 + x + \lambda$ and $p(\gamma) = 0$. The tower field method for the AES s-box works as follows:

1. $a_h\gamma + a_l = \delta(x),\ a_h, a_l \in \mathbb{F}_{2^4}$
2. $d = \lambda\, a_h^2 + a_l \cdot (a_h + a_l) \in \mathbb{F}_{2^4}$
3. $d' = d^{14} \in \mathbb{F}_{2^4}$
4. $a'_h = d'\, a_j \in \mathbb{F}_{2^4}$
5. $a'_l = d'(a_h + a_l) \in \mathbb{F}_{2^4}$
6. $S(x) = \mathrm{Aff}(\delta^{-1}(a'_h\gamma + a'_l)) \in \mathbb{F}_{2^8}$

At the third step, the exponentiation to the 14 can be performed as $d^{14} = (d^3)^4 \cdot d^2$ leading to one CPRR evaluation (for $d \mapsto d^3$) and one ISW multiplication (plus some linear squarings).[2] This gives a total of 4 ISW multiplications and one CPRR evaluation for the masked AES implementation.

2.3.3.3 Method for PRESENT

As a 4-bit s-box, the PRESENT s-box can be efficiently secured with the CRV method using only 2 ISW multiplications, where the field multiplications are fully tabulated. The algebraic decomposition method would give a less efficient implementation with 3 CPRR evaluations. Another possible approach is to use the fact that the PRESENT s-box can be expressed as the composition of two quadratic functions $S(x) = F \circ G(x)$. This representation (which is recalled in Table **??**) was put forward by Poschmann *et al.* in [**JC:PMKLW11**] to design an efficient *threshold implementation* of PRESENT. In our context, this representation can be used to get a masked s-box evaluation based on 2 CPRR evaluations. Note that this method is asymptotically slower than CRV with 2 full-table ISW multiplications. However, due to additional linear operations in CRV, $F \circ G$ might actually be better for small values of d.

Table 2.1: PRESENT s-box $S(x) = F \circ G(x)$.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2
$G(x)$	7	E	9	2	B	0	4	D	5	C	A	1	8	3	6	F
$F(x)$	0	8	B	7	A	3	1	C	4	6	F	9	E	D	5	2

[2] The authors of [**CHES:KimHonLim11**] suggest to perform $d^3 = d^2 \cdot d$ with a full tabulated multiplication but this would actually imply a flaw as described in [**FSE:CPRR13**]. That is why we use a CPRR evaluation for this multiplication.

2.4 Bitslice Strategy

Bitslicing is an implementation strategy initially proposed by Biham in [**FSE:Biham97b**]. It consists in performing several parallel evaluations of a Boolean circuit in software where the logic gates can be replaced by instructions working on registers of several bits. As nicely explained in [**CHES:MatNak07**], "*in the bitslice implementation one software logical instruction corresponds to simultaneous execution of ℓ hardware logical gates, where ℓ is a register size* [...] *Hence bitslice can be efficient when the entire hardware complexity of a target cipher is small and an underlying processor has many long registers.*"

In the context of higher-order masking, bitslicing can be used at the s-box level to perform several secure s-box computations in parallel. One then needs a compact Boolean representation of the s-box, and more importantly a representation with the least possible nonlinear gates. Specifically, based on a Boolean circuit for an s-box S, one can perform ℓ parallel evaluations of S in software by replacing each gate of the circuit with the corresponding bitwise instruction, where ℓ is the bit-size of the underlying CPU architecture. It results that the only nonlinear operations in the parallel s-box processing are bitwise AND instructions between ℓ-bit registers which can be efficiently secured using the ISW scheme. Such an approach achieves important speedup compared to polynomial methods since (i) ISW-based ANDs are substantially faster than ISW-based field multiplications in practice, (ii) all the s-boxes within a cipher round are computed in parallel. Such an approach was applied in [**FSE:GLSV14**] to design block ciphers with efficient masked computations. To the best of our knowledge, it has never been applied to obtain fast implementations of classical block ciphers such as AES or PRESENT.

2.4.1 ISW Logical AND

The ISW scheme can be easily adapted to secure a bitwise logical AND between two m-bit registers. From two d-sharings $(a_1, a_2, \ldots, a_d)$ and $(b_1, b_2, \ldots, b_d)$ of two m-bit strings $a, b \in \{0,1\}^m$, the ISW scheme computes an output d-sharing $(c_1, c_2, \ldots, c_d)$ of $c = a \wedge b$ as follows:

1. for every $1 \leq i < j \leq d$, sample an m-bit random value $r_{i,j}$,
2. for every $1 \leq i < j \leq d$, compute $r_{j,i} = (r_{i,j} \oplus a_i \wedge b_j) \oplus a_j \wedge b_i$,
3. for every $1 \leq i \leq d$, compute $c_i = a_i \wedge b_i \oplus \bigoplus_{j \neq i} r_{i,j}$.

If the architecture size is ℓ, we can hence perform ℓ secure logical AND in parallel. Moreover a logical AND is a single instruction so we expect the above ISW logical AND to be faster than the ISW field multiplications.

2.5 ARM Assembly

In this thesis most of the implementation produced to benchmark and compare the different masking schemes are developed in ARM assembly, a low level implementation language widely used in mobile devices and smart cards. Most ARM cores are RISC processors composed of sixteen 32-bit registers, labeled `R0`, `R1`, ..., `R15`. Registers `R0` to `R12` are known as *variable*

registers and are available for computation.[3] The three last registers are usually reserved for special purposes: `R13` is used as the stack pointer (`SP`), `R14` is the link register (`LR`) storing the return address during a function call, and `R15` is the program counter (`PC`). The link register `R14` can also be used as additional variable register by saving the return address on the stack (at the cost of push/pop instructions). The gain of having a bigger register pool must be balanced with the saving overhead, but this trick enables some improvements in many cases.

In ARM v7, most of the instructions can be split into the following three classes: *data instructions*, *memory instructions*, and *branching instructions*. The data instructions are the arithmetic and bitwise operations, each taking one clock cycle (except for the multiplication which takes two clock cycles). The memory instructions are the load and store (from and to the RAM) which require 3 clock cycles, or their variants for multiple loads or stores ($n + 2$ clock cycles). The last class of instructions is the class of branching instructions used for loops, conditional statements and function calls. These instructions take 3 or 4 clock cycles. This classification is summarized in Table **??**.

Table 2.2: ARM instructions.

Class	Examples	Clock cycles
Data instructions	`EOR`, `ADD`, `SUB`, `AND`, `MOV`	1
Memory instructions	`LDR`, `STR` / `LDM`, `STM`	3 or $n + 2$
Branching instructions	`B`, `BX`, `BL`	3 or 4

One important specificity of the ARM assembly is the *barrel shifter* allowing any data instruction to shift one of its operands at no extra cost in terms of clock cycles. Four kinds of shifting are supported: the logical shift left (`LSL`), the logical shift right (`LSR`), the arithmetic shift right (`ASR`), and the rotate-right (`ROR`). All these shifting operations are parameterized by a shift length in $[\![1, 32]\!]$ (except for the logical shift left `LSL` which lies in $[\![0, 31]\!]$). The latter can also be relative by using a register but in that case the instruction takes an additional clock cycle.

Another feature of ARM assembly that can be very useful is the branch predication. This feature allows any instruction to be conditionally executed with respect to a (data dependent) flag value (*e.g.* the carry flag, the zero flag, *etc.*). In the context of secure embedded implementations, we have to ensure a data-independent operation flow in order to prevent timing and/or simple side-channel attacks. We must therefore avoid data-dependent conditional branches. For this reason we do not make use of the branch predication, except when it involves non-sensitive data such as loop counters.

2.6 Open Problems

As explained in this chapter, gadgets performing operation on d-sharing are the ground building block of masking schemes. As the multiplication gadgets require a quadratic number of field operations with respect to the masking order, an interesting line of work could be

[3] Note that some conventions exist for the first four registers `R0`–`R3`, also called *argument registers*, and serving to store the arguments and the result of a function at call and return respectively.

to find efficient ways to perform those operations. In fact, field multiplications are the core operations in the evaluation of multiplication gadgets. As soon as the field size grows, field multiplication can not be fully tabulated due to the size of the look-up tables. Hence, studying and optimizing at the implementation level different approaches to perform those field multiplications, such as the Karatsuba multiplication or the exp-log multiplication, can lead to efficient multiplication gadgets with different time and memory trade-offs.

In the past few years, several new secure multiplication schemes have been proposed based on the ISW multiplication method to get more efficient (at the cost of security) or more secure (at the cost of effiency) multiplication gadgets. However, no work made a concrete comparison between these different secure multiplications to better understand the performance gains and overheads that correspond to their different security guarantees. Moreover, several lines of work are based on the fact that the CPRR evaluation scheme should outperform the ISW multiplication scheme to justify their efficiency. However, no concrete comparisons at an implementation level have been made to corroborate the state-of-the-art intuition.

One of the missing part of all the proposed masking schemes to get an efficient polynomial decomposition of an s-box is a concrete comparison of them in practice. In fact, no work has produced a fair implementation of all the solutions and analysed the performance results to have clear insight on which method to use. It also seems like going down in a smaller field to perform the decomposition allows to get more efficient implementation results. Pushing forward this approach could be of interest by going over the Boolean field. Under this field, the costly field multiplication becomes a simple bitwise AND and with a bitslice strategy this could allow some significant improvement in the performance gain.

The idea of working on smaller field could also be ported to generic methods. In fact, the CRV method could be extend to any finite field to produce more efficient decomposition. One could produce generic methods for any s-box that is defined with respect to the s-box input and output and the field size. Then, the developer could instantiate the decomposition with these parameters on any finite field of characteristic 2 to get the most suitable decomposition depending on his requirements. For instance, as the bitslice strategy seems to be promising to obtain efficient masking schemes, it can be interesting to have a decomposition method for generic s-boxes over the Boolean field.

Finally, most of the masking schemes are very consuming in terms of randomness requirement due to the multiplication gadgets but also the refresh gadgets. Minimizing the randomness consumption is hence an important line of work in order to produce efficient implementations of masking schemes. Several formal verification tools have been developed in the recent years to ensure the probing security of masking schemes by placing refresh gadgets at carefully chosen positions. However, it sometimes happen that those tools lead to add refresh gadgets at uncessary positions (in terms of probing security). It can be interesting to find a formal method that allows to guarantee the probing security of an implementation with a tight number of inserted refresh gadgets, compared to existing solutions, in order to lower the randomness usage.

Chapter 3

Decomposition

3.1 Introduction

As explained in Chapter **??**, the most widely-used solution is to consider the representation as a polynomial function over a finite field (using Lagrange's interpolation theorem) and to find an efficient way to evaluate this polynomial. In order to obtain an efficient implementation of an s-box protected with higher-order masking, the first challenge is to find an efficient representation of it, where the number of multiplications involved in the evaluation is minimized.

The first generic method to mask any s-box at any masking order d was proposed in 2012 by Carlet, Goubin, Prouff, Quisquater and Rivain [**FSE:CGPQR12**] (following prior work by Rivain and Prouff for the AES block cipher [**CHES:RivPro10**]). The core idea is to split the evaluation of the polynomial representation of the s-box into simple operations over $\mathbb{F}_{2^n}$ (namely, addition, multiplication by a constant, squaring and regular multiplication of two distinct elements), where n denotes the number of input bits of the s-box. Among these operations, only the regular multiplication of two distinct elements is non-linear (since squaring over a characteristic 2 finite field is linear), and one can use the secure multiplication algorithms to evaluate them [**CHES:RivPro10**; **EC:BBPPTV16**; **CHES:BCPZ16**]. Carlet *et al.* [**FSE:CGPQR12**] defined the *masking complexity* (also known as *multiplicative complexity* and *non-linear complexity*) of an s-box as the minimal number of such multiplications necessary to evaluate the corresponding polynomial and they adapted known methods for polynomial evaluation based on *cyclotomic classes* (see [**FSE:CGPQR12**] for details).

This technique was later improved by Roy and Vivek in [**CHES:RoyViv13**] using *cyclotomic cosets addition chains*. They notably presented a polynomial evaluation method for the DES s-boxes that requires 7 non-linear multiplications (instead of 10 in [**FSE:CGPQR12**]). They also presented a lower-bound on the length of such a chain and showed that the multiplicative complexity of the DES s-boxes is lower bounded by 3. In 2014, Coron, Roy and Vivek [**CHES:CorRoyViv14**] proposed a heuristic method which may be viewed as an extension of the ideas developed in [**FSE:CGPQR12**] and [**CHES:RoyViv13**]. The so-called CRV method considers the s-box as a polynomial over $\mathbb{F}_{2^n}$ and has heuristic multiplicative complexity $O(2^{n/2}/\sqrt{n})$ instead of $O(2^{n/2})$ proven multiplicative complexity for the previous methods. They also proved a lower bound of $\Omega(2^{n/2}/\sqrt{n})$ on the multiplicative complexity of any generic method to evaluate n-bit s-boxes. For all the tested s-boxes their method is at least as efficient as the previous proposals and it often requires fewer non-linear multiplications (e.g. only 4 for the DES s-boxes).

3.2 Approach

In this chapter, we propose a generalized decomposition method for s-boxes. More precisely, in our approach any $n\lambda$-bit s-box for some integers $n \geq 1$ and $\lambda \geq 1$ can be seen as a polynomial (or a vector of $m \geq 1$ polynomials) over $\mathbb{F}_{2^\lambda}^n$. We first start by defining the multiplicative complexity for s-boxes and some lower bounds. We prove a lower bound of $\Omega(2^{\lambda n/2}\sqrt{m/\lambda})$ for the complexity of any method to evaluate $n\lambda$-bit to $m\lambda$-bit s-boxes. We then describe our generalized decomposition method for which we provide concrete parameters to achieve a decomposition for several triplets (n, m, λ) and for exemplary s-boxes of popular block ciphers (namely PRESENT [**CHES:BKLPPR07**], SC2000 [**FSE:SYYTIYTT01**], CLEFIA [**FSE:SSAMI07**] and KHAZAD [**NESSIE:BarRij00**]).

Depending on the s-box, our generalized method allows one to choose the parameters n, m and λ to obtain the best possible s-box decomposition in terms of multiplications over $\mathbb{F}_{2^\lambda}$. In particular, for 8×8 s-boxes, the CRV decomposition method [**CHES:CorRoyViv14**] ($n=1$, $m=1$, $\lambda=8$) is a special case of this generalized decomposition method. We also investigate a particular case of our decomposition, the Boolean case, where $n=8$, $m=8$, $\lambda=1$. For this specific instantiation, we can apply a bitslicing strategy for the implementation part and some improvement tailored for the Boolean field.

3.3 Multiplicative Complexity of Generic S-boxes

3.3.1 Notations and notions

Let λ be a positive integer. Then $\mathbb{F}_{2^\lambda}$ denotes the finite field with 2^λ elements. Let $\mathcal{F}_{\lambda,n}$ be the set of functions from $\mathbb{F}_{2^\lambda}^n$ to $\mathbb{F}_{2^\lambda}$. Using Lagrange's interpolation theorem, any function $f\in\mathcal{F}_{\lambda,n}$ can be seen as a multivariate polynomial over $\mathbb{F}_{2^\lambda}[x_1,x_2,\ldots,x_n]/(x_1^{2^\lambda}-x_1, x_2^{2^\lambda}-x_2,\ldots,x_n^{2^\lambda}-x_n)$:

$$f(x)=\sum_{u\in[0,2^\lambda-1]^n} a_u\, x^u\ , \tag{3.1}$$

where $x=(x_1,x_2,\ldots,x_n)$, $x^u=x_1^{u_1}\cdot x_2^{u_2}\cdot\ldots\cdot x_n^{u_n}$, and $a_u\in\mathbb{F}_{2^\lambda}$ for every $u=(u_1,\ldots,u_n)\in[0,2^\lambda-1]^n$.

The *multiplicative complexity* of a function in $\mathcal{F}_{\lambda,n}$ (also called the non-linear complexity) is defined as follows:

Definition 3.3.1. *Let f be a function in $\mathcal{F}_{\lambda,n}$. The multiplicative complexity of f is the minimal number of $\mathbb{F}_{2^\lambda}$-multiplications required to evaluate it.*

In the following, an s-box S is characterized with respect to 3 parameters: the number of input elements n; the number of output elements m; and the bit-size of the elements λ. In other words, an s-box with λn input bits and λm outputs bits is represented as follows:

$$\forall x\in\mathbb{F}_{2^\lambda}^n, \mathcal{S}(x)=(f_1(x),f_2(x),\ldots,f_m(x)), \tag{3.2}$$

where the functions $f_1,f_2,\ldots,f_m\in\mathcal{F}_{\lambda,n}$ are called the *coordinate functions* of S.

The multiplicative complexity of an s-box $S: x\mapsto(f_1(x),f_2(x)\ldots,f_m(x))$, denoted $C(S)$, is naturally defined as the multiplicative complexity of the family of its coordinate functions. We shall also call the multiplicative complexity of a given circuit the actual number of multiplication gates involved in the circuit, so that the multiplicative complexity of a circuit gives an upper bound of the multiplicative complexity of the underlying s-box.

3.3.2 Multiplicative Complexity Lower Bound for S-boxes

Roy and Vivek presented in [**CHES:RoyViv13**] a lower-bound on the length of cyclotomic coset addition chains and used it to derive a logarithmic lower bound on the multiplicative complexity of an s-box (i.e. on the minimal number of such multiplications necessary to evaluate the corresponding polynomial). As mentioned in Chapter **??**, Coron *et al.* [**CHES:CorRoyViv14**] improved this lower bound and showed that the non-linear complexity of any generic method to evaluate n-bit s-boxes when seen as a polynomial defined over $\mathbb{F}_{2^n}$ is in $\Omega(2^{n/2}/\sqrt{n})$.

In the following section, we generalize their approach and provide a new lower bound on the multiplicative complexity of a sequence of n-variate polynomials over $\mathbb{F}_{2^\lambda}$. Following [**CHES:RoyViv13**], we define the multiplicative complexity notion for such a sequence as follows:

Definition 3.3.2 (Polynomial chain)**.** *Let $\lambda \geq 1$, $n \geq 1$, $l \geq 1$ and $m \geq 1$ be four integers and let $f_1, \ldots, f_m \in \mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$ be a sequence of n-variate polynomials over $\mathbb{F}_{2^\lambda}$. A* polynomial chain $\vec{\pi}$ *for $(f_1, \ldots, f_m)$ is a sequence $\vec{\pi} = (\pi_i)_{i \in \{-n,\ldots,\ell\}}$ and a list $(i_1, \ldots, i_m) \in \{-n, \ldots, \ell\}^m$ with*

$$\pi_{-n} = x_n,\ \pi_{1-n} = x_{n-1},\ \ \ldots,\ \pi_{-1} = x_1,\ \pi_0 = 1,$$

$$\pi_{i_j} = f_j(x_1, \ldots, x_n) \bmod (x_1^{2^\lambda} + x_1, \ldots, x_n^{2^\lambda} + x_n),\ \forall j \in \{1, \ldots, m\},$$

and such that for every $i \in \{1, \ldots, \ell\}$, one of the following condition holds:

1. *there exist j and k in $\{-n, \ldots, i-1\}$ such that $\pi_i = \pi_j \cdot \pi_k$;*
2. *there exist j and k in $\{-n, \ldots, i-1\}$ such that $\pi_i = \pi_j + \pi_k$;*
3. *there exists j in $\{-n, \ldots, i-1\}$ such that $\pi_i = \pi_j^2$;*
4. *there exists j in $\{-n, \ldots, i-1\}$ and $\alpha \in \mathbb{F}_{2^\lambda}$ such that $\pi_i = \alpha \cdot \pi_j$.*

Given such a polynomial chain $\vec{\pi}$ for $(f_1, \ldots, f_m)$, the multiplicative complexity of $\vec{\pi}$ *is the number of times the first condition holds in the whole chain $\vec{\pi}$. The* multiplicative complexity of $(f_1, \ldots, f_m)$ *over $\mathbb{F}_{2^\lambda}$, denoted $\mathcal{M}(f_1, \ldots, f_m)$ is the minimal multiplicative complexity over all polynomial chains for $(f_1, \ldots, f_m)$.*

We will provide a heuristic method which given a sequence of n-variate polynomials over $\mathbb{F}_{2^\lambda}$ provides an evaluation scheme (or a circuit) with "small" multiplicative complexity. Following, Coron *et al.* [**CHES:CorRoyViv14**], Proposition **??** provides a $\Omega(2^{n\lambda/2}\sqrt{m/\lambda})$ lower bound on this multiplicative complexity. As in [**CHES:CorRoyViv14**], the proof is a simple combinatorial argument inspired by [**SICOMP:PatSto73**].

Proposition 3.3.3. *Let $\lambda \geq 1$, $n \geq 1$, $l \geq 1$ and $m \geq 1$ be four integers. There exists $f_1, \ldots, f_m \in \mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$ a sequence of n-variate polynomials over $\mathbb{F}_{2^\lambda}$ such that $\mathcal{M}(f_1, \ldots, f_m) \geq \sqrt{\frac{m2^{n\lambda}}{\lambda}} - (2n + m - 1)$.*

Proof. We consider a sequence of n-variate polynomials $f_1, \ldots, f_m$ in the algebra $\mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$ with multiplicative complexity $\mathcal{M}(f_1, \ldots, f_m) = r$ for some integer $r \geq 1$. If we consider only the non-linear operations in a polynomial chain of minimal multiplicative complexity $\vec{\pi} = (\pi_i)_{i \in \{-n,\ldots,\ell\}}$, we can see that there exist indices $m_0, m_1, \ldots, m_{n+r+(m-1)}$ with $m_i \in \{-n, \ldots, \ell\}$ for $i \in \{0, \ldots, n + r + (m-1)\}$ such that

- $\pi_{m_k} = \pi_{-k-1} = x_{k+1}$ for $k \in \{0, \ldots, n-1\}$);
  (*i.e.* construction of the mononials in $\mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$.)

- for $k \in \{n, \ldots, n+r-1\}$, there exist field elements $\beta_k, \beta'_k \in \mathbb{F}_{2^\lambda}$ and $\beta_{k,i,j}, \beta'_{k,i,j} \in \mathbb{F}_{2^\lambda}$ for $(i,j) \in \{0, \ldots, k-1\} \times \{0, \ldots, \lambda-1\}$ such that

$$\pi_{m_k} = \left(\beta_k + \sum_{i=0}^{k-1}\sum_{j=0}^{\lambda-1} \beta_{k,i,j}\pi_{m_i}^{2^j}\right) \cdot \left(\beta'_k + \sum_{i=0}^{k-1}\sum_{j=0}^{\lambda-1} \beta'_{k,i,j}\pi_{m_i}^{2^j}\right) \mod (x_1^{2^\lambda} + x_1, \ldots, x_n^{2^\lambda} + x_n)$$

  (*i.e.* construction of elements in the span of previously constructed π_{m_k});

- for $k \in \{n+r, \ldots, n+r+(m-1)\}$ there exist field elements $\beta_k \in \mathbb{F}_{2^\lambda}$ and $\beta_{k,i,j} \in \mathbb{F}_{2^\lambda}$ for $(i,j) \in \{0, \ldots, n+r-1\} \times \{0, \ldots, \lambda-1\}$ such that

$$f_{k+1-(n+r)} = \pi_{m_k} = \beta_k + \sum_{i=0}^{n+r-1}\sum_{j=0}^{\lambda-1} \beta_{k,i,j}\pi_{m_i}^{2^j} \mod (x_1^{2^\lambda} + x_1, \ldots, x_n^{2^\lambda} + x_n)$$

  (*i.e.* construction of the coordinate functions with the span of previously constructed π_{m_k}).

The total number of parameters β in this evaluation scheme of $\vec{\pi}$ is simply equal to:

$$\sum_{k=n}^{n+r-1} 2 \cdot (1 + k \cdot \lambda) + m(1 + (n+r) \cdot \lambda) = r^2\lambda + r(\lambda m + 2\lambda n - \lambda + 2) + \lambda mn + m,$$

and each parameter can take any value in $\mathbb{F}_{2^\lambda}$. The number of sequence of n-variate polynomials $f_1, \ldots, f_m \in \mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$ with multiplicative complexity $\mathcal{M}(f_1, \ldots, f_m) = r$ is thus upper-bounded by $2^{\lambda(r^2\lambda + r(\lambda m + 2\lambda n - \lambda + 2) + \lambda mn + m)}$.

Since the total number of sequences of n-variate polynomials $f_1, \ldots, f_m \in \mathbb{F}_{2^\lambda}[x_1, \ldots, x_n]$ defined $\mod(x_1^{2^\lambda} + x_1, \ldots, x_n^{2^\lambda} + x_n)$ is $((2^\lambda)^{2^{n\lambda}})^m$, in order to be able to evaluate all such polynomials with at most r non-linear multiplications, a necessary condition is to have

$$r^2\lambda + r(\lambda m + 2\lambda n - \lambda + 2) + \lambda mn + m \geq m2^{n\lambda}$$

and therefore

$$r^2\lambda + r(\lambda m + 2\lambda n - \lambda + 2) - (m2^{n\lambda} - \lambda mn - m) \geq 0.$$

Eventually, we obtain

$$r \geq \frac{\sqrt{\lambda 4m2^{n\lambda} + (\lambda m + 2\lambda n - \lambda + 2)^2} - (\lambda m + 2\lambda n - \lambda + 2)}{2\lambda}, \tag{3.3}$$

and

$$r \geq \frac{\sqrt{4\lambda m2^{n\lambda}} - 2(\lambda m + 2\lambda n - \lambda + 2)}{2\lambda} \geq \sqrt{\frac{m2^{n\lambda}}{\lambda}} - (2n + m - 1).$$

□

3.3.3 Some results for the Boolean case.

In the Boolean case (when $\lambda = 1$), an s-box can be seen as Boolean circuit. We give hereafter some results from the state-of-the-art on the multiplicative complexity of s-boxes in the Boolean case. We shall also call multiplicative complexity of a given circuit the actual number of multiplication gates involved in the circuit, so that the multiplicative complexity of a circuit gives an upper bound of the multiplicative complexity of the underlying s-box.

The best known circuit for the AES s-box in terms of multiplicative complexity is due to Boyar *et al.* [**JC:BoyMatPer13**]. This circuit achieves a multiplicative complexity of 32 which was obtained by applying logic minimization techniques to the compact representation of the AES s-box by Canright [**CHES:Canright05**] (and saving 2 multiplications compared to the original circuit).

In [**AIS:CouTheHul13**], Courtois *et al.* use SAT-solving to find the multiplicative complexity of small s-boxes. Their approach consists in writing the Boolean system obtained for a given s-box and a given (target) multiplicative complexity t as a SAT-CNF problem. For each value of t, the solver either returns a solution or a proof that no solution exists, so that the multiplicative complexity is the first value of t for which a solution is returned. They apply this approach to find Boolean circuits with the smallest multiplicative complexity for a random 3×3 s-box (meant to be used in CTC2 [**EPRINT:Courtois07**]), the 4×4 s-box of PRESENT, and for several sets of 4×4 s-boxes proposed for GOST [**CHES:PosLinWan10**]. These results have recently been extended by Stoffelen who applied the Courtois *et al.* approach to find optimal circuits for various 4×4 and 5×5 s-boxes [**FSE:Stoffelen16**]. These results are summarized in Table **??**, where for comparison we also include known results for AES [**JC:BoyMatPer13**] and some bitslice-oriented block ciphers [**DPAR00**; **FSE:GLSV14**].

The main limitation of the SAT-solving approach is that it is only applicable to small s-boxes due to the combinatorial explosion of the underlying SAT problem, and getting the decomposition of an s-box of size *e.g.* $n = 8$ seems out of reach. Moreover, the method is not generic in the sense that the obtained decomposition stands for a single s-box and does not provide an upper bound for the multiplicative complexity of s-boxes of a given size.

3.3.4 Parallel Multiplicative Complexity

We introduce hereafter the notion of *parallel multiplicative complexity* for s-boxes. The k-parallel multiplicative complexity of an s-box is the least number of k-parallel multiplications required to compute it. We formalize this notion hereafter:

Definition 3.3.4. *The* k-parallel multiplicative complexity $\mathcal{M}^{(k)}(f_1, f_2, \ldots, f_m)$ *of a family of functions* $f_1, f_2, \ldots, f_m \in \mathcal{F}_{\lambda,n}$*, is the minimal integer* t *for which there exist functions* $g_i, h_i \in \mathcal{F}_{\lambda,n}$ *for* $i \in [\![1, tk]\!]$ *such that:*

$$\begin{cases} g_1, h_1, g_2, h_2, \ldots, g_k, h_k \in \langle 1, x_1, x_2, \ldots, x_n \rangle \ , \\ \forall i \in [\![1, t-1]\!] : g_{ik+1}, h_{ik+1}, \ldots, g_{(i+1)k}, h_{(i+1)k} \\ \qquad\qquad \in \langle 1, x_1, x_2, \ldots, x_n, g_1 \cdot h_1, \ldots, g_{ik} \cdot h_{ik} \rangle \end{cases} \tag{3.4}$$

and

$$f_1, f_2, \ldots, f_m \in \langle 1, x_1, x_2, \ldots, x_n, g_1 \cdot h_1, \ldots, g_{tk} \cdot h_{tk} \rangle \ . \tag{3.5}$$

Table 3.1: Multiplicative complexities of various s-boxes.

S-box S	size $n \times m$	$C(S)$	Ref
Lightweight block ciphers			
CTC2	3×3	3	[**AIS:CouTheHul13**]
PRESENT	4×4	4	[**AIS:CouTheHul13**]
Piccolo	4×4	4	[**FSE:Stoffelen16**]
RECTANGLE	4×4	4	[**FSE:Stoffelen16**]
CAESAR submissions			
LAC	4×4	4	[**FSE:Stoffelen16**]
Minalpher	4×4	5	[**FSE:Stoffelen16**]
Prøst	4×4	4	[**FSE:Stoffelen16**]
Ascon	5×5	5	[**FSE:Stoffelen16**]
ICEPOLE	5×5	6	[**FSE:Stoffelen16**]
PRIMATEs	5×5	$\{6, 7\}$	[**FSE:Stoffelen16**]
AES and Keccak			
Keccak	5×5	5	[**FSE:Stoffelen16**]
AES	8×8	≤ 32	[**JC:BoyMatPer13**]
Bitslice-oriented block ciphers			
NOKEON	4×4	4	[**DPAR00**]
Fantomas	8×8	≤ 11	[**FSE:GLSV14**]
Robin	8×8	≤ 12	[**FSE:GLSV14**]

The main motivation for introducing this notion comes from the following scenario. Assume we want to perform t s-box computations on an ℓ-bit architecture, where $\ell > t$. Since each register contains only λ bits elements out of ℓ, one can perform up to $\tau = \lceil \ell/\lambda \rceil$ multiplication with a single call to a parallelized version of the multiplication algorithm (modulo some packing of the operands and unpacking of the results, see Section **??** of Chapter **??**). This would in turn enable to perform $k = \lceil t/\tau \rceil$ s-box computations in parallel. If the used decomposition of the s-box has a k-parallel multiplicative complexity of s, then the resulting implementation involve s multiplication instructions. This number of instructions is the main efficiency criterion when such an implementation is protected with higher-order masking which makes the k-parallel multiplicative complexity an important parameter for an s-box in this context.

Remark 3.3.5. *Note that in the Boolean case, if ℓ grows significantly the k-parallel multiplicative complexity is equivalent to the notion of multiplicative depth of a Boolean circuit.*

As an illustration, we give in Figure **??** gives a sorted version of the AES circuit by Boyar *et al.* [**JC:BoyMatPer13**] optimized with respect to the parallel multiplicative complexity. One can observed 6 groups of AND gates: a first group of 8 AND gates (in blue), 4 groups of 2 AND gates (in orange), and a last group of 16 AND gates (in blue). Each group can be fully parallelized but two different groups cannot be computed in parallel due to dependence between variables. We deduce that we can evaluate the circuit based on sixteen 2-parallel AND gates, ten 4-parallel AND gates, seven 8-parallel AND gates, and six 16-parallel AND gates.

– *top linear transformation* –

$$
\begin{array}{lll|lll|lll|lll}
y_{14} &= x_3 \oplus x_5 & & y_1 &= t_0 \oplus x_7 & & y_{15} &= t_1 \oplus x_5 & & y_{17} &= y_{10} \oplus y_{11} \\
y_{13} &= x_0 \oplus x_6 & & y_4 &= y_1 \oplus x_3 & & y_{20} &= t_1 \oplus x_1 & & y_{19} &= y_{10} \oplus y_8 \\
y_{12} &= y_{13} \oplus y_{14} & & y_2 &= y_1 \oplus x_0 & & y_6 &= y_{15} \oplus x_7 & & y_{16} &= t_0 \oplus y_{11} \\
y_9 &= x_0 \oplus x_3 & & y_5 &= y_1 \oplus x_6 & & y_{10} &= y_{15} \oplus t_0 & & y_{21} &= y_{13} \oplus y_{16} \\
y_8 &= x_0 \oplus x_5 & & t_1 &= x_4 \oplus y_{12} & & y_{11} &= y_{20} \oplus y_9 & & y_{18} &= x_0 \oplus y_{16} \\
t_0 &= x_1 \oplus x_2 & & y_3 &= y_5 \oplus y_8 & & y_7 &= x_7 \oplus y_{11} & & &
\end{array}
$$

– *middle non-linear transformation* –

$$
\begin{array}{lll|lll|lll|lll}
t_2 &= y_{12} \wedge y_{15} & & t_{23} &= t_{19} \oplus y_{21} & & t_{34} &= t_{23} \oplus t_{33} & & z_2 &= t_{33} \wedge x_7 \\
t_3 &= y_3 \wedge y_6 & & t_{15} &= y_8 \wedge y_{10} & & t_{35} &= t_{27} \oplus t_{33} & & z_3 &= t_{43} \wedge y_{16} \\
t_5 &= y_4 \wedge x_7 & & t_{26} &= t_{21} \wedge t_{23} & & t_{42} &= t_{29} \oplus t_{33} & & z_4 &= t_{40} \wedge y_1 \\
t_7 &= y_{13} \wedge y_{16} & & t_{16} &= t_{15} \oplus t_{12} & & z_{14} &= t_{29} \wedge y_2 & & z_6 &= t_{42} \wedge y_{11} \\
t_8 &= y_5 \wedge y_1 & & t_{18} &= t_6 \oplus t_{16} & & t_{36} &= t_{24} \wedge t_{35} & & z_7 &= t_{45} \wedge y_{17} \\
t_{10} &= y_2 \wedge y_7 & & t_{20} &= t_{11} \oplus t_{16} & & t_{37} &= t_{36} \oplus t_{34} & & z_8 &= t_{41} \wedge y_{10} \\
t_{12} &= y_9 \wedge y_{11} & & t_{24} &= t_{20} \oplus y_{18} & & t_{38} &= t_{27} \oplus t_{36} & & z_9 &= t_{44} \wedge y_{12} \\
t_{13} &= y_{14} \wedge y_{17} & & t_{30} &= t_{23} \oplus t_{24} & & t_{39} &= t_{29} \wedge t_{38} & & z_{10} &= t_{37} \wedge y_3 \\
t_4 &= t_3 \oplus t_2 & & t_{22} &= t_{18} \oplus y_{19} & & z_5 &= t_{29} \wedge y_7 & & z_{11} &= t_{33} \wedge y_4 \\
t_6 &= t_5 \oplus t_2 & & t_{25} &= t_{21} \oplus t_{22} & & t_{44} &= t_{33} \oplus t_{37} & & z_{12} &= t_{43} \wedge y_{13} \\
t_9 &= t_8 \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_{11} &= t_{10} \oplus t_7 & & t_{31} &= t_{22} \oplus t_{26} & & t_{41} &= t_{40} \oplus t_{37} & & z_{15} &= t_{42} \wedge y_9 \\
t_{14} &= t_{13} \oplus t_{12} & & t_{28} &= t_{25} \wedge t_{27} & & t_{43} &= t_{29} \oplus t_{40} & & z_{16} &= t_{45} \wedge y_{14} \\
t_{17} &= t_4 \oplus t_{14} & & t_{32} &= t_{31} \wedge t_{30} & & t_{45} &= t_{42} \oplus t_{t41} & & z_{17} &= t_{41} \wedge y_8 \\
t_{19} &= t_9 \oplus t_{14} & & t_{29} &= t_{28} \oplus t_{22} & & z_0 &= t_{44} \wedge y_{15} & & & \\
t_{21} &= t_{17} \oplus y_{20} & & t_{33} &= t_{33} \oplus t_{24} & & z_1 &= t_{37} \wedge y_6 & & &
\end{array}
$$

– *bottom linear transformation* –

$$
\begin{array}{lll|lll|lll|lll}
t_{46} &= z_{15} \oplus z_{16} & & t_{49} &= z_9 \oplus z_{10} & & t_{61} &= z_{14} \oplus t_{57} & & t_{48} &= z_5 \oplus z_{13} \\
t_{55} &= z_{16} \oplus z_{17} & & t_{63} &= t_{49} \oplus t_{58} & & t_{65} &= t_{61} \oplus t_{62} & & t_{56} &= z_{12} \oplus t_{48} \\
t_{52} &= z_7 \oplus z_8 & & t_{66} &= z_1 \oplus t_{63} & & s_0 &= t_{59} \oplus t_{63} & & s_3 &= t_{53} \oplus t_{66} \\
t_{54} &= z_6 \oplus z_7 & & t_{62} &= t_{52} \oplus t_{58} & & t_{51} &= z_2 \oplus z_5 & & s_1 &= \overline{t_{64} \oplus s_3} \\
t_{58} &= z_4 \oplus t_{46} & & t_{53} &= z_0 \oplus z_3 & & s_4 &= t_{51} \oplus t_{66} & & s_6 &= \overline{t_{56} \oplus t_{62}} \\
t_{59} &= z_3 \oplus t_{54} & & t_{50} &= z_2 \oplus z_{12} & & s_5 &= t_{47} \oplus t_{65} & & s_7 &= \overline{t_{48} \oplus t_{60}} \\
t_{64} &= z_4 \oplus t_{59} & & t_{57} &= t_{50} \oplus t_{53} & & t_{67} &= t_{64} \oplus t_{65} & & & \\
t_{47} &= z_{10} \oplus z_{11} & & t_{60} &= t_{46} \oplus t_{57} & & s_2 &= \overline{t_{55} \oplus t_{67}} & & &
\end{array}
$$

Figure 3.1: A circuit for AES with parallelizable AND gates.

3.4 Generic Framework

In this section, we study a generic decomposition framework to evaluate any $n\,\lambda$-bit to $m\,\lambda$-bit function over a finite field $\mathbb{F}_{2^\lambda}$, for any $n, m, \lambda \geq 1$. Depending on the function, we can choose

the parameters n, m and λ in order to obtain the best possible decomposition in terms of multiplicative complexity.

3.4.1 Decomposition of a Single Coordinate Function

Let us define the *linear power class* of a function $\phi \in \mathcal{F}_{\lambda,n}$, denoted by $\mathcal{C}_\phi$, as the set

$$\mathcal{C}_\phi = \{\phi^{2^i} : i = 0, \ldots, \lambda - 1\}. \tag{3.6}$$

$\mathcal{C}_\phi$ corresponds to the set of functions in $\mathcal{F}_{\lambda,n}$ that can be computed from ϕ using only the squaring operation. It is not hard to see that the set of $\{\mathcal{C}_\phi\}_\phi$ are equivalence classes partitioning $\mathcal{F}_{\lambda,n}$. For any set $\mathcal{B} \subseteq \mathcal{F}_{\lambda,n}$, let us define the *linear power closure* of $\mathcal{B}$ as the set

$$\overline{\mathcal{B}} = \bigcup_{\phi \in \mathcal{B}} \mathcal{C}_\phi,$$

and the *linear span* of $\mathcal{B}$ as the set

$$\langle \mathcal{B} \rangle = \Big\{ \sum_{\phi \in \mathcal{B}} a_\phi \phi \mid a_\phi \in \mathbb{F}_{2^\lambda} \Big\}.$$

Let f be a function in $\mathcal{F}_{\lambda,n}$. The proposed decomposition makes use of a basis of functions $\mathcal{B} \subseteq \mathcal{F}_{\lambda,n}$ and consists in writing f as:

$$f(x) = \sum_{i=0}^{t-1} g_i(x) \cdot h_i(x) + h_t(x), \tag{3.7}$$

where $g_i, h_i \in \langle \overline{\mathcal{B}} \rangle$ and $t \in \mathbb{N}$. By definition, the functions g_i and h_i can be written as

$$g_i(x) = \sum_{j=1}^{|\mathcal{B}|} \ell_j(\varphi_j(x)) \quad \text{and} \quad h_i(x) = \sum_{j=1}^{|\mathcal{B}|} \ell'_j(\varphi_j(x)),$$

where the ℓ_j, ℓ'_j are *linearized polynomials* over $\mathcal{F}_{\lambda,1}$ (i.e. polynomials for which the exponents of all the monomials are powers of 2) and where $\{\varphi_j\}_{1 \le j \le |\mathcal{B}|} = \mathcal{B}$. We now explain how to find such a decomposition by solving a linear system.

Solving a linear system. In the following, we shall consider a basis $\mathcal{B}$ such that $1 \in \mathcal{B}$ and we will denote $\mathcal{B}^* = \mathcal{B} \setminus \{1\} = \{\phi_1, \phi_2, \ldots, \phi_{|\mathcal{B}|-1}\}$. We will further heuristically assume $|\mathcal{C}_{\phi_i}| = \lambda$ for every $i \in \{1, 2, \ldots, |\mathcal{B}| - 1\}$. We then get $|\overline{\mathcal{B}}| = 1 + \lambda|\mathcal{B}^*| = 1 + \lambda(|\mathcal{B}| - 1)$.

We first sample t random functions g_i from $\langle \overline{\mathcal{B}} \rangle$. This is simply done by picking $t \cdot |\overline{\mathcal{B}}|$ random coefficients $a_{i,0}, a_{i,j,k}$ of $\mathbb{F}_{2^\lambda}$ and setting $g_i = a_{i,0} + \sum_{j,k} a_{i,j,k} \phi_j^{2^k}$ for every $i \in [0, t-1]$ where $1 \le k \le \lambda$ and $1 \le j \le |\mathcal{B}| - 1$. Then we search for a family of $t+1$ functions $\{h_i\}_i$ satisfying Equation (**??**). This is done by solving the following system of linear equations over $\mathbb{F}_{2^\lambda}$:

$$A \cdot c = b \tag{3.8}$$

where $b = (f(e_1), f(e_2), \ldots, f(e_{2^{n\lambda}}))^\mathsf{T}$ with $\{e_i\} = \mathbb{F}_{2^\lambda}^n$ and where A is a block matrix defined as

$$A = (\mathbf{1} | A_0 | A_1 | \ \cdots \ | A_t), \tag{3.9}$$

where $\mathbf{1}$ is the all-one column vector and where

$$A_i = (A_{i,0}|A_{i,1}| \ \cdots \ |A_{i,|\mathcal{B}|-1}), \tag{3.10}$$

with

$$A_{i,0} = (g_i(e_1), g_i(e_2), \ldots, g_i(e_{2^{n\lambda}}))^{\mathsf{T}}, \tag{3.11}$$

for every $i \in [0, t]$, with

$$A_{i,j} = \begin{pmatrix} \phi_j(e_1) \cdot g_i(e_1) & \phi_j^2(e_1) \cdot g_i(e_1) & \ldots & \phi_j^{2^{\lambda-1}}(e_1) \cdot g_i(e_1) \\ \phi_j(e_2) \cdot g_i(e_2) & \phi_j^2(e_2) \cdot g_i(e_2) & \ldots & \phi_j^{2^{\lambda-1}}(e_2) \cdot g_i(e_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_j(e_{2^{n\lambda}}) \cdot g_i(e_{2^{n\lambda}}) & \phi_j^2(e_{2^{n\lambda}}) \cdot g_i(e_{2^{n\lambda}}) & \ldots & \phi_j^{2^{\lambda-1}}(e_{2^{n\lambda}}) \cdot g_i(e_{2^{n\lambda}}) \end{pmatrix}, \tag{3.12}$$

for every $i \in [0, t-1]$ and $j \in [1, |\mathcal{B}| - 1]$, and with

$$A_{t,j} = \begin{pmatrix} \phi_j(e_1) & \phi_j^2(e_1) & \ldots & \phi_j^{2^{\lambda-1}}(e_1) \\ \phi_j(e_2) & \phi_j^2(e_2) & \ldots & \phi_j^{2^{\lambda-1}}(e_2) \\ \vdots & \vdots & \ddots & \vdots \\ \phi_j(e_{2^{n\lambda}}) & \phi_j^2(e_{2^{n\lambda}}) & \ldots & \phi_j^{2^{\lambda-1}}(e_{2^{n\lambda}}) \end{pmatrix}, \tag{3.13}$$

for every $j \in [1, |\mathcal{B}| - 1]$.

It can be checked that the vector c, solution of the system, gives the coefficients of the h_i's over the basis $\overline{\mathcal{B}}$ (plus the constant term in first position). A necessary condition for this system to have a solution whatever the target vector b (*i.e.* whatever the coordinate function f) is to get a matrix A of full rank. In particular, the following inequality must hold:

$$(t+1)|\overline{\mathcal{B}}| + 1 \geq 2^{n\lambda} \ . \tag{3.14}$$

Another necessary condition to get a full-rank matrix is that the squared linear power closure $\overline{\mathcal{B}} \times \overline{\mathcal{B}}$ spans the entire space $\mathcal{F}_{\lambda,n}$. More details about the choice of such basis are discussed in the following.

3.4.2 S-box Decomposition

Let $S\colon x \mapsto (f_1(x), f_2(x), \ldots, f_m(x))$ be an s-box as defined in Section **??**. We could apply the above decomposition method to each of the m coordinate functions f_i, which could roughly result in multiplying by m the multiplicative complexity of a single function in $\mathcal{F}_{\lambda,n}$. As suggested in [**JC:BoyMatPer13**] to construct an efficient circuit for the inversion in $\mathbb{F}_{16}$, we can actually do better: the product involved in the decomposition of a coordinate function can be added to the basis for the subsequent decompositions. Indeed, the term $g_{i,j}(x) \cdot h_{i,j}(x)$ involved in the computation of $f_i(x)$ can be reused in the computation of the following $f_{i+1}(x)$, ..., $f_m(x)$. In our decomposition process, this means that the $g_j \cdot h_{i,j}$ functions can be added to the basis for the decomposition of the next coordinate functions

$f_{i+1}, \ldots, f_m$. Specifically, we start with some basis $\mathcal{B}_1$ and, for every $i \geq 1$, we look for a decomposition

$$f_i(x) = \sum_{j=0}^{t_i-1} g_{i,j}(x) \cdot h_{i,j}(x) + h_{i,t_i}(x), \tag{3.15}$$

where $t_i \in \mathbb{N}$ and $g_{i,j}, h_{i,j} \in \langle \overline{\mathcal{B}}_i \rangle$. Once such a decomposition has been found, we carry on with the new basis $\mathcal{B}_{i+1}$ defined as:

$$\mathcal{B}_{i+1} = \mathcal{B}_i \cup \{g_{i,j} \cdot h_{i,j}\}_{j=0}^{t_i-1}. \tag{3.16}$$

This update process implies that, for each decomposition, the basis grows and hence the number t_i of multiplicative terms in the decomposition of f_i might decrease. In this context, the necessary condition on the matrix rank (see Equation (**??**)) is different for every i. In particular, the number t_i of multiplications at step i satisfies:

$$t_i \geq \frac{2^{n\lambda} - 1}{\lambda |\mathcal{B}_i^*| + 1} - 1 \ , \tag{3.17}$$

where as above $\mathcal{B}_i^*$ stands for $\mathcal{B}_i \setminus \{1\}$.

3.4.3 Optimal parameters

Assuming that satisfying the lower bound on t_i (see Equation (**??**)) is sufficient to get a full-rank system, we can deduce optimal parameters for our generalized decomposition method. Specifically, if we denote $s_i = |\mathcal{B}_i^*|$, we get a sequence $(s_i)_i$ that satisfies

$$\begin{cases} s_1 = r + n \\ s_{i+1} = s_i + t_i \quad \text{with} \quad t_i = \left\lceil \frac{2^{n\lambda}-1}{\lambda s_i + 1} \right\rceil - 1 \end{cases} , \tag{3.18}$$

for i from 1 to $m-1$, where r denotes the number of multiplications involved in the construction of the first basis $\mathcal{B}_1$ (the n *free* elements of $\mathcal{B}_1$ being the monomials x_1, x_2, …, x_n). From this sequence, we can determine the optimal multiplicative complexity of the method $C(S)$ which then satisfies

$$C(S) = \min_{r \geq r_0} (r + t_1 + t_2 + \cdots + t_m) \ , \tag{3.19}$$

where r_0 denotes the minimal value of r for which we can get an initial basis $\mathcal{B}_1$ satisfying the spanning property (that is $\langle \overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1 \rangle = \mathcal{F}_{\lambda,n}$) and where the t_i's are viewed as functions of r according to the sequence Equation (**??**).

Table **??** provides a set of optimal parameters r, t_1, t_2, …, t_m and corresponding $C(S)$ for several s-box sizes and several parameters λ and $n = m$ (as for bijective s-boxes). For the sake of completeness, we included the extreme cases $n = 1$, *i.e.* standard CRV method [**CHES:CorRoyViv14**], and $\lambda = 1$, *i.e.* Boolean case [**CHES:GouRiv16**]. We obtain the same results as in [**CHES:CorRoyViv14**] for the standard CRV method. For the Boolean case, our results slightly differ from [**CHES:GouRiv16**]. This is due to our improved generation of $\mathcal{B}_1$ and to our bound on the t_i's (see Equation (**??**)) which is slightly more accurate than in [**CHES:GouRiv16**].

Table 3.2: Theoretical optimal parameters for our decomposition method.

(λ, n)	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$
4-bit s-boxes				
(1,4)	7	2	2,1,1,1	7
	8	3	1,1,1,1	7
	9	4	1,1,1,1	8
(2,2)	4	1	2,1	4
	5	2	1,1	4
	6	3	1,1	5
(4,1)	3	1	1	2
	4	2	1	3
6-bit s-boxes				
(1,6)	14	7	4,3,2,2,2,2	22
	15	8	4,3,2,2,2,2	23
(2,3)	8	4	4,2,2	12
	9	5	3,2,2	12
	10	6	3,2,2	13
(3,2)	6	3	3,2	8
	7	4	3,2	9
	8	5	2,2	9
(6,1)	4	2	3	5
	5	3	2	5
	6	4	2	6
8-bit s-boxes				
(1,8)	24	17	9,7,6,5,4,4,4,3	59
	25	18	9,7,5,5,4,4,4,3	59
	28	19	9,6,5,5,4,4,4,3	59
	29	20	8,6,5,5,4,4,4,3	59
	30	21	8,6,5,5,4,4,4,3	60
(2,4)	15	9	9,5,4,4	31
	16	10	8,5,4,4	31
	17	11	8,5,4,3	31
	18	12	7,5,4,3	31
	19	13	7,5,4,3	32
(4,2)	8	5	8,4	17
	9	6	7,4	17
	10	7	6,4	17
	11	8	6,3	17
	12	9	5,3	17
	13	10	5,3	18
(8,1)	5	3	7	10
	6	4	6	10
	7	5	5	10
	8	6	4	10
	9	7	3	10
	10	8	3	11

(λ, n)	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$
9-bit s-boxes				
(1,9)	35	25	14,10,8,7,6,6,5,5,5	91
	36	26	14,10,8,7,6,6,5,5,5	92
	37	27	13,10,8,7,6,6,5,5,5	92
(3,3)	13	9	13,6,5	33
	14	10	12,6,5	33
	15	11	11,6,5	33
	16	12	11,6,5	34
(9,1)	8	6	7	13
	9	7	6	13
	10	8	6	14
10-bit s-boxes				
(1,10)	49	38	20,14,12,10,9,8,8,7,7,7	140
	50	39	20,14,12,10,9,8,8,7,7,7	141
	51	40	20,14,12,10,9,8,8,7,7,7	142
(2,5)	25	19	20,11,9,7,7	73
	26	20	20,11,9,7,7	74
	27	21	19,11,9,7,7	74
(5,2)	13	10	16,7	33
	14	11	15,7	33
	15	12	14,7	33
	16	13	13,7	33
	17	14	12,7	33
	18	15	11,7	33
	19	16	11,7	34
(10,1)	9	7	12	19
	10	8	11	19
	11	9	10	19
	12	10	9	19
	13	11	8	19
	14	12	7	19
	15	13	7	20

3.4.4 Basis Selection

Let us recall that the basis $\mathcal{B}_1$ needs to be such that the squared basis $\overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1$ spans the entire space $\mathcal{F}_{\lambda,n}$, *i.e.* $\langle \overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1 \rangle = \mathcal{F}_{\lambda,n}$ in order to have a solvable linear system. This is called the *spanning property* in the following. This property can be rewritten in terms of linear algebra. For every $\mathcal{S} \subseteq \mathcal{F}_{\lambda,n}$, let us define $\mathrm{Mat}(\mathcal{S})$ as the $(\lambda n \times |\mathcal{S}|)$-matrix for which each column corresponds to the evaluation of one function of $\mathcal{S}$ in every point of $\mathbb{F}_{2^\lambda}^n$, that is

$$\mathrm{Mat}(\mathcal{S}) = \begin{pmatrix} \varphi_1(e_1) & \varphi_2(e_1) & \dots & \varphi_{|\mathcal{S}|}(e_1) \\ \varphi_1(e_2) & \varphi_2(e_2) & \dots & \varphi_{|\mathcal{S}|}(e_2) \\ \vdots & \vdots & \ddots & \vdots \\ \varphi_1(e_{2^{n\lambda}}) & \varphi_2(e_{2^{n\lambda}}) & \dots & \varphi_{|\mathcal{S}|}(e_{2^{n\lambda}}) \end{pmatrix}, \tag{3.20}$$

where $\{\varphi_1, \varphi_2, \dots, \varphi_{|\mathcal{S}|}\} = \mathcal{S}$. Then, we have

$$\langle \overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1 \rangle = \mathcal{F}_{\lambda,n} \iff \mathrm{rank}(\mathrm{Mat}(\overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1)) = 2^{\lambda n}. \tag{3.21}$$

To construct the basis $\mathcal{B}_1$, we proceed as follows. We start with the basis composed of all monomials of degree 1 plus unity, *i.e.* the following basis:

$$\mathcal{B}_1 = \{1, x_1, x_2, \dots, x_n\}. \tag{3.22}$$

Then, we iterate $\mathcal{B}_1 \leftarrow \mathcal{B}_1 \cup \{\phi \cdot \psi\}$, where ϕ and ψ are randomly sampled from $\langle \overline{\mathcal{B}}_1 \rangle$ until reaching a basis with the desired cardinality and satisfying $\mathrm{rank}(\mathrm{Mat}(\overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1)) \geq 2^{\lambda n}$. We add the constraint that, at each iteration, a certain amount of possible products are tried and only the best product is added to the basis, namely the one inducing the greatest increase in the rank of $\mathrm{Mat}(\overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1)$. To summarize, the construction of the basis $\mathcal{B}_1$ is given in Algorithm **??**.

Algorithm 1 $\mathcal{B}_1$ construction algorithm

Input: Parameters λ, n, and N
Output: A basis $\mathcal{B}_1$ such that $\langle \overline{\mathcal{B}}_1 \times \overline{\mathcal{B}}_1 \rangle = \mathcal{F}_{\lambda,n}$

$\mathcal{B}_1 = \{1, x_1, x_2 \dots, x_n\}$
rank $= 0$
while rank$< 2^{n\lambda}$ **do**
  for $i = 1$ to N **do**
    $\phi, \psi \xleftarrow{\$} \langle \overline{\mathcal{B}}_1 \rangle$
    $\mathcal{S}_i \leftarrow \mathcal{B}_1 \cup \{\phi \cdot \psi\}$
    $r_i \leftarrow \mathrm{rank}(\mathrm{Mat}(\mathcal{S}_i \times \mathcal{S}_i))$
  $j \leftarrow \mathrm{argmax}\ r_i$
  if $r_j =$ rank **then return** error
  rank $\leftarrow r_j$
  $\mathcal{B}_1 \leftarrow \mathcal{S}_j$
return $\mathcal{B}_1$

Table **??** gives the size of the smallest randomized basis we could achieve using Algorithm **??** for various parameters. The number of tries was $N = 1000$ before adding a product of random linear combination to the current basis.

Table 3.3: Achievable smallest randomized basis computed according to Algorithm **??**.

	4-bit s-boxes			5-bit s-boxes		6-bit s-boxes				7-bit s-boxes	
(λ, n)	(1,4)	(2,2)	(4,1)	(1,5)	(5,1)	(1,6)	(2,3)	(3,2)	(6,1)	(1,7)	(7,1)
$\lvert\mathcal{B}_1\rvert$	7	4	3	10	4	14	8	6	4	19	4
r	2	1	1	4	2	7	4	3	2	11	2

	8-bit s-boxes				9-bit s-boxes			10-bit s-boxes			
(λ, n)	(1,8)	(2,4)	(4,2)	(8,1)	(1,9)	(3,3)	(9,1)	(1,10)	(2,5)	(5,2)	(10,1)
$\lvert\mathcal{B}_1\rvert$	26	14	8	5	35	13	5	49	25	11	6
r	17	9	5	3	25	9	3	38	19	8	4

3.5 Application

3.5.1 Boolean case

We first study our decomposition method for the case where we want to decompose s-boxes over the Boolean field, namely the case where $n = m$ and $\lambda = 1$. We first exhibit an improvement, called the rank drop improvement, in order to obtain decomposition leading to more efficient implementations and then provide some experimental results on several s-boxes. Finally, we exploit the fact that our method is highly parallelizable to get more efficient decompositions on the Boolean field.

3.5.1.1 Exploiting rank drops

The rank drop improvement is based on the observation that even if the matrix A is not full-rank, the obtained system can still have a solution for a given s-box. Specifically, if A is of rank $2^n - \delta$ then we should get a solution for one s-box out of 2^δ in average. Hence, instead of having t_i satisfying the condition $(t_i + 1)|\mathcal{B}| \geq 2^n$, we allow a rank drop in the system of equations, by taking $t_i \geq \frac{2^n - \delta}{|\mathcal{B}_i|} - 1$ for some integer δ for which solving 2^δ systems is affordable. We hence hope to get smaller values of t_i by trying 2^δ systems. Note that heuristically, we can only hope to achieve the above bound if δ is slightly lower than the maximal rank 2^n (*e.g.* $\delta \leq \frac{2^n}{4}$). As an illustration, Table **??** provides the obtained parameters for a δ up to 32. In the last column, we can see that the rank-drop improvement (theoretically) saves a few multiplications.

Remark 3.5.1. *When $\lambda = 1$, the probability of getting a solvable system should be around $(\frac{1}{2})^\delta$, which means that by trying about 2^δ random systems we should find a solution with a rank drop. For other values of λ, the possible gain from this improvement becomes marginal as the number of random systems to try grows exponentially in λ. Therefore we choose to only consider the rank-drop improvement for $\lambda = 1$.*

In practice, we observe that the condition $(t_i + 1)|\mathcal{B}_i| \geq 2^n - \delta$ is not always sufficient to get a matrix A of rank $2^n - \delta$. We shall then start with $t_i = \frac{2^n - \delta}{|\mathcal{B}_i|} - 1$ and try to solve $\alpha \cdot 2^\delta$ systems, for some constant α. In case of failure, we increment t_i and start again until a solvable system is found. The overall process is summarized in Algorithm **??**.

Table 3.4: Optimal parameters with rank-drop improvements.

n	δ	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$	Gain
4	4	7	2	1,1,1,1	6	1
5	8	11	5	2,1,1,1,1	11	2
	8	12	6	1,1,1,1,1	11	2
6	16	15	8	3,2,2,2,1,1	19	4
	16	16	9	2,2,2,2,1,1	19	4
7	32	23	15	4,3,3,2,2,2,2	33	5
	32	24	16	3,3,3,2,2,2,2	33	5
8	32	31	22	7,5,5,4,4,3,3,3	56	5
8	32	32	23	6,5,5,4,4,3,3,3	56	5
9	32	47	37	10,8,7,6,6,5,5,5,4	93	3
	32	48	38	9,8,7,6,6,5,5,5,4	93	3
	32	63	52	15,12,11,9,9,8,7,7,7,6	143	2
	32	64	53	15,12,10,9,9,8,7,7,7,6	143	2
10	32	65	54	15,12,10,9,8,8,7,7,7,6	143	2
	32	66	55	15,12,10,9,8,8,7,7,6,6	143	2
	32	67	56	14,12,10,9,8,8,7,7,6,6	143	2

The execution time of Algorithm **??** is dominated by the calls to a linear-solving procedure (Step 6). The number of trials is in $o(n\,\alpha\,2^{\delta})$, where the constant in the $o(\cdot)$ is the average incrementation of t_i (*i.e.* the average number of times Step 13 is executed per i). In our experiments, we observed that the optimal value of $t_1 = \frac{2^n-\delta}{s_1} - 1$ is rarely enough to get a solvable system for f_1. This is because we start with the minimal basis as in the single-Boolean-function case. We hence have a few incrementations for $i = 1$. On the other hand, the next optimal t_i's are often enough or incremented a single time.

3.5.1.2 Experimental results

Selected s-boxes. We used Algorithm **??** to obtain the decomposition of various $n \times n$ s-boxes for $n \in [\![4, 8]\!]$, namely the eight 4×4 s-boxes of Serpent [**WEB:AndBihKnu98**], the s-boxes S_5 (5×5) and S_6 (6×6) of SC2000 [**FSE:SYYTIYTT01**], the 8×8 s-boxes S_0 and S_1 of CLEFIA [**FSE:SSAMI07**], and the 8×8 s-box of Khazad [**NESSIE:BarRij00**]. The obtained results are summarized in Table **??**. Note that we chose these s-boxes to serve as examples for our decomposition method. Some of them may have a mathematical structure allowing more efficient decomposition (*e.g.* the CLEFIA S_0 s-box is based on the inversion over $\mathbb{F}_{256}$ and can therefore be computed with a 32-multiplication circuit as the AES). We further give in appendix the obtained 2-parallel decomposition for the s-boxes of SC2000, CLEFIA and Khazad.

We observe that Algorithm **??** (almost) achieves the optimal parameters given in Table **??** for $n \in \{4, 5, 6\}$. For $n = 8$, we only get the optimal multiplicative complexity of the generalized method but without the rank-drop improvement. This might be due to the fact that when n increases the value of δ becomes small compared to 2^n and the impact of exhaustive search is lowered.

Algorithm 2 Boolean decomposition with exhaustive search

Input: An s-box $S \equiv (f_1, f_2, \ldots, f_m)$, parameters $s_1 = |\mathcal{B}_1|$, α, and δ
Output: A basis $\mathcal{B}_1$ and the functions $\{h_{i,j}\}_{i,j}$ and $\{g_{i,j}\}_{i,j}$

1: $i = 1$; $t_1 = \frac{2^n - \delta}{|s_1|} - 1$
2: **do** $\alpha \cdot 2^\delta$ **times**:
3: **if** $i = 1$ **then** randomly generate $\mathcal{B}_1 \supseteq \mathcal{B}_0$ with $|\mathcal{B}_1| = s_1$
4: randomly sample t_i functions $g_{i,j} \in \langle \mathcal{B}_i \rangle$
5: compute the corresponding matrix A
6: **if** $A \cdot c = b_{f_i}$ has a solution **then**
7: store the corresponding functions $\{h_{i,j}\}_j$ and $\{g_{i,j}\}_j$
8: **if** $i = n$ **then return** $\mathcal{B}_1$, $\{h_{i,j}\}_{i,j}$, $\{g_{i,j}\}_{i,j}$
9: $\mathcal{B}_{i+1} = \mathcal{B}_i \cup \{h_{i,j} \cdot g_{i,j}\}_j$; $t_{i+1} = \frac{2^n - \delta}{|\mathcal{B}_{i+1}|} - 1$; i++
10: **goto** Step 2
11: **endif**
12: **enddo**
13: t_i++; **goto** Step 2

Random s-boxes. We also applied Algorithm **??** on random s-boxes to observe the average effectiveness of our method. Specifically, we generated 1000 s-boxes (using the Knuth shuffle algorithm) and we decomposed each of these s-boxes using Algorithm **??** with the following parameters: $n = 8$, $\alpha = 8$ and $\delta = 16$. The obtained multiplicative complexities were 59 (for 9 s-boxes), 60 (for 536 s-boxes), and 61 (455 s-boxes), which is equivalent to what we got for Khazad and CLEFIA in half of the trials and better (by 1 or 2 multiplications) for the rest. These results are summarized in Table **??**.

We also looked at the influence of the two parameters α and δ on our decomposition method. For this purpose, we applied Algorithm **??** to 1000 random s-boxes with smaller parameters α and δ. The results are summarized in Figure **??**. We observe a mean difference of 1.68 multiplications between the two extreme pairs of parameters *i.e.* $(\alpha, \delta) = (4, 4)$ and $(\alpha, \delta) = (16, 8)$. As expected, we also see that the δ parameter has more impact than the α parameter.

3.5.1.3 Parallelization

The decomposition method over $\mathbb{F}_2$ is highly parallelizable. In practice, most SPN block ciphers have a nonlinear layer applying 16 or 32 s-boxes and most processors are based on a 32-bit or a 64-bit architecture. Therefore we shall focus our study on the k-parallel multiplicative complexity of our method for $k \in \{2, 4\}$.

The parallelization of the method is slightly tricky since all the multiplicative terms $g_{i,j} \cdot h_{i,j}$ cannot be computed in parallel. Indeed, the resulting products are fed to the basis so that they are potentially involved in the linear combinations producing the next functions $g_{i+1,j}$, $h_{i+1,j}$, $\ldots$, $g_{m,j}$, $h_{m,j}$. In order to fully parallelize our method we customize Algorithm **??** as follows. We keep a counter q of the number of products added to the basis. Each time a new f_{i+1} is to be decomposed, if the current counter q is not a multiple of k, then the first q_0 products $g_{i+1,j} \cdot h_{i+1,j}$ will be bundled with the last q_1 products $g_{i,j} \cdot h_{i,j}$ in the parallel

Table 3.5: Achieved parameters for several s-boxes.

	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$
$n = 4$				
Serpent S_2	7	2	1, 1, 0, 1	5
Serpent S_7	7	2	1, 0, 1, 1	5
Serpent S_1, S_3–S_6, S_8	7	2	1, 1, 1, 1	6
$n = 5$				
SC2000 S_5	11	5	2, 2, 1, 1, 1	12
$n = 6$				
SC2000 S_6	15	8	3, 2, 2, 2, 1, 1	19
$n = 8$				
Khazad &	31	22	11, 5, 5, 4, 4, 3, 3, 3	61
CLEFIA (S_0, S_1)	31	22	9, 6, 5, 4, 4, 4, 3, 3	61
	31	22	11, 5, 5, 4, 4, 3, 3, 3	61

Table 3.6: Results of Algorithm **??** for random s-boxes ($n = 8$, $\alpha = 8$, $\delta = 16$).

$C(S)$	59	60	61
# s-boxes (over 1000)	9	536	455

version of our improved decomposition, where

$$\begin{cases} q_0 = (k - q) \bmod k; \\ q_1 = q \bmod k. \end{cases} \tag{3.23}$$

We must then ensure that the functions $\{g_{i+1,j}, h_{i+1,j}\}_{j=0}^{q_0-1}$ are independent of the few last products $\{g_{i,j} \cdot h_{i,j}\}_{j=t_i-q_1}^{t_i-1}$. This can be done at no cost for the $g_{i+1,j}$'s which can be generated without the last q_1 products, and this adds a constraint on the linear system for the first q_0 searched $h_{i+1,j}$ functions.

We observed in our experiments that for small values of k, the parallelization constraint has a limited impact on Algorithm **??**. We actually obtained the same parameters as in Table **??** for all the selected s-boxes (Serpent, SC2000, CLEFIA, and Khazad) for a parallelization degree of $k = 2$, except for the s-box S_3 of Serpent that requires 1 more multiplication. The decomposition for the s-boxes of SC2000, CLEFIA and Khazad given in appendix have been obtained for the 2-parallel version of Algorithm **??**. For these decompositions, all the multiplications can be bundled by pair.

We also experimented on random s-boxes for parallelization degrees $k = 2$ and $k = 4$ with parameters $n = 8$, $\alpha = 8$, $\delta = 16$. These results are summarized in Table **??**. We see that the obtained parallel multiplicative complexity is only one or two parallel multiplications more than this optimal $\lceil C(S)/k \rceil$ (and sometimes achieves the optimal).

We obtained that 90% of the s-boxes have a 2-parallel multiplicative complexity of 31 (which means one more multiplication is needed in 50% of the case with respect to the theoretical optimum) and 32 for 10% of the cases. For 4-parallel multiplicative complexity, all

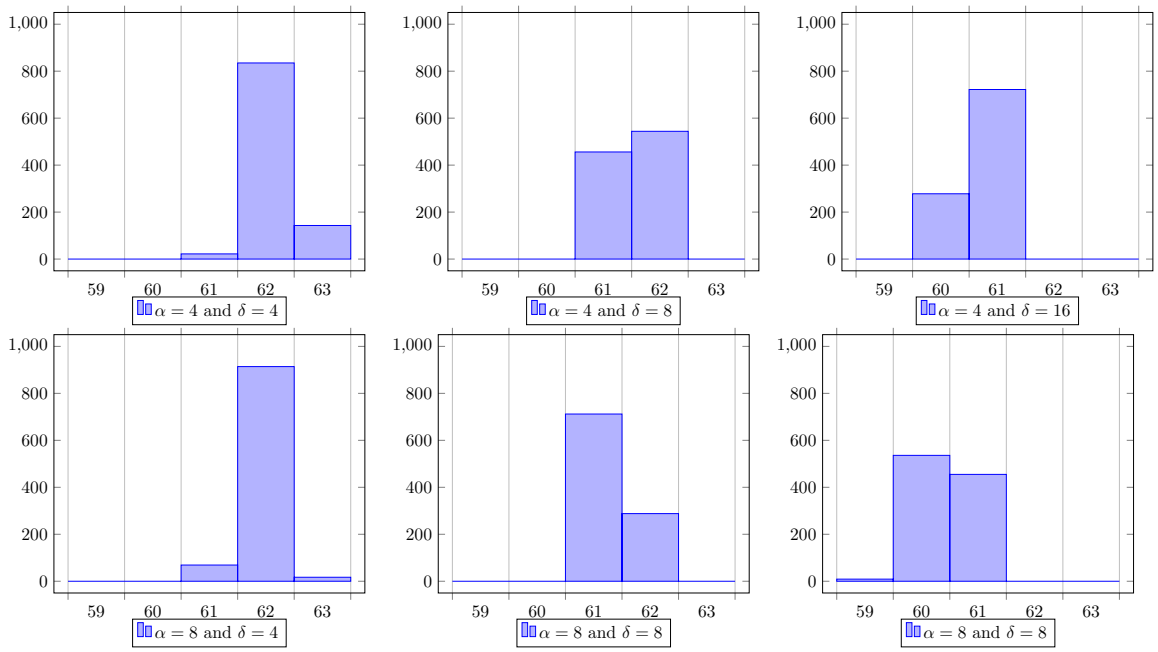


Figure 3.2: Number of Random S-boxes for each Multiplicative Complexity for $n = 8$

Table 3.7: Results of Algorithm **??** for random s-boxes ($n = 8$, $\alpha = 8$, $\delta = 16$).

Parallel mult. complexities	$C(S) =$ 59	60	61	$C^{*,(2)} =$ 31	32	$C^{*,(4)} =$ 16
# s-boxes (over 1000)	9	536	455	900	100	1000

the random s-boxes are evaluated with 16 multiplications, which means 1 more multiplication in 50% of the cases.

3.5.2 Median case

We now report some experimental results achieved using our generalized decomposition method on different values for n, m, and λ. Table **??** compares the achievable parameters vs. the optimal estimate for random s-boxes. Note that in the table, the parameters $|\mathcal{B}_1|$, r, $t_1, t_2, \ldots, t_n$ correspond to the parameters in the achievable decomposition for randomly chosen s-boxes. The last column gives the probability of obtaining a successful decomposition for random s-boxes and for randomly chosen coefficients in the basis computation as well as the decomposition step. In all the cases we made 10 trials to compute the probabilities (except for 8-bit s-boxes over $\mathbb{F}_{2^2}$ where 100 trials were made).

Table 3.8: Optimal and achievable parameters for random s-boxes.

Optimal/Achievable	(λ, n)	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$	proba.
4-bit s-boxes						
Optimal	(2,2)	5	2	1,1	4	-
Achievable	(2,2)	5	2	1,1	4	0.2
6-bit s-boxes						
Optimal	(2,3)	8	4	4,2,2	12	-
Achievable	(2,3)	8	4	5,2,2	13	0.3
Optimal	(3,2)	6	3	3,2	8	-
Achievable	(3,2)	6	3	4,2	9	0.9
8-bit s-boxes						
Optimal	(2,4)	16	11	8,5,4,3	31	-
Achievable	(2,4)	16	11	9,6,5,3	34	0.02
Optimal	(4,2)	10	7	6,4	17	-
Achievable	(4,2)	10	7	7,4	18	1.0
9-bit s-boxes						
Optimal	(3,3)	15	11	11,6,5	33	-
Achievable	(3,3)	15	11	14,6,5	36	0.8

In the experiments, successive basis elements were added by products of random linear combinations of elements from the current basis. The basis $\mathcal{B}_1$ was chosen such that the corresponding matrix for the first coordinate function resulted in full rank (implying that the spanning property of the basis $\mathcal{B}_1$ was satisfied). The basis was successively updated with the t_i products formed in the decomposition step of the ith coordinate function. While the parameter t_1 is invariant of the chosen s-box, the other t_i are indeed dependent on it. As we see from Table **??**, the probabilities increase with the size of the field used for the decomposition.

Table **??** gives the concrete parameters to achieve decomposition for s-boxes of popular block ciphers (namely PRESENT [**CHES:BKLPPR07**], DES S1 and S8 [**DES77**], SC2000 S6 [**FSE:SYYTIYTT01**], CLEFIA S0 and S1 [**FSE:SSAMI07**] and KHAZAD [**NESSIE:BarRij00**]). Note that for all the cases considered the parameters from Table **??**

yield a decomposition. As above, the last column of the table gives the success probability over the random choice of the coefficients in the basis computation as well as the decomposition step. Here again, in all the cases we made 10 trials to compute the probabilities (except for 8-bit s-boxes over $\mathbb{F}_{2^2}$ where 100 trials were made).

Table 3.9: Achievable parameters to decompose specific s-boxes.

s-box	(λ, n)	$\lvert\mathcal{B}_1\rvert$	r	$t_1, t_2, \ldots, t_n$	$C(S)$	proba.
4-bit s-boxes						
PRESENT [**CHES:BKLPPR07**]	(2,2)	5	2	1,1	4	0.3
(6,4)-bit s-boxes						
DES S1 [**DES77**]	(2,3)	7	4	5,2	11	0.3
DES S8 [**DES77**]	(2,3)	7	4	5,2	11	0.5
6-bit s-boxes						
SC2000 S6 [**FSE:SYYTIYTT01**]	(2,3)	8	4	5,2,2	13	0.2
SC2000 S6 [**FSE:SYYTIYTT01**]	(3,2)	6	3	4,2	9	0.8
8-bit s-boxes						
CLEFIA S0 [**FSE:SSAMI07**]	(4,2)	10	7	7,4	18	1.0
CLEFIA S0 [**FSE:SSAMI07**]	(2,4)	16	11	9,5,4,3	32	0.01
CLEFIA S1 [**FSE:SSAMI07**]	(4,2)	10	7	7,4	18	1.0
CLEFIA S1 [**FSE:SSAMI07**]	(2,4)	16	11	9,6,5,3	34	0.01
KHAZAD [**NESSIE:BarRij00**]	(4,2)	10	7	7,4	18	1.0
KHAZAD [**NESSIE:BarRij00**]	(2,4)	16	11	9,5,4,3	32	0.02

3.5.3 Full-field case

We now focus on the full-field case, namely when $n = m = 1$. This case is basically the CRV decomposition method introduced in Section **??**. In this case, we propose an improvement where we had some small structure to the basis to get a more efficient software implementation. Note that for this specific case, namely $n = 1$, the linear power classes introduced in Section **??** are equivalent to cyclotomic classes introduced in Section **??**.

3.5.3.1 Improving CRV with CPRR

As suggested in [**C:CPRR15**], CRV can be improved by using CPRR evaluations instead of ISW multiplications in the first phase of CRV. In Chapter **??**, we demonstrate that depending on the field size the CPRR is indeed faster than ISW (*i.e.* when full-table multiplication cannot be afforded). Instead of multiplying two previously computed powers x^{α_j} and x^{α_k}, the new power x^{α_i} is derived by applying the quadratic function $x \mapsto x^{2^w+1}$ for some $w \in [\![1, \lambda - 1]\!]$. In the masking world, securely evaluating such a function can be done with a call to CPRR. The new chain of cyclotomic classes $C_{\alpha_1=0} \cup C_{\alpha_2=1} \cup C_{\alpha_3} \cup \ldots \cup C_{\alpha_\ell}$ must then satisfy $\alpha_i = (2^w + 1)\alpha_j$ for some $j < i$ and $w \in [\![1, \lambda - 1]\!]$. The different conditions to get a solvable system in the case of the CRV decomposition (see Section **??**) are recalled here:

1. the set L of cyclotomic classes is such that $t \cdot |L| \geq 2^n$;

2. all the monomials can be reached by multiplying two monomials from x^L, that is $\{x^i \cdot x^j \bmod (x^{2^n} - x)\ ;\ i, j \in L\} = x^{[\![0,2^n-1]\!]}$; and

3. every class (but $C_0 = \{0\}$) have the maximal cardinality of n. Under this additional constraint, Condition **??** leads to the following inequality: $t \cdot (1 + n \cdot (r-1)) \geq 2^n$.

We have implemented the search of such chains of cyclotomic classes satisfying conditions **??**,**??**, and **??** . We were able to validate that for every $\lambda \in [\![4, 10]\!]$ and for the parameters (r, t) given in [**CHES:CorRoyViv14**], we always find such a chain leading to a solvable system. For the sake of code compactness, we also tried to minimize the number of CPRR exponents $2^w + 1$ used in these chains (since in practice each function $x \mapsto x^{2^w+1}$ is tabulated). For $\lambda \in \{4, 6, 7\}$ a single CPRR exponent (either 3 or 5) is sufficient to get a *satisfying chain* (*i.e.* a chain of cyclotomic classes fulfilling the above conditions and leading to a solvable system). For the other values of λ, we could prove that a single CPRR exponent does not suffice to get a satisfying chain. We could then find satisfying chains for $\lambda = 5$ and $\lambda = 8$ using 2 CPRR exponents (specifically 3 and 5). For $\lambda > 8$, we tried all the pairs and triplets of possible CPRR exponents without success, we could only find a satisfying chain using the 4 CPRR exponents 3, 5, 9 and 17.

3.5.3.2 Optimizing CRV parameters.

We can still improve CRV by optimizing the parameters (r, t) depending on the ratio $\theta = \frac{C_{\text{CPRR}}}{C_{\text{ISW}}}$, where C_{CPRR} and C_{ISW} denote the costs of ISW and CPRR respectively. The cost of the CRV method satisfies

$$\begin{aligned} C_{\text{CRV}} &= (r-2)\, C_{\text{CPRR}} + (t-1)\, C_{\text{ISW}} = \big((r-2)\cdot\theta + t - 1)\big)\, C_{\text{ISW}} \\ &\geq \Big((r-2)\cdot\theta + \lceil \frac{2^\lambda}{(r-1)\cdot\lambda + 1} \rceil - 1\Big)\, C_{\text{ISW}}, \end{aligned}$$

where the inequality holds from conditions **??** and **??**. This lower bound ensures that the system contains enough unknowns to be solvable. In practice, it was observed in [**CHES:CorRoyViv14**] that this is a sufficient condition most of the time to get a solvable system (and our experiments corroborate this fact). Our optimized version of CRV hence consists in using the parameter r minimizing the above lower bound and the corresponding $t = \lceil \frac{2^\lambda}{(r-1)\cdot\lambda+1} \rceil$ as parameters for given bit-length n and cost ratio θ.

As an illustration, Figure **??** plots the optimal parameter r with respect to the ratio θ for several values of λ. We observe that a ratio slightly lower than 1 implies a change of optimal parameters for all values of n except 4 and 9. In other words, as soon as CPRR is slightly faster than ISW, using a higher r (*i.e.* more cyclotomic classes) and therefore a lower t is a sound trade. For our implementations of ISW and CPRR (see Chapter **??**), we obtained a ratio θ greater than 1 only when ISW is based on the full-table multiplication. In that case, no gain can be obtain from using CPRR in the first phase of CRV, and one should use the original CRV parameters. On the other hand, we obtained θ-ratios of 0.88 and 0.75 for half-table-based ISW and exp-log-based ISW respectively. For the parallel versions of the secure multiplications, these ratios become 0.69 (half-table ISW) and 0.58 (exp-log ISW). We can observe from Figure **??** that for $\theta \in [0.58, 0.88]$, the optimal CRV parameters are constants for all values of λ except for $\lambda = 9$ for which a gap occurs around $\theta \approx 0.66$. Figures **??**–**??** plot the d^2 constant in the CRV cost obtained with these ratios for different values of

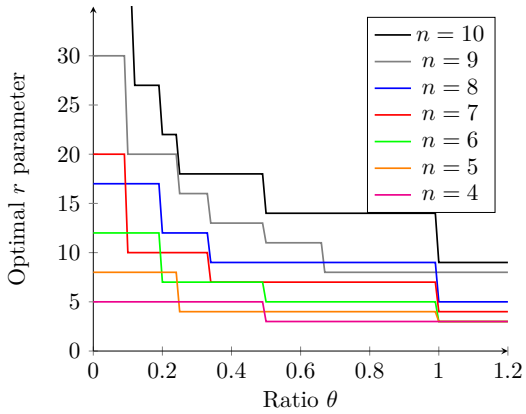


Figure 3.3: Optimal r parameter w.r.t. the CPRR-ISW cost ratio θ for $\lambda \in [\![4, 8]\!]$.

r for $\lambda = 6$, $\lambda = 8$, and $\lambda = 10$. These figures clearly illustrate the asymptotic gain that can be obtained by using optimized parameters.

For $\lambda \in \{6, 8, 10\}$, we checked whether we could find satisfying CPRR-based chains of cyclotomic classes, for the obtained optimal parameters. For $\lambda = 6$, the optimal parameters are $(r, t) = (5, 3)$ (giving 3 CPRR plus 2 ISW) which are actually the original CRV parameters. We could find a satisfying chain for these parameters. For $\lambda = 8$, the optimal parameters are $(r, t) = (9, 4)$ (giving 7 CPRR plus 3 ISW). For these parameters we could not find any satisfying chain. We therefore used the second best set of parameters that is $(r, t) = (8, 5)$ (giving 6 CPRR plus 4 ISW) for which we could find a satisfying chain. For $\lambda = 10$, the optimal parameters are $(r, t) = (14, 8)$ (giving 12 CPRR plus 7 ISW). For these parameters we could neither find any satisfying chain. So once again, we used the second best set of parameters, that is $(r, t) = (13, 9)$ (giving 11 CPRR plus 8 ISW) and for which we could find a satisfying chain. All the obtained satisfying CPRR-based chains of cyclotomic classes are given in Table **??**.

Table 3.10: Cyclotomic classes with CPRR.

λ	r	t	L	exp. $2^w + 1$
Original CRV parameters				
4	3	2	$C_0 \cup C_1 \cup C_3$	3
5	4	3	$C_0 \cup C_1 \cup C_3 \cup C_{15}$	3,5
6	5	3	$C_0 \cup C_1 \cup C_5 \cup C_{11} \cup C_{31}$	5
7	6	4	$C_0 \cup C_1 \cup C_5 \cup C_{19} \cup C_{47} \cup C_{63}$	5
8	7	6	$C_0 \cup C_1 \cup C_5 \cup C_{25} \cup C_{95} \cup C_{55} \cup C_3$	3,5
9	9	8	$C_0 \cup C_1 \cup C_9 \cup C_{17} \cup C_{25} \cup C_{51} \cup C_{85} \cup C_{103} \cup C_{127}$	3,5,9,17
10	11	11	$C_0 \cup C_1 \cup C_3 \cup C_9 \cup C_{17} \cup C_{45} \cup C_{69} \cup C_{85} \cup C_{207} \cup C_{219} \cup C_{447}$	3,5,9,17
Optimized CRV parameters				
8	8	5	$C_0 \cup C_1 \cup C_5 \cup C_{25} \cup C_{95} \cup C_{55} \cup C_3 \cup C_9$	3,5
10	13	9	$C_0 \cup C_1 \cup C_3 \cup C_{15} \cup C_{17} \cup C_{27} \cup C_{51} \cup C_{57} \cup C_{85} \cup C_{123} \cup C_{159} \cup C_{183} \cup C_{205}$	3,5,9,17

Table **??** compares the performances of the original CRV method and the improved versions for our implementation of ISW (half-table and exp-log variants for the field multiplication.

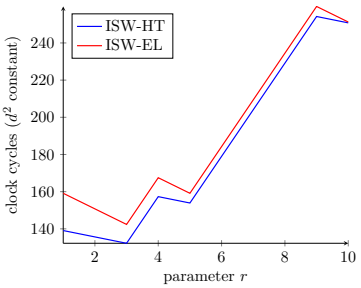


Figure 3.4: CRV d^2 const. ($\lambda = 6$).

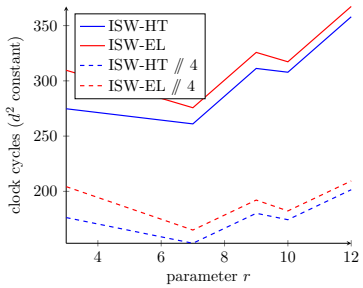


Figure 3.5: CRV d^2 const. ($\lambda = 8$).

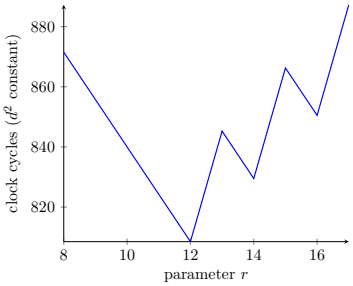


Figure 3.6: CRV d^2 const. ($\lambda = 10$).

See Chapter **??** for more details) and CPRR.[1] For the improved methods, we give the ratio of asymptotic performances with respect to the original version. This ratio ranks between 79% and 94% for the improved version and between 75% and 93% for the improved version with optimized parameters.

Table 3.11: Performances of CRV original version and improved version (with and without optimized parameters).

	Original CRV ([**CHES:CorRoyViv14**])			CRV with CPRR ([**C:CPRR15**])				Optimized CRV with CPPR			
	# ISW	# CPRR	clock cycles	# ISW	# CPRR	clock cycles	ratio	# ISW	# CPRR	clock cycles	ratio
$\lambda = 6$ (HT)	5	0	$142.5\,d^2 + O(d)$	2	3	$132\,d^2 + O(d)$	93%	2	3	$132\,d^2 + O(d)$	93%
$\lambda = 6$ (EL)	5	0	$167.5\,d^2 + O(d)$	2	3	$142\,d^2 + O(d)$	85%	2	3	$142\,d^2 + O(d)$	85%
$\lambda = 8$ (HT)	10	0	$285\,d^2 + O(d)$	5	5	$267.5\,d^2 + O(d)$	94%	4	6	$264\,d^2 + O(d)$	93%
$\lambda = 8$ (EL)	10	0	$335\,d^2 + O(d)$	5	5	$292.5\,d^2 + O(d)$	87%	4	6	$284\,d^2 + O(d)$	85%
$\lambda = 10$ (EL)	19	0	$997.5\,d^2 + O(d)$	10	9	$858\,d^2 + O(d)$	86%	8	11	$827\,d^2 + O(d)$	83%
$\lambda = 8$ (HT) $/\!/4$	10	0	$775\,d^2 + O(d)$	5	5	$657.5\,d^2 + O(d)$	85%	4	6	$634\,d^2 + O(d)$	82%
$\lambda = 8$ (EL) $/\!/4$	10	0	$935\,d^2 + O(d)$	5	5	$737.5\,d^2 + O(d)$	79%	4	6	$698\,d^2 + O(d)$	75%

[1] We only count the calls to ISW and CPRR since other operations are similar in the three variants and have linear complexity in d.

Chapter 4

Composition

4.1 Introduction

The second challenge we want to address in this thesis is the one of the composition of masked operations. In fact, once a sound representation of the s-box function is found, one need to carefully study how to implement it. As shown in Chapter **??**, the independence property, *i.e.* the probing security, of the masking scheme can be compromised with misuse multiplications. Hence, one must carefully place some refresh gadgets in the implementation to produce fresh encodings of the sensitive variables. However, the use of such gadgets comes at a price in randomness consumption and efficiency.

In the past couple of years, several formal tools have been developed to evaluate the probing security of implementations at a given masking order. Among the most efficient ones, Barthe et al. developed `maskVerif` [**EC:BBDFGS15**] and Coron developed `CheckMasks` [**EPRINT:Coron17b**]. Both tools take as input a shared circuit and return a formal security proof when no attack is found. However, this evaluation is not tight and false negatives may occur and hence imply the addition of unnecessary refresh gadgets. Moreover, while such tools are very convenient to evaluate the security of concrete implementations, they suffer from an important limitation which is their exponential complexity in the size of the circuit and consequently in the masking order. As a result, these tools are impractical beyond a small number of shares (typically $d = 5$). In a recent work, Bloem et al. [**EC:BGIKMW18**] further developed a new tool to verify the security of masked implementations subject to *glitches*, which is an important step towards provable and practical security of hardware implementations. However this tool still suffers from the same efficiency limitations as the previous ones.

The method of Barthe et al. [**CCS:BBDFGS16**] allows one to safely compose t-NI and t-SNI gadgets and get probing security at any order. Nevertheless, it is not tight and makes use of more refresh gadgets than required. In many contexts, randomness generation is expensive and might be the bottleneck for masked implementations. For instance, Journault and Standaert describe an AES encryption shared at the order $d = 32$ for which up to 92% of the running time is spent on randomness generation [**CHES:JouSta17**]. In such a context, it is fundamental to figure out whether the number of t-SNI refresh gadgets inserted by Barthe et al.'s tool `maskComp` is actually minimal to achieve t-probing security. In this Chapter, we find out that it is not and we provide a new method which *exactly* identifies the concrete probing attacks in a Boolean shared circuit.

Let us take a simple example. We consider the small randomized circuit referred to as Circuit 1 and illustrated in Figure **??** with $[\oplus]$ a t-NI sharewise addition, $[\otimes]$ a t-SNI multiplication, and two Boolean sharings $[x_1]$ and $[x_2]$. Applying Barthe et al's tool `maskComp` on this circuit automatically inserts a t-SNI refresh gadget in the cycle formed by gates $[x_1]$, $[\oplus]$, and $[\otimes]$ as represented in Figure **??**. However, it can be verified that for any masking order t, the initial circuit is t-probing secure without any additional refresh gadget. Therefore, in the following, this Chapter aims to refine the state-of-the-art method [**CCS:BBDFGS16**] to only insert refresh gadgets when absolutely mandatory for t-probing security.

More specifically, the contribution of this chapter can be summarized as follows:

(1) We introduce formal definitions of the probing, non-interfering, and strong-non-interfering security notions for shared circuits based on concrete security games. Although these definitions are no more than a reformulation of existing security notions, we believe that they provide a simple and precise framework to reason about probing security.

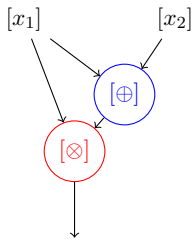


Figure 4.1: Graph representation of Circuit 1.

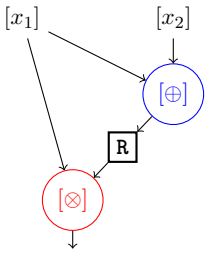


Figure 4.2: Graph representation of Circuit 1 after `maskComp`.

(2) From the introduced game-based definitions, we provide a reduction of the probing security of a given *standard shared circuit* –*i.e.* a shared circuit composed of ISW multiplication gadgets, sharewise addition gadgets and SNI refresh gadgets– to the probing security of a simpler circuit of multiplicative depth 1 and for which the adversary is restricted to probe the multiplication inputs (which are linear combinations of the circuit inputs).

(3) We give an algebraic characterization of the final security game, which allows us to express the probing security of any standard shared circuit in terms of linear algebra.

(4) We show how to solve the latter problem with a new exact and proven method. Our method takes the description of any standard shared circuit and either produces a probing-security proof (valid at any order) or exhibits a probing attack (*i.e.* a set of $t < d$ probes that reveal information on the circuit d-shared input for some d). We provide a concrete tool implementing our method in Sage.

(5) We apply our tool to the efficient implementation of the AES s-box developed by Goudarzi and Rivain in [**EC:GouRiv17**]. Based on the previous state of the art, this s-box was implemented using one SNI refresh gadget per multiplication gadget (to refresh one of the operands), hence making a total of 32 refresh gadgets (which was later on confirmed by the `maskComp` tool). Our new method formally demonstrates that the same d-shared implementation is actually t-probing secure with *no* refresh gadget for any $d = t + 1$. We provide implementation results and a performance analysis: this new implementation achieves an asymptotic gain up to 43%. The code can be found on GitHub [**github**].

(6) We extend our results to larger circuits by establishing new compositional properties on t-probing secure gadgets. In particular, these new composition properties perfectly apply to the case of SPN-based block ciphers. We also show that they apply to a wide range of Boolean circuits with common gadgets and input sets.

4.2 Approach

In Section **??**, useful notions are introduced, security definitions for composition are formalized through concrete security games, and some useful security results are provided. Section **??**

provides our security reduction for standard shared circuits. Section **??** then details our new method to exactly determine the probing security of a standard shared circuit. It also gives an upper bound on the number of required refresh gadgets together with an exhaustive method to make a standard shared circuit achieve tight probing security. In Section **??**, our new method is extended to apply to larger circuits, and in particular to SPN-based block ciphers, with new compositional properties. Finally, Section **??** describes the new tool we implemented to experiment our method on concrete circuits.

4.3 Formal Security Notions

4.3.1 Notations

In this chapter, we denote by $\mathbb{F}_2$ the finite field with two elements and by $[\![i, j]\!]$ the integer interval $\mathbb{Z} \cap [i, j]$ for any two integers i and j. For a finite set $\mathcal{X}$, we denote by $|\mathcal{X}|$ the cardinal of $\mathcal{X}$ and by $x \leftarrow \mathcal{X}$ the action of picking x from $\mathcal{X}$ independently and uniformly at random. For some (probabilistic) algorithm $\mathcal{A}$, we further denote $x \leftarrow \mathcal{A}(in)$ the action of running algorithm $\mathcal{A}$ on some inputs in (with fresh uniform random tape) and setting x to the value returned by $\mathcal{A}$.

4.3.2 Basic Notions

A *Boolean circuit* is a directed acyclic graph whose vertices are input gates, output gates, constant gates of fan-in 0 that output constant values, and operation gates of fan-in at most 2 and fan-out at most 1 and whose edges are wires. In this chapter we consider Boolean circuits with two types of operation gates: addition gates (computing an addition on $\mathbb{F}_2$) and multiplication gates (computing a multiplication on $\mathbb{F}_2$). A *randomized circuit* is a Boolean circuit augmented with random-bit gates of fan-in 0 that outputs a uniformly random bit.

A *d-Boolean sharing* of $x \in \mathbb{F}_2$ is a random tuple $(x_1, x_2, \ldots, x_d) \in \mathbb{F}_2^d$ satisfying $x = \sum_{i=1}^{d} x_i$. The sharing is said to be *uniform* if, for a given x, it is uniformly distributed over the subspace of tuples satisfying $x = \sum_{i=1}^{d} x_i$. A uniform sharing of x is such that any m-tuple of its *shares* x_i is uniformly distributed over $\mathbb{F}_2^m$ for any $m \leq d$. In the following, a d-Boolean sharing of a given variable x is denoted by $[x]$ when the sharing order d is clear from the context. We further denote by Enc a probabilistic *encoding* algorithm that maps $x \in \mathbb{F}_2$ to a fresh uniform sharing $[x]$.

A *d-shared circuit* C is a randomized circuit working on d-shared variables. More specifically, a d-shared circuit takes a set of n input sharings $[x_1]$, ..., $[x_n]$ and computes a set of m output sharings $[y_1]$, ..., $[y_m]$ such that $(y_1, \ldots, y_m) = f(x_1, \ldots, x_n)$ for some deterministic function f. A *probe* on C refers to a wire index (for some given indexing of C's wires). An *evaluation* of C on input $[x_1]$, ..., $[x_n]$ under a set of probes $\mathcal{P}$ refers to the distribution of the tuple of wires pointed by the probes in $\mathcal{P}$ when the circuit is evaluated on $[x_1]$, ..., $[x_n]$, which is denoted by $C([x_1], \ldots, [x_n])_{\mathcal{P}}$.

We consider a special kind of shared circuits which are composed of *gadgets*. In the chapter, we specifically consider three types of gadgets, namely ISW-multiplication gadgets ($[\otimes]$), ISW-refresh gadgets ($[\mathtt{R}]$) and sharewise addition gadgets ($[\oplus]$).

$\underline{\mathsf{ExpReal}(\mathcal{A}, C)}$:
1: $(\mathcal{P}, x_1, \ldots, x_n) \leftarrow \mathcal{A}()$
2: $[x_1] \leftarrow \mathsf{Enc}(x_1), \ldots, [x_n] \leftarrow \mathsf{Enc}(x_n)$
3: $(v_1, \ldots, v_t) \leftarrow C([x_1], \ldots, [x_n])_{\mathcal{P}}$
4: Return $(v_1, \ldots, v_t)$

$\underline{\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)}$:
1: $(\mathcal{P}, x_1, \ldots, x_n) \leftarrow \mathcal{A}()$
2: $(v_1, \ldots, v_t) \leftarrow \mathcal{S}(\mathcal{P})$
3: Return $(v_1, \ldots, v_t)$

Figure 4.3: t-probing security game.

Definition 4.3.1. *A* standard shared circuit *is a shared circuit exclusively composed of ISW-multiplication gadgets, ISW-refresh gadgets and sharewise addition gadgets as described above.*

4.3.3 Game-Based Security Definitions

In the following, we recall the *probing*, *non-interfering* and *strong non-interfering* security notions introduced in [**C:IshSahWag03**; **CCS:BBDFGS16**] and we formalize them through concrete security games. Each of these games is defined for a given n-input d-shared circuit C and it opposes an *adversary* $\mathcal{A}$, which is a deterministic algorithm outputting a set of (plain) inputs x_1, ..., x_n and a set of probes $\mathcal{P}$, to a simulator $\mathcal{S}$, which aims at simulating the distribution $C([x_1], \ldots, [x_n])_{\mathcal{P}}$.

4.3.3.1 Probing Security.

We first recall the definition from [**C:IshSahWag03**]. Our game-based definition is then given with a proposition to state the equivalence of both notions.

Definition 4.3.2 (from [**C:IshSahWag03**])**.** *A circuit is t-probing secure if and only if any set of at most t intermediate variables is independent from the secret.*

Probing Security Game. The t-probing security game is built based on two experiments as described in Figure **??**. In both experiments, an adversary $\mathcal{A}$ outputs a set of probes $\mathcal{P}$ (indices of circuit's wires) such that $|\mathcal{P}| = t$ and n input values $x_1, \ldots, x_n \in \mathbb{F}_2$.

In the first (real) experiment, referred to as $\mathsf{ExpReal}$, the chosen input values $x_1, \ldots, x_n$ are mapped into n sharings $[x_1], \ldots, [x_n]$ with encoding algorithm Enc. The resulting encodings are given as inputs to the shared circuit C. The real experiment then outputs a random evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}}$ of the chosen gates through a t-uple $(v_1, \ldots, v_t)$.

In the second experiment, referred to as ExpSim, the probing simulator $\mathcal{S}$ takes the (adversary chosen) set of probes $\mathcal{P}$ and outputs a simulation of the evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}}$, which is returned by the simulation experiment. The simulator wins the game if and only if the two experiments return identical distributions.

Proposition 4.3.3. *A shared circuit C is t-probing secure if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-probing security game defined in Figure* **??**, i.e. *the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ output identical distributions.*

Proof. From right to left, if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-probing security game defined in Figure **??**, then any set of probes is independent

from the secret as $\mathcal{S}$ has no knowledge of the secret inputs. Thus C is trivially t-probing secure by Definition **??**. From left to right, if the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ do not output identical distributions, then there exists a set of at most t intermediate variables which cannot be perfectly simulated without the knowledge of the input secrets. As a consequence, the circuit is not t-probing secure by Definition **??**. □

A shared circuit C which is t-probing secure is referred to as a *t-private circuit*. It is not hard to see that a d-shared circuit can only achieve t-probing security for $d > t$. When a d-shared circuit achieves t-probing security with $d = t + 1$, we call it a *tight private circuit*.

4.3.3.2 Non-Interfering Security.

The non-interfering security notion is a little bit stronger. Compared to the probing security notion, it additionally benefits from making the security evaluation of composition of circuits easier. We recall its original definition from [**CCS:BBDFGS16**] before we give an equivalent formal game-based definition.

Definition 4.3.4 (from [**CCS:BBDFGS16**])**.** *A circuit is t-non-interfering (t-NI) if and only if any set of at most t intermediate variables can be perfectly simulated from at most t shares of each input.*

Non-Interfering Security Game. The t-non-interfering (t-NI) security game is built based on two experiments as described in Figure **??**. In both experiments, an adversary $\mathcal{A}$ outputs a set of probes $\mathcal{P}$ (indices of circuit's wires) such that $|\mathcal{P}| = t$ and n input sharings $[x_1], \ldots, [x_n] \in \mathbb{F}_2^d$.

The first (real) experiment, referred to as $\mathsf{ExpReal}$, simply returns an evaluation of C on input sharings $[x_1]$, …, $[x_n]$ under the set of probes $\mathcal{P}$.

The second experiment, referred to as ExpSim, is defined for a two-round simulator $\mathcal{S} = (\mathcal{S}^1, \mathcal{S}^2)$. In the first round, the simulator $\mathcal{S}^1$ takes the (adversary chosen) set of probes $\mathcal{P}$ and outputs n sets of indices $\mathcal{I}_1, \ldots, \mathcal{I}_n \subseteq \{1, \ldots, d\}$, such that $|\mathcal{I}_1| = \cdots = |\mathcal{I}_n| = t$. In the second round, in addition to the set of probes $\mathcal{P}$, the simulator $\mathcal{S}^2$ receives the (adversary chosen) input sharings restricted to the shares indexed by the sets $\mathcal{I}_1, \ldots, \mathcal{I}_n$, denoted $[x_1]_{\mathcal{I}_1}$, …, $[x_n]_{\mathcal{I}_n}$, and outputs a simulation of $C([x_1], \ldots, [x_n])_{\mathcal{P}}$, which is returned by the simulation experiment. The simulator wins the game if and only if the two experiments return identical distributions.

Proposition 4.3.5. *A shared circuit C is* t-non-interfering secure *if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-non-interfering security game defined in Figure* **??**, i.e. *the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ output identical distributions.*

Proof. From right to left, if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-non interfering security game defined in Figure **??**, then any set of probes can be perfectly simulated from sets of at most t shares of each input. Thus C is trivially t-non-interfering from Definition **??**. From left to right, if the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ do not output identical distributions, then there exists a set of at most t intermediate variables which cannot be perfectly simulated from sets $\mathcal{I}_j$ of input shares whose cardinals are less than t. As a consequence, the circuit is not t-non interfering secure from Definition **??**. □

$\underline{\mathsf{ExpReal}(\mathcal{A}, C)}$:
1: $(\mathcal{P}, [x_1], \ldots, [x_n]) \leftarrow \mathcal{A}()$
2: $(v_1, \ldots, v_t) \leftarrow C([x_1], \ldots, [x_n])_{\mathcal{P}}$
3: Return $(v_1, \ldots, v_t)$

$\underline{\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)}$: *
1: $(\mathcal{P}, [x_1], \ldots, [x_n]) \leftarrow \mathcal{A}()$
2: $\mathcal{I}_1, \ldots, \mathcal{I}_n \leftarrow \mathcal{S}^1(\mathcal{P})$
3: $(v_1, \ldots, v_t) \leftarrow \mathcal{S}^2(\mathcal{P}, [x_1]_{\mathcal{I}_1}, \ldots, [x_n]_{\mathcal{I}_n})$
4: Return $(v_1, \ldots, v_t)$

* For t-NI: $|\mathcal{I}_1| = \cdots = |\mathcal{I}_n| = t$.
For t-SNI: $|\mathcal{I}_1| = \cdots = |\mathcal{I}_n| = |\mathcal{P}_{int}| \leq t$.

Figure 4.4: t-(S)NI security game.

4.3.3.3 Strong Non-Interfering Security.

The strong non-interfering security is a stronger notion than non-interfering security as it additionally guarantees the independence between input and output sharings. The latter property is very convenient to securely compose gadgets with related inputs.

Definition 4.3.6 (Strong non-interfering security from [**CCS:BBDFGS16**])**.** *A circuit is t-strong non-interfering (t-SNI) if and only if any set of at most t intermediate variables whose t_1 on the internal variables (*i.e. *intermediate variables except the output's ones) and t_2 on output variables can be perfectly simulated from at most t_1 shares of each input.*

Strong Non-Interfering Security Game. The t-strong-non-interfering (t-SNI) security game is similar to the t-NI security game depicted in Figure **??**. The only difference relies in the fact that the first-round simulator $\mathcal{S}^1$ outputs n sets of indices $\mathcal{I}_1, \ldots, \mathcal{I}_n \subseteq \{1, \ldots, d\}$, such that $|\mathcal{I}_1| = \cdots = |\mathcal{I}_n| = |\mathcal{P}_{int}| \leq t$ where $\mathcal{P}_{int} \subseteq \mathcal{P}$ refers to the probes on internal wires, *i.e.* the probes in $\mathcal{P}$ which do not point to outputs of C.

Proposition 4.3.7. *A shared circuit C is* t-strong-non-interfering secure *if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-SNI security game defined in Figure* **??**, i.e. *the random experiments* $\mathsf{ExpReal}(\mathcal{A}, C)$ *and* $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ *output identical distributions.*

Proof. From right to left, if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the t-non interfering security game defined in Figure **??**, then any set of probes can be perfectly simulated from sets of at most $|\mathcal{P}_{int}| = t_1$ shares of each input. Thus C is trivially t-strong non-interfering from Definition **??**. From left to right, if the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$ do not output identical distributions, then there exists a set of at most t intermediate variables which cannot be perfectly simulated from sets $\mathcal{I}_j$ of input shares whose cardinals are less than t_1. As a consequence, the circuit is not t-strong non interfering secure from Definition **??**. □

4.3.4 Useful Security Results

This section states a few useful security results. From the above definitions, it is not hard to see that for any shared circuit C we have the following implications:

$$C \text{ is } t\text{-SNI} \;\Rightarrow\; C \text{ is } t\text{-NI} \;\Rightarrow\; C \text{ is } t\text{-probing secure}$$

while the converses are not true. While the ISW-multiplication (and refresh) gadget defined above was originally shown to achieve probing security, it actually achieves the more general notion of strong non-interfering security as formally stated in the following theorem:

Theorem 4.3.8 ([**CCS:BBDFGS16**])**.** *For any integers d and t such that $t < d$, the d-shared ISW-multiplication gadget $[\otimes]$ and the d-shared ISW-refresh gadget $[\mathtt{R}]$ are both t-SNI.*

The next lemma states a simple implication of the t-SNI notion:

Lemma 4.3.9. *Let C be a n-input $(t+1)$-shared t-SNI circuit. Then for every $(x_1, \ldots, x_n) \in \mathbb{F}_2^n$, an evaluation of C taking n uniform and independent $(t+1)$-Boolean sharings $[x_1], \ldots, [x_n]$ as input produces a sharing $[y]$ (of some value $y \in \mathbb{F}_2$ which is a function of $x_1, \ldots, x_n$) which is uniform and mutually independent of $[x_1], \ldots, [x_n]$.*

Proof of Lemma **??**. Let $\mathcal{I} \subseteq \{0, 1, \ldots, t\}$ such that $|\mathcal{I}| = t$. From Definition **??**, consider an adversary $\mathcal{A}$ which outputs a set of probes $\mathcal{P}$ matching the output shares $[y]_\mathcal{I}$. The t-SNI property implies that there exists an algorithm $\mathcal{S}$ performing a perfect simulation of $[y]_\mathcal{I}$ independently of $[x_1], \ldots, [x_n]$, that is

$$\mathrm{P}([x_1], \ldots, [x_n], [y]_\mathcal{I}) = \mathrm{P}([x_1], \ldots, [x_n]) \cdot \mathrm{P}([y]_\mathcal{I}) \ . \tag{4.1}$$

Moreover, since for a given $y = f(x_1, \ldots, x_n)$, the sharing $[y]$ is perfectly defined by $[y]_\mathcal{I}$, the above rewrites

$$\mathrm{P}([x_1], \ldots, [x_n], [y]) = \mathrm{P}([x_1], \ldots, [x_n]) \cdot \mathrm{P}([y]) \ , \tag{4.2}$$

which implies the mutual independence between $[y]$ and $[x_1], \ldots, [x_n]$.

Let us now show that $[y]$ is a uniform sharing *i.e.* the t-tuple $[y]_\mathcal{I}$ is uniformly distributed over $\mathbb{F}_2^t$, for any $\mathcal{I} \subseteq \{0, 1, \ldots, t\}$ such that $|\mathcal{I}| = t$. We proceed by contradiction: we assume that $[y]$ is not a uniform sharing and then show that C cannot be t-SNI. If $[y]$ is not a uniform sharing, then the t-tuple $[y]_\mathcal{I}$ is not uniformly distributed over $\mathbb{F}_2^t$. This implies that there exists a set $\mathcal{J} \subseteq \mathcal{I}$ with $\mathcal{J} \neq \emptyset$ such that the sum $\sum_{i \in \mathcal{J}} y_i$ is not uniformly distributed over $\mathbb{F}_2$. Then for any $y \in \mathbb{F}_2$, we have

$$\sum_{i \in [\![0,t]\!] \setminus \mathcal{J}} y_i + \sum_{i \in \mathcal{J}} y_i = y \ , \tag{4.3}$$

which implies that both $[y]_\mathcal{J}$ and $[y]_{[\![0,t]\!] \setminus \mathcal{J}}$ are not uniformly distributed and statistically dependent on y. This implies that the tuple $[y]_\mathcal{J}$ cannot be perfectly simulated independently of y which contradicts the t-SNI property. □

4.4 A Security Reduction

This section provides a reduction for the t-probing security of a standard $(t+1)$-shared circuit C as defined in Section **??**. Through a sequence of games we obtain a broad simplification of the problem of verifying whether C is probing secure or not. At each step of our reduction, a new game is introduced which is shown to be equivalent to the previous one, implying that for any adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins the new game if and only if the circuit C is t-probing secure. We get a final game (see Game 3 hereafter) in which only the inputs of the multiplication gadgets can be probed by the adversary and the circuit is

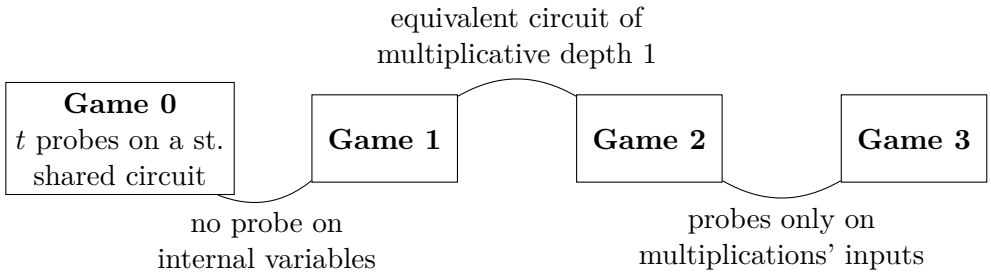


Figure 4.5: Overview of the sequence of games.

flattened into an (equivalent) circuit of multiplicative depth one. This allows us to express the probing security property as a linear algebra problem, which can then be solved efficiently as we show in Section **??**.

In a nutshell, our Game 0 exactly fits the game-based definition of t-probing security given in the previous section. Then, with Game 1, we prove that verifying the t-probing security of a standard shared circuit C is exactly equivalent to verifying the t-probing security of the same circuit C where the attacker $\mathcal{A}$ is restricted to probe inputs of refresh gadgets, pairs of inputs of multiplication gadgets, and inputs and outputs of sharewise additions (i.e., no internal gadgets variables). Game 2 then shows that verifying the t-probing security of a standard shared circuit C with a restricted attacker $\mathcal{A}$ is equivalent to verifying the t-probing security of a functionally equivalent circuit C' of multiplicative depth one where all the outputs of multiplication and refresh gadgets in C are replaced by fresh input sharings of the same values in the rest of the circuit. Finally, with Game 3, we show that we can even restrict the adversary to probe only pairs (x_i, y_j) where x_i (resp. y_j) is the i^{th} share of x (resp. the j^{th} share of y) and such that x and y are operands of the same multiplication in C. These three games are deeply detailed hereafter and proofs of their consecutive equivalence are provided at each step. An overview is displayed on Figure **??**.

4.4.1 Game 1.

In a nutshell, our first game transition relies on the fact that each probe in a t-SNI gadget can be replaced by 1 or 2 probes on the input sharing(s) of the gadget. In particular, one probe on a refresh gadget is equivalent to revealing one input share, one probe on a multiplication gadget is equivalent to revealing two input shares (one share per input sharing). Formally, in the random experiments $\mathsf{ExpReal}(\mathcal{A}, C)$ and $\mathsf{ExpSim}(\mathcal{A}, \mathcal{S}, C)$, the set of probes $\mathcal{P}$ returned by $\mathcal{A}$, noted $\mathcal{P}'$ in the following, has a different form explicitly defined below.

Let us associate an index g to each gadget in the standard shared circuit and denote by $\mathcal{G}$ the set of gadget indices. Let us further denote by $\mathcal{G}_r$, $\mathcal{G}_m$ and $\mathcal{G}_a$ the index sets of refresh gadgets, multiplication gadgets and addition gadgets, such that $\mathcal{G} = \mathcal{G}_r \cup \mathcal{G}_m \cup \mathcal{G}_a$. Then we can denote by $\mathcal{I}_g$ and $\mathcal{J}_g$ the indices of circuit wires which are the shares of the (right and left) input operands of gadget $g \in \mathcal{G}$ (where $\mathcal{J}_g = \emptyset$ if gadget g is a refresh). Similarly, we denote by $\mathcal{O}_g$ the indices of circuit wires which represent the output of gadget $g \in \mathcal{G}$. From these notations, an admissible set of probes $\mathcal{P}'$ from the adversary in the new game is of the

$\mathsf{ExpReal}_1(\mathcal{A}, C)$:
1: $(\mathcal{P}', x_1, \ldots, x_n) \leftarrow \mathcal{A}()$
2: $[x_1] \leftarrow \mathsf{Enc}(x_1), \ldots, [x_n] \leftarrow \mathsf{Enc}(x_n)$
3: $(v_1, \ldots, v_q) \leftarrow C([x_1], \ldots, [x_n])_{\mathcal{P}'}$
4: Return $(v_1, \ldots, v_q)$

$\mathsf{ExpSim}_1(\mathcal{A}, \mathcal{S}, C)$:
1: $(\mathcal{P}', x_1, \ldots, x_n) \leftarrow \mathcal{A}()$
2: $(v_1, \ldots, v_q) \leftarrow \mathcal{S}(\mathcal{P}')$
3: Return $(v_1, \ldots, v_q)$

Figure 4.6: Game 1.

form

$$\mathcal{P}' = \mathcal{P}'_r \cup \mathcal{P}'_m \cup \mathcal{P}'_a$$

where

$$\begin{aligned} \mathcal{P}'_r &\subseteq \bigcup_{g \in \mathcal{G}_r} \mathcal{I}_g \\ \mathcal{P}'_m &\subseteq \bigcup_{g \in \mathcal{G}_m} \mathcal{I}_g \times \mathcal{J}_g \\ \mathcal{P}'_a &\subseteq \bigcup_{g \in \mathcal{G}_a} \mathcal{I}_g \bigcup_{g \in \mathcal{G}_a} \mathcal{J}_g \bigcup_{g \in \mathcal{G}_a} \mathcal{O}_g \end{aligned}$$

and $|\mathcal{P}'| = t$. That is, each of the t elements of $\mathcal{P}'$ either is a pair of index in $\mathcal{I}_g \times \mathcal{J}_g$ for a multiplication gadget g, or a single index in $\mathcal{I}_g$ for a refresh gadget g, or a single index in $\mathcal{I}_g \cup \mathcal{J}_g \cup \mathcal{O}_g$ for an addition gadget. Note that in the latter case, the index can correspond to any wire in the addition gadget (which is simply composed of $t + 1$ addition gates).

Let t_m be the number of probes on multiplication gadgets, *i.e.* $t_m = |\mathcal{P}'_m|$, and t_{ar} the number of probes on refresh or addition gadgets, *i.e.* $t_{ar} = |\mathcal{P}'_a \cup \mathcal{P}'_r|$, so that $t_m + t_{ar} = t$. The evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$ then returns a q-tuple for $q = 2t_m + t_{ar}$, which is composed of the values taken by the wires of index $i \in \mathcal{P}'_a \cup \mathcal{P}'_r$, and the values taken by the wires of index i and j with $(i, j) \in \mathcal{P}'_m$. The new experiments $\mathsf{ExpReal}_1(\mathcal{A}, C)$ and $\mathsf{ExpSim}_1(\mathcal{A}, \mathcal{S}, C)$, carefully written in Figure **??**, each output a q-tuple and, as before, the simulator wins Game 1 if and only if the associated distributions are identical.

Proposition 4.4.1. *A standard shared circuit C is t-probing secure if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins Game 1 defined above,* i.e. *the random experiments $\mathsf{ExpReal}_1(\mathcal{A}, C)$ and $\mathsf{ExpSim}_1(\mathcal{A}, \mathcal{S}, C)$ output identical distributions.*

Proof. We first show:

$$\forall \mathcal{A}_1, \exists \mathcal{S}_1 \text{ wins Game 1} \Rightarrow \forall \mathcal{A}_0, \exists \mathcal{S}_0 \text{ wins the } t\text{-probing security game} \tag{4.4}$$

Let us consider an adversary $\mathcal{A}_0$ that outputs some values $x_1, \ldots, x_n \in \mathbb{F}_2$ and a set of probes $\mathcal{P}$. By definition, $\mathcal{P}$ can be partitioned into three subsets of probes, *i.e.* $\mathcal{P} = \mathcal{P}_a \cup \mathcal{P}_r \cup \mathcal{P}_m$, where $\mathcal{P}_a$ represents the probes on addition gadgets, $\mathcal{P}_r$ the probes on refresh gadgets, and $\mathcal{P}_m$ the probes on multiplication gadgets. Let us denote by $\mathcal{P}_r^{(g)} \subseteq \mathcal{P}_r$ (resp. $\mathcal{P}_m^{(g)} \subseteq \mathcal{P}_m$) the set of probes that point to the wires of the refresh gadget of index $g \in \mathcal{G}_r$ (resp. the multiplication gadget of index $g \in \mathcal{G}_m$). The t-SNI property of the refresh and multiplication gadgets (see Definition **??**) implies that:

- For every $g \in \mathcal{G}_r$, there exists a simulator $\mathcal{S}_{\mathsf{SNI}}^{(g)}$ that given the set of probes $\mathcal{P}_r^{(g)}$ (on the internal wires of gadget g), outputs a set of probes $\mathcal{P}_r'^{(g)} \subseteq \mathcal{I}_g$ (on the input shares of gadget g) such that $|\mathcal{P}_r'^{(g)}| = |\mathcal{P}_r^{(g)}|$, and given the input shares pointed by $\mathcal{P}_r'^{(g)}$, outputs a perfect simulation of the internal wires pointed by $\mathcal{P}_r^{(g)}$;
- For every $g \in \mathcal{G}_m$, there exists a simulator $\mathcal{S}_{\mathsf{SNI}}^{(g)}$ that given the set of probes $\mathcal{P}_m^{(g)}$ (on the internal wires of gadget g), outputs a set of pairs of probes $\mathcal{P}_m'^{(g)} \subseteq \mathcal{I}_g \times \mathcal{J}_g$ (on the input shares of each operand of gadget g) such that $|\mathcal{P}_m'^{(g)}| = |\mathcal{P}_m^{(g)}|$, and given the input shares pointed by $\mathcal{P}_m'^{(g)}$, outputs a perfect simulation of the internal wires pointed by $\mathcal{P}_m^{(g)}$;

We define $\mathcal{A}_1$ as the adversary that returns the same values $x_1, \ldots, x_n \in \mathbb{F}_2$ as $\mathcal{A}_0$ and the set of probes $\mathcal{P}' = \mathcal{P}'_a \cup \mathcal{P}'_r \cup \mathcal{P}'_m$ defined from $\mathcal{P}$ as:

$$\mathcal{P}'_r = \bigcup_{g \in \mathcal{G}_r} \mathcal{P}_r'^{(g)} \ , \quad \mathcal{P}'_m = \bigcup_{g \in \mathcal{G}_m} \mathcal{P}_m'^{(g)} \ , \quad \mathcal{P}'_a = \mathcal{P}_a,$$

where $\mathcal{P}_r'^{(g)}$ and $\mathcal{P}_m'^{(g)}$ denote the sets of probes defined by the simulators $\mathcal{S}_{\mathsf{SNI}}^{(g)}$ on input $\mathcal{P}_r^{(g)}$ and $\mathcal{P}_m^{(g)}$ respectively. From the left side of implication Equation (**??**), there exists a simulator $\mathcal{S}_1$ that wins Game 1 for the inputs $(x_1, \ldots, x_n)$ and the built set of probes $\mathcal{P}'$. We define the simulator $\mathcal{S}_0$ as the simulator that computes $\mathcal{P}'$ from $\mathcal{P}$ as explained above and then call $\mathcal{S}_1$ to get a perfect simulation of $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$. Then $\mathcal{S}_0$ applies the simulator $\mathcal{S}_{\mathsf{SNI}}^{(g)}$ to get a perfect simulation of the internal wires pointed by $\mathcal{P}_r^{(g)}$ (resp. $\mathcal{P}_m^{(g)}$) from the input shares pointed by $\mathcal{P}_r'^{(g)}$ (resp. $\mathcal{P}_m'^{(g)}$) which are obtained from the evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$. This way $\mathcal{S}_0$ obtains (and returns) a perfect simulation of $C([x_1], \ldots, [x_n])_{\mathcal{P}}$ and the two experiments $\mathsf{ExpReal}(\mathcal{A}_0, C)$ and $\mathsf{ExpSim}(\mathcal{A}_0, \mathcal{S}_0, C)$ output identical distributions, which demonstrates the implication Equation (**??**).

Let us now show:

$$\forall \mathcal{A}_0, \exists \mathcal{S}_0 \text{ wins the } t\text{-probing security game} \Rightarrow \forall \mathcal{A}_1, \exists \mathcal{S}_1 \text{ wins Game 1} \tag{4.5}$$

By contraposition, we can equivalently show that

$$\exists \mathcal{A}_1, \forall \mathcal{S}_1, \mathcal{S}_1 \text{ fails in Game 1} \Rightarrow \exists \mathcal{A}_0, \forall \mathcal{S}_0, \mathcal{S}_0 \text{ fails in the } t\text{-probing security game} \tag{4.6}$$

Let us thus assume that an adversary $\mathcal{A}_1$ exists which outputs some values $x_1, \ldots x_n$ and a set of probes $\mathcal{P}' = \mathcal{P}'_a \cup \mathcal{P}'_r \cup \mathcal{P}'_m$ such that no algorithm $\mathcal{S}_1$ can output a perfect simulation of $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$. We show that we can then define an adversary $\mathcal{A}_0$ for which no simulator $\mathcal{S}_0$ can win the t-probing security game. The adversary $\mathcal{A}_0$ outputs the same values $x_1, \ldots x_n$ as $\mathcal{A}_1$ and the set of probes $\mathcal{P} = \mathcal{P}_a \cup \mathcal{P}_r \cup \mathcal{P}_m$ such that

$$\mathcal{P}_a = \mathcal{P}'_a \ \text{ and } \ \mathcal{P}_r = \mathcal{P}'_r \tag{4.7}$$

We show in the following how to construct $\mathcal{P}_m$ so that no simulator $\mathcal{S}_0$ can output a perfect simulation of $C([x_1], \ldots, [x_n])_{\mathcal{P}}$.

If $\mathcal{P}'_m = \emptyset$ then we have $\mathcal{P} = \mathcal{P}'$ and the statement directly holds. Let us now consider $\mathcal{P}'_m = \{(i,j)\}$. From the left-side implication of **??**, we get that no simulator $\mathcal{S}_1$ can perform a perfect simulation of

$$(v_1, \ldots, v_q) = C([x_1], \ldots, [x_n])_{\mathcal{P}'} \ , \tag{4.8}$$

where $q = t + 1$. Without loss of generality we assume that v_1 and v_2 are the wires pointed by the indices i and j. We can assume that there exists a simulator $\mathcal{S}_0$ computing a perfect simulation of $(v_3, \ldots, v_q)$, *i.e.* the wires pointed by $\mathcal{P}'_a \cup \mathcal{P}'_r$. (Otherwise we can simply define $\mathcal{A}_0$ as returning the set of probes $\mathcal{P}'_a \cup \mathcal{P}'_r$ and Equation (**??**) directly holds). We deduce that no simulator can achieve a perfect simulation of (v_1, v_2) given $(v_3, \ldots, v_q)$. In a standard shared circuit, the shares of the input of a multiplication gadget are linear combinations of the input shares $[x_1]$, ..., $[x_N]$ of the input of the circuit or the shares in output of refresh or multiplication gadgets. We hence get that v_1 and v_2 can be expressed as

$$v_1 = f_1(x_1, \ldots, x_N) + g_1(v_3, \ldots, v_q) + r_1$$
$$v_2 = f_2(x_1, \ldots, x_N) + g_2(v_3, \ldots, v_q) + r_2$$

for some deterministic function f_1, f_2, g_1, g_2 and where

$$(r_1, r_2) \in \{(0,0), (0,r), (r,0), (r,r)\}$$

for some uniform random r over $\mathbb{F}_2$. (Note that r_1 and r_2 cannot be uniform independent random elements of $\mathbb{F}_2$ otherwise the (v_1, v_2) could be straightforwardly simulated). We then have four cases:

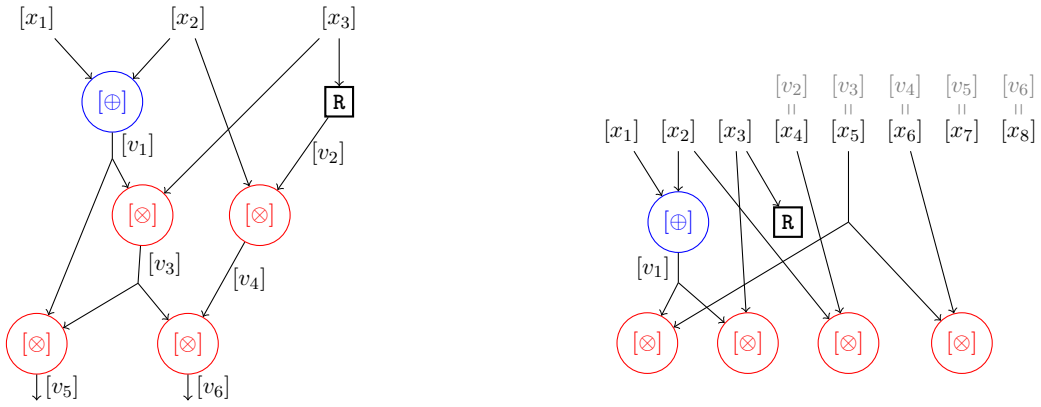
- For $(r_1, r_2) = (0,0)$, we have either f_1 or f_2 non constant (otherwise (v_1, v_2) could be simulated). If f_1 (resp. f_2) is non constant, then v_1 (resp. v_2) cannot be simulated given $(v_3, \ldots, v_q)$ and we define $\mathcal{P}_m = \{i\}$ (resp. $\mathcal{P}_m = \{j\}$).
- For $(r_1, r_2) = (0,r)$, we have f_1 non constant (otherwise (v_1, v_2) could be simulated). Then v_1 cannot be simulated given $(v_3, \ldots, v_q)$ and we define $\mathcal{P}_m = \{i\}$.
- For $(r_1, r_2) = (r,0)$, we have f_2 non constant (otherwise (v_1, v_2) could be simulated). Then v_2 cannot be simulated given $(v_3, \ldots, v_q)$ and we define $\mathcal{P}_m = \{j\}$.
- For $(r_1, r_2) = (r,r)$, we have $f_1 + f_2$ non constant (otherwise (v_1, v_2) could be simulated).[1] Then the product $v_1 \cdot v_2$ satisfies

  $$v_1 \cdot v_2 = \big([f_1 + f_2](x_1, \ldots, x_N) + \delta + r'\big) \cdot r'$$

  where $\delta = [g_1 + g_2](v_3, \ldots, v_q)$ is a constant given $(v_3, \ldots, v_q)$ and where $r' = v_2$ is a uniform random element of $\mathbb{F}_2$. It is not hard to see that the distribution of $v_1 \cdot v_2$ cannot be simulated without knowing $[f_1 + f_2](x_1, \ldots, x_N)$. We then define $\mathcal{P}_m = \{\psi(i,j)\}$ where $\psi(i,j)$ denotes the index of the cross-product $w_i \cdot w_j$ computed in the target ISW-multiplication gadget, with w_i and w_j denoting the wires indexed by i and j.

For the general case where $\mathcal{P}'_m$ contains more than one pair, we can proceed as above to show that no $\mathcal{S}_0$ can simulate $C([x_1], \ldots, [x_n])_{\mathcal{P}^{(1)}}$ where $\mathcal{P}^{(1)}$ is obtained by replacing one pair (i,j) from $\mathcal{P}'$ by a single index i, j or $\psi(i,j)$ as described above. Then we reiterate the same principle to show that no $\mathcal{S}_0$ can simulate $C([x_1], \ldots, [x_n])_{\mathcal{P}^{(2)}}$ where $\mathcal{P}^{(2)}$ is obtained from $\mathcal{P}^{(1)}$ by replacing one more pair (i,j) by a single index. And so on until the set of probes has no more pairs but only t wire indices as in the original probing security game. □

[1] Indeed if $f_1 = f_2$ then (v_1, v_2) can be simulated by $(g_1(\cdots) + r', g_2(\cdots) + r')$ for some uniform random r'.



Circuit C Circuit C'

Figure 4.7: Illustration of the Flatten transformation.

$\underline{\mathsf{ExpReal}_2(\mathcal{A}, C)}$:
1: $C' \leftarrow \mathsf{Flatten}(C)$
2: $(\mathcal{P}', x_1, \ldots, x_N) \leftarrow \mathcal{A}()$
3: $[x_1] \leftarrow \mathsf{Enc}(x_1), \ldots, [x_N] \leftarrow \mathsf{Enc}(x_N)$
4: $(v_1, \ldots, v_q) \leftarrow C'([x_1], \ldots, [x_N])_{\mathcal{P}'}$
5: Return $(v_1, \ldots, v_q)$

$\underline{\mathsf{ExpSim}_2(\mathcal{A}, \mathcal{S}, C)}$:
1: $C' \leftarrow \mathsf{Flatten}(C)$
2: $(\mathcal{P}', x_1, \ldots, x_N) \leftarrow \mathcal{A}()$
3: $(v_1, \ldots, v_q) \leftarrow \mathcal{S}(\mathcal{P}')$
4: Return $(v_1, \ldots, v_q)$

Figure 4.8: Game 2.

4.4.2 Game 2.

Our second game transition consists in replacing the circuit C by a functionally equivalent circuit C' of multiplicative depth one and with an extended input. In a nutshell, each output of a multiplication or a refresh gadget in C is replaced by a fresh new input sharing of the same value in the rest of the circuit. The new circuit hence takes N input sharings $[x_1], \ldots, [x_n], [x_{n+1}], \ldots, [x_N]$, with $N = n + |\mathcal{G}_m| + |\mathcal{G}_r|$. The two circuits are functionally equivalent in the sense that for every input $(x_1, \ldots, x_n)$ there exists an extension $(x_{n+1}, \ldots, x_N)$ such that $C([x_1], \ldots, [x_n])$ and $C'([x_1], \ldots, [x_N])$ have output sharings encoding the same values. This transformation is further referred to as Flatten in the following, and is illustrated on Figure **??**.

The resulting Game 2 is illustrated on Figure **??**. Although the additional inputs x_{n+1}, $\ldots$, x_N are deterministic functions of the original inputs x_1, $\ldots$, x_n, we allow the adversary to select the full extended input x_1, $\ldots$, x_N for the sake of simplicity. This slight adversarial power overhead does not affect the equivalence between the games.

Proposition 4.4.2. *A standard shared circuit C is t-probing secure if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins Game 2 defined above,* i.e. *the random experiments $\mathsf{ExpReal}_2(\mathcal{A}, C)$ and $\mathsf{ExpSim}_2(\mathcal{A}, \mathcal{S}, C)$ output identical distributions.*

Proof. Without loss of generality, we assume that the Flatten transformation does not change

the gadget indexing. We first show:

$$\forall \mathcal{A}_2,\ \exists \mathcal{S}_2 \text{ wins Game 2} \Rightarrow \forall \mathcal{A}_1,\ \exists \mathcal{S}_1 \text{ wins Game 1 } (\Leftrightarrow C \text{ } t\text{-probing secure}) \tag{4.9}$$

For each adversary $\mathcal{A}_1$ returning $(x_1, \ldots, x_n)$ and $\mathcal{P}'$, we define $\mathcal{A}_2$ as the adversary that returns the same choice of probes $\mathcal{P}'$ and the extended input $(x_1, \ldots, x_N)$ such that the n first elements match the choice of $\mathcal{A}_1$ and the $N - n$ matches the decoded outputs of the corresponding multiplication and refresh gadgets. Then, by Lemma **??**, the t-SNI property of multiplication and refresh gadgets implies that each sharing in the output of these gadgets is independent of the input sharings. Since all the probes in $\mathcal{P}'$ on multiplication and refresh gadgets point to input shares only, the output of each such gadget can be replaced by a fresh uniform sharing of the underlying plain value (which deterministically depends on x_1, …, x_n) without modifying the evaluation. We hence get that $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$ in Game 1 and $C'([x_1], \ldots, [x_N])_{\mathcal{P}'}$ in Game 2 output identical distributions. We can then simply define $\mathcal{S}_1$ as the simulator $\mathcal{S}_2$ winning against the defined adversary $\mathcal{A}_2$. We thus get a simulator that outputs the same distribution as ExpReal_1 from which we get Equation (**??**). Let us now show:

$$\forall \mathcal{A}_1,\ \exists \mathcal{S}_1 \text{ wins Game 1 } (\Leftrightarrow C \text{ } t\text{-probing secure}) \Rightarrow \forall \mathcal{A}_2,\ \exists \mathcal{S}_2 \text{ wins Game 2} \tag{4.10}$$

For each adversary $\mathcal{A}_2$ returning $(x_1, \ldots, x_N)$ and $\mathcal{P}'$, we define $\mathcal{A}_1$ as the adversary that returns the same choice of probes $\mathcal{P}'$ and the truncated input with the n first elements of $(x_1, \ldots, x_n)$. For the same reason as above, the evaluations $C([x_1], \ldots, [x_n])_{\mathcal{P}'}$ in Game 1 and $C'([x_1], \ldots, [x_N])_{\mathcal{P}'}$ in Game 2 then output identical distributions and we can simply define $\mathcal{S}_2$ as the simulator $\mathcal{S}_1$ winning against the defined adversary $\mathcal{A}_1$. We thus get a simulator that outputs the same distribution as ExpReal_2 from which we get Equation (**??**). □

Corollary 4.4.3. *A standard shared circuit C is t-probing secure if and only if the standard shared circuit* $\mathsf{Flatten}(C)$ *is t-probing secure.*

4.4.3 Translation to linear algebra.

At this point, the problem of deciding the t-probing security of a Boolean standard shared circuit C has been equivalently reduced to the problem of deciding the t-probing security of a circuit $C' = \mathsf{Flatten}(C)$ when the attacker is restricted to probes on multiplication and refresh gadgets' inputs, and intermediate variables of sharewise additions. In order to further reduce it, we translate the current problem into a linear algebra problem. In the following, we denote by $x_{i,j}$ the jth share of the ith input sharing $[x_i]$ so that

$$[x_i] = (x_{i,0}, x_{i,1}, \ldots, x_{i,t}) \ ,$$

for every $i \in [\![1, N]\!]$. Moreover, we denote by $\overrightarrow{x_j} \in \mathbb{F}_2^N$ the vector composed of the jth share of each input sharing:

$$\overrightarrow{x_j} = (x_{0,j}, x_{1,j}, \ldots, x_{N,j}) \ .$$

As a result of the $\mathsf{Flatten}$ transformation, each probed variable in the q-tuple $(v_1, \ldots, v_q) = C([x_1], \ldots, [x_N])_{\mathcal{P}'}$ is a linear combination of the input sharings $[x_1]$, …, $[x_N]$. Moreover, since the addition gadgets are sharewise, for every $k \in [\![1, q]\!]$, there is a single share index j such that the probed variable v_k only depends of the jth shares of the input sharings, giving:

$$v_k = \overrightarrow{a_k} \cdot \overrightarrow{x_j} \ , \tag{4.11}$$

for some constant coefficient vector $\overrightarrow{a_k} \in \mathbb{F}_2^N$. Without loss of generality, we assume that the tuple of probed variables is ordered w.r.t. the share index j corresponding to each v_k (*i.e.* starting from $j = 0$ up to $j = t$). Specifically, the q-tuple $(v_1, \ldots, v_q)$ is the concatenation of $t + 1$ vectors

$$\overrightarrow{v_0} = M_0 \cdot \overrightarrow{x_0} \ , \quad \overrightarrow{v_1} = M_1 \cdot \overrightarrow{x_1} \ , \quad \ldots \quad \overrightarrow{v_t} = M_t \cdot \overrightarrow{x_t} \ , \tag{4.12}$$

where the matrix M_j is composed of the row coefficient vectors $\overrightarrow{a_k}$ for the probed variable indices k corresponding to the share index j.

Lemma 4.4.4. *For any $(x_1, \ldots, x_N) \in \mathbb{F}_2^N$, the q-tuple of probed variables $(v_1, \ldots, v_q) = C([x_1], \ldots, [x_N])_{\mathcal{P}'}$ can be perfectly simulated if and only if the M_j matrices satisfy*

$$\mathrm{Im}(M_0^T) \cap \mathrm{Im}(M_1^T) \cap \cdots \cap \mathrm{Im}(M_t^T) = \emptyset \ .$$

Moreover, if the M_j matrices are full-rank (which can be assumed without loss of generality), then the above equation implies that $(v_1, \ldots, v_q)$ is uniformly distributed.

Proof. Without loss of generality we can assume that the M_j matrices are full-rank since otherwise the probed variables $v_1, \ldots, v_q$ would be mutually linearly dependent and simulating them would be equivalent to simulating any subset $(v_k)_{k \in \mathcal{K} \subseteq [\![1,q]\!]}$ defining a free basis of $(v_1, \ldots, v_q)$, and which would then induce full-rank matrices M_j.

Throughout this proof, we denote $\overrightarrow{x} = (x_1, \ldots, x_N)$. We first show that a non-null intersection implies a non-uniform distribution of $(v_1, \ldots, v_q)$ which is statistically dependent on $\overrightarrow{x}$. Indeed, a non-null intersection implies that there exist a non-null vector $\overrightarrow{w} \in \mathbb{F}_2^N$ satisfying

$$\overrightarrow{w} = \overrightarrow{u_0} \cdot M_0 = \overrightarrow{u_1} \cdot M_1 = \cdots = \overrightarrow{u_t} \cdot M_t \ . \tag{4.13}$$

for some (constant) vectors $\overrightarrow{u_0}$, $\ldots$, $\overrightarrow{u_t}$. It follows that

$$\sum_{j=0}^{t} \overrightarrow{u_j} \cdot \overrightarrow{v_j} = \sum_{j=0}^{t} \overrightarrow{w} \cdot \overrightarrow{x_j} = \overrightarrow{w} \cdot \overrightarrow{x} \ ,$$

which implies that the distribution of the q-tuple $(v_1, \ldots, v_q) = (\overrightarrow{v_0} \parallel \cdots \parallel \overrightarrow{v_t})$ is non-uniform and dependent on $\overrightarrow{x}$.

We now show that a null intersection implies a uniform distribution (which can then be easily simulated). The uniformity and mutual independence between the sharings $[x_1]$, $\ldots$, $[x_N]$ implies that we can see $\overrightarrow{x_1}$, $\ldots$, $\overrightarrow{x_t}$ as t uniform and independent vectors on $\mathbb{F}_2^N$, and $\overrightarrow{x_0}$ as

$$\overrightarrow{x_0} = \overrightarrow{x} + \overrightarrow{x_1} + \cdots + \overrightarrow{x_t} \ .$$

The joint distribution of $\overrightarrow{v_1}$, $\ldots$, $\overrightarrow{v_t}$ is hence clearly uniform. Then each coordinate of $\overrightarrow{v_0}$ is the result of the inner product $\overrightarrow{r} \cdot \overrightarrow{x_0}$ where $\overrightarrow{r}$ is a row of M_0. By assumption, there exists at least one matrix M_j such that $\overrightarrow{r} \notin \mathrm{Im}(M_j^T)$. It results that $\overrightarrow{r} \cdot \overrightarrow{x_j}$ is a uniform random variable independent of $\overrightarrow{v_1}$, $\ldots$, $\overrightarrow{v_t}$ and the other coordinates of $\overrightarrow{v_0}$ (since M_0 is full-rank). Since the latter holds for all the coordinates of $\overrightarrow{x_0}$ we get overall uniformity of $(\overrightarrow{v_0} \parallel \cdots \parallel \overrightarrow{v_t})$ which concludes the proof. □

Lemma **??** allows us to reduce the t-probing security of a standard shared circuit to a linear algebra problem. If an adversary exists that can choose the set of probes $\mathcal{P}'$ such that the transposes of induced matrices M_1, $\ldots$, M_t have intersecting images, then the distribution

$\mathsf{ExpReal}_3(\mathcal{A}, C)$:
1: $C' \leftarrow \mathsf{Flatten}(C)$
2: $(\mathcal{P}'', x_1, \ldots, x_N) \leftarrow \mathcal{A}()$
3: $[x_1] \leftarrow \mathsf{Enc}(x_1), \ldots, [x_N] \leftarrow \mathsf{Enc}(x_N)$
4: $(v_1, \ldots, v_q) \leftarrow C'([x_1], \ldots, [x_N])_{\mathcal{P}''}$
5: Return $(v_1, \ldots, v_q)$

$\mathsf{ExpSim}_3(\mathcal{A}, \mathcal{S}, C)$:
1: $C' \leftarrow \mathsf{Flatten}(C)$
2: $(\mathcal{P}'', x_1, \ldots, x_N) \leftarrow \mathcal{A}()$
3: $(v_1, \ldots, v_q) \leftarrow \mathcal{S}(\mathcal{P}'')$
4: Return $(v_1, \ldots, v_q)$

Figure 4.9: Game 3.

of $(v_1, \ldots, v_q)$ depends on $(x_1, \ldots, x_N)$ and a perfect simulation is impossible (which means that the circuit is not probing secure). Otherwise, the tuple $(v_1, \ldots, v_q)$ can always be simulated by a uniform distribution and the circuit is probing secure. This statement is the basis of our verification method depicted in the next section. But before introducing our verification method, we can still simplify the probing security game as shown hereafter by using Lemma **??**.

4.4.4 Game 3.

In this last game, the adversary is restricted to probe the multiplication gadgets only. Formally, $\mathcal{A}$ returns a set of probes $\mathcal{P}' = \mathcal{P}'_r \cup \mathcal{P}'_m \cup \mathcal{P}'_a$ such that $\mathcal{P}'_r = \emptyset$ and $\mathcal{P}'_a = \emptyset$. Such a set, denoted $\mathcal{P}''$ is hence composed of t pairs of inputs from $\bigcup_{g \in \mathcal{G}_m} \mathcal{I}_g \times \mathcal{J}_g$. The evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}''}$ then returns a q-tuple for $q = 2t$. The new experiments $\mathsf{ExpReal}_3(\mathcal{A}, C)$ and $\mathsf{ExpSim}_3(\mathcal{A}, \mathcal{S}, C)$, displayed in Figure **??**, each output a q-tuple and, as before, the simulator wins Game 3 if and only if the associated distributions are identical.

Proposition 4.4.5. *A standard shared circuit C is t-probing secure if and only if for every adversary $\mathcal{A}$, there exists a simulator $\mathcal{S}$ that wins Game 3 defined above,* i.e. *the random experiments $\mathsf{ExpReal}_3(\mathcal{A}, C)$ and $\mathsf{ExpSim}_3(\mathcal{A}, \mathcal{S}, C)$ output identical distributions.*

Proof. We first show:

$$\forall \mathcal{A}_2, \exists \mathcal{S}_2 \text{ wins Game 2} \Rightarrow \forall \mathcal{A}_3, \exists \mathcal{S}_3 \text{ wins Game 3 } (\Leftrightarrow C \text{ } t\text{-probing secure}) \tag{4.14}$$

For each adversary $\mathcal{A}_3$ that returns a set of inputs $(x_1, \ldots, x_N)$ and a set probes $\mathcal{P}''$, we define an adversary $\mathcal{A}_2$ that outputs the same set of inputs and the same set of probes $\mathcal{P}''$. By assumption, there exists $\mathcal{S}_2$ that can perfectly simulate $C'([x_1], \ldots, [x_N])_{\mathcal{P}''}$ and win Game 2. As a consequence, the simulator $\mathcal{S}_3 = \mathcal{S}_2$ wins Game 3 as well. We now show:

$$\forall \mathcal{A}_3, \exists \mathcal{S}_3 \text{ wins Game 3} \Rightarrow \forall \mathcal{A}_2, \exists \mathcal{S}_2 \text{ wins Game 2 } (\Leftrightarrow C \text{ } t\text{-probing secure}) \tag{4.15}$$

which is equivalent to show the contrapositive statement:

$$\exists \mathcal{A}_2, \forall \mathcal{S}_2 \text{ fails Game 2} \Rightarrow \exists \mathcal{A}_3, \forall \mathcal{S}_3 \text{ fails Game 3 } (\Leftrightarrow C \text{ } t\text{-probing secure}) \tag{4.16}$$

We denote by $(x_1, \ldots, x_N)$ the set of inputs and by $\mathcal{P}'$ the set of probes returned by $\mathcal{A}_2$. As previously, we denote $\mathcal{P}' = \mathcal{P}'_a \cup \mathcal{P}'_r \cup \mathcal{P}'_m$ such that $\mathcal{P}'_a$ are the probes on addition gadgets, $\mathcal{P}'_r$ are probes on refresh gadgets inputs, and $\mathcal{P}'_m$ are probes on pairs of inputs of multiplication gadgets. We further denote by M_0, $\ldots$, M_t the induced matrices from the probes $\mathcal{P}'$ as

defined in Lemma **??**. By assumption of the contrapositive statement Equation (**??**), we have

$$\mathrm{Im}(M_0) \cap \mathrm{Im}(M_1) \cap \cdots \cap \mathrm{Im}(M_t) \neq \emptyset \ .$$

Moreover, we have $q \leq 2t$ implying that at least one M_j matrix has a single row and consequently the above intersection is of dimension one *i.e.* it is defined as the span of a single vector $\overrightarrow{w} \in \mathbb{F}_2^N$:

$$\mathrm{Im}(M_0) \cap \mathrm{Im}(M_1) \cap \cdots \cap \mathrm{Im}(M_t) = \langle \overrightarrow{w} \rangle \ . \tag{4.17}$$

Since $\overrightarrow{w}$ is the single row of at least one M_j matrix we have that $\overrightarrow{w}$ is directly induced by a probed variable v_k. In other words, a sharing $(\overrightarrow{w} \cdot \overrightarrow{x_0}, \ldots, \overrightarrow{w} \cdot \overrightarrow{x_t})$ appears in the circuit (either as input of some gadget, or as output of an addition gadget). We now argue that this sharing must appear in input of a multiplication gadget. Assume by contradiction that this sharing does not appear in input of a multiplication gadget, then it appears in a refresh or an addition gadget. Let us then denote by t_{ar} the number of matrices M_j that have $\overrightarrow{w}$ as row (*i.e.* the jth share of the considered sharing has been probed). The remaining $t - t_{ar}$ matrices M_j have at least 2 rows (since otherwise their image does not include $\overrightarrow{w}$). We deduce that $q \geq t_{ar} + 2(t - t_{ar}) = 2t + t_{ar}$ which is impossible since $q \leq 2t$. We hence obtain that $\overrightarrow{w}$ must be induced by a sharing $(\overrightarrow{w} \cdot \overrightarrow{x_0}, \ldots, \overrightarrow{w} \cdot \overrightarrow{x_t})$ in input of a multiplication gadget.

We can then define $\mathcal{A}_3$ as the adversary that outputs the same set of inputs than $\mathcal{A}_2$ and a set of probes $\mathcal{P}''$ defined according to $\mathcal{P}'$ as follows:

- for every pair $(i_1, i_2) \in \mathcal{P}'_m$, include (i_1, i_2) to $\mathcal{P}''$,
- for every probe $i \in \mathcal{P}'_a \cup \mathcal{P}'_r$, let j be the share index corresponding to the wire indexed by i, then include the wire index of the multiplication input share $\overrightarrow{w} \cdot \overrightarrow{x_j}$.

It is not hard to see that the M_j matrices induced by the new set of probes $\mathcal{P}''$ still satisfies Equation (**??**) which implies that no simulator $\mathcal{S}_3$ can produce a perfect simulation of $C'([x_1], \ldots, [x_N])_{\mathcal{P}''}$. In other words, our contrapositive statement Equation (**??**) holds which concludes the proof. □

4.5 Probing-Security Verification for Standard Shared Circuits

In this section, we describe a formal verification method that for any $t \in \mathbb{N}$ checks whether a standard $(t+1)$-shared circuit C achieves t-probing security. Specifically, our tool either provides a formal proof that C is t-probing secure for every $t \in \mathbb{N}$ (where C is a standard shared circuit with sharing order $t+1$), or it exhibits a *probing attack* against C for a given t, namely it finds a set of probes $\mathcal{P}$ (indices of wires) in the $(t+1)$-shared instance of C, such that $|\mathcal{P}| = t$, for which the evaluation $C([x_1], \ldots, [x_n])_{\mathcal{P}}$ cannot be simulated without some knowledge on the plain input $(x_1, \ldots, x_n)$.

4.5.1 Linear Algebra Formulation

As demonstrated in the previous section, the t-probing security game for a standard $(t+1)$-shared circuit C can be reduced to a game where an adversary selects a set of probes $\mathcal{P}''$ solely pointing to input shares of the multiplication gadgets of a *flattened* circuit C'. In the following, we will denote by m the number of multiplication gadgets in C (or equivalently in

C') and by $g \in [\![1, m]\!]$ the index of a multiplication gadget of C. We will further denote by $[a_g]$ and $[b_g]$ the input sharings of the g-th multiplication gadget so that we have

$$[a_g] = (\overrightarrow{a_g} \cdot \overrightarrow{x_0}, \ldots, \overrightarrow{a_g} \cdot \overrightarrow{x_t}) \text{ and } [b_g] = (\overrightarrow{b_g} \cdot \overrightarrow{x_0}, \ldots, \overrightarrow{b_g} \cdot \overrightarrow{x_t}) \ , \tag{4.18}$$

for some constant coefficient vectors $\overrightarrow{a_g}, \overrightarrow{b_g} \in \mathbb{F}_2^N$, recalling that $\overrightarrow{x_j}$ denotes the vector with the jth share of each input sharing $[x_1]$, ..., $[x_N]$. In the following, the vectors $\{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ are called the *operand vectors*.

In Game 3, the adversary chooses t pairs of probes such that each pair points to one share of $[a_g]$ and one share of $[b_g]$ for a multiplication gadget g. Without loss of generality, the set of pairs output by the adversary can be relabelled as a set of triplet $\mathcal{P} = \{(g, j_1, j_2)\}$ where $g \in [\![1, m]\!]$ is the index of a multiplication gadget, j_1 and j_2 are share indices. For any triplet $(g, j_1, j_2) \in \mathcal{P}$ the two input shares $\overrightarrow{a_g} \cdot \overrightarrow{x_{j_1}}$ and $\overrightarrow{b_g} \cdot \overrightarrow{x_{j_2}}$ are added to the $(2t)$-tuple of probed variables to be simulated. This set of triplets exactly defines a sequence of $t+1$ matrices M_1, ..., M_t, defined iteratively by adding $\overrightarrow{a_g}$ to the rows of M_{j_1} and $\overrightarrow{b_g}$ to the rows of M_{j_2} for each $(g, j_1, j_2) \in \mathcal{P}$. Equivalently, the matrix M_j is defined as

$$M_j = \mathsf{rows}(\{\overrightarrow{a_g} \ ; \ (g, j, *) \in \mathcal{P}\} \cup \{\overrightarrow{b_g} \ ; \ (g, *, j) \in \mathcal{P}\}) \ , \tag{4.19}$$

for every $j \in [\![0, t]\!]$ where rows maps a set of vectors to the matrix with rows from this set.

Lemma **??** then implies that a probing attack on C consists of a set of probes $\mathcal{P} = \{(g, j_1, j_2)\}$ such that the transposes of the induced M_j have intersecting images. Moreover, since the total number of rows in these matrices is $2t$, at least one of them has a single row $\overrightarrow{w}$. In particular, the image intersection can only be the span of this vector (which must match the row of all single-row matrices) and this vector belongs to the set of operand vectors $\{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$. In other words, there exists a probing attack on C if and only if a choice of probes $\mathcal{P} = \{(g, j_1, j_2)\}$ implies

$$\mathrm{Im}(M_0^T) \cap \mathrm{Im}(M_1^T) \cap \cdots \cap \mathrm{Im}(M_t^T) = \langle \overrightarrow{w} \rangle \ . \tag{4.20}$$

for some vector $\overrightarrow{w} \in \{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$. In that case we further say that there is a probing attack on the operand vector $\overrightarrow{w}$.

In the remainder of this section, we describe an efficient method that given a set of vector operands $\{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ (directly defined from a target circuit C) determines whether there exists a parameter t and a set $\mathcal{P} = \{(g, j_1, j_2)\}$ (of cardinality t) for which Equation (**??**) can be satisfied. We prove that (1) if such sets $\mathcal{P}$ exist, our method returns one of these sets, (2) if not sets is returned by our method then the underlying circuit is t-probing secure for any sharing order $(t+1)$.

4.5.2 Method Description

The proposed method loops over all the vector operands $\overrightarrow{w} \in \{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ and checks whether there exists a probing attack on $\overrightarrow{w}$ (*i.e.* whether a set $\mathcal{P}$ can be constructed that satisfies Equation (**??**)).

For each $\overrightarrow{w} \in \{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ the verification method is iterative. It starts from a set $\mathcal{G}_1 \subseteq [\![1, m]\!]$ defined as

$$\mathcal{G}_1 = \{g \ ; \ \overrightarrow{a_g} = \overrightarrow{w}\} \cup \{g \ ; \ \overrightarrow{b_g} = \overrightarrow{w}\} \ . \tag{4.21}$$

Namely $\mathcal{G}_1$ contains the indices of all the multiplication gadgets that have $\overrightarrow{w}$ as vector operand. Then the set of *free vector operands* $\mathcal{O}_1$ is defined as

$$\mathcal{O}_1 = \{\overrightarrow{b_g} \ ; \ \overrightarrow{a_g} = \overrightarrow{w}\} \cup \{\overrightarrow{a_g} \ ; \ \overrightarrow{b_g} = \overrightarrow{w}\} \ . \tag{4.22}$$

The terminology of *free* vector operand comes from the following intuition: if a probing adversary spends one probe on gadget $g \in \mathcal{G}_1$ such that $\overrightarrow{a_g} = \overrightarrow{w}$ to add $\overrightarrow{w}$ to a matrix M_j (or equivalently to get the share $\overrightarrow{w} \cdot \overrightarrow{x_j}$), then she can also add $\overrightarrow{b_g}$ to another matrix $M_{j'}$ (or equivalently get the share $\overrightarrow{b_g} \cdot \overrightarrow{x_{j'}}$) for *free*. The adversary can then combine several free vector operands to make $\overrightarrow{w} \in \mathrm{Im}(M_{j'}^T)$ occur without directly adding $\overrightarrow{w}$ to $M_{j'}$ (or equivalently without directly probing $\overrightarrow{w} \cdot \overrightarrow{x_{j'}}$). This is possible if and only if $\overrightarrow{w} \in \langle \mathcal{O}_1 \rangle$.

The free vector operands can also be combined with the operands of further multiplications to generate a probing attack on $\overrightarrow{w}$. To capture such higher-degree combinations, we define the sequences of sets $(\mathcal{G}_i)_i$ and $(\mathcal{O}_i)_i$ as follows:

$$\mathcal{G}_{i+1} = \{g \ ; \ \overrightarrow{a_g} \in \overrightarrow{w} + \langle \mathcal{O}_i \rangle\} \cup \{g \ ; \ \overrightarrow{b_g} \in \overrightarrow{w} + \langle \mathcal{O}_i \rangle\} \ , \tag{4.23}$$

and

$$\mathcal{O}_{i+1} = \{\overrightarrow{b_g} \ ; \ \overrightarrow{a_g} \in \overrightarrow{w} + \langle \mathcal{O}_i \rangle\} \cup \{\overrightarrow{a_g} \ ; \ \overrightarrow{b_g} \in \overrightarrow{w} + \langle \mathcal{O}_i \rangle\} \ . \tag{4.24}$$

for every $i \geq 1$. The rough idea of this iterative construction is the following: if at step $i+1$ a probing adversary spends one probe on gadget $g \in \mathcal{G}_{i+1}$ such that $\overrightarrow{a_g} \in \overrightarrow{w} + \langle \mathcal{O}_i \rangle$, then she can add $\overrightarrow{a_g}$ together with some free vector operands of previous steps to M_j in order to get $\overrightarrow{w} \in \mathrm{Im}(M_j^T)$. Then she can also add $\overrightarrow{b_g}$ to another matrix $M_{j'}$, making $\overrightarrow{b_g}$ a new free vector operand of step $i+1$.

Based on these definitions, our method iterates the construction of the sets $\mathcal{G}_i$ and $\mathcal{O}_i$. At step i, two possible stop conditions are tested:

1. if $\mathcal{G}_i = \mathcal{G}_{i-1}$, then there is no probing attack on $\overrightarrow{w}$, the method stops the iteration on $\overrightarrow{w}$ and continues with the next element in the set of vector operands;

2. if $\overrightarrow{w} \in \langle \mathcal{O}_i \rangle$, then there is a probing attack on $\overrightarrow{w}$, the method stops and returns `True` (with $\overrightarrow{w}$ and the sequence of sets $(\mathcal{G}_i, \mathcal{O}_i)_i$ as proof);

The method returns `True` if there exists a concrete probing attack on a vector $\overrightarrow{w} \in \{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ for a certain sharing order $t+1$. Otherwise, it will eventually stops with vector operand $\overrightarrow{w}$ since the number of multiplications is finite and since $\mathcal{G}_i \subseteq \mathcal{G}_{i+1}$ for every $i \geq 1$. When all the possible vector operands have been tested without finding a probing attack, the method returns `False`. Algorithm **??** hereafter gives a pseudocode of our method where NextSets denotes the procedure that computes $(\mathcal{G}_{i+1}, \mathcal{O}_{i+1})$ from $(\mathcal{G}_i, \mathcal{O}_i)$.

Algorithm 3 Search probing attack

Input: A set of vector operands $\{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$

Output: `True` if there is probing attack on some $\overrightarrow{w} \in \{\overrightarrow{a_g}, \overrightarrow{b_g}\}_g$ and `False` otherwise

```
for all w ∈ {a_g, b_g}_g do
    (G_1, O_1) ← NextSets(∅, ∅, {a_g, b_g}_g, w)
    if w ∈ ⟨O_1⟩ then return True
    for i = 1 to m do
        (G_{i+1}, O_{i+1}) ← NextSets(G_i, O_i, {a_g, b_g}_g, w)
        if G_{i+1} = G_i then break
        if w ∈ ⟨O_i⟩ then return True
return False
```

In the rest of the section we first give some toy examples to illustrate our methods and then provides a proof of its correctness.

4.5.3 Toy Examples

Two examples are provided hereafter to illustrate our iterative method in the absence then in the presence of a probing attack.

In the very simple example of Figure **??**, two variables are manipulated in multiplications in the circuit C: $\overrightarrow{w}_1 = \overrightarrow{x_1}$ and $\overrightarrow{w}_2 = \overrightarrow{x_1} + \overrightarrow{x_2}$. The set of multiplications $\mathcal{G}$ is of cardinal one since it only contains one multiplication $(\overrightarrow{w}_1, \overrightarrow{w}_2)$. Following the number of variables, the method proceeds at most in two steps:

1. As depicted in Algorithm **??**, the method first determines whether there exists a probing attack on $\overrightarrow{w}_1$. In this purpose, a first set $\mathcal{G}_1$ is built, such that $\mathcal{G}_1 = (\overrightarrow{w}_1, \overrightarrow{w}_2)$ and $\mathcal{O}_1 = \overrightarrow{w}_2$. Since $\mathcal{G}_1 \neq \emptyset$ and $\overrightarrow{w}_1 \neq \overrightarrow{w}_2$, then a second set must be built. However, there is no multiplication left, that is $\mathcal{G}_2 = \mathcal{G}_1$ and so there is no attack on $\overrightarrow{w}_1$.

2. The method then focuses on $\overrightarrow{w}_2$. In this purpose, a dedicated set $\mathcal{G}_1$ is built, such that $\mathcal{G}_1 = (\overrightarrow{w}_2, \overrightarrow{w}_1)$ and $\mathcal{O}_1 = \overrightarrow{w}_1$. Since $\mathcal{G}_1 \neq \emptyset$ and $\overrightarrow{w}_2 \neq \overrightarrow{w}_1$, then a second set must be built. However, there is no multiplication left, that is $\mathcal{G}_2 = \mathcal{G}_1$ and so there is no attack on $\overrightarrow{w}_2$ either. Since there is no input variable left, the method returns `False`, which means that there is no possible probing attack on this circuit.

Figure **??** provides a second Boolean circuit. It manipulates five variables $\overrightarrow{w}_i$ as operands of multiplication gadgets: $\overrightarrow{w}_1 = \overrightarrow{x_1}$, $\overrightarrow{w}_2 = \overrightarrow{x_2}$, $\overrightarrow{w}_3 = \overrightarrow{x_3}$, $\overrightarrow{w}_4 = \overrightarrow{x_1} + \overrightarrow{x_2}$, and $\overrightarrow{w}_5 = \overrightarrow{x_2} + \overrightarrow{x_3}$. The set of multiplications $\mathcal{G}$ is of cardinal three with $(\overrightarrow{w}_1, \overrightarrow{w}_2)$, $(\overrightarrow{w}_4, \overrightarrow{w}_5)$, and $(\overrightarrow{w}_3, \overrightarrow{w}_4)$. Following the number of variables, the method proceeds at most in five steps:

1. The method first determines whether there exists a probing attack on $\overrightarrow{w}_1$. In this purpose, a first set $\mathcal{G}_1$ is built, such that $\mathcal{G}_1 = (\overrightarrow{w}_1, \overrightarrow{w}_2)$ and $\mathcal{O}_1 = \overrightarrow{w}_2$. Since $\mathcal{G}_1 \neq \emptyset$ and $\overrightarrow{w}_1 \neq \overrightarrow{w}_2$, then a second set must be built. $\mathcal{G}_2 = \mathcal{G}_1 \cup \{(\overrightarrow{w}_4, \overrightarrow{w}_5), (\overrightarrow{w}_4, \overrightarrow{w}_3)\}$ since $\overrightarrow{w}_4 = \overrightarrow{w}_1 + \overrightarrow{w}_2$. However, $\overrightarrow{w}_1 \notin \mathcal{O}_2 (=< \overrightarrow{w}_2, \overrightarrow{w}_3, \overrightarrow{w}_5 >)$, so a third set must be built. Since there is no multiplication left, that is $\mathcal{G}_3 = \mathcal{G}_2$, there is no attack on $\overrightarrow{w}_1$.

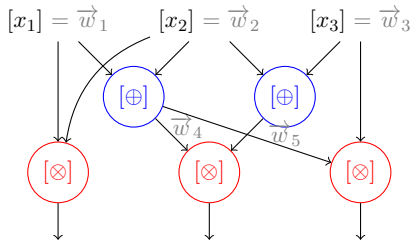


Figure 4.10: Graph representation of a second Boolean circuit.

2. The method then focuses on $\overrightarrow{w}_2$. In this purpose, a dedicated set $\mathcal{G}_1$ is built, such that $\mathcal{G}_1 = (\overrightarrow{w}_2, \overrightarrow{w}_1)$ and $\mathcal{O}_1 = \overrightarrow{w}_1$. Since $\mathcal{G}_1 \neq \emptyset$ and $\overrightarrow{w}_2 \neq \overrightarrow{w}_1$, then a second set must be built. $\mathcal{G}_2 = \mathcal{G}_1 \cup \{(\overrightarrow{w}_4, \overrightarrow{w}_5), (\overrightarrow{w}_4, \overrightarrow{w}_3)\}$ since $\overrightarrow{w}_4 = \overrightarrow{w}_2 + \overrightarrow{w}_1$. And in that case, $\overrightarrow{w}_2 \in \mathcal{O}_2 (=< \overrightarrow{w}_1, \overrightarrow{w}_3, \overrightarrow{w}_5 >)$ since $\overrightarrow{w}_2 = \overrightarrow{w}_3 + \overrightarrow{w}_5$. Thus the method returns `True` and there exists an attack on $\overrightarrow{w}_2 = \overrightarrow{x_2}$ for some masking order t.

4.5.4 Proof of Correctness

This section provides a proof of correctness of the method. This proof is organized in two propositions which are based on some invariants in Algorithm **??**. The first proposition shows that if the method returns `True` for some operand vector $\overrightarrow{w}$ and corresponding sets $(\mathcal{G}_i, \mathcal{O}_i)$ then there exists a probing attack on $\overrightarrow{w}$ (*i.e.* a set $\mathcal{P}$ can be constructed that satisfies Equation (**??**)). The second proposition shows that if the method returns `False` then there exists no probing attack for any $\overrightarrow{w}$, namely the underlying circuit is t-probing secure as soon as masked variables are masked with $t + 1$ shares.

Proposition 4.5.1. *For every $i \in \mathbb{N}$, if $\overrightarrow{w} \in \langle \mathcal{O}_i \rangle$ then there exists $t \in \mathbb{N}$ and $\mathcal{P} = \{(g, j_1, j_2)\}$ with $|\mathcal{P}| = t$ implying $\bigcap_{j=0}^{t} \mathrm{Im}(M_j^T) = \overrightarrow{w}$.*

Proof of Proposition **??**. To prove this proposition, we demonstrate by induction the following invariant:

Invariant: $\forall s \in \mathbb{N}$, $\exists\, t \in \mathbb{N}$ such that with t carefully chosen probes on multiplications from $\mathcal{G}_i$, we are able to get:

- r matrices M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$, $0 \leq j \leq r-1$, where $r = t + 1 - s$;
- s matrices M_j such that $\langle \mathcal{O}_i \rangle \subseteq \mathrm{Im}(M_j)$, $r \leq j \leq t$.

We show that the invariant holds for $i = 1$. Let $s \in \mathbb{N}$ and let $\ell_1 = |\mathcal{O}_1|$. If we place s probes on each multiplication gadget $g \in \mathcal{G}_1$, we can have $r = s \cdot \ell_1$ matrices $M_j = \mathsf{rows}(\overrightarrow{w})$, and s matrices $M_j = \mathsf{rows}(\mathcal{O}_1)$. We thus get the desired invariant with $t = r + s - 1 = s(\ell_1 + 1) - 1$.

We now show that the invariant holds for $i + 1$ if it holds for i. Let $s \in \mathbb{N}$ and let $\ell_{i+1} = |\mathcal{O}_{i+1}|$. By assumption, for $s' = s \cdot (\ell_{i+1} + 1)$, there exists t' such that with t' carefully chosen probes on multiplications from $\mathcal{G}_i$, we are able to get:

- r' matrices M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$, $0 \leq j \leq r' - 1$, where $r' = t' + 1 - s'$;
- s' matrices M_j such that $\langle \mathcal{O}_i \rangle \subseteq \mathrm{Im}(M_j)$, $r' \leq j \leq t'$.

In what follows, the s' last matrices are called the *unfinished matrices*. If we place s probes on each multiplication gadget $g \in \mathcal{G}_{i+1}$, we can add a vector operand from $\overrightarrow{w} + \langle \mathcal{O}_{i+1} \rangle$ to $s \cdot \ell_{i+1}$ of the unfinished matrices. We thus obtain $s \cdot \ell_{i+1}$ more matrices M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$. We can further add all the ℓ_{i+1} operands from $\mathcal{O}_{i+1}$ to the s remaining unfinished matrices. We then get s matrices M_j such that $\langle \mathcal{O}_{i+1} \rangle \subseteq \mathrm{Im}(M_j)$, which show the inductive statement.

From the above invariant, we can easily demonstrate the proposition statement. Indeed if we have $\overrightarrow{w} \in \langle \mathcal{O}_i \rangle$ for some $i \in \mathbb{N}$ then the invariant implies that for $s = 1$, there exists $t \in \mathbb{N}$ and $\mathcal{P} = \{(g, j_1, j_2)\}$ such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$ for $0 \leq j \leq t$ and $\langle \mathcal{O}_i \rangle \subseteq \mathrm{Im}(M_t)$, implying $\overrightarrow{w} \in \mathrm{Im}(M_t)$ as well. We then get $\bigcap_{j=0}^{t} \mathrm{Im}(M_j) = \overrightarrow{w}$. □

Proposition 4.5.2. *Let $i > 1$ such that $\mathcal{G}_1 \subset \cdots \subset \mathcal{G}_{i-1} = \mathcal{G}_i$ and $\overrightarrow{w} \notin \langle \mathcal{O}_i \rangle$. Then for any $t \in \mathbb{N}$ and $\mathcal{P} = \{(g, j_1, j_2)\}$ with $|\mathcal{P}| = t$ we have $\overrightarrow{w} \notin \bigcap_{j=0}^{t} \mathrm{Im}(M_j^T)$.*

Proof of Proposition **??**. Let us denote $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$ such that

$$\mathcal{P}_1 = \{(g, j_1, j_2) \ ; \ g \in \mathcal{G}_i\} \quad \text{and} \quad \mathcal{P}_2 = \{(g, j_1, j_2) \ ; \ g \notin \mathcal{G}_i\}$$

with $|\mathcal{P}_1| = t_1$ and $|\mathcal{P}_2| = t_2$, with $t_1 + t_2 = t$. The set $\mathcal{P}_1$ provides at most t_1 matrices M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$ plus t_1 operand vectors from $\mathcal{O}_i$ to be distributed among the remaining matrices. Then the set $\mathcal{P}_2$ provides $2t_2$ additional vectors from $\{\overrightarrow{a_g}, \overrightarrow{b_g} \ ; \ g \notin \mathcal{G}_i\}$ to be distributed among the remaining matrices. However none of these additional vectors is included $\overrightarrow{w} + \langle \mathcal{O}_i \rangle$ which implies that at least two of them are necessary to produce one additional matrix M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$. We conclude that we can get at most $t_1 + t_2 = t$ matrices M_j such that $\overrightarrow{w} \in \mathrm{Im}(M_j)$ which implies $\overrightarrow{w} \notin \bigcap_{j=0}^{t} \mathrm{Im}(M_j)$. □

4.5.5 Towards Efficient Construction of Tight t-Private Circuits

Our formal verification method exactly reveals all the t-probing attacks on standard shared circuits. A sound countermeasure to counteract these attacks is the use of refresh gadgets. We discuss here how to transform a flawed standard shared circuit into a t-private circuit with exactly the minimum number of refresh gadgets.

In a first attempt, we easily show that refreshing the left operands of each multiplication in C is enough to provide t-probing security.

Proposition 4.5.3. *A standard shared circuit C augmented with t-SNI refresh gadgets operating on the left operand of each multiplication gadget is t-probing secure.*

Proof of Proposition **??**. Let C be a standard shared circuit augmented with t-SNI refresh gadgets operating on the left operand of each multiplication gadget. From Corollary **??**, the analysis of the t-probing security of C can be reduced to the analysis of the t-probing security of $\mathsf{Flatten}(C)$. In the latter, each multiplication takes as its left operand a new fresh encoding. Now let us assume that there exists a probing attack on C. We know from the linear algebra formulation above that this attack is characterized by a vector $\overrightarrow{w}$ and a set of $t + 1$ matrices such that

$$\mathrm{Im}(M_0) \cap \mathrm{Im}(M_1) \cap \cdots \cap \mathrm{Im}(M_t) = \langle \overrightarrow{w} \rangle \ . \tag{4.25}$$

We also know that there exists at least one index $0 \leq i \leq t$, such that matrix M_i is completely defined by the row vector $\overrightarrow{w}$. Now let us assume that $\overrightarrow{w}$ represents a probe on the left operand of a multiplication. Since this operand is a new fresh encoding that is used nowhere else, then it cannot be recovered from the linear combination of other operands. As a consequence, all the matrices must be defined by the same row vector $\overrightarrow{w}$. But at most t probes are available to target this last operand which is not enough to feed the $t+1$ matrices and consequently leads to a contradiction. Let us now assume that $\overrightarrow{w}$ represents a probe on the right operand of a multiplication. In that case, probes on right operands (including probe $\overrightarrow{w}$) can feed up to t matrices in order to fulfill Equation (**??**). Without loss of generality, we assume these matrices to be $M_0, \dots M_{t-1}$. The last matrix M_t is then necessarily built from probes on left operands. Since all of them are fresh encodings, then $\mathrm{Im}(M_t)$ cannot include $\overrightarrow{w}$, which gives the second contradiction and completes the proof. □

In a second attempt, we need to slightly modify Algorithm **??** so that it conducts an analysis on all the possible operands in order to return a complete list of the flawed ones. So far, it stops at the first flaw. With such a list for a standard shared circuit, we can show that refreshing only the flawed operands is enough to provide t-probing security.

Proposition 4.5.4. *A standard shared circuit C augmented with t-SNI refresh gadgets operating on each flawed operand, as revealed by our method, of its multiplication gadgets is t-probing secure.*

Proof of Proposition **??**. Let us consider a standard shared circuit C augmented with t-SNI refresh gadgets operating on each one of its α flawed operands. For each of these α flawed operands represented by the vector $\overrightarrow{w}$, there are a certain number β of sets of probes associated to sets of matrices $(M_i^j)_{0 \leq i \leq t}$ for $(1 \leq j \leq \beta)$ whose intersecting images are equal to $\overrightarrow{w}$. In each of these β sets of matrices, at least one matrix is exactly equal to $\overrightarrow{w}$. Refreshing the corresponding operand each time it is used in a multiplication makes it impossible to get a matrix equal to $\overrightarrow{w}$ anymore in any of the β sets. As a consequence, all these sets of probes do not lead to a probing attack anymore. Furthermore, since we only turned operands into fresh encodings that are not reused, then this transformation do not lead to new probing attacks. □

Propositions **??** and **??** provide an upper bound of the required number of refresh gadgets in a standard shared circuit to achieve probing security at any order t. If we denote by m the number of multiplications in a standard shared circuit C and by o the number of flawed operands returned by our method, then C is to be augmented of at most $r = \min(m, o)$ refresh gadgets to achieve probing security at any order t. Given this upper bound, an iterative number of refresh gadgets from 1 to r can be inserted at each location in C in order to exhibit a tight private circuit with a minimum number of refresh gadgets.

4.6 Further Steps

Now that we are able to exactly determine the t-probing security of standard shared circuits, a natural follow-up consists in studying the t-probing security of their composition. In a first part, we establish several compositional properties, and then we show how they apply to the widely deployed SPN-based block ciphers. We eventually discuss the extension of our results to generic shared circuits.

4.6.1 Generic Composition

This section is dedicated to the statement of new compositional properties on tight private circuits. In a first attempt, we show that the composition of a t-private circuit whose outputs coincide with the outputs of t-SNI gadgets with another t-private circuit is still a t-private circuit.

Proposition 4.6.1. *Let us consider a standard shared circuit C composed of two sequential circuits:*

- *a t-probing secure circuit C_1 whose outputs are all outputs of t-SNI gadgets,*
- *a t-probing secure circuit C_2 whose inputs are C_1's outputs.*

Then, $C = C_2 \circ C_1$ is t-probing secure.

Proof. As the outputs of the first circuit C_1 are the outputs t-SNI gadgets, we get from Lemma **??** that the input encodings of C_1 and the input encodings of C_2 are independent and uniformly distributed. Then, the proof is straightforward from Proposition **??**. Basically, the analysis of C's t-probing security can be equivalently reduced to the analysis of the t-probing security of $C' = \mathsf{Flatten}(C)$ in which each output of a t-SNI gadget is replaced by a fresh new input sharing of the corresponding value in the rest of the circuit, *i.e.* C_2. As a consequence, C is t-probing secure if and only if both C_1 and C_2 are t-probing secure, which is correct by assumption. □

In a second attempt, we establish the secure composition of a standard shared circuit that implements a (shared) linear surjective transformation through several sharewise addition gadgets, that we refer to as a t-linear surjective circuit, and a standard t-probing circuit.

Proposition 4.6.2. *Let us consider a standard shared circuit C composed of two sequential circuits:*

- *a t-linear surjective circuit C_1, exclusively composed of sharewise additions,*
- *a t-probing secure circuit C_2 whose inputs are C_1's outputs.*

Then, $C = C_2 \circ C_1$ is t-probing secure.

Proof. We consider a standard shared circuit C with input $\overrightarrow{x} = (x_1, \ldots, x_n)$ composed of a t-linear surjective circuit C_1 as input to a t-probing secure circuit C_2. We denote by $\overrightarrow{y} = (y_1, \ldots, y_{n'})$ the set of C_1's outputs, or equivalently the set of C_2's inputs. From Proposition **??**, the t-probing security of C can be reduced to the t-probing security of circuit $C' = \mathsf{Flatten}(C)$ for probes restricted to the multiplications' operands. In our context, C_1 is exclusively composed of sharewise additions, so the probes are restricted to C_2. From Lemma **??**, any set of probed variables on C_2's multiplications operands $(v_1, \ldots, v_q)$ can be written as the concatenation of the $t+1$ vectors

$$\overrightarrow{v_0} = M_0 \cdot \overrightarrow{y_0} \ , \quad \overrightarrow{v_1} = M_1 \cdot \overrightarrow{y_1} \ , \quad \ldots \quad \overrightarrow{v_t} = M_t \cdot \overrightarrow{y_t} \ ,$$

where

$$\mathrm{Im}(M_0^T) \cap \mathrm{Im}(M_1^T) \cap \cdots \cap \mathrm{Im}(M_t^T) = \emptyset \ . \tag{4.26}$$

To achieve global t-probing security for C, we need to achieve a null intersection for matrices that apply on C's inputs instead of C_2's inputs. As C_1 implements a linear surjective transformation f, there exists a matrix M_f of rank n' such that

$$\forall\ 0 \leq i \leq t, \quad \overrightarrow{y_i} = M_f \cdot \overrightarrow{x_i}.$$

As a consequence, any set of probes $(v_1, \ldots, v_q)$ in C' as defined in Game 3 can equivalently be rewritten as the concatenation of the $t+1$ vectors

$$\overrightarrow{v_0} = M_0 \cdot M_f \cdot \overrightarrow{x_0}\ , \quad \overrightarrow{v_1} = M_1 \cdot M_f \cdot \overrightarrow{x_1}\ , \quad \ldots \quad \overrightarrow{v_t} = M_t \cdot M_f \cdot \overrightarrow{x_t}.$$

By contradiction, let us assume that

$$\mathrm{Im}(M_f^T \cdot M_0^T) \cap \mathrm{Im}(M_f^T \cdot M_1^T) \cap \cdots \cap \mathrm{Im}(M_f^T \cdot M_t^T) \neq \emptyset,$$

that is, there exists a non-null vector $\overrightarrow{w}$ such that

$$\overrightarrow{w} \in \mathrm{Im}(M_f^T \cdot M_0^T) \cap \mathrm{Im}(M_f^T \cdot M_1^T) \cap \cdots \cap \mathrm{Im}(M_f^T \cdot M_t^T).$$

Equivalently, there exists $\overrightarrow{z_0}, \overrightarrow{z_1}, \ldots, \overrightarrow{z_t}$ such that

$$\overrightarrow{w} = M_f^T \cdot M_0^T \cdot \overrightarrow{z_0} = M_f^T \cdot M_1^T \cdot \overrightarrow{z_1} = \ldots = M_f^T \cdot M_1^T \cdot \overrightarrow{z_t}.$$

From Equation (**??**), there exist at least two distinct indices i and j in $\{0, \ldots, t\}$, such that

$$M_i^T \cdot \overrightarrow{z_i} \neq M_j^T \cdot \overrightarrow{z_j}.$$

As $\overrightarrow{w} = M_f^T \cdot M_i^T \cdot \overrightarrow{z_i} = M_f^T \cdot M_j^T \cdot \overrightarrow{z_j}$, the difference $M_i^T \cdot \overrightarrow{z_i} - M_j^T \cdot \overrightarrow{z_j}$ belongs to M_f^T's kernel. But from the surjective property of M_f, M_f^T has full column rank n', and thus a null kernel:

$$\dim(\mathrm{Ker}(M_f^T)) = n' - \dim(\mathrm{Im}(M_f^T)) = 0.$$

As a consequence, $M_i^T \cdot \overrightarrow{z_i} - M_j^T \cdot \overrightarrow{z_j} = 0$ and since $M_i^T \cdot \overrightarrow{z_i} \neq M_j^T \cdot \overrightarrow{z_j}$ we have a contradiction which completes the proof. □

Eventually, we claim that two t-private circuits on independent encodings form a t-private circuit as well.

Proposition 4.6.3. *Let us consider a standard shared circuit C composed of two parallel t-probing secure circuits which operate on independent input sharings. Then, $C = C_1 \| C_2$ is t-probing secure.*

Proof. As the input sharings are independent, the result is straightforward from Lemma **??**. □

4.6.2 Application to SPN-Based Block Ciphers

An SPN-based block cipher is a permutation which takes as inputs a key k in $\{0,1\}^\kappa$ and a plaintext p in $\{0,1\}^n$ and outputs a ciphertext c in $\{0,1\}^n$, where n and κ are integers. As illustrated in Figure **??**, it is defined by successive calls to a round function and by an optional expansion algorithm KS. The round function is a combination of a non linear permutation S and a linear permutation L.

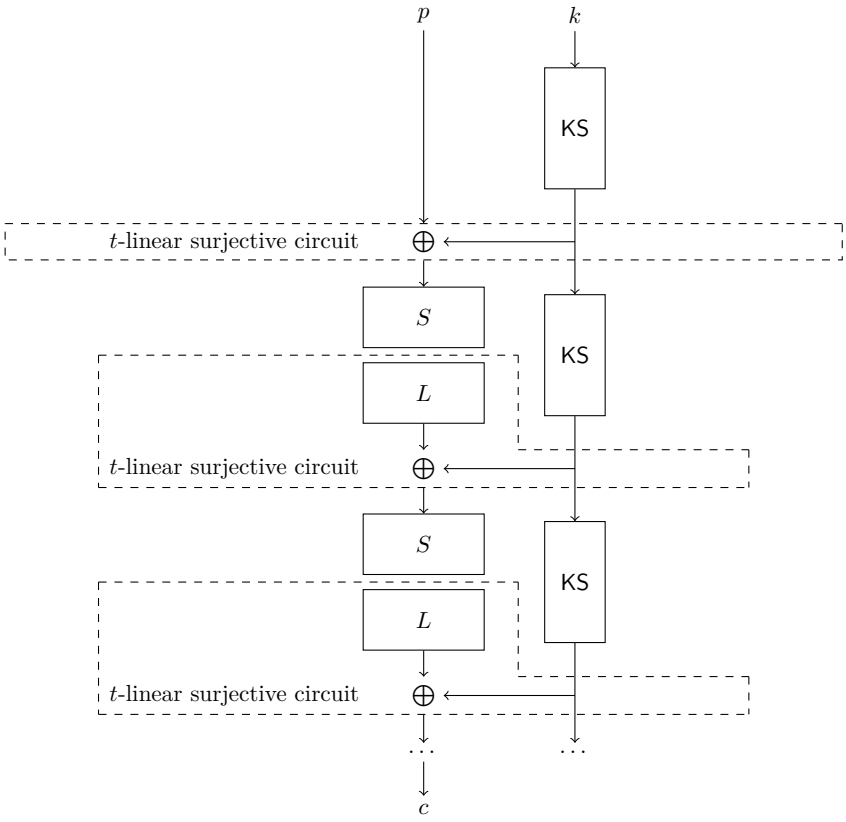


Figure 4.11: Structure of an SPN-Based Block Cipher.

Proposition 4.6.4. *Let C be a standard shared circuit implementing an SPN block cipher as pictured in Figure* **??**. *And let C_S and C_{KS} be the standard shared (sub-)circuits implementing S and KS respectively. If both conditions*

1. *C_S's and C_{KS}'s outputs are t-SNI gadgets' outputs,*
2. *C_S and C_{KS} are t-probing secure (for any sharing order $t+1$),*

are fulfilled, then C is also t-probing secure.

Note that if S's and KS's outputs are not t-SNI gadgets' outputs, then the linear surjective circuit can be extended to the last t-SNI gadgets' outputs of these circuits without loss of generality.

Proof. As S and KS are t-probing secure, it follows from Proposition **??**, that when implemented in parallel on independent input encodings, their composition is t-probing secure as well. Then, as the output of their composition matches the outputs of t-SNI gadgets, then they can be sequentially composed with a t-probing secure circuit from Proposition **??**. Finally, the composition of linear surjective circuits with t-probing secure circuits is ensured by Proposition **??**, which completes the proof. □

Remark 4.6.5. *This result can not be directly applied to the AES full block cipher as the outputs of the key schedule function are not SNI gadgets.*

4.6.3 Extension to Generic Shared Circuits

We discuss hereafter two straightforward extensions of our work. Namely some constraints on gadgets that compose the standard shared circuits can be relaxed, and the considered circuit can easily be extended to work on larger finite fields.

4.6.3.1 On Standard Shared Circuits.

The method presented in this Chapter through Sections **??** and **??** aims to accurately establish the t-probing security of a *standard shared circuit* for any sharing order $t+1$. Namely, it is restricted to Boolean shared circuits exclusively composed of ISW-multiplication gadgets, ISW-refresh gadgets, and sharewise addition gadgets. While the assumption on addition gadgets is quite natural, the restrictions made on the multiplication and refresh gadgets can be relaxed. The reduction demonstrated in Section **??** only expects the refresh gadgets to be t-SNI secure to ensure the equivalence between Game 1 and the initial t-probing security game. Afterwards, t-probing security is equivalently evaluated on a corresponding *flattened* circuit with probes on multiplications' operands only. Therefore, there is no restriction on the choice of refresh gadgets but their t-SNI security. While multiplication gadgets are also expected to be t-SNI secure for the equivalence between Game 1 and the initial t-probing security game to hold, this feature is not enough. To prove this equivalence, multiplication gadgets are also expected to compute intermediate products between every share of their first operand and every share of their second operand. Otherwise, our method could still establish the probing security of a circuit, but not in a tight manner, meaning that security under Game 3 would imply probing security but insecurity under Game 3 would not imply insecurity w.r.t. the original probing insecurity notion. Our method would hence allow false negatives, as state-of-the-art methods currently do. Beyond the advantages of providing an

exact method, this restriction is not very constraining since not only the widely deployed ISW-multiplication gadgets but also the large majority of existing multiplication gadgets achieve this property.

4.6.3.2 On Circuits on Larger Fields.

Since ISW-multiplication gadgets and ISW-refresh gadgets can straightforwardly be extended to larger fields our reduction and verification method could easily be extended to circuits working on larger fields.

4.7 Application

Following the results presented in previous sections, we developed a tool in sage that takes as input a standard shared circuit and determines whether or not it is t-probing secure with Algorithm **??**. Specifically, the standard shared circuit given as input to the tool is expressed as a set of instructions (`XOR`, `AND`, `NOT`, `REFRESH`) with operands as indices of either shared input values or shared outputs of previous instructions. Namely, the `XOR` instructions are interpreted as sharewise addition gadgets of fan-in 2, the `NOT` instructions as sharewise addition gadgets of fan-in 1 with the constant shared input $(1, 0, \ldots, 0)$, the `AND` instructions as ISW-multiplication gadgets of fan-in 2, and the `REFRESH` instructions as ISW-refresh gadgets of fan-in 1. As an application, we experimented our tool on several standard shared circuits. First, we analyzed the t-probing security of the small examples of Section **??** as a sanity check. Then, we investigated the t-probing security of the AES s-box circuit from [**JC:BoyMatPer13**] and compared the result with what the `maskComp` tool produces. Additionally, we studied the impact of our tool to practical implementations (for both the randomness usage and the performance implications).

4.7.1 Application to Section ?? Examples

In order to have some sanity checks of our new method on simple standard shared circuits, we applied our tool to the examples given in Section **??**, namely the standard shared circuits depicted in Figure **??** and Figure **??**. Specifically, we first translated the two standard shared circuits into a list of instructions that is given to our tool. For each circuit, the first instruction gives the number of shared inputs. Then, each of the following instruction matches one of the four possible operations among `XOR`, `AND`, `NOT`, and `REFRESH` together with the indices of the corresponding one or two operands. The output of each such operation is then represented by the first unused index. At the end, from the generated list of instructions the tool derives a list of pairs of operands, namely the inputs to the multiplications in the circuit. Finally, Algorithm **??** is evaluated on the obtained list of operands.

The first example is based on a standard shared circuit that takes 2 shared inputs and then performs two operations, namely a sharewise addition (`XOR`) and an ISW-multiplication (`AND`). The `AND` instruction takes two inputs, namely the output of the `XOR` and one of the two inputs of the circuit, which means that there is only two possible target vectors for an attack to be mounted. They are displayed in the list `list_comb`. For both these two vectors successively displayed with variable `comb`, the tool generates their respective sets $\mathcal{G}_1$ and $\mathcal{O}_1$, as defined in Section **??**. Then since $\mathcal{G}_2$ is equal to $\mathcal{G}_1$ for both vectors, the tool outputs

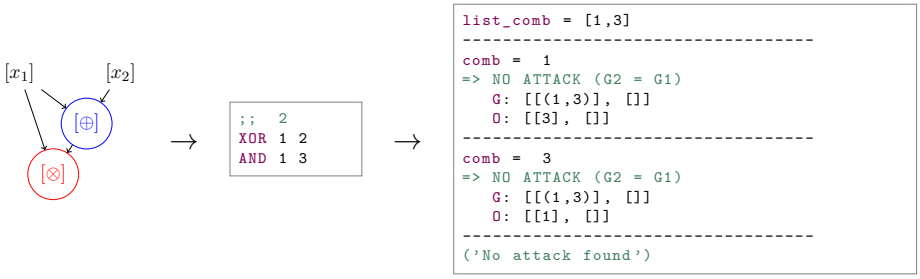


Figure 4.12: New method applied on example 1.

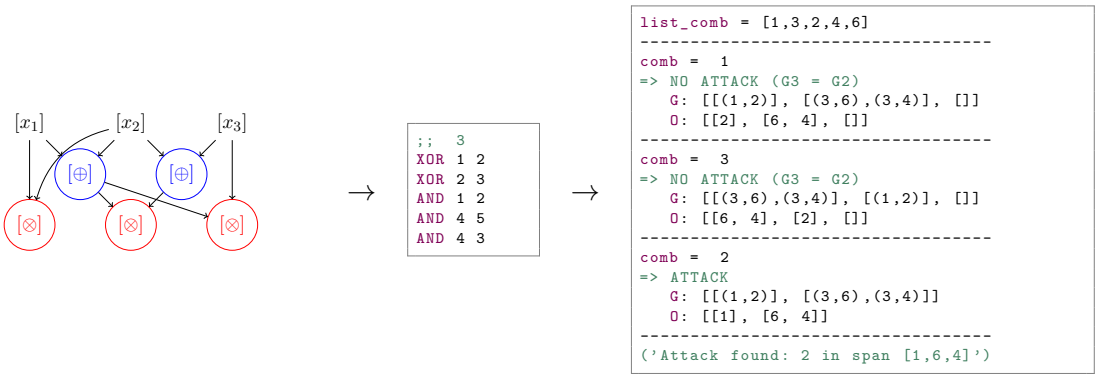


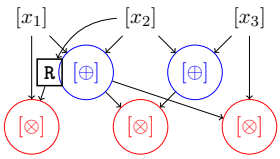
Figure 4.13: New method applied on example 2.

that no attack could be found. The circuit is thus t-probing secure. The complete process is described in Figure **??**.

The second example is based on a standard shared circuit that takes 3 shared inputs and then performs 5 operations, namely 2 sharewise additions (`XOR`) and 3 ISW-multiplications (`AND`). The three `AND` instructions take five distinct inputs, which means that there are five possible target vectors for an attack to be mounted. For the two first target vectors, no attack could be found as the tool expressed all the multiplications in the circuit with two sets $\mathcal{G}_1$ and $\mathcal{G}_2$ without finding any attack. For the third target vector, after the construction of $\mathcal{G}_2$ an attack was found as the target vector belonged to the span of the set $\mathcal{O}_2$. The complete process is described in Figure **??**. Moreover, we verified that by adding a refresh gadget on the operand on which our tool find an attack prior to the multiplication where it is used, the tool is not able any more to find an attack on the new circuit for this example. The results can be find in Figure **??**.

4.7.2 Application to AES s-box

At Eurocrypt 2017, Goudarzi and Rivain [**EC:GouRiv17**] proposed an efficient software implementation of the s-box of the AES for higher-order masking. Based on the Boolean circuit of Boyer *et al.* [**JC:BoyMatPer13**], their implementation evaluates the s-box on a state under bitslice representation with only 32 AND gates. In order to be t-probing secure without doubling the number of shares in the encoding of sensitive variables, a conservative



→

```
;;  3
XOR 1 2
XOR 2 3
REF 2
AND 1 6
AND 4 5
AND 4 3
```

→

```
-----------------------------------
list_comb =  [3, 8, 6, 1, 4]
-----------------------------------
comb =  3
=> NO ATTACK (G3 = G2)
   G: [[(3,6),(3,4)], [(1,8)], []]
   O: [[6, 4], [8], []]
-----------------------------------
comb =  8
=> NO ATTACK (G2 = G1)
   G: [[(1,8)], []]
   O: [[1], []]
-----------------------------------
comb =  6
=> NO ATTACK (G2 = G1)
   G: [[(3,6)], []]
   O: [[3], []]
-----------------------------------
comb =  1
=> NO ATTACK (G2 = G1)
   G: [[(1,8)], []]
   O: [[8], []]
-----------------------------------
comb =  4
=> NO ATTACK (G2 = G1)
   G: [[(3,4)], []]
   O: [[3], []]
-----------------------------------
('No attack found')
```

Figure 4.14: New method applied on example 2 augmented with a refresh.

choice was made to add a refresh gadget prior to each multiplication. As explained in Section **??**, a major drawback of such conservative approach is the performance overhead induced by the number of calls to refresh gadgets due to the randomness usage.

In order to obtain efficient implementations of the AES s-box and to be tight on the number of randomness requirement, we have applied our tool to the circuit of the s-box reordered by Goudarzi and Rivain without any refreshing gadget. Interestingly, we obtained that no attack can be found for any masking order. More precisely, the tool first identified 36 distinct target vectors out of the 64 possible operands of multiplication gadgets (it can be easily checked on the circuit found in Section 6 of [**EC:GouRiv17**]). For each of the 36 target vectors, the corresponding set $\mathcal{G}_1$ is constructed. Then, for every variable the algorithm stops as the respective sets $\mathcal{G}_2$ are always equal to the respective sets $\mathcal{G}_1$. The complete report of the tool results can be found in Table **??**. In the first and third columns of Table **??**, the expressions of the target vectors as linear combinations of input variables or multiplications input are given and in the second and fourth columns, the corresponding sets $\mathcal{G}_1$ are displayed, all in hexadecimal form.

To prove the security of the AES s-box circuit, our tool took only 427 ms. This speed is mainly due to the fact that for each possible target variable, only the set $\mathcal{G}_1$ is computed. For comparison, we looked at the time taken by the `maskVerif` tool of [**EC:BBDFGS15**]. For a masking order $t = 2$, `maskVerif` found no attack in 35.9 sec and for $t = 3$ in approximately 10 hours.

For the sake of comparison, we also applied the `maskComp` tool on the same circuit. We obtained that `maskComp` adds refresh gadgets prior to each multiplication in the circuit, transforming it into a new t-NI secure circuit. Since our tool has shown that the circuit is t-probing secure with no refresh gadgets, adding those refresh gadgets implies an overhead in the t-probing security that can lead to less efficient practical implementations. As an

Table 4.1: Results for AES s-box circuit.

Target	$\mathcal{G}_1$	Target	$\mathcal{G}_1$
8E	{(8E, 80), (96875, 8E)}	C6	{(C6, 86), (418605, C6)}
72	{(9, 72), (C2D0B, 72)}	29B040	{(29B040, D9), (29B040, E7)}
3457E	{(3457E, 1B040)}	21	{(21, 5F), (683645, 21)}
16875	{(16875, A0000)}	96875	{(96875, 8E), (96875, 80)}
C37B	{(C37B, D835)}	44C37B	{(44C37B, 41), (44C37B, 74)}
18605	{(18605, 36875)}	236875	{(5457E, 236875)}
D9	{(E7, D9), (29B040, D9)}	5F	{(21, 5F), (683645, 5F)}
683645	{(683645, 5F), (683645, 21)}	5457E	{(5457E, 87), (5457E, 236875), (5457E, F2)}
E7	{(E7, D9), (29B040, E7)}	86	{(C6, 86), (418605, 86)}
C2D0B	{(C2D0B, 72), (C2D0B, 9)}	418605	{(418605, 86), (418605, C6)}
74	{(41, 74), (44C37B, 74)}	D835	{(C37B, D835)}
A0000	{(16875, A0000)}	641B4E	{(641B4E, 2D), (641B4E, 28)}
20D835	{(20D835, 59), (20D835, 69)}	28	{(28, 2D), (641B4E, 28)}
F2	{(87, F2), (5457E, F2)}	87	{(87, F2), (5457E, 87)}
69	{(69, 59), (20D835, 69)}	1B040	{(3457E, 1B040)}
9	{(9, 72), (C2D0B, 9)}	59	{(69, 59), (20D835, 59)}
2D	{(28, 2D), (641B4E, 2D)}	80	{(8E, 80), (96875, 80)}
41	{(41, 74), (44C37B, 41)}	36875	{(18605, 36875)}

illustration, we have implemented a bitslice version of the AES s-box circuit for a generic masking order to see the impact in performances between a full refresh approach (*i.e.* the conservative choice of Goudarzi and Rivain and the result of `maskComp`) and a no refresh approach (our new tool). Each of this two approaches produces a circuit that is at least t-probing secure for any masking order t and that is securely composable with other circuits (since `maskComp` produce a t-NI circuit and from the result of Section **??**. To be consistent with the state of the art, the randomness in our implementations can be obtained from a TRNG with two different settings: a first setting with a *free* TRNG that outputs 32-bit of fresh randomness every 10 clock cycles (as in [**EC:GouRiv17**]) and a second setting with a constrained TRNG that outputs 32-bit of fresh randomness every 80 clock cycles (as in [**CHES:JouSta17**]). The performance results can be found in Table **??**. For both approaches, the number of refresh gadgets used and the number of randomness needed are displayed. Then, the timing in clock cycles for both settings are shown. We can see that our tool allows to divide by 2 the number of required randomness and benefits from an asymptotic gain of up to 43% in speed. The comparison of the timings for several masking orders are depicted in Figure **??**.

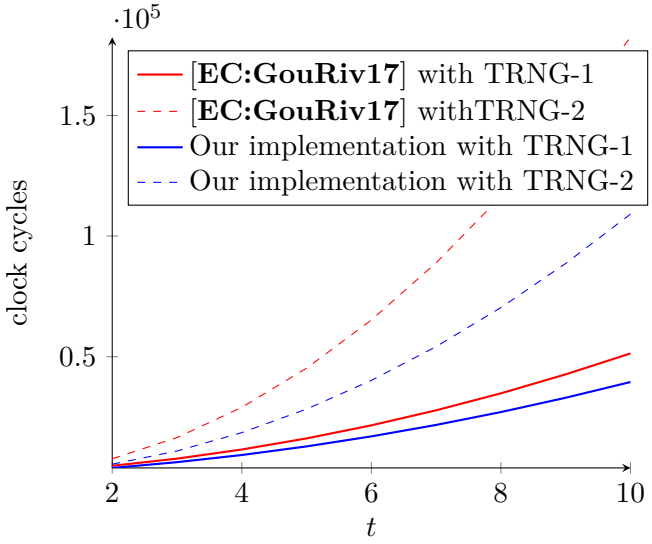


Figure 4.15: Timings of a t-probing secure AES s-box implementation.

	Nb. of Refresh	Nb. of Random	Timing (Set. 1)	Timing (Set. 2)
[**EC:GouRiv17**]	32	$32\,t(t-1)$	$408\,t^2 + 928\,t + 1262$	$1864\,t^2 - 528\,t + 1262$
this thesis	0	$16\,t(t-1)$	$295.5\,t^2 + 905.5\,t + 872$	$1069\,t^2 + 132\,t + 872$

Table 4.2: Performance results of the implementation AES s-box depending on the number of refresh gadgets.

Chapter 5

Implementation

5.1 Introduction

In this chapter, we present a case study on ARM (v7) architectures, which are today the most widespread in embedded systems (privileged targets of side-channel attacks). We provide an extensive and fair comparison between the different methods of the state of the art and the ones proposed in Chapter **??**, and a benchmarking on optimized implementations of higher-order masked blockciphers. For this purpose, we follow a bottom-up approach and start by investigating the efficient implementation of the base-field multiplication, which is the core elementary operation of the ISW-based masking schemes. We propose several implementations strategies leading to different time-memory trade-offs. We then investigate the main building blocks of existing masking schemes, namely the ISW (and its variants) and CPRR schemes. We optimize the implementation of these schemes and we describe parallelized versions that achieve significant gains in performances. From these results, we propose fine-tuned variants of our generic decomposition methods for three cases (full field case, median case and Boolean case) and the algebraic decomposition methods, which allows us to compare them in a practical and optimized implementation context. We also investigate efficient polynomial methods for the specific s-boxes of two important blockciphers, namely AES and PRESENT.

As an additional contribution, we put forward an alternative strategy to polynomial methods which consists in applying bitslicing at the s-box level. More precisely, the s-box computations within a block cipher round are bitsliced so that the core nonlinear operation is not a field multiplication anymore (nor a quadratic polynomial) but a bitwise logical AND between two m-bit registers (where m is the number of s-box computations). This allows us to translate compact hardware implementations of the AES and PRESENT s-boxes into efficient masked implementations in software. This approach has been previously used to design blockciphers well suited for masking [**FSE:GLSV14**] but, to the best of our knowledge, has never been used to derive efficient higher-order masked implementations of existing standard blockciphers such as AES or PRESENT. We further provide implementation results for full blockciphers and discuss the security aspects of our implementations.

Our results clearly demonstrate the superiority of the bitslicing approach (at least on 32-bit ARM architectures). Our masked bitslice implementations of AES and PRESENT are significantly faster than state-of-the-art polynomial methods with fine-tuned low-level implementations. In particular, an encryption masked at the order 10 only takes a few milliseconds with a 60 MHz clock frequency (specifically 8ms for AES and 5ms for PRESENT).

Each implementation presented in this chapter can be found on GitHub for public usage at the following link [**github**].

5.2 Approach

In this chapter, we describe the algorithmic and implementation tricks used to produce efficient implementation of protected block-ciphers with higher-order masking. All the implementation were made in ARM assembly and can be found on GitHub. To do so, we proceed with a bottom-up approach. First, we described the different building blocks that are used in the implementation of a block-ciphers, from the field multiplication to secure non-linear operations. Then, we give implementation performances for the evaluation of secure generic s-boxes following the results of Chapter **??**. We also provide implementation results for

structured s-boxes such as the one of the AES and the one of PRESENT. Finally, we describe and give implementation results for the two block-ciphers AES and PRESENT, using the different s-boxes implemented. Thank to the result of Chapter **??**, the implementation of the AES can be done with minimal randomness consumption, as no refreshes of the sharings are needed.

5.3 Building Blocks

5.3.1 Field Multiplications

In this section, we focus on the efficient implementation of the multiplication over $\mathbb{F}_{2^\lambda}$ where λ is small (typically $\lambda \in [\![4, 10]\!]$). The fastest method consists in using a precomputed table mapping the $2^{2\lambda}$ possible pairs of operands (a, b) to the output product $a \cdot b$. The size of this table is given in Table **??** with respect to λ.

In the context of embedded systems, one is usually constrained on the code size and spending several kilobytes for (one table in) a cryptographic library might be prohibitive. That is why we investigate hereafter several alternative solutions with different time-memory trade-offs. Specifically, we look at the classical binary algorithm and exp-log multiplication methods. We also describe a tabulated version of Karatsuba multiplication, and another table-based method: the *half-table multiplication*. The obtained implementations are compared in terms of clock cycles, register usage, and code size (where the latter is mainly impacted by precomputed tables).

In the rest of this section, the two multiplication operands in $\mathbb{F}_{2^\lambda}$ will be denoted a and b. These elements can be seen as polynomials $a(x) = \sum_{i=0}^{\lambda-1} a_i x^i$ and $b(x) = \sum_{i=0}^{\lambda-1} b_i x^i$ over $\mathbb{F}_2[x]/p(x)$ where the a_i's and the b_i's are binary coefficients and where p is a degree-λ irreducible polynomial in $\mathbb{F}_2[x]$. In our implementations, these polynomials are simply represented as λ-bit strings $a = (a_{\lambda-1}, \ldots, a_0)_2$ or equivalently $a = \sum_{i=0}^{\lambda-1} a_i \, 2^i$ (and similarly for b).

5.3.1.1 Binary Multiplication

The binary multiplication algorithm is the most basic way to perform a multiplication on a binary field. It consists in evaluating the following formula:

$$a(x) \cdot b(x) = \big(\cdots\big((b_{\lambda-1}a(x)x + b_{\lambda-2}a(x))x + b_{\lambda-3}a(x)\big)\cdots\big)x + b_0 a(x) \ , \qquad (5.1)$$

by iterating over the bits of b. A formal description is given in Algorithm **??**.

Table 5.1: Size of the full multiplication table (in kilobytes) w.r.t. λ.

$\lambda =$	4	5	6	7	8	9	10
Table size	0.25 KB	1 KB	4 KB	16 KB	64 KB	512 KB	1048 KB

Algorithm 4 Binary multiplication algorithm

Input: $a(x), b(x) \in \mathbb{F}_2[x]/p(x)$
Output: $a(x) \cdot b(x) \in \mathbb{F}_2[x]/p(x)$
1: $r(x) \leftarrow 0$
2: **for** $i = \lambda - 1$ **down to** 0 **do**
3: $\quad r(x) \leftarrow x \cdot r(x) \bmod p(x)$
4: $\quad$ **if** $b_i = 1$ **then** $r(x) \leftarrow r(x) + a(x)$
return $r(x) \bmod p(x)$

The reduction modulo $p(x)$ can be done either inside the loop (at Step 3 in each iteration) or at the end of the loop (at Step 6). If the reduction is done inside the loop, the degree of $x \cdot r(x)$ is at most λ in each iteration. So we have

$$x \cdot r(x) \bmod p(x) = \begin{cases} x \cdot r(x) - p(x) & \text{if } r_{\lambda-1} = 1 \\ x \cdot r(x) & \text{otherwise} \end{cases} \tag{5.2}$$

The reduction then consists in subtracting $p(x)$ from $x \cdot r(x)$ if and only if $r_{\lambda-1} = 1$ and doing nothing otherwise. In practice, the multiplication by x simply consists in left-shifting the bits of r and the subtraction of p is a simple XOR. The tricky part is to conditionally perform the latter XOR with respect to the bit $r_{\lambda-1}$ as we aim for a branch-free code. This is achieved using the *arithmetic right shift*[1] instruction (sometimes called signed shift) to compute $(r \ll 1) \oplus (r_{\lambda-1} \times p)$ by putting $r_{\lambda-1}$ at the sign bit position, which can be done in 3 ARM instructions (3 clock cycles) as follows:

```
LSL $tmp, $res, #(32-n)          ;; tmp = r_{n-1}
AND $tmp, $mod, $tmp, ASR #32    ;; tmp = p & (tmp ASR 32)
EOR $res, $tmp, $res, LSL #1     ;; r = (r_{n-1} * p)^(r << 1)
```

Step 4 consists in conditionally adding a to r whenever b_i equals 1. Namely, we have to compute $r \oplus (b_i \cdot a)$. In order to multiply a by b_i, we use the rotation instruction to put b_i in the sign bit and the arithmetic shift instruction to fill a register with b_i. The latter register is then used to mask a with a bitwise AND instruction. The overall Step 4 is performed in 3 ARM instructions (3 clock cycles) as follows:

```
ROR $opB, #31                    ;; b_i = sign(opB)
AND $tmp, $opA, #opB, ASR #32    ;; tmp = a & (tmp ASR 32)
EOR $res, $tmp                   ;; r = r^(a * b_i)
```

Variant. If the reduction is done at the end of the loop, Step 3 becomes a simple left shift, which can be done together with Step 4 in 3 instructions (3 clock cycles) as follows:

```
ROR $opB, #31                    ;; b_i = sign(opB)
AND $tmp, $opA, $opB, ASR #32    ;; tmp = a & (tmp ASR 32)
EOR $res, $tmp, $res, LSL #1     ;; r = (a * b_i)^(r << 1)
```

[1] This instruction performs a logical right-shift but instead of filling the vacant bits with 0, it fills these bits with the leftmost bit operand (*i.e.* the sign bit).

The reduction must then be done at the end of the loop (Step 6), where we have $r(x) = a(x) \cdot b(x)$ which can be of degree up to $2\lambda - 2$. Let r_h and r_ℓ be the polynomials of degree at most $\lambda - 2$ and $\lambda - 1$ such that $r(x) = r_h(x) \cdot x^\lambda + r_\ell(x)$. Since we have $r(x) \bmod p(x) = (r_h(x) \cdot x^\lambda \bmod p(x)) + r_\ell(x)$, we only need to reduce the high-degree part $r_h(x) \cdot x^\lambda$. This can be done by tabulating the function mapping the $\lambda - 1$ coefficients of $r_h(x)$ to the $\lambda - 2$ coefficients of $r_h(x) \cdot x^\lambda \bmod p(x)$. The overall final reduction then simply consists in computing $T[r \gg \lambda] \oplus (r \cdot (2^\lambda - 1))$, where T is the corresponding precomputed table.

5.3.1.2 Exp-Log Multiplication

Let $g \in \mathbb{F}_{2^\lambda}$ be a generator of the multiplicative group $\mathbb{F}^*_{2^\lambda}$. We shall denote by $\exp_g$ the exponential function defined over $[\![0, 2^\lambda - 1]\!]$ as $\exp_g(\ell) = g^\ell$, and by $\log_g$ the discrete logarithm function defined over $\mathbb{F}^*_{2^\lambda}$ as $\log_g = \exp_g^{-1}$. Assume that these functions can be tabulated (which is usually the case for small values of λ). The multiplication between field elements a and b can then be efficiently computed as

$$a \cdot b = \begin{cases} \exp_g(\log_g(a) + \log_g(b) \bmod 2^\lambda - 1) & \text{if } a \neq 0 \text{ and } b \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{5.3}$$

Le us denote $t = \log_g(a) + \log_g(b)$. We have $t \in [\![0, 2^{\lambda+1} - 2]\!]$ giving

$$t \bmod 2^\lambda - 1 = \begin{cases} t - 2^\lambda + 1 & \text{if } t_\lambda = 1 \\ t & \text{otherwise} \end{cases} \tag{5.4}$$

where t_λ is the most significant bit in the binary expansion $t = \sum_{i=0}^{\lambda} t_i \, 2^i$, which can be rewritten as $t \bmod 2^\lambda - 1 = (t + t_\lambda) \cdot (2^\lambda - 1)$. This equation can be evaluated with 2 ARM instructions[2] (2 clock cycles) as follows:

```
ADD $tmp, $tmp, LSR #n          ;;tmp = tmp + tmp>>n
AND $tmp, #(2^n-1)              ;;tmp = tmp & (2^n-1)
```

Variant. Here again, a time-memory trade-off is possible: the $\exp_g$ table can be doubled in order to handle a $(\lambda + 1)$-bit input and to perform the reduction. This simply amounts to considering that $\exp_g$ is defined over $[\![0, 2^{\lambda+1} - 2]\!]$ rather than over $[\![0, 2^\lambda - 1]\!]$.

Zero-testing. The most tricky part of the exp-log multiplication is to manage the case where a or b equals 0 while avoiding any conditional branch. Once again we can use the arithmetic right-shift instruction to propagate the sign bit and use it as a mask. The test for zero can then be done with 4 ARM instructions (4 clock cycles) as follows:

```
RSB $tmp, $opA, #0              ;; tmp = 0 - a
AND $tmp, $opB, $tmp, ASR #32   ;; tmp = b & (tmp ASR 32)
RSB $tmp, #0                    ;; tmp = 0 - tmp
AND $res, $tmp, ASR #32         ;; r = r & (tmp ASR 32)
```

[2]Note that for $\lambda > 8$, the constant $2^\lambda - 1$ does not lie in the range of constants enabled by ARM (*i.e.* rotated 8-bit values). In that case, one can use the BIC instruction to perform a logical AND where the second argument is complemented. The constant to be used is then 2^λ which well belongs to ARM constants whatever the value of λ.

5.3.1.3 Karatsuba Multiplication

The Karatsuba method is based on the following equation:

$$a \cdot b = (a_h + a_\ell)(b_h + b_\ell)\, x^{\frac{\lambda}{2}} + a_h\, b_h\, (x^n + x^{\frac{\lambda}{2}}) + a_\ell\, b_\ell\, (x^{\frac{\lambda}{2}} + 1) \bmod p(x) \tag{5.5}$$

where a_h, a_ℓ, b_h, b_ℓ are the polynomials with degree at most $\lfloor \frac{\lambda}{2} \rfloor - 1$ such that $a(x) = a_h\, x^{\frac{\lambda}{2}} + a_\ell$ and $b(x) = b_h\, x^{\frac{\lambda}{2}} + b_\ell$. The above equation can be efficiently evaluated by tabulating the following functions:

$$\begin{aligned} (a_h + a_\ell, b_h + b_\ell) &\mapsto (a_h + a_\ell)(b_h + b_\ell)\, x^{\frac{\lambda}{2}} \bmod p(x)\ , \\ (a_h, b_h) &\mapsto a_h\, b_h\, (x^\lambda + x^{\frac{\lambda}{2}}) \bmod p(x)\ , \\ (a_\ell, b_\ell) &\mapsto a_\ell\, b_\ell\, (x^{\frac{\lambda}{2}} + 1) \bmod p(x)\ . \end{aligned}$$

We hence obtain a way to compute the multiplication with 3 look-ups and a few XORs based on 3 tables of 2^λ elements.

In practice, the most tricky part is to get the three pairs $(a_h||b_h)$, $(a_\ell||b_\ell)$ and $(a_h+a_\ell||b_h+b_\ell)$ to index the table with the least instructions possible. The last pair is a simple addition of the two first ones. The computation of the two first pairs from the operands $a \equiv (a_h||a_\ell)$ and $b \equiv (b_h||b_\ell)$ can then be seen as the transposition of a 2×2 matrix. This can be done with 4 ARM instructions (4 clock cycles) as follows:

```
EOR $tmp0, $opA, $opB, LSR #(n/2)   ;; tmp0 = [a_h|a_l^b_h]
EOR $tmp1, $opB, $tmp0, LSL #(n/2)  ;; tmp1 = [a_h|a_l|b_l]
BIC $tmp1, #(2^n*(2^(n/2)-1))       ;; tmp1 = [a_l|b_l]
EOR $tmp0, $tmp1, LSR #(n/2)        ;; tmp0 = [a_h|b_h]
```

5.3.1.4 Half-Table Multiplication

The half-table multiplication can be seen as a trade-off between the Karatsuba method and the full-table method. While Karatsuba involves 3 look-ups in three 2^λ-sized tables and the full-table method involves 1 look-up in a $2^{2\lambda}$-sized table, the half-table method involves 2 look-ups in two $2^{\frac{3\lambda}{2}}$-sized tables. It is based on the following equation:

$$a \cdot b = b_h\, x^{\frac{\lambda}{2}}\, (a_h\, x^{\frac{\lambda}{2}} + a_\ell) + b_\ell\, (a_h\, x^{\frac{\lambda}{2}} + a_\ell) \bmod p(x)\ , \tag{5.6}$$

which can be efficiently evaluated by tabulating the functions:

$$\begin{aligned} (a_h, a_\ell, b_h) &\mapsto b_h\, x^{\frac{\lambda}{2}}\, (a_h\, x^{\frac{\lambda}{2}} + a_\ell) \bmod p(x)\ , \\ (a_h, a_\ell, b_\ell) &\mapsto b_\ell\, (a_h\, x^{\frac{\lambda}{2}} + a_\ell) \bmod p(x)\ . \end{aligned}$$

Once again, the barrel shifter is useful to get the input triplets efficiently. Each look-up can be done with two ARM instructions (for a total of 8 clock cycles) as follows:

```
EOR  $tmp,$opB,$opA,LSL#n             ;;tmp=[a_h|a_l|b_h|b_l]
LDRB $res,[$tab1,$tmp,LSR#(n/2)       ;;res=T1[a_h|a_l|b_h]
EOR  $tmp,$opA,$opB,LSL#(32-n/2)      ;;tmp=[b_l|0..|a_h|a_l]
LDRB $tmp,[$tab2,$tmp,ROR#(32-n/2)]   ;;tmp=T2[a_h|a_l|b_l]
```

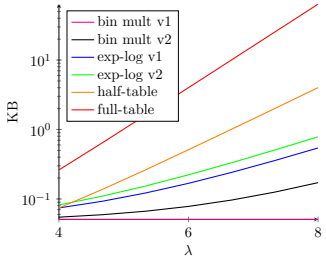
Table 5.2: Multiplication performances.

	bin mult v1	bin mult v2	exp-log v1	exp-log v2	kara.	half-tab	full-tab
clock cycles ($\lambda \leq 8$)	$10\lambda + 3$ (+3)	$7\lambda + 3$ (+3)	18 (+2)	16 (+2)	19 (+2)	10 (+3)	4 (+3)
clock cycles ($\lambda > 8$)	$10\lambda + 4$ (+3)	$7\lambda + 15$ (+3)	35 (+2)	31 (+2)	38 (+2)	n/a	n/a
registers	5	5	5 (+1)	5 (+1)	6 (+1)	5 (+1)	5
code size ($\lambda \leq 8$)	52	$2^{\lambda-1} + 48$	$2^{\lambda+1} + 48$	$3 \cdot 2^{\lambda} + 40$	$3 \cdot 2^{\lambda} + 42$	$2^{\frac{3\lambda}{2}+1} + 24$	$2^{2\lambda} + 12$

5.3.1.5 Performance

The obtained performance is summarized in Table **??** in terms of clock cycles, register usage, and code size. For clock cycles, the number in brackets indicates instructions that need to be done only once when multiple calls to the multiplication are performed (as in the secure multiplication procedure described in the next section). These are initialization instructions such as loading a table address into a register. For $\lambda > 8$, elements take two bytes to be stored (assuming $\lambda \leq 16$) which implies an overhead in clock cycles and a doubling of the table size. For most methods, the clock cycles and register usage are constant w.r.t. $\lambda \geq 8$, whereas the code size depends on λ. For the sake of illustration, we therefore additionally display the code size (and corresponding LUT sizes) in Figure **??** for several values of λ.

Remark 5.3.1. *For $\lambda > 8$, elements take two bytes to be stored (assuming $\lambda \leq 16$). Two options are then possible for look-up tables: one can either store each λ-bit element on a full 32-bit word, or store two λ-bit elements per 32-bit word (one per half-word). The former option has a strong impact on the LUT sizes, and hence on the code size, which is already expected to be important when $\lambda > 8$. Therefore we considered the latter option, which has an impact on performances since we either have to load two bytes successively, or to load one 32-bit word and select the good half-word (which is actually costlier than two loads).*



λ	4	6	8	10
Binary v1	0	0	0	0
Binary v2	8 B	32 B	128 B	1 KB
Exp-log v1	32 B	128 B	0.5 KB	4 KB
Exp-log v2	48 B	192 B	0.75 KB	6 KB
Karatsuba	48 B	192 B	0.75 KB	6 KB
Half-table	0.13 KB	1 KB	8 KB	128 KB
Full-table	0.25 KB	4 KB	64 KB	2048 KB

Figure 5.1: Full code size (left graph) and LUT size (right table) w.r.t. λ.

We observe that all the methods provide different time-memory trade-offs except for Karatsuba which is beaten by the exp-log method (v1) both in terms of clock cycles and code size. The latter method shall then always be preferred to the former (at least on our architecture). As expected, the full-table method is by far the fastest way to compute a field multiplication, followed by the half-table method. However, depending on the value of λ, these methods might be too consuming in terms of code size due to their large precomputed tables. On the other hand, the binary multiplication (even the improved version) has very poor performance in terms of clock cycles and it should only be used for extreme cases where the code size is very constrained. We consider that the exp-log method v2 (*i.e.* with doubled

exp-table) is a good compromise between code size an speed whenever the full-table and half-table methods are not affordable (which might be the case for *e.g.* $\lambda \geq 8$). In the following, we shall therefore focus our study on secure implementations using the exp-log (v2), half-table or full-table method for the base field multiplication.

5.3.2 Secure Non Linear Operations

In this section we describe optimized low-level implementations of the four following secure multiplications and one secure evaluation of quadratic functions:

ISW (Ishai-Sahai-Wagner, Crypto'03): probing secure multiplication,
BDF$^+$ (Barthe *et al.* , Eurocrypt'17): bounded-moment secure multiplication,
BBP$^+$ (Belaïd *et al.* , Eurocrypt'16): ISW gadget with randomness saving,
BCPZ (Battistello *et al.* , CHES'16): ISW gadget with additional refreshing,
CPRR (Carlet *et al.* , CRYPTO'18): secure quadratic evaluation.

Each secure operation is described at the algorithmic level and at the implementation level (with possible implementation tricks). For the ISW-like multiplications and the CPRR evaluation, we also provide a parallelized version of the code Then, we provided some implementation results (in both clock cycles and code size) and compare the different schemes in terms of performances, randomness consumption, and security guarantees.

Remark 5.3.2. *For the rest of this section, the field multiplication between two elements a and b over $\mathbb{F}_{2^\lambda}$, for any λ, is denoted by $a \cdot b$.*

5.3.2.1 ISW: the Standard Probing-Secure Multiplication

Let us recall the ISW multiplication scheme introduced in Chapter **??**.

Algorithm 5 ISW (Ishai-Sahai-Wagner, Crypto'03)

Input: sharings $(a_1, \ldots, a_d) \in \{0,1\}^{32\times d}$ and $(b_1, \ldots, b_d) \in \{0,1\}^{32\times d}$
Output: sharing $(c_1, \ldots, c_d) \in \{0,1\}^{32\times d}$ such that $\bigoplus_i c_i = (\bigoplus_i a_i) \cdot (\bigoplus_j b_j)$

1: **for** $i = 1$ to d **do**
2: $\quad c_i \leftarrow a_i \cdot b_i$
3: **for** $i = 1$ to d **do**
4: $\quad$ **for** $j = i + 1$ to d **do**
5: $\quad\quad s \leftarrow \{0,1\}^{32}$
6: $\quad\quad s' \leftarrow (s \oplus (a_i \cdot b_j)) \oplus (a_j \cdot b_i)$
7: $\quad\quad c_i \leftarrow c_i \oplus s$
8: $\quad\quad c_j \leftarrow c_j \oplus s'$
return $(c_1, \ldots, c_d)$

From two sharings $(a_1, \ldots, a_d)$ and $(b_1, \ldots, b_d)$, the ISW multiplication simply computes all the d^2 crossed products $a_i \cdot b_j$ which are then summed in d new shares c_i with new random elements $r_{i,j}$. Each new random element is involved twice in the new shares implying $\bigoplus_i c_i = \bigoplus_{i,j} a_i \cdot b_j = (\bigoplus_i a_i) \cdot (\bigoplus_j b_j)$.

From the implementation viewpoint, we use the approach suggested in [**EC:Coron14**] that directly accumulates each intermediate result $r_{i,j}$ in the output share c_i so that the memory cost is $O(d)$ instead of $O(d^2)$ when the $r_{i,j}$'s are stored. In order to push forward

Table 5.3: Implementation results for the ISW multiplication over $\mathbb{F}_2$

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
Straight ISW	75	291	1155	4611	18435	164	164	164	164	164	10	$d(d-1)/2$
Unrolled ISW	58	231	949	3876	15682	132	464	1848	7500	30324	8	$d(d-1)/2$

the optimization, we also propose a version of the code where the nested loops are unrolled for specific values of d, namely when d is a power of 2. The performance of our low-level implementations is summarized in Table **??**. We observe that unrolling the loops allows us to save 15% to 23% clock cycles with an overhead factor from 3 to 200 times the code size. The only case where the unrolling fully benefits in both time and memory is for $d = 2$.

5.3.2.2 BDF$^+$: a Bounded-Moment Secure Multiplication

At Eurocrypt 2017, Barthe *et al.* introduced a new way to compute a secure multiplication specifically tailored for the bitwise context (*i.e.* for bitsliced implementations) [**EC:BDFGSS17**]. Their scheme handles registers holding all the shares of a given bit whereas in traditional ISW-based scheme, the shares of a variable are stored in different registers for security reasons. Nevertheless, Barthe *et al.* show that their multiplication is secure in the relaxed bounded moment model, which is argued to be sound in practice.

Intuitively the BDF$^+$ multiplication can be decomposed into different steps: the loading of the input shares a and b; the computation of the partial products between a and b; the loading of fresh randomness r; and the compression phase where these partial products are XORed all together and separated by the fresh randomness.

Its implementation is especially efficient when the number of shares d is equal to the size of the registers in the target architecture. This has been shown in [**CHES:JouSta17**] for the case $d = 32$. However, a question left open in the latter work is the scenario where the number of shares mismatches the register size. This issue is addressed hereafter.

For this purpose, we generalize the BDF$^+$ algorithm to a scenario where d can be lower than the register size. We propose a parallel version of this algorithm in which several sharings are stored in a register (*e.g.* 4 sharings of order $d = 8$ in one 32-bit register) and we describe an efficient way to perform sharing-wise rotations to keep good performance in such a non-optimal scenario. The main restriction is that our generalization only works for masking orders that are a power of 2 (so that the sharing size divides the register size), including the case $d = 2$ which was not taken into account in the original publication. The optimized BDF$^+$ multiplication is described in Algorithm **??**.

Encoding. In order to make full use of the register when d is less than 32 (*i.e.* d is not equal to the architecture size), but d is a power of 2, we fill the input registers with $k = 32/d$ words of d shares. We thus process k secure multiplications in parallel. More specifically, let us denote $w_0, \ldots, w_{31}$ the bits of a 32-bit register $\boldsymbol{w}$ (from MSB to LSB). For $d = 16$, $\boldsymbol{w}$ encodes 2 secret bits z_0 and z_1 such that $\bigoplus_{i=0}^{15} w_i = z_0$ and $\bigoplus_{i=16}^{31} w_i = z_1$. For $d = 8$, $\boldsymbol{w}$ encodes 4 secret bits z_0, z_1, z_2 and z_3 such that $\bigoplus_{i=0}^{7} w_i = z_0$ and $\bigoplus_{i=8}^{15} w_i = z_1$ and $\bigoplus_{i=16}^{23} w_i = z_2$ and $\bigoplus_{i=24}^{31} w_i = z_3$, and so on.

Efficient sharing-wise rotation. Algorithm **??** can directly be applied on multi-sharing

Algorithm 6 BDF$^+$ (Barthe *et al.* , Eurocrypt'17)

Input: shares $a = (a_1, \cdots, a_d) \in \{0,1\}^{32}$, shares $b = (b_1, \cdots, b_d) \in \{0,1\}^{32}$
Output: shares $c = (c_1, \cdots, c_d) \in \{0,1\}^{32}$ corresponding to $a \cdot b$

1: $x_1 \leftarrow a \wedge b$
2: $r \leftarrow \{0,1\}^{32}$
3: $y_1 \leftarrow x_1 \oplus r$
4: **if** $d = 2$ **then**
5: $\quad x_2 \leftarrow a \wedge \mathsf{ROT}(b, 1)$
6: $\quad y_2 \leftarrow y_1 \oplus x_2$
7: $\quad c \leftarrow y_2 \oplus \mathsf{ROT}(r, 1)$
8: $\quad$ **for** $i = 1$ to $d/2 - 1$ **do**
9: $\quad\quad$ **if** $i \mod 2 = 0$ **then**
10: $\quad\quad\quad r \leftarrow \{0,1\}^{32}$
11: $\quad\quad x_{2i} \leftarrow a \wedge \mathsf{ROT}(b, i)$
12: $\quad\quad x_{2i+1} \leftarrow \mathsf{ROT}(a, i) \cdot b$
13: $\quad\quad y_{3i-1} \leftarrow y_{3i-2} \oplus x_{2i}$
14: $\quad\quad y_{3i} \leftarrow y_{3i-1} \oplus x_{2i+1}$
15: $\quad\quad y_{3i+1} \leftarrow y_{3i} \oplus \mathsf{ROT}(r, i \mod 2)$
16: $\quad x_d \leftarrow a \wedge \mathsf{ROT}(b, d/2)$
17: $\quad c \leftarrow y_{3\lfloor (d-1)/2 \rfloor + 1} \oplus x_d$
return c

input registers. The only operation which needs to be modified accordingly is the rotation $\mathsf{ROT}(w, i)$. We propose an efficient low-level implementation for such a sharing-wise rotation. Our method relies on the observation that applying an i-bit rotation to every d-bit chunk in a word w can be obtained by the following equation:

$$\mathsf{ROT}(w, i) = \left((w \ll i) \cdot \mathtt{mask}_{d,i}\right) \oplus \left((w \gg d - i) \cdot \overline{\mathtt{mask}}_{d,i}\right) \tag{5.7}$$

where $\mathtt{mask}_{d,i}$ is a selection mask defined as

$$\mathtt{mask}_{d,i} = \underbrace{\mathtt{11\ldots1}}_{d-i}\underbrace{\mathtt{00\ldots0}}_{i} \parallel \cdots \parallel \underbrace{\mathtt{11\ldots1}}_{d-i}\underbrace{\mathtt{00\ldots0}}_{i} \ ,$$

and $\overline{\mathtt{mask}}_{d,i}$ denotes its complement. From this equation we can directly compute the sharing-wise rotation. The main trick in the implementation is to efficiently deal with the generation of $\mathtt{mask}_{d,i}$ and the sharing-wise rotation.

The mask generation is decomposed into two steps. The first step allows to setup the mask correctly: $\mathtt{mask}_{d,0}$ is initialized with the value `0xFFFFFFFF`. We then need a `correction` value which will be used to update the mask correctly. `correction` is initialized with values given in Table **??**. Note that these operations are performed only once at the beginning of the multiplication. The second step will update the mask for the rotation according to the offset of the rotation given by the following formula:

$$\mathtt{mask}_{d,i} = \mathtt{mask}_{d,0} \oplus (\mathtt{correction} \ll i)$$

In practice, we only store $\mathtt{mask}_{d,0}$ and `correction` in two registers and we update them accordingly in each iteration of the loop. The cost of the update is 2 cycles.

Table 5.4: Possible values for correction

d	2	4	8	16
`correction`	0x5555555	0x11111111	0x01010101	0x00010001

Table 5.5: Performance results for BDF$^+$ (generic and unrolled)

	clock cycles					code size (bytes)					registers	random usage
d	2	4	8	16	32	2	4	8	16	32		
BDF$^+$ generic	n/a	77	146	285	n/a	n/a	248	244	240	n/a	13	$\lceil(d-1)/4\rceil$
BDF$^+$ unrolled	34	47	81	149	120	280	356	504	808	748	13	$\lceil(d-1)/4\rceil$

```
;;mask update
EOR     $mask, $mask, $correction
LSL     $correction, $correction, #1
```

Note that we make use of another register in order to store $\texttt{mask}_{d,1}$ (*i.e.* the rotation by 1) which is always needed to compute the rotations of the random values (instead of computing it again each time).

The rotation $\mathsf{ROT}(w, i)$ is then quite straightforward to implement as describes hereafter:

```
;; rotation of $w by $i
    AND     $tmp, $mask, $w, LSL $i
    LSR     $w, $w, $(d-i)
    BIC     $w, $w, $mask
    EOR     $w, $tmp, $w
```

Since the offsets of the shift lie in a register, we cannot benefit from the combined *barrel shifter*. Hence the overall cost of one rotation is 5 cycles.

In Table **??**, we report results of our implementation of the BDF$^+$ multiplication for d ranging from 2 to 32 for the generic version and an unrolled version (where the main advantage is to be able to hardcode the masks and values for the shifts). We observe that the unrolled version for $d = 32$ is faster and has less code size than for $d = 16$. This is easily explained by the fact that we can make full use of the *barrel shifter* in the case $d = 32$. Moreover, we observe that the unrolled version is 40% to 80% faster than the regular version. This is due to the fact that we can hardcode the masks, which makes the *barrel shifter* work again. The code size of the unrolled version ranges from 0.3 to 3 times the generic one. Note also that the code size of the generic version is decreasing as d grows because we compute the `correction` value iteratively (*i.e.* it needs $\log(32/d)$ iterations).

5.3.2.3 BBP$^+$: Towards Optimal Randomness Consumption

Belaïd *et al.* at Eurocrypt 2016 [**EC:BBPPTV16**] tackled the problem of minimizing the amount of randomness required in a secure multiplication. They described a generic algorithm which makes use of less randomness than ISW, reducing the former randomness requirement from $\frac{d(d-1)}{2}$ to $\frac{d^2}{4} + d$. As opposed to the ISW multiplication (which achieves $(d-1)$-SNI security), this algorithm is only proven $(d-1)$-NI secure. The original description of this

secure multiplication (see [**EC:BBPPTV16**]) is generic for any masking order $d \geq 4$ (specific algorithms for the case where $d = 2$ and 3 are given in their paper). However, it makes use of several conditional branches to process additional operations depending on the parity of the order d and/or of the loop index i.

We rewrote the algorithm such that all the conditional branches are removed, without affecting the correctness (see Algorithm **??**). These changes lead to several improvements in practice: first replacing if/ statement with loops allows avoiding several conditional branches treatment that are quit expensive in ARM assembly. Moreover, by rewriting the algorithm in such a way, we can compute all the randomness on-the-fly and avoid multiple load and store instructions for the correction step. Such improvements come at the cost of a less generic algorithm (it only works for even orders d). For the sake of comparison, we have implemented both algorithms to show the performance gained in clock cycles and code size (see Table **??**). We can see that our improvements allow a gain in timing ranging from 18% to 20% with an overhead of only 80 bytes of memory. Furthermore, we also unrolled the nested loops in order to get better results in timings. The timing gain ranges from 17% to 60% with an overhead factor between 3.5 and 50 for the code size for $d \geq 8$ only. For smaller values of d, the unrolled version is better for both timing and code size.

Algorithm 7 BBP$^+$ (Belaïd *et al.* , Eurocrypt'16) w/o conditional branches

Input: sharings $(a_1, \ldots, a_d) \in \{0,1\}^{32\times d}$ and $(b_1, \ldots, b_d) \in \{0,1\}^{32\times d}$
Output: sharing $(c_1, \ldots, c_d) \in \{0,1\}^{32\times d}$ such that $\bigoplus_i c_i = (\bigoplus_i a_i) \cdot (\bigoplus_j b_j)$

1: $c_1 \leftarrow a_1 \cdot b_1$
2: $c_2 \leftarrow a_2 \cdot b_2$
3: **for** $i = 3$ to $d - 1$ by 2 **do**
4: $\quad c_i \leftarrow a_i \cdot b_i$
5: $\quad c_{i+1} \leftarrow a_{i+1} \cdot b_{i+1}$
6: $\quad s_i \leftarrow \{0,1\}^{32}$
7: **for** $i = 1$ to $d - 1$ by 2 **do**
8: $\quad r_{i,i+1} \leftarrow \{0,1\}^{32}$
9: $\quad \text{LoopRow}(i, i + 3)$
10: $\quad c_i \leftarrow c_i \oplus (r_{i,i+1} \oplus a_i \cdot b_{i+1} \oplus a_{i+1} \cdot b_i)$
11: $\quad \text{LoopRow}(i + 1, i + 3)$
12: $\quad c_{i+1} \leftarrow c_{i+1} \oplus r$

Algorithm 8 LoopRow Procedure

Input: indexes i, t randoms $(s_j)_{j\in\{3,\ldots,d-1\}}$

1: **for** $j = d$ down to t by 2 **do**
2: $\quad r_{i,j} \leftarrow \{0,1\}^{32}$
3: $\quad c_i \leftarrow c_i \oplus \big(r \oplus (a_i \cdot b_j \oplus a_j \cdot b_i) \oplus s_{j-1} \oplus (a_i \cdot b_{j-1} \oplus a_{j-1} \cdot b_i)\big)$
4: $\quad c_j \leftarrow c_j \oplus r_{i,j}$

5.3.2.4 BPCZ: Towards Security against Horizontal Attacks

At CHES 2016, Battistello *et al.* described a horizontal side-channel attack on the standard ISW multiplication [**CHES:BCPZ16**]. This attack essentially consists in reducing the

Table 5.6: Implementation results for the BBP$^+$ multiplication

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
Original BBP$^+$	n/a	334	1204	4552	17680	n/a	344	344	344	344	12	$d + d^2/4$
Optimized BBP$^+$	88	274	970	3658	14218	428	428	428	428	428	12	$d + d^2/4$
Unrolled BBP$^+$	36	161	775	3018	11920	100	344	1544	5996	23732	11	$d + d^2/4$

noise in the targeted values by averaging them. More precisely, during the computation of Algorithm **??**, each share a_i (resp. b_i) is manipulated d times. Hence one can *average* the noise and reduce it by a factor $\sqrt{d}$ (in a standard deviation metric). Such an attack is inherent to the ISW scheme and implies that despite the probing-security, increasing the masking order d implies increasingly high noise requirements for the masking countermeasure to bring security improvements (i.e., for the noise to be large enough after averaging, it has to increase before averaging).

Battistello *et al.* also proposed a mitigation of such a horizontal attack. Their multiplication, given in Algorithm **??**, is similar to the standard ISW multiplication but the matrix of the crossed products $a_i \cdot b_j$ is computed differently (see Algorithm **??**): refreshings are regularly inserted to avoid the repeated apparition of each share a_i (resp. b_i). The RefreshMasks operation is a simple ISW-based refreshing as described later in Section **??**. The authors also proved that their multiplication is $(d-1)$-SNI secure.

Algorithm 9 BCPZ (Battistello *et al.* , CHES'16)

Input: shares a_i such that $\sum_i a_i = a$, shares b_i such that $\sum_i b_i = b$
Output: shares c_i such that $\sum_i c_i = a \cdot b$

1: $M_{i,j} \leftarrow \text{MatMult}((x_1, \ldots, x_d), (y_1, \ldots, y_d))$
2: **for** $i = 1$ to d **do**
3: $\quad c_i \leftarrow M_{i,i}$
4: **for** $i = 1$ to d **do**
5: $\quad$ **for** $j = i + 1$ to d **do**
6: $\quad\quad s \leftarrow \mathbb{F}$
7: $\quad\quad s' \leftarrow (s + M_{i,j}) + M_{j,i}$
8: $\quad\quad c_i \leftarrow c_i + s$
9: $\quad\quad c_j \leftarrow c_j + s'$
return $c_1, ..., c_d$

The implementation of Algorithm **??** is straightforward (same as ISW). The main challenge is to efficiently implement Algorithm **??** in a recursive way. In fact, due to the restrictive amount of registers available, using functions to perform the recursion in ARM assembly becomes very costly. Each recursive call needs to have access to several informations: the correct set of input sharings, namely the start of $\vec{X}_1$, $\vec{X}_2$, $\vec{Y}_1$ and $\vec{Y}_2$ as well as the correct addresses for the output sharings. This means that several registers containing those information need to be pushed to the stack prior to each call to a recursive function and popped before the computation. As push and pop are basically load and store in ARM assembly the total cost of managing the inputs and outputs of a recursive function is approximately equal to a dozen of clock cycles for each recursive calls. This costs, on top of the associated jumps for each recursive function, is equivalent to the computation of

Algorithm 10 MatMult

Input: the d-sharings $(x_i)_{i\in[1..d]}$ and $(y_i)_{i\in[1..d]}$ of x^* and y^* respectively
Output: the d^2-sharing $(M_{i,j})_{i\in[1..d],j\in[1..d]}$ of $x^* \cdot y^*$

1: **if** $n = 1$ **then**
2: $\quad \overrightarrow{M} \leftarrow [x_1 \cdot y_1]$
3: $\quad \vec{X}^{(1)} \leftarrow (x_1, \ldots, x_{d/2})$, $\vec{X}^{(1)} \leftarrow (x_{d/2+1}, \ldots, x_d)$
4: $\quad \vec{Y}^{(1)} \leftarrow (y_1, \ldots, y_{d/2})$, $\vec{Y}^{(1)} \leftarrow (y_{d/2+1}, \ldots, y_d)$
5: $\quad \overrightarrow{M}^{(1,1)} \leftarrow \text{MatMult}(\vec{X}^{(1)}, \vec{Y}^{(1)})$
6: $\quad \vec{X}^{(1)} \leftarrow \text{RefreshMasks}(\vec{X}^{(1)})$, $\vec{Y}^{(1)} \leftarrow \text{RefreshMasks}(\vec{Y}^{(1)})$
7: $\quad \overrightarrow{M}^{(1,2)} \leftarrow \text{MatMult}(\vec{X}^{(1)}, \vec{Y}^{(2)})$
8: $\quad \overrightarrow{M}^{(2,1)} \leftarrow \text{MatMult}(\vec{X}^{(2)}, \vec{Y}^{(1)})$
9: $\quad \vec{X}^{(2)} \leftarrow \text{RefreshMasks}(\vec{X}^{(2)})$, $\vec{Y}^{(2)} \leftarrow \text{RefreshMasks}(\vec{Y}^{(2)})$
10: $\quad \overrightarrow{M}^{(2,2)} \leftarrow \text{MatMult}(\vec{X}^{(2)}, \vec{Y}^{(2)})$
11: $\quad \overrightarrow{M} \leftarrow \begin{pmatrix} \overrightarrow{M}^{(1,1)} & \overrightarrow{M}^{(1,2)} \\ \overrightarrow{M}^{(2,1)} & \overrightarrow{M}^{(2,2)} \end{pmatrix}$
return $\overrightarrow{M}$

Table 5.7: Implementation results for the BCPZ multiplication

	clock cycles					code size (bytes)					register	random usage
d	2	4	8	16	32	2	4	8	16	32		
BCPZ (macros)	108	498	2106	8698	35386	240	648	2324	9368	38168	13	$(\log(d)-1)d^2/2-(d/2-1)d$
BCPZ (functions)	134	593	2529	10473	42649	400	476	780	1996	6860	13	$(\log(d)-1)d^2/2-(d/2-1)d$

a complete ISW multiplication. Therefore and since we restrict ourselves in this study to $d \leq 32$, we developed the MatMult procedure with macros. Specifically, for each masking order d that is a power of 2, we simply implement Algorithm **??** using macros for each possible input sharing size $d \in \{2, 4, \ldots, 32\}$, which allows us to save several clock cycles. However the main drawback of implementing the MatMult procedure in such way is that the code size grows exponentially. To lower down the explosion of the code size, we have also implemented a version of the code where the terminal case macro (for $d = 2$) is implemented as a function. This allows us to divide by up to 5 the code size while having a performance decrease of around 20%. Both timing and code size for the BPCZ multiplication with the two versions of the MatMult procedure are given in Table **??**.

5.3.2.5 CPRR evaluation.

The CPRR scheme was initially proposed in [**FSE:CPRR13**] as a variant of ISW to securely compute multiplications of the form $x \mapsto x \cdot \ell(x)$ where ℓ is linear, without requiring refreshing. It was then shown in [**C:CPRR15**] that this scheme (in a slightly modified version) could actually be used to securely evaluate any quadratic function f over $\mathbb{F}_{2^\lambda}$. The method is based

Algorithm 11 Quadratic Evaluation

Input: shares a_i such that $\sum_i a_i = a$, a look-up table for a algebraic degree-2 function h
Output: share c_i such that $\sum_i c_i = h(a)$

1: **for** $i = 0$ to d **do**
2: $\quad c_i = h(a_i)$
3: **for** $i = 0$ to d **do**
4: $\quad$ **for** $j = i + 1$ to d **do**
5: $\quad\quad s \leftarrow \mathbb{F}_{2^\lambda}$
6: $\quad\quad s' \leftarrow \mathbb{F}_{2^\lambda}$
7: $\quad\quad t \leftarrow s$
8: $\quad\quad t \leftarrow t + h(a_i + s')$
9: $\quad\quad t \leftarrow t + h(a_j + s')$
10: $\quad\quad t \leftarrow t + h((a_i + s') + a_j)$
11: $\quad\quad t \leftarrow t + h(s')$
12: $\quad\quad s' \leftarrow t$
13: $\quad\quad c_i \leftarrow c_i + s$
14: $\quad\quad c_j \leftarrow c_j + s'$
return $c_1, ...c_d$

on the following equation

$$f(x_1 + x_2 + \cdots + x_d) = \sum_{1 \le i < j \le d} f(x_i + x_j + s_{i,j}) + f(x_j + s_{i,j}) + f(x_i + s_{i,j}) + f(s_{i,j}) + \sum_{i=1}^{d} f(x_i) + (d + 1 \bmod 2) \cdot f(0) \quad (5.8)$$

which holds for every $(x_i)_i \in (\mathbb{F}_{2^\lambda})^d$, every $(s_{i,j})_{1 \le i < j \le d} \in (\mathbb{F}_{2^\lambda})^{d(d-1)/2}$, and every quadratic function f over $\mathbb{F}_{2^\lambda}$.

From a d-sharing $(x_1, x_2, \ldots, x_d)$, the CPRR scheme computes an output d-sharing $(y_1, y_2, \ldots, y_d)$ as follows:

1. for every $1 \le i < j \le d$, sample two random values $r_{i,j}$ and $s_{i,j}$ over $\mathbb{F}_{2^\lambda}$,
2. for every $1 \le i < j \le d$, compute $r_{j,i} = r_{i,j} + f(x_i + s_{i,j}) + f(x_j + s_{i,j}) + f((x_i + s_{i,j}) + x_j) + f(s_{i,j})$,
3. for every $1 \le i \le d$, compute $y_i = f(x_i) + \sum_{j \ne i} r_{i,j}$,
4. if d is even, set $y_1 = y_1 + f(0)$.

According to **??**, we then have $\sum_{i=1}^{d} y_i = f(\sum_{i=1}^{d} x_i)$, which shows that the output sharing $(y_1, y_2, \ldots, y_d)$ well encodes $y = f(x)$. The overall computation is summarized in Algorithm **??**.

In [**FSE:CPRR13**; **C:CPRR15**] it is argued that in the gap where the field multiplication cannot be fully tabulated ($2^{2\lambda}$ elements is too much) while a function $f : \mathbb{F}_{2^\lambda} \to \mathbb{F}_{2^\lambda}$ can be tabulated (2^λ elements fit), the CPRR scheme is (likely to be) more efficient than the ISW scheme. This is because it essentially replaces (costly) field multiplications by simple look-ups. We present in the next section the results of our study for our optimized ARM implementations.

5.3.2.6 Parallelization

Both ISW and CPRR schemes work on λ-bit variables, each of them occupying a full 32-bit register. Since in most practical scenarios, we have $\lambda \in [\![4, 8]\!]$, this situation is clearly suboptimal in terms of register usage, and presumably suboptimal in terms of timings. A natural idea to improve this situation is to use parallelization. A register can simultaneously store $m := \lfloor 32/\lambda \rfloor$ values, we can hence try to perform m ISW/CPRR computations in parallel (which would in turn enable to perform m s-box computations in parallel). Specifically, each input share is replaced by m input shares packed into a 32-bit value. The ISW (resp. CPRR) algorithm loads packed values, and performs the computation on each unpacked λ-bit chunk one-by-one. Using such a strategy allows us to save multiple load and store instructions, which are among the most expensive instructions of ARMv7 assembly (3 clock cycles). Specifically, we can replace m load instructions by a single one for the shares a_i, b_j in ISW (resp. x_i, x_j in CPRR) and the random values $r_{i,j}$, $s_{i,j}$ (read from the TRNG), we can replace m store instructions by a single one for the output shares, and we can replace m XOR instructions by a single one for some of the addition involved in ISW (resp. CPRR). On the other hand, we get an overhead for the extraction of the λ-bit chunks from the packed 32-bit values. But each of these extractions takes a single clock cycle (thanks to the barrel shifter), which is rather small compared to the gain in load and store instructions.

Remark 5.3.3. *Note that using parallelization in our implementations does not compromise the probing security. Indeed, we pack several bytes/nibbles within one word of the cipher state but we never pack (part of) different shares of the same variable together. The probing security proofs hence apply similarly to the parallel implementations.*[3]

5.3.3 Refresh gadgets

5.3.3.1 ISW Refresh

The ISW-based mask refreshing is pretty similar to an ISW multiplication, but it is actually much faster since it involves no field multiplications and fewer additions (most terms being multiplied by 0). It simply consists in processing:

$$\text{for } i = 1 \,..\, d : \text{ for } j = i + 1 \,..\, d : \; r \leftarrow \$;\; a_i \leftarrow a_i + r;\; a_j \leftarrow a_j + r;$$

A straightforward implementation of this process is almost 3 times faster than the fastest ISW multiplication, namely the full-table one (see Figure **??**).

We can actually do much better. Compared to a standard ISW implementation, the registers of the field multiplication are all available and can hence be used in order to save several loads and stores. Indeed, the straightforward implementation performs $d - i + 1$ loads and stores for every $i \in [\![1, d]\!]$, specifically 1 load-store for a_i and $d - i$ for the a_j's. Since we have some registers left, we can actually pool the a_j's loads and stores for several a_i's. To do so, we load several shares a_i, a_{i+1}, ..., a_{i+k} with the `LDM` instruction (which has a cost of $k + 2$ instead of $3k$) and process the refreshing between them. Then, for every

[3] Putting several shares of the same variable in a single register would induce a security flaw in the probing model where full registers can be probed. For this reason, we avoid doing so and we stress that parallelization does not result in such an undesired result. However, it should be noted that in some other relevant security models, such as the single-bit probing model or the *bounded moment leakage model* [**EPRINT:BDFGSS16**], this would not be an issue anyway.

$j \in [\![i+k+1, d]\!]$, we load a_j, perform the refreshing between a_j and each of the a_i, a_{i+1}, ..., a_{i+k}, and store a_j back. Afterwards, the shares a_i, a_{i+1}, ..., a_{i+k} are stored back with the `STM` instruction (which has a cost of $k+2$ instead of $3k$). This allows us to load (and store) the a_j only once for the k shares instead of k times, and to take advantage of the `LDM` and `STM` instructions. In practice, we could deal with up to $k = 8$ shares at the same time, meaning that for $d \leq 8$ all the shares could be loaded and stored an single time using `LDM` and `STM` instructions.

Table 5.8: Timings of the ISW-based mask refreshing.

Straightforward version	$7.5\,d^2 + 1.5\,d - 7$
Optimized version	$\leq 2.5\,d^2 + 2.5\,d + 2$

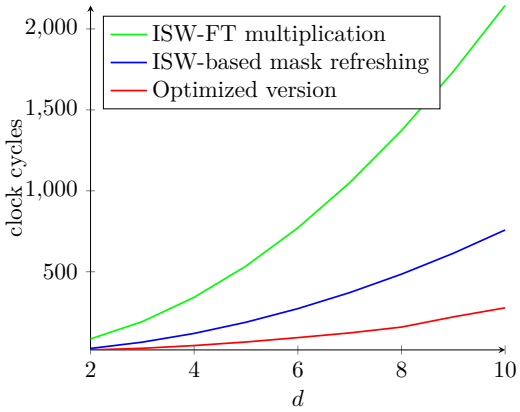


Figure 5.2: Timings of mask refreshing.

The performance of our implementations of the ISW-based mask refreshing is given in Table **??** and plotted in Figure **??** for illustration.[4] The performances of our implementations of the ISW-based mask refreshing are plotted in Figure **??**. Our optimized refreshing is up to 3 times faster than the straightforward implementation and roughly 10 times faster that the full-table-based ISW multiplication.

5.3.3.2 BDF$^+$ Refresh

Barthe *et al.* in [**EC:BDFGSS17**], along with their multiplication gadget, also provide a refreshing gadget described in Algorithm **??**. It simply consists in XORing the share to refresh by a random value and a rotation of it. The iteration of the BDF$^+$ refresh $\lceil (d-1)/3 \rceil$ times makes it SNI secure. The overall BDF$^+$ refresh needs $d\lceil (d-1)/3 \rceil$ random bits and performs $2\lceil (d-1)/3 \rceil$ additions and $\lceil (d-1)/3 \rceil$ ROT. There are no particular implementations tricks except that we use the same ROT algorithm introduced in Section **??** in order to keep the correctness with the specific encoding. Implementation results can be found in Table **??**.

[4] Note that the timings of the optimized version cannot be fully interpolated with a quadratic polynomial but we provide a tight quadratic upper bound (see Table **??**).

Algorithm 12 BDF$^+$ Refresh

Input: shares a
Output: shares c
1: $r \leftarrow \{0,1\}^{32}$
2: $c \leftarrow a \oplus r \oplus \mathsf{ROT}(r,1)$ **return** c

Table 5.9: Implementation results for the BDF$^+$ refresh

	clock cycles					code size (bytes)					register usage	random usage
d	2	4	8	16	32	2	4	8	16	32		
BDF$^+$ Refresh	25	25	25	25	16	116	116	116	116	110	10	d

5.3.4 Performance and Comparisons

5.3.4.1 Secure Multiplications Comparison

We now compare the four multiplication algorithms over $\mathbb{F}_2$, *i.e.* when working on the Boolean field. In Table **??** we gather the four multiplications we studied in this section and we compare them at an algorithmic level. Namely, we give the operation counts (in terms of 32-bit XOR, 32-bit AND, and sharing-wise ROT) to perform a secure 32-bit AND between two sharings. The NI/SNI row specifies if the considered multiplication is SNI- or NI-secure. The row "max use of shares" represents (informally) the level of protection against horizontal side-channels attacks: $O(d)$ means that each share is processed a linear number in d times (*i.e.* no protection) and $O(1)$ means that each share is processed a constant number of times (*i.e.* protection).

We differentiate two cases for the BDF$^+$ multiplication. A first case where we consider the multiplication alone, which is SNI until $d = 3$ and only NI secure afterwards. A second case where we consider the composition of the multiplication with one iteration of the BDF$^+$ refresh (described in Section **??**), which is SNI secure up to $d = 8$ and only NI secure afterwards (see [**EC:BDFGSS17**]). The cost difference between these two versions is simply the cost of an elementary refresh (i.e., the addition of a share of zero). Finding the number of such refreshes that are required to be SNI at any order is an open problem. Note that for BDF$^+$, the results are given for d calls to the multiplication (since each call allows to compute $32/d$ elements).

We note that we did not perform the same addition for the BBP$^+$ multiplication since it would imply the need of a more expensive SNI refresh on the output, which would contradict the goal of [**EC:BBPPTV16**] to minimize randomness by mixing NI and SNI multiplications instead of solely SNI multiplications (and in particular, if an SNI multiplication is required, one could use the ISW one, or the BDF$^+$ up to order 8).

We also recall that this table does not mention the different risks of an unsatisfied (independence) assumption. Namely the fact that the BDF$^+$ multiplication can suffer from a reduced security order due to couplings while for the other algorithms, the main risk of security order reduction comes from transition-based leakages.

Based on the results in the previous sections, we can compare the performance of our implementations of the multiplications for bitsliced inputs with higher-order masking in ARM v7. We make the comparison for five masking orders, namely 2, 4, 8, 16 and 32. Moreover, we

Algorithm:	ISW	BDF$^+$ (BM model)	BDF$^+$ w. refresh (BM model)	BBP$^+$	BCPZ
NI/SNI:	SNI	SNI (up to $d = 3$)	SNI (up to $d = 8$)	NI	SNI
Max use of shares:	$O(d)$	$O(d)$	$O(d)$	$O(d)$	$O(1)$
XOR-32 count:	$2d(d-1)$	$d(3d/2-1)$	$d(3d/2+1)$	$(7d^2-6d)/4$	$d^2\log(d)+2d$
AND-32 count:	d^2	d^2	d^2	d^2	d^2
ROT count:	0	$d(5d/4-1)$	$5d^2/4$	0	0
Random bits:	$16d(d-1)$	$32d\lceil(d-1)/4\rceil$	$32d\lceil(d-1)/4\rceil+32$	$8d^2+16d-1$	$16d^2\log(d)+d$

Table 5.10: Comparison of the multiplications at the algorithmic level.

also give the performance results for two sets of TRNG. For the first one (called the TRNG-1 settings in the following), we make the same assumption as in [**EC:GouRiv17**] that we need to wait 10 clock cycles to get a fresh 32-bit random word. For the second one (called the TRNG-2 settings in the following), we make the same assumption as in [**CHES:JouSta17**] that we need to wait 80 clock cycles to get a fresh 32-bit random word. Finally, in order to have a fair comparison between the four algorithms the implementation results are given for the computation of a multiplication between two shared 32-bit operands. This means that for the 3 ISW-based multiplication (ISW, BCPZ, BBP$^+$) the results are given for a single call to their respective functions, whereas for the BDF$^+$ multiplication the results are given for d calls to the function (since each calls allows to compute $32/d$ elements). The overall results are given in Tables **??** and **??** for respectively the TRNG-1 and the TRNG-2 settings. As illustration, we also plot the performances in clock cycles (log scale) for both TRNG-1 and TRNG-2 settings in Figure **??** and Figure **??** respectively.

Table 5.11: Multiplication performances for TRNG-1.

	TRNG-1									
	clock cycles					code size (bytes)				
d	2	4	8	16	32	2	4	8	16	32
ISW	75	291	1155	4611	18435	164	164	164	164	164
ISW unrolled	58	231	949	3876	15682	132	464	1848	7500	30324
BDF$^+$	n/a	308	1168	4560	n/a	n/a	248	244	240	n/a
BDF$^+$ unrolled	68	188	648	2384	3840	280	356	504	808	748
BDF$^+$ (+ refresh)	n/a	408	1568	5360	n/a	n/a	360	356	352	n/a
BDF$^+$ unrolled (+ refresh)	118	288	1048	3184	5440	392	468	616	920	960
BBP$^+$	88	274	970	3658	14218	428	428	428	428	428
BBP$^+$ unrolled	36	161	775	3018	11910	100	344	1544	5996	23732
BCPZ (macros)	108	498	2106	8698	35386	240	648	2334	9368	38168
BCPZ (macros + functions)	134	593	2529	10473	42649	400	476	780	1996	6860
ISW Refresh	51	72	239	933	3761	236	236	236	236	236
BDF$^+$ Refresh	50	50	50	50	50	128	128	128	128	128

Table 5.12: Multiplication performances for TRNG-2.

	TRNG-2									
	clock cycles					code size (bytes)				
d	2	4	8	16	32	2	4	8	16	32
ISW	166	837	3703	15531	63571	500	500	500	500	500
ISW unrolled	149	777	3497	14796	60818	480	872	2508	9264	36600
BDF$^+$	n/a	672	2624	10384	n/a	n/a	596	592	588	n/a
BDF$^+$ unrolled	250	552	2104	8208	27136	448	500	876	1204	1192
BDF$^+$ (+ refresh)	n/a	1136	3552	12240	n/a	n/a	1016	1012	1008	n/a
BDF$^+$ unrolled (+ refresh)	482	1016	3032	10064	30848	868	920	1296	1624	1612
BBP$^+$	270	820	2790	10210	38970	800	800	800	800	800
BBP$^+$ unrolled	127	525	2504	9479	36581	436	716	2096	7172	27776
BCPZ (macros)	199	1408	7202	32358	136942	576	1032	2988	11372	45932
BCPZ (macros + functions)	225	1503	7625	34133	144205	760	836	1128	2344	7208
ISW Refresh	142	345	2241	10761	46713	412	412	412	412	412
BDF$^+$ Refresh	116	116	116	116	116	420	420	420	420	420

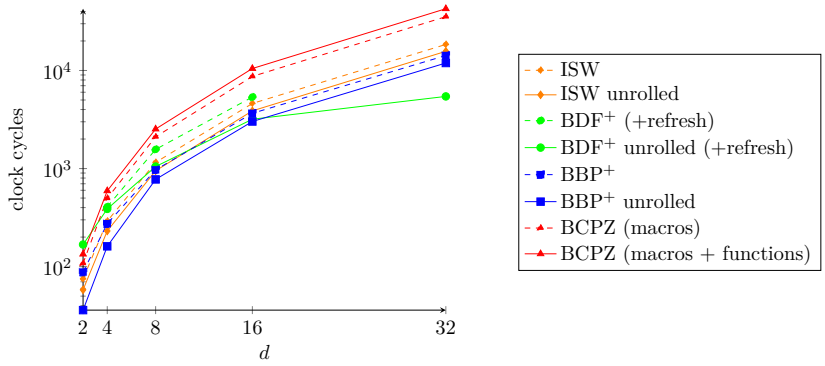
As expected the BCPZ offers the worst performances because of the many refreshings which intend to provide resistance to horizontal side-channel attacks, for both of the TRNG settings.

The BBP$^+$ multiplication outperforms the ISW multiplication (up to 25% faster) even in the case where the randomness is cheap. The difference becomes more significant in the TRNG-2 context (up to 40% faster), since BBP$^+$ have reduced randomness requirements.

For the TRNG-2 settings, we can also observe that unrolling the loops does not offer an interesting tradeoff as the gain in timing is not very significant compared to the code size overhead.

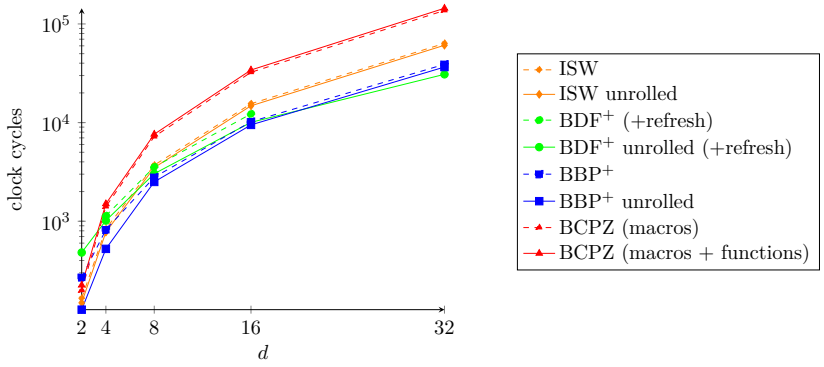
As shown in Table 2 of [**EC:BDFGSS17**], the iteration of the BDF$^+$ refresh requires a bit less randomness than ISW one but is more computationally involved. This is well reflected in Tables **??** and **??**: the ISW refresh has better performance than the BDF$^+$ refresh for the TRNG-1 setting while it is the opposite for the TRNG-2 setting.

Figure 5.3: Multiplication performances for TRNG-1 in clock cycles



Overall, BDF$^+$ and BBP$^+$ multiplications provide the best performance in both TRNG settings thanks to their lower randomness requirements (compared to the classical ISW). Of course these two multiplications also have weaker security guaranties (in terms of composability

Figure 5.4: Multiplication performances for TRNG-2 in clock cycles



and resistance against horizontal attacks). On the other hand, ISW and BCPZ offer better security guaranties and hence are more involved in terms of randomness requirements, making these differences more visible in the TRNG-2 setting.

5.3.4.2 ISW vs CPRR

As argued in Section **??**, we consider three variants for the base field multiplication in the ISW scheme, namely the full-table method, the half-table method and the exp-log method (with doubled exp table). The obtained ISW variants are labeled ISW-FT, ISW-HT and ISW-EL in the following. The obtained performance are summarized in Table **??** where the clock cycles with respect to d have been obtained by interpolation. Note that we did not consider ISW-FT for $n > 8$ since the precomputed tables are too huge. The timings (for $n \leq 8$) are further illustrated in Figure **??** with respect to d. The obtained performance are illustrated in Figure **??** with respect to d. Note that we did not consider ISW-FT for $n > 8$ since the precomputed tables are too huge.

Table 5.13: Performance of ISW and CPRR schemes.

	ISW-FT	ISW-HT	ISW-EL	CPRR
Clock cycles (for $n \leq 8$)	$21.5\,d^2 - 0.5\,d$	$28.5\,d^2 - 0.5\,d$	$33.5\,d^2 - 0.5\,d$	$25d^2 - 8\,d$
Clock cycles (for $n > 8$)	n/a	n/a	$52.5d^2 - 4.5d + 12$	$37d^2 - 14\,d$
Code size (bytes)	$184 + 2^{2n}$	$244 + 2^{\frac{3n}{2}+1}$	$280 + 3 \cdot 2^n$	$196 + 2^n$
Random usage (bytes)	$\frac{d(d-1)}{2}$	$\frac{d(d-1)}{2}$	$\frac{d(d-1)}{2}$	$d(d-1)$

These results show that CPRR indeed outperforms ISW whenever the field multiplication cannot be fully tabulated. Even the half-table method (which is more consuming in code-size) is slower than CPRR. For $n \leq 8$, a CPRR evaluation asymptotically costs 1.16 ISW-FT, 0.88 ISW-HT, and 0.75 ISW-EL.

We implemented parallel versions of ISW and CPRR for $n = 4$ and $n = 8$. For the former case, we can perform $m = 8$ evaluations in parallel, whereas for the later case we can perform $m = 4$ evaluations in parallel. For $n = 4$, we only implemented the full-table multiplication

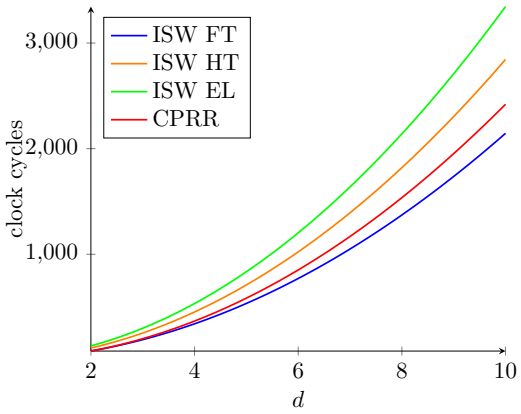


Figure 5.5: Timings of ISW and CPRR schemes.

for ISW, since we consider that a 256-byte table in code is always affordable. For $n = 8$ on the other hand, we did not implement the full-table, since we consider that a 64-KB table in code would be to much in most practical scenarios. Tables **??** and **??** give the obtained performance in terms of clock cycles and code size. For comparison, we give both the performance of the m-parallel case and that of the m-serial case. We also exhibit the asymptotic ratio *i.e.* the ratio between the d^2 constants of the serial and parallel case. These performance are illustrated on Figures **??** and **??**. Figures **??** and **??** give the obtained performance in terms of clock cycles.

Table 5.14: Performance of parallel ISW and CPRR schemes for $n = 8$.

	ISW-HT $/\!/$ 4	ISW-HT $\times$ 4	ISW-EL $/\!/$ 4	ISW-EL $\times$ 4	CPRR $/\!/$ 4	CPRR $\times$ 4
Clock cycles	$77.5\,d^2 - 1.5\,d + 2$	$114d^2 - 2d$	$95.5\,d^2 - 1.5\,d + 2$	$134\,d^2 - 2\,d$	$54\,d^2 - 22\,d$	$100\,d^2 - 32\,d$
Asympt. ratio	68%	100%	70%	100%	54%	100%
Code size	8.6 KB	8.1 KB	1.5 KB	0.9 KB	580 B	408 B
Random usage	$\frac{d(d-1)}{2}$	$2d(d-1)$	$\frac{d(d-1)}{2}$	$2d(d-1)$	$d(d-1)$	$4d(d-1)$

Table 5.15: Performances of parallel ISW and CPRR schemes for $n = 4$.

	ISW-FT $/\!/$ 8	ISW-FT $\times$ 8	CPRR $/\!/$ 8	CPRR $\times$ 8
Clock cycles	$72\,d^2$	$172d^2 - 4d$	$94\,d^2 - 42\,d - 3$	$200\,d^2 - 64\,d$
Asympt. ratio	42%	100%	47%	100%
Code size	804 B	388 B	596 B	168 B
Random usage	$\frac{d(d-1)}{2}$	$4d(d-1)$	$d(d-1)$	$8(d-1)$

These results show the important gain obtained by using parallelism. For ISW, we get an asymptotic gain around 30% for 4 parallel evaluations ($n = 8$) compared to 4 serial evaluations, and we get a 58% asymptotic gain for 8 parallel evaluations ($n = 4$) compared

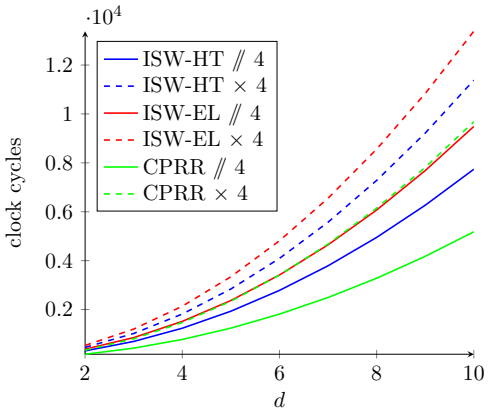


Figure 5.6: Timings of (parallel) ISW and CPRR schemes for $n = 8$.

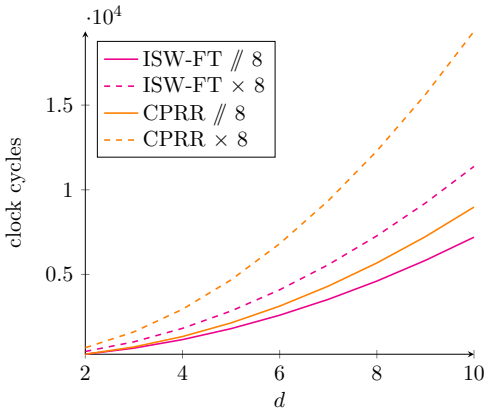


Figure 5.7: Timings of (parallel) ISW and CPRR schemes for $n = 4$.

to 8 serial evaluations. For CPRR, the gain is around 50% (timings are divided by 2) in both cases ($n = 8$ and $n = 4$). We also observe that the efficiency order keeps unchanged with parallelism, that is: ISW-FT > CPRR > ISW-HT > ISW-EL.

5.3.4.3 Register Usage for ISW and CPRR Implementations

The register usage of our implementations of ISW and CPRR is given in Table **??**. For ISW, the bottleneck for "temporary variables" is the register usage of the multiplication. From the ISW loop, one can check that we do not need the multiplication operands once they have been multiplied. This allowed us to save 2 registers in the full-table and exp-log multiplications, and 1 register in the half-table multiplication. These savings are indicated under brackets in Table **??** (and they are taken into account in the total).

Table 5.16: Register usage

	Input/output addresses	LUT & TRNG addresses	Loop counters	ISW / CPRR variables	Temporary variables	Total
ISW-FT/EL	3	2	2	1	5 (-2)	11
ISW-HT	3	2	2	1	5 (-1)	12
CPRR	2	2	2	2	4	12

Regarding parallel versions, ISW implementations need two more registers compared to standard implementations in order to store the extracted n-bit chunks from packed 32-bit values for a_i and b_j. In the case of CPRR, a single additional register is sufficient for manipulating the shares of x with respect to the two loop counters i and j. As it can be deduced from Table **??**, our parallel implementations of ISW and CPRR hence uses 13 registers, except for ISW-HT that needs 14 registers (*i.e.* it uses the link register), which is pretty tight.

5.4 Generic Implementations

5.4.1 Boolean decomposition

We now describe our implementations of a bitsliced s-box layer protected with higher-order masking based on our decomposition method over $\mathbb{F}_2$. Our implementations evaluate 16 $n \times n$ s-boxes in parallel where $n \in \{4, 8\}$, and they are developed in generic 32-bit ARM assembly. They take n input sharings $[\vec{x}_1]$, $[\vec{x}_2]$, ..., $[\vec{x}_n]$ defined as

$$[\vec{x}_i] = (\vec{x}_{i,1}, \vec{x}_{i,2}, \ldots, \vec{x}_{i,d}) \text{ such that } \sum_{j=1}^{d} \vec{x}_{i,j} = \vec{x}_i \tag{5.9}$$

where $\vec{x}_i$ is a 16-bit register containing the i-th bit of the 16 s-box inputs. Our implementations then output n sharings $[\vec{y}_0]$, $[\vec{y}_1]$, ..., $[\vec{y}_n]$ corresponding to the bitsliced output bits of the s-box. Since we are on a 32-bit architecture with 16-bit bitsliced registers, we use a degree-2 parallelization for the multiplications. Namely, the 16-bit ANDs are packed by pairs and replaced by 32-bit ANDs which are applied on shares using the ISW scheme.

The computation is then done in three stages. First, we need to construct the shares of the elements of the minimal basis $\mathcal{B}_0$. Once the first stage is completed, all the remaining multiplications are done between linear combinations of the elements of the basis. Let us denote by $[\vec{t}_i]$ the sharings corresponding to the elements of the basis which are stored in memory. After the first stage we have $\{[\vec{t}_i]\} = \{[\vec{x}^u] \mid u \in \mathcal{U}\}$. Each new $\vec{t}_i$ is defined as

$$\Big(\sum_{j<i} a_{i,j} \vec{t}_j\Big) \odot \Big(\sum_{i<j} b_{i,j} \vec{t}_j\Big) \tag{5.10}$$

where $\odot$ denote the bitwise multiplication, and where $\{a_{i,j}\}_j$ and $\{b_{i,j}\}_j$ are the binary coefficients obtained from the s-box decomposition (namely the coefficients of the functions $g_{i,j}$ and $h_{i,j}$ in the span of the basis). The second stage hence consists in a loop on the remaining multiplications that

1. computes the linear-combination sharings $[\vec{r}_i] = \sum_{j<i} a_{i,j}[\vec{t}_j]$ and $[\vec{s}_i] = \sum_{j<i} b_{i,j}[\vec{t}_j]$
2. refreshes the sharing $[\vec{r}_i]$
3. computes the sharing $[\vec{t}_i]$ such that $\vec{t}_i = \vec{r}_i \odot \vec{s}_i$

where the last step is performed for two successive values of i at the same time by a call to a 32-bit ISW-AND. The sums in Step 1 are performed on each share independently. The refreshing procedure is simply a ISW refresh as explained in Section **??**.

Once all the basis sharings $[\vec{t}_i]$ have been computed, the third stage simply consists in deriving each output sharing $[\vec{y}_i]$ as a linear combination of the $[\vec{t}_i]$, which is refreshed before being returned.

5.4.1.1 Linear parts.

We now explain how to optimize the evaluation of the linear parts of the implementation. In fact during the evaluation of a decomposition, we need a large number of linear combinations of elements of the basis. This linear parts have significant impacts on the performances (see Table **??**) for small masking order. Therefore, in our ARM implementation we try to

propose optimization in order to reduce as much as possible this linear overheads. To do so, we use windowing technique to perform evaluations of elements in the basis. More precisely, the evaluation is computed with $k = \frac{\text{cardinality of the basis}}{\text{architecture size}}$ calls to a macro that performs v conditional XORS, where v is the architecture size. In our case, since the implementation is done on a 32-bit ARM architecture, v is equal to 32 and k is at most 4. Therefore, we test the size of the basis to perform the sound number of calls to the windowing macro. Moreover, we also define a macro just for the computation of the first coordinate function, which is composed of 31 conditional xors (since the first coordinate function is only composed of elements of $\mathcal{B}_0$).

5.4.2 Median case decomposition

Based on our generic decomposition method, we now describe our implementation of an s-box layer protected with higher-order masking in ARM v7. We focused our study on the common scenario of a layer applying 16 8-bit s-boxes to a 128-bit state. We apply our generalized decomposition with parameters $n = m = 2$ and $\lambda = 4$ (medium case) to compare the obtained implementation to the ones for the two extreme cases.

Our implementation is based on the decomposition obtained for the CLEFIA S0 s-box with parameters $(r, t_1, t_2) = (7, 7, 4)$. Note that it would have the same performances with any other 8-bit s-box with the same decomposition parameters (which we validate on all our tested random 8-bit s-boxes). The input (x_1, x_2) of each s-box is shared as $([x_1], [x_2])$ where

$$[x_i] = (x_{i,1}, x_{i,2}, \ldots, x_{i,d}) \quad \text{such that} \quad \sum_{j=1}^{d} x_{i,j} = x_i \ . \tag{5.11}$$

Note that for those chosen parameters (n, m, λ), the input x_1 and x_2 are 4-bit elements, *i.e.* the inputs of the 8-bit s-boxes are split into 2. The output of the computation is a pair $([y_1], [y_2])$ where y_1 and y_2 are the two 4-bit coordinates of the s-box output.

The computation is then pretty regular. It can be summarize with the following pseudo-code:

Algorithm 13 S-box evaluation pseudo-code

Input: $([x_1], [x_2])$
Output: $([y_1], [y_2]) = S([x_1], [x_2])$
1: $\mathcal{B} = ([x_1], [x_2])$
2: **for** $i = 3$ to 21 **do**
3: $\quad \mathcal{B} = \text{compute\_pairwise\_products}([\text{u}_\text{i}], [\text{v}_\text{i}]) \cup \mathcal{B}$
4: $([y_1], [y_2]) = \text{evaluate\_sbox}(\mathcal{B})$
5: **return** $([y_1], [y_2])$

We start from a basis that contains the input sharings $\{[z_1], [z_2]\} = \{[x_1], [x_2]\}$. Then for $i = 3$ to 21 each of the 18 multiplications is performed between two linear combinations of the elements of the basis, that is

$$[z_i] = [u_i] \odot [v_i] \ , \tag{5.12}$$

where $\odot$ denotes the ISW multiplication with refreshing of one of the operand and where

$$u_{i,j} = \sum_{k<i} \ell_{i,k}(z_{k,j}) \ \text{ and } \ v_{i,j} = \sum_{k<i} \ell'_{i,k}(z_{k,j}) \ \text{ for every } j \in [1, d], \tag{5.13}$$

for some linearized polynomials $\ell_{i,k}$ and $\ell'_{i,k}$ obtained from the s-box decomposition. Once all the products have been computed, the output sharings $[y_1]$ and $[y_2]$ are simple linear combinations of the computed $[z_i]$.

To make the most of the 32-bit architecture, the s-box evaluations are done eight-by-eight since we can fill a register with eight 4-bit elements. The ISW-based multiplications can then be parallelized as suggested in Section **??** except for the field multiplications between two shares. To perform those multiplications, we simply need to unpack the eight 4-bit elements in each 32-bit operand, and then to sequentially perform the 8 field multiplications. These field multiplications are fully tabulated which only takes 0.25KB of ROM on $\mathbb{F}_{16}$ (following the results of Section **??**).

5.4.3 Algebraic Decomposition

We implemented the search of sound algebraic decompositions for $\lambda \in [\![4, 10]\!]$. Here again, we looked for *full rank* systems *i.e.* systems that would work with any target s-box. For each value of λ, we set r to the smallest integer satisfying condition (2) *i.e.* $r \geq \log_2(\lambda) - 1$, and then we looked for a t starting from the lower bound $t \geq \frac{2(2^\lambda - \lambda(r+1))}{\lambda(\lambda-1)}$ (obtained from condition (1)) and incrementing until a solvable system can be found. We then increment r and reiterate the process with t starting from the lower bound, and so on. For $\lambda \leq 8$, we found the same parameters as those reported in [**C:CPRR15**]. For $\lambda = 9$ and $\lambda = 10$ (these cases were not considered in [**C:CPRR15**]), the best parameters we obtained were $(r, t) = (3, 14)$ and $(r, t) = (4, 22)$ respectively. All these parameters are recalled in Table **??**.

Table 5.17: Obtained parameters for the algebraic decomposition method.

λ	4	5	6	7	8	9	10
(r, t)	(1,2)	(2,2)	(2,3)	(2,6)	(2,9)	(3,14)	(4,22)

Remark 5.4.1. *To avoid redundancy with the linear combinations of the g_i's, the authors of [***C:CPRR15***] suggest to derive p_i's that have null linear parts. In order to save some memory spaces, we make the choice of searching for p_i's with linear parts and remove the linear terms $\ell_i(g_i(x))$ in Equation (**??**). Hence, the goal is now to look for a decomposition $S(x) = \sum_{i=1}^{t} p_i(q_i(x))$, where the p_i's contain linear terms.*

5.4.4 Performance Comparisons

We now compare the performance results of the implementations of the four following decomposition methods (3 instantiations of our generic decomposition method and the algebraic decomposition):

- **Boolean case** ($\lambda = 1$, $n = 8$)
- **Median case** ($\lambda = 4$, $n = m = 2$)
- **Plain field case** ($\lambda = 8$, $n = 1$)
- **Algebraic decomposition**

For each implementation, the performances are compared for the computation of 16 s-boxes on a 32-bit ARM architecture. Table **??** summarizes the obtained performances in clock cycles with respect to the masking order d.

Table 5.18: Performances in clock cycles.

	CRV	**AD**	**Boolean case**	**Median case**
	$4 \times 4/\!/\!/$ s-boxes	$4 \times 4/\!/\!/$ s-boxes	$16/\!/\!/$ s-boxes	$2 \times 8/\!/\!/$
$\lambda = 8$	$2576\,d^2 + 5476\,d + 2528$	$2376\,d^2 + 3380\,d + 5780$	$656\,d^2 + 19786\,d + 5764$	$2757\,d^2 + 17671\,d + 2402$
	$2 \times 8/\!/\!/$ s-boxes	$2 \times 8/\!/\!/$ s-boxes	$16/\!/\!/$ s-boxes	n/a
$\lambda = 4$	$337\,d^2 + 563\,d + 434$	$564\,d^2 + 270\,d + 660$	$59\,d^2 + 1068\,d + 994$	n/a

Boolean case. For the Boolean case, it is worth noticing that packing and unpacking the bitslice registers for the parallelization of the ISW-ANDs implies a linear overhead in d. For $d \in [\![2, 20]\!]$, this overhead is between 4% and 6% of the overall s-box computations for $n = 8$, and between 7% and 11% for $n = 4$ (and this ratio is asymptotically negligible). For $d = 2$, the overhead slightly exceeds the gain, but for every $d \geq 3$, parallelizing the ISW-ANDs always results in an overall gain of performances.

We observe that the Boolean case is asymptotically faster than the optimized implementations of CRV and AD methods (3.6 times faster for $n = 8$ and 5.7 times faster for $n = 4$). However, we also see that the linear coefficient is significantly greater for the Boolean case, which comes from the computation of the linear combinations in input of the ISW-ANDs (*i.e.* the sharings $[\vec{r}_i]$ and $[\vec{s}_i]$). As an illustration, Figures **??** and **??** plots the obtained timings with respect to d. We see that for $n = 4$, our implementation is always faster than the optimized AD and CRV. On the other hand, for $n = 8$, our implementation is slightly slower for $d \leq 8$. We stress that our implementations could probably be improved by optimizing the computation of the linear combinations.

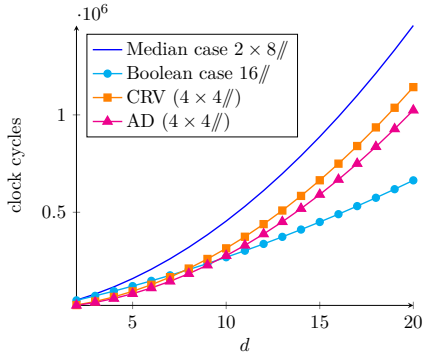


Figure 5.8: Timings for $n = 8$.

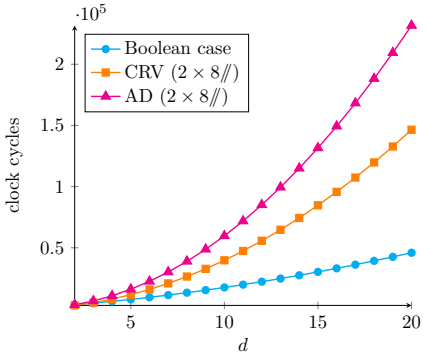


Figure 5.9: Timings for $n = 4$.

The RAM consumption and code size of our implementations are given in Table **??**. We believe these memory requirements to be affordable for not-too-constrained embedded devices. This is especially significant for $n = 8$ where CRV and AD needs a high amount of storage for the lookup tables of the linearized polynomials. On the other hand, we observe a big gap regarding the RAM consumption. The Boolean method is indeed more consuming in RAM because of all the $[\vec{t}_i]$ sharings that must be stored while such a large basis is not required

for the CRV and AD methods, and because of some optimizations in the computation of the linear combinations.

Table 5.19: Code sizes and RAM consumptions.

	CRV	**AD**	**Boolean case**	**Median case**
$\lambda = 8$	$4 \times 4/\!/\!/$ s-boxes	$4 \times 4/\!/\!/$ s-boxes	$16/\!/\!/$ s-boxes	$2 \times 8/\!/\!/$
Code size	27.5 KB	11.2 KB	$4.6KB$	8.7 KB
RAM	$80d$ bytes	$188d$ bytes	$644d$ bytes	$92d$ bytes
$\lambda = 4$	$2 \times 8/\!/\!/$ s-boxes	$2 \times 8/\!/\!/$ s-boxes	$16/\!/\!/$ s-boxes	n/a
Code size	3.2 KB	2.6 KB	2.2 KB	n/a
RAM	$24d$ bytes	$64d$ bytes	$132d$ bytes	n/a

Median case. These results show that the median case is slightly less efficient in terms of timings. However, it provides an interesting tradeoff in terms of memory consumption. Indeed the bitslice implementation has the drawback of being quite consuming in terms of RAM (with $644d$ bytes needed) and the CRV-based implementation has the drawback of having an important code size (27.5 KB) which is mainly due to the *half-table* multiplication and the tabulation linearized polynomials over $\mathbb{F}_{256}$. In terms of code size, the median case is the best. The median case offers a nice alternative when both RAM and code size are constrained. It also needs the same amount of randomness than the CRV decomposition and more than twice less than the bitslice decomposition.

Additionally, implementations based on our medium case decomposition might provide further interesting tradeoffs on smaller (8-bit or 16-bit) architectures where bitslice would be slowed down and where the optimized CRV-based implementation might be too consuming in terms of code size.

5.5 Structured S-boxes

5.5.1 Polynomial representation

As detailed in Chapter **??**, there is exist two main approach to decompose the AES s-box, namely the Rivain-Prouff method and the Kim-Hong-Lim method. For PRESENT, the usual solution is to evaluate the $F \circ G$ representation.

5.5.2 Bitslice representation

5.5.2.1 Secure Bitsliced AES S-box

For the AES s-box, we based our work on the compact representation proposed by Boyar *et al.* in [**JC:BoyMatPer13**]. Their circuit is obtained by applying logic minimization techniques to the tower-field representation of Canright [**CHES:Canright05**]. It involves 115 logic gates including 32 logical AND. The circuit is composed of three parts: the *top linear transformation* involving 23 XOR gates and mapping the 8 s-box input bits x_0, x_1, ..., x_7 to 23 new bits x_7, y_1, y_2, ..., y_{21}; the *middle non-linear transformation* involving 30 XOR gates and 32 AND gates and mapping the previous 23 bits to 18 new bits z_0, z_1, ..., z_{17}; and the *bottom linear transformation* involving 26 XOR gates and 4 XNOR gates and mapping the 18 previous bits to the 8 s-box output bits $s_0, s_1, \ldots, s_7$. In particular, this circuit improves

the usual count of 34 AND gates involved in previous tower-field representations of the AES s-box.

Using this circuit, we can perform the 16 s-box computations of an AES round in parallel. That is, instead of having 8 input bits mapped to 8 output bits, we have 8 (shared) input 16-bit words $X_0, X_1, \ldots, X_7$ mapped to 8 (shared) output 16-bit words $S_1, S_2, \ldots, S_8$. Each word X_i (resp. S_i) contains the ith bits input bit (resp. output bit) of the 16 s-boxes. Each XOR gate and AND gate of the original circuit is then replaced by the corresponding (shared) bitwise instruction between two 16-bit words.

Parallelizing AND gates. For our masked bitslice implementation, a sound complexity unit is one call to the ISW-AND since this is the only nonlinear operation, *i.e.* the only operation with quadratic complexity in d (compared to other operations that are linear in d). In a straightforward bitslice implementation of the considered circuit, we would then have a complexity of 32 ISW-AND. This is suboptimal since each of these ISW-AND is applied to 16-bit words whereas it can operates on 32-bit words. Our main optimization is hence to group together pairs of ISW-AND in order to replace them by a single ISW-AND with fully filled input registers. This optimization hence requires to be able to group AND gates by pair that can be computed in parallel. To do so, we reordered the gates in the middle non-linear transformation of the Boyar *et al.* circuit, while keeping the computation consistent. We were able to fully parallelize the AND gates, hence dropping our bitslice complexity from 32 down to 16 ISW-AND. We thus get a parallel computation of the 16 AES s-boxes of one round with a complexity of 16 ISW-AND, that is one single ISW-AND per s-box. Since an ISW-AND is (significantly) faster than any ISW multiplication, our masked bitslice implementation breaks through the barrier of one ISW field multiplication per s-box. Our reordered version of the Boyar *et al.* circuit is described in Figure **??**. It can be checked that every two consecutive AND gates can be performed in parallel.

Remark 5.5.1. *The Boyar* et al. *circuit involves many intermediate variables (denoted by t_i) that are sometimes needed multiple times. This strongly impact the performances in practice by requiring several loads and stores, which –as mentioned earlier– are the most expensive ARM instructions, and further implies a memory overhead. In order to minimize this impact, we also reordered the gates in the linear transformations so that all the intermediate variables can be kept in registers. This is only possible for linear transformations since they are applied on each share independently. On the other hand, the shares of the intermediate variables in input of an ISW-AND must be stored in memory as they would not fit altogether in the registers.*

5.5.3 Secure Bitsliced PRESENT S-box

For our masked bitsliced implementation of the PRESENT s-box, we used the compact representation given by Courtois *et al.* in [**EPRINT:CouHulMou11**], which was obtained from Boyar *et al.* 's logic minimization techniques improved by involving OR gates. This circuit is composed of 4 nonlinear gates (2 AND and 2 OR) and 9 linear gates (8 XOR and 1 XNOR).

Remark 5.5.2. *An ISW-OR can be computed from an ISW-AND by using De Morgan's law, that is $a \vee b = \overline{\bar{a} \cdot \bar{b}}$, and since the negation in the masking world simply consists in complementing a single of the d shares (which can be efficiently done with a single instruction). One can also easily get an ISW-NAND and an ISW-NOR.*

– top linear transformation –

$$
\begin{array}{llll}
y_{14} = x_3 \oplus x_5 & y_1 = t_0 \oplus x_7 & y_{15} = t_1 \oplus x_5 & y_{17} = y_{10} \oplus y_{11} \\
y_{13} = x_0 \oplus x_6 & y_4 = y_1 \oplus x_3 & y_{20} = t_1 \oplus x_1 & y_{19} = y_{10} \oplus y_8 \\
y_{12} = y_{13} \oplus y_{14} & y_2 = y_1 \oplus x_0 & y_6 = y_{15} \oplus x_7 & y_{16} = t_0 \oplus y_{11} \\
y_9 = x_0 \oplus x_3 & y_5 = y_1 \oplus x_6 & y_{10} = y_{15} \oplus t_0 & y_{21} = y_{13} \oplus y_{16} \\
y_8 = x_0 \oplus x_5 & t_1 = x_4 \oplus y_{12} & y_{11} = y_{20} \oplus y_9 & y_{18} = x_0 \oplus y_{16} \\
t_0 = x_1 \oplus x_2 & y_3 = y_5 \oplus y_8 & y_7 = x_7 \oplus y_{11} &
\end{array}
$$

– middle non-linear transformation –

$$
\begin{array}{llll}
t_2 = y_{12} \cdot y_{15} & t_{23} = t_{19} \oplus y_{21} & t_{34} = t_{23} \oplus t_{33} & z_9 = t_{44} \cdot y_{12} \\
t_3 = y_3 \cdot y_6 & t_{15} = y_8 \cdot y_{10} & t_{35} = t_{27} \oplus t_{33} & z_{10} = t_{37} \cdot y_3 \\
t_5 = y_4 \cdot x_7 & t_{26} = t_{21} \cdot t_{23} & t_{42} = t_{29} \oplus t_{33} & z_4 = t_{40} \cdot y_1 \\
t_7 = y_{13} \cdot y_{16} & t_{16} = t_{15} \oplus t_{12} & z_{14} = t_{29} \cdot y_2 & z_6 = t_{42} \cdot y_{11} \\
t_8 = y_5 \cdot y_1 & t_{18} = t_6 \oplus t_{16} & t_{36} = t_{24} \cdot t_{35} & z_{13} = t_{40} \cdot y_5 \\
t_{10} = y_2 \cdot y_7 & t_{20} = t_{11} \oplus t_{16} & t_{37} = t_{36} \oplus t_{34} & z_{15} = t_{42} \cdot y_9 \\
t_{12} = y_9 \cdot y_{11} & t_{24} = t_{20} \oplus y_{18} & t_{38} = t_{27} \oplus t_{36} & z_7 = t_{45} \cdot y_{17} \\
t_{13} = y_{14} \cdot y_{17} & t_{30} = t_{23} \oplus t_{24} & t_{39} = t_{29} \cdot t_{38} & z_8 = t_{41} \cdot y_{10} \\
t_4 = t_3 \oplus t_2 & t_{22} = t_{18} \oplus y_{19} & z_5 = t_{29} \cdot y_7 & z_{16} = t_{45} \cdot y_{14} \\
t_6 = t_5 \oplus t_2 & t_{25} = t_{21} \oplus t_{22} & t_{44} = t_{33} \oplus t_{37} & z_{17} = t_{41} \cdot y_8 \\
t_9 = t_8 \oplus t_7 & t_{27} = t_{24} \oplus t_{26} & t_{40} = t_{25} \oplus t_{39} & z_{11} = t_{33} \cdot y_4 \\
t_{11} = t_{10} \oplus t_7 & t_{31} = t_{22} \oplus t_{26} & t_{41} = t_{40} \oplus t_{37} & z_{12} = t_{43} \cdot y_{13} \\
t_{14} = t_{13} \oplus t_{12} & t_{28} = t_{25} \cdot t_{27} & t_{43} = t_{29} \oplus t_{40} & z_2 = t_{33} \cdot x_7 \\
t_{17} = t_4 \oplus t_{14} & t_{32} = t_{31} \cdot t_{30} & t_{45} = t_{42} \oplus t_{41} & z_3 = t_{43} \cdot y_{16} \\
t_{19} = t_9 \oplus t_{14} & t_{29} = t_{28} \oplus t_{22} & z_0 = t_{44} \cdot y_{15} & \\
t_{21} = t_{17} \oplus y_{20} & t_{33} = t_{32} \oplus t_{24} & z_1 = t_{37} \cdot y_6 &
\end{array}
$$

– bottom linear transformation –

$$
\begin{array}{llll}
t_{46} = z_{15} \oplus z_{16} & t_{49} = z_9 \oplus z_{10} & t_{61} = z_{14} \oplus t_{57} & t_{48} = z_5 \oplus z_{13} \\
t_{55} = z_{16} \oplus z_{17} & t_{63} = t_{49} \oplus t_{58} & t_{65} = t_{61} \oplus t_{62} & t_{56} = z_{12} \oplus t_{48} \\
t_{52} = z_7 \oplus z_8 & t_{66} = z_1 \oplus t_{63} & s_0 = t_{59} \oplus t_{63} & s_3 = t_{53} \oplus t_{66} \\
t_{54} = z_6 \oplus z_7 & t_{62} = t_{52} \oplus t_{58} & t_{51} = z_2 \oplus z_5 & s_1 = \overline{t_{64} \oplus s_3} \\
t_{58} = z_4 \oplus t_{46} & t_{53} = z_0 \oplus z_3 & s_4 = t_{51} \oplus t_{66} & s_6 = \overline{t_{56} \oplus t_{62}} \\
t_{59} = z_3 \oplus t_{54} & t_{50} = z_2 \oplus z_{12} & s_5 = t_{47} \oplus t_{65} & s_7 = \overline{t_{48} \oplus t_{60}} \\
t_{64} = z_4 \oplus t_{59} & t_{57} = t_{50} \oplus t_{53} & t_{67} = t_{64} \oplus t_{65} & \\
t_{47} = z_{10} \oplus z_{11} & t_{60} = t_{46} \oplus t_{57} & s_2 = \overline{t_{55} \oplus t_{67}} &
\end{array}
$$

Figure 5.10: AES s-box circuit for efficient bitslice implementation.

PRESENT has 16 parallel s-box computations per round, as AES. We hence get a bitsliced implementation with 16-bit words that we want to group for the calls to ISW-AND. However for the chosen circuit, we could not fully parallelize the nonlinear gates because of the dependency between three of them. We could however group the two OR gates after a slight

reordering of the operations. We hence obtain a masked bitsliced implementation computing the 16 PRESENT s-boxes in parallel with 3 calls to ISW-AND. Our reordered version of the circuit is depicted in Figure **??**. We clearly see that the two successive OR gates can be computed in parallel. For the sake of security, we also refresh one of the two input sharings in the 3 calls to ISW-AND. As for the bitslice AES s-box, the implied overhead is manageable.

$$
\begin{array}{lll|lll}
t_1 & = & x_2 \oplus x_1 & t_7 & = & \overline{x_3 \oplus t_5} \\
t_2 & = & x_1 \cdot t_2 & t_8 & = & x_3 \vee t_5 \\
t_3 & = & x_0 \oplus t_2 & t_9 & = & t_7 \vee t_6 \\
y_3 & = & x_3 \oplus t_3 & y_2 & = & t_6 \oplus t_8 \\
t_4 & = & t_1 \cdot t_3 & y_0 & = & y_2 \oplus t_7 \\
t_5 & = & t_4 \oplus x_1 & y_1 & = & t_3 \oplus t_9 \\
t_6 & = & t_1 \oplus y_3 & & & \\
\end{array}
$$

Figure 5.11: PRESENT s-box circuit for efficient bitslice implementation.

5.5.3.1 Implementation and Performance

Table **??** gives the performance obtained for our masked bitslice implementations of the AES and PRESENT s-boxes. For comparison, we also recall the performances of the fastest polynomial methods for AES and PRESENT (*i.e.* parallel versions of KHL and $F \circ G$) as well as the fastest generic methods for $n = 8$ and $n = 4$ (*i.e.* parallel versions of the algebraic decomposition method for $n = 8$ and CRV for $n = 4$). The timings are further plotted in Figures **??** and **??** for the sake of illustration.

Table 5.20: Performance of secure bitsliced s-boxes and comparison.

	Clock cycles	Code size
Bitslice AES s-box (16$/\!/$)	$288\,d^2 + 1097\,d + 1080$	3.1 KB
KHL (2$\times$ 8$/\!/$)	$764\,d^2 + 512\,d + 538$	4 KB
Alg. decomp. for $n = 8$ (4$\times$ 4$/\!/$)	$2376\,d^2 + 3812\,d + 5112$	10.3 KB
Bitslice PRESENT s-box (16$/\!/$)	$61.5\,d^2 + 178.5\,d + 193$	752 B
$F \circ G$ (2$\times$ 8$/\!/$)	$376\,d^2 - 168\,d + 204$	1.4 KB
CRV for $n = 4$ (2$\times$ 8$/\!/$)	$295\,d^2 + 667\,d + 358$	2.1 KB

These results clearly demonstrate the superiority of the bitslicing approach. Our masked bitsliced implementations of the AES and PRESENT s-boxes are significantly faster than state-of-the art polynomial methods finely tuned at the assembly level.

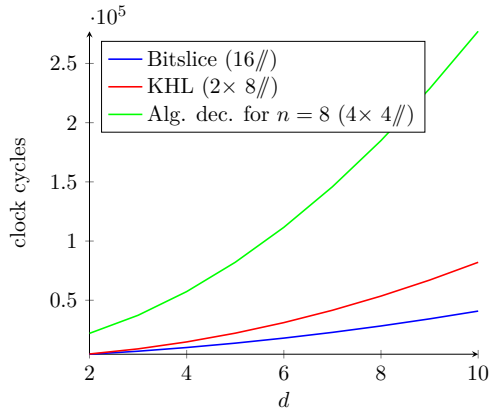


Figure 5.12: Timings for 16 AES s-boxes.

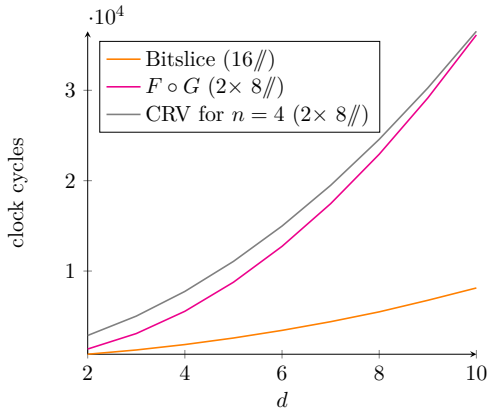


Figure 5.13: Timings for 16 PRESENT s-boxes.

5.6 Cipher Implementations

This section finally describes masked implementations of the full PRESENT and AES blockciphers. These blockciphers are so-called *substitution-permutation networks*, where each round is composed of a key addition layer, a nonlinear layer and a linear diffusion layer. For both blockciphers, the nonlinear layer consists in the parallel application of 16 s-boxes. The AES works on a 128-bit state (which divides into sixteen 8-bit s-box inputs) whereas PRESENT works on a 64-bit state (which divides into sixteen 4-bit s-box inputs). For detailed specifications of these blockciphers, the reader is referred to [**FIPS197**] and [**CHES:BKLPPR07**]. For both blockciphers, we follow two implementation strategies: the standard one (with parallel polynomial methods for s-boxes) and the bitsliced one (with bitsliced s-box masking).

For the sake of efficiency, we assume that the key is already expanded, and for the sake of security we assume that each round key is stored in (non-volatile) memory under a shared form. In other words, we do not perform a masked key schedule. Our implementations start by masking the input plaintext with $d-1$ random m-bit strings (where m is the block cipher bit-size) and store the d resulting shares in memory. These d shares then compose the sharing of the block cipher state that is updated by the masked computation of each round. When all the rounds have been processed, the output ciphertext is recovered by adding all the output shares of the state. For the bitsliced implementations, the translation from standard to bitsliced representation is performed before the initial masking so that it is done only once. Similarly, the translation back from the bitsliced to the standard representation is performed a single time after unmasking.

The secure s-box implementations are done as described in previous sections. It hence remains to deal with the key addition and the linear layers. These steps are applied to each share of the state independently. The key-addition step simply consists in adding each share of the round key to one share of the state. The linear layer implementations are described hereafter.

5.6.1 Standard AES linear layer.

We based our implementation on a classical optimized version of the unmasked block cipher. We use a transposed representation of the state matrix in order to store each row in a 32-bit register. The MixColumns can then be processed for each column simultaneously by working with row registers, and the ShiftRows simply consists of three rotate instructions.

5.6.2 Bitsliced AES linear layer.

The MixColumns step can be efficiently computed in bitslice representation by transforming the state matrix $(b_{i,j})_{i,j}$ into a new state matrix:

$$c_{i,j} = \mathsf{xtimes}(b_{i,j} \oplus b_{i+1,j}) \oplus b_{i+1,j} \oplus b_{i+2,j} \oplus b_{i+3,j} \ , \tag{5.14}$$

where xtimes denotes the multiplication by x over $\mathcal{F}_{2^8} \equiv \mathcal{F}_2[x]/p(x)$ with $p(x) = x^8 + x^4 + x^3 + x + 1$. The xtimes has a simple Boolean expression, which is:

$$(b_7, b_6, \ldots, b_0)_2 \xrightarrow{\mathsf{xtimes}} (b_6, b_5, b_4, b_3 \oplus b_7, b_2 \oplus b_7, b_1, b_0 \oplus b_7, b_7)_2 \tag{5.15}$$

Let W_k denotes the word corresponding to the kth bits of the state bytes $b_{i,j}$, and let $W_k^{(i,j)}$ denotes the $(4 \cdot i + j)$th bit of W_k (that is the kth bit of the state byte $b_{i,j}$). The words Z_k in output of the bitslice MixColumns satisfy

$$Z_k^{(i,j)} = X_k^{(i,j)} \oplus W_k^{(i+1,j)} \oplus W_k^{(i+2,j)} \oplus W_k^{(i+3,j)} \ , \tag{5.16}$$

where $X_k^{(i,j)}$ is the kth bit of the output of $\mathsf{xtimes}(b_{i,j} \oplus b_{i+1,j})$. This gives

$$Z_k = X_k \oplus (W_k \lll 4) \oplus (W_k \lll 8) \oplus (W_k \lll 12) \ , \tag{5.17}$$

where $\lll$ denotes the rotate left operator on 16 bits. Following the Boolean expression of xtimes, the word X_k satisfies $X_k = Y_{k-1}$ if $k \in \{7, 6, 5, 2\}$, $X_k = Y_{k-1} \oplus Y_7$ if $k \in \{4, 3, 1\}$, and $X_0 = Y_7$, with $Y_k = W_k \oplus (W_k \lll 4)$.

The above equation can be efficiently evaluated in ARM assembly, taking advantage of the barrel shifter. The only subtlety is that the above rotate operations are on 16 bits whereas the ARM rotation works on 32 bits. But this can be simply circumvented by using left shift instead of left rotate with a final reduction of the exceeding bits. The obtained implementation takes 43 one-cycle instructions for the overall bitslice MixColumns.

The ShiftRows is the most expensive step of the AES linear layer in bitslice representation. It must be applied on the bits of each vector W_k (since each nibble of W_k correspond to a different row of the state). Each row can be treated using 6 ARM instructions (6 clock cycles) as shown in the following code:

```
;; R3 =b0 b1 b2 b3 ... b16
AND     R0, R3, #0xF0        ;; R0 = b5 b6 b7 b8
AND     R2, R0, #0x80        ;; R2 = b8
BIC     R0, R0, #0x80        ;; R0 = b5 b6 b7 0
EOR     R0, R0, R2, LSR #4   ;; R0 = b8 b5 b6 b7
BIC     R3, #0xF0            ;; R3 = b1...b4 0000 b9...b16
EOR     R3, R0, LSL #1       ;; R3 = b1...b4b8b5b6b7...b16
```

This must be done for 3 rows, which makes 18 clock cycles per word W_k, that is a total of $8 \times 18 = 144$ clock cycles.

5.6.3 PRESENT linear layer.

The linear layer of PRESENT, called the pLayer, consists in a permutation of the 64 bits of the state. In both representations, our implementation of this step use the straightforward approach where each bit is taken at a given position i in a source register and put at given position j in a destination register. Such a basic operation can be done in two ARM instructions (2 clock cycles), with a register `$one` previously set to 1, as follows:

```
AND $tmp, $src, $one, LSR #i
EOR $dst, $tmp, LSL #j
```

In both representations, our pLayer implementation is hence composed of 64×2 instructions as above, plus the initialization of destination registers, for a total of 130 instructions for the standard representation and 132 for the bitslice representation.[5]

5.6.4 Translation to and from bitslice format.

For both blockciphers the bitslice translations (forward and backward) can be seen as bit permutations. They are hence implemented using a similar approach as for the pLayer of PRESENT. This requires around $2 \times 64 = 128$ cycles for PRESENT and $2 \times 128 = 256$ clock cycles for AES. Further note that in the case of PRESENT, the forward bitslice translation is exactly the pLayer transformation, which allows some code saving.

5.6.5 Performance

In our standard implementation of AES, we used the parallel versions of KHL and RP (with ISW-EL) for the s-box. For the standard implementation of PRESENT, we used the parallel versions of the $F \circ G$ method and of the CRV method. The obtained performance are summarized in Table **??**. The timings are further plotted in Figures **??** and **??** for illustration.

These results clearly confirm the superiority of the bitslice implementations in our context. The bitslice AES implementation asymptotically takes 38% of the timings of the standard AES implementation using the best parallel polynomial method for the s-box (namely KHL). This ratio reaches 18% for PRESENT (compared to the $F \circ G$ method). It is also interesting to observe that PRESENT is slower than AES for standard masked implementations whereas it is faster for masked bitslice implementations. In the latter case, a PRESENT computation asymptotically amounts to 0.58 AES computation. This ratio directly results from the number of calls to ISW-AND which is $10 \times 16 = 160$ for AES (16 per round) and $31 \times 3 = 93$ for PRESENT (3 per round).

In order to illustrate the obtained performance in practice, Table **??** gives the corresponding timings in milliseconds for a clock frequency of 60 MHz. For a masking order of 10, our bitsliced implementations only take a few milliseconds.

[5] We do not count the initialization of the register `$one` which is done once and not for each share.

Table 5.21: Performances of masked blockciphers implementation.

	Clock cycles	Code (KB)	Random (bytes)
Bitslice AES	$2880\,d^2 + 13675\,d + 11872$	7.5	$320d(d-1)$
Standard AES (KHL $/\!/$)	$7640\,d^2 + 6229\,d + 6311$	4.8	$560d(d-1)$
Standard (AES RP-HT $/\!/$)	$9580\,d^2 + 5129\,d + 7621$	12.4	$400d(d-1)$
Standard (AES RP-EL $/\!/$)	$10301\,d^2 + 6561\,d + 7633$	4.1	$400d(d-1)$
Bitslice PRESENT	$1906.5\,d^2 + 10972.5\,d + 7712$	2.2	$372d(d-1)$
Standard PRESENT ($F \circ G$ $/\!/$)	$11656\,d^2 + 341\,d + 9081$	1.9	$496d(d-1)$
Standard PRESENT (CRV $/\!/$)	$9145\,d^2 + 45911\,d + 11098$	2.6	$248d(d-1)$

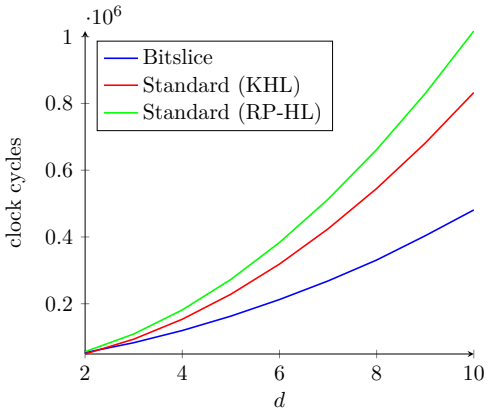


Figure 5.14: Timings of masked AES.

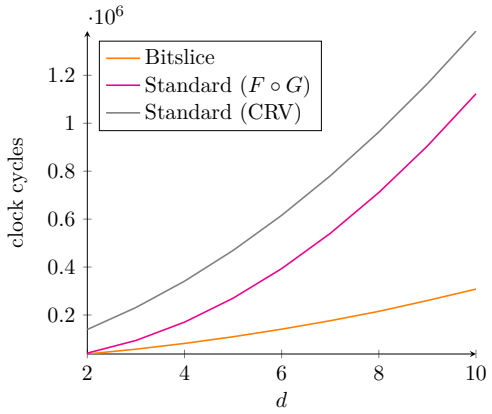


Figure 5.15: Timings of masked PRESENT.

Table 5.22: Timings for masked bistlice AES and PRESENT with a 60 Mhz clock.

	$d = 2$	$d = 3$	$d = 4$	$d = 5$	$d = 10$
Bitslice AES	0.89 ms	1.39 ms	1.99 ms	2.7 ms	8.01 ms
Bitslice PRESENT	0.62 ms	0.96 ms	1.35 ms	1.82 ms	5.13 ms

Chapter 6

Conclusion and Open Questions

6.1 Conclusion

In this thesis, we have studied efficient schemes to protect block ciphers against physical attacks. As the evaluation of the s-boxes is the main bottleneck, the first challenge was to find sound representation of the s-boxes. For this purpose, we studied the s-box as a polynomial over a finite field and proposed a generalization of an existing method to decompose any polynomial over any finite field. As opposed to solutions using SAT solvers, our decomposition technique can work for any s-box size. The main advantage of this generalized method is its efficiency in terms of multiplicative complexity and its flexibility with respect to its instantiations. Indeed, our implementation results showed interesting trade-offs between timing and memory costs. In fact, for small masking orders working on a full field allows to obtain fast implementation results but with some costs on code size, whereas median field allows to obtain slower implementation but with a tight memory usage. However, as the masking order grows, the Boolean field quickly becomes the best solution thanks to the bitslicing strategy.

Then, we analyzed the security of masking schemes in a theoretical framework and more specifically the composable security of secure implementations using ISW multiplication gadgets. By a security reduction with the use of game based techniques, we were able to reformulate the probing security of those particular circuits into a problem of linear algebra. From the linear algebra, we identified a criterion to determine whether or not a tight shared circuit is probing secure for any adversary. From this criterion, we have developed a formal verification tool that can determine, given the description of a secure circuit, if a probing attack is possible. For the AES s-box Boolean circuit, our verification tool determined that no refresh gadgets are required, as opposed to other verification tools that insert refresh gadgets after each multiplication. This led to a reduction of the randomness consumption by a factor of two in the implementation of the AES s-box with a bitslicing strategy.

Finally, we have implemented in ARM assembly most of the state-of-the-art masking schemes and different instantiations of our decomposition framework. We followed a bottom-up approach that allowed us to obtain optimized implementations for the building blocks of the construction of protected block ciphers. We studied different techniques to perform secure multiplications with various trade-offs between efficiency, randomness consumption and practical security. Finally, we have studied the use of the bitslice strategy at the s-box level which allowed us to outperform all known polynomial-based solutions. Moreover, for the AES s-box the bitsliced implementation requires little randomness thanks to our verification tool. The numerous implementations developed during this thesis have been made available online for public use [**github**].

6.2 Open Questions

Here, we give a few open problems and directions that might be interesting for future work.

Question 6.1. *How could homomorphic encryption benefit from our generic decomposition tool?*

In homomorphic encryption, the problem of evaluating efficiently the s-box is similar to the one in masking schemes. In fact, to lower the impact of the growth of the noise in homomorphic encryption, one needs to find encryption schemes that have a minimal multiplicative depth. Therefore, one may want to adapt our generic decomposition method to target not only the multiplicative complexity but also the multiplicative depth and hence get efficient implementation of homomorphic encryption. Specifically, since the notion of parallel multiplicative complexity introduced in Chapter **??** is equivalent to the notion of multiplicative depth, as soon as the architecture size is big enough (see Chapter **??**), our generic framework could be adapted to find representations that lower the multiplicative depth.

Question 6.2. *Could we optimize SAT solver techniques to work on bigger s-boxes?*

One major drawback of our generic decomposition framework is that it is a heuristic method. To get the tightest decomposition of a function with respect to the number of multiplications, the ideal solution is to use SAT solvers to find the exact representation of the s-box with the optimal multiplicative complexity. However, as the size of the s-box grows, the complexity of SAT solvers explodes. Hence, one interesting line of work would be to see if it is possible to reduce the number of equations that the SAT solvers try to solve to find representations in a reasonable amount of time.

Question 6.3. *Is our verification tool extendable to more generic problems?*

Currently our verification tool introduced in Chapter **??** only works on Boolean circuit composed of ISW multiplications. It would be of interest to extend the results to more general circuits (*e.g.* arithmetic circuits), but also to more generic secure multiplication (namely multiplications that do not necessarily manipulate all the cross products). Moreover, we only applied our tool to the example given in Chapter **??**. Applying the tool to other existing solutions such as generic decomposition techniques to reduce the randomness can hence be a interesting line of work.

List of Illustrations

Figures

Tables

Résumé

Depuis leur introduction à la fin des années 1990, les attaques par canaux auxiliaires sont considérées comme une menace majeure contre les implémentations cryptographiques. Parmi les stratégies de protection existantes, une des plus utilisées est le masquage d'ordre supérieur. Elle consiste à séparer chaque variable interne du calcul cryptographique en plusieurs variables aléatoires. Néanmoins, l'utilisation de cette protection entraîne des pertes d'efficacité considérables, la rendant souvent impraticable pour des produits industriels.

Cette thèse a pour objectif de réduire l'écart entre les solutions théoriques, prouvées sûres, et les implémentations efficaces déployables sur des systèmes embarqués. Plus particulièrement, nous nous intéressons à la protection de chiffrement par bloc tel que l'AES, dont l'enjeu principal revient à protéger les boîtes-s avec un surcoût minimal.

Nous essayons d'abord de trouver des représentations mathématiques optimales pour l'évaluation des boîtes-s en minimisant le nombre de multiplications (un paramètre déterminant pour l'efficacité du masquage, mais aussi pour le chiffrement homomorphe). Pour cela, nous définissons une méthode générique pour décomposer n'importe quelle fonction sur un corps fini avec une complexité multiplicative faible. Ces représentations peuvent alors être évaluées efficacement avec du masquage d'ordre supérieur. La flexibilité de la méthode de décomposition permet également de l'ajuster facilement selon les nécessités du développeur.

Nous proposons ensuite une méthode formelle pour déterminer la sécurité d'un circuit évaluant des schémas de masquages. Cette technique permet notamment de déterminer de manière exacte si une attaque est possible sur un circuit protégé ou non. Par rapport aux autres outils existants, son temps de réponse n'explose pas en la taille du circuit, ce qui permet d'obtenir une preuve de sécurité quelque soit l'ordre de masquage employé. De plus, elle permet de diminuer de manière stricte l'emploi d'outils coûteux en aléas, requis pour renforcer la sécurité des opérations de masquages.

Enfin, nous présentons des résultats d'implémentation en proposant des optimisations tant sur le plan algorithmique que sur celui de la programmation. Nous utilisons notamment une stratégie d'implémentation *bitslice* pour évaluer les boîtes-s en parallèle. Cette stratégie nous permet d'atteindre des records de rapidité pour des implémentations d'ordres élevés. Les différents codes sont développés et optimisés en assembleur ARM, un des langages les plus répandus dans les systèmes embarqués tels que les cartes à puces et les téléphones mobiles. Ces implémentations sont, en outre, disponibles en ligne pour une utilisation publique.

Mots Clés

cryptographie, attaques par canaux auxiliaires, implémentation efficace, contremesures, sécurité prouvée, assembleur.

Abstract

Since their introduction at the end of the 1990s, side-channel attacks are considered to be a major threat against cryptographic implementations. Higher-order masking is considered to be one the most popular existing protection strategies. It consists in separating each internal variable in the cryptographic computation into several random variables. However, the use of this type of protection entails a considerable efficiency loss, making it unusable for industrial solutions.

The goal of this thesis is to reduce the gap between theoretical solutions, proven secure, and efficient implementations that can be deployed on embedded systems. More precisely, I am analysing the protection of block ciphers such as the AES encryption scheme, where the main issue is to protect the s-boxes with minimal overhead in costs.

I have tried, first, to find optimal mathematical representations in order to evaluate the s-boxes while minimizing the number of multiplications (a decisive parameter for masking schemes, but also for homomorphic encryption). For this purpose, I have defined a generic method to decompose any function on any finite field with a low multiplicative complexity. These representations can, then, be efficiently evaluated with higher-order masking. The flexibility of the decomposition technique allows also easy adjusting to the developer's needs.

Secondly, I have proposed a formal method for measuring the security of circuits evaluating masking schemes. This technique allows to define with exact precision whether an attack on a protected circuit is feasible or not. Unlike other tools, its response time is not exponential in the circuit size, making it possible to obtain a security proof regardless of the masking order used. Furthermore, this method can strictly reduce the use of costly tools in randomness required for reinforcing the security of masking operations.

Finally, we present the implementation results with optimizations both on algorithmic and programming fronts. We particularly employ a *bitslice* implementation strategy for evaluating the s-boxes in parallel. This strategy leads to speed record for implementations protected at high order. The different codes are developed and optimized under ARM assembly, one of the most popular programming language in embedded systems such as smart cards and mobile phones. These implementations are also available online for public use.

Keywords

cryptography, side-channel attacks, efficient implementation, countermeasures, provable security, assembly.