Cryptographic primitives adapted to connected car requirements

Etienne Tehrani

INSTITUT POLYTECHNIQUE DE PARIS

NNT : 2022IPPAT028

Thèse de doctorat

Cryptographic Primitives Adapted to Connected Car Requirements

Thèse de doctorat de l'Institut Polytechnique de Paris
préparée à Télécom Paris

École doctorale n°626 École doctorale de l'Institut Polytechnique de Paris (EDIPP)
Spécialité de doctorat: Information, Communications, Électronique

Thèse présentée et soutenue à Palaiseau, le 8 Juillet 2022, par

Etienne Tehrani

Composition du Jury :

Lilian Bossuet Professur, Université Jean Monnet (Laboratoire Hubert Curien)	Président
Nele Mentens Professeur Associée, Katholieke Universiteit Leuven (COSIC)	Rapporteuse
Guy Gogniat Professeur, Université Bretagne Sud (STICC)	Rapporteur
Lilian Bossuet Professur, Université Jean Monnet (Laboratoire Hubert Curien)	Examinateur
Karine Heydemann Maîtresse de Conférence, Sorbonne Université (LIP6)	Examinatrice
Sylvain Guilley Professeur, Télécome Paris (LTCI)	Examinateur
Jean-Luc Danger Directeur d'études, Télécom Paris (LTCI)	Directeur de thèse
Tarik Graba Maître de conférences, Télécom Paris (LTCI)	Co-directeur de thèse

Abstract

Communications are one of the key functions in future vehicles. They are developing at high speed and require a high degree of security. Whether intra- or extra-vehicular, reliable communications are imperative in order to ensure rapid decision-making according to the environment. Nevertheless, it can be intercepted and even modified by potential attackers and therefore requires the use of protections. Cryptography is an obvious answer to secure communications, but the algorithms used today seem inadequate in terms of complexity and latency, especially to satisfy the physical and economic constraints of large-scale embedded systems. Although the automotive industry generally relies on standards, it is not uncommon for some players to prefer the use of ad hoc technologies that are more adapted to cost, performance and safety constraints. As long as the communication is internal, a standard is not necessary, but with inter-vehicle communications, the absence of a standard imposes a great versatility on the vehicle to ensure the compatibility of the communications, while respecting the constraints of complexity and latency. A second emerging problem specific to embedded systems is the protection against attacks, especially physical attacks due to the fact that the system can be accessed by the user. These attacks are formidable because they can bypass the mathematical protection of cryptographic algorithms. This is the case of the attacks by side-channels which exploit the leaks related to the activity of calculation by the means of consumption or electromagnetic radiation. It is therefore essential to take this aspect into account in the implementation of encryption algorithms to obtain a high level of security.

Within the context of the "Connected Cars and CyberSecurity (C3S)" Chair, the main objectives of the thesis are to study the feasibility of implementing a wide variety of symmetric lightweight encryption algorithms and their protection. More precisely, the goal is to implement several families of algorithms on a hardware that is surface constrained and that would have good performance in terms of physical security and latency. An optimal solution is to have an agile implementation, able to quickly execute different lightweight encryption algorithms, using few resources and guaranteeing protection against physical attacks. The implementation can be fully hardware-based or a dedicated processor implementation taking advantage of the presence of on-board Electronic Control Units. To ensure the best implementation compromise, this last architecture can start from a modification of the instruction set of a RISC processor to satisfy the agility property of light cryptography algorithms. This agility guarantees a great flexibility, a common protection base for the algorithms, for a cost comparable to that of a standard microcontroller.

We have studied many encryption algorithms and have proposed a first approach with a fully hardware architecture and a second approach with a dedicated processor in order to efficiently implement Lightweight Cryptography in a constrained embedded system. Our main contributions can be summarized as follows:

- Classification of Lightweight Encryption Algorithms by Block
- Configurable hardware implementation dedicated to the acceleration of the execution of Lightweight Block Ciphers of the SPN type
- RISC-V hardware extension dedicated to accelerating the execution of Lightweight Block Ciphers
- RISC-V hardware extension dedicated to the protection of Lightweight Block Ciphers against Auxiliary Channel Attacks

Each contribution is experimentally supported by tests to evaluate hardware cost, performance, latency and physical security against auxiliary channel attacks.

Contents

List of Figures

List of Tables

Acronyms

$\oplus$	The XOR instruction
ADD	The addition instruction
AES	Advanced Encryption Standard
AND	The binary AND instruction
CAN	Control Area Network
CPA	Correlation Power Analysis
CSR	Control Status Register
DES	Data Encryption Stadard
DPA	Differential Power Analysis
FF	Flip Flop
FPGA	Field Programmable Gate Array
GE	Guessing Entropy
GFN	Generalised Feistel Network
HD	Hamming Distance Model
HW	Hamming Weight Model
ISA	Instruction Set Architecture
LBC	Lightweight Block Cipher
LBC-ISA	Lightweight Block Cipher Extension of the Istruction Set Architecture
LUT	Look-Up Table
LWC	Lightweight Cryptography
NICV	Normalized Inter-Class Variance for detection of side-channel leakage
ProtLBC-ISA	Protected Extension of the ISA for LBC
risc	Reduced Instruction Set Computer
RSM	Rotating S-Box Masking
RV	Random Variable
SCA	Side-Channel Analysis
SNR	Signal to Noise Ratio
SPN	Substitution Permutation Network
XOR	The binary exclusive OR instruction

Résumé en français

Introduction

Contexte

Les communications font partie des fonctions clés dans les futurs véhicules. Elles se développent à grande vitesse et nécessitent une fort besoin de sécurité. Qu'elle soit intra- ou extra- véhiculaire, une communication fiable est impérative afin d'assurer la prise de décision rapide en fonction de l'environnement. Néanmoins, elle peut être interceptée et même modifiée par des attaquants potentiels et nécessite donc l'utilisation de protections. La Cryptographie est une réponse évidente pour sécuriser la communication, mais les algorithmes utilisés aujourd'hui semblent inadéquats en termes de complexité et de latence, surtout pour satisfaire les contraintes physico-économiques des systèmes embarqués utilisés à grande échelle. Bien que l'industrie automobile se base généralement sur des standards, il n'est pas rare que certains acteurs préfèrent l'utilisation de technologies ad hoc plus adaptées aux contraintes de coût, de performance et de sécurité. Tant que la communication est interne, un standard ne s'impose pas, mais avec les communications entre véhicules l'absence de standard impose au véhicule une grande versatilité pour assurer la compatibilité des communications, tout en respectant les contraintes de complexité et latence. Un second problème émergeant et propre aux systèmes embarqués est la protection contre les attaques, notamment les attaques physiques du fait que le système peut être accessible à l'utilisateur. Ces attaques sont redoutables car elles peuvent contourner la protection mathématique des algorithmes cryptographique. C'est le cas des attaques par canaux auxiliaires qui exploitent les fuites liées à l'activité du calcul par le biais de la consommation ou du rayonnement électromagnétique. Il est donc essentiel de prendre cet aspect en compte dans les implémentations des algorithmes de chiffrement pour obtenir un haut niveau de sécurité.

Dans le cadre de la chaire «Connected Cars and CyberSecurity (C3S)», les objectifs principaux de la thèse sont d'étudier la faisabilité d'implémentation d'une grande variété d'algorithmes de chiffrement symétrique ainsi que leur protection. Plus précisément la finalité est d'implémenter plusieurs familles d'algorithmes sur un matériel contraint en surface et qui aurait de bonnes performances en termes de sécurité physique et latence. Une solution optimale est d'avoir une implémentation générique, capable d'exécuter rapidement différents algorithmes de chiffrement légers, utilisant peu de ressources et garantissant une protection contre les

attaques physiques. L'implémentation peut être totalement matérielle ou alors hybride matérielle / logicielle en tirant parti de la présence d'Unités de Contrôle Électroniques embarquée. Pour assurer le meilleur compromis d'implémentation, cette dernière architecture peut partir d'une modification du jeu d'instruction d'un processeur RISC pour satisfaire la propriété d'agilité des algorithmes de cryptographie légère. Cette généricité garantit une grande flexibilité, une base de protection commune aux algorithmes, pour un coût comparable à celui d'un micro-contrôleur standard.

Contributions

Nous avons étudié de nombreux algorithmes de chiffrement et avons proposé une première architecture matérielle, une seconde architecture hybride matérielle/logicielle afin d'implémenter de façon efficace la Cryptographie Légère dans un système embarqué contraint. Nos principales contributions se résument à:

- Classification des Algorithmes de Chiffrement Légers par Bloc
- Implémentation matérielle configurable dédiée à l'accélération de l'exécution des Algorithmes de Chiffrement Légers par Bloc de type SPN
- Extension matérielle du RISC-V dédiée à l'accélération de l'exécution des Algorithmes de Chiffrement Légers par Bloc
- Extension matérielle du RISC-V dédiée à la protection des Algorithmes de Chiffrement Légers par Bloc contre les Attaques par Canaux Auxiliaires

Chaque contribution est étayée expérimentalement par des tests pour évaluer le coût matériel, les performances, la latence et la sécurité physique face aux attaques par canaux auxiliaires.

Organisation

Le Chapitre 2 présente le contexte technique avec une base de connaissances prérequises pour appréhender le travail de recherche. Une attention particulière a été portée dans la description de chaque algorithme étudié lors de cette thèse. Même si une sous-partie des algorithmes a été conçue pour validation dans un circuit FPGA, tous ont contribué à trouver l'architecture la plus efficace.

Le Chaptire 3 décrit l'implémentation matérielle générique et configurable dédiée à l'accélération de l'exécution de certains Algorithmes Légers de Chiffrement par Bloc (LBC).

Le Chapitre 4 décrit l'implémentation hybride matérielle-logicielle, basée sur l'Architecture d'Ensemble d'Instructions (ISA) du RISC-V. Cette implémentation est en fait une extension de l'ISA du RISC-V avec des instructions matérielles dédiées à l'accélération de l'exécution des LBCs.

Le Chapitre 5 décrit l'implémentation de protections génériques contre les attaques par canaux auxiliaires rendant l'exécution des LBC résiliente aux attaques

par observation de la consommation énergétique. Cette implémentation est également une extension de l'ISA du RISC-V par le biais d'instructions matérielles.

Enfin, le Chapitre 6 est une discussion générale des coûts et des gains de chaque contribution. Les perspectives à tirer de ces recherches y sont également abordées.

Implémentation Générique Entièrement Matérielle

Les Algorithmes Légers de Chiffrement par Bloc (LBC) suivent certains principes permettant d'assurer la sécurité mathématique et consistent en trois étapes (*Cf.* Sect. 2.4). Ces trois étapes ont permis d'identifier un moyen générique d'accélérer l'exécution des algorithmes LBC tout en réduisant significativement le surcoût de surface lié à l'implémentation de plusieurs algorithmes en parallèle. Nous avons donc réalisé une architecture unique et générique utilisant les mêmes instructions pour plusieurs algorithmes. Cette première réalisation est entièrement matérielle.

Architecture

L'architecture de cette première implémentation est basée sur la structure des algorithmes SPN. Un module est dédié à chacune des trois étapes d'un chiffrement par bloc. Le module **S** correspond aux S-Boxs, ou l'étape de Confusion. Le module **P** correspond au P-Layer, ou l'étape de Diffusion. Le module **K** correspond à l'étape d'Ajout de la Clé. De plus, chaque module est configurable au niveau de l'ordre dans lequel les 3 étapes sont exécutées. Ceci se fait par le biais d'une série de multiplexeurs dont le module s'appelle Route_Mux. Cette implémentation est représentée dans la Fig. 1

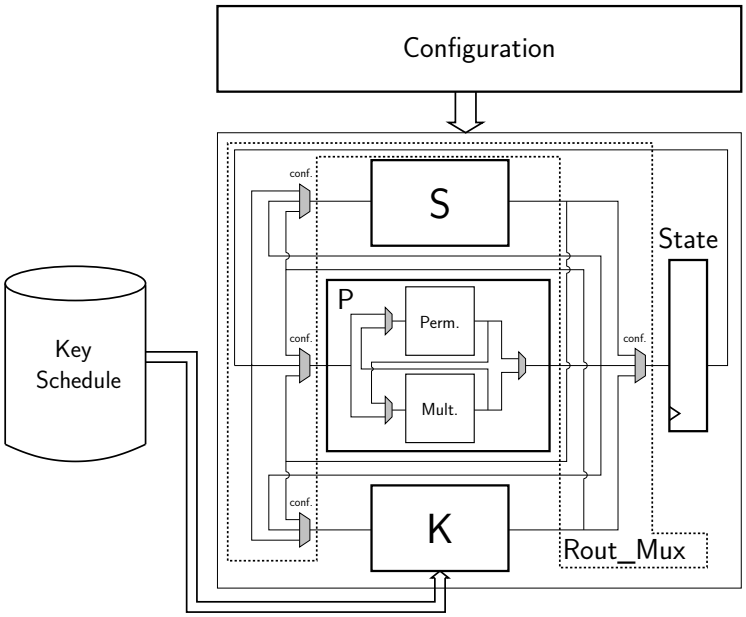


Figure 1: Architecture Entièrement Matérielle pour les Algorithmes de Chiffrement de Type SPN

Chacun des modules est présenté ensuite.

Le Module K: L'ajout de la clé

Le module **K** est composé d'un XOR 64-bit qui va XORer les 64-bit du bloc au 64-bit de la clé de ronde. Cette clé de ronde est pré-calculée par le *Key Scheduling* et correspond à la clé de ronde de l'algorithme XORé avec les éventuelles constantes de ronde et adapté au format 64-bit. Le soucie majeur de ce module est le fait de garder en mémoire ces clés tout en accédant à une de ces clés par cycle. Cela peut être géré en utilisant une mémoire sécurisée ainsi qu'un *bypass* entre le module et la mémoire. Cette solution n'a pas été implémentée et reste théorique.

Le Module S: Les S-Boxs

Ce module est composé de seize S-Box 4×4 en parallèle et identiques, c'est-à-dire qui ont la même table de substitution. Une S-Box se comporte comme une LUT 4×4 et c'est ainsi qu'elles sont réalisées. Ces S-Box, bien qu'identiques entre elles, peuvent être configurées entre chaque algorithme et au cours d'une exécution.

Le Module P: Le P-Layer

L'étape de Diffusion d'un algorithme SPN peut être divisée en deux sous-étapes: la première de ces sous-étapes est le *MixColumn*, qui correspond à une multiplication de matrices et la seconde est une permutation au niveau bit, qui correspond parfois à la fonction *ShiftRow*.

La Multiplication générique de Matrices

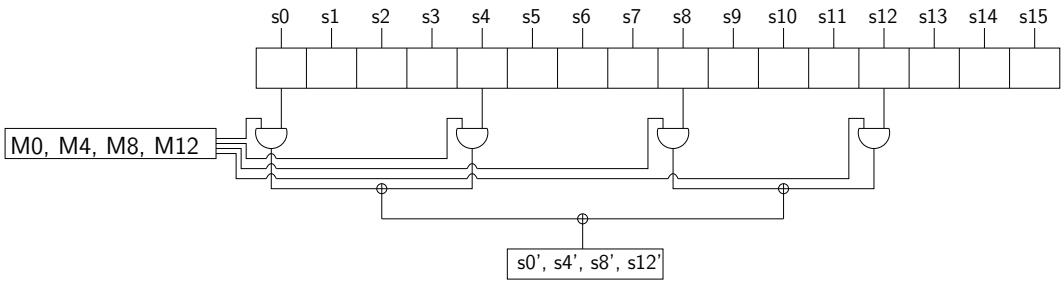
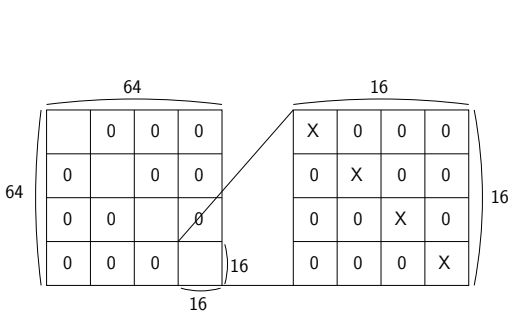


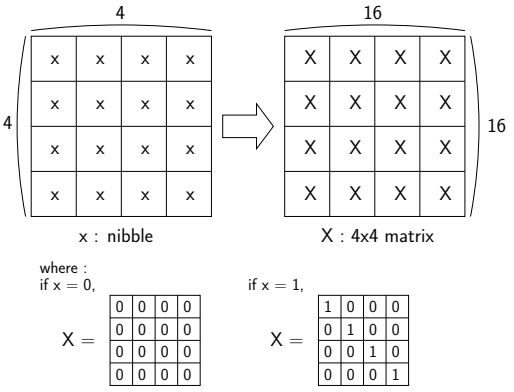
Figure 2: La Multiplication de Matrices Génériques

La multiplication de matrices de la Figure 2 va utiliser les quatre même bits de l'entrée (s0, s4, s8, s12) et quatre groupes de bits différents issus de la matrice (M0, M4, M8, M12).L'opération entre ces bits va permettre d'obtenir quatre bits en sortie (s0', s4', s8', s12') qui feront partie du résultat de la multiplication. Cette méthode permet de n'utiliser que 256 bits de la matrice au lieu des 2048 bits d'une matrice 64×64.

Cette réduction des paramètres est possible en ne considérant que des matrices de niveau nibble (4 bits) ou s'y rapportant. En effet, certains algorithmes utilisent des matrices 64×64 ou 4×4 dont l'information utile peut être transformée en une matrice 16×16, comme le montrent les Figures 3a et 3b.



(a) Transformer une Matrice 64×64 de type Prince en une matrice 16×16



(b) Transformer une Matrice 4×4 en une Matrice 16×16

Figure 3: L'Uniformisation des Tailles de Matrices

Permutation Niveau Bit Générique

Parmi les différents algorithmes étudiés, certains utilisent le *MixColumn*, qui est équivalente au *ShiftRow* avec une rotation de certains nibbles. Pour les autres algorithmes, comme une simple rotation est insuffisante, il est nécessaire d'utiliser une permutation au niveau bit. Dans le cas d'une implémentation spécifique, ces permutations sont une réorganisation des connexions avec un coût nul pour un algorithme donné, mais assez complexe pour la rendre générique à plusieurs algorithmes. Comme il est très coûteux de concevoir une permutation générique 64×64, il a fallu se concentrer sur certains aspects de ces permutations qui permettent d'assurer la diffusion.

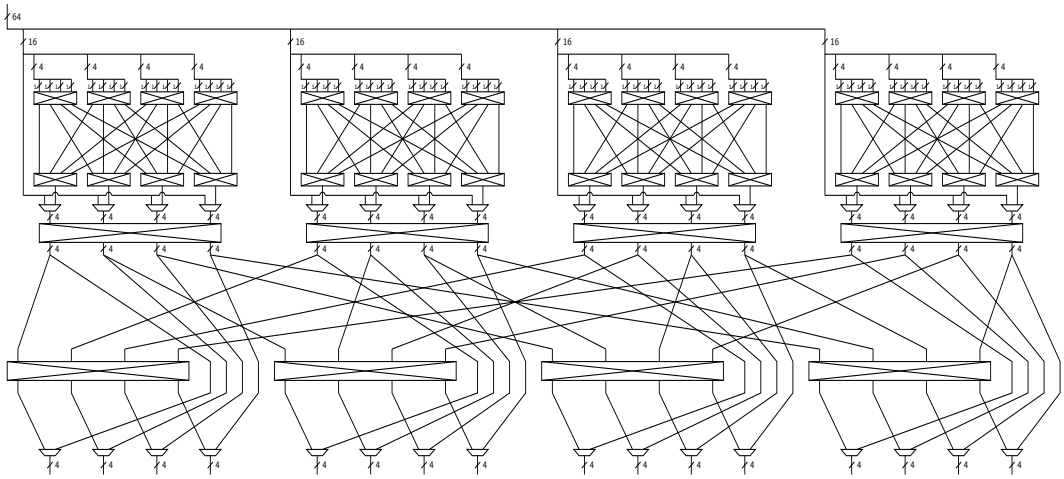


Figure 4: L'Instruction de Permutation au Niveau Bit

La solution choisie est représentée par la Fig. 4. Elle permet aussi bien d'exécuter *ShiftRow* que les permutations au niveau bit de PRESENT ou de GIFT en une seule instruction. Chaque module permettant la permutation de 4 bits ou de 4 nibbles sont implémentés comme des commutateurs Banyan (*Cf.* Sect. 3.6.2). Cela

permet à la fois d'accélérer l'exécution mais également de réduire significativement l'utilisation de la mémoire. En effet, on passe de 8 bits pour un crossbar à 5 bits en utilisant un commutateur Banyan, ce qui représente 200 bits de configuration au lieu de 320.

Résultats

Les coûts d'implémentation sont présentés dans la Table 1 qui donne le détail de chaque module. On observe que tous les modules qui demandent un haut

Table 1: Cout de modules de l'Architecture pour un niveau d'agilité **C**

Module		Coût		Pourcentage de Surface
		Surface	GE	
Configuration	Route_Mux	695	678	26.9
	S-Box	348	339	
	Permutation	956	932	
	Multiplication	1390	1355	
	Other	87	85	
Route_Mux		2791	2720	21.6
S		1784	1739	13.8
P	Permutation	2059	2007	21.7
	Multiplication	744	725	
K		175	171	1.4

niveau de configuration représentent une part importante du coût matériel. Notamment les parties Permutation et Route_Mux qui ont un coût quasi-nul dans le cas d'implémentations classiques.

Table 2: Les Différents Niveaux d'Agilité de l'Architecture

Algorithme	Niveaux d'Agilité
PRESENT	**A**
GIFT	**C**
SKINNY	**B**
Midori	**D**
PRINCE	
MANTIS	

De plus, cette architecture peut être modifiée pour correspondre à différents niveaux d'agilité, présentés dans la Table 2. C'est-à-dire qu'en ajustant les modules il est possible d'adapter l'architecture pour accélérer l'exécution d'un nombre croissant d'algorithmes. En comparant ces résultats au coût d'implémentation de la somme de chacun des algorithmes indépendamment dans la Table 3 on observe qu'au-delà d'un certain niveau d'agilité, l'architecture générique permet des économies de ressources conséquentes. En effet, à partir de six implémentations d'algorithmes, la réduction des coûts matériels est de 62%.

Table 3: Comparaison des Coûts entre les Niveaux d'Agilité et la Somme des Implémentations

Niveau d'Agilité	Surface de L'Architecture Générique (en GE)	Somme du coût des Implémentations (Ratio de Complexité)
A	8494	1.72
B	8245	1.96
C	9212	1
D	9631	0.625

Bien que ces résultats sont très bons, en termes d'agilité et complexité, une autre approche consiste à combiner les technologies matérielles et logicielles.

Implémentation Générique Hybride Matérielle-Logicielle

Dans les voitures modernes, la présence d'Unités de Contrôle Électroniques (ECU) à base de microcontrôleurs repose sur une exécution logicielle des algorithmes. Nous avons donc jugé pertinent de baser notre implémentation sur un jeu d'instructions logicielles ISA adapté, tout particulièrement l'ISA du RISC-V (*Cf.* Sect. 2.8). Ce jeu d'instructions (ISA) est libre et ouvert et a été conçu dans l'optique d'être étendu pour répondre au mieux à une diversité d'utilisations. Cette possibilité d'extension d'instructions permet l'ajout d'instructions matérielles spécifiques afin de réduire la latence d'exécution logicielle de divers algorithmes.

Classification des Algorithmes Légers de Chiffrement par Bloc

Afin de réduire les surcoûts liés à l'extension il est important de bien choisir les instructions qui seront implémentées. Pour ce faire, nous avons établi une classification, présentée en Fig. 4, de nombreux LBCs en fonctions des instructions requises à l'accélération de leur exécution.

Table 4: Classification des Algorithmes Légers de Chiffrement par Bloc

Catégorie	Algorithme	Confusion	Diffusion	
			Type	Niveau
I	Simon	AND	Rotation	Bit
	Speck	ADD	Rotation	Bit
	Simeck	AND	Rotation	Bit
	RC5	ADD	Rotation	Bit
	XTea	ADD	Rotation	Bit
II	GOST	S-Box	Rotation	Bit
	Rectangle	S-Box	Rotation	Bit
III a	PRESENT	S-Box	Permutation	Bit
	GIFT	S-Box	Permutation	Bit
III b	PRINCE	S-Box x2	MatMult x3	Bit
IV	TWINE	S-Box	MatMult	Nibble
	SKINNY	S-Box	MatMult	Nibble
	Midori	S-Box	MatMult	Nibble
	MANTIS	S-Box	MatMult x3	Nibble

Cette classification divise les LBCs en quatre catégories, à chaque catégorie est associée un type d'instruction qui permet d'en accélérer l'exécution:

- Catégorie **I**: Les instructions utilisées sont déjà présentent dans l'ISA
- Catégorie **II**: Nécessite l'ajout d'une instruction S-Box
- Catégorie **III**: Nécessite l'ajout d'une instruction S-Box et d'une instruction de Permutation ou de Multiplication de Matrices, au niveau bit
- Catégorie **IV**: Nécessite l'ajout d'une instruction S-Box et d'une instruction de Multiplication de Matrices au niveau nibble

L'extension qui a été étudiée s'est concentrée sur l'ajout de ces catégories d'instructions.

Le VexRisc Core

Il existe de nombreuses implémentations de l'ISA du RISC-V, et celle que nous avons choisie est le VexRisc Core 32-bit. En effet, il permet l'ajout de nouvelles instructions par l'utilisation de plug-ins qui n'impactent pas le cœur du RISC-V 32-bit.

Proposition d'Extension d'ISA

Une implémentation 32-bit pour manipuler des blocs de 64 bits ajoute tout de même une contrainte. En effet, la sortie des instructions est sur 32 bits, cela nécessite donc de dédoubler chaque instruction afin d'obtenir le résultat sur 64 bits, comme représenté en Fig. 5. Ainsi, lorsqu'une instruction sera évoquée, on parlera en fait d'un double instruction la première renvoyant la partie basse de la sortie, et la seconde renvoyant la partie haute de la sortie.

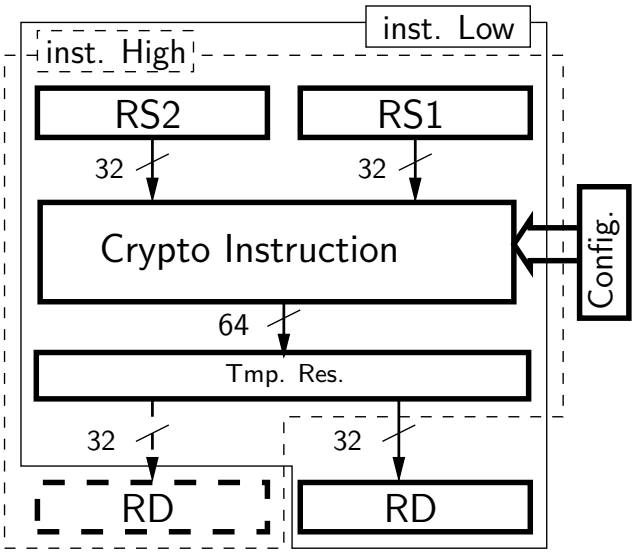


Figure 5: Sortie 64-bit Utilisant Deux Instructions RISC-V 32-bit

En nous basant sur la classification établie, nous avons décidé d'implémenter cinq instructions (ou groupes d'instructions) pour cette extension:

- L'instruction S-Box: **SBOX_C**
- L'instruction permutation au niveau bit de PRESENT: **PRESENT_D**
- L'instruction permutation au niveau bit de GIFT: **GIFT_D**
- Les instructions de multiplication de matrices au niveau bit de PRINCE: **PRINCE_D**
- L'instruction de multiplication de matrices au niveau nibble: **NMAT_D**

Comme expliqué dans la partie précédente, les instructions génériques au niveau bit peuvent être très coûteuses, c'est pourquoi nous avons choisi de réaliser certaines instructions spécifiques à un algorithme. L'étape de Diffusion de PRINCE évolue au cours de l'exécution, c'est pourquoi plusieurs instructions sont nécessaires. Les instructions **SBOX_C** et **NMAT_D** sont quant à elles génériques et sont utilisées dans plusieurs algorithmes.

L'Instruction SBOX_C

Cette instruction prends quatre bits en entrée et les transforme en quatre bits de sortie par le biais d'une table de substitution. Son comportement est donc le même que celui d'une LUT lorsque la table est configurable. Nous avons donc implémenté l'instruction **SBOX_C** comme une LUT. La configuration de cette instruction a en principe lieu avant l'exécution mais certains algorithmes peuvent nécessiter une reconfiguration au cours de l'exécution.

Les Instructions PRESENT_D et GIFT_D

Ces deux instructions utilisées des permutations au niveau nibble. L'implémentation choisie n'est pas générique et dans les deux cas, l'implémentation se résume à une réorganisation des fils.

Le Groupe d'Instruction PRINCE_D

L'algorithme de PRINCE utilise trois étapes de Diffusion différentes au cours de son exécution, c'est pourquoi nous avons implémenté trois instructions différentes. **PRINCE_DF** qui est la Diffusion du début d'algorithme, **PRINCE_DM** qui est la Diffusion du milieu d'algorithme et **PRINCE_DL** qui est la Diffusion de fin d'algorithme.

L'Instruction NMAT_D

Cette instruction convient à tous les algorithmes qui utilisent une structure *ShiftRox MixColumn* au niveau nibble. Elle permet la multiplication par n'importe quelle matrice de nibble 16×16 qui serait composée de 1 ou de 0 et nécessite 256 bits de paramètres. Bien que ces matrices 16×16 contiennent toute l'information, il est nécessaire de les transformer en matrice 64×64 pour les multiplier aux blocs 64-bit. La méthode est présentée dans la Figure 6

L'utilisation de cette instruction comprend donc plusieurs étapes, qui sont décrites dans Fig. 7. Les 256 bits de paramètre sont d'abord stockés dans huit registres CSR 32-bit. Les paramètres de chaque registre sont ensuite ANDés avec chaque nibble d'entrée, puis chaque résultat est XORé afin d'obtenir une sortie 64-bit.

Le *datapath* de la multiplication en elle-même est montrée dans la Figure 8.

Une fois ces instructions choisies et implémentées, il a s'agit de les tester et de les valider.

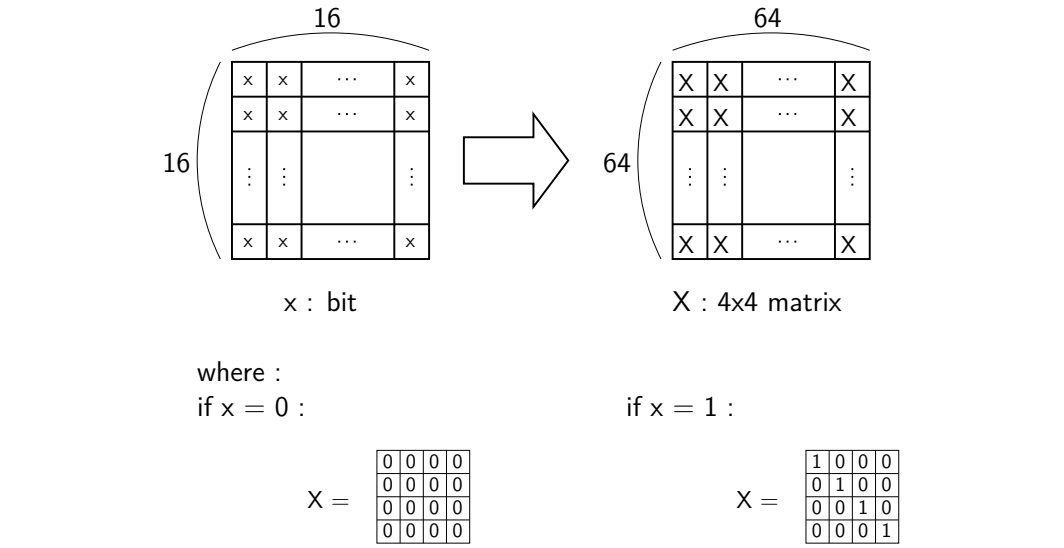


Figure 6: 256-bit de Configuration d'une Matrice 64×64 pour NMAT_D

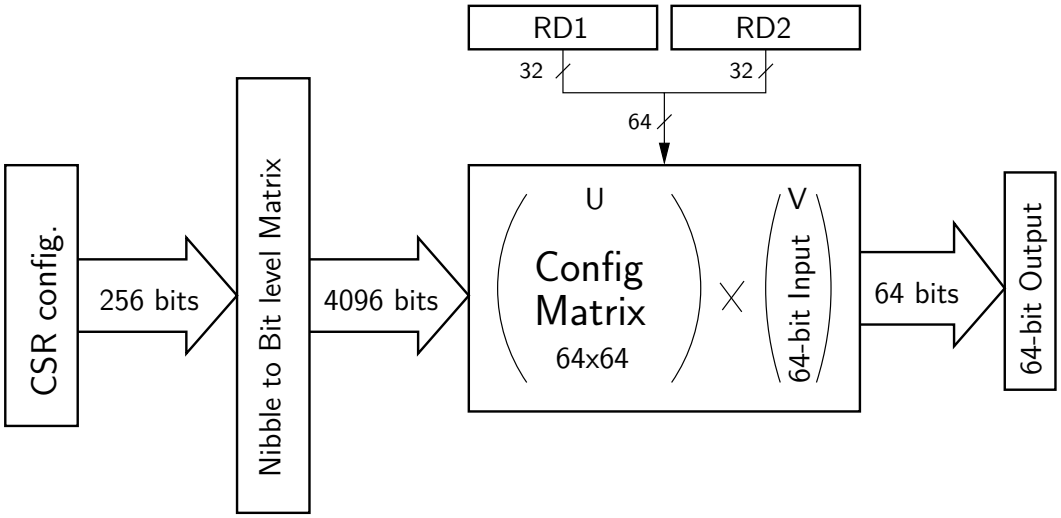


Figure 7: The NMAT_D Instruction

Plateforme de Test et de Validation

La plateforme que nous avons utilisée est basée sur l'implémentation VexRisc du RISC-V. Elle est représentée en Figure 9 et contient les éléments suivants:

- Un VexRisc Core
- Deux mémoires *on-chip* indépendantes de 32KB pour les programmes et les données
- Un module de communication permettant la connexion à un ordinateur hôte via une interface USB

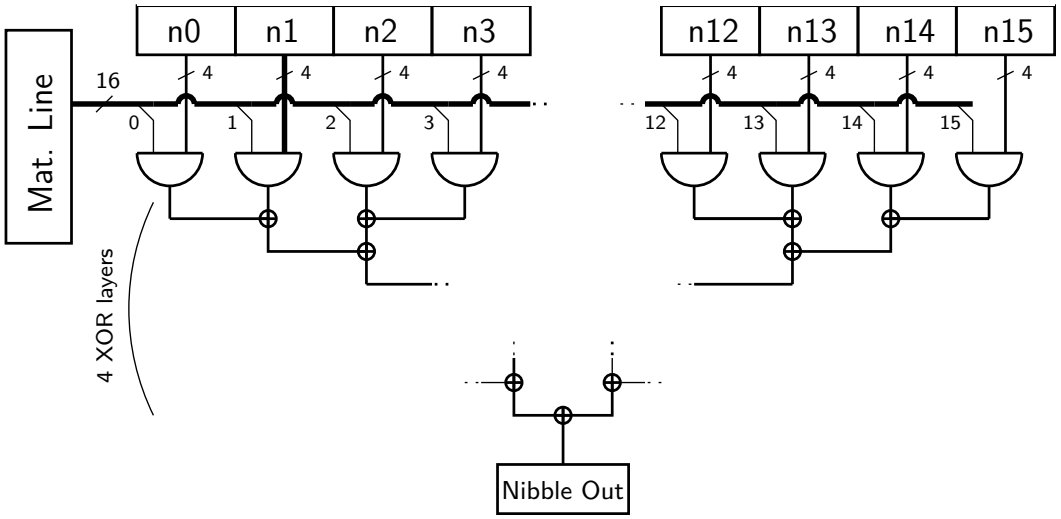


Figure 8: The NMAT_D multiplication datapath

- Un module de configuration qui inclut des registres d'adresse mémoire dédiés capable de lire et d'écrire depuis l'ordinateur hôte via le module USB
- Un module avec port GPIO (General Purpose Input Output) pour les interactions directes avec les utilisateurs

La carte cible est la Chipwhisperer CW305 [2]. Elle est basée sur un FPGA Artix-7, et a été conçue pour permettre les analyses de consommation par canaux auxiliaires.

Afin de tester la validité de chacune de nos instructions, chaque algorithme a été implémenté en C en utilisant la chaine de compilation GNU GCC pour RISC-V. A chaque fois, deux modes d'exécution ont été implémentés:

- Logiciel, qui utilise l'ISA RISC-V Standard
- Accélérée, qui utilise la version étendue LBC-ISA

Ce qui nous permet de tester à la validité des instructions mais également de comparer leurs performances.

Coûts et Gains de l'Implémentation

Nous avons d'abord comparé les coûts d'implémentation du RISC-V sans extension à ceux de divers configuration d'extension permettant d'accélérer l'exécution d'un ou plusieurs algorithmes.

Ces résultats, présentés dans le Tab. 5 montrent que les deux instructions les plus coûteuses que sont **SBOX_C** et **NMAT_S** font ensemble doubler le coût en LUT et moins de tripler le cout en mémoire. Le surcoût lié à l'utilisation d'instruction génériques n'est donc pas négligeable mais reste dans un ordre de grandeur semblable.

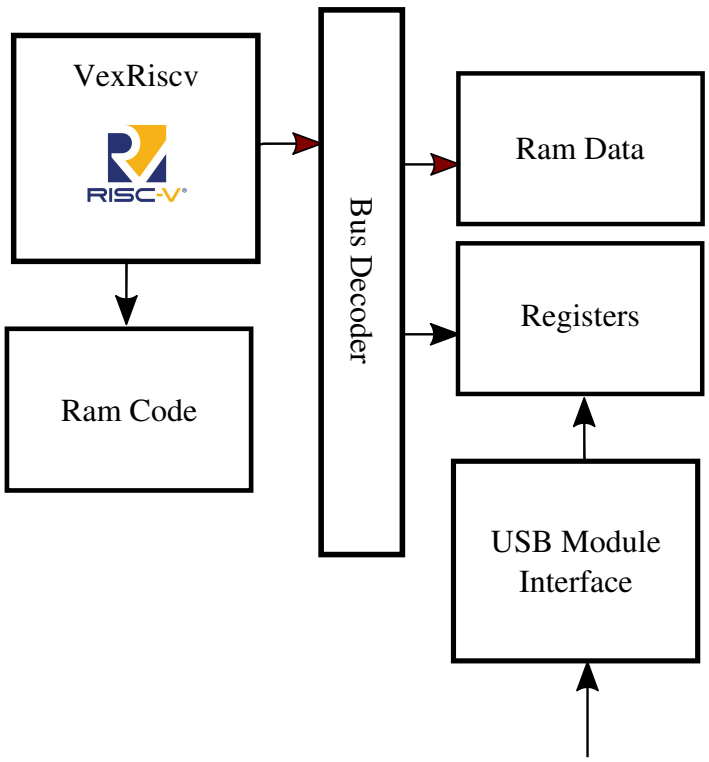


Figure 9: VexRisc Platform

Table 5: Coût en Surface de l'Extension du RISC-V par Algorithme

Algorithme	Instructions LBC-ISA	LUTs	%	FFs	%
None		973	-	765	-
PRESENT	SBOX_C + PRESENT_D	1173	+21	767	+0.2
GIFT	SBOX_C + GIFT_D	1103	+13	767	+0.2
PRINCE	SBOX_C + PRINCE_D	1438	+48	769	+0.5
PRINCE	2xSBOX_C + PRINCE_D	1512	+55	769	+0.5
Midori	SBOX_C + NMAT_D	1778	+82	1023	+33.7
TWINE	SBOX_C + NMAT_D	1778	+82	1023	+33.7
SKINNY	SBOX_C + NMAT_D	1778	+82	1023	+33.7

Nous nous sommes ensuite intéressés à l'accélération d'exécution engendrée par ces instructions, les résultats sont dans le Tableau 6 et sont donnés en nombre d'instructions.

La Table 6 montre qu'en termes d'accélération de l'exécution le facteur de gain est beaucoup plus important. En effet, il varie entre 8 et 100 ce qui représente entre 1 et 2 ordres de grandeur d'écart.

Table 6: Nombre d'Instructions par Algorithme en fonction de l'ISA

Algorithm	Base ISA	LBC-ISA	Facteur de Gain
PRESENT	12544	358	35
GIFT	10661	319	33
PRINCE with reconfiguration	17357	2313	8
PRINCE without reconfiguration	17357	126	138
Midori	18944	232	81
TWINE	41279	622	66
SKINNY	40887	409	100

Implémentation Générique Protégée

Bien que les algorithmes de chiffrement soient mathématiquement protégés il n'en reste pas moins certaines failles qui menacent leur niveau de sécurité. Ces failles se trouvent notamment dans le matériel sur lequel sont exécutés ces algorithmes. En effet, dû à la nature des portes CMOS (*Cf.* Sect. 2.6.2) tout changement de valeur d'une porte engendre une consommation d'énergie qui peut être exploitée pour une attaque dite physique. Ce type d'attaque s'appelle les attaques par canaux auxiliaires SCA (*Cf.* Sect. 2.6), qui font partie des attaques physiques, et il est important de pouvoir s'en protéger. Cette dernière partie de thèse va consister à l'implémentation d'une autre extension du RISC-V permettant la protection contre les attaques par observation de consommation.

Méthode d'Évaluation de la Sécurité

Dans un premier temps, il s'agit d'identifier la présence de la menace en essayant d'attaquer l'exécution de chaque type d'algorithme afin d'extraire la clé secrète. Pour ce faire, nous allons utiliser différents outils permettant de repérer les fuites, d'attaquer ces fuites puis d'observer le résultat de ces attaques. Chacun de ces outils est basé sur l'utilisation de traces de consommation d'énergie. Ces traces sont utiles lorsqu'elles sont en grand nombre puisqu'elles permettent une étude statistique de la consommation énergétique significative en effaçant l'impact du bruit.

La Normalized Inter-Class Variance

La NICV permet d'identifier chaque endroit de l'exécution où une fuite potentielle a lieu. Elle compare l'activité de l'intégralité du circuit à celle d'une valeur donnée en entrée. Cela permet d'observer des piques de corrélation comme sur la Figure 10.

Il est important de noter que ces piques indiquent les fuites potentielles mais que chaque pique n'est pas exploitable. Il faut ensuite mettre en adéquation ces graphiques aux simulations d'exécution des algorithmes.

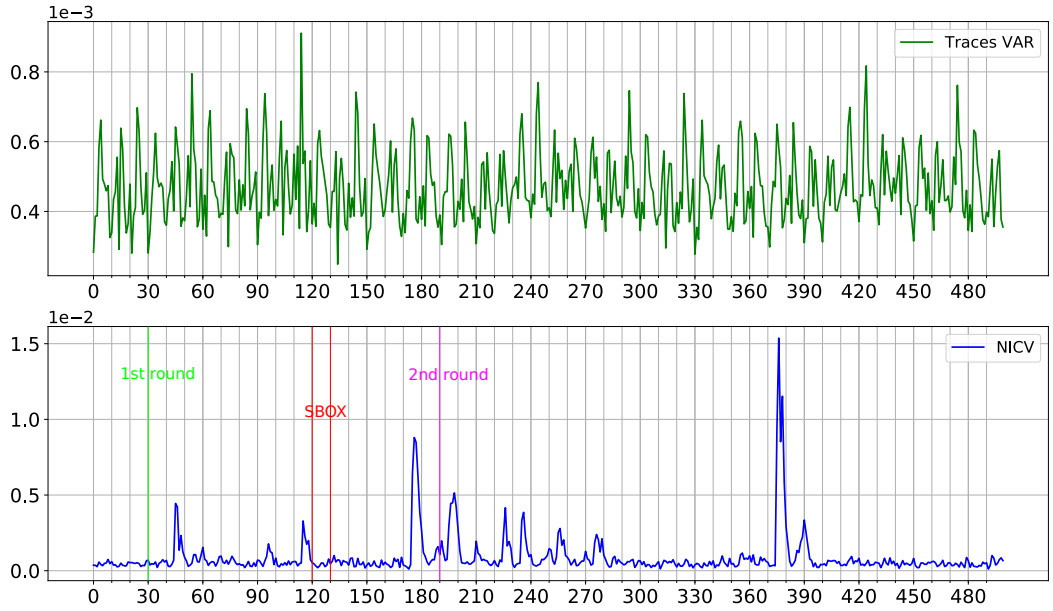


Figure 10: La NICV de PRESENT pour 32768 traces

L'Analyse de Consommation Énergétique par Corrélation

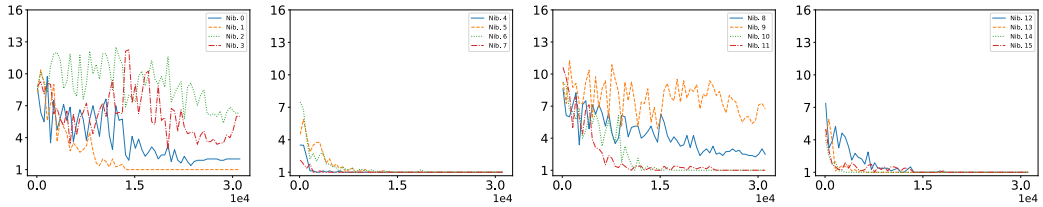
L'Analyse de Consommation Énergétique par Corrélation (CPA) est le concept de base de l'attaque utilisée, elle est expliquée en détails Section 5.2.3. Le principe est d'observer la consommation énergétique en sortie de S-Box qui est la seule instruction non-linéaire des algorithmes de chiffrement étudiés. C'est là qu'une corrélation discriminante peut apparaitre entre le *Poids de Hamming* et les traces d'exécution. Pour chaque valeur de nibble possible en sortie de S-Box, une hypothèse est faite et en comparant les résultats de cette hypothèse aux traces il est possible de déterminer la valeur cachée. Chaque hypothèse donne lieu à un coefficient, la valeur avec le plus haut coefficient est considéré comme la plus probable. Lorsque cette valeur la plus probable est la même que la valeur du nibble de la clé, on dit alors que ce nibble est *cassé* Lorsque suffisamment de nibble sont *cassés* alors on dit que la clé est *cassée* et l'algorithme de chiffrement n'est alors plus une protection.

La Guessing Entropy

La valeur la plus probable a son importance dans une analyse CPA mais il est également important d'observer la place de la valeur du nibble de la clé. En effet, la Guessing Entropy (GE) va donner la position dans le classement CPA de la vraie valeur du nibble de la clé en fonction du nombre de traces utilisées. La position utilisée est en réalité la moyenne des positions obtenue lors de huit attaques sur différents sous-groupes de traces. Cela permet d'observer l'évolution de l'efficacité d'une attaque au fur et à mesure que le nombre de traces augmente. C'est cette mesure que nous avons choisie pour représenter comment l'exécution des algorithmes se comporte face aux attaques.

Évaluation de la Sécurité des Implémentations non-protégées

Pour différents types d'algorithmes, représentés par PRESENT, GIFT, PRINCE et Midori nous avons observé l'évolution de la GE pour des paquets de traces de taille croissante, allant de 100 à 32768. Les résultats pour PRESENT sont présentés dans la Figure. 11.



(a) Nib. 0 - 3 (b) Nib. 4 - 7 (c) Nib. 8 - 11 (d) Nib. 12 - 15

Figure 11: La Guessing Entropy de l'Exécution de PRESENT pour jusqu'à 30000 Traces de Consommation Énergétique

On observe que la majorité des nibbles a pu être *cassés* avant la barre des 15000 traces mais que certains semblent beaucoup moins sensibles à l'attaque. On note néanmoins que pour ces derniers nibbles la tendance est à la décroissance de la GE qui semble tendre vers 1.

Les résultats des autres algorithmes sont présentés dans la Section 5.3. Dans l'ensemble, l'attaque fonctionne de la même manière sur tous les algorithmes, une majorité des nibbles sont *cassés* avant 15000 traces et certains y sont moins sensibles. Nous n'avons pas d'explication à ce phénomène qui rend certains nibble plus difficiles à attaquer. Parmi les pistes évoquées il y a la possibilité que les S-Box ne s'exécutent pas toutes en parallèle ce qui créerait un décalage dans le flot d'exécution qui modifierait le moment de sortie des S-Box.

Proposition d'Extension d'ISA pour la Protection

Les résultats précédents mettent en évidence la nécessité de pouvoir se défendre contre les SCA. C'est pourquoi la suite de nos recherches se sont tournées vers l'implémentation de protections contre ces attaques. Pour ce faire nous avons maintenu la forme de cette implémentation comme une extension matérielle et logicielle de l'ISA du RISC-V.

Parmi les divers contremesures existantes, celle qui a retenu notre attention est la protection par Masquage avec Rotation des S-Box (RSM), décrite dans la Section 2.7.2.2. Cette protection va masquer le texte d'entrée qui sera décalé d'un nibble après chaque S-Box. La particularité de cette protection est donc qu'elle s'applique directement aux S-Box et nécessite la modification des S-Box en fonction des nibble du masque. Ce qui est adapté à notre implémentation pour deux raisons:

- Les algorithmes sur lesquels nous nous concentrons utilisent tous la S-Box, cette protection devrait donc s'appliquer à chacun d'en eux.

- L'instruction S-Box de la LBC-ISA permet de configurer chaque S-Box indépendamment, et donc de le faire en fonction de chacun des nibbles du masque

Le dernier point qu'il reste à traiter pour l'implémentation de RSM est la rotation. Pour ce faire, nous avons opté d'effectuer la rotation sur les entrées et sorties des S-Box, ce qui évite de revoir les reconfigurer en permanence. Nous avons donc implémentié un *Barrel Shifter* 64-bit au niveau nibble, qui permet d'effectuer n'importe quelle rotation au niveau nibble sur un mot de 64 bits. La Figure 12 montre comment ce *Barrel Shifter* permet d'assigner chaque nibble à la S-Box qui correspond à son nibble de masque. On peut voir que grâce aux S-Box

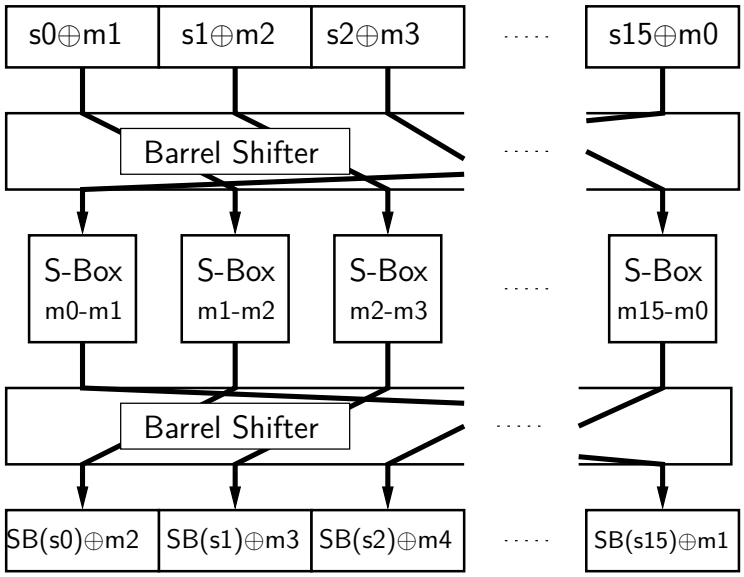


Figure 12: L'Etape de Confusion avec RSM

modifiées, les *Barrel Shifter* permettent d'assurer la rotation du masquage sans rotation du texte.

Nous avons implémenté deux instructions **NShift** et **InvNShift** qui forment l'extension protégée, nommée **ProtLBC-ISA**. Les instructions implémentées, sont de type R (*Cf.* Fig. 13) ce qui permet d'avoir deux entrées 64-bit mais ne laisse plus l'option de préciser la valeur du décalage (*shift amount*). Il est donc nécessaire de définir ce *shift amount* par le biais de registres CSR.

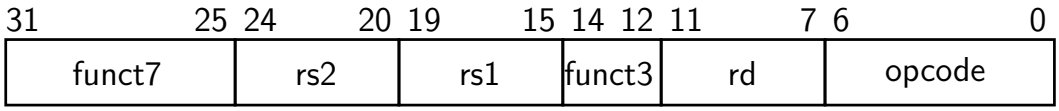


Figure 13: Les Instructions RISC-V de Type R

L'utilisation d'un tel registre est de toute façon nécessaire puisque l'index de rotation initiale du masque est secret et doit donc être stocké dans un de ces registres. La valeur de l'index initial contenue dans ce registre est ensuite incrémentée

après chaque étape de Confusion. De plus, comme le montre la Fig. 12, chaque utilisation de **NShift** implique l'utilisation de la rotation inverse, qui est exécuté par **InvNShift**. Ces deux instructions sont donc l'inverse l'une de l'autre, elles utilisent donc des paramètres légèrement différents, mais le matériel utilisé pour les implémenter est le même.

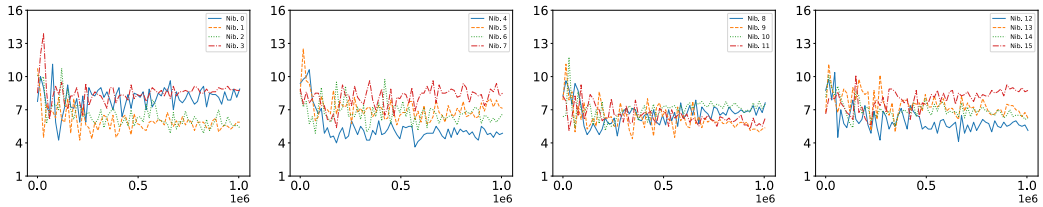
Le coût de cette implémentation est résumé dans le Tableau 7, la version complète du tableau est dans la Section 5.1.3.1

Table 7: Coût Matériel de l'Implémentation Générique Protégée

Configuration	Instruction Supplémentaire	LUTs	%	FFs	%
RISC-V ISA de Base	-	978	-	775	-
RISC-V Etendu LBC-ISA (PRESENT)	S-Box + PRESENT_D	1178	+20	777	+0.2
RISC-V Etendu LBC-ISA (ENTIERE)	S-Box + PRESENT_D + GIFT_D + PRINCE_DF + PRINCE_DM + PRINCE_DL + MMULT_D	2135	+118	1038	+34
RISC-V Protégé Extension ProtLBC-ISA (PRESENT)	S-Box + PRESENT_D + Nibble-Level Barrel Shifter	1993	+104	814	+5.0
RISC-V Protégé Extension ProtLBC-ISA (ENTIERE)	S-Box + PRESENT_D + GIFT_D + PRINCE_DF + PRINCE_DM + PRINCE_DL + MMULT_D + Nibble-Level Barrel Shifter	2619	+168	1070	+38.1

Évaluation de la Sécurité de l'Implémentation Protégée

Nous avons utilisé la même méthode d'évaluation que pour les versions non protégée, excepté que nous avons augmenté le nombre de traces. Pour les algorithmes PRESENT et GIFT, nous avons capturé 1048576 de traces et pour les algorithmes PRINCE et Midori, nous avons capturé 2097152 de traces. La Figure 14 montre l'évolution de la GE pour l'attaque de PRESENT. Les résultats des autres algorithmes sont présentés Section 5.4.



(a) Nib. 0 - 3 (b) Nib. 4 - 7 (c) Nib. 8 - 11 (d) Nib. 12 - 15

Figure 14: La Guessing Entropy de l'Exécution de PRESENT Protégé pour jusqu'à 1 million de Traces de Consommation Énergétique

Ces courbes montrent que même pour des paquets de plus d'1 million de traces la GE ne converge pas vers 1, la tendance est plutôt à des valeurs entre 4 et 10. Cette absence de convergence vers 1 montre que le bon nibble de clé n'est jamais identifié par l'attaque comme étant le plus probable. Ainsi, l'efficacité de la protection a bien été montrée pour un nombre de traces supérieur à 1 million, soit environ 100 fois plus que pour les versions non protégées. Les résultats pour les autres algorithmes montrent des comportements similaires ce qui confirme également que la même protection va fonctionner sur d'autres algorithmes utilisant des S-Box.

Conclusion

Nous avons montré lors de cette étude qu'il était possible de grandement réduire la latence de nombreux algorithmes de chiffrement légers par blocs avec un nombre restreint d'instructions génériques dédiées. Pour ce faire, nous avons d'abord adopté une approche entièrement matérielle qui s'est avérée trop coûteuse et ne profitant pas assez des outils offert par le contexte.

Nous avons également montré qu'en choisissant avec attention ces instructions et en utilisant l'ISA du RISC-V, il était possible de réduire les coûts matériels. En effet, l'utilisation du logiciel a permis de réduire certaines contraintes liées à l'aspect générique. Il est donc bien possible d'adopter une approche générique à l'implémentation des LBC et d'obtenir un véritable gain de latence. Le coût matériel est quant à lui raisonnable et adaptable à l'agilité requise.

Pour ce qui est de l'implémentation des contre-mesures, nous avons pu montrer qu'une approche générique permettait d'assurer une très bonne protection. En effet, avec l'ajout matériel d'une seule instruction, nous sommes parvenus à montrer que la protection pouvait être augmentée d'un facteur de 100 en termes de nombre de traces. Cette protection a tout de même un coût en termes de latence puisque le nombre d'instructions augmente.

Lors de ces recherches nous nous sommes concentré sur l'objectif d'assurer une protection générique à des algorithmes dont l'exécution a été accélérée par des instructions elles aussi génériques. Ce que nous sommes parvenue à faire. Cependant nous avons laissé plusieurs questions en suspend et il aurait été intéressant de prendre le temps d'y répondre. Parmi elles, l'explication de la résistance de certains nibbles aux SCA pourrait donner lieu à des travaux intéressants.

Chapter 1

Introduction

1.1 Motivation

Communication in modern car is developing at a very high pace and with it comes the need to secure these communications. Indeed, whether it is internally, or between vehicles, reliable communication is essential to allow modern car to take decisions and react to its surrounding autonomously. Nevertheless, communication signals can be intercepted and even modified by attackers and therefore require protection. Cryptography is the evident answer to secure communication, but existing cryptography is believed by some to be inadequate and the cost of implementing it in all the embedded systems too high. Although the car industry relies on standards, it also sometimes prefer to use *ad hoc* technologies more suited for its constraints in terms of cost, performance and security. As long as the communication is internal this does not represent an issue, but once the cars start communicating with each other, then lacking standard becomes a time sensitive problem.

A second issue with security of embedded communication is the protection against physical attacks. Indeed, these attacks can bypass the protection offered by sound cryptographic algorithms by exploiting the fact that these algorithms run on an accessible hardware device. Nowadays, protection against those physical attacks must be considered when designing the implementation of a cryptographic algorithm and such protections are far from being standardized.

In the context of The Connected Cars and CyberSecurity (C3S) Chaire [60], which financed this work, we wanted to see how multiple cryptographic algorithms could be handled by an embedded system. In other words, is it possible to keep a small hardware footprint while keeping good performances. What drove this work was therefore to find low-resource agile implementation for lightweight cryptographic algorithms while providing protection against physical attacks. The main constraints were therefore agility, low latency and low hardware cost. It is important to note that this only applies to the datapath and that *key scheduling* is not part of this work.

Although a fully hardware implementation is proposed, the context of embedded Electronic Control Units in modern cars, lead us to also study software implementations and to propose a dedicated processor. This allowed flexibility, which fully

hardware implementations do not provide, while keeping hardware cost reasonable when compared to a standard microcontroller.

1.2 Contributions

Throughout this work we have studied several cryptographic algorithms and proposed a hardware architecture and a dedicated processor to implement efficiently Lightweight Cryptography (LWC) in an embedded system. Our main contributions can be summarized as follows:

- Classification of Lightweight Encryption Algorithms by Block
- Configurable hardware implementation dedicated to the acceleration of the execution of Lightweight Block Ciphers of the SPN type
- RISC-V hardware extension dedicated to accelerating the execution of Lightweight Block Ciphers
- RISC-V hardware extension dedicated to the protection of Lightweight Block Ciphers against Auxiliary Channel Attacks

Each contribution is experimentally supported by tests to evaluate hardware cost, performance, latency and physical security against auxiliary channel attacks.

During our study, we also came across an interesting phenomenon of leakage linked to the internal pipeline execution order in a processor core and the instructions order in the executed software.

1.3 Organisation

Chapter 2 give a thorough background regarding the state of the art of all the concepts studied during this work. As it will be explained later, all studied ciphers were not exploited to support our results, but they were nevertheless all considered for their potential use. All these ciphers are described in Appendix A.

Chapter 3 describes the configurable agile hardware implementation dedicated to the accelerated execution of the selected Lightweight Block Ciphers (LBC).

Chapter 4 describes the dedicated processor implementation, based on the RISC-V open Instruction Set Architecture (ISA). This implementation extends the RISC-V ISA with additional hardware instructions dedicated to the acceleration of LBCs execution

Chapter 5 describes an agile protection against side-channel attacks implemented to make the execution of LBCs resilient against power observation attacks. The implementation of this countermeasure expand also the RISC-V ISA with a dedicated set of instructions. The protection is evaluated experimentally for several LBCs by performing correlation attacks and comparing its resistance to the unprotected dedicated processor.

Finally, Chapter 6 is an overall discussion of the costs and benefits of each of the contributions, along with a look at the perspectives this work offers.

Chapter 2

State-of-the-Art

Understanding the state-of-the-art of cryptography is essential to understanding this work and this chapter will cover the basis of the principles and ideas used. It will first explain the basic cryptographic principle, before presenting those specific to ciphers and most specifically Lightweight Block Ciphers. Each of the ciphers considered for this research will be detailed in Appendix A. It will then explain the basic cryptanalysis principles which are used during an attack and more specifically a side-channel attack. Finally, it will present the RISC-V ISA and some of its implementations.

2.1 Introduction to Cryptography

Whether it is at a very large scale with the Internet or at a very small scale with embedded technology, communication is at the heart of modern development in technology. Just as it becomes more and more essential to how our society work, it becomes more and more essential to protect it. These protections used are based on cryptography which is the science of protecting sensitive information. It is based on four principles [15]:

- **Confidentiality**: which makes sure that only the allowed party (or user) have access an understandable form of the message.
- **Authenticity**: which makes sure that users can identify themselves.
- **Integrity**: which makes sure that the transmitted message has not been tempered with (due to transmission or a third party).
- **Non-Repudiation**: which makes sure that no allowed party can deny being the sender or receiver of a message

A cryptographic algorithm or *cipher* will first *encrypt* a *plaintext* into a *ciphertext*. This means that the original message or *plaintext*, which can be understood by anyone, is transformed or *encrypted* in an unintelligible form or *ciphertext*. Once the *ciphertext* has been obtained, it is then sent to the receiver. While the message is on its way to the receiver, it can be intercepted as the communication lines can hardly be entirely secured. Thanks to the encryption, the intercepted

message isn't the hidden *plaintext*, but the unintelligible *ciphertext*, meaning that the actual information is still secure. The receiver then receives the message and uses the same *cipher* to transform back or *decrypt* the *ciphertext* back into the *plaintext*.

The security of this operation is based on the mathematical soundness of the ciphers. The encryption process uses a hidden key to transform the plaintext. This process will be further explained later. The size of the keys is important in gaging the security of an algorithm. These keys are usually at least 128-bit long, meaning that a *brute force* attack on the ciphertext would require testing out 2^{128} possibilities, which would take more time to compute on the best supercomputer available than the age of the universe, by an unfathomable margin. There are other types of attacks, some of which will be covered later, but reliable attacks cannot only use the mathematical properties of the algorithms.

Although ciphers share these properties, there are many of them and each have their specificities.

2.2 Symmetric Cryptography

In symmetric cryptography, a single hidden key is shared between both users. Keeping this secret is the keystone of symmetric cryptography as if any of the users has his key disclosed, the entire security falls apart. The main issue with this type of cipher is therefore to ensure that both users get access to the secret key without having anyone else get a hold of that key. Symmetric cryptography can be divided in two main classes of algorithms which are *stream ciphers* and *block ciphers*.

2.2.1 Stream Ciphers

Stream ciphers [69] are based on bit sequence encryption using a pseudo random *key-stream*. This *key-stream* is generated with a *Linear Feedback Shift Register* (LFSR) using the secret key as a seed. There are two main types of stream ciphers:

- **Synchronous Stream Ciphers**: which use a *key-stream* independent of the plaintext or ciphertext. The *key-stream* is generally XORed with the plaintext. This method requires bit-level synchronization between the sender and the receiver.
- **Self-Synchronizing Stream Ciphers**: which uses a *key-stream* based of previous ciphertext or ciphertexts. This method is less demanding but is also more sensible to certain types of attacks, as any bit disruption in the plaintext results in the same bit being modified in the ciphertext.

This type of cipher is well adapted to certain types of communication where the package size is unknown and can vary such as mobile phone conversations. They are also efficient in terms of hardware and fit well embedded applications. Their main issue remains their levels of security which appears to be lower than that of block ciphers [48].

2.2.2 Block Ciphers

As their name suggest, block ciphers do their encryption and decryption on *blocks*, which is a subdivision of a message in groups of a defined number of bit. The most common block sizes are 64-bits and 128-bits, although some more recent algorithms have opted to increase their block sizes ([13], [27], [52]). Block Ciphers apply a series of mathematical functions (linear and non-linear) which together form a *round*. The linear functions are two folds, first there is the *key addition*, second the *permutation*, which can also be combined with XORs to make matrix multiplication functions. The non-linear function is the *substitution*. These operations will be covered in more details later. This *round* is repeated a certain number of times, which depends on the algorithm itself and the security/latency equilibrium. The plaintext, the ciphertext and the intermediate values or collectively referred to as the *state*.

Block ciphers usually require a mode of operation in order to be used correctly, there are four of them [28]:

- **Electronic Code Book (ECB) mode**: Each plaintext is encrypted (and decrypted) separately. This may cause a security issue as the same plaintext is encrypted into the same ciphertext meaning that patterns may appear over large amount of data or when the same data block is used regularly.
- **Cipher Block Chaining (CBC) mode**: Each plaintext (resp. ciphertext) block is XORed with the previous ciphertext (resp. plaintext) block during encryption (resp. decryption). An initializing vector is used for the first plaintext (resp. ciphertext).
- **Cipher Feedback (CFB) mode**: Operates like the self-synchronizing stream ciphers (*Cf.* Sect. 2.2.1).
- **Output Feedback (OFB) mode**: Operates like the synchronous stream ciphers (*Cf.* Sect. 2.2.1).

2.2.3 Key Scheduling

The secret key is essential in providing security, and the level of security is based on the key size. Nonetheless, using the exact same key multiple times in a row in each round of a block ciphers can cause certain patterns to appear which could lead to specific attacks which strongly decrease the security level. The solution to this problem is to use *key scheduling*. Key scheduling consists in modifying the key using a specific algorithm in order to obtain a different key for each of the cipher's rounds called *round keys*. It is essential in ensuring the security and must itself be as mathematically reliable as the cipher itself to ensure it cannot be attacked [30].

Each algorithm has its own specific *key scheduling*, which may use both linear and non-linear functions. To lower complexity some ciphers re-use some parts of the cipher itself. *Key scheduling* is also useful in ensuring the entire 128 bits of key information is used to encrypt 64-bit size block ciphers.

2.3 Types of Block Ciphers

There are two main architectures for common block ciphers, Feistel Networks and Substitution and Permutation Networks (SPN). The algorithms that belong to these families have similar architectures with some differences in the sub-part functions and execution order. The most iconic examples are the Data Encryption Standard (DES) [75] and the Advanced Encryption Standard (AES) [29] that have respectively a Feistel and an SPN architecture. Another interesting subcategory of block ciphers, especially in the context of Lightweight Block Ciphers, is the α-reflective architecture which can be illustrated with the PRINCE cipher [19].

2.3.1 Feistel and Generalized Feistel Network (GFN)

The Feistel type of block cipher corresponds to the structure of the DES cipher. DES was amongst the first standard cipher, the first widely used free cipher, and has a very important place in the history of ciphers [72]. It was designed in the 1970s at IBM and was slightly modified in consultation with the National Security Agency (NSA) before being accepted by the National Bureau of Standards (NBS), which is now the National Institute of Standard and Technology (NIST) [59]. The intervention of the NSA lead to high levels of suspicions concerning the actual security of the algorithm. Indeed, some of its design was classified, and they advocated for a very short key length of only 56-bits which was much smaller than the initial design of IBM. These concerns created a strong interest in the field of cryptography, which was academically disregarded at the time, and lead to its study as an academic science. The surge over the distrust of the DES lead to an ultimate discovery in 1998 that the DES was indeed too small and that a brute-force attack, which consists in testing every single key, could break it in under three days. It also led to the development of a new cipher, using the same mathematical properties as DES but with a longer key size, called triple-DES in 1985. DES was a pioneer in the world of non-military use ciphers and its impact still resonates through the most recent ciphers. Amongst its influence is the remaining distrust concerning any NSA suggestion in cryptography.

The Feistel Network or Generalised Feistel Network (GFN) algorithms specificity is that only half of the *state* goes through the round process before permuting each half (*Cf.* Fig. 2.1). The value the block has during the encryption or decryption, between the plaintext and the ciphertext, is called the *state*. This *state* is therefore what is being manipulated by the encryption or decryption. In a Feistel or GFN type cipher, this state is being divided into two part, the right part and the left part. During a round, each part receives a different treatment and at the end of the round those parts are switched.

During a round, the first part is temporarily modified, and this temporary value is then XORed to the second part. The temporary value is obtained by applying linear and non-linear functions to the first part. The linear function usually includes some form of permutation to shuffle the bits around, also known as a *Diffusion Step* and a XORing with the round key, also known as the *AddKey Step*. The non-

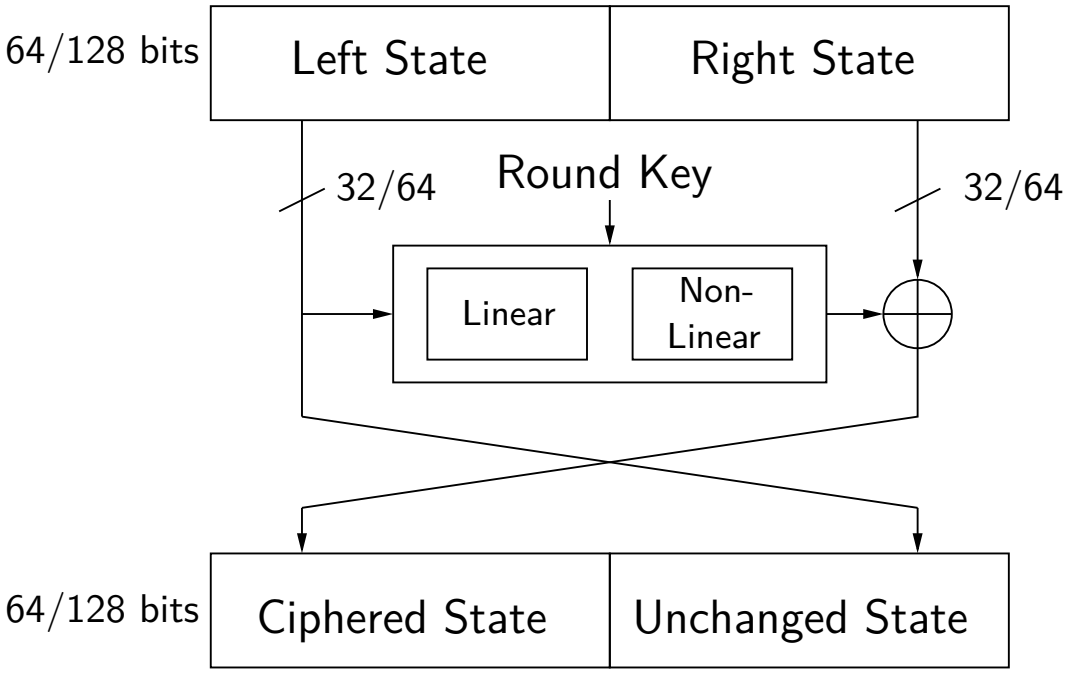


Figure 2.1: Generic Feistel Round

linear function is usually either a simple AND or ADD function or an S-Box, also known as the *Confusion Step*, this will be covered in more details in Section 2.4.3. Combining both these types of functions offer extra security compared to either being used alone. Both the non-modified first part and the XORed second part are then permuted meaning that the next round will apply the modifications to the other part.

Mathematically, it corresponds to: Let,

- L_i be the left part of the state during round i,
- R_i be the right part of the state during round i,
- $\lambda()$ be the linear function, which corresponds to the *Diffusion Step*,
- $\kappa()_i$ be the XORing function with the round key from round i which corresponds to the *Key Addition Step*,
- $\mu()$ be the non-linear function which corresponds to the *Confusion Step*.

During a round:

$$\begin{aligned} tmp &= \mu(\kappa(\lambda(L_i)_i)) \\ R_i &= R_i \otimes tmp \\ R_{i+1} &= L_i \\ L_{i+1} &= Ri \end{aligned}$$

This type of cipher has a very interesting property, which is that both encryption and decryption use the same hardware. This design allows to greatly lower the area requirements of implementing such ciphers. Nonetheless, since only half the state is getting ciphered during each round, it means that the number of rounds is usually higher in Feistel and GFN ciphers.

2.3.2 Substitution Permutation Network (SPN)

The Substitution Permutation Network (SPN) corresponds to the type of the AES. The AES is the current NIST standard for cryptography and is widely used in many fields. It is build around the similar principles as the GFN expect that the entire state is modified at each round. Here also, there are both linear and non-linear functions which modify the state at each round.

The AES algorithm uses a 128-bit block size and a variable key length of 128, 192 or 256 bits. Each key size has its own dedicate cipher AES-128, AES-192 and AES-256. In this algorithm, the state is seen a 4×4 byte matrix. It uses fours functions:

- **The Sub Byte Step**: This function is the only non-linear step. It is an 8-bit substitution which turns each input byte into an output byte according to a fixed table, also called an 8×8 S-Box. The table is based on certain mathematical properties in order to avoid patters from forming and from being exploited by attackers. During this step, the S-Box is applied to each byte of the state.
- **The Shift Rows Step**: This step rotates the bytes of each line of the 4×4 matrix by a defined value.
- **The Mix Column Step**: This step combines some columns of the state with a linear transformation.
- **The AddRound Key Step**: This step XORs each bit of the round key with the corresponding bit of the state.

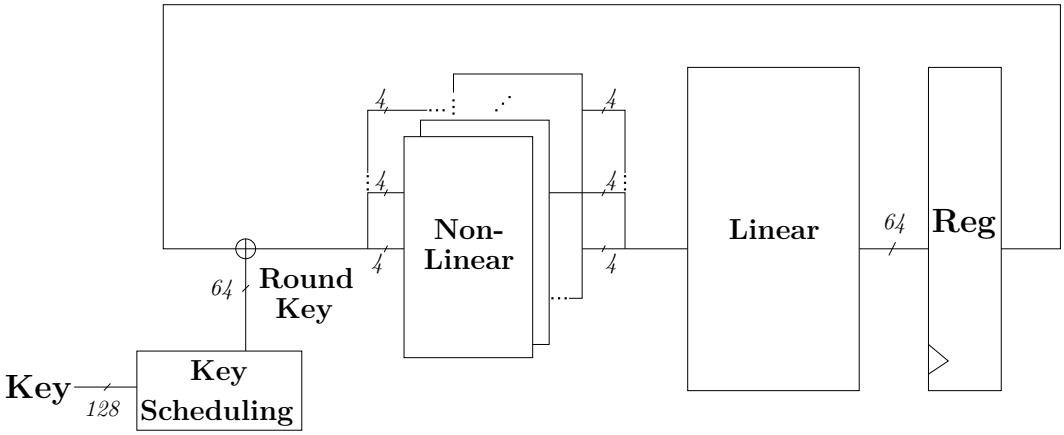


Figure 2.2: Generic SPN Round

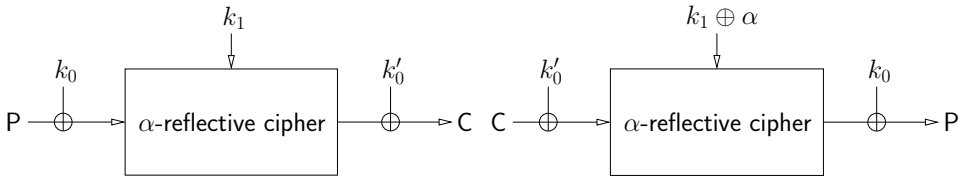
Those four functions are specific to the AES algorithm and although many ciphers imitate those step, some merge the Shift Rows step and the Mix Column step in a single *Confusion Step* Therefore a more generic way to look at SPN algorithms' rounds is shown in Figure 2.2.

2.3.3 The α-reflective Algorithms

This is not actually a type of algorithm, but rather a subtype as its rounds use the same structure as SPN algorithms. The specificity of α-reflective algorithms is their overall structure which can be divided into three parts:

- **The First Rounds**: where the rounds are composed of the S-Box, the Shift Rows and the Mix Column.
- **The Middle Round**: which is composed of the S-Box followed by the Mix Column (without the Shift Rows) followed by the inverse S-Box.
- **The Last Rounds**: where the rounds are composed of the inverse S-Box, the inverse Shift Rows and the Mix Column.

This means that unlike the rest of the algorithms, they use two different *Diffusion Steps* and two different *Confusion Steps*. The fact that both the S-Box and the inverse S-Box are used in the cipher means that the exact same algorithm can be used for encryption and decryption. The only difference is that the key used is changed to a related key as shown in Fig. 2.3



(a) $\alpha - reflective$ Encryption (b) $\alpha - reflective$ Decryption

Figure 2.3: The $\alpha - reflective$ Property

2.4 Three Steps Model of a Block Cipher Round

The name of the SPN type comes from the effect each step has on the state. The *Sub Byte Step* corresponds to the substitution where each byte is replaced by another byte, this step is also referred to as the *Confusion Step*. The *Shift Rows* and *Mix Column Steps* correspond to the permutation where each byte's position is changed linearly, this step is also referred to as the *Diffusion Step*. Although it is less apparent, these steps can also apply to Feistel and GFN type ciphers.

Each block cipher is a succession of rounds, which can be divided in three main steps and repeated a defined number of time. Those three steps are:

- Key addition
- Confusion

- Diffusion

Each algorithm has a different way of implementing these steps and the heart of this classification was to identify the similarities that could be found within those steps. These similarities can then be used to define a few agile instructions which would accelerate the execution of each of those ciphers.

A representation of the general execution pattern of LBC algorithms is shown in Alg. 1. This representation is mostly based on SPN structures but fits the principles of any LBC.

Algorithm 1 Generic Lightweight Block Cipher Algorithm

BLOCK CIPHER ALGORITHM

Data: PlainText (64-bit), Key (128-bit)
Parameters: S-Box parameters (64-bit)
Result: CipherText (64-bit)

$roundKey[\#ofRounds] \leftarrow GenerateRoundKeys(Key)$

for $round = 0$ to $\#ofRounds$ **do**
  $state \leftarrow state \oplus roundKey[round]$
  $state \leftarrow SBox(state)$
  $state \leftarrow Diffusion(state)$
end forreturn $state$

2.4.1 Key Addition Step

The key addition is the step where the secret key is mixed with the state. This addition can take on different forms from one algorithm to another, but each can be summed up as a XOR between the 64-bit block and a 64-bit round key. The round key is a subkey which has been extracted from the 128-bit key. In most cases has been modified either by using a XOR between some parts of the key or a constant or both, some algorithms also use rotation and other basic instructions or their own *S-Box* to calculate a round key which is unique to each round of the algorithm.

Some algorithms also use whitening keys. These whitening keys are based on the key or part of the key. They are XORed to the plaintext (resp. ciphertext) before the first round and after the last round of encryption (resp. decryption). These keys are not useful in terms of mathematical protection, but are rather used as a way to slow down physical attacks. Indeed, these attacks use the values inside a round for their attacks, usually the first or last round, and thanks to the whitening key, these values are scrambled by another unknown value. Although this value can also be attacked, it does avoid simpler attack models from working. These whitening keys therefore have very little impact on the overall cipher as this XORing is only done twice, but is a burden for attackers who are slowed down.

Some algorithms also use Constants which are XORed to the state, or part of the state. Since these constants are combined to the state through the same

linear operation as the key, both those operations can be merged in a single XOR. This can only be done as long as a non-linear operation does not separate the key addition and the constant addition.

2.4.2 Key Scheduling

The process of calculating the round keys is called the key scheduling (*Cf.* Section 2.2.3). Each algorithm has a different key scheduling, which unlike the encryption algorithm follow very different rules for each cipher.

2.4.3 Confusion Step

The *Confusion Step's* function is to make sure that the ciphertext is as different from the plaintext as possible. This is the only non-linear step of any block cipher. This step can take on different shapes from one algorithm to another.

Some algorithms use basic instructions such as AND or ADD. The other main ways in which this step is implemented in ciphers is through the use of S-Boxes. These S-Boxes can have different size usually 4×4 for 64-bit blocks and 8×8 for 128-bit blocks. An S-Box transforms the input bits into different output bits according to an S-Box table. These tables are usually unique to each algorithm, although some share the same. Each table is calculated in order to respect certain mathematical properties which ensure a minimum level of entropy.

2.4.4 Diffusion Step

The *Diffusion Step's* function is to make sure that any small change in the plaintext has great repercussions on the ciphertext. This step can be seen as a permutation step where each bit is given a new position, or rather the information of a bit is spread to another position.

There are multiple ways to design this step. Some algorithms use a basic rotation. Other algorithms use a bit-level permutation. At bit-level we can also find algorithms using matrix multiplication.

The last type of diffusion, which is the most similar to the *Diffusion Step* found in the AES, is a two part permutation. Its first part corresponds to a Shift Row and its second to a Mix Colmun step. Although these can be seen as two separate steps, they can also be interpreted as a single nibble-level matrix multiplication.

All these different *Diffusion Steps* are linear functions and can therefore be merged together as a single matrix multiplication.

2.5 Lightweight Block Ciphers

Both DES, standardized over 40 years ago in 1977 [62], and AES, standardized over 20 years ago in 2001 [29], are becoming less and less adapted to the low amount resources available to embedded systems, such as those used in connected cars. Indeed, DES is not even a standard anymore and had to be replaced several times by more performant iteration of its algorithm. One of these is the Triple DEA (TDEA) [11] which is the Data Encryption Algorithm (DEA), itself a variation of the DES, applied three times in a row, the first and third time as encryptions and the second time as a decryption with different keys each time. This observation lead to the development of a new generation of ciphers, called lightweight or in the case of block ciphers: Lightweight Block Ciphers (LBC).

There are two ways to develop such algorithms, either as an improvement of an existing algorithm, like TDEA [11] or as entirely new algorithms, like PRESENT [18]. In either case, they must respect certain objective of reducing the different costs of ciphers, in terms of area, latency and/or power consumption. In other words all the resources which are scarce in embedded systems. There are multiple ways to achieve such an objective [51] amongst which are:

- **Smaller block size**: Memory is a very costly resource in terms of area and lowering the block size has a strong impact on memory costs. Having a smaller plaintext does lead to some security concerns which require limiting the size of the encrypted data.
- **Smaller key size**: Reducing the key size has an impact on area costs and some algorithms proposed smaller key sizes such as 80 or 96 for example. Nonetheless, this is no longer an option since the National Institute of Standards and Technology (NIST) requires using at least a 128 bit key [10] for any cipher past 2030.
- **Simpler rounds**: The *Confusion Step* of LBCs tend to use 4×4 S-Boxes rater than 8×8 S-Boxes as their hardware cost is significantly smaller. In the same vein is the use of bit-level permutation as their *Diffusion Step* which correspond to wire reorganisation which has practically no hardware cost. Having simpler round reduces area costs but does lower the security of each round and therefore requires additional rounds which has a latency cost.
- **Simpler key schedules**: The key scheduling mechanism in itself can be costly and simplifying them will have a positive impact on every resource (area, latency and power consumption). Using simpler key scheduling does imply having certain security concerns as some types of attacks such as related keys, weak keys, known keys or chosen keys attacks. This type of attacks can be handled by ensuring that all keys are generated independently using a secure key derivation function (KDF) [23].

A plethora of LBCs exist since none have been standardized by the NIST yet. This is about to change as the NIST has started the process of finding their next

standard or standards in lightweight cryptography [89], [88], [57]. This process started in 2019, right after the start of this work, and is in its final stage of selection, of the 57 ciphers initially submitted, only 10 remain in contention. Although some of my work is based on the absence of lightweight cryptographic standard, it still has a purpose once contextualised as some industries will rather rely on their own technology than a standard one.

2.5.1 Studied Lightweight Block Ciphers

In order to determine how to classify those algorithms we must first understand what the already established classifications imply. The main way to identify the ciphers we chose for this study is as Lightweight Block Ciphers.

The lightweight part refers to the fact that the ciphers studied were designed not only to fit security criteria but also area, latency and/or energy consumption criteria. These ciphers are therefore adapted to constrained environment with high levels of restrictions. Another aspect of such ciphers is that very few have been standardized, none have been standardized by the NIST and PRESENT [40] is the only one to have been standardized by the International Organisation for Standardization (ISO). This means that no one algorithm particularly stands out, and we therefore had to look at a large range of ciphers.

The studied algorithms and some of their properties are shown in Tab. 2.1, they will all be explained in detail further in Appendix A.

The block size corresponds to the amount of bits which is encrypted at a time. Some algorithms show multiple versions (even more than those found in Tab. 2.1) but the block size which is common to most LBCs is a 64-bit block.

The Key size corresponds to the amount of bits in the key, the higher the amount, the higher the security is. In order to ensure the right level of security, at least a 128-bit key is recommended [46].

The Focus corresponds to the resource usage which has been optimised for this algorithm. There are three main axes of optimisation.

- Latency, which looks to reduce the time it takes to encrypt data.
- Area, which looks to reduce the size of a dedicated ASIC implementation of that cipher.
- Energy, which looks to reduce the overall energy consumption of an implementation of that cipher.

The Structure corresponds to the overall way the data is processed for the algorithms. The Feistel Network or GFN algorithms' specificity is that only half of the state goes through the round process before permuting each half (*Cf.* Section 2.3.1). The SPN algorithms modify their entire state during each round, using the same three steps (*Cf.* Section 2.3.2). Finally, the $\alpha - reflective$ algorithms are those which use the exact same algorithm as both encryption and decryption (*Cf.* Section 2.3.3).

Table 2.1: Summary of properties of Lightweight Block Ciphers

Cipher	Block size (bits)	Key size (bits)	Structure	Number of Rounds	Focus	Ref.
NOEKEON	128	128	SPN	16	Area	[26]
Piccolo	64	128	GFN	33	Energy	[73]
LED	64	128	SPN	48	Area	[38]
Simon	64	128	Feistel	44	Area	[12]
	128	128	Feistel	68	Area	[12]
Speck	64	128	Feistel	27	Area	[12]
	128	128	Feistel	32	Area	[12]
Simeck	64	128	Feistel	44	Area	[93]
RC5	64	0-2048	N/A	2-512	Latency	[68]
XTea	32	128	GFN	32	Energy	[54]
GOST	64	256	GFN	32	Area	[61]
Rectangle	64	128	SPN	25	Area	[95]
PRESENT	64	128	SPN	32	Area	[18]
GIFT	64	128	SPN	28	Area	[9]
	128	128	SPN	40	Area	[9]
PRINCE	64	128	SPN α	10	Latency	[19]
TWINE	64	128	GFN	36	Area	[77]
SKINNY	64	128	SPN	32	Area	[14]
	128	128	SPN	40	Area	[14]
Midori	64	128	SPN	16	Energy	[8]
	128	128	SPN	20	Energy	[8]
MANTIS	64	128	SPNα	14	Latency	[14]

2.6 Side-Channel Attacks

Cryptography is a sound science, once a cipher has been validated, it becomes extremely hard, if not impossible to break it by using mathematical approaches. The science which looks to break such algorithms is called cryptanalysis. Any method which is able to break a cipher by using fewer samples than a brute force attack belongs to this category. Some mathematical methods sometimes exist, but they cannot reduce the sample amount significantly when attacking modern cryptography. The most efficient cryptanalysis comes from exploiting other means than the cipher itself, namely the hardware which is executing the algorithm. Indeed, cryptography needs to be executed via hardware and this creates a weakness. Any piece of hardware has a specific behavior during any execution, and this behaviour can be analysed in order to recuperate the information used during the execution such as the key or the plaintext. This information is said to have leaked from the hardware. This type of attack is based on the observation of factors linked to the hardware rather than the algorithm itself in order to retrieve the key or the plaintext directly. The leaked information can be exploited in different ways, the main types of attacks exploit variation in:

- Timing
- Power Consumption
- Electromagnetic Radiation.

These attacks can break a cipher with few measurements.

2.6.1 Timing Attacks

Any operation is executed in a given amount of time. Therefore, by analysing the execution time it because possible to deduce which operation has been executed. This is particularly important in the case of conditional operation as observing the time will indicate with condition was true. When this condition is dependent on the value of a bit, then the value of this bit is no longer hidden. This type of attacks was first introduced in [44] in 1996 by Kocher *et. al*.

2.6.2 Power Attacks

Computation on an electronic device consumes energy. The consumed energy is variable and depends on the value of resulting bit value. Indeed, hardware devices use CMOS cells as their basic building block. A CMOS cell as shown in Fig. 2.4 derives current from a constant power supply V_{DD}.

During the output commutation from $0 \rightarrow 1$ or $1 \rightarrow 0$ there is a charge and a discharge of the output capacitance C_{load}. The energy-per-transition dissipated by the gate can be expressed with the following equation:

$$E = \int_0^{V_{dd}} C_{load} \cdot V_{dd} \frac{\mathrm{d}V_c}{\mathrm{d}t} \mathrm{d}t = C \cdot V_{dd}^2$$

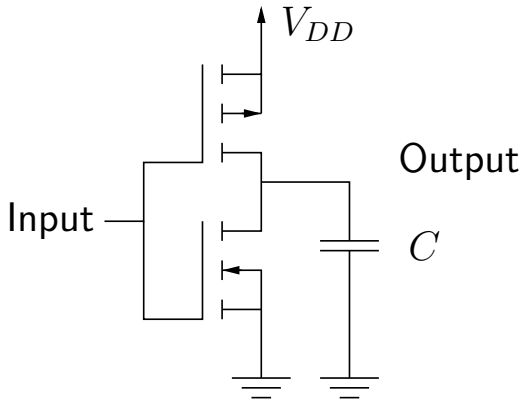


Figure 2.4: A CMOS cell

hence, the fundamental dynamic power consumption of a CMOS gate is:

$$P_{dyn} = A \cdot f \cdot C \cdot V_{dd}^2$$

where f is the frequency of the CMOS device and A is the toggle activity.

The difference in power consumption when the output state of a gate changes can be observed and exploited to retrieve information about the computation. Thus, an attacker can collect power traces with different plaintexts and, with the correct attack model which links the power consumption to the activity in the circuit, could theoretically recover a secret key. The specificities of a power attacks will be discussed later.

2.6.3 Electromagnetic Attacks

This type of attack exploits the same CMOS weakness, except that instead of collecting the power consumption, it observes the electromagnetic radiation of the device. Indeed, any current flow induces an electromagnetic radiation. Unlike the power attacks where the power consumption is measured for the entire device, electromagnetic attacks can be localised by focusing on certain parts of the circuit. This means that depending on where the probe is placed, the radiation will differ. This also means that this method can be more effective as it mainly collects useful information and much less noise due to the activity of the rest of the device.

2.6.4 Attack and Leakage Model

In order to analyse the leaked information obtained through any of these techniques, an attacker must compare it to hypothesis made on the secret key. This comparison uses an attack model, this section will explain the basic concepts of such a model.

Let,

- L be a random variable (RV) that represents the observed leakage (measurement)
- K be the secret cryptographic key, an n-bit data; unknown to the attacker, assumed to be uniformly distributed on $\mathbb{F}_2^n$

- X be the input (plaintext) or the output (ciphertext) of the cryptographic algorithm; known to the attacker
- $Z = f(X, K)$ for a given function f be a sensitive variable used internally that depends only on X and K. We assume that this sensitive variable Z can be computed exhaustively from all possible K by the attacker. Furthermore, Z causes the leakages; put differently, when the key guess is correct, Z and L are dependent.

For each key guess (for each value k of K), the SCA will analyse whether the RVs Z and L are dependant. To note, a key guess is usually not a key-size guess but rather a smaller part of the key, usually a nibble of a byte which allows to test all hypothesis exhaustively. When the greatest dependency is obtained for the correct key value, noted $k*$, then the key is broken.

Because of measurement noise, multiple (X, Z) couples are generally required to compute this dependency. Indeed, the leakage L can be decomposed into two parts a deterministic part $\varphi()$ and the noise N:

$$L = \varphi(Z) + N$$

where $\varphi()$ is the leakage function that links the computation to the measurable activity. The Noise N is assumed to have a Gaussian distribution ([22], [71], [47]). The observed leakage is generated by the computation in the CMOS cells (as explained in Sect. 2.6.2). Therefore, the leakage can be assumed to come from the changing of state. By using a known state (the initial plaintext for example) and comparing to an unknown state (after the key has been added), it is therefore possible to use the *Hamming Distance Model* (HD):

$$\begin{aligned} L &= HD(Z, R) + N \\ &= HW(Z \oplus R) + N \end{aligned}$$

where R is the reference, known, state.

When R is initialized to zero, it then becomes the *Hamming Weight Model* (HW):

$$L = HW(Z) + N$$

Once the appropriate attack and leakage model has been chosen, the next step is to use a distinguisher to test the dependency between the model and the measurements. In the next section, we review some common distinguishers found in the literature.

2.6.5 Distinguishers

Distinguishers are the mathematical characteristics being used to analyse the leakage values and narrow in on the relevant value while discarding the random values mostly due to noise. There are multiple distinguishers but the most commonly used are based on correlation. Nonetheless, other distinguishers have been proposed [32], [90], and have shown interesting results.

2.6.5.1 Difference of Means

This distinguisher is used in the Differential Power Analysis (DPA) [43]. As with most power-based attacks, the first step is to collect power traces. These traces are collected each with a different random plaintext and the same secret key. The attackers then has to guess a value for the key k by predicting the value of one bit Z_j of the sensitive variable. The leakage measurements L are separated in two partitions according to the hypothetically predicted value $Z_j = 0$ or $Z_j = 1$ computed for each key hypothesis. Each hypothesis $Z_j = 0$ and $Z_j = 1$ therefore results in two separate expectation (or mean) $\mathbb{E}[]$. The difference of mean of those two partitions is:

$$\Delta(k) = \mathbb{E}[L|Z_j = 0] - \mathbb{E}[L|Z_j = 1]$$

The $k*$ value for which this difference of means $\Delta(k)$ is maximised is the most likely value of the key. This method specifically applies to a single bit but can easily be generalised to a guess on a multi-bit value, typically 4 or 8.

2.6.5.2 Pearson Correlation Coefficient

This distinguisher is used in the Correlation Power Analysis (CPA) [21], it is strongly based of the DPA. The attacker predicts the appropriate leakage model function $\varphi(Z)$, then estimates the Pearson correlation coefficient ρ_k for every key guess k:

$$\rho_k = \frac{Cov[L; \varphi(Z)]}{\sigma_L \times \sigma_{\varphi(Z)}}$$

where $Cov[]$ is the covariance and σ_L and $\sigma_{\varphi(Z)}$ are respectively the standard deviation of the physical leakage and of the leakage model. DPA and CPA are very similar, the main advantage CPA has is that it reduces the impact of noise on the power traces, meaning the key will break with fewer traces.

2.6.6 Leakage Channel Metrics

The leakage of a cryptographic device needs to be analysed, to do so some metrics exist that will indicate how the leakage behaves. For instance, it will indicate where the leakage is most likely to be relevant to the secret and not just noise.

2.6.6.1 Signal to Noise Ratio

The signal-to-noise ratio (SNR) is used as a leakage metric to estimate the usability of a specific leakage point. It is defined as:

$$SNR = \frac{Var[\varphi(Z)]}{Var[N]}$$

where $Var[]$ is the variance and N the noise.

2.6.6.2 T-test

The T-test [76] is a type of statistical test which compares the mean of two groups. This test works in many fields of research and is often used, due to its practical approach to statistical problem solving. The specificity of this test is its ability to identify statistical differences by using small sample sizes.

Its principles are summed up in the following equation:

$$t = \frac{\mu_1 - \mu_2}{\sigma\sqrt{(1/n_1) - (1/n_2)}}$$

where t is the t-statistic, μ_1 and μ_2 are the means values of the studied samples, σ is the standard deviation for the two groups and n_1 and n_2 are the amount of samples of each group. The t value represents the probability that the difference in means is statistically significant. Thereby, the bigger t is, the more the difference in means in the samples is representative of the entire results.

In cryptanalysis, this test can be used to compare the means of different datasets. This will lead to the identification of phenomenons which have a specific behavior during the execution. Indeed, when using power traces, the t-test will differentiate the noise from the statistically significant power consumption. This method will therefore lead to identifying the potential leakage points.

2.6.6.3 Normalized Inter-Class Variance

The NICV [17] is another metric to estimate the leakage points most likely to lead to an attack. It is the envelope, or maximum of all possible correlation computable from X with $Y \in \mathbb{R}$ (the leakage measured by the attacker). The NICV is defined so that:

$$0 \leq \rho^2[L(X); Y] \leq \frac{Var[\mathbb{E}[Y|X]]}{Var[Y]} = NICV \leq 1$$

where ρ is the Pearson correlation coefficient. This metric gives a frame of how and where the attack is most likely to succeed.

2.6.7 Attack Metrics

A side channel attacks can be more or less successful, depending on many factors such as the number of traces, the attack model or even some implementation consideration which make the attacks more complicated. Therefore, it is important to be able to evaluate the level of success of a given attack.

2.6.7.1 Minimum Traces to Disclosure

This metric is the average number of measurements needed to perform a successful attack.

2.6.7.2 Success Rate

This metric is defined by the probability of the best key guess to be the right key with a given set of traces. It therefore requires using multiple same-sized sub-sets and looking at the percentage of successful attacks.

2.6.7.3 Guessing Entropy

The Guessing Entropy (GE) works similarly to the success rate, except that it does not look at the percentage of success. Instead, it looks at the average rank of the right key among all the key guesses. Is allows for a more gradual evaluation of the results rather than the more binary principle of the success rate.

2.7 Introduction to Countermeasures

Side-Channel Attacks are a real threat to the security of any cryptographic design, therefore these devices should be equipped to resist such attacks. There are many countermeasure most of which are based on a few basic principles.

2.7.1 Hiding

The concept of hiding-based protection is to cancel out the observable dependencies created by the hardware during its execution. Rather than lowering the information that is leaking, these methods seek to augment the leaking as a hole. This has the effect of evening out the behavior making any information as significant as the secret, therefore undetectable.

2.7.1.1 Noise Generators

A big part of what takes side-channel attacks time to break an algorithm is getting rid of the noise. Therefore, one way to disrupt such attacks is to artificially add more noise, through a noise generator. The amount of traces required to break a cipher will thereby increase. This method has a very low area over-cost [42] but is theoretically weak as attacks are already designed to differentiate noise.

2.7.1.2 Activity Balancing

These countermeasures looks to compensate the energy consumption at a gate level by adding *hiding* logic to the *necessary* logic. Theoretically, this would make the energy consumption constant throughout the entire execution. In practice, the compensation is never perfect and information can still leak and be exploited by SCA. Nonetheless, this method is still quite common and can be found in the literature, where it can also be named dual-rail [24], [37].

2.7.1.3 Shuffling

This countermeasure is about randomizing the execution order of independent operations [67]. This method is very common as it requires very little over-cost in software implementations.

2.7.2 Masking

Masking one of if not the most used countermeasure. The basic principle of masking is to manipulate a data that is randomized or *disguised*. The mask is a given value (hidden or public) which is XORed to the manipulated value, the state for instance, in order to dissimulate any existing correlation. This can be done at gate or algorithmic level which makes it a flexible method. This method does have some issues such as the possibility of glitches [56]. Masking can be done at higher order, but will always be extremely weak to attacks one order above it. Amongst the existing countermeasures based on masking, we will detail how the Rotating S-Box Masking (RSM) works, which we used during this work.

2.7.2.1 Threshold Implementation

The Threshold Implementation [55] is a masking-based protection which works on transformed subparts of the input, called *shares*. Let $(x, y, ...)$ be the input vector such that each variable can be divided into n *shares* x_i with $x = x_1 \oplus x_2 ... \oplus x_n$. The basic principle is that if instead of manipulating the entire input at a time, a part of the masked input is not taken into account then the manipulation can never be dependent of the input. This idea is based on three properties:

- Non-Completeness
- Correctness
- Uniformity

Non-Completeness means that every function is independent of at least one *share* of the input variable. Correctness ensures that the sum of the parts is always equal to the desired output. Uniformity ensures that the conditional probability distribution is uniform. In other words, the probability that a vector takes on a given value is the same, no matter the given value.

This method therefore allows manipulating secret data without ever exposing it all at once to the observations linked with the execution. This division along with the masking has the effect of making the observation unusable for an attack. The Threshold Implementation offers protection against first order attacks.

2.7.2.2 Rotating S-Box Masking

The Rotating S-Box Masking (RSM) protection [53] was designed with LBCs using S-Boxes in mind. Its basic principle is to make sure that the state is masked at all time, while changing the mask at each round. The 64-bit mask is divided in 16 nibbles, these nibbles do not change, but the 64-bit mask changes by rotating these nibbles at each round. Therefore, at each given round, the mask is a one nibble rotation of the preceding round mask. The initial rotation is determined by the RSM Index which is randomly chosen and securely stored at the beginning of each cipher.

Since the S-Box is a non-linear step of the algorithm, the state cannot be masked when using it. This is a huge problem since losing the mask at any point creates the possibility of leakage, and the S-Box is already a weak point for side-channel attacks. The solution is to incorporate the mask to the S-Box itself. This means that the S-Box table is modified from:

$$\mathbf{x} \mapsto \mathbf{S(x)}$$

to:

$$\mathbf{x} \oplus \mathbf{m_i} \mapsto \mathbf{S(x)} \oplus \mathbf{m_{i+1}}$$

where x is the S-Box input, S is the S-Box function, m_i is the current round's mask and m_{i+1} is the next round mask. That way, the state enters the RSM

S-Box while masked with the current round's mask and outputs it masked with next round's mask. This solution allows to both avoid the unmasking of the state before using the S-Box and takes care of the mask rotation.

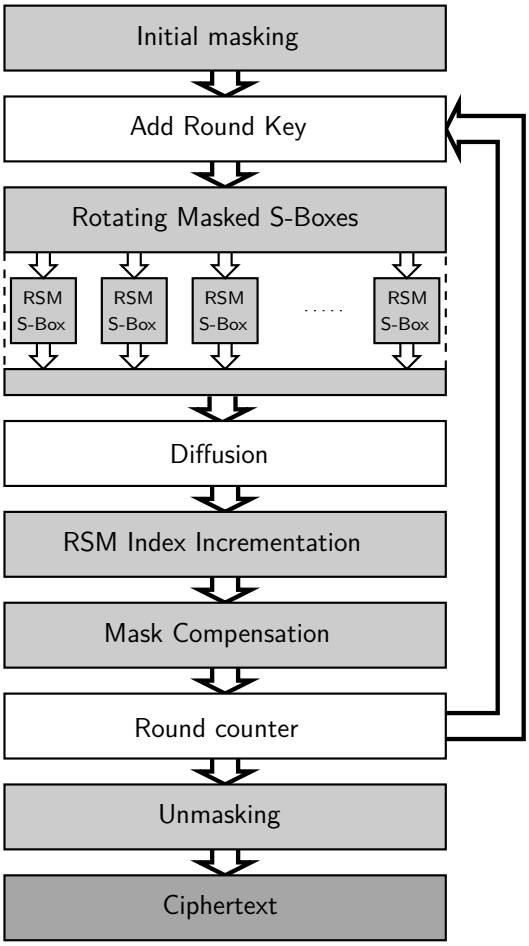


Figure 2.5: The RSM Datapath on a Generic LBC

The entire RSM datapath is shown in Fig. 2.5, it uses a generic LBC algorithm structure as an example. One of the requirements for the RSM protection is that the nibbles of the mask never change. To avoid modifications of the mask during the *Diffusion Step*, a compensation mask must be added to the state before the next round. The changes applied to the mask are linear and can therefore be corrected through a simple XOR with the right value. When the *Diffusion Step* is applied to the masked state, the output is $\mathbf{D(x \oplus m)}$ where D is the *Diffusion Step* function, m is the mask and x is the state. Since the *Diffusion Step* is linear, it is equivalent to $\mathbf{D(x) \oplus D(m)}$, therefore, the state is no longer masked with m but with $D(m)$. In order to have the state be masked with m once again, it must be XORed with $\mathbf{m \oplus D(m)}$. It is essential that the state is XORed with the entire compensation mask at once, since XORing the state with only **D(m)** would unmask the state. Once this compensation mask is applied, the state goes back to $\mathbf{D(x \oplus m)}$ and the cipher can go on to the next round.

2.8 RISC-V

The RISC-V is an open-souce Instruction Set Architecture (ISA) [91] based on the reduced instruction set computer, or RISC principles. It is at the heart of the RISC-V Foundation [66] which regroups a large community of academics and industrials working on implementing RISC-V processors in many different ways and for many different purposes. The free access to the RISC-V and many tools developed by its community makes it a perfect fit for academic study.

2.8.1 History

The RISC-V project started in 2010 at the University of California, Berkeley [58] by Prof. Krst Asanović and graduate students Yunsup Lee and Andrew Waterman. It was initially funded by the UC Berkeley PerLab Industrial sponsors for research on parallel processing systems. The open-source nature of the project led to a fast growth.

Amongst the specificities of the RISC-V is the fact that it is only an ISA, meaning that no given implementation is required, which offers a lot of freedom. Indeed, the technology itself has existed for decades but never before had it been open-sourced. Giving everyone free access to the same unified ISA meant collaborative work would not only be facilitated but also made available for future work. Any implementation, any extension, any enhancement could be shared and any further work could be built on that rather than having to start from scratch. This led to the development of many RISC-V ISA cor implementations ([3], [7], [78] to name a few) which each have their own specificities.

This flexibility along with the risc architecture and the open-source aspect, makes RISC-V a very interesting candidate for any embedded technology. RISC-V is therefore of interest to a number of industrial companies, part of the RISC-V Foundation [65]. Today, the RISC-V Foundation is the organisation which regroups all the information about the project and finances conferences to help spread the research made with this ISA.

2.8.2 RISC-V Base Variants and Extensions

There are multiple Base Variants of the RISC-V meant for different sizes and different uses, they are described in Tab. 2.2.

Table 2.2: Base Variants of the RISC-V ISA

Name	Description	Number of Registers	Instruction Count
RV32I	Base Integer Instruction Set 32-bit	32	49
RV32E	Base Integer Instruction Set (embedded) 32-bit	16	49
RV64I	Base Integer Instruction Set 64-bit	32	14
RV128I	Base Integer Instruction Set 128-bit	32	14

Moreover, a large part of the design for the RISC-V is based on the fact that it can be extended. The goal of these extensions is to allow users to have access

to the most compact possible implementation they would need. This means that none of these extensions is mandatory, which leaves the basic RISC-V quite small and lets the user modulate it for its needs. Some of these extensions have even been made official by the RISC-V foundation and are presented in Tab. 2.3.

Table 2.3: RISC-V Foundation Standardized Extension of the RISC-V ISA

Name	Description	Instruction Count
M	Standard Extension for Integer Multiplication and Division	8
A	Standard Extension for Atomic Instructions	11
F	Standard Extension for Single-Precision Floating Point	25
D	Standard Extension for Double-Precision Floating Point	25
G	Shorthand for the base and above extensions	n/a
Q	Standard Extension for Quad-Precision Floating Point	27
C	Standard Extension for Compressed Instructions	36

There are other existing extensions which have not been made official by the RISC-V foundation. Among them is the RISC-V Crypto Extension Proposal [94] which looks to accelerate the execution of standardized hash functions and the AES cipher. This project had not started when our research started, and ours took a different path as it focuses on non-standardized Lightweight Block Ciphers. In addition to this proposal, the Standard Extension for Bit Manipulation [45], [4] is also an interesting extension to look at as most of our work seeks to accelerate the execution of LBC which are often slowed down by their bit-level functions.

Based off these extended ISAs multiple Cores were implemented, each with their specificities.

2.8.3 RISC-V Cores

There are many existing implementations of this ISA, it is important to select the one that most fits your criteria. The RISC-V Foundation maintains a comprehensive list of compliant cores and platform on the official foundation website [66]. Here is a short presentation of some RISC-V cores that we considered during this study.

2.8.4 RISC-V Rocket Core

The RISC-V Rocket Core [7] uses the RV32 ISA. The Rocket Core is written in Chisel [25], a hardware construction language based on the Scala programming language. Chisel allows to have an easy to configure architecture with a variety of parameters which allow to propose many implementations, among which are multi-core implementations. Among these parameters are the *knobs* which are simpler to change:

- Tiles: Number, type
- Memory: Physical/Virtual address bits
- Caches: Set, ways, width for L1 and L2

- Core: Floating Point Unit, Frequency
- Uncore: Coherence Protocols

It is possible to implement new extensions for this core, but, as this design targets high performance multi-core systems, we found that we will need to modify too many aspects for our low complexity target.

2.8.4.1 SiFive

SiFive [74] is a fabless semiconductor company which specialises in the development of RISC-V Cores. Their portfolio includes a variety of Cores for a variety of use. Their main families of cores are:

- **E** Cores: 32-bit embedded cores
- **S** Cores: 64-bit embedded cores
- **U** Cores: 64-bit application processors

Their P550 Series's performance is announced to be $3\times$ the performance of an ARM Cortex A75 per mm^2. Some cores have been validated on silicon and development board are commercially available. SiFive is also one of the most active contributors to the open source software environment (compilers, operating systems) for the RISC-V.

2.8.4.2 Pulp/Pulpino

The PULP project [63] includes multiple implementation as shown in Fig. 2.6 (which was taken from their website), with different target applications. The PULP is a silicon-proven Parallel Ultra Low Power platform targeting high energy efficiencies. Its platform is organized in clusters of RISC-V cores that share a tightly-coupled data memory. The PULPino and PULPissimo contain only a single RISC-V core, there is no multi-core support. The PULPino is lighter but the PULPissimo is more advanced with extensions for effective signal processing.

Each of them is implemented in SystemVerilog. PULPissimo supports the use of multiple variants of the RISC-V:

- RV32I
- RV32C extension
- RV32M extension
- It can also be configured to support RV32F extension

2.8.4.3 OpenHW group

OpenHW group [36] is a non-profit organisation which seeks to develop open-source cores, related IP, tools and software. They are currently developing two implementations of the RISC-V, CV32E40P [34] and CVA6 [35] which are directly related to the Pulp cores families.

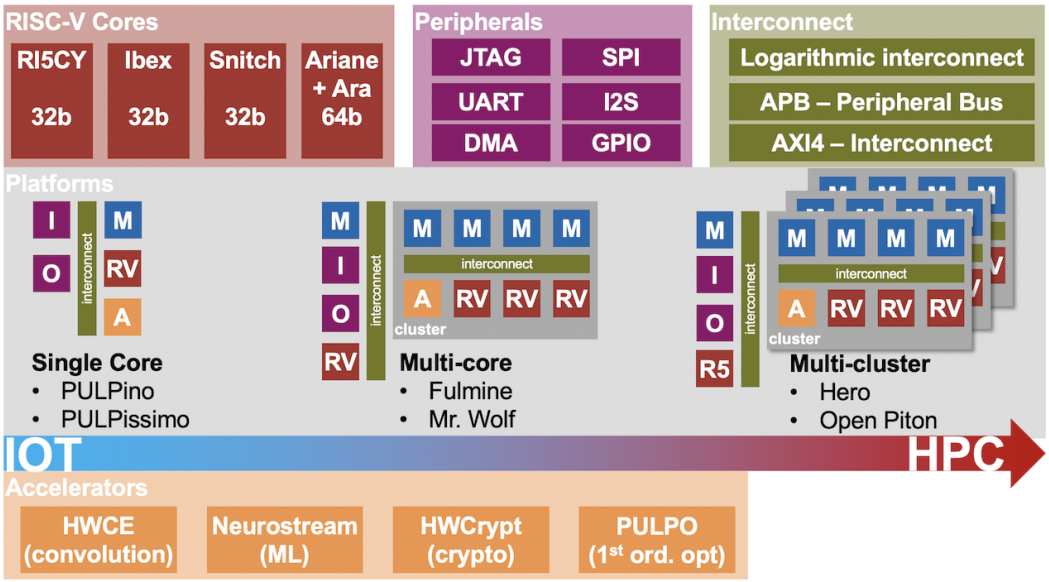


Figure 2.6: The PULP Family (taken from the PULP website [63])

- The former is a 32-bit, in-order RISC-V core with a 4 stage pipeline that implements the RV32IM[F]C variation.
- The Latter is a 64-bit, in-order RISC-V core with a 6 stage pipeline that implements the RV64IMAC variation.

2.8.4.4 Syntacore

Syntacore [79] is a semiconductor IP company which specialises in customizable RISC-V ISA based microprocessors. They offer a large family of cores called SCR which are described with varying size and RISC-V extensions available. Some of these implementations, SCR5 and SCR7, support full OS.

Among them, the SCR1 is the only open-source one [80] and its specificities are:

- RV32I or RV32E ISA base with optional M and C extension available.
- 2 to 4 stage pipeline
- Written with SystemVerilog
- Optimised for area and power
- Verification suite provided

These characteristics are part of what we were interested in for our work implementation. Nonetheless, no easy way to add custom instructions is provided and would therefore require the core modifications

2.8.4.5 Alibaba/T-head

The T-Head [81] is the semiconductor chip business entity of Alibaba [6]. It sells multiple silicon valid chips amongst which the 9 series uses the RISC-V ISA. They each support a large variety of RISC-V variations, some of which are summed up in Tab. 2.4.

Table 2.4: RISC-V ISA Variation Supported By the T-HEAD 9 Series Chips

Model	Supported RISC-V Variation
E902	RV32E[M]C
E906	RV32IMA[F][D]C[P]
C906	RV64IMA[F][D]C[V]
C910	RV64GC

V is the Standard Extension for Vector Operations, it adds 186 instructions and has not yet been made official by the RISC-V foundation.

2.8.4.6 VexRiscv Core

The VexRiscv Core [3], among the numerous available cores based on a RISC-V ISA, integrates the possibility to easily add personalized instructions by means of a plug-in mechanism.

The VexRiscv Core is written in SpinalHDL [5], SpinalHDL is an open-source (LGPL/MIT License) high-level hardware description language and RTL generation tool set based on the Scala programming language. The SpinalHDL description is used to generate Verilog RTL netlists which can then be used in a standard digital design flow.

The main specifications of the VexRiscv Core are:

- 32-bit RISC-V ISA
- Instruction Set extensions support: RV32I[M][C][A]
- 5 stages pipeline
- Separated instruction and data memory interfaces
- Debug extension and standard jtag interface
- Optional hardware MUL/DIV extensions

The high level implementation in SpinalHDL of the Vexriscv processor allows the addition of user defined instructions through a plug-in mechanism.

For simple register to register instructions which can be executed in one cycle, the designer can add, in a single SpinalHDL file the plug-in description. This description should include the instruction opcodes to match in the decode stage, how the instruction should be inserted in the processor pipeline and the additional computation logic.

The following SpinalHDL code extract shows a commented example of plug-in to add a custom instruction.

```
class customInstPlugin extends Plugin[VexRiscv]{
  //....
  // -> The plug-in configuration
  override def setup(pipeline: VexRiscv): Unit = {
    //....
    // -> We add an entry in the decoder service
    decoderService.add(
      key = M"0000011----------000-----0110011",
      // -> List of contole signals that will be activated
      List(
        IS_CUSTOM_INST           -> True,
        REGFILE_WRITE_VALID      -> True,
        BYPASSABLE_EXECUTE_STAGE -> True,
        BYPASSABLE_MEMORY_STAGE  -> True,
        RS1_USE                  -> True,
        RS2_USE                  -> True
      )
    )
  }

  // -> How does the plug-in interacts with the processor pipline
  override def build(pipeline: VexRiscv): Unit = {
    // ....
    // -> A new scope on the execute stage with internal signals
    execute plug new Area {

    val rs1 = execute.input(RS1).asUInt
    val rs2 = execute.input(RS2).asUInt

    val rd = UInt(32 bits)
    // -> here is the combinational logic of the custom instruction
    rd := rs1 + rs2

    // -> propagate the result to the output of the EXEC stage
    when(execute.input(IS_CUSTOM_INST)) {
      execute.output(REGFILE_WRITE_DATA) := rd.asBits
    }
    }
  }
}
```

As is, this description is incomplete but, you can refer to the VexRiscV official documentation [3] for a complete and functional plug-in example.

Without going too deep into SpinalHDL specifics, the main steps to define a plug-in are:

1. define a new class that extends the `Plugin` class,

   ```
    class customInstPlugin extends Plugin[VexRiscv]{
   % }%%TODO remove before submission
   ```

2. add an entry to the decoder service that matches the opcode we want to use,

   ```
   decoderService.add(
     // -> the opcode we want to match
     key = M"0000011----------000-----0110011",
     ...
   %  )%%TODO remove before submission
   ```

3. specify how the instruction will interact with the processor pipeline, the register file input/outputs that will be used and additional control signals,

```
IS_CUSTOM_INST           -> True, // -> It is our instruction
REGFILE_WRITE_VALID      -> True, // -> We write the result to the reg file
BYPASSABLE_EXECUTE_STAGE -> True, // -> The instuction will be executed in
    one cycle
BYPASSABLE_MEMORY_STAGE  -> True, //    We can bypass EXEC/MEM stages
RS1_USE                  -> True, // -> We will use both register file
    outputs
RS2_USE                  -> True  //    RS1 and RS2
```

4. describe the instruction logic in the execute stage.

```
// -> rs1/rs2  nets are connected to the inputs of the EXEC stage
//    that correspond to the outputs of the register file
val rs1 = execute.input(RS1).asUInt
val rs2 = execute.input(RS2).asUInt

// -> declares a net for the output
val rd = UInt(32 bits)
// -> here is the combinational logic of the custom instruction
rd := rs1 + rs2
```

5. and finally, redirect the output to the register file

```
when(execute.input(IS_CUSTOM_INST)) {
  execute.output(REGFILE_WRITE_DATA) := rd.asBits
    }
```

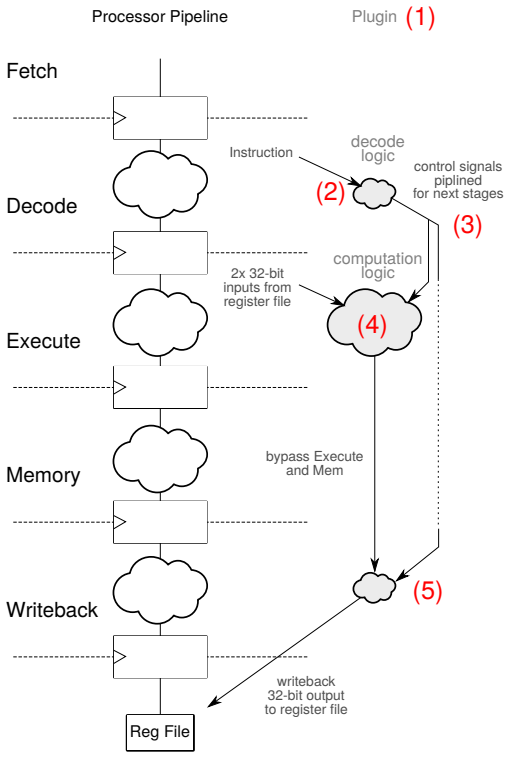


Figure 2.7: The Plug-in Insertion in the VexRiscv Pipeline

When the RTL code is generated, the plug-in decode and execution logic will be automatically inserted the different sub-parts of the processor. Figure 2.7 shows

how the plug-in description is spread into the processor pipeline. For simple register to register instructions, will get their inputs from the register file at the beginning of the Execute stage and the result will be written back to the register file one clock cycle after. Bypass logic is automatically added as the following pipeline stages are not needed here. For more complex multi-cycle instructions, the plug-in mechanism, allows to specify when the result will be ready and eventually add additional stall logic when necessary.

Despite the fact that SpinalHDL is not the most common hardware description language, the plug-in mechanism, makes this core easy to customize without having to maintain a completely different version. Also, regarding the core performances, the VexRiscv Core won the "RISC-V SoftCPU Contest" [31] back in 2018, which rewarded small, powerfull and inovative softcore RISC-V implementations.

2.8.5 Comparison of different Core Sizes

We considered diverse cores for our work, and used a few criteria to chose which one most fit our needs.

- Size, we wanted to minimize all area costs
- Flexibility, we canted to be able to extend the RISC-V without having to change the core

In terms of flexibility, the VexRiscv Core has a plug-in mechanism which allows extensions with a single file (*Cf.* Sect. 4.6.1). This put it very high on our radar, but size was the most important factor.

We therefore had to compare the cost of implementation for some of these cores. The chosen cores focused on having a small size, we therefore only compared the VexRiscv Core, the SCR1 and the Pulpino. The results in Tab. 2.5 were obtained by using the Cadence gpdk045 library. They are divided in two frequency constraints, a very low frequency of 25MHz and a 150MHz frequency which is more than the working frequency we expected. The synthesis results were obtained for as close configurations as possible. The VexRiscv Core uses a 5 stage pipeline and was configured to enable bypass between instructions and branch anticipation. The SCR1 was configured so as to use a 4 stage pipeline and have no interruption controller nor cabled multiplication. The Pulpino uses a 2 stage pipeline, has an interruption controller but no cabled multiplication.

Table 2.5: Synthesis Results of Different Cores at Different Working Frequencies

Frequency	Core	Total Cells	Area(μm^2)
25MHz	VexRiscv	5773	22406
	SCR1	8414	23246
	Pulpino	10090	27408
150MHz	VexRiscv	6241	22770
	SCR1	10025	27049
	Pulpino	11176	30209

These results show that the VexRiscv is small compared to other open-source cores. This further confirmed our choice of working with the VexRiscv Core to extend the RISC-V.

2.9 Conclusion

This State-of-the-Art has shown that a plethora of Lightweight Block Ciphers exist. Each of these ciphers has its own specificities and an agile implementation would require walking the line between agility and cost.

Although ciphers appear very different, the three steps model accentuate their similarities. This lead to focus our agile approach on the differences between these steps in each cipher.

While these ciphers are mathematically secured, Side-Channel Attacks have shown that none are safe from eavesdropping. This means that implementing protection measures is essential to the security of a modern implementation. RSM and the Threshold Implementation, offer protection against first order attacks but, we found that RSM is much lighter in terms of resources [53] and easier to implement as an agile hardware protection. And since area is one of our prime concern, we chose to work with the RSM protection.

Hence, a hardware approach seems like a right way to accelerate execution of any protected cipher. Nonetheless, the existence of an open-source extendable ISA like that of the RISC-V does lead to also consider a hybrid approach which benefits from the efficiency of hardware implementations while maintaining the ease of use and versatility of software ones. Finally, the VexRiscv Core, with its plug-in mechanism, offers a perfect means to develop RISC-V extensions by adding hardware based instructions while maintaining a good level of performances in terms of speed and area.

Finally, a note on why keys scheduling was not considered for this work. Each cipher has a very different key scheduling algorithm, meaning that finding a common ground for them would be overly complicated. Moreover, just as the key is securely stored, so can the round keys, which stay the same for as long as the key stays the same. Therefore, the key scheduling can be pre-calculated before the execution, and will not have an impact on the real-time latency. On the contrary, this process can even be tweaked to encompass extra calculation that unify the round key to have them all be 64-bit XORable values. This lead to being able to use the same simple *Key Addition Step* for each cipher. This is only possible in an environment where some software can be used, which is coherent with our project.

The next chapter will present the fully hardware implementation we initially worked on and which we used as a basis for the RISC-V extension.

Chapter 3

Fully Hardware Agile Implementation

The three steps model of LBCs (*Cf.* Sect. 2.4) allowed us to identify common aspects of multiple ciphers. This leads to finding the best way to accelerate the execution of many ciphers by keeping the overhead as small as possible. Indeed, the best way to minimize the overhead is to focus on parts that can be found in multiple algorithms and therefore only need one implementation to work for multiple accelerations. That is part of the reason why this first implementation only takes into account SPN ciphers. This same observation is coherent with the idea of an agile implementation, where similar functions would use the same hardware, rather than having specific hardware dedicated to each cipher. Our first implementation was designed to be fully hardware.

3.1 Implementation

The very first implementation was an independent configurable hardware architecture [82]. It was based exclusively off SPN structured ciphers with 4×4 Sbox and either a bit-level permutation or a nibble-level matrix multiplication. The configurable aspects where parameters in the permutation and/or the matrix multiplication, the S-Boxes, and the order in which each step was applied. Fig. 3.1 shows the general architecture of such an implementation, here the key scheduling is done before the execution and each round key is stored. The configuration is also done before the execution during the *algorithm loading step*. This step includes the parametrisation of each module and sub-module along with the parametrisation of the Rout_Mux which allows to choose the order in which each module is executed. In this figure, the **S** module corresponds to the S-Box (aka the *Confusion Step*), the **P** module corresponds to the P-Layer (aka the *Diffusion Step*), the **K** module corresponds to the Key Addition. Due to the round structure, each of these configurable rounds ends by storing the result in a dedicated state register.

This architecture was a first attempt at an agile architecture to accelerate the execution of LBCs, and served as a basis for the rest of the project. The implementation is therefore detailed in the rest of this Section.

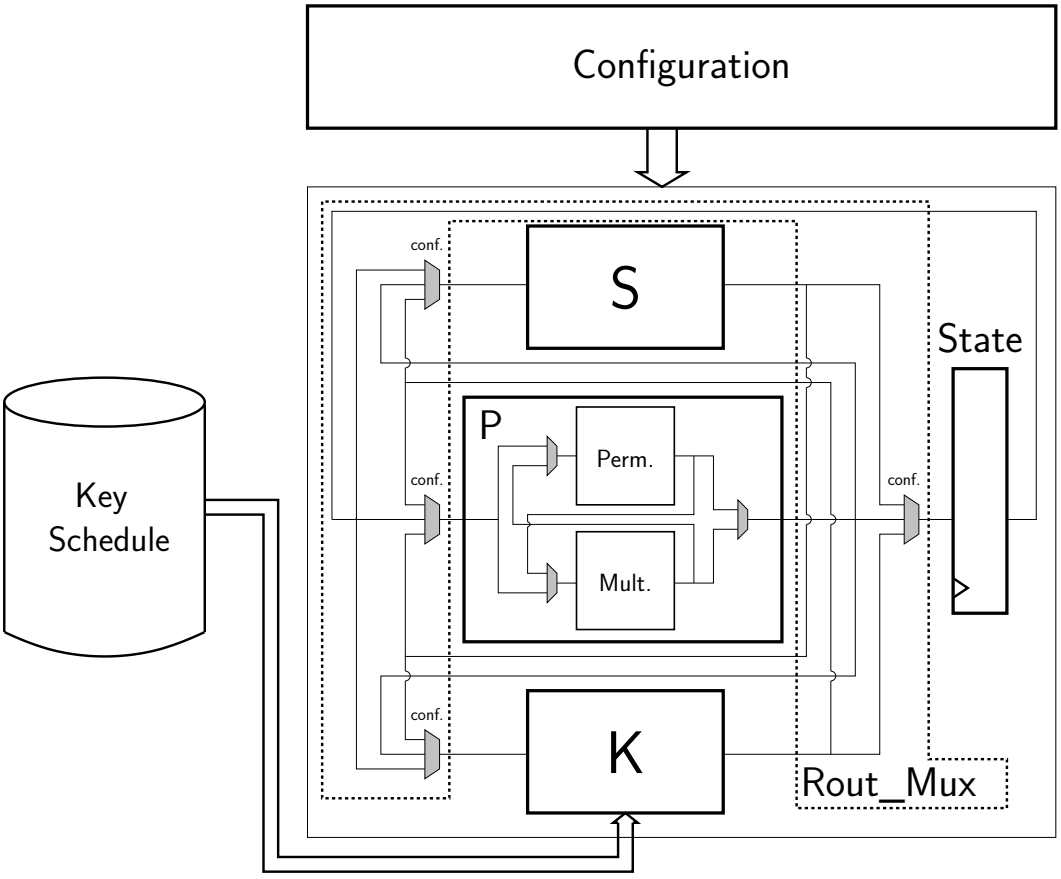


Figure 3.1: Fully Hardware Agile Architecture for SPN Ciphers

3.2 Configuration

The configuration parameters allow the agile architecture to select which algorithm will be executed. These parameters are set during the *algorithm loading step*, before the encryption begins. This allows the algorithm used to be changed dynamically. The parameters can also be changed during the execution, especially the Rout_Mux as most algorithms skip some modules during their first or their last round. Changing other parameters, such as those of the S-Boxes or the P-Layer can be useful for $\alpha - reflective$ algorithms but has a latency cost. The configuration memory bits are distributed throughout the architecture. A shift register mechanism is used (*Cf.* Fig. 3.2) to limit the complexity of the external configuration interface. Once the shift register's content is valid, the configuration data is written by block in a configuration register chosen by a selection signal.

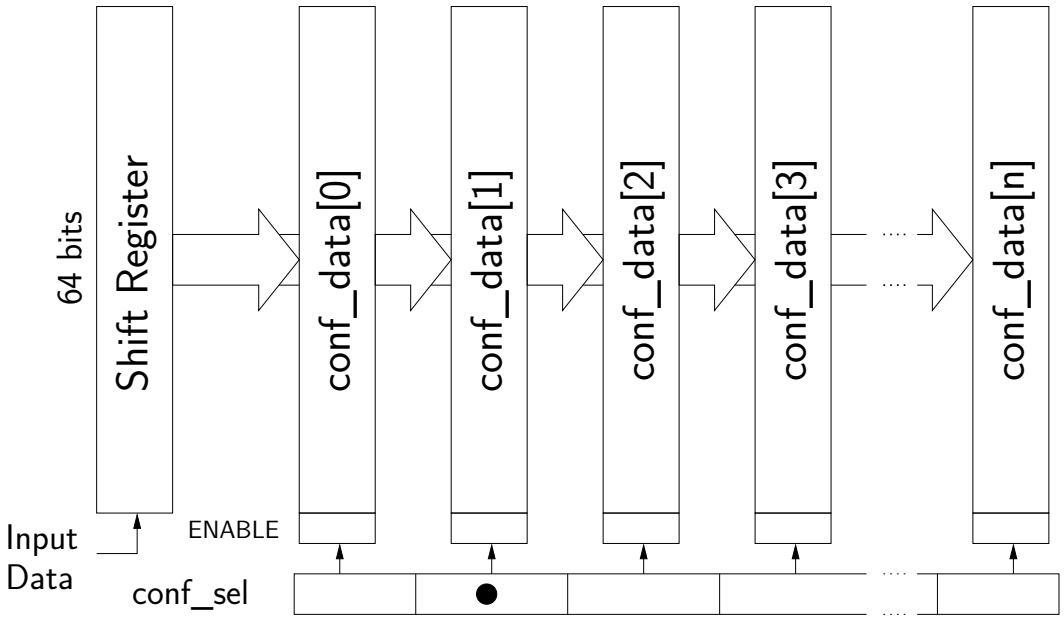


Figure 3.2: The Configuration Scheme

3.3 Instruction ordering

Although all the studied ciphers can be executed by using the same modules with different parameters, they use it in different orders. This order is defined during the configuration. The module usage is described in Table. 3.1 The **P** module

Table 3.1: Module Ordering for LBCs

Cipher	Module Ordering
PRESENT	$\mathbf{K} \rightarrow \mathbf{S} \rightarrow \mathbf{P}_{m}$
GIFT	$\mathbf{S} \rightarrow \mathbf{P}_{m} \rightarrow \mathbf{K}$
PRINCE	$\mathbf{S} \rightarrow \mathbf{P}_{mp} \rightarrow \mathbf{K} \mid \mathbf{S} \rightarrow \mathbf{P*}_{m} \rightarrow \mathbf{S'} \mid \mathbf{K} \rightarrow \mathbf{P'}_{mp} \rightarrow \mathbf{S'}$
SKINNY	$\mathbf{S} \rightarrow \mathbf{K} \rightarrow \mathbf{P}_{pm}$
Midori	$\mathbf{S} \rightarrow \mathbf{P}_{pm} \rightarrow \mathbf{K}$
MANTIS	$\mathbf{S} \rightarrow \mathbf{K} \rightarrow \mathbf{P}_{pm} \mid \mathbf{S} \rightarrow \mathbf{P*}_{m} \rightarrow \mathbf{S} \mid \mathbf{P'}_{mp} \rightarrow \mathbf{K} \rightarrow \mathbf{S}$

contains two sub-modules *Perm.* and *Mult.* the order of these sub-modules is represented as an index. $\mathbf{P}_{pm}$ means *Perm.* followed by *Mult.* and $\mathbf{P}_{mp}$ means *Perm.* followed by *Mult.*, $\mathbf{P}_{m}$ means only *Mult.* is used. Certain ciphers require mid-execution re-ordering and use different configuration for certain modules. When the configuration is changed, **'** or ***** is added to the module name.

3.4 K Module: The Key Addition

In the **K** module, the *round key* considered here is not necessarily what algorithms call their round key, it also encompasses the round constant if any. The *round key* is obtained through the key scheduling which is different for every algorithm and is calculated during the *algorithm loading step*. Those *round keys* are then

stored to be used by the **K** module. Thereby, the **K** module is simply composed of 64 XOR gates in parallel to add the state to the software precomputed round key. This precomputed round key actually corresponds to the cipher's round key XORed with potential round constants and adapted to a 64-bit format. The main security issue is that the round keys need to be stored in a secure environment, which is coherent with the natural use of encryption where the key needs to be stored securely. This mechanism also echoes the fact that the **K** module needs to access this secure memory once per round, therefore once per cycle. Memory does not usually have this feature but this can be handled by using a bypass between this section of the memory and the **K** module. That part was not implemented and is a theoretical solution.

3.5 S Module: The S-Box

The **S** module which corresponds to the *Confusion Step* is rather straight forward, for each of the 16 possible nibble inputs there is an output defined through information stored within a RAM bloc. This module is therefore similar to a LUT. The substitution of each of the 16 nibbles is done in parallel and therefore requires 16 actual S-Boxes.Most algorithms use the same S-Box throughout the entire encryption process but some of them require two different S-Boxes. This is the case with some $\alpha - reflective$ algorithms which use both a non-involutory S-Box and its inverse, such as PRINCE [19]. Including a second S-Box means having to store twice as much information and add a mechanism to switch from one S-Box to the other. This mechanism was not implemented.

3.6 P Module: The P-Layer

The *Diffusion Step* of an SPN structured algorithm can include two different parts. The first part, equivalent to the MixColumn function (from AES), is a matrix multiplication. The second part is a bit-level Permutation which is sometimes equivalent to the ShiftRow function (from AES).

3.6.1 Agile Matrix Multiplication

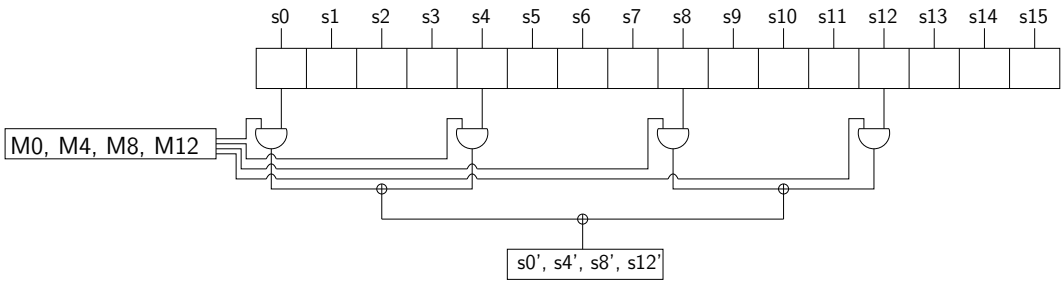


Figure 3.3: The Agile Matrix Multiplication

The matrix multiplication (*Cf.* Fig. 3.3) uses the same four bits of the input (s0, s4, s8, s12) with four different bit sets of the matrix (M0, M4, M8, M12) to

compute four bits of the output (s0', s4', s8', s12'). This means that the matrix multiplication uses four times 64 bits values or a unique 256 bits value. It seems like a lot but it is still smaller than a full 64×64 matrix, which requires 2048 bits.

The matrices used for the matrix multiplication are of two types. The first type is considering a 4×4 matrix composed of nibbles whose value is either F or 0 (SKINNY [14]). They are represented as 4×4 matrix but the multiplication actually applies to each bit of the state's 4×4 matrix of nibbles, they can therefore be considered as 4×4 matrix of *nibbles*. The second type is a 64×64 bit matrix which is mostly filled with 0s but has four 16×16 sub-matrices composed of 1s and 0s along its diagonal (Prince [19]). The latter matrix are themselves composed of 16 4×4 diagonal matrices, therefore the 16×16 matrix can only have 1s placed along the 4×4 matrix's diagonals (*Cf.* Fig. 3.4). This property makes it possible to reduce the 64×64 matrix (2048 bits of information) to the information on the 4×4 matrices' diagonals which compose the 16×16 sub-matrix. There are 16 4×4 matrix in each of the four 16×16 sub-matrices, each of which have 4 bits on their diagonal, which amount to a total of $4 \times 16 \times 4 = 256bits$ of useful information (*Cf.* Fig. 3.4). The first type only consists of a 4×4 matrix of nibbles with a single bit either at 1 or 0. In this type of matrix, the value of a bit is the same as the other bits of the same nibble, in other words the $4 \times 4 \times 4 = 64bits$ of the matrix can be summed up as $4 \times 4 = 16bits$ of useful information.

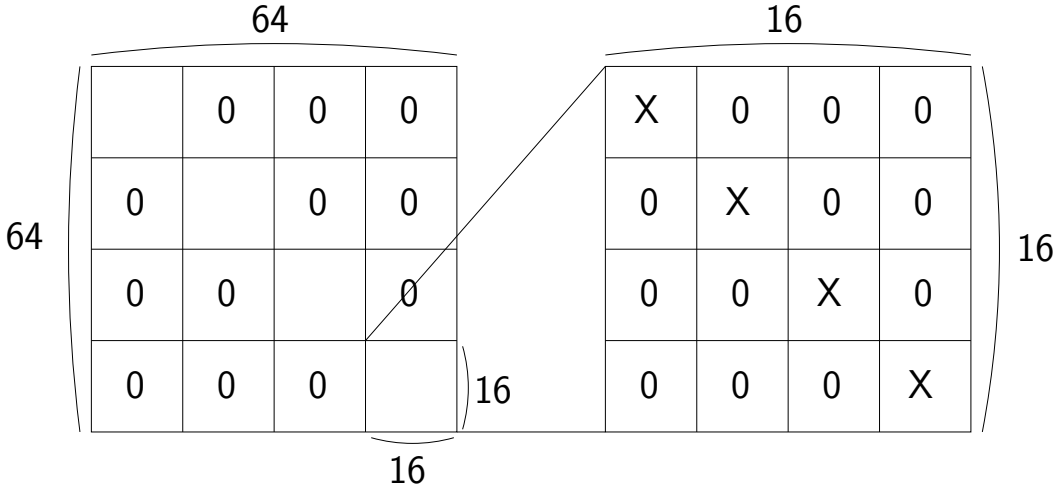


Figure 3.4: Turning the Useful Information of a 64×64 Prince-type Matrix into a 16×16 Matrix

This meant that either some algorithms were discarded in order to maintain a lower amount of required information, or they had to be harmonised. Doing so meant turning 16 bits of information into 256 bits without changing the result of the multiplication. This was achieved by changing every nibble of the 4×4 matrix into a 4×4 matrix, either filled with 0s if the nibble was a 0 or the 4×4 identity matrix if the nibble was a 1 (*Cf.* Fig. 3.5). The result is a 16×16 matrix with $4 \times 4 \times 4 = 64bits$ of useful information, which is the same as one of the 16×16 sub matrix of the 64×64 matrix. The 16×16 matrix thereby obtained was then duplicated four times in order to have the 256 bits of useful information as with

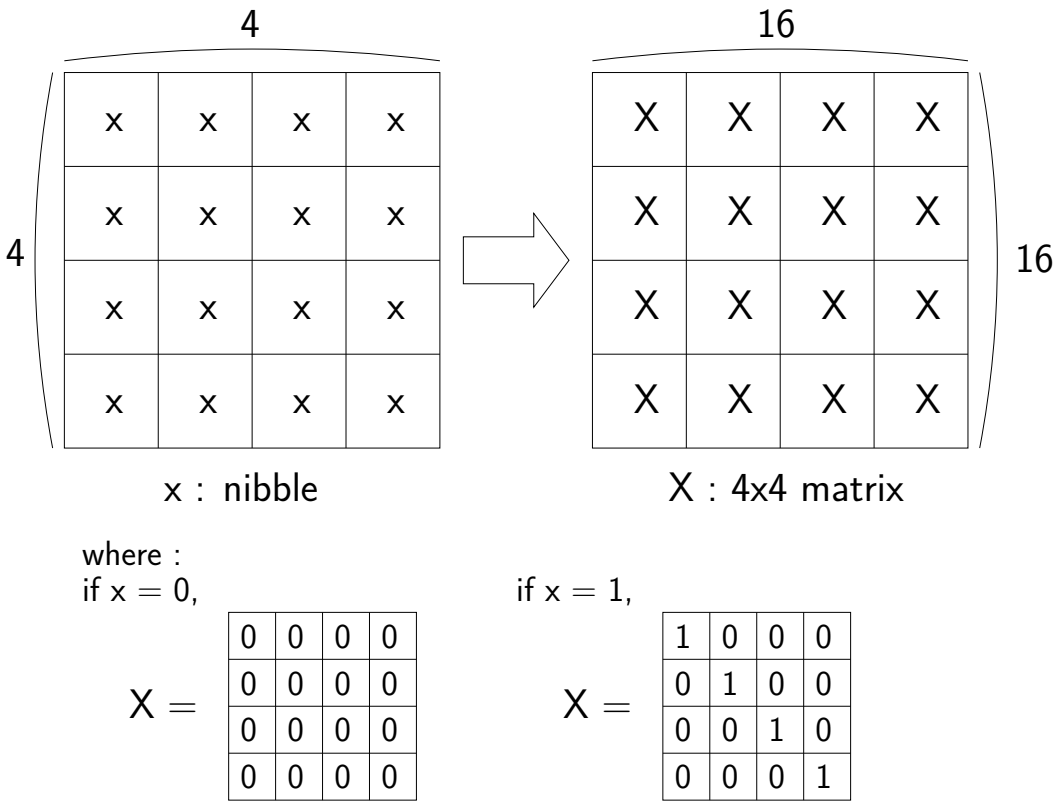


Figure 3.5: Turning the Useful Information of a 4×4 Matrix into a 16×16 Matrix

the 64×64 matrix. This choice is still costly as 256 bits is meaningful but is much less than the $64 \times 64 = 2048bits$ of the entire matrix. The question remains nonetheless on whether adding the second type of matrix is worth the cost, this will be discussed later.

3.6.2 Agile Bit-Level Permutation

The other sub-part of the **P** module is the Permutation sub-module. This part required a lot of attention for multiple reasons. First, in a classic LBC hardware implementation, this part is generally implemented as a simple reordering of wires and uses no specific hardware. Second, the algorithms which do not use a matrix multiplication require permutation at a bit level rather than the at the nibble level as with AES's Shift Row. Third, designing a configurable permutation meant being able to route a signal through a crossbar-like module, but crossbars are expensive in terms of both area and configuration memory. It was therefore essential to find a lighter solution. The most efficient solution to optimise those crossbars was using Banyan switches [33] as shown in Fig. 3.7.

The chosen solution (*Cf.* Fig. 3.6) was based on a thorough study of each algorithms' requirements in terms of permutation flexibility and the crossbar-like modules were optimised to better fit the situation (these optimised crossbar-like modules will further be referred to as Banyan switches). The Banyan Switches are

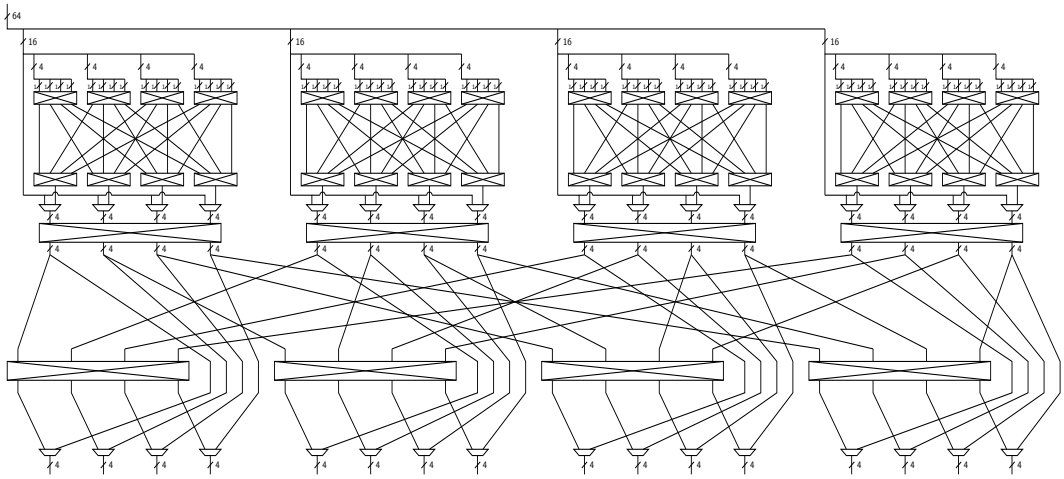


Figure 3.6: The Agile Bit-Level Permutation Instruction

composed of a set of five switches (*Cf.* Fig. 3.7). Each switch allows the reordering of two inputs and is controlled by a single configuration bit. The 4×4 Banyan Switch can thus be configured to reorder its 4 inputs to get any permutation at the output. The $4 \times 3 \times 2 = 24$ permutations can be controlled with only 5 configuration bits. This structure allows to reduce the area in terms of logic and configuration memory from 320 bits down to 200 bits.

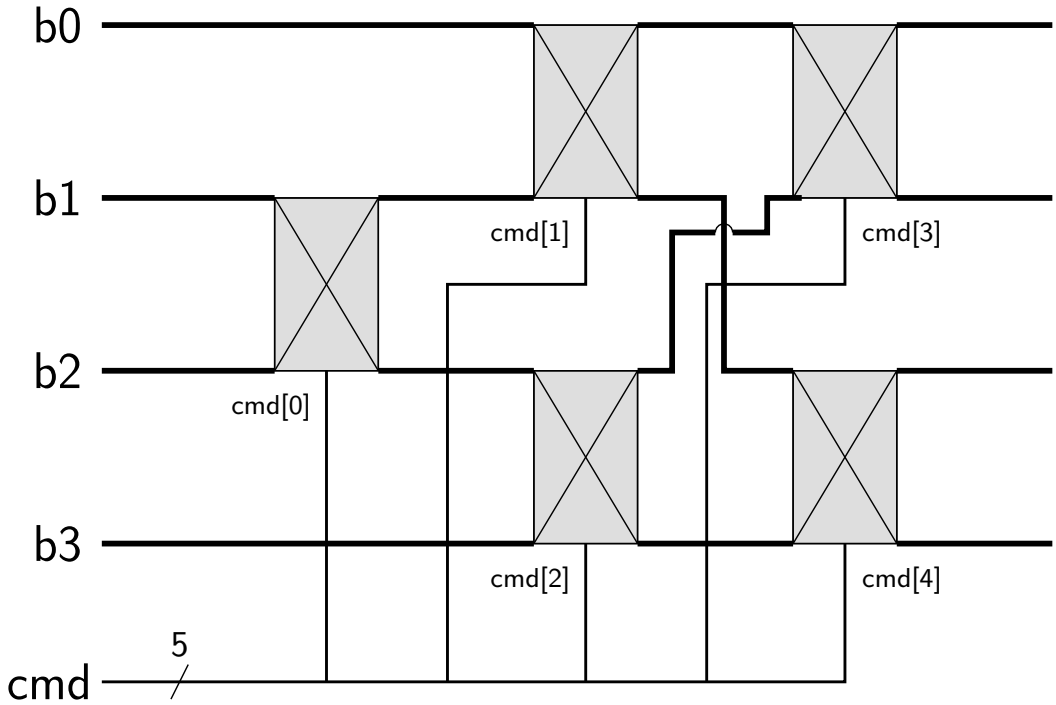


Figure 3.7: Optimised 4×4 Crossbar-Like Module, The 4×4 Banyan Switch

Permutation was a key issue as it is usually achieved by simply reordering the wires. The overhead of making this bloc configurable could thus be significant.

The simplest way to go is to consider a 64×64 crossbar which would allow any permutation but would be incredibly costly both in terms of area and in terms of the size of configuration memory. It was therefore essential to identify similarities between the different permutations in order to limit the area of this module. This was possible because these permutations are not random as they need to respect certain diffusion properties.

The result was two levels of permutations, the bit-level permutation and the nibble-level permutation. The bit-level permutation (PRESENT [18], GIFT [9]) allows any layer-input bit to end up as any layer-output bit within the same 16-bit word, the restriction being that two bits from the same input nibble may not end up in the same output nibble. This restriction is coherent with the diffusion properties of a cipher as a bit level permutation needs to spread the information as much as possible in order to ensure the diffusion. It requires four sets of Banyan switches each using the same parameters, which apply to each of the four 16-bit words of the 64-bits state. It is composed of two layers (*Cf.* Fig. 3.8). Between these layers, the connection wires are fixed and cannot be configured. They link each bit of a nibble to a different nibble. The first layer defines which bit of each nibble will be connected to which nibble of the second layer, through the use of a 4×4 Banyan switch for each nibble. The second layer reorders the bits within each nibble with a 4×4 Banyan switch for each nibble.

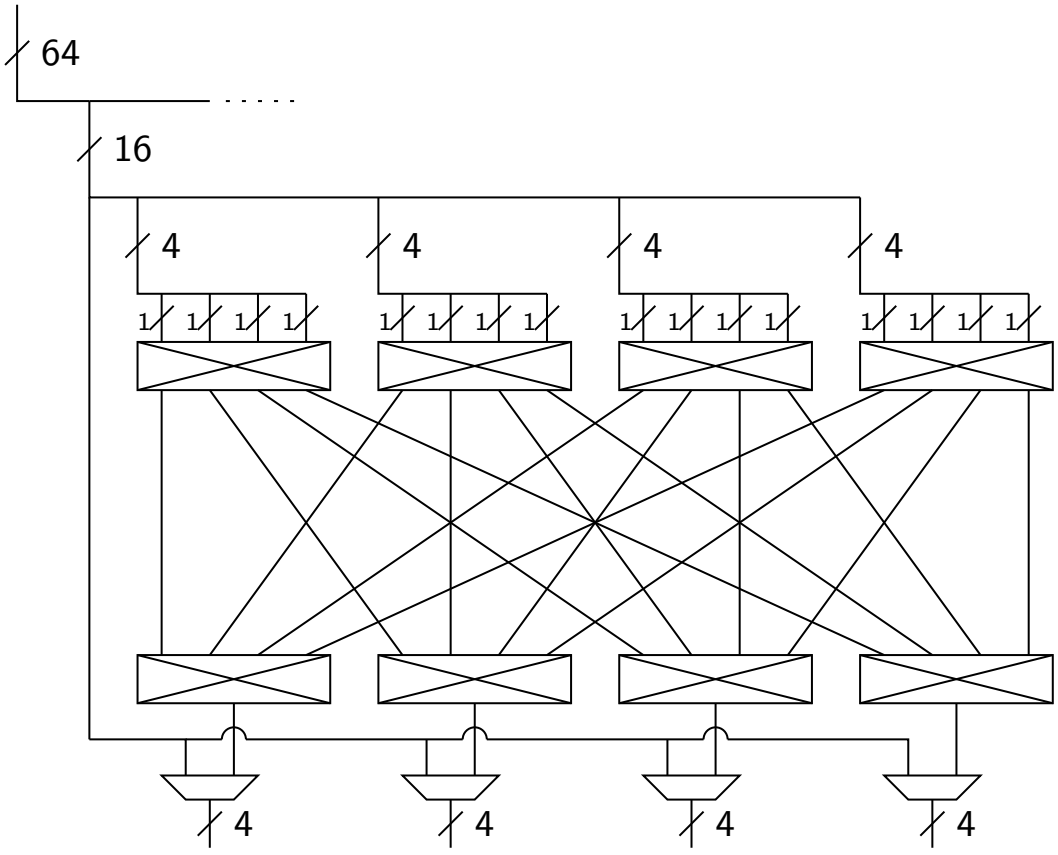


Figure 3.8: Bit-level permutation layer

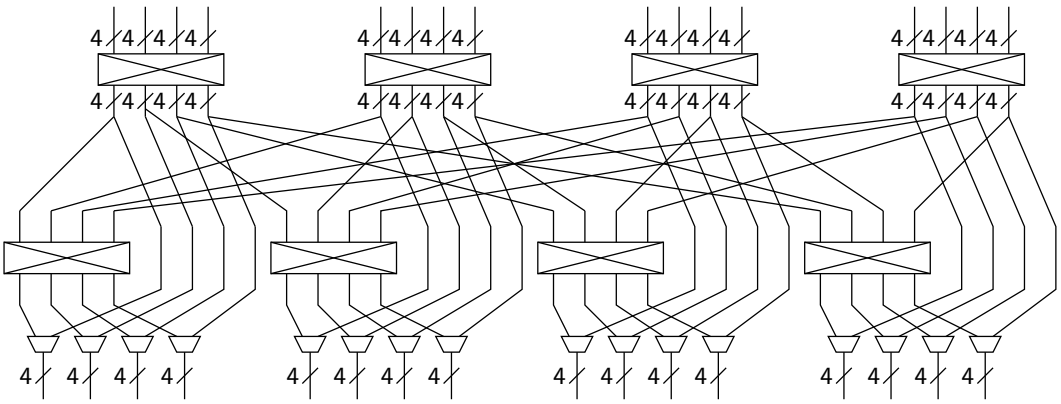


Figure 3.9: Nibble-level permutation layer

Nibble-level permutation (*Cf.* Fig. 3.9) works similarly and therefore, once again, any layer-input nibble may end up as any layer-output nibble and the restriction is that two nibbles of the same 16-bit input may not end in the same 16-bit word output. There is an exception to this rule in the case of an actual Shift Register where each nibble is reordered but every nibble stays within the same 16-bit word. It also requires two layers of Banyan switches separated with transition wires, which can be configured. There is a set of multiplexers which allows either to connect the four nibbles of a 16-bit word to four different 16-bit words or to keep each nibble within the same 16-bit word, which is needed for the Shift Row function. The first layer defines which nibble goes to which 16-bit word, and the second layer reorders each 16-bit word. They each use four 4×4 nibble Banyan switches.

3.7 Implementation Results

The architecture was implemented targeting the Cadence Free45PDK standard cells library. Post synthesis results are used to evaluate the area and complexity of our design.

First the agile architecture complexity is evaluated for different levels of agility. Second the agile architecture is compared to the cost of implementation of classic LBCs to identify the gain of using such an architecture.

The agile architecture can be divided in multiple sub-parts which have been presented in detail in this Section. The results of Table 3.2 show the cost of each of these parts. Making the architecture agile has an important cost. For instance, Permutation is usually free in terms of area but making it configurable will obviously make it more costly. It is also true for the **S** module which requires a configurable table and cannot be optimised through the use of specific logic functions. The other two main parts are also new to such an architecture as they do not exist in a non-agile implementation of cryptographic algorithms. The Route_Mux allows to order each step at each round and therefore uses an important amount of multiplexers in order to select the path for the entire 64-bit state. Finally, the most costly sub-part is the configuration which gathers all the parameters used

Table 3.2: Cost of Architecture's Sub-Parts for the Level of Agility **C**

Sub-part		Cost		Area percentage
		Area	GtE	
Configuration	Route_Mux	695	678	26.9
	S-Box	348	339	
	Permutation	956	932	
	Multiplication	1390	1355	
	Other	87	85	
Route_Mux		2791	2720	21.6
S		1784	1739	13.8
P	Permutation	2059	2007	21.7
	Multiplication	744	725	
K		175	171	1.4

to select which algorithm is implemented within the architecture. This last sub-part is divided between the different aspects which require configuration. It appears that the Multiplication requires the most important part of the parameters. Indeed, configuration of the Matrix multiplication has a cost of 256 bits (*Cf.* Section 3.6.1) which is a lot more that the 64 bits required for the **S** module or the 176 bits used to define the algorithm's route at each round. The overhead of the agile architecture is therefore important but most of it is due to the very nature of an agile hardware architecture which has incompressible costs. It would therefore seem that this architecture is not efficient when compared to a single algorithm but, the more algorithms it implements, the more interesting it becomes.

The next step was thereby comparing different levels of agility in order to identify how much adding new algorithms costs. This will then lead to a comparison between the cost of the agile architecture and the cost of implementing multiple algorithms.

Table 3.3: Different Agility Levels of the Architecture

Algorithm	Level of Agility
PRESENT	**A**
GIFT	
SKINNY	**B**
Midori	
PRINCE	**C**
MANTIS	**D**

Each level of agility, **A**, **B**, **C** and **D** from Table 3.3 allows the implementation of a certain set of algorithms. Each of these levels is compared to the cost of each of the algorithms it can handle in Table 3.4. The cost is given en Gate Equivalent (GtE).

Figure 3.10 illustrates the complexity reduction provided by the agile architecture when the level of agility increases. It shows that balance is achieved around an

agility of four algorithms and that once this limit is exceeded, the agile architecture offers a real gain.

Table 3.4: Comparison between different levels of agility and the sum of the algorithms it can implement

Level of Agility	Area of the Agile Architecture (in GtE)	Sum of the Implementations (complexity ratio)
A	8494	1.72
B	8245	1.96
C	9212	1
D	9631	0.625

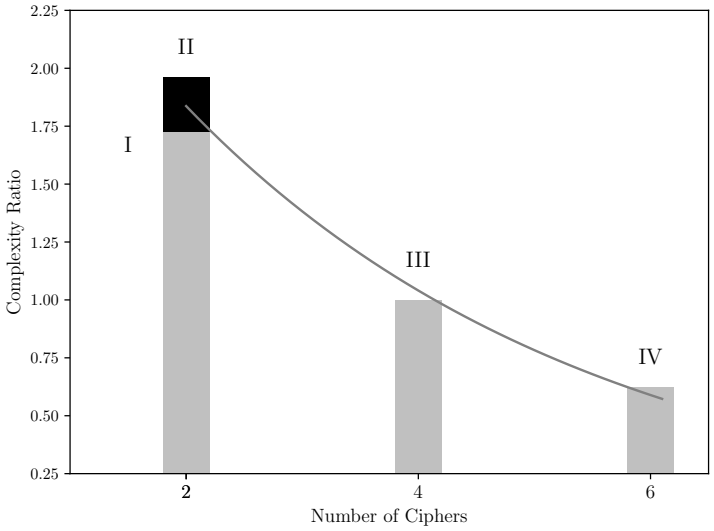


Figure 3.10: Complexity Ratio for different levels of agility

3.8 Conclusion

This first implementation showed how costly agility can be and that some costs were incompressible. Nonetheless, the results were promising as it did show how an agile architecture could offer an area gain if the level of agility was high enough.

This first implementation does have some issues. One being that this implementation could only handle SPN types of algorithms, excluding any Feistel or GFN type ciphers. The second being that this implementation was not based on any standard architecture which could lead to compatibility issues. There are two main ways of implementing such instructions, the first being to add a co-processor which has a specific communication channel with the processor which is the one used in this Section. The second one called ASIP (Application-Specific Instruction set Processor) is to extend an existing ISA with specific instructions.

All these issues lead us to reassess how to solve the problem of the agile accelerated execution of LBCs as implementing a coprocessor would not fit our needs. In the next chapter we will take a new hardware/software hybrid approach with the use of a dedicated processor for the RISC-V (*Cf.* Sect. 2.8).

Chapter 4

Dedicated Processor Implementation of the RISC-V for LBC Agility

Including software in the implementation required finding a coherent ISA. The ISA that was chosen was that of the RISC-V (*Cf.* Sect. 2.8) which is open source with a strong base of development and extension possibilities. The new paradigm for our work is to implement an extension for the RISC-V, the Lightweight Block Cipher ISA (LBC-ISA). Once again, the key scheduling is not part of the extension and is assumed to be pre-calculated during the *algorithm loading step*. This extension is dedicated to the acceleration of the execution of LBCs. This acceleration is achieved by using additional instruction which aim to be as agile as possible, meaning that they should accelerate the execution of multiple ciphers. Some instructions may remain cipher-specific as the incompressible cost of making some instructions agile is too high (*Cf.* Section 3.1).

The next Section will present the classification we used along with restrictions on types and sizes of the ciphers' parameters.

4.1 Classification of Lightweight Block Ciphers

The basic idea behind adapting cryptographic primitives for use in connected cars is to maintain a low cost in terms of area and latency all the while being able to handle a diversity of different ciphers. This means finding an agile way to implement multiple algorithms than fits the requirements of car embedded technology. In order to determine how to classify those algorithms we must first understand what the already established classifications imply. The main way to identify the ciphers we chose for this study is as Lightweight Block Ciphers (LBC).

In order to ensure the agility, the existing similarities between ciphers must be exploited. There is a plethora of available ciphers and some similarities come from existing classifications. Hence, our study only takes into account some ciphers with specific criteria. The main criterion is that they belong to the Lightweight Cryptography, which corresponds to a more recent generation of ciphers which have been designed to solve certain flaws such as area or latency costs of pre-existing ciphers. Amongst the other criteria is that they need to be block ciphers, as opposed to stream ciphers for example.

They must also respect some implementation and security requirements such as the block and key size. This study only takes into account symmetric-key cryptographic algorithms. Therefore, the key is a secret and in order to ensure minimal levels of security, each cipher must use a key of at least 128 bits. Therefore, any cipher or version of a cipher with a smaller key is disregarded for lack of security according to the NIST recommendations [10].

The studied ciphers are only block cipher, which means they encrypt the data by having a defined portion of the data go through a small process, called a round, multiple times until the required security level is achieved. The number of rounds varies from one cipher to another, so can the block size. In order to have a homogeneous block size between each cipher, we fixed this block size to be 64 bits. This decision was based on wanting to fit the existing Control Area Network (CAN) standard [39]. Indeed, due to this standard, messages in cars are generally 64-bit. This also has an impact in lowering the overall area cost of implementation. The most common standard symmetric-key cryptographic algorithm used to secure digital information is the AES [64], which uses a 128-bit block. By only studying 64-bit block cipher, we therefore had to disregard AES. In order to lower the hardware cost, we chose to work with a 32-bit VexRiscv Core (*Cf.* Sect. 4.3). This choice also meant using 32-bit instructions, with two 32-bit inputs. We would therefore be able to manipulate 64-bit inputs with a single instruction but not a 128-bit input. Moreover, this becomes an even larger issue when dealing with both the *Confusion Step* and the *Diffusion Step*. Indeed, 64-bit block size ciphers tend to use 4×4 S-Boxes whereas 128-bit block size ciphers tend to use 8×8 S-Boxes. Having both types of S-Boxes would have a large area cost. The *Diffusion Step* also requires permuting bits all over, which takes much longer when all the bits cannot be used as the input of a single instruction. Therefore, the rest of our work only takes into account 64-bit block size ciphers, which is why NOEKEON had to be set aside.

Once these basic choices have been made, there are still a large amount of ciphers left with a large amount of specificities and identifying their common traits requires further classification. These ciphers are presented in Section 2.5.1 and Appendix A. This is why the first step in our study was to establish our own classifications of this group of ciphers. These classifications evolved as our study advanced and focuses on different aspects of the ciphers.

Analysis of the different types of operations shows there are two main frames of classification, according to the *Confusion Step* and according to the *Diffusion Step*. The objective of this classification is to find a minimal amount of instructions to add to a microprocessor in order to accelerate the execution of LBCs. This means each algorithm will be classified according to the additional instructions that fit its operations. We ended up dividing each algorithm into 4 different categories.

Category **I** applies to algorithms which only use basic instructions, both for the *Confusion Step* and *Diffusion Step*. These algorithms were usually developed for software use and have good performances without needing any extra instruction, the only exception is rotation which is not a basic instruction in every instruction

Table 4.1: Classification of Lightweight Block Ciphers

Category	Algorithm	Confusion	Diffusion	
			Type	Level
I	Simon	AND	Rotation	Bit
	Speck	ADD	Rotation	Bit
	Simeck	AND	Rotation	Bit
	RC5	ADD	Rotation	Bit
	XTea	ADD	Rotation	Bit
II	GOST	S-Box	Rotation	Bit
	Rectangle	S-Box	Rotation	Bit
III a	PRESENT	S-Box	Permutation	Bit
	GIFT	S-Box	Permutation	Bit
III b	PRINCE	S-Box x2	MatMult x3	Bit
III c	Piccolo	S-Box	MatMult	Bit
	LED	S-Box	MatMult	Bit
IV	TWINE	S-Box	MatMult	Nibble
	SKINNY	S-Box	MatMult	Nibble
	Midori	S-Box	MatMult	Nibble
	MANTIS	S-Box	MatMult x3	Nibble

set (including the RISC-V ISA). This is the only category which does not use an S-Box based *Confusion Step*.

Category **II** applies to algorithms which use a bit-level rotation as their *Diffusion Step*. They do not require additional instructions other than the S-Box.

Category **III a** applies to algorithms which use a bit-level permutation as their *Diffusion Step* and a 4×4 Sbox as their *Confusion Step*. These permutations are extremely cheap in terms of hardware but cannot be made agile at a reasonable cost. Therefore, algorithms from this category should be hand-picked in advance and each require a specific instruction.

Category **III b** applies to algorithms which use a bit-level matrix multiplication with a hollow matrix and a 4×4 Sbox as their *Confusion Step*. Just like category **III a** this type of algorithms requires a specific instruction for each cipher in this category. The only algorithm in this category is PRINCE [19], which is an α-reflective algorithm (*Cf.* Fig. 2.5.1) and therefore uses a set of 2 different S-Boxes and a set of 3 different matrix multiplication. The same would be true for MANTIS [14] in category **IV** expect that its S-Box is involutory. We chose to not take this aspect into account regarding their classification as doubling up on Confusion and *Diffusion Steps* can be achieved using the same hardware with different parameters.

Category **III c** applies to algorithms which use a bit-level matrix multiplication as the *Diffusion Step* and a 4×4 Sbox as their *Confusion Step*. Every bit of the matrix is relevant as it can take on nibble values different from 0 or from 1. This is what differentiates their *Diffusion Step* from that of category **IV**.

Finally, category **IV** applies to algorithms with a nibble-level diffusion which can be implemented as a matrix multiplication and a 4×4 Sbox as their *Confusion Step*. This category is the broadest because matrix multiplication can be made into an agile instruction which requires configuration but can apply to any nibble-level matrix.

According to this classification, algorithms can be divided in two parts according to their *Confusion Step*. On the one hand those using basic instructions from category **I** and on the other those from categories **II**, **III** and **IV** which are using 4×4 S-Boxes. Algorithms which didn't fit either of those criteria were disregarded for being too unique. This classification is essential in identifying how the execution of LBCs can be accelerated while focusing on agile instructions, which can be used for multiple algorithms. The instructions that seem the most adapted to such an approach are:

- The 4×4 Sbox
- The bit-level permutation
- The bit-level matrix multiplication
- The nibble-level matrix multiplication

These will therefore serve as instruction basis for the rest of this study.

4.2 RISC-V Rationale

The RISC-V ISA (*Cf.* Sect. 2.8) is a free and open specification maintained by the RISC-V Foundation. We chose to work with this particular ISA and there are multiple reasons why. First off, RISC-V is at the heart of plenty of research nowadays. Indeed, its architecture takes into account many of the advantages of a risc architecture with a very light ISA. Second, unlike the Intel or the ARM ISA, the RISC-V is open source meaning any, and everyone can work on it. This availability and popularity means there are lots of tools available and plenty of areas being researched. Third, the RISC-V is only an ISA, meaning that its implementation isn't fixed and can therefore be adapted to any specific need. Finally, one of the RISC-V's specificity is its innate extendability. Some space has been purposefully dedicated to future instructions. Thereby, the RISC-V is a perfect fit for our desire to accelerate the execution of carefully chosen additional instructions.

4.3 VexRiscv Core

In order to work with the RISC-V, We implemented a dedicated processor platform based on a 32-bit VexRiscv CPU Core [3]. The platform (a microcontroller grade SoC) uses in addition to the CPU, a specific interconnect, program and data memory blocks and some communication peripheral. Though we use an FPGA Board as a means to develop our project, the end result is to target an ASIC. The specification for this core is provided in Section 2.8.4.6.

Except for a few fixed elements, the majority of the core functionalities are configurable and plug-in based: Branch prediction, Register file, Hazard controller and many others, most of them optional. This allows to scale the core architecture depending on the targeted application. One of the main reasons this core was picked is due to its plug-in mechanism which allows adding and removing hardware extensions without tempering with the core itself. The reason we picked a 32-bit RISC-V even though we are working on 64-bit block size is in order to reduce the size of the implementation. This choice causes some growth in the complexity of the execution as the 32-bit RISC-V doesn't allow 64-bit outputs.

4.4 Dedicated Hardware Cryptographic Instructions

The RISC-V extension we propose is hardware-based, meaning that each software instruction is supported by a hardware implementation of that instruction. Each additional instruction therefore has an area cost and must be chosen and implemented with care in order to maintain low area overhead with good acceleration performances.

In this Section, we will detail the instructions that are part of our extension and how they were implemented.

4.4.1 Double 32-bit Instructions for 64-bit Functions

Similar to standard arithmetic and logic instructions, the new instructions will be executed in one clock cycle in the execution pipeline stage. From two 32-bit source registers, coming from the register file, the result is computed combinatorially and stored back into one 32-bit destination register of the same register file. Since we handle 64-bit blocks, we use two 32-bit source registers for the block. Also, we will need two consecutive, and slightly different, instructions to respectively compute, the high and the low 32-bit parts of the result. The first instruction does all the computation and returns the first 32-bits of the output. Since the hardware instruction is based on logic, the second instruction also computes the 64-bit result and returns the last 32-bits of the output. Figure 4.1 shows how both those instructions use the same 64-bit input and the same logic and configuration but require using to 32-bit output registers from two separate instructions. Also, some of our instructions (*Cf.* Sect. 4.4.3 and Sect. 4.4.6) require a configuration step. This step stores instruction specific parameters on dedicated RISC-V Control Status Registers (CSR). These registers can be accessed during the execution to match the instruction to the selected cipher.

In our actual workflow, these instructions are called separately by the user. So, each cryptographic plug-in will feature two instructions, to be used as such:

```
// Add round Key
l = l ^ rdKl[i];
h = h ^ rdKh[i];

// S-Box
t = l;
```

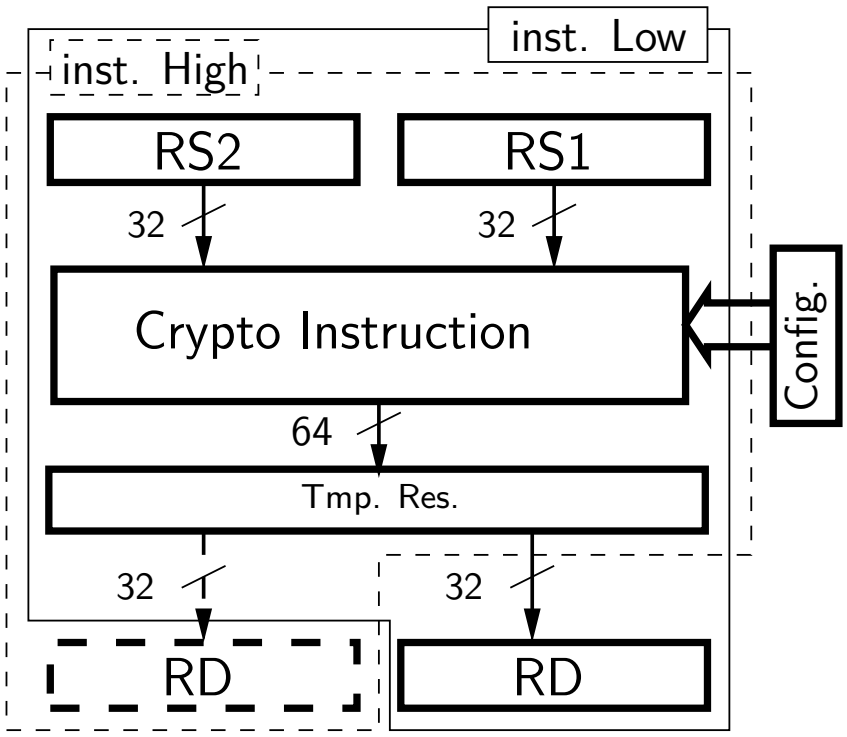


Figure 4.1: 64-bit Output Using Two 32-bit RISC-V Instructions

```
 asm volatile ("SBOX_C␣␣%0,%1,%2":"=r"(l):"r"(t),"r"(h));
 asm volatile ("SBOX_CH␣%0,%1,%2":"=r"(h):"r"(t),"r"(h));
 // P-Layer
 t = l;
 asm volatile ("PRESENT_D␣␣%0,%1,%2":"=r"(l):"r"(t),"r"(h));
 asm volatile ("PRESENT_DH␣%0,%1,%2":"=r"(h):"r"(t),"r"(h));
% ))))%% TODO: remove these before submission
```

For the rest of this study, when talking about an instruction it will actually be referring to two instructions, both compute the same result except that the first stores the lowest significant bits and the second stores the highest significant bits in separate registers.

4.4.2 Proposed ISA Extension

Once the RISC-V was chosen as an ISA, the VexRiscv Core chosen as an implementation and the test platform setup, the instructions used for the extension had to be chosen. As explained in Section 4.1 those instructions where deducted from Table 4.1 The concerned algorithms from Category **II**, **III** and **IV** require 7 additional hardware instructions which are part of our extension for the RISC-V ISA, the **LBC-ISA**:

- The S-Box instruction: **SBOX_C**
- The PRESENT bit-level permutation instruction: **PRESENT_D**
- The GIFT bit-level permutation instruction: **GIFT_D**

- The PRINCE bit-level matrix multiplication family of **PRINCE_D** composed of three instructions:
  - **PRINCE_DF**
  - **PRINCE_DM**
  - **PRINCE_DL**
- The nibble-level matrix multiplication instruction: **NMAT_D**

The **SBOX_C** instruction is used by each of the ciphers as their *Confusion Step*. The **_D** instructions are used for the *Diffusion Step*. The **PRESENT_D** and **GIFT_D** instructions are cipher-specific bit-level permutations. **PRINCE_D** corresponds to the three PRINCE specific bit-level matrix multiplications. **NMAT_D** is an agile nibble-level matrix multiplication.

The algorithms from Category **I** are those who only use basic instructions meaning their execution cannot be accelerated through the use of additional instructions. The studied algorithms are therefore those which use one of these instructions. Our implementation is based on an *algorithm loading step* which requires storing the parameters for the Confusion and *Diffusion Step* in memory. On the one hand, the amount of parameters required for an agile 64×64 bit-level permutation is 4096 bits (the amount of bits in a 64×64 matrix), which we considered too high for a lightweight implementation. Even with an optimisation based on some properties specific to the permutation of these algorithms, the amount of parameters remained high (*Cf.* Sect. 3.7) and also required important resources in terms of hardware. Implementing an agile instruction for algorithms using bit-level permutation was therefore not an option. Hence, we had to implement a low-overhead specific instruction for each of the algorithms with a bit-level *Diffusion Step*. On the other hand, unlike the bit-level permutation, an agile implementation at nibble-level is conceivable without the overhead being too high for two reasons:

- The matrix multiplication is done at nibble level, reducing the total amount of incompressible parameters.
- The matrices used are only composed of 1s and 0s meaning that all multiplications can implemented as a series of ANDs and XORs.

The latter aspect lead to set aside the algorithms from category **III c**, namely Piccolo and LED. Indeed, by reducing the useful information at bit level, we consider nibbles with value 0 or 1, the size of the matrix is divided 16-fold since these parameters constitute a 16×16 matrix. Therefore, the amount of parameters required for an agile nibble-level diffusion is only 256 bits. By reducing the amount of memory needed 16-fold, we were able to use an agile version for nibble-level *Diffusion Steps*, while keeping the implementation cost reasonable.

Table 4.2 shows which algorithm uses which instruction. The detailed implementation of each of these instructions will be given in the following Sections.

Table 4.2: Instructions Used by Each Algorithm

Cipher	Confusion	Diffusion
GOST	SBOX_C	None
Rectangle	SBOX_C	None
PRESENT	SBOX_C	PRESENT_D
GIFT	SBOX_C	GIFT_D
PRINCE	SBOX_C (x2)	PRINCE_D (x3)
Midori	SBOX_C	NMAT_D
TWINE	SBOX_C	NMAT_D
SKINNY	SBOX_C	NMAT_D
MANTIS	SBOX_C	NMAT_D (x3)

4.4.3 SBOX_C Instruction

In order to make the S-Box agile, we had to implement an instruction whose parameters can be modified to correspond to any algorithm. As S-Boxes have the same function as LUTs, we chose these as the best way to implement this function. Although a configuration is required, for most ciphers, the same S-Boxes are used throughout the entire cipher which means this step can be done beforehand. Indeed, any parameters are set during the *algorithm loading step* in order to avoid any delay during the actual encryption.

The S-Box configuration is done before the computation. Nonetheless, for some few ciphers, namely α-reflective algorithms [20] such as PRINCE, the *Confusion Step* is slightly different. Indeed, PRINCE uses two different S-Boxes during its execution (*Cf.* Sect. 2.5.1), which requires a change of S-Box. This can be done though mid-execution re-configuration of the S-Box or by adding a second S-Box instruction with different parameters from the start. In addition to doubling the FF cost, both those solutions have a cost in terms of latency for the former and area for the latter. The solution chosen in this research was to use mid-execution re-configuration.

As for implementation, we initially used straightforward RTL implementation for our S-Boxes, which was not optimal as it requires a large amount of additional FFs. Since the S-Box is equivalent to a 4×4 LUT and we worked with a Xilinx FPGA, we were able to use the **CFGLUT5** primitive. This primitive not only lowered the additional Flip-Flops (FF) requirements to almost 0, it also lowered the total amount of LUTs used. Moreover, each CFGLUT5 is configured in parallel during the *algorithm loading step* which makes their latency overhead negligible as re-configuration only happens when the key or the mask is changed, which happens once every thousand or more encryption for most algorithms. Finally, with a naive implementation, the power activity is generated by the propagation of information through 4 stages of LUTs along with the routing whereas for CFGLUT5 it is generated by a single LUT. This is especially relevant in the resilience against side-channel attacks (*Cf.* Chapt. 5). A more basic description of the CFGLUT5 is

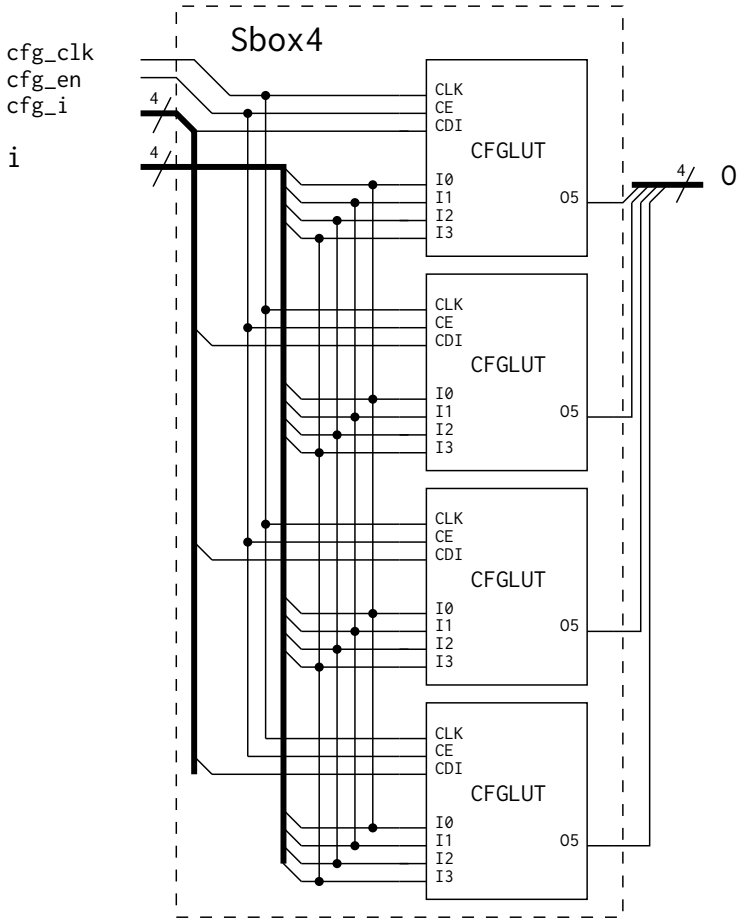


Figure 4.2: Sbox_4 Architecture using CFGLUT5 Primitives

given in Appendix B In this section, we explain how and why we implemented the **SBOX_C** using the Xilinx **CFGLUT5** primitive.

4.4.3.1 Building 4×4 Sbox using the CFGLUT5 Primitive

Figure 4.2 shows how we assembled 4 **CFGLUT5** primitives to build a Sbox_4 which implements a 4×4 Sbox.

Four **CFGLUT5** primitives are grouped sharing the configuration clock and activation inputs. The configuration inputs are separate to allow parallel configuration of the 4 primitives. The four functional inputs are shared and the outputs of the primitives are grouped to form a 4-bit output.

Figure 4.3 shows a configuration sequence and the operation of the Sbox_4 block.

In this example, the Sbox_4 is configured with the Present S-Box (`0xC56B90AD3EF84712`). Each **CFGLUT5** primitive is configured with on bit of each nibble.

During operation, the input `i` of the Sbox_4 will come from the output of a register. The output of the S-box will be ready before the next clock cycle.

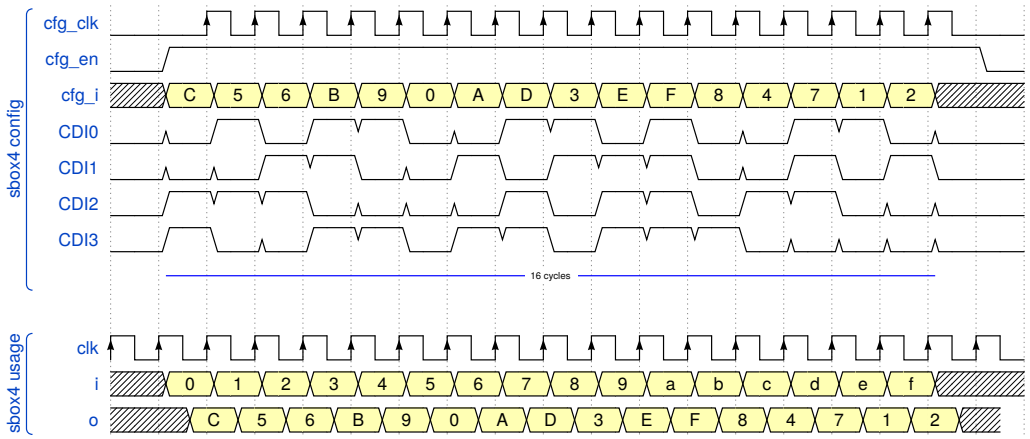


Figure 4.3: Configuration Operation of the Sbox_4 with CFGLUT5

4.4.3.2 SBOX_C Architecture

The **SBOX_C** is constructed by assembling 2 Sbox_8_4, which itself is 8 4×4 Sbox assembled together, as shown in Fig. 4.4. The Sbox_8_4 is constructed by assembling 8 4×4 Sbox. The configuration is still done in parallel, in 16 cycles, with a 32-bit input word.

The Sbox_8_4 have been described hierarchically in SystemVerilog. The integration, as a VexRiscv plugin is still possible, as SpinalHDL allows the instantiation of existing RTL modules (Verilog or VHDL) using a blackbox component.

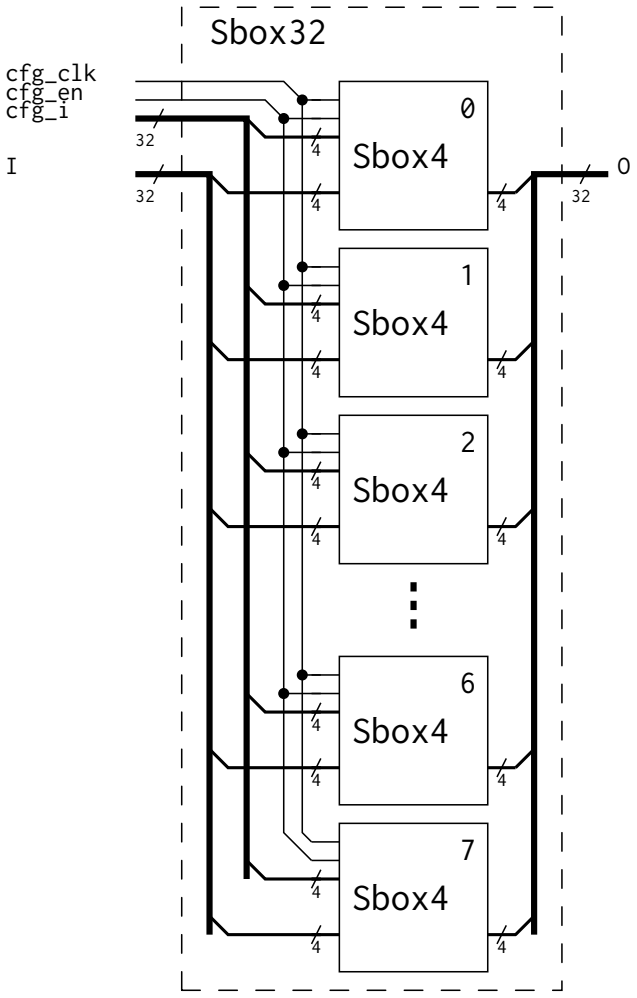


Figure 4.4: Sbox_8_4 Architecture using CFGLUT5 Primitives

4.4.3.3 CFGLUT5 against RTL Resources Usage Comparison

There are multiple reasons why we chose to use CFGLUT5 over a straightforward RTL.

First, the synthesis results presented in Tab. 4.3 show resources for the addition of the SBOX_C instruction compared to the base version of the VexRiscv for both CFGLUT5 and RTL implementations. They show that the **CFGLUT5 SBOX_C** is much more compact both in terms of fpga LUTs and FFs. Indeed, the naive RTL implementation uses 15 multiplexers (MUX) to calculate each of the 4 output bits of each of the 16 4×4 S-Box. Each of these MUX requires a configuration bit, this represents a minimum of $15 \times 16 \times 16 = 960bits$. This amount can be seen in the 1057 additional FFs. Moreover, the **CFGLUT5** primitive uses only a single fpga LUT for a total of 64 fpga LUTs for **SBOX_C**. In practice, some additional LUTs are added, but they still represent a smaller amount of fpga LUTs than the Naive RTL **SBOX_C** which is twice as big.

The second reason is linked to the power activity of each implementation. On

Table 4.3: Cost Comparison Between Basic RISC-V ISA, Naive RTL and CFGLUT5 Implementations

Additional Implementation	LUT	FF
None (Basic RISC-V ISA)	973	765
Naive RTL SBOX_C	1379	1822
CFGLUT5 SBOX_C	1171	766

the one hand, Compared to the **CFGLUT5 SBOX_C**, the naive RTL **SBOX_C** uses a larger amount of fpga LUTs which are likely to be spread over a larger area. This means that the datapath of this implementation is likely to manipulate the same input over multiple fpga LUT thereby increasing the activity during their use. On the other hand, the activity of the **CGFLUT5** primitive is limited to its single fpga LUT. Therefore the amount of leaked information is much lower than with the naive RTL implementation. Lowering the amount of leakage is essential when developing hardware with SCA in mind. Figure 4.5 illustrates this phenomenon by comparing the CPA graphs of an execution of the PRESENT cipher using both implementations.

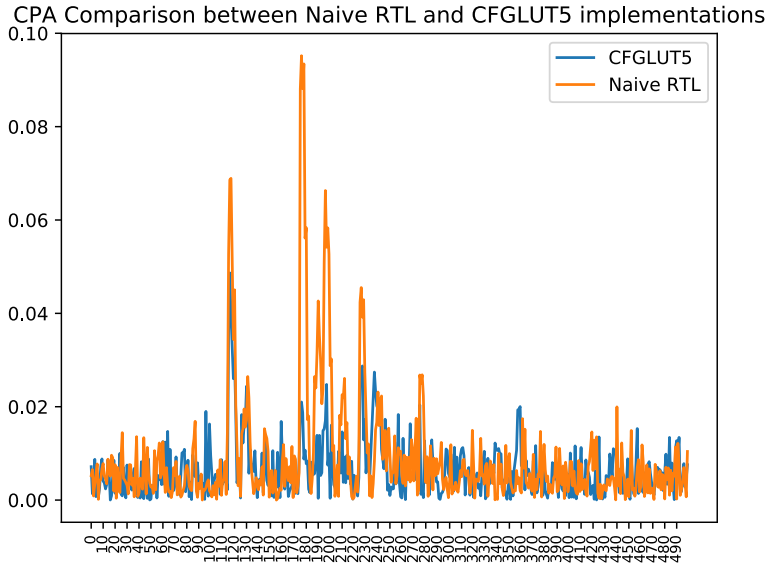


Figure 4.5: Comparison of Leakage between Naive RTL and CFGLUT5 Implementations

4.4.4 PRESENT_D and GIFT_D Instructions

Both these algorithms use a similar *Diffusion Step*. They use a bit-level permutation as their *Diffusion Step* (*Cf.* Appendix A). This permutation correspond to a reorganisation of wires in terms of hardware. Though it is possible to make an agile bit-level diffusion layer, specific functions are extremely cheap in terms of hardware

area, whereas agile instructions can have high implementation cost as shown in Section 3.7. Indeed, both PRESENT and GIFT use a bit-level permutation which requires no logic gates and therefore has a near zero cost in terms of area.

4.4.5 PRINCE_D Instruction

The *Diffusion Step* of PRINCE corresponds to a bit-level matrix multiplication with a very sparse unitary matrix. Once again, this type of diffusion has a very high parameter cost in order to be made agile. The hardware cost of such an instruction is 4096 ANDs to mask the input for each output bit plus 192 XORs, three XORs to generate each of the 64 bits of the output for each of the bit-level matrix multiplication and an additional 4096 ANDs for the bit-level permutation. Therefore, the option of using cipher-specific instructions was chosen. Since PRINCE is an α-reflective algorithm, it also requires using the inverse matrix for the second half of the algorithm and a third matrix, which corresponds to the MixColum part of the *Diffusion Step* (*Cf.* Appendix A). This *Diffusion Step* is therefore divided into three instructions **PRINCE_DF** for the **D**iffusion step of the **F**irst rounds, **PRINCE_DM** for the **D**iffusion step of the **M**iddle round and **PRINCE_DL** for the **D**iffusion step of the **L**ast rounds of the cipher.

4.4.6 NMAT_D Instruction

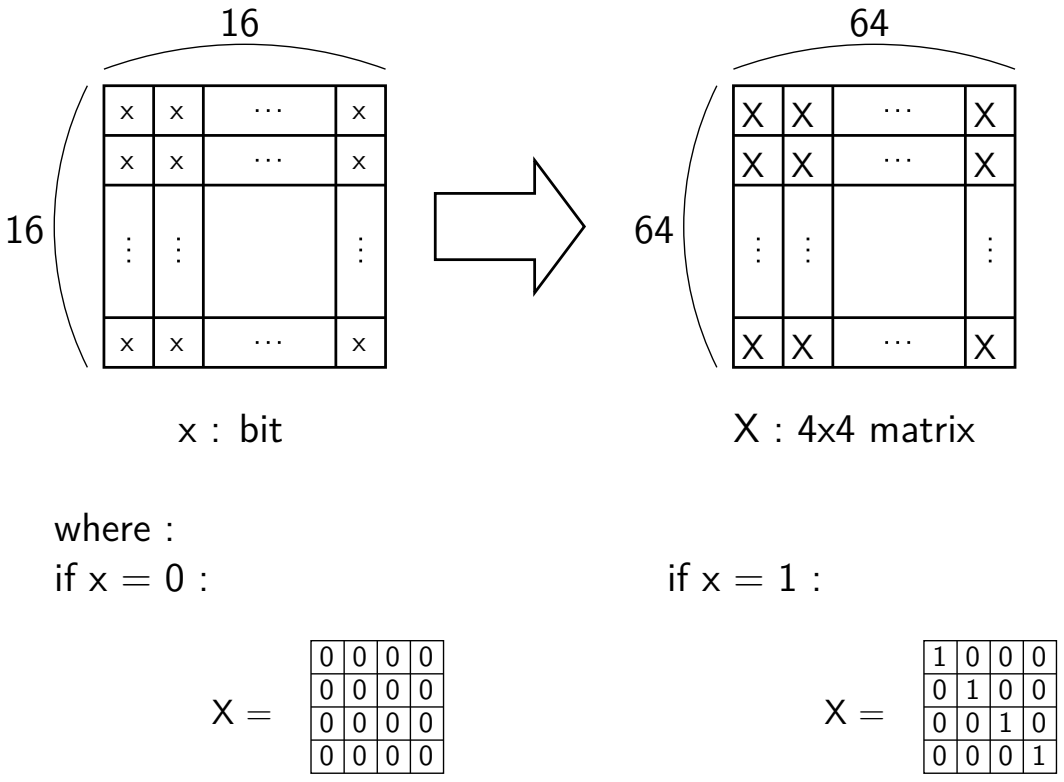


Figure 4.6: 256-bit of Configuration for the 64×64 Matrix of NMAT_D

Algorithms using **NMAT_D** instruction have a diffusion layer similar to that of AES in that they use a *Shift Row* and/or a *MixColumn* function. Though they are presented as two separate instructions, they can be merged into one. Both these instructions can be seen as a single matrix multiplication U×V where V is the 64-bit state seen as a vector and U is a 64×64 normalized matrix specific to the *Shift Row* and *MixColumn* combination. Once again, this U matrix requires parameterisation which can be handled before the cipher's execution during the *algorithm loading step*. Having to handle the parameters for a 64×64 matrix would require 4 kbit worth of memory, which is incoherent with our lightweight objectives.

With this in mind, and based on the fact that the diffusion layer of these algorithms only handles nibbles and not bits, it is possible to parametrize this 64×64 matrix by using a 16×16 matrix, which requires only 256 bits. In order to go from this 16×16 parameter matrix to a 64×64 functional matrix, the transformation used is to turn any 1 from the 16×16 matrix to a 4×4 Identity matrix in the 64×64 matrix, and to turn any 0 from the 16×16 matrix a to a 4×4 null matrix in the 64×64 matrix as shown in Fig. 4.6. This optimisation logic is the same as the one used in Section 3.6.1

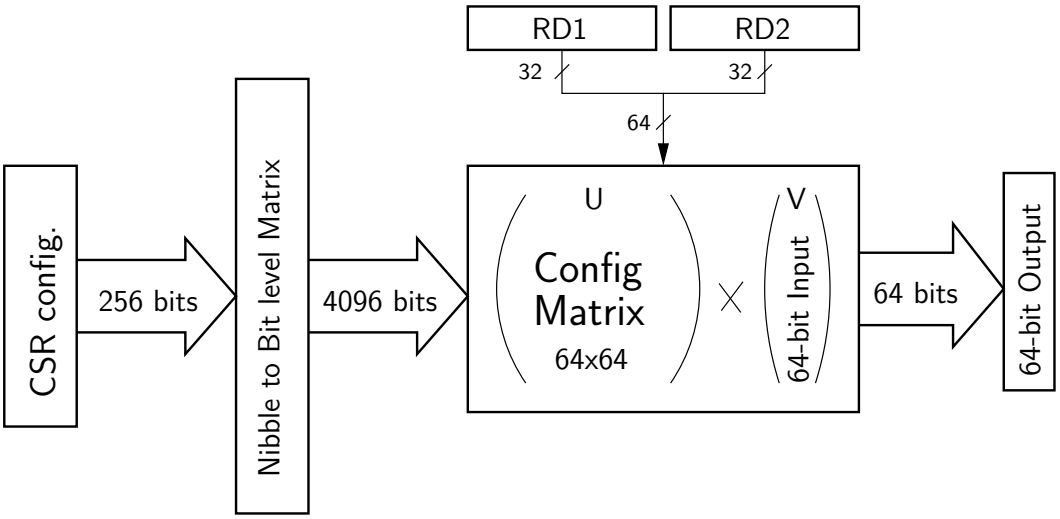


Figure 4.7: The NMAT_D Instruction

The **NMAT_D** instruction requires multiple steps which are shown in Figure 4.7 First, the parameters are stored in eight 32-bit CSR registers. Each of those registers contains the parameters for two lines of the 16×16 matrix, or eight lines of the 64×64 matrix. Once the parameters have been stored in CSR registers, each input nibble is ANDed with those parameters before getting XORed together to generate the 64-bit output.

Figure 4.8 represents the implementation of the **NMAT_D** multiplication. The parameters contained in the CSR Register are used to select the nibbles used for the multiplication. The remaining nibbles are then XORed together in a 4 layer XOR tree in order to generate the resulting nibble. This operation is used to

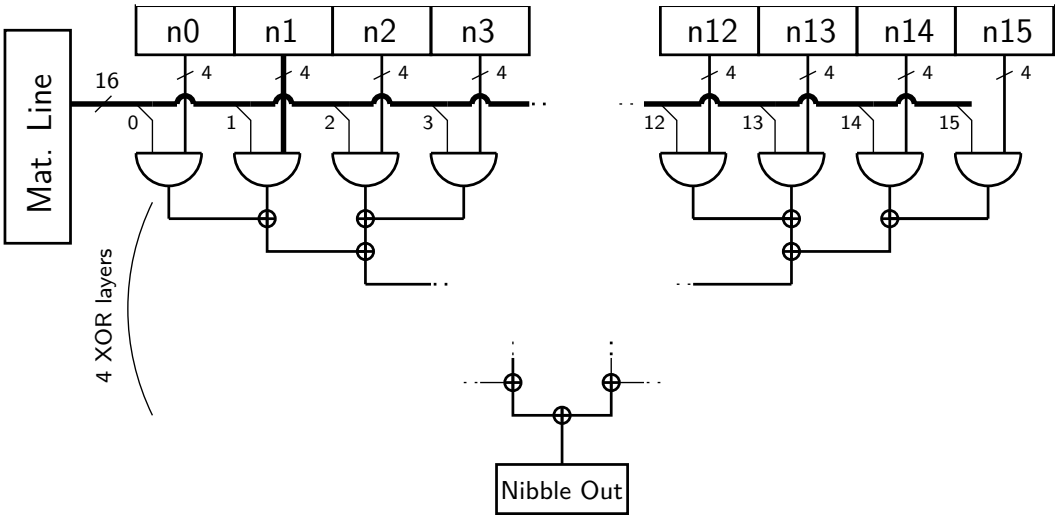


Figure 4.8: The NMAT_D multiplication datapath

generate each of the 16 output nibbles.

4.5 Theoretical Results

Throughout this study, different methods were used to measure the costs and gains of this implementation. The numbers published in [85] where based on theoretical implementations and single round observations. Table 4.4 shows the number of instructions in a single round for each of the studied algorithms with the Base ISA compared to the number of instructions required once additional instructions have been added. The *accelerated* results are based on simple hand written assembly code and therefore do not take into account many real-life factors. Indeed, it doesn't even take into account any delays caused by branching which is an important cost especially when the number of rounds is high. It does not take into account the doubling of instructions due to the use of a 32-bit core either. Each column corresponds to the instructions required to accelerate the algorithms from the corresponding category (*Cf.* Sect. 4.1).

This table also include algorithms from Category **I** as the RISC-V does not include a rotation instruction and during this theoretical approach we first considered rotation as one of the additional instruction that could accelerate the execution of LBCs. Nonetheless, as the results show, the gain factor of less than 1.5 is extremely low and only affected algorithms which already had low software latency. After these observations, we chose to not have rotation be part of our extended LBC-ISA. Moreover, the rotation instruction is proposed in the Standard Extension for Bit Manipulation (*Cf.* Sect. 2.8.2), meaning that this instruction is already part of a soon-to-exist extension.

The results from Tab 4.4 nonetheless showed a high gain factor of over 26 and up to 128 for any algorithm using at least two additional instructions which was a promising theoretical result and led us to continue exploring this option. As a comparison, the results of Marshall *et al.* in [49] showed a 4x to 10x acceleration when working on accelerating the execution of AES using a similar approach.

Table 4.4: Comparison of the Number of Executed Instructions in One Round between the Non-Accelerated Ciphers and the Theoretically Accelerated Cipher

Algorithm	Instructions per round					Gain
	None	**I**	**II**	**III**	**IV**	factor
Simon	15	11	11	11	11	**1.36**
Speck	12	10	10	10	10	**1.20**
Simeck	12	10	10	10	10	**1.20**
RC5	15	11	11	11	11	**1.36**
XTea	24	16	16	16	16	**1.50**
GOST	221	213	10	10	10	**22.1**
Rectangle	236	222	19	19	19	**12.4**
PRESENT	555	555	352	5	5	**111.0**
GIFT	641	641	438	5	5	**128.2**
PRINCE	820	820	617	9	9	**91.1**
TWINE	174	174	126	126	6	**29.0**
Midori	350	350	147	147	9	**38.9**
SKINNY	234	234	31	31	9	**26.0**
MANTIS	348	348	145	145	5	**69.6**

4.6 Experimental Test and Validation

Once these instructions were implemented, the next step was to test and validate them. This means making sure each instruction can indeed be used to execute each algorithm and that any given input returns the right output.

Taking advantage of the VexRiscv plug-in modular architecture, each specific hardware instruction is added as a separate plug-in to the CPU core (*Cf.* Sect. 2.8.4.6). Thus, depending on the targeted cryptographic algorithms, we can configure the processor by choosing which plug-in to enable. This allows the evaluation of hardware resource usage of each additional instruction independently.

We then needed a platform that let us implement the VexRiscv Core of the RISC-V. This platform was designed *in-house* for core tests and validations. It is used to execute each algorithm with random plaintext and key inputs and compare the results to that of already verified implementations.

In this Section we explore how we were able to use plug-in mechanism of the VexRiscv to add the LBC-ISA extension on a test platform which uses the VexRiscv Core.

4.6.1 Software Workflow of the Plug-in Addition

The VexRiscv Core plug-in mechanism allows the implementation of additional hardware-based instruction. The code for the ciphers was written in C but our instructions are not part of the GCC standard RISC-V toolchain. We therefore had to find a way to use our additional instructions without deeply modifying the toolchain. Modifying the toolchain would have been possible, but only make sens for a stable addition to the ISA.

In this section we will explain the compilation flow we have implemented for our tests. This flow is summarized in Fig. 4.9.

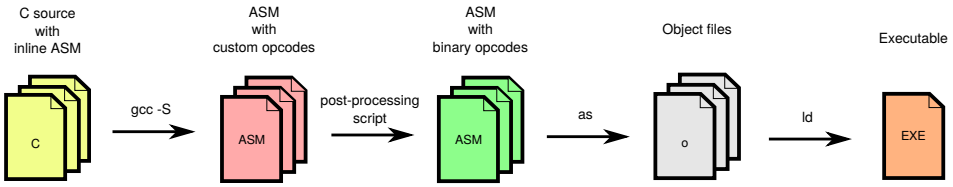


Figure 4.9: Compilation Flow of a Plug-in Added Instruction

We will use the **PRESENT_D** instruction as an example to detail each step.

First off, we used inline assembly in our C code to explicitly call our instruction. The extended inline assembly supported by GNU GCC, allows reading and writing C variables from assembly as shown in the following code example:

```
----------------Present_cipher.c---------------
// C code with inline ASM

...
// s1, s2 --> inputs
// d --> output
uint32_t s1, s2;
uint32_t d;
// inline asm passing variables from the C code
asm volatile ("PRESENT_D␣%[rd],%[rs1],%[rs2]":\
             [rd]"=r" (d) : [rs1]"r" (s1) , [rs2]"r" (s2))
...
-----------------------------------------------
```

This code is compiled into an assembly file using the `-S` GCC flag to generate an intermediate assembly file before calling the assembler.

```
riscv-gcc -c -S Present_cipher.c -o Present_cipher.t.s
```

GCC will compile the C code, allocate registers for the C variables (optimisations are even possible), whereas the inline assembly code will be included, as is, without being modified.

The corresponding line in the intermediate assembly file will be similar to the following.

```
----------------Present_cipher.t.s-------------
/*
   ASM with custom instructions mnemonics
   Register allocation has been done by the C compiler (GCC)
   In this example:
    - a3,a5 <- opperends
    - a6    <- result
*/
...
PRESENT_D a6,s3,a5
...
-----------------------------------------------
```

As this instruction is not known by the assembler (GNU AS), we have to replace **PRESENT_D** by its binary value. We have developed a simple script that parses the intermediate assembly file and replaces our custom instructions by their values.

```
asm-fixup Present_cipher.t.s -o Present_cipher.s
```

For our cryptographic instructions we have chosen free opcodes from the RISC-V ISA reserved for custom extensions. Their format is known and the script can compute the correct 32 bit instruction value for the selected registers. Then, replace the instruction using the .word directive, which allows in GNU AS, to insert an arbitrary 32 bit value.

```
--------------------------A.s--------------------
/*
   A python script (asm-fixup), replaces the custom instructions by the
   corresponding binary value using GNU AS .word directive
*/
...
#  PRESENT_D a6,s3,a5
# rd  : 16 (10000)
# rs1 : 19 (10011)
# rs2 : 15 (01111)
.word 0b10000000111111001100010000011 0011
...
-----------------------------------------------
```

A final call to GNU AS generates object files which are linked using GNU LD to generate the final executable.

```
riscv-as -c Present_cipher.s -o Present_cipher.o
```

4.6.2 Test Platform

The test platform is depicted in Figure 4.10. It includes the following elements:

- One VexRiscv Core : configured as follows:
  - implements the RV32I ISA,
  - enabled Configuration and Status Registers CSR,
  - Light Shifter,
  - Simple instruction and Data Bus Interface.
- Two independent 32 KB On-chip memories for program and data.
- A communication module to connect with a host computer through a USB interface chip on a test board.
- A configuration registers module which includes dedicated memory addressable registers that can be set and read from the host PC through the USB module.
- A General Purpose Input Output (GPIO) module for direct user interaction.

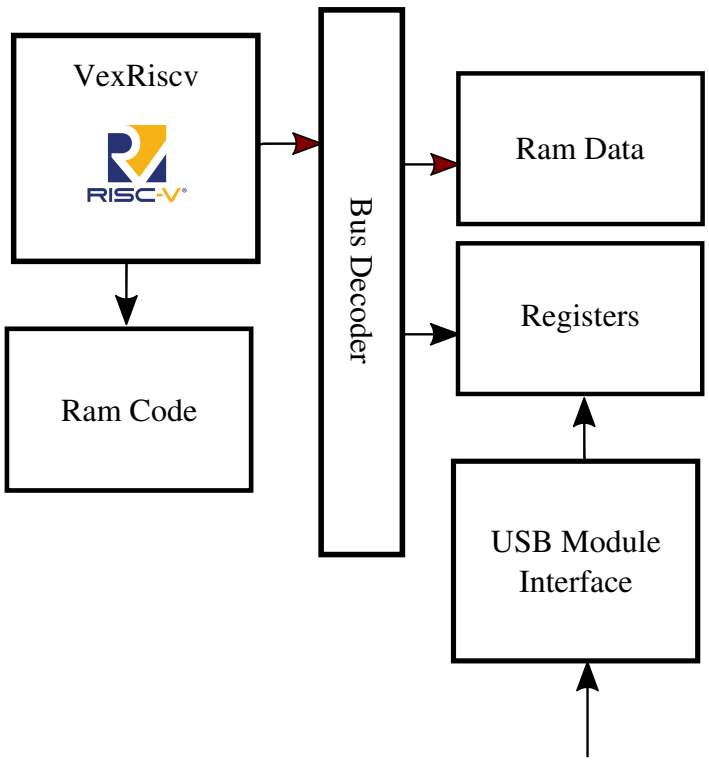


Figure 4.10: VexRiscv Platform

4.6.3 Target FPGA Board

We use The Chipwhisperer CW305 target board [2]. It is build around an Artix-7 FPGA and brings features like a simplified USB interface, for host to NICV communication, and GPIOs (LEDS and buttons...). The board is designed to be used as a target for side channel power analysis. It includes several probing points and means to measure the voltage and current intensity of the different elements of the board.

The target FPGA is an Artix-7 xc7a100t device. It includes 101440 Logic Cells, 126800 Registersa and 4860 Kb of RAM. Its working frequency can vary between 5 and 160MHz.

The embedded USB interface can be used to configure the FPGA. At runtime, the same USB interface can be used to communicate with the device. It is viewed as a simple data/address bus interface, which can be used to read or write into the configuration registers of the platform.

4.6.4 Test and validation

Each block cipher algorithms is implemented in C and compiled with the standard GCC RISC-V toolchain. Two versions of each program have been developed allowing the following execution modes:

- *software*: using the standard RISC-V ISA,

- *accelerated*: using the proposed hardware instructions of the LBC-ISA.

To synthesize the platform hardware design, we have used the Vivado 2019.1.1 Design suite. The RISC-V software binary programs were converted to onchip memory initialization files and are pre-loaded into the program onchip memory when the bitstream is generated. To test and validate our design we have implemented several bitstreams with different plug-ins enabled and the corresponding software.

To configure the FPGA and communicate with the design, we have used Chipwhisperer software libraries. The Chipwhisperer software libraries are composed of a set of open-source Python modules that runs on the host computer.

The developed test program accomplish the following tasks:

- Loads a configuration bitstream file and program the FPGA.
- Uploads a key and a plaintext block using the platform configuration registers.
- Sends the start signal to the processor.
- Waits for test completion by polling a dedicated register acknowledged by the program that runs on the processor.
- Reads back the ciphertext block and the performance figures (number of executed instructions and the program duration in clock cycles) written by the processor program.

4.7 Implementation Analysis

4.7.1 Area Evaluations

Table 4.5 shows the hardware resources for the processor core and for each additional instruction. As we can see, the configurable instructions **NMAT_D** needs significant additional resources. Some of it is due to the configurable nature of this instruction which requires flip-flops (FF) to store its parameters. A similar trend was observed with a naive implementation of the S-Boxes as LUTs, this lead us to find a more optimised way to implement them with **CFGLUT5** (*Cf.* Sect. 4.4.3). In our implementation, this configuration memory uses standard flip-flop registers as it requires parallel read access. On the other hand, the specific fixed instructions **GIFT_D** and **PRESENT_D**, have very small overhead as they are implemented as simple combinatorial logic blocks in the Arithmetic and Logic Unit (ALU). This is also true for the **PRINCE_D** family of instruction which regroups the three instructions **PRINCE_DF**, **PRINCE_DM** and **PRINCE_DL**.

Table 4.6 shows the hardware resource usage overhead for the implementation of accelerated Lightweight Block Cipher algorithms. The acceleration of each algorithm requires the use of several LBC hardware instructions. The **SBOX_C** instruction is needed by all the algorithms and its cost can vary due to the logic LUTs, which accounts for routing, and are different for every algorithm. Nonetheless, the total LUT cost is around 200, and adding the hardware for a second set of

Table 4.5: Resource Usage Overhead for the Additional Instructions

Instructions	LUTs	%	FFs	%
None	973	-	765	-
SBOX_C	1171	+20	766	+0.1
SBOX_C x2	1371	+41	767	+0.3
PRESENT_D	1135	+17	766	+0.1
GIFT_D	1131	+16	766	+0.1
PRINCE_D	1113	+14	768	+0.4
NMAT_D	1545	+59	1022	+34
ALL	2113	+117	1028	+34

S-Boxes is around 400. As shown in Tab. 4.2 Midori, TWINE and SKINNY each use the **NMAT_D** configurable agile instruction as their *Diffusion Step*, which significantly increases the hardware resource overhead. There is a clear trade-off between the implementation agility and the hardware cost.

In the case of PRINCE, the **SBOX_C** instruction requires a mid-execution reconfiguration since PRINCE uses 2 different S-Boxes throughout its execution. There are therefore two option on how to implement PRINCE, each with different costs. The first is to use a single **SBOX_C** hardware and have the reconfiguration during the execution, which has an important latency cost. The second is to use two **SBOX_C** hardware and configure both of them with different values, which has an important area cost. The costs of both option are explored in Tab. 4.6 and Tab. 4.7.

In the case of MANTIS, the **NMAT_D** instruction can also be used except it requires a set of 3 matrices due to its α-reflective nature. This large amount of reconfiguration led us to set MANTIS aside in favor of the other ciphers.

Table 4.6: Resource Usage Overhead for Lightweight Block Ciphers

Cipher	LBC-ISA Instructions	LUTs	%	FFs	%
None		973	-	765	-
PRESENT	SBOX_C+PRESENT_D	1173	+21	767	+0.3
GIFT	SBOX_C+GIFT_D	1103	+13	767	+0.3
PRINCE	SBOX_C+PRINCE_D	1438	+48	769	+0.5
PRINCE	SBOX_Cx2+PRINCE_D	1638	+68	770	+0.7
Midori	SBOX_C+NMAT_D	1778	+83	1023	+34
TWINE	SBOX_C+NMAT_D	1778	+83	1023	+34
SKINNY	SBOX_C+NMAT_D	1778	+83	1023	+34

4.7.2 Latency Evaluations

Section 4.7.1 showed the cost of extending the RISC-V ISA with hardware-based instructions. Yet, these addition instructions aim to accelerate the execution of LBC. In this section, we present the results of the latency evaluation which com-

pares the instruction count of the execution of each cipher with the basic RISC-V ISA and with the LBC-ISA. The execution of each of these ciphers was done at 100MHz and no matter the configuration, the critical path was not affected by the addition of each module. To note, no evaluation of the maximum attainable frequency was done but the observed slack suggests that the frequency could be increased.

4.7.2.1 Implementation Results

Table 4.7 shows program execution instructions count for different algorithms for both Base ISA and LBC-ISA modes. The number of executed instructions is captured by software by reading the corresponding CSR of the VexRiscv Core. The machine instructions-retired counter `minstret` is a standard CSR defined in the RISC-V ISA.

Table 4.7: Instruction Count for Ciphers According to the ISA Used

Cipher	Base ISA	LBC-ISA	Gain factor
PRESENT	12544	359	35
GIFT	10661	320	33
PRINCE (SBOX_C x1)	17357	2313	8
PRINCE (SBOX_C x2)	17357	138	126
Midori	18944	190	81
TWINE	41279	621	66
SKINNY	40887	409	100

The gain factor is between 8 and 126 for the 6 considered algorithms, when using the specific LBC-ISA instructions. As a comparison, the results of Marshall *et al.* in [49] showed a 4x to 10x acceleration when working on accelerating the execution of AES using a similar approach. This important gain is due to the fact that bit manipulation in hardware is a single cycle instruction with reasonable complexity. Whereas in software, bit manipulation is extremely costly for instance, in the implementation of PRESENT used for these results, each bit has to be singled out to allow the permutation between single bits. The Basic RISC-V ISA implementations were made naively and little optimisation was brought to any of the algorithms, meaning that a more refined implementation could lead to different results. The algorithm which both least and most profits from this acceleration is PRINCE. On the one hand, the single **SBOX_C** implementation requires mid-execution reconfiguring, which takes a long time and explains the low results compared with other algorithms. On the other hand, the double **SBOX_C** implementation requires an additional hardware instruction, which has a larger area cost. The other results are also very positive as they show a gain factor of at least 33 which is far more than the area overhead which less than doubles the area cost.

4.7.2.2 Comparison with Theoretical Results

Table 4.8 sums up the theoretical results from Section 4.5 for the algorithms which gained most with the extended LBC-ISA where the results are given for the entire encryption. Once again, these results are theoretical and are the multiplication of the *per round* results of Table 4.4 by the number of rounds. This approach is obviously flawed as some rounds are different and this does not take into account any of the useful instructions which are not directly parts of a round such as jumps. The purpose of this table is simply to change the form of the results in order to compare them to the practical results of Table 4.7 and better understand the existing discrepancies.

Table 4.8: Theoretical Instruction Count with Hand-Written Assembly Code

Block Cipher	No Acceleration	With Acceleration (Theoretical)	Gain factor
PRESENT	17760	160	111
GIFT	18060	140	129
PRINCE	9020	99	91
Midori	5600	144	39
TWINE	6264	216	29
SKINNY	7488	288	26

The first discrepancy is the estimation in terms of number of instructions for the non-accelerated version of the algorithms which were not the same. For the PRESENT and GIFT algorithms, some aspects of the software implementation were optimised which lead to a lower instruction count for the non-accelerated execution. For PRINCE with one **SBOX_C**, the difference with theoretical results is large and is mostly due to the necessity to reconfigure the S-Boxes during the execution. Whereas for PRINCE with two **SBOX_C**, the results are quite close. For the algorithms using the **NMAT_D** instruction, the number of executed instructions for the Base ISA version is higher due to the software implementation which was made to mimic the accelerated version which lead to some additional latency. This approach was taken to have comparable codes for both LBC-ISA and Base ISA versions.

Second, the estimations were based on the implementation of a single round. Therefore, the number of instructions for the entire cipher is simply a multiplication of a single round by the number of rounds, which does not take into account the additional instructions necessary to handle loops, such as the jump and the incrementation of the counter.

Third, the assumption that the Round Key values would be accessible from dedicated registers was made. These registers have not yet been implemented in our version which therefore requires two instructions to fetch the 64-bit value of the Round Key from the memory. Adding these specific Round Key registers will allow us to accelerate our execution.

Finally, the theoretical results did not take into account the fact that the core would be 32-bit. This means that in practice, each LBC-ISA instruction is doubled (*Cf.* Sect. 4.4.1). Therefore, each round uses at least 3 more instructions than what the theoretical results suggest.

The two evaluations were made under different assumptions which makes the comparison of our results uneven. Nonetheless, we can see that the overall number of instructions has less than doubled from an incomplete theoretical result to an actual implementation. Moreover, other than for PRINCE with a single **SBOX_C** the overall Gain Factor is comparable as it went from being between 26 and 129 to being between 33 and 125 even though the actual value for each algorithm has changed.

4.8 Conclusion

In this section, we proposed a RISC-V extension called LBC-ISA which consists in 7 additional instructions. Among these instructions, some are agile and can be used by multiple algorithms, and other are specific and can only accelerate the execution of a single cipher.

Each of these instructions has been tested on an FPGA board on which had been implemented a platform which used the VexRiscv Core. They were then validated by using them as inline assembly instructions in the C code of each cipher and comparing the results to already validated results.

For each of these instructions, we have provided an evaluation of the costs both on their own and as part of the configuration dedicated to each cipher. These results showed that the most costly instruction was the agile **NMAT_D**, which was expected. Thanks to the use of **CFGLUT5**, the cost of **SBOX_C**, the other agile instruction, remained on par with the cost of cipher-specific instructions. Moreover, the cost of configuration accelerating a single cipher is less than double the cost of the Basic RISC-V ISA. When the configuration uses the complete LBC-ISA instruction, the area overhead is of 117% just a little more than doubling of the Basic RISC-V ISA cost.

These results are promising, especially when compared to the latency evaluation of these ciphers. Indeed, the executions using the LBC-ISA are from 1 to 2 orders of magnitude faster than those who do not. Thereby, some ciphers can be accelerated more than a 100 times. This entire acceleration is uses very little additional instructions and low area overhead.

Nonetheless, none of these implementations took into account protection against physical attacks. No matter the chosen implementation, physical attack is a real security issue, which must be taken into account when high levels of security are required. The next chapter will present in depth how protection against such threats was implemented and test its effectiveness against certain types of physical attacks.

Chapter 5

Protected Processor Against Side-Channel Attacks

Cryptographic algorithms are mathematically sound but the hardware on which they are implemented can be the source of information leakage. This information can be exploited by an attacker in order to retrieve the cryptographic key, wrecking their mathematical strength. This type of attack is called Side-Channel Attacks and it is essential to protect the hardware from such attacks.

The information can leak in different forms, meaning that a variety of attacks are available which each require specific responses. The leaking information we are interested in with this work is based on the power consumption. Indeed, power consumption varies when a bit in a register changes from a 0 to a 1 or a 1 to a 0 or when it stays the same during a clock cycle. By collecting a large amount of power traces, it is possible to reduce the impact of the noise to a minimum. This leads to finding correlations in behaviors which can then lead to an identification of hidden constant values, such as the key value.

The most sensitive point to observe such discrepancies is around the S-Box. Since this is the only non-linear instruction of the encryption, it is where the power consumption can reflect the values used during the execution. This is also why protections usually have a special interest in protecting the S-Boxes specifically. This will be further explained in Section 5.2.3.

Despite their importance, protection against side-channel attacks are still rather new and still lack a standard. In this chapter, we propose, test and validate a RISC-V extension, dedicated to the acceleration of LBC and coherent with the LBC-ISA proposed in Chapter 4. This extension aims at the same kind of results regarding constraint of size, speed and agility.

5.1 Proposed ISA Extension for Protection

The results from Section 5.3 show how non-protected implementations are sensitive to SCA. The mathematical soundness of the ciphers are therefore not sufficient to protect the data and the hardware itself requires protection. To be more specific the hardware cannot be protected in a way that would stop information from leaking, what is protected is the nature of the information it leaks. The protection

needs to make sure that even though leakage happens, the information leaked cannot be used to decrypt hidden information.

There are multiple ways to implement a protection against SCA (*Cf.* Sect. 2.7). One of our main constraints in choosing which protection to use was our objective of having a single agile protection. We also wanted to minimize the area cost of adding it to the RISC-V extension. The protection we chose was the RSM [53] for multiple reasons which will be detailed in the next Section.

5.1.1 Rotating S-Box Masking Protection Implementation

The RSM protection [53] was seen as fitting for our implementation. Most importantly, the RSM protection is one of the most effective in terms of ratio efficiency vs. security. Moreover, most of the RSM protection steps did not require additional hardware resources. Indeed, RSM is an effective and low complexity masking scheme, where the non-linear substitution function are tabulated. The S-Boxes are modified to unmask, substitute and re-mask atomically the state. They can also be reconfigured independently with the LBC-ISA implementation proposed (*Cf.* Sect. 4.4.3). This was therefore particularly coherent with our existing implementation as the S-Boxes could be easily changed into the protected S-Boxes with this architecture. Finally, most of the RSM protection steps did not require acceleration and a single additional hardware instructions would be enough to optimise its execution latency (*Cf.* Sect. 5.1.2).

The RSM protection is detailed in Section. 2.7.2.2. Instead of rotating the mask by rotating the S-Boxes, it is easier here, to rotate the state before and after the S-Boxes using two nibble level barrel-shifters (*Cf.* Fig. 5.1). This protection seemed particularly appropriate to implement an agile masking scheme since the 4×4 Sbox is used in many LBCs, by protecting it, all those ciphers should be protected.

Each S-Box is adapted to a certain nibble of the mask, which requires 16 different S-Box configurations. The existing SBOX instruction does not require modification since each S-Box is already configured independently during the *algorithm loading step*. It is also during this *algorithm loading step* that each table is calculated according to the mask. Since every nibble is used, only the mask nibbles are necessary and the initial RSM Index is not required.

RSM was developed with the perspective of an ASIC, therefore using a more software-based approach requires changing a few aspects of its implementation. For instance, in our implementation, the S-Boxes are not physically rotated, they are fixed, and the state is rotated. This 64-bit rotation does not exist in the basic RISC-V 32-bit ISA nor does it exist in the extended LBC-ISA. As part of our implementation, these rotations are handled by a single hardware dedicated instructions that rotate the entire state in a single instruction (technically two, for the low/high parts as explained in Section 4.4). Although this nibble-level barrel shifter hardware is unique, it can be used by two separate instructions which will be referred to as **NShift** and **invNShift**, and described later in Chapt. 5.1.2. This rotation is at the heart of the protection, especially the initial RSM Index of the

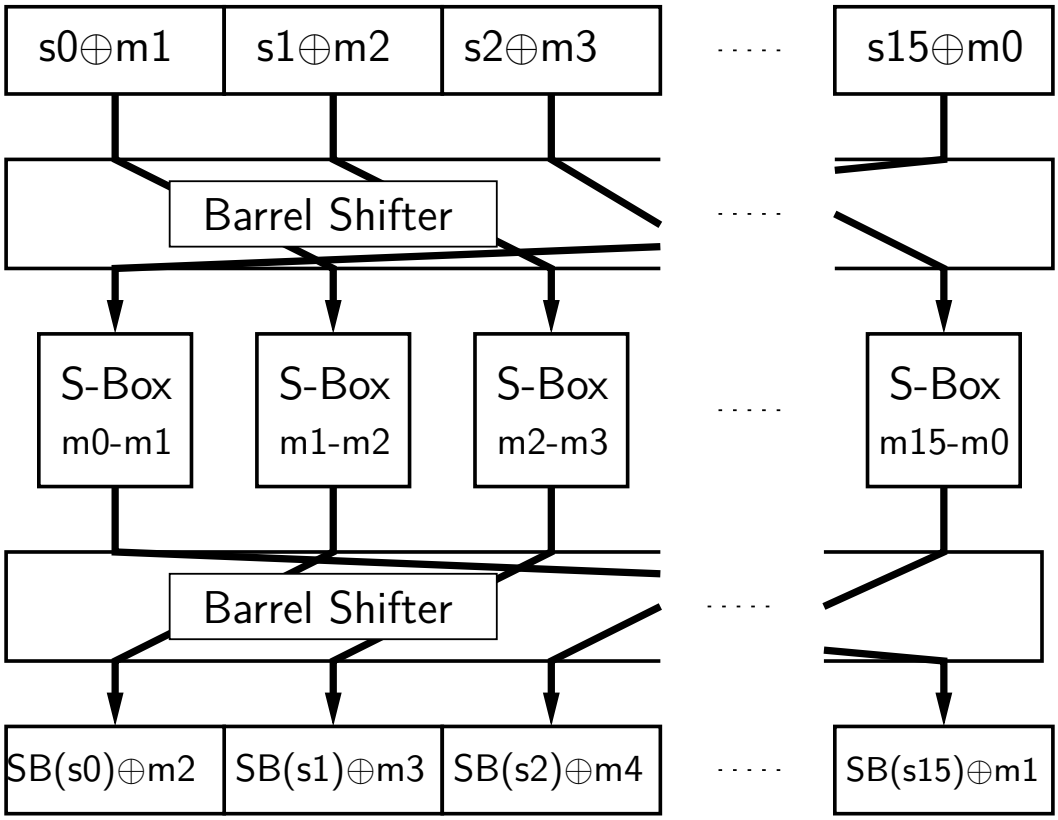


Figure 5.1: The RSM Confusion Step

rotation, which unlike the mask itself is hidden information. The shift amount of this initial Index of rotation is set with Mask_init and incremented at each round with Mask_next. These two instructions are actually CSR instructions, the former sets the value of a CSR Register to a hidden random value between 0 and 15 the ladder increments the value of this CSR Register by 1 modulo 16.

The entire execution of a protected algorithm is presented in Alg. 2. It shows each of the three steps of an unprotected algorithm along with all the instructions dedicated to the implementation of the RSM protection. Compared to Alg. 1 in Section 2.4, the S-Box table has been modified into an RSM S-Box table, which does not change the implementation of the S-Box itself. Most of the instructions are the direct handling of the mask itself, the main difference applied to the state is those two nibble-shifts before and after the *Confusion Step*. These nibble-shifts are the instructions which were identified as the most area efficient way to accelerate the execution of the RSM protection for the RISC-V protection extension.

Algorithm 2 Protected Generic Lightweight Block Cipher Algorithm

PROTECTED BLOCK CIPHER ALGORITHM

Data: PlainText (64-bit), Key (128-bit)
Parameters: Mask (64-bit), Initial RSM value (5-bit), S-Box parameters (1024-bit)
Result: CipherText (64-bit)

Mask_init()
$roundKey[\#ofRounds] \leftarrow GenerateRoundKeys(Key)$
curr_mask $\leftarrow$ **invNShift(Mask)**
state $\leftarrow$ **Plaintext** $\oplus$ **curr_mask**

for $round = 0$ to $\#ofRounds$ **do**
  $state \leftarrow state \oplus roundKey[round]$
  state $\leftarrow$ **NShift(state)**
  $state \leftarrow SBox(state)$
  state $\leftarrow$ **invNShift(state)**
  $state \leftarrow Diffusion(state)$
  Mask_next()
  curr_mask $\leftarrow$ **invNShift(Mask)**
  comp_mask $\leftarrow$ **Diffusion(curr_mask)** $\oplus$ **curr_mask**
  state $\leftarrow$ **state** $\oplus$ **comp_mask**
end for
state $\leftarrow$ **state** $\oplus$ **curr_mask return** $state$

5.1.2 Protection Instructions

The RSM protection, as its name suggests, requires a rotation. Rotation is not part of the base RISC-V ISA, but even if it were, to implement the RSM protection we need a 64-bit rotation instruction of the masked state, yet our implementation is on a 32-bit RISC-V. This means that even the basic rotation proposed in [4] would not be adapted to our requirements. The addition of a 64-bit barrel shifter, and corresponding instruction, was therefore added to implement the RSM protection with reasonable latency. To limit the hardware complexity, this barrel shifter was limited to nibble granularity.

In terms of hardware implementation, the only addition to the protection for LBCs extension of the RISC-V called ProtLBC-ISA was this nibble-level barrel shifter. It required the addition of four instructions to the ISA the low and high versions of **NShift** and **invNShift**. **NShift** and **invNShift** are both R-type instructions:

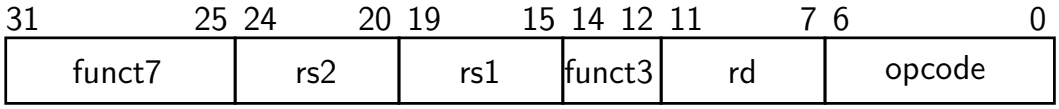


Figure 5.2: R-Type Instruction of the RISC-V

NShift and **invNShift** both use the same hardware but use different parameters. **NShift** rotates to the left a given amount and **invNShift** rotates to the right of the same amount. Quite like the other additional instructions, the nibble shift

instruction required to implement the RSM protection are not adapted to a 32-bit microcontroller. Indeed, a 32-bit micro-controller is not adapted to execute a 64-bit rotation. Nevertheless, the rotations used are only nibble-level rotation, which allowed the optimisation of the implementation of this barrel shifter.

The RISC-V instructions can only have two 32-bit registers as input and one 32-bit register as output, the rest of the bits are reserved to identifying which instruction is being called. Nonetheless, the nibble shifters need to have a shift amount parameter in order to be agile. To include this agility without tempering with the instruction format of the RISC-V, we used available CSR to store the parameter value for the shift amount. This means that before using any of those instructions, it is necessary to assign a value to a CSR register by writing in a dedicated CSR register, which will then assign a random initial shift amount value, this operation is done using **Mask_init**. In terms of hardware, it required the use of a single additional 5-bit CSR register, which has very minor costs but is important to note. Since the RSM protection requires incrementing this shift amount value, it is also included that writing in a separate CSR will add one (modulo 16) to the current shift amount value, this operation is done using **Mask_next**.

Using a separate CSR registers to store, initiate and increment the value has a second important use as it avoids having to store the hidden index value in a normal register. It is important to be aware that this index is important to the security of the RSM protection. For practical evaluation reasons, the implementation that was used does not hide that information as it was important to have control over it. Nevertheless, in an actual implementation, it should be randomly generated at the beginning of the execution. This random generation should be done at a hardware level, using a non-deterministic method such as TRNG, details on this implementation are out of scope for this work, for reference, the work of Saarinen *et al.* [70] shows how such a TRNG could be implemented.

One last thing that was looked at carefully was the registers themselves, each variable had its own dedicated register. This precaution was taken because crushing the value of a register with another value generates an activity, if this activity is correlated to an unmasked state, the entire protection could be broken.

5.1.3 Implementation Analysis

The cost of this protected extension is shown by looking at both the hardware cost and the latency costs.

5.1.3.1 Hardware Resources

The results of this protected implementation using ProtLBC-ISA are compared to the implementation of the basic RISC-V and to the implementation of the extended LBC-ISA with the S-Box instruction along with the same "_D" instructions. All the synthesis results are shown in Tab. 5.1.

Table 5.1 shows the area costs of each possible configuration of the RISC-V extension from the Basic RISC-V ISA to the complete ProtLBC-ISA. Each configuration corresponds to the additional instructions required to accelerate the

execution of a given cipher or group of ciphers. The configurations marked with a specific cipher such as "PRESENT" corresponds to the resources required to accelerate the execution of this cipher. The configurations marked with "NMAT" correspond to the resources required to accelerate the execution of any cipher using the **NMAT_D** instruction. The configurations marked with "ALL" correspond to the combination of all the other configurations. Each configuration is presented both for the non-protected and the protected extension. The LUT column corresponds to the total amount of LUTs required for the implementation and the FF colmun to the amount of flip-flops required. The former corresponds to the area dedicated to logic operations and routing whereas the latter corresponds to the memory requirements. Each of these colmuns is followed by a column showing the augmentation percentage compared to the Basic RISC-V ISA implementation.

These results show a clear increase between the non-protected and the protected extensions. First off, it can be noted that the FF increase is most largely linked to the agility of an instruction rather than to its protection but protection still has a cost. For instance, the FF increase on specific instructions is less than 0.5% for non-protected configurations but goes up to 5% for protected configurations. This increase is most likely due to the memory cost of adding a mask to the implementation. Nonetheless, this increase stays far from the 33% increase due to the use of the agile **NMAT_D** instruction. Secondly, protection also caused an increase when looking at the costs in terms of LUTs. Indeed, the smallest area required for the non-protected extension corresponds to the "GIFT" which shows a 13% increase compared to the Basic ISA. Whereas the smallest area increase for the protected extension which also corresponds to the "GIFT" configuration more than doubles the size of the implementation. This difference lowers as the non-protected extension cost grows. Indeed, the non-protected "ALL" configuration shows a 117% increase compared to the Base ISA whereas the protected "ALL" configuration increase the LUT cost by 167%. This increase is therefore of only 50% when implementing the entire ProtLBC-ISA extension.

Table 5.1: Resource Usage Overhead for the Agile Protected Implementation

ISA	Configuration	Additional Instructions	LUTs	%	FFs	%
Reference Basic RISC-V ISA	None	-	973	-	765	-
Non-Protected Extended RISC-V LBC-ISA	PRESENT	S-Box + PRESENT_D	1173	+21	767	+0.1
	GIFT	S-Box + GIFT_D	1103	+13	767	+0.1
	PRINCE	S-Box + PRINCE_DF + PRINCE_DM + PRINCE_DL	1438	+48	769	+0.5
	NMAT	S-Box + NMAT_D	1778	+83	1023	+34
	ALL	S-Box + PRESENT_D + GIFT_D + PRINCE_DF + PRINCE_DM + PRINCE_DL + NMAT_D	2113	+117	1028	+34
Protected RISC-V Extension ProtLBC-ISA	Basic Protection	Nibble-Level Barrel Shifter	1702	+75	799	+0.4
	PRESENT	S-Box + PRESENT_D + Nibble-Level Barrel Shifter	1971	+103	804	+0.5
	GIFT	S-Box + GIFT_D + Nibble-Level Barrel Shifter	1968	+102	804	+0.5
	PRINCE SBOX_C x1	S-Box + PRINCE_DF + PRINCE_DM + PRINCE_DL + Nibble-Level Barrel Shifter	2027	+108	802	+0.5
	PRINCE SBOX_C x2	S-Box + PRINCE_DF + PRINCE_DM + PRINCE_DL + Nibble-Level Barrel Shifter	2227	+129	803	+0.5
	NMAT	S-Box + NMAT_D + Nibble-Level Barrel Shifter	2432	+150	1057	+38
	ALL (with Protection)	S-Box + PRESENT_D + GIFT_D + PRINCE_DF + PRINCE_DM + PRINCE_DL + NMAT_D + Nibble-Level Barrel Shifter	2595	+167	1061	+39

5.1.3.2 Latency Analysis

As shown in Alg 2, the RSM protection adds instruction to each round of a cipher. This means that the latency is necessarily increased. Table 5.2 compares the latency cost of non-protected executions of ciphers using the basic RISC-V ISA to that of protected executions using ProtLBC-ISA.

Table 5.2: Instruction Count for Execution of Ciphers using Base RISC-V ISA and ProtLBC-ISA

Cipher	Base ISA	ProtLBC-ISA	Gain Factor
PRESENT	12544	812	15
GIFT	10661	788	14
PRINCE (SBOX_C x1)	17357	4642	4
PRINCE (SBOX_C x2)	17357	323	54
Midori	18944	412	46
TWINE	41279	1429	29

As expected from Tab. 4.7, the results from PRINCE (SBOX_Cx1) are quite low compared to the acceleration of other ciphers. Nonetheless, it still shows that even with protection our extension accelerates by a factor of at least 4 any of the studied ciphers. These results also show that for other ciphers, the protected execution is between 15 and 54 times faster. When using protection, each round is impacted by additional instructions. Therefore, these results showing over a magnitude order of acceleration are very encouraging as they show just how much a few instructions can offer when dealing with LBCs.

In this section we covered the 9 instructions of ProtLBC-ISA, the costs of their implementations and the latency gain they offer. In the next section we will present the means to evaluate the protection offered by this extension.

5.2 Side-Channel Security Evaluation

In order to determine the security of the implementation, different metrics where used for different purposes.

- The Normalized Inter-Class Variance (NICV) [16]
- The Correlation Power Analysis (CPA) [21]
- The Guessing Entropy (GE) [50]

The NICV was used to establish the framework of the analysis. The CPA was used as the main tool for the attack as it highlights the correlation between the power traces and the hidden values. The GE was used as a way to evaluate and compare the security level of non-protected and protected implementations.

5.2.1 Experimental Setup

Working on SCA requires the use of a large amount of power traces. For the acquisition of the power consumption execution traces, two boards were used. The first is the cw-305 Chipwhisperer FPGA board [2], where the protected RISC-V is implemented. The second, is the analysis board, the cw-1173 Chipwhisperer-Lite [1] which is used to collect the execution power traces.

The working frequency of the RISC-V was set to 10MHz, and the sampling frequency for the power traces was set to 100MHz, which corresponds to the maximum sampling rate of the analysis board. This frequency of 10MHz was chosen so as to have 10 samples per clock cycle which is sufficient for our analysis. Whether on protected or non-protected implementations, we captured 500 samples per trace. This allowed to capture the entire first round along with some margin.

For each algorithm, we gathered 32768 power traces for the non-protected version and either 1048576 or 2097152 power traces for the protected versions. Some algorithms required an additional 524288 power traces for the non-protected version due to loop unrolling issues (*Cf.* Sect. 5.3.4). Each trace was retrieved using the same key and a different non-redundant random plaintext. In the case of protected implementations, the same mask was used, but initial index value was changed, so that every one was used for an equal part of the overall traces. The mask used was the same for each cipher and its value is $0x693c5a96c3a5c3a5$, it was chosen so that every nibble had a Hamming Weight of 2.

5.2.2 Normalized Inter-Class Variance

Multiple methods can be used to determine the feasibility of side-channel attacks, such as the Test Vector Leakage Assessment [87] but it was decided that the NICV (*Cf.* Sect. 2.6.6.3) would be a better fit for our work. The NICV graph compares the power activity of the overall hardware with the power activity correlated to the given input value. Fig. 5.3 shows the overall power activity variation trace and the NICV graph of a non-protected version of PRESENT. The peaks observed on the NICV graph are where the leakages are susceptible to happen as they show a suspect activity correlated with the input value. This does not mean that each of these peaks can be attacked, simply that this is where an attack is most likely to work. NICV therefore allowed us to identify the most fragile point in the execution. Nonetheless, this method requires fine-tuning for a few reasons:

- The NICV indicates potential leakage, but does not guaranty it is correlated to the secret information
- The height of the NICV peak does not indicate how likely this peak is to suffer leakage

In the NICV graph of Fig. 5.3, we see multiple peaks, with the highest value around abscissa 370, yet this sampling point corresponds to the slot delay at the end of the second round. First order attack models can only exploit the information of the first round, therefore having an NICV peak at the end of the second round

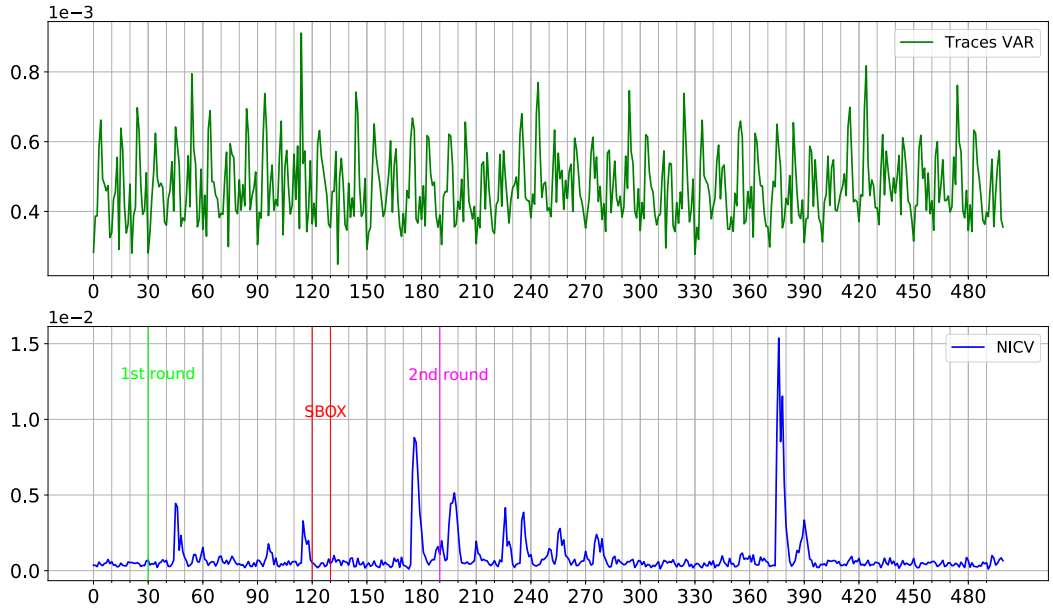


Figure 5.3: The NICV for non-protected PRESENT with 32768 traces

suggests that the attack used should not work at this point. By using the simulation traces of PRESENT in addition to the NICV curve, it is possible to identify the peaks that will lead to successful attacks. Indeed, some instructions such as the S-Box are more susceptible to attacks. If an NICV peak is observed close to the S-Box, then the spectrum of the attack should be limited to its range. In Fig. 5.3 a small peak is observed at abscissa 110, this peak corresponds to the execution cycle of the SBOX instruction of the first round. Although it is not the highest peak in term of NICV, it is the focus point for the attack. To note, this graph is traced for a specific nibble, but the same behavior can be observed with most of the other nibbles. The NICV graph was therefore combined with the use of the simulated execution traces of PRESENT to pinpoint which peaks had the most potential of being attackable.

5.2.3 Correlation Power Analysis

The NICV allows precise selection of a range of samples on which to apply the CPA attack (*Cf.* Sect. 2.6). This analysis uses execution power traces to retrieve the nibbles of the secret key one by one.

The concept of the CPA is to execute a given cipher with a fixed key and a changing plaintext and to record the power traces of each of those executions. The mean of each of these power traces is then compared to the expected value for each possible hypothesis and whichever hypothesis comes closest, or rather has the highest CPA peek is deemed the most likely hypothesis. If the hypothesis corresponds to the real key, then the key has been broken, otherwise the cipher is unaffected by the attack.

The fact that we use 64-bit blocks means it is extremely inefficient to use the

CPA on each of the possible values of the 64-bit round key. The hypothesis is actually done on a 4-bit sub-part of the state, which is the output of the 4×4 Sbox. Indeed, S-Boxes are the only non-linear operation of the cipher and apply to groups of 4 bits, which allows two things. First non-linearity ensures that there is a correlation between the power traces and the Hamming Weight or Hamming Distance of the hidden key, ensuring that the right hypothesis behaves is an identifiable way. Second, by having a 4-bit input and a 4-bit output, the amount of possible values is reduced to 16, which can be brute forced. Therefore, the output of the S-Boxes is the most sensitive point of the cipher and where the attack will most likely take place [86].

For each nibble, hypotheses are made for its value from the 16 possible values. The *Pearson Correlation Coefficient* is computed for each hypothesis, and the one with the highest coefficient value is determined to be the most probable value for this nibble of the key. If this identified nibble corresponds to the real key nibble, then this nibble of the key is considered broken. All the nibbles do not need to be broken for the security to be inefficient as with a sufficient amount of broken nibbles, a more refined attack could still work.

5.2.4 Guessing Entropy

In order to evaluate the attackability of the studied algorithms, we used the Guessing Entropy GE as our indicator (*Cf.* Sect. 2.6.7.3). When using the CPA, each hypothesis is given a rank based on the highest value of its CPA graph. The hypothesis with rank 1 is the nibble value most likely to be correct according to the attack. When this value is the same as the real key nibble value, the nibble is said to be broken. Therefore, the higher the rank of a given value is, the most likely this value is to be the real key value, according to the attack.

This rank can vary due to multiple factors, the main one being the amount of power traces used for the CPA. Indeed, the higher the amount of power traces are used, the higher the correlation is for the right key nibble. The guessing entropy graph represents the rank variation according to the amount of power traces used. The rank used is actually an average between the ranks obtained for different power traces sets of the same size.

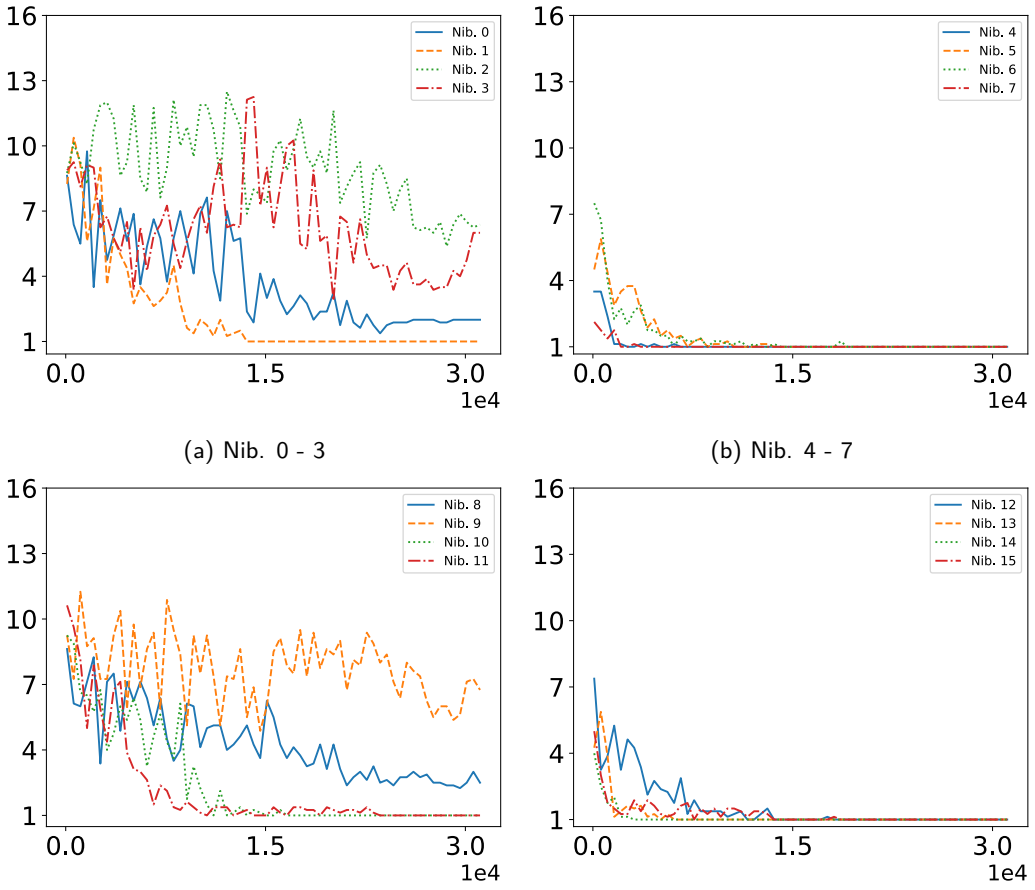
The GE was therefore used a measure of the resistance to SCA and to compare the results of non-protected and protected implementations. If a nibble can be broken, then as the sample size grows, the guessing entropy converges towards one.

The guessing entropy of the non-protected PRESENT in Fig. 5.4 shows that 11 of the 16 nibbles can be broken with less than 32k traces. It is interesting to note that even though most nibbles break for set sizes around 10000, some nibbles do not seem to break at all. Indeed, CPA attacks rely on Hamming Weight to determine the right key nibble which is sometimes based on very small changes. This means that in certain cases, a much larger amount of traces is needed to break a nibble. Still, we can see an overall decrease of the guessing entropy which suggests that with enough traces, each nibble will be broken.

5.3 Security Evaluation of Non-Protected Implementations

The first security evaluation has to be done on non-protected implementation in order to verify that the leaking information does allow the attacks to work. Our first evaluation used the GE of each of the 16 nibbles for growing sizes of power traces groups. These group size go from 100 up to 32768 power traces, and for each group size, the GE is calculated for 4 different groups to have an average. This evaluation was done for PRESENT, GIFT, PRINCE and Midori. These algorithms were selected in order to have each additional instruction used at least once.

5.3.1 Non-Protected PRESENT



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

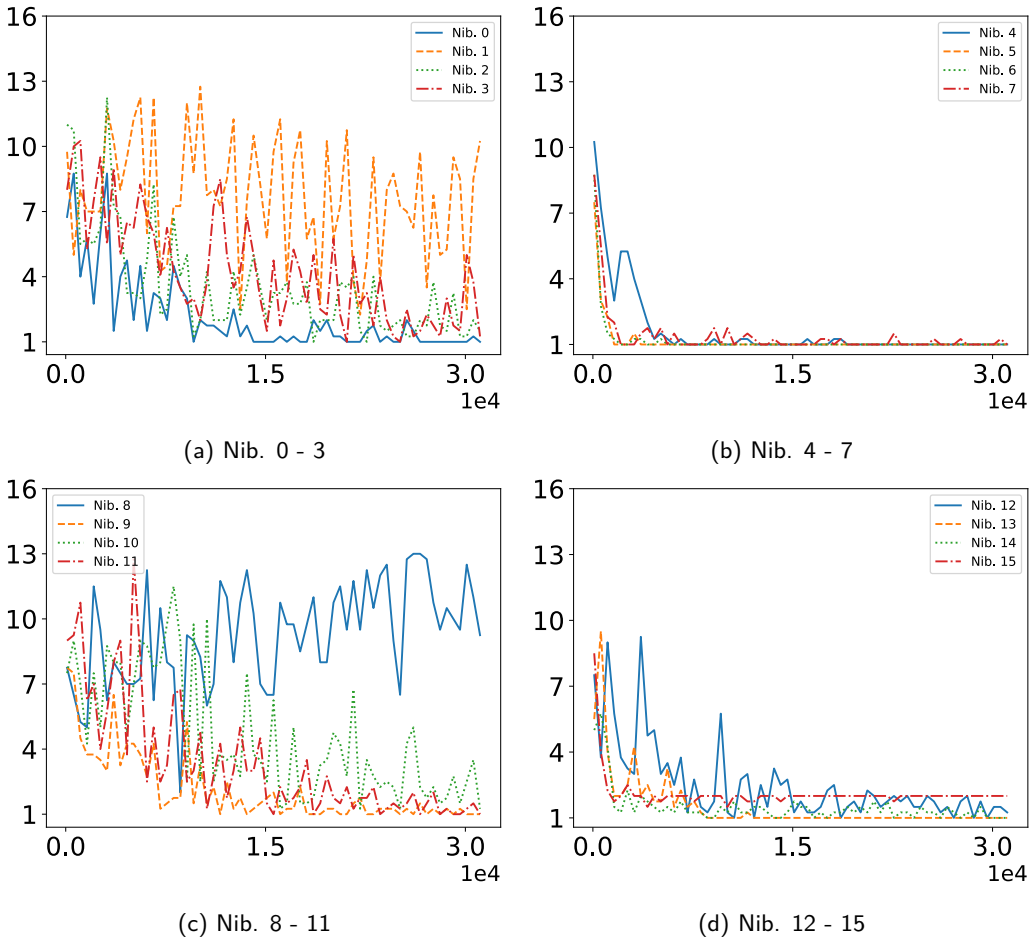
(d) Nib. 12 - 15

Figure 5.4: Guessing Entropy of a Non-protected Execution of PRESENT, up to 32768 Power Traces

Figure 5.4 is the graph of the GE for each nibble of the non-protected PRESENT cipher with up to 32768 power traces. It shows how 11 of the 16 nibbles of PRESENT see their GE converge towards 1 and the 5 others are slowly deceasing. This means that the value of those 11 nibbles has been broken by the attack with less than 15k power traces, which shows the vulnerability of such implementation

to SCA. Of the non-broken nibbles, most seem to slowly decrease and would eventually converge towards 1.

5.3.2 Non-Protected GIFT



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

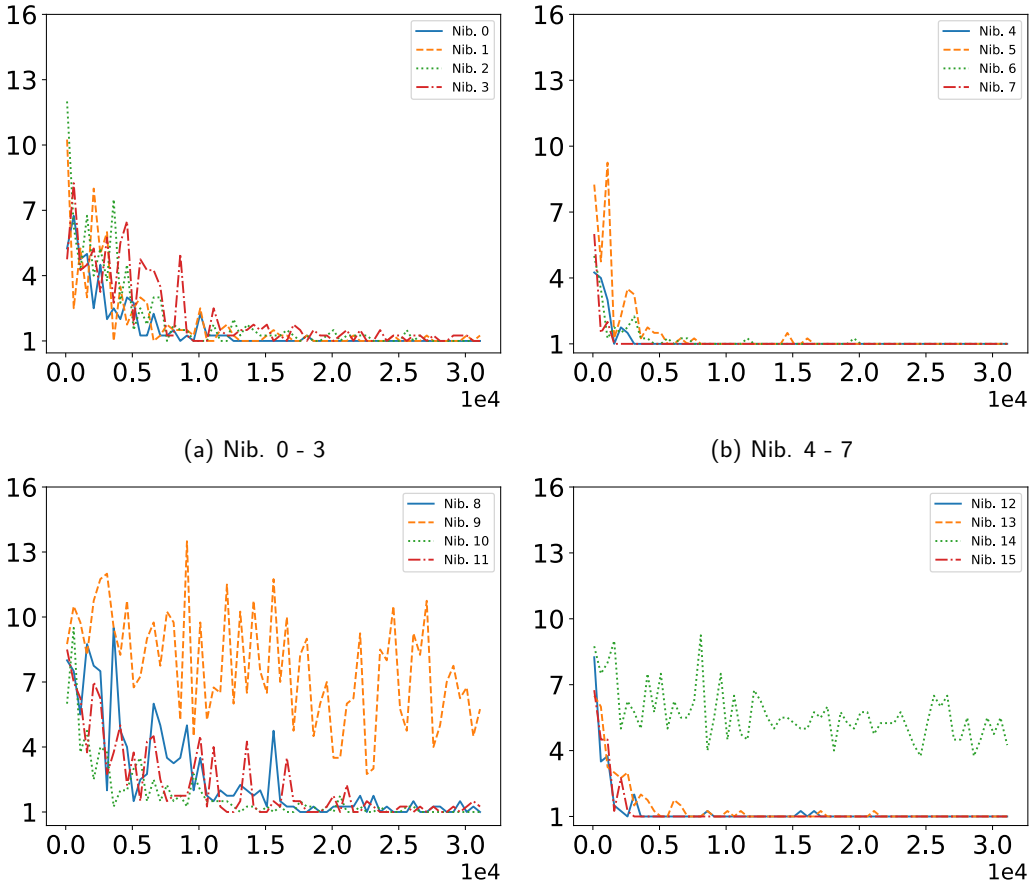
(d) Nib. 12 - 15

Figure 5.5: Guessing Entropy of a Non-protected Execution of GIFT, up to 32768 Power Traces

Figure 5.5 is the graph of the GE for each nibble of the non-protected GIFT cipher with up to 32768 power traces. It shows how 13 of the 16 nibbles of GIFT see their GE converge towards 1. This means that the value of those 13 nibbles has been broken by the attack with in most cases less than 15k power traces, which shows the vulnerability of such implementation to SCA. Of the non-broken nibbles, one of the other nibbles shows a very slow decrease, another converges toward 2, and the last shows a behavior that is unexpected as it tends to increase. The fact that three nibbles that were not broken is probably due to a specific interaction we were unable to identify.

5.3.3 Non-Protected PRINCE

Fig 5.6 is the graph of the GE for each nibble of the non-protected PRINCE cipher with up to 32768 power traces.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

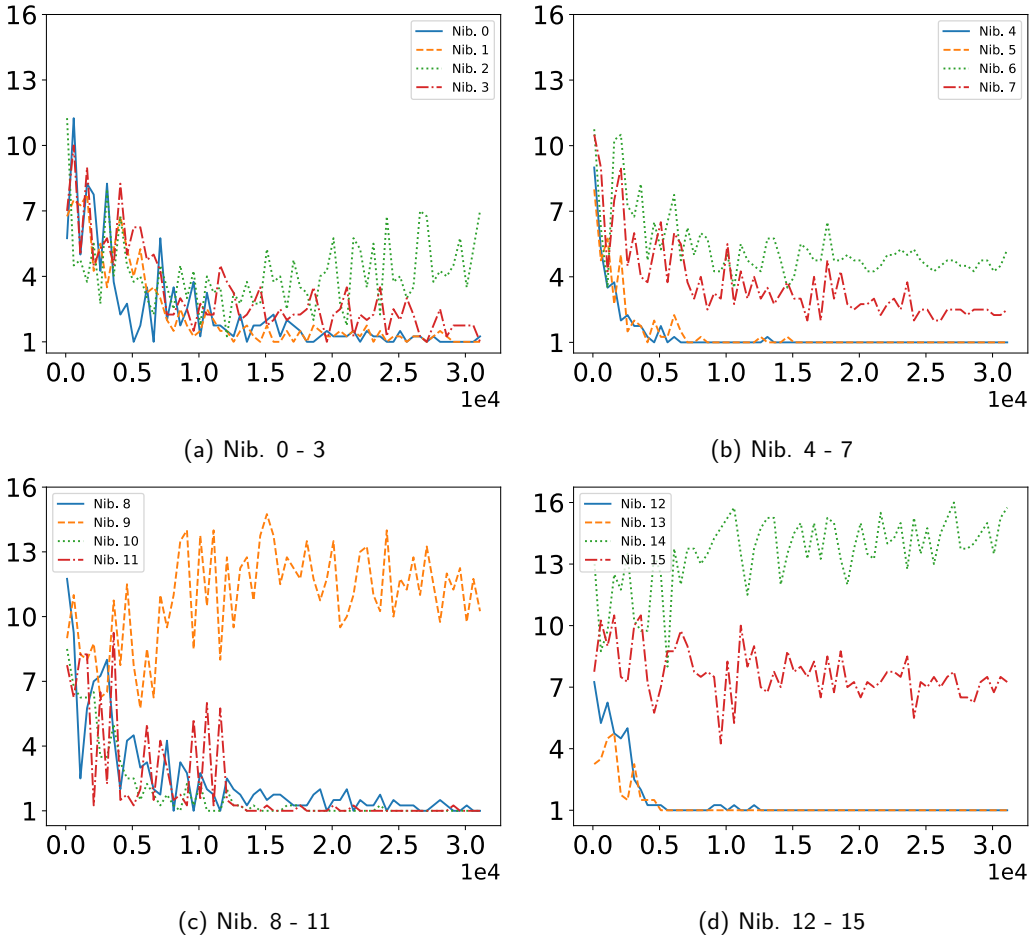
(d) Nib. 12 - 15

Figure 5.6: Guessing Entropy of a Non-protected Execution of PRINCE, up to 32768 Power Traces

It shows that 14 of the 16 nibbles of PRINCE see their GE converge towards 1, meaning that they are broken by the attack. Most nibbles see their GE at 1 after 5000 to 15000 power traces which shows great vulnerability against SCA. The behaviour of the other two is different as they seem to either decrease slowly or converge towards a non-1 value.

5.3.4 Non-Protected Midori

Figure 5.7 is the graph of the GE for each nibble of the non-protected Midori cipher with up to 32768 power traces. It shows how 10 of the 16 nibbles of Midori see their GE converge towards 1. This means that the value of those 10 nibbles has been broken by the attack with in most cases less than 15k power traces, which shows the vulnerability of such implementation to SCA. Amongst the non-



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

(d) Nib. 12 - 15

Figure 5.7: Guessing Entropy of a Non-protected Execution of Midori, up to 32768 Power Traces

converging nibbles, most seem to converge towards a value other than 1. Once again, the explanation for this phenomenon could not be identified.

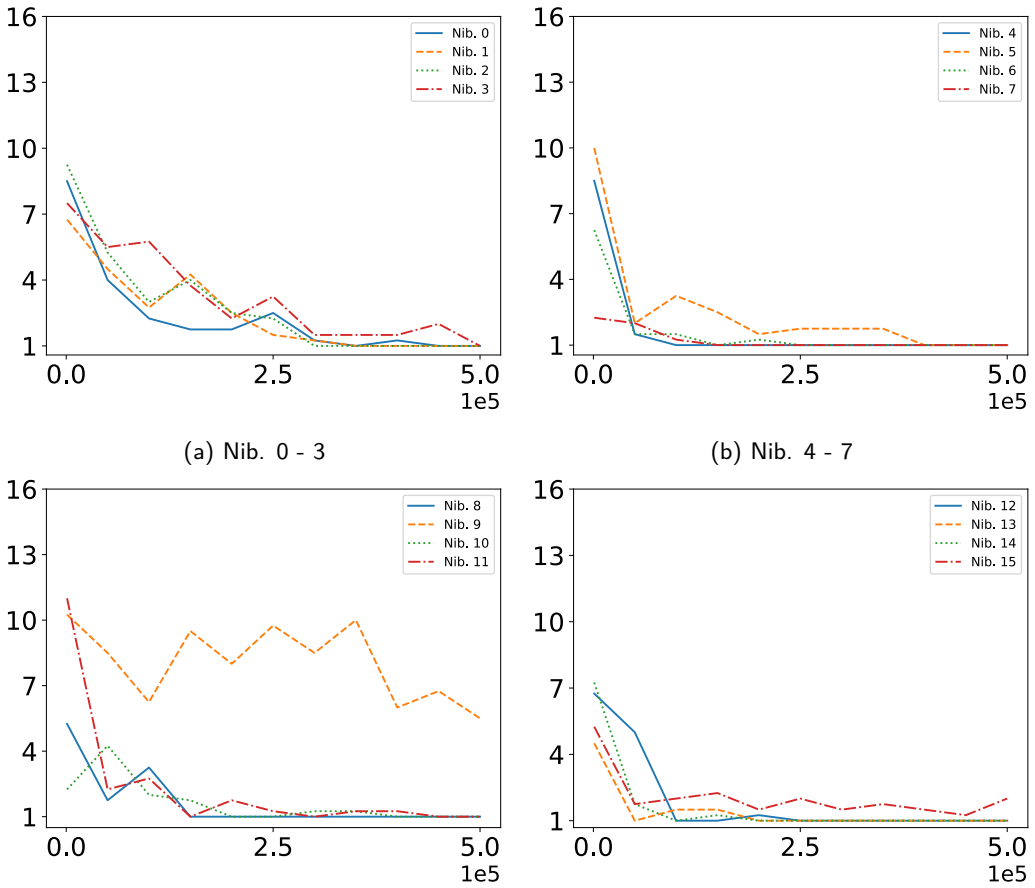
5.3.5 Conclusion

These results show that with unprotected executions, ciphers can be broken with low amounts of power traces. Indeed, most nibbles were broken with under 15000 traces. These graphs therefore illustrate the need for protection consideration for the implementation of LBCs.

Nonetheless, for these attacks, some nibbles could not be broken. The explanation as to why certain nibbles are less sensitive to the CPA attacks has not been found. Among the hypothesis, it could be possible that the execution of the 16 S-Boxes would not be done in parallel, despite our parallel implementation, which could lead to the S-Box output to be found at different points for different nibbles. This would lead to a lower correlation for some nibbles but not others. It could be possible that by enlarging the study points area those nibbles would become more sensitive to the attack, but this solution was not tested.

Another possible way to identify the issue or even make it disappear would be to use another type of attack, another attack model or another distinguisher. Once again, these solutions were not tested but could be a way to go around the difficulties of attacking those nibbles.

Moreover, during our collection of guessing entropy for Midori, we came across another interesting phenomenon. Our initial results showed that the guessing entropy of Midori would decrease down to 1 the same way as it does in Fig. 5.7, except for much higher values of around 200,000 power traces as shown in Fig. 5.8. The reason we found for such a difference between Midori and other algorithms was that due to the low number of rounds, the compiler had unrolled the rounds loop. This was verified by forcing the compiler avoid unrolling, which gave the results shown in Fig. 5.8.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

(d) Nib. 12 - 15

Figure 5.8: Guessing Entropy of a Non-protected Execution of Midori, up to 500000 Power Traces

Out hypothesis as to why this happened is that the unrolling lead to a reorganisation of the registers used to store the results of **SBOX_C**. This could have resulted in lowered correlation as this register was not the same for each round. This meant that value stored could get crushed by a value with much less in common, unlike in a rolled version where the same register always used for the same

purpose. We did not dig any further to identify the actual cause of this gap but this observation seemed noteworthy nonetheless.

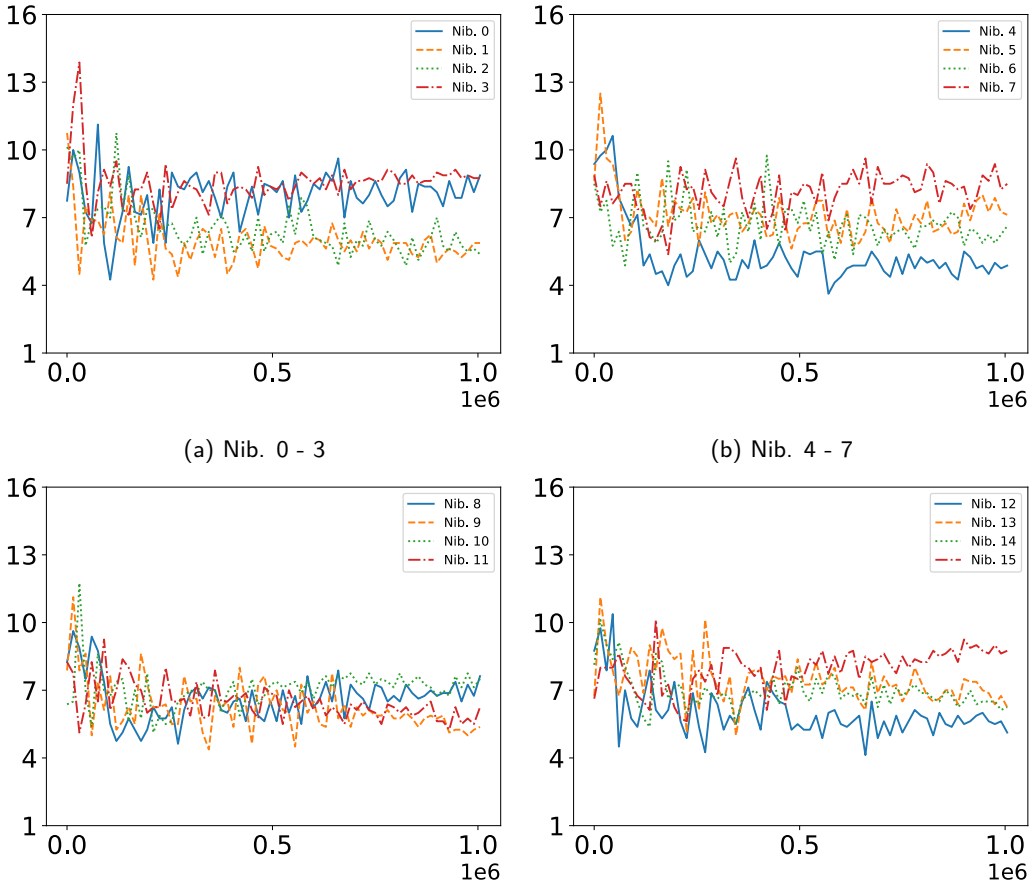
5.4 Security Evaluation of Protected Implementations

In order to test our protected implementation and its resistance to SCA, we used the same measures as in Section 5.3. The only difference is that we used a larger scope with up to 1 or 2 million power traces, to make sure the protection would work for much higher values. In order to ensure maximum security, the mask and/or key should be changed regularly meaning more often than once every 1 million encryptions. It was therefore not necessary to verify the security passed 1 million power traces.

The results for each algorithm are shown with two sets of graphs, they show the GE for each nibble of each cipher for up to 1 million power traces.

5.4.1 Protected PRESENT

The Guessing Entropy of the protected version of PRESENT is shown in Fig. 5.9. It shows that with up to 1048576 power traces, the GE of every nibble is rather stable between 4 and 10. Therefore, these graph show that with the protection, the PRESENT cipher is much less sensitive to CPA attacks. Indeed, some nibbles of the non-protected version would start breaking around the 10000 traces mark. Here with 100 times more power traces, there is no sign of the attack working. None of the curves seem to be decreasing towards 1 but rather to converge towards another value. This could indicate that the attack is still a large amount of traces away from being able to work.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

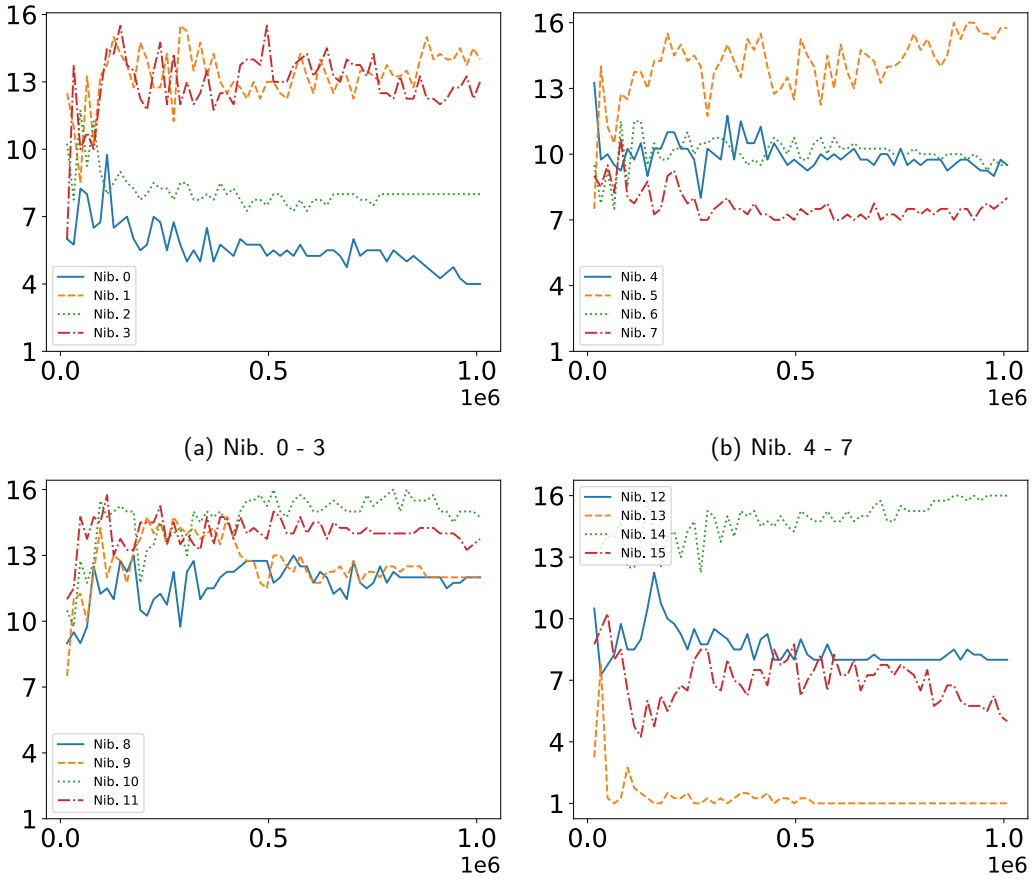
(c) Nib. 8 - 11

(d) Nib. 12 - 15

Figure 5.9: Guessing Entropy of a Protected Execution of PRESENT, up to 1 million power traces

5.4.2 Protected GIFT

The Guessing Entropy of the protected version of GIFT is shown in Fig. 5.10. It shows that most nibbles seem to converge towards a given value. They all converge towards non-1 values except for one. Nibble 13 converges towards 1 which would mean that this nibble is broken. Nonetheless, this doesn't mean that the entire key is broken. Indeed, by nature the mask hides the value which creates correlation with a different value. It is nonetheless possible that the value behind which it hides is the one with the most correlation. Indeed, if we consider the value to be random, there is a statistical chance that this value corresponds to the key nibble. Since all the other nibbles have random values, it would be very hard for an attacker to identify which nibble has actually been broken. Once again, the fact that the values converge towards a non-1 value shows that 1 million power traces is probably far from enough for the attack to work.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

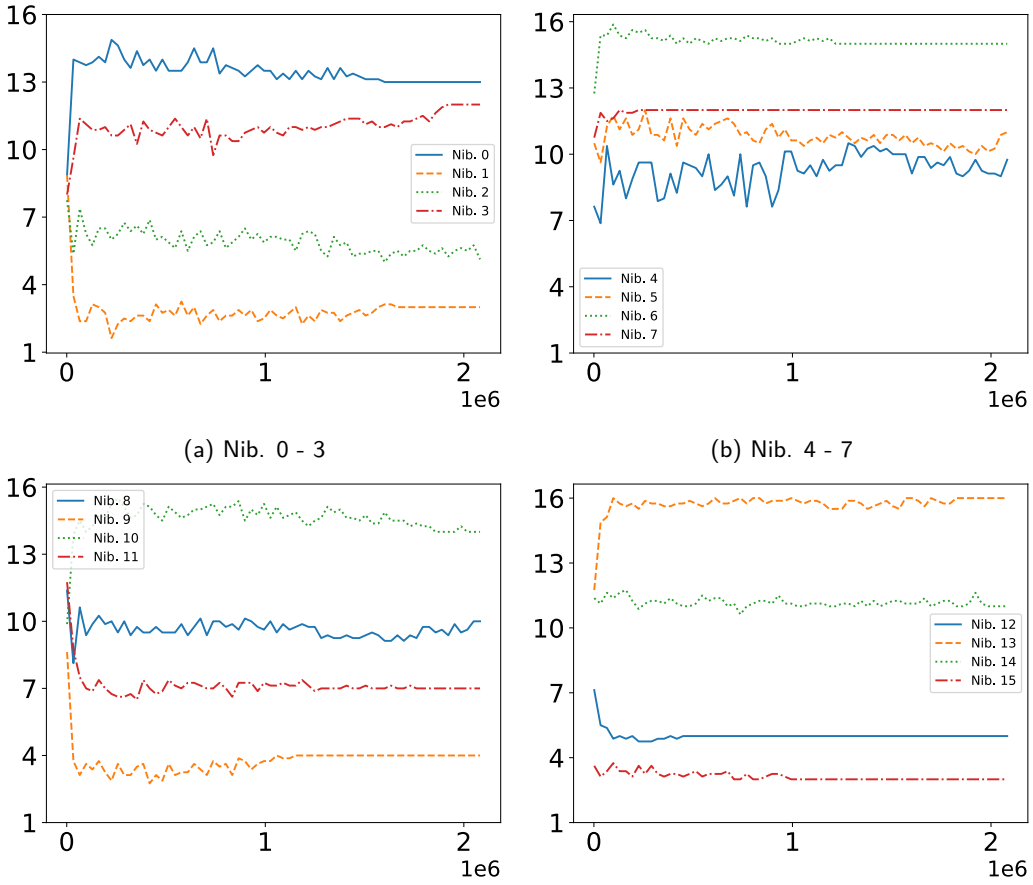
(d) Nib. 12 - 15

Figure 5.10: Guessing Entropy of a Protected Execution of GIFT, up to 1 million power traces

5.4.3 Protected PRINCE

Fig 5.11 is the graph of the GE for each nibble of the protected PRINCE cipher with up to 2097152 power traces. It shows that the GE of each nibble converges towards a value which appear random. In this case, none of the nibble has a value of 1 which comes to show that none are broken. This statement is true for up to at least 2097152 power traces, twice as much as with PRESENT or GIFT. The fact that the behavior of the GE shows no signs of decrease towards 1 could indicate, as suggested by [53], that the attack would be unlikely to succeed for very large amounts of traces.

These results also confirm that the protection that worked with the PRESENT and GIFT ciphers, which have similar *Diffusion Steps*, also works for PRINCE which has a very different *Diffusion Step*.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

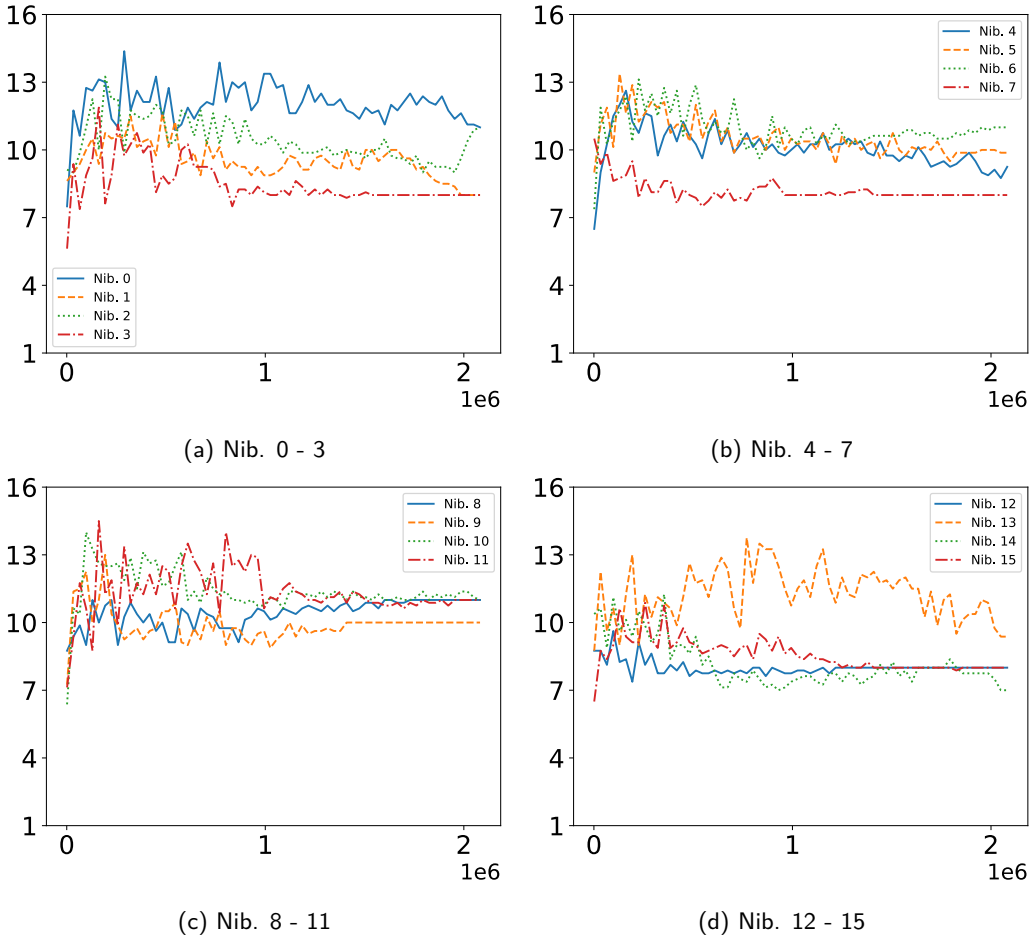
(d) Nib. 12 - 15

Figure 5.11: Guessing Entropy of a Non-protected Execution of PRINCE, up to 2 million Power Traces

5.4.4 Protected Midori

The Guessing Entropy of the protected version of Midori is shown in Fig. 5.12. The results for Midori are very close to those of PRESENT. All the GE values are rather stable between 7 and 12 and seem to converge towards a non-1 value. The amount of power traces was increased to 2097152, just as for PRINCE. As expected when looking at the PRESENT and GIFT results, it would seem that the values tend to keep on converging rather than decreasing. As for PRINCE, these results seem to indicate that an attack would be unlikely to succeed even for very large amounts of traces.

These results also confirm that the same protection also works with the agile **NMAT_D** instruction. This indicated that the protection should work with other algorithms using this instruction, although the results are not available.



(a) Nib. 0 - 3

(b) Nib. 4 - 7

(c) Nib. 8 - 11

(d) Nib. 12 - 15

Figure 5.12: Guessing Entropy of a Protected Execution of Midori, up to 2 million power traces

5.4.5 Leakage Due to Instruction Ordering

During this study, we came across an unexpected source of leakage which lead to the protection completely failing. The NICV was useful in identifying the source of this leakage during the analysis of the protected implementation. Initially, the code presented in Alg. 2 was slightly different. After the *Confusion Step*, the order in which the instructions were executed was as in Alg.3.

Algorithm 3 Leakage due to Instruction Ordering

RSM_next()
curr_mask $\leftarrow$ **invNShift(Mask)**
$state \leftarrow Diffusion(state)$
comp_mask $\leftarrow$ **Diffusion(curr_mask)** $\oplus$ **curr_mask**

With this ordering, the *Diffusion* instruction is used twice consecutively, first with the state as input then with curr_mask. This should not be a problem, especially considering that each result is stored in a different register.

Nonetheless, after close inspection, it was noticed that the output of the Arith-

metic and Logic Unit (ALU), was showing the value of *Diffusion(state)*, before being replaced by the value of *Diffusion(curr_mask)*. This results in a power activity correlated to the Hamming distance between those two values.

Let us take a look at what happens at this particular node:

- Let N_n be the state of the node when it holds the value of *Diffusion(state)*, or $D(us \oplus m_{i+1})$, where us is the unmasked state, and m_{i+1} the mask after the *Confusion Step*.
- Let N_{n+1} be the state of the node when it holds the value of *Diffusion(curr_mask)*, or $D(m_{i+1})$
- Let $HD(A, B)$ be the Hamming distance between A and B
- Let $HW(A)$ be the Hamming weight of A

We have:

$$HD(N_n, N_{n+1}) = HW(D(us \oplus m_{i+1}) \oplus D(m_{i+1}))$$

Since D is linear:

$$\begin{aligned} HD(N_n, N_{n+1}) &= HW(D(us) \oplus D(m_{i+1}) \oplus D(m_{i+1})) \\ &= HW(D(us)) \\ &= HW(us) \end{aligned}$$

This shows that the power activity linked to these two consecutive *Diffusion* instructions is directly correlated to the unmasked state.

The source of leakage was identified thanks to the NICV indicating where the leakage was coming from. Although we had taken into account this potential source of leakage by using different registers for each variable, we did not anticipate that the micro-architecture and datapath of the RISC-V implementation could lead to breaking the protection. As it turns out, even if this intermediate value is not in an accessible register, the activity resulting from modifying its value is sufficient to break the protection. It is therefore essential to pay great attention to the instructions ordering to avoid the unmasking of sensitive data.

5.5 Conclusion

Overall this research has shown excellent results for the protection, which requires at least 100 times as many power traces to break than without protection. These results are also positive when considering the agile aspect as the same RSM instructions were used to protect 4 different algorithms, each using different diffusion instructions.

Moreover, the cost of the protected extension has proven to be quite light when compared to the non-protected implementation. The fact that the protection used the same **SBOX_C** instruction allowed to minimize the overhead of the

protection. This is also true of the latency since even the protected execution duration is reduced at least 4 fold and up to 54 fold. These results also come to show that with a carefully chosen S-Box implementation, the protection against correlation power attacks can be achieved with a single additional instruction and low overhead.

The next chapter will conclude this work by presenting the different results of each contribution. It will also give insight on future works that could be done to complete and expand this one.

Chapter 6

Conclusion

Throughout this work, many steps were taken to adapt and refine an agile acceleration of the execution of Lightweight Block Cipher. From our initial idea of having a dedicated configurable ASIC to a more malleable ASIP which lead to thinking our implementation as an extension to the RISC-V ISA. This work also includes the implementation of an agile RSM protection against side channel attacks for LBCs, still as a RISC-V extension.

This chapter sums up the results we obtained and what we learned from them. It also presents the perspectives such a work offers and how it could be extended.

6.1 Results of an Agile Implementation

As this study showed, an agile implementation for the acceleration of the execution LBC can be achieved with less than 10 additional instructions, our extension used 7, even in a restricted environment. These instructions were validated on 7 ciphers. Moreover, the different evaluation methods showed that:

- Both the configurable instructions **SBOX_C** and **NMAT_D** can be used by multiple LBC
- By choosing only 7 instructions, it is possible to greatly accelerate the execution of multiple ciphers up to 100 folds
- Despite the agility, the area overhead is a less than doubled when implementing the instructions for any algorithm

This study also showed that on a fully hardware implementation configurability has a large impact on the overhead. Indeed, Tab. 3.2 shows how over 30% of the cost came from the routing and its configuration, which is much lower on an FPGA. Nonetheless, the results from chapter 3 were a great basis to work on for the RISC-V extension. It allowed to identify some of common grounds between algorithms, how to implement them and where to set the limit in terms of agility. The RISC-V processor approach led us to use cipher-specific instructions instead of bit-level agility.

Out of the 7 final instructions of the LBC-ISA 2 are agile and 5 are cipher specific. The first agile instruction is **SBOX_C** which has the function of 16 parallel 4×4

S-Boxes and is used in every of the studied algorithms. The large cost of this high level of agility instruction was avoided through the use of the Xilinx primitive **CFGLUT5** reduced its area to the same as a cipher-specific instruction. Indeed, despite its agility Tab. 4.6 shows that the cost is around 200 LUTs which represents a 20% size increase. This is very close to cipher-specific instructions which each represent between a 14% and 17% size increase. The second agile instruction is **NMAT_D** which is also used by multiple ciphers. For this instruction which uses a straightforward RTL implementation, the cost is higher but still under a 60% size increase.

The ProtLBC-ISA also added an extra hardware instruction which is used for two software instructions **NShift** and **InvNShift**. This single additional hardware is sufficient to implement the entire RSM protection. Indeed, this protection applies directly to the S-Box and each of the studied LBC uses an S-Box. Once again the area overhead is quite low with a 75% increase compared to the basic RISC-V ISA implementation.

These results show a low overhead but still represent an increase. Yet this increase is small in comparison to the execution acceleration observed by using these instructions. Indeed, this non-protected accelerated execution is over 8 times faster, and up to 125 times faster. The smallest acceleration is for PRINCE with the S-Box reconfiguration and the largest acceleration is for PRINCE with a second **SBOX_C** to avoid reconfiguration. The results for the protected acceleration about half of those for non-protected executions. Most LBC use bit or nibble manipulation in some way, which in software is extremely costly, picking a single bit out requires multiple instructions. Even without agility considerations, such high numbers come to show just how much this type of acceleration is needed when using software to handle LBCs.

If a standard LBC doesn't emerge in the automotive industry, it would therefore still be possible to greatly accelerate the execution of many if not any cipher for a very low cost.

6.2 Results of an Agile Protection Against SCA

When looking at the protection against SCA, the area resource are less of a consideration as protection has some incompressible costs. Therefore, this study on agile protection focused on showing that protection against SCA does not need to be cipher-specific. This study showed that by protecting the most sensitive leakage choke point, the S-Box, the exact same protection will work for all the ciphers studied here, and most likely any cipher using a 4×4 S-Box.

CPA attacks were carried out to evaluate the security of such an agile protection. On the one hand, results showed that without the protection, ciphers would break with a few thousand power traces. This means that a physical attack could be realistically used to recover the hidden key on those ciphers. On the other hand, an agile RSM protection was applied to those ciphers' execution and showed an important increase in the resilience to CPA attacks. Indeed, the experimental results showed that the Guessing Entropy (GE) no longer decreased to 1 but

instead seemed to converge towards a random value. The amount of power traces needed to recover the key was greatly increased for this second set of attacks, up to 2 million power traces.

The fact that the GE remained constant even after so many traces means that this study was far from reaching the limits of this protection. It could even be possible that the ciphers stay protected no matter how many power traces are used. What these results also show is that the exact same protection can work to protect very different ciphers. Although the evaluation was done on four of the ciphers, it is reasonable to think that this same protection would work on any cipher using a 4×4 S-Box.

6.3 Perspectives

Although this work provided a sturdy basis for a RISC-V extension dedicated to Lightweight Block Ciphers and their protection, there are some problematics that were not explored.

First off, a few questions remain pending:

- Why are some nibbles less sensitive to the attack in non-protected versions ?
- Why do unrolled execution show a higher resilience to SCA ?
- In some protected implementations, some nibble had a GE of 1, could this be exploited for a different type of attack ?
- How many traces are required to break the protected version, or is there even a number ?

Some solutions were proposed but not tested.

On the one hand, this work has also shown that agile implementations often require using cipher-specific functions. Other works such as the RISC-V Cryptographic Extension Proposal [94] focuses on accelerating the standardized AES. On the other hand, most of the acceleration is observed when nibble or bit manipulation is involved. Therefore, an argument could be made that such part of this extension is too specific and could be replaced by a much more generic bit- or nibble-manipulation extension. For instance, some instructions from the Standard Extension for Bit Manipulation [45] that was added recently could be used to replace the cipher-specific instructions. This approach was indeed used in the extension proposed in [49] to accelerate the execution of the AES with great results.

Regarding the protected implementation, the evaluation could be extended to find how many traces are required to break the key. This would give a much more precise answer to how often the secret key used should be changed. Indeed, this information is important because changing the key can be a time-consuming operation since each round key has to be pre-calculated, particularly when protected since the entire S-Box configuration has to be changed. Finding the limit would allow to reduce the overall latency.

Another way to reduce this latency is to accelerate the calculation of round keys. Indeed, we established that key schedule was out of bound for this study, yet this process can be time and energy consuming. Therefore, accelerating this process, ideally in an agile way could offer latency gains.

Finally, the RSM protection we used was designed to work against first order attacks, and was not evaluated against higher order attacks. The agility proposed in this work could be extended to higher order protection. Both an agility in terms of the spectrum of ciphers but also in terms of level of protection. Indeed, the order of protection could be used as a parameter to have an extension that would adapt to the security needs.

6.4 List of Scientific Productions

In this section, we present the list of publications and presentations given during this thesis. There were four publications:

[82] Etienne Tehrani, Jean-Luc Danger, and Tarik Graba. Generic architecture for lightweight block ciphers: A first step towards agile implementation of multiple ciphers. In *IFIP International Conference on Information Security Theory and Practice*, pages 28–43. Springer, 2018

[85] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, Sylvain Guilley, and Jean-Luc Danger. Classification of lightweight block ciphers for specific processor accelerated implementations. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 747–750. IEEE, 2019

[83] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. Risc-v extension for lightweight cryptography. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 222–228. IEEE, 2020

[84] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. Rsm protection of the present lightweight cipher as a risc-v extension. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 325–332. IEEE, 2021

Along with two workshop presentations:

- A talk about "Acceleration of Lightweight Cryptography on Microprocessors" in 2019 at the *Cryptachi* workshop
- A talk about "RISC-V Extension for Lightweight Cryptography" in 2019 at a *RISC-V Foundation Cryptographic Extension Task Group* meeting

Bibliography

[1] Cw1173 chipwhisperer-lite. https://rtfm.newae.com/Capture/ChipWhisperer-Lite/. Accessed: 2021-03.

[2] Cw305 artix fpga target. https://rtfm.newae.com/Targets/CW30520Artix20FPGA/. Accessed: 2021-03.

[3] A fpga friendly 32 bit risc-v cpu implementation. https://github.com/SpinalHDL/VexRiscv. Accessed: 2020-04.

[4] Risc-v bitmanip (bit manipulation) extension. https://github.com/riscv/riscv-bitmanip. Accessed: 2021-04, Version: 0.94-draft.

[5] Spinalhdl hardware description language. https://github.com/SpinalHDL/SpinalHDL. Accessed: 2020-04.

[6] Alibaba. Alibaba website. https://www.alibaba.com/. 2022-04-02.

[7] CHIPS Alliance. Rocket chip github. https://github.com/chipsalliance/rocket-chip. 2022-03-15.

[8] Subhadeep Banik, Andrey Bogdanov, Takanori Isobe, Kyoji Shibutani, Harunaga Hiwatari, Toru Akishita, and Francesco Regazzoni. Midori: A block cipher for low energy. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 411–436. Springer, 2014.

[9] Subhadeep Banik, Sumit Kumar Pandey, Thomas Peyrin, Yu Sasaki, Siang Meng Sim, and Yosuke Todo. Gift: a small present. In *International Conference on Cryptographic Hardware and Embedded Systems*, pages 321–345. Springer, 2017.

[10] Elaine Barker and Quynh Dang. Nist special publication 800-57 part 1, revision 4. *NIST, Tech. Rep*, 16, 2016.

[11] Elaine Barker and Nicky Mouha. Recommendation for the triple data encryption algorithm (tdea) block cipher. Technical report, National Institute of Standards and Technology, 2017.

[12] Ray Beaulieu, Stefan Treatman-Clark, Douglas Shors, Bryan Weeks, Jason Smith, and Louis Wingers. The simon and speck lightweight block ciphers. In *Design Automation Conference (DAC), 2015 52nd ACM/EDAC/IEEE*, pages 1–6. IEEE, 2015.

[13] Christof Beierle, Alex Biryukov, Luan Cardoso dos Santos, Johann Großschädl, Léo Perrin, Aleksei Udovenko, Vesselin Velichkov, Qingju Wang, and Alex Biryukov. Schwaemm and esch: lightweight authenticated encryption and hashing using the sparkle permutation family. *NIST round*, 2, 2019.

[14] Christof Beierle, Jérémy Jean, Stefan Kölbl, Gregor Leander, Amir Moradi, Thomas Peyrin, Yu Sasaki, Pascal Sasdrich, and Siang Meng Sim. The skinny family of block ciphers and its low-latency variant mantis. In *Annual Cryptology Conference*, pages 123–153. Springer, 2016.

[15] Mihir Bellare and Phillip Rogaway. Introduction to modern cryptography. *Ucsd Cse*, 207:207, 2005.

[16] S. Bhasin, J. Danger, S. Guilley, and Z. Najm. Nicv: Normalized inter-class variance for detection of side-channel leakage. In *2014 International Symposium on Electromagnetic Compatibility, Tokyo*, pages 310–313, 2014.

[17] Shivam Bhasin, Jean-Luc Danger, Sylvain Guilley, and Zakaria Najm. Nicv: normalized inter-class variance for detection of side-channel leakage. In *2014 International Symposium on Electromagnetic Compatibility, Tokyo*, pages 310–313. IEEE, 2014.

[18] Andrey Bogdanov, Lars R Knudsen, Gregor Leander, Christof Paar, Axel Poschmann, Matthew JB Robshaw, Yannick Seurin, and Charlotte Vikkelsoe. Present: An ultra-lightweight block cipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 450–466. Springer, 2007.

[19] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. ryptology ePrint Archive, Report 2012/529, 2012. `https://eprint.iacr.org/2012/529.pdf`.

[20] Julia Borghoff, Anne Canteaut, Tim Güneysu, Elif Bilge Kavun, Miroslav Knezevic, Lars R Knudsen, Gregor Leander, Ventzislav Nikov, Christof Paar, Christian Rechberger, et al. Prince–a low-latency block cipher for pervasive computing applications. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 208–225. Springer, 2012.

[21] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[22] Suresh Chari, Josyula R Rao, and Pankaj Rohatgi. Template attacks. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 13–28. Springer, 2002.

[23] Lily Chen et al. Recommendation for key derivation through extraction-then-expansion. *NIST Special Publication*, 800:56C, 2011.

[24] Zhimin Chen and Yujie Zhou. Dual-rail random switching logic: a countermeasure to reduce side channel leakage. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 242–254. Springer, 2006.

[25] chisel/firrtl. chisel/firrtl hardware complier framework. `https://www.chisel-lang.org/`. 2022-03-24.

[26] Joan Daemen, Michaël Peeters, Gilles Van Assche, and Vincent Rijmen. Nessie proposal: Noekeon. In *First Open NESSIE Workshop*, pages 213–230, 2000.

[27] CE Dobraunig, Maria Eichlseder, Stefan Mangard, Florian Mendel, BJM Mennink, Robert Primas, and Thomas Unterluggauer. Isap v2. 0. 2020.

[28] Morris Dworkin. Recommendation for block cipher modes of operation. methods and techniques. Technical report, National Inst of Standards and Technology Gaithersburg MD Computer security Div, 2001.

[29] Morris J Dworkin, Elaine B Barker, James R Nechvatal, James Foti, Lawrence E Bassham, E Roback, James F Dray Jr, et al. Advanced encryption standard (aes). 2001.

[30] Scott Fluhrer, Itsik Mantin, and Adi Shamir. Weaknesses in the key scheduling algorithm of rc4. In *International Workshop on Selected Areas in Cryptography*, pages 1–24. Springer, 2001.

[31] RISC-V Foundation. Risc-v softcpu contest winners demonstrate cutting-edge risc-v implementations for fpgas. `https://riscv.org/announcements/2018/12/risc-v-softcpu-contest-winners-demonstrate-cutting-edge-risc-v-implementations-for-fpgas/`. 2022-04-07.

[32] Benedikt Gierlichs, Lejla Batina, Pim Tuyls, and Bart Preneel. Mutual information analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 426–442. Springer, 2008.

[33] L Rodney Goke and G Jack Lipovski. Banyan networks for partitioning multiprocessor systems. In *ACM SIGARCH Computer Architecture News*, volume 2, pages 21–28. ACM, 1973.

[34] OpenHW Group. Openhw group core-v cv32e40p risc-v ipi github. `https://github.com/openhwgroup/cv32e40p`. 2022-04-02.

[35] OpenHW Group. Openhw group cva6 risc-v cpu github. `https://github.com/openhwgroup/cva6`. 2022-04-02.

[36] OpenHW Group. Openhw group website. `https://www.openhwgroup.org/`. 2022-04-02.

[37] Sylvain Guilley, Laurent Sauvage, Florent Flament, Vinh-Nga Vong, Philippe Hoogvorst, and Renaud Pacalet. Evaluation of power constant dual-rail logics countermeasures against dpa with design time security metrics. *IEEE Transactions on Computers*, 59(9):1250–1263, 2010.

[38] Jian Guo, Thomas Peyrin, Axel Poschmann, and Matt Robshaw. The led block cipher. In *Proceedings of the 13th International Conference on Cryptographic Hardware and Embedded Systems*, CHES'11, pages 326–341, Berlin, Heidelberg, 2011. Springer-Verlag.

[39] Road vehicles – Controller area network (CAN) – Part 1: Data link layer and physical signalling. Standard, International Organization for Standardization, Geneva, CH, December 2015.

[40] ISO/IEC 29192-2:2012. a) PRESENT: a lightweight block cipher with a block size of 64 bits and a key size of 80 or 128 bits; Information technology – Security techniques – Lightweight cryptography – Part 2: Block ciphers.

[41] Jérémy Jean, Ivica Nikolić, and Thomas Peyrin. Tweaks and keys for block ciphers: the tweakey framework. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 274–288. Springer, 2014.

[42] Najeh Kamoun, Lilian Bossuet, and Adel Ghazel. Correlated power noise generator as a low cost dpa countermeasures to secure hardware aes cipher. In *2009 3rd International Conference on Signals, Circuits and Systems (SCS)*, pages 1–6. IEEE, 2009.

[43] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[44] Paul C Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Annual International Cryptology Conference*, pages 104–113. Springer, 1996.

[45] Bastian Koppelmann, Peer Adelt, Wolfgang Mueller, and Christoph Scheytt. Risc-v extensions for bit manipulation instructions. In *2019 29th International Symposium on Power and Timing Modeling, Optimization and Simulation (PATMOS)*, pages 41–48. IEEE, 2019.

[46] Arjen K Lenstra and Eric R Verheul. Selecting cryptographic key sizes. *Journal of cryptology*, 14(4):255–293, 2001.

[47] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks: Revealing the secrets of smart cards*, volume 31. Springer Science & Business Media, 2008.

[48] Charalampos Manifavas, George Hatzivasilis, Konstantinos Fysarakis, and Yannis Papaefstathiou. A survey of lightweight stream ciphers for embedded systems. *Security and Communication Networks*, 9(10):1226–1246, 2016.

[49] Ben Marshall, G Richard Newell, Dan Page, Markku-Juhani O Saarinen, and Claire Wolf. The design of scalar aes instruction set extensions for risc-v. *Cryptology ePrint Archive*, 2020.

[50] James L Massey. Guessing and entropy. In *Proceedings of 1994 IEEE International Symposium on Information Theory*, page 204. IEEE, 1994.

[51] Kerry McKay, Lawrence Bassham, Meltem Sönmez Turan, and Nicky Mouha. Report on lightweight cryptography. Technical report, National Institute of Standards and Technology, 2016.

[52] Bart Mennink. Elephant v2. 2021.

[53] Maxime Nassar, Youssef Souissi, Sylvain Guilley, and Jean-Luc Danger. RSM: A small and fast countermeasure for AES, secure against 1st and 2nd-order zero-offset SCAs. In *2012 Design, Automation & Test in Europe Conference & Exhibition, DATE 2012, Dresden, Germany, March 12-16, 2012*, pages 1173–1178, 2012.

[54] Roger M Needham and David J Wheeler. Tea extensions. *Report, Cambridge University, Cambridge, UK*, October 1997.

[55] Svetla Nikova, Christian Rechberger, and Vincent Rijmen. Threshold implementations against side-channel attacks and glitches. In *International conference on information and communications security*, pages 529–545. Springer, 2006.

[56] Svetla Nikova, Vincent Rijmen, and Martin Schläffer. Secure hardware implementation of nonlinear functions in the presence of glitches. *Journal of Cryptology*, 24(2):292–321, 2011.

[57] NIST. Lightweight cryptography standardization: Nist announces finalists. `https://csrc.nist.gov/Projects/lightweight-cryptography/finalists`. 2022-02-02.

[58] Univeristy of Berkeley. Berkeley university website. `https://www.berkeley.edu/`. 2022-03-24.

[59] National Institute of Standards and Technology. Nist: National institute of standards ans technology. `https://www.nist.gov/`. 2022-04-11.

[60] Telecom Paris. Chaire connected cars and cybersecurity. `https://chairec3s.wp.imt.fr/`. 2022-02-13.

[61] Axel Poschmann, San Ling, and Huaxiong Wang. 256 bit standardized crypto for 650 GE–GOST revisited. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 219–233. Springer, 2010.

[62] NIST FIPS PUB. 46-3. data encryption standard. *Federal Information Processing Standards, National Bureau of Standards, US Department of Commerce*, 1977.

[63] PULP. Pulp website. https://pulp-platform.org/index.html. 2022-04-02.

[64] Vincent Rijmen and Joan Daemen. The design of rijndael: Aes. *The Advanced Encryption Standard. Springer, Berlin*, 2002.

[65] RISC-V. Risc-v international members. https://riscv.org/members/. 2022-03-24.

[66] RISC-V. Risc-v: The free and open risc instruction set architecture. https://riscv.org/. 2022-03-15.

[67] Matthieu Rivain, Emmanuel Prouff, and Julien Doget. Higher-order masking and shuffling for software implementations of block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 171–188. Springer, 2009.

[68] Ronald L Rivest. The RC5 encryption algorithm. In *International Workshop on Fast Software Encryption*, pages 86–96. Springer, 1994.

[69] Rainer A Rueppel. Stream ciphers. In *Analysis and Design of Stream Ciphers*, pages 5–16. Springer, 1986.

[70] Markku-Juhani O Saarinen, G Richard Newell, and Ben Marshall. Building a modern trng: an entropy source interface for risc-v. In *Proceedings of the 4th ACM Workshop on Attacks and Solutions in Hardware Security*, pages 93–102, 2020.

[71] Werner Schindler, Kerstin Lemke, and Christof Paar. A stochastic model for differential side channel cryptanalysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 30–46. Springer, 2005.

[72] Bruce Schneier. The legacy of des. https://www.schneier.com/blog/archives/2004/10/the_legacy_of_d.html. 2004-10-06.

[73] Kyoji Shibutani, Takanori Isobe, Harunaga Hiwatari, Atsushi Mitsuda, Toru Akishita, and Taizo Shirai. Piccolo: an ultra-lightweight blockcipher. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 342–357. Springer, 2011.

[74] SiFive. Sifive website. https://www.sifive.com/. 2022-03-24.

[75] Data Encryption Standard et al. Data encryption standard. *Federal Information Processing Standards Publication*, 112, 1999.

[76] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[77] Tomoyasu Suzaki, Kazuhiko Minematsu, Sumio Morioka, and Eita Kobayashi. Twine: A lightweight block cipher for multiple platforms. In *Selected Areas in Cryptography*, page 339. Springer.

[78] Syntacore. Scr7 high-performance 64bit application core. https://syntacore.com/page/products/processor-ip/scr7. 2022-03-15.

[79] Syntacore. Syntacore custom cores and tools. https://syntacore.com/. 2022-04-10.

[80] Syntacore. Syntacore scr1 github. https://github.com/syntacore/scr1. 2022-04-10.

[81] Alibaba T-Head. T-head website. https://www.t-head.cn/. 2022-04-02.

[82] Etienne Tehrani, Jean-Luc Danger, and Tarik Graba. Generic architecture for lightweight block ciphers: A first step towards agile implementation of multiple ciphers. In *IFIP International Conference on Information Security Theory and Practice*, pages 28–43. Springer, 2018.

[83] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. Risc-v extension for lightweight cryptography. In *2020 23rd Euromicro Conference on Digital System Design (DSD)*, pages 222–228. IEEE, 2020.

[84] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. Rsm protection of the present lightweight cipher as a risc-v extension. In *2021 24th Euromicro Conference on Digital System Design (DSD)*, pages 325–332. IEEE, 2021.

[85] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, Sylvain Guilley, and Jean-Luc Danger. Classification of lightweight block ciphers for specific processor accelerated implementations. In *2019 26th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 747–750. IEEE, 2019.

[86] Stefan Tillich, Christoph Herbst, and Stefan Mangard. Protecting aes software implementations on 32-bit processors against power analysis. In *International Conference on Applied Cryptography and Network Security*, pages 141–157. Springer, 2007.

[87] Michael Tunstall and Gilbert Goodwill. Applying tvla to public key cryptographic algorithms. *IACR Cryptol. ePrint Arch.*, 2016:513, 2016.

[88] Meltem Sönmez Turan, Kerry McKay, Donghoon Chang, Cgdas a Calık, Lawrence Bassham, Jinkeon Kang, and John Kelsey. Status report on the second round of the nist lightweight cryptography standardization process. Technical report, Tech. rep. https://doi. org/10.6028/NIST. IR. 8369. Gaithersburg, MD, USA . . . , 2021.

[89] Meltem Sönmez Turan, Kerry A McKay, Çagdas Çalik, Donghoon Chang, Lawrence Bassham, et al. Status report on the first round of the nist lightweight cryptography standardization process. *National Institute of Standards and Technology, Gaithersburg, MD, NIST Interagency/Internal Rep.(NISTIR)*, 2019.

[90] Nicolas Veyrat-Charvillon and François-Xavier Standaert. Mutual information analysis: how, when and why? In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 429–443. Springer, 2009.

[91] Andrew Waterman, Yunsup Lee, David A. Patterson, Krste Asanovic, Volume I User level Isa, Andrew Waterman, Yunsup Lee, and David Patterson. The risc-v instruction set manual, 2014.

[92] David J Wheeler and Roger M Needham. Tea, a tiny encryption algorithm. In *International workshop on fast software encryption*, pages 363–366. Springer, 1994.

[93] Gangqiang Yang, Bo Zhu, Valentin Suder, Mark D Aagaard, and Guang Gong. The simeck family of lightweight block ciphers. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 307–329. Springer, 2015.

[94] Alexander Zeh, Andy Glew, Barry Spinney, Ben Marshall, Daniel Page, Derek Atkins, Ken Dockser, Markku-Juhani O Saarinen, Nathan Menhorn, and Richard Newell. Risc-v cryptographic extension proposals. 2021.

[95] Wentao Zhang, Zhenzhen Bao, Dongdai Lin, Vincent Rijmen, Bohan Yang, and Ingrid Verbauwhede. Rectangle: a bit-slice lightweight block cipher suitable for multiple platforms. *Science China Information Sciences*, 58(12):1–15, 2015.

Appendix A

Lightweight Block Ciphers

Throughout this work we studied many Lightweight Block Ciphers, which all work in different ways. This chapter describes each of these ciphers in detail.

NOEKEON

NOEKEON [26] is an SPN cipher with a 128-bit key size, a 128-bit block size and 16 rounds. Its state is manipulated as a one-dimension array of 8 bytes numbered 0 through 15.

Let,

- *Nr* be the number of rounds, which is 16,
- *Roundct* be the round constants, with the shape $Roundct[i] = (\prime 00\prime, \prime 00\prime, \prime 00\prime, RC[i])$ and RC[i] the round constant of round i,
- *State* be a pointer of the state, divided into four 32-bit words a[0], a[1], a[2] and a[3],
- *Gamma* be a non-linear function (*Cf.* Alg. 7) equivalent to an S-Box (*Cf.* Tab. A.1), which corresponds to the Confusion Step,
- *Theta* (*Cf.* Alg. 8) be a linear function, which corresponds to the Key Addition Step,
- *Pi1* (*Cf.* Alg. 9) and *Pi2* (*Cf.* Alg. 10) be linear functions, which correspond to the Diffusion Step.

The NOEKEON encryption algorithm in pseudocode notation is shown in Alg. 4.

Its decryption algorithm in pseudocode notation is shown in Alg. 5.

Its round function in pseudocode notation is shown in Alg. 6.

Its *Gamma* function in pseudocode notation is shown in Alg. 7.

The *Gamma* function corresponds to sixteen 4×4 S-Boxes in parallel, for which the substitution table is shown in Tab. A.1.

The *Theta* function in pseudocode notation is shown in Alg. 8.

Algorithm 4 NOEKEON Encryption Algorithm

NOEKEON ENCRYPTION
function NOEKEON($WorkingKey, State$)
  for $i = 0; i < nr; i++$ **do**
    $Round(WorkingKey, State, Roundct[i], 0)$
  end for
  $State[0] \leftarrow State[0] \oplus Rounct[nr]$
  $Theta(WorkingKey, State)$
end function

Algorithm 5 NOEKEON Encryption Algorithm

NOEKEON DECRYPTION
function INVERSENOEKEON($WorkingKey, State$)
  $Theta(NullVector, WorkingKey)$
  for $i = Nr; i > 0; i--$ **do**
    $Round(WorkingKey, State, 0, Roundct[i])$
  end for
  $Theta(WorkingKey, State)$
  $State[0] \leftarrow State[0] \oplus Rounct[0]$
end function

Algorithm 6 NOEKEON Round function in pseudocode notation

function ROUND($Key, State, Constant1, Constant2$)
  $State[0] \leftarrow State[0] \oplus Constant1$
  $Theta(Key, State)$
  $State[0] \leftarrow State[0] \oplus Constant2$
  $Pi1(State)$
  $Gamma(State)$
  $Pi2(State)$
end function

The two rotation functions *Pi1* and *Pi2* in pseudocode notation are shown in Alg. 9 and Alg. 10.

The NOEKEON cipher differs from the most common LBCs by the fact that:

- It can only use 128-bit block size.

Algorithm 7 The *Gamma* Function

function GAMMA(a)

$a[1] \leftarrow a[1] \oplus (\neg a[3] \odot \neg a[2])$

$a[0] \leftarrow a[0] \oplus (a[2] \odot a[1])$

$tmp \leftarrow a[3]; a[3] \leftarrow a[0]; a[0] \leftarrow tmp$

$a[2] \leftarrow a[2] \oplus a[0] \oplus a[1] \oplus a[3]$

$a[1] \leftarrow a[1] \oplus (\neg a[3] \odot \neg a[2])$

$a[0] \leftarrow a[0] \oplus (a[2] \odot a[1])$

end function

Table A.1: The S-Box Table of the *Gamma* Function of NOEKEON

$a_1a_2a_3a_4$	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$Gamma(a_1a_2a_3a_4)$	7	A	2	C	4	8	F	0	5	9	1	E	3	D	B	6

Algorithm 8 The *Theta* Function

function THETA(k, a)

$tmp \leftarrow a[0] \oplus a[2]$

$tmp \leftarrow tmp \oplus (tmp \ggg 8) \oplus (tmp \lll 8)$

$a[1] \leftarrow a[1] \oplus tmp$

$a[3] \leftarrow a[3] \oplus tmp$

$a[0] \leftarrow a[0] \oplus k[0]; a[1] \leftarrow a[1] \oplus k[1]; a[2] \leftarrow a[2] \oplus k[2]; a[3] \leftarrow a[3] \oplus k[3]$

$tmp \leftarrow tmp \oplus a[1] \oplus a[3]$

$tmp \leftarrow tmp \oplus (tmp \ggg 8) \oplus (tmp \lll 8)$

$a[0] \leftarrow a[0] \oplus tmp$

$a[2] \leftarrow a[2] \oplus tmp$

end function

Algorithm 9 The *Pi1* Function

function PI1(a)

$a[1] \leftarrow a[1] \lll 1; a[2] \leftarrow a[2] \lll 5; a[3] \leftarrow a[3] \lll 2$

end function

Algorithm 10 The *Pi2* Function

function PI2(a)

$a[1] \leftarrow a[1] \ggg 1; a[2] \leftarrow a[2] \ggg 5; a[3] \leftarrow a[3] \ggg 2$

end function

Piccolo

Piccolo [73] is a GFN cipher with a 128-bit key size (it also supports 80-bit key size), a 64-bit block size and 33 rounds. It is a GFN type cipher because like a Feistel network it modifies only half the state during each round. Unlike a Feistel type algorithm, it does not divide the state into a right and left part, it divides it in four equal part and modifies the second and forth quarter. Also, unlike a Feistel type algorithm, it does not switch the position of the two halves, it uses a byte-level permutation at the end of each round. Its state is manipulated as a one-dimension array of four 16-bit words numbered 0 through 3.

Let,

- r be the number of rounds, which is 33 for 128-bit key size,
- X be the 64-bit state, it can be divided into four 16-bit words X_0, X_1, X_2, X_3,
- wk_i be the four 16-bit whitening keys,
- rk_i be the $2r$ 16-bit round keys,
- Y be the 64-bit output of the Piccolo function,
- RP be a linear round permutation (*Cf.* Fig. A.1), which corresponds to the Diffusion Step,
- F be a combination of linear and non-linear function (*Cf.* Fig. A.2), which corresponds to a mix between the Diffusion and the Confusion Step.

The Piccolo encryption algorithm in pseudocode notation is shown in Alg. 11.

Algorithm 11 PICCOLO Encryption Algorithm

PICCOLO ENCRYPTION
function PICCOLO($X, wk_0, ..., wk_3, rk_0, ..., rk_{2r-1}$)
  $X_0|X_1|X_2|X_3 \leftarrow X$
  $X_0 \leftarrow X_0 \oplus wk_0; X_2 \leftarrow X_2 \oplus wk_1$
  for $i = 0; i < 2r - 2; i++$ **do**
    $X_1 \leftarrow X_1 \oplus F(X_0) \oplus rk_{2i}; X_3 \leftarrow X_3 \oplus F(X_2) \oplus rk_{2i+1}$
    $X_0|X_1|X_2|X_3 \leftarrow RP(X_0|X_1|X_2|X_3)$
  end for
  $X_1 \leftarrow X_1 \oplus F(X_0) \oplus rk_{2r-2}; X_3 \leftarrow X_3 \oplus F(X_2) \oplus rk_{2r-1}$
  $X_0 \leftarrow X_0 \oplus wk_2; X_2 \leftarrow wk_3$
  $Y \leftarrow X_0|X_1|X_2|X_3$
end function

Its RP function is represented in Fig. A.1.
Its F function is represented in Fig. A.2.
The M function used in F uses the **M** diffusion matrix which is defined as:

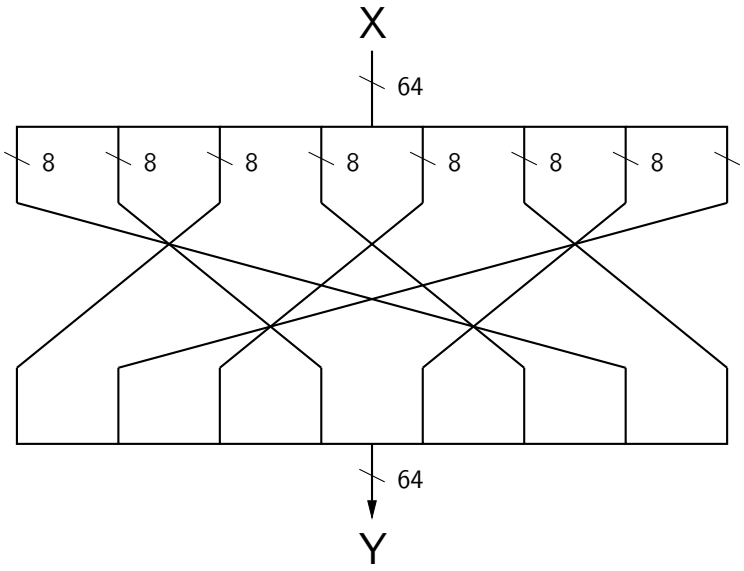


Figure A.1: The Piccolo Round Permutation "*RP*" Function

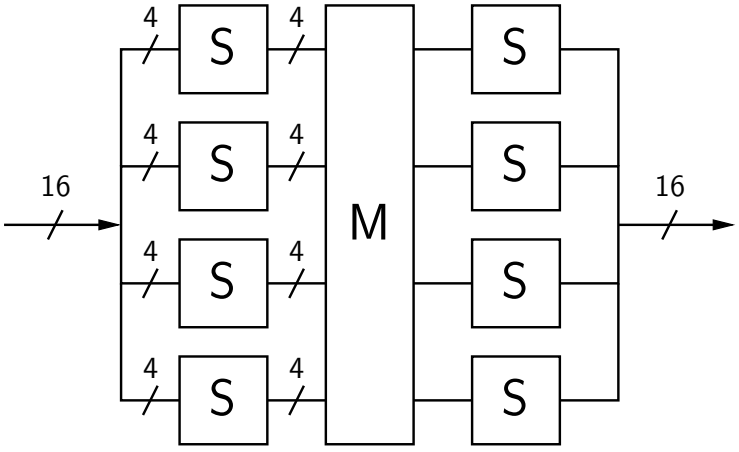


Figure A.2: The Piccolo "*F*" Function

$$\mathbf{M} = \begin{pmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{pmatrix}$$

In the *M* function, it gets multiplied to the 16-bit word sub-state $X_i = (x_{i0}|x_{i1}|x_{i2}|x_{i3})$ as such:

$$^t(x_{i0}, x_{i1}, x_{i2}, x_{i3}) \leftarrow \mathbf{M} \cdot^t (x_{i0}, x_{i1}, x_{i2}, x_{i3})$$

In the *F* function, *S* a 4×4 S-Box function, for which the substitution table is shown in Tab. A.2.

The Piccolo cipher differs from the most common LBCs by a few aspects:

Table A.2: The S-Box Table of the *F* Function of Piccolo

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

- It uses two layers of Confusion per round.
- It uses two layers of Diffusion, which cannot be merged since they are separated by a Confusion Step.
- The Matrix it uses is not just 1s and 0s.

LED

LED [38] is an SPN cipher with a 128-bit key size, a 64-bit block and 48 rounds. This cipher does not use Key Scheduling. The state is manipulated as a 4×4 nibble matrix. The structure of its rounds is very similar to that of AES, expect that it does the key addition only once every four rounds. Another main difference with AES is the fact that the 4×4 state matrix is arranged in lines rather than columns. If m is the state and $m_0|m_1|...|m_{14}|m_{15}$ its 16 4-bit nibbles, then the matrix is:

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

Let,

- K_1 and K_2 be the two half of the Key *K*, such that $K = K_1|K_2$,
- *addRoundKey* be a XOR between two 4×4 nibble matrix, which corresponds to the Key Addition Step,
- *AddConstants* be a function that XORs the state to a constant matrix,
- *SubCells* be a non-linear S-Box function, which corresponds to the Confusion Step,
- *ShiftRows* and *MixColumn* be two linear functions, which correspond to the Diffusion Step,
- *Round* be the succession of *AddConstants*, *SubCells*, *ShiftRows* and *MixColumn*.
- *step* be the succession of four *Round()* function.

The LED encryption algorithm in pseudocode notation is shown in Alg. 12.
The LED *Round* function in pseudocode notation is shown in Alg. 13.

Algorithm 12 LED Encryption Algorithm

LED ENCRYPTION
function LED($state, K_1, K_2$)
  for $i = 0, i < 6, i++$ **do**
    $addRoundKey(state, K_1)$
    $step(state)$
    $addRoundKey(state, K_2)$
    $step(state)$
  end for
  $addRoundKey(state, K_1)$
end function

Algorithm 13 LED Round Algorithm

function ROUND($state$)
  $AddConstants(state)$
  $SubCells(state)$
  $ShiftRows(state)$
  $MixColumn(state)$
end function

The *AddConstants* function XORs the state to the following constant 4×4 nibble matrix:

$$\begin{bmatrix} 0 & (rc_5|rc_4|rc_3) & 0 & 0 \\ 1 & (rc_2|rc_1|rc_0) & 0 & 0 \\ 2 & (rc_5|rc_4|rc_3) & 0 & 0 \\ 3 & (rc_2|rc_1|rc_0) & 0 & 0 \end{bmatrix}$$

Where $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$ are six bits which are updated by shifting them one position to the left and changing the rc_0 bit to $rc_5 \oplus rc_4 \oplus 1$. Theses six bits are initialized at zero.

The *SubCells* function is sixteen 4×4 S-Boxes in parallel, for which the substitution table is shown in Table A.3. To note, the S-Box used is the PRESENT

Table A.3: The S-Box Table of LED

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

S-Box (*Cf.* Tab. A.6).

The *ShiftRows* function in pseudocode is shown in Alg. 14. In this algorithm, L_i corresponds to the i^{th} line of the state matrix, numbered 0 through 3. The shift amount is given in nibbles.

The *MixColumn* function multiplies each column of the state with the corresponding column of a matrix **M**. This **M** matrix can be seen as four applications in a row of a matrix **A**, which more hardware friendly than **M**.

Algorithm 14 LED ShiftRows Algorithm

function SHIFTROWS($state$)
$L_1 \leftarrow L_1 \lll 1$
$L_2 \leftarrow L_2 \lll 2$
$L_3 \leftarrow L_3 \lll 3$
end function

$$(\mathbf{A})^4 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 4 & 1 & 2 & 2 \end{pmatrix}^4 = \begin{pmatrix} 4 & 1 & 2 & 2 \\ 8 & 6 & 5 & 6 \\ B & E & A & 9 \\ 2 & 2 & F & B \end{pmatrix} = \mathbf{M}$$

The LED cipher differs from the most common LBCs by a few aspects:

- It doesn't use Key Scheduling.
- It uses multiple rounds in a row without adding any secret key information.
- The Matrix it uses is not just 1s and 0s

Simon

Simon [12] is a Feistel cipher with a 128-bit key size, a 64-bit block size (it also supports 128-bit and other block size) and 44 rounds (68 for 128-bit block size). As a Feistel cipher, it manipulates the state as a concatenation of two parts the right and the left parts. All the rounds of this cipher are the same. This algorithm is part of the Simon & Speck family which was submitted by the NSA. As such, the scientific community is reluctant to push the use of these ciphers, due to the history of DES which is explained in Section 2.3.1.

Let,

- x be the left half of the state,
- y be the right half of the state,
- k be the round key,
- f be the function used to modify half the state at each round, which is a combination of rotations and AND instructions,
- R_k be the round function of Simon using the k round key.

The encryption round function of Simon is defined by:

$$R_k(x, y) = (y \oplus f(x) \oplus k, x)$$

where,

$$f(x) = ((x \lll 1) \odot (x \lll 8)) \oplus (x \lll 2)$$

The decryption round function of Simon is defined by:

$$R_k^{-1}(x, y) = (y, x \oplus f(y) \oplus k)$$

The Simon cipher differs from the most common LBCs by the fact that:

- It only uses basic instructions XOR, AND and the Rotation to manipulate the state.

Speck

Speck [12] is a Feistel cipher with a 128-bit key size, a 64-bit block size (it also supports 128-bit and other block size) and 27 rounds (32 for the 128-bit block size). Its specificity is that its round is composed of a set of two Feistel round functions which are slightly different. All the rounds of this cipher are the same.

Let,

- x be the left half of the state,
- y be the right half of the state,
- k be the round key,
- R_k be the round function of Speck using the k round key.

The encryption round function of Speck (for 64-bit block size) is defined by:

$$R_k(x, y) = (((x \ggg 8) + y) \oplus k, (y \lll 3) \oplus ((x \ggg 8) + y) \oplus k)$$

which can be reinterpreted as:

$$(x, y) \mapsto (y, ((x \ggg 8) + y) \oplus k) and, (x, y) \mapsto (y, (x \lll 3) \oplus y)$$

to fit the Feistel framework.

The decryption round function of Speck (for 64-bit block size) is defined by:

$$R_k^{-1}(x, y) = ((((x \oplus k) - ((x \oplus y) \ggg 3)) \lll 8), ((x \oplus y) \ggg 3))$$

The Speck cipher differs from the most common LBCs by a few aspects:

- It only uses basic instructions XOR, AND, ADD and the Rotation to manipulate the state.
- It uses two different round functions when interpreted within the Feistel framework

Simeck

Simeck [93] is a Feistel cipher with a 128-bit key size, a 64-bit block size and 44 rounds. It also supports other smaller key size and block size which are too small to fit modern security standards. All the rounds of this cipher are the same.

Let,

- x be the left half of the state,
- y be the right half of the state,
- k be the round key,
- f be the function used to modify half the state at each round, which is a combination of rotations and AND instructions,
- R_k be the round function of Simeck using the k round key.

The encryption round function of Simeck is defined by:

$$R_k(x, y) = (y \oplus f(x) \oplus k, x)$$

where,

$$f(x) = (x \odot (x \lll 5)) \oplus (x \lll 1)$$

The Simeck cipher differs from the most common LBCs by the fact that:

- It only uses basic instructions XOR, AND, ADD and the Rotation to manipulate the state.

RC5

The RC5 [68] is one of the most unique algorithms studied in this work. It does not fit either the Feistel/GFN or the SPN structure. It is highly configurable, the algorithm is defined for:

- any block size greater than 0, although the recommended values are 32, 64 and 128 bits.
- any key size greater than 0, although the recommended number of key bytes are 0, 1, ..., 255.
- any number of rounds, although the recommended values 0, 1, ..., 255.

The state is manipulated as two w sized words, where w is half the chosen block size.

Let,

- A be the first w-bit half of plaintext/state,
- B be the second w-bit half of the plaintext/state,
- r be the number of rounds,
- E be the key expansion array, it contains $2(r + 1)$ w-bit words.

The RC5 encryption algorithm in pseudocode notation is shown in Alg. 15.
The RC5 decryption algorithm in pseudocode notation is shown in Alg. 16.
The RC5 cipher differs from the most common LBCs by a few aspects:

Algorithm 15 RC5 Encryption Algorithm

RC5 ENCRYPTION
function $\mathrm{RC5}(A, B, E)$
  $A \leftarrow A + E[0];$
  $B \leftarrow B + E[1];$
  for $i = 1, i < r, i++$ **do**
    $A \leftarrow ((A \oplus B) \lll B) + E[2 * i]$
    $B \leftarrow ((B \oplus A) \lll A) + E[2 * i + 1]$
  end for
end function

Algorithm 16 RC5 Decryption Algorithm

RC5 DECRYPTION
function $\mathrm{INVRC5}(A, B, E)$
  for $i = r, i > 1, i--$ **do**
    $B \leftarrow ((B - E[2 * i + 1] \ggg A) \oplus A$
    $A \leftarrow ((A - E[2 * i] \ggg B) \oplus B$
  end for
  $B \leftarrow B - E[1];$
  $A \leftarrow A - E[0];$
end function

- It only uses basic instructions XOR, ADD and the Rotation to manipulate the state.
- Its block size, key size and number of rounds are highly configurable.

XTea

XTea [54] is an extended version of the GFN type Tea [92] cipher. It uses a 64-bit block size, a 128-bit key size and 32 rounds. Other values of block size which use different multiples of 32-bit words are also supported. It manipulates the state as two 32-bit words and the key as four 32-bit words. Its key scheduling is done on the fly and directly part of the encryption/decryption.

Let,

- v be the plaintext/state,
- n be the number of 32-bit words in the plaintext,
- k be the four 32-bit word secret key,
- δ be a constant used for the key scheduling.

The XTea encryption algorithm in pseudocode notation is shown in Alg. 17.
The XTea decryption algorithm in pseudocode notation is shown in Alg. 18.
The XTea cipher differs from the most common LBCs by a few aspects:

- It only uses basic instructions XOR, ADD and the Rotation to manipulate the state.
- Its block size, is configurable.

Algorithm 17 XTea Encryption Algorithm

XTEA ENCRYPTION
function XTEA(v, n, k)
  $z \leftarrow v[n-1], sum \leftarrow 0, \delta \leftarrow$ `0x9e3779b9`
  $q \leftarrow 6 + 52/n$
  while $q - - > 0$ **do**
    $sum \leftarrow sum + \delta$
    $e \leftarrow (sum \gg 2) \odot 3$
    for $p = 0; p < n; p + +$ **do**
      $v[p] \leftarrow v[p] + ((z \ll 4) \oplus (z \gg 5)) + (z \oplus k[(p \odot 3) \oplus e]) + sum$
      $z \leftarrow v[p]$
    end for
  end while
end function

Algorithm 18 XTea Decryption Algorithm

XTEA DECRYPTION
function INVXTEA(v, n, k)
  $\delta \leftarrow$ `0x9e3779b9`
  $q \leftarrow 6 + 52/n$
  $sum \leftarrow q * \delta$
  while $sum! = 0$ **do**
    $e \leftarrow (sum \gg 2) \odot 3$
    for $p = n - 1; p > 0; p - -$ **do**
      $z \leftarrow v[p-1]$
      $v[p] \leftarrow v[p] - ((z \ll 4) \oplus (z \gg 5)) + (z \oplus k[(p \odot 3) \oplus e] + sum$
    end for
    $z \leftarrow v[n-1]$
    $v[0] \leftarrow v[0] - ((z \ll 4) \oplus (z \gg 5)) + (z \oplus k[(p \odot 3) \oplus e] + sum$
    $sum \leftarrow sum - \delta$
  end while
end function

GOST

GOST [61] is a GNF cipher, with a 64-bit block size, a 256-bit key size and 32 rounds. As a Feistel cipher, it manipulates the state as two 32-bit parts.

Let,

- L_i be the left half of the state during the i^{th} round,
- R_i be the right half of the state during the i^{th} round,
- K_i be the round key during the i^{th} round,
- S be the S-Box function. It uses eight 4×4 different S-Boxes,
- G_i be the round function of GOST during the i^{th} round.

The encryption round function of GOST is defined by:

$$G_i(R_i, L_i) \leftarrow (R_i, L_i \oplus (S(K_i + R_i) \ll 11))$$

Only the last round differs as the two halves are not swapped.

The *S* function is the S-Box function of GOST, for which eight different 4×4 S-Boxes are used, $S_1, S_2, ..., S_8$. Their substitution tables are shown in Table A.4.

Table A.4: The S-Box Tables of GOST

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S_1(x)$	4	A	9	2	D	8	0	E	6	B	1	C	7	F	5	3
$S_2(x)$	E	B	4	C	6	D	F	A	2	3	8	1	0	7	5	9
$S_3(x)$	5	8	1	D	A	3	4	2	E	F	C	7	6	0	9	B
$S_4(x)$	7	D	A	1	0	8	9	F	E	4	6	C	B	2	5	3
$S_5(x)$	6	C	7	1	5	F	D	8	4	A	9	E	0	3	B	2
$S_6(x)$	4	B	A	0	7	2	1	D	3	6	8	5	9	C	F	E
$S_7(x)$	D	B	4	1	3	F	5	9	0	A	E	7	6	8	2	C
$S_8(x)$	1	F	D	0	5	7	A	4	9	2	3	E	6	B	8	C

The GOST cipher differs from the most common LBCs by a few aspects:

- It uses an integer addition rather than a XOR as its Key Addition operation.
- It is Feistel type but uses S-Boxes
- It uses eight different S-Boxes for each of the eight nibbles during each round.

Rectangle

Rectangle [95] is an SPN cipher, with a 64-bit block size, a 128-bit key size and 25 rounds. The state is manipulated as a 4×16 bit matrix. If **W** is the state and $w_0|w_1|...|w_{62}|w_{63}$ its 64 bits, then the matrix is:

$$\begin{bmatrix} w_{15} & ... & w_2 & w_1 & w_0 \\ w_{31} & ... & w_{18} & w_{17} & w_{16} \\ w_{47} & ... & w_{34} & w_{33} & w_{32} \\ w_{63} & ... & w_{50} & w_{49} & w_{48} \end{bmatrix}$$

which, for convenience can be renamed to ease comprehension as:

$$\begin{bmatrix} a_{0,15} & ... & a_{0,2} & a_{0,1} & a_{0,0} \\ a_{1,15} & ... & a_{1,2} & a_{1,1} & a_{1,0} \\ a_{2,15} & ... & a_{2,2} & a_{2,1} & a_{2,0} \\ a_{3,15} & ... & a_{3,2} & a_{3,1} & a_{3,0} \end{bmatrix}$$

Let,

- K_1 and K_2 be the two half of the Key *K*, such that $K = K_1|K_2$,
- *addRoundKey* be a XOR between two 4×4 nibble matrix, which corresponds to the Key Addition Step,
- *AddConstants* be a function that XORs the state to a constant matrix,

- *SubCells* be a non-linear S-Box function, which corresponds to the Confusion Step,
- *ShiftRows* and *MixColumn* be two linear functions, which correspond to the Diffusion Step,
- *Round* be the succession of *AddConstants*, *SubCells*, *ShiftRows* and *MixColumn*.
- *step* be the succession of four *Round()* function.

The Rectangle *Round* function in pseudocode notation is shown in Alg. 19. All

Algorithm 19 The Rectangle Round Algorithm

function ROUND($state$)
  $AddRoundKey(state)$
  $SubColumn(state)$
  $ShiftRow(state)$
end function

the rounds are the same except that the last round is followed by an additional AddRoundKey.

The *AddRoundKey* function XORs the state to the round key.

The *SubColumn* function applies an S-Box to each of the sixteen 4-bit column, for which the substitution table is shown in Table A.5. In this table, C_j correspond to the j^{th} column of the state matrix defined by $C_j = a_{3,j}|a_{2,j}|a_{1,j}|a_{0,j}$

Table A.5: The S-Box Table of Rectangle

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

The *ShiftRow* function in pseudocode is shown in Alg. 20, it applies a rotation to each of the matrix's lines. In this algorithm, L_i corresponds to the i^{th} line of the state matrix, defined by $L_i = a_{i,15}|a_{i,14}|...|a_{i,1}|a_{i,0}$. The shift amount is given in bits.

Algorithm 20 Rectangle ShiftRow Algorithm

function SHIFTROWS($state$)
  $L_1 \leftarrow L_1 \lll 1$
  $L_2 \leftarrow L_2 \ggg 12$
  $L_3 \leftarrow L_3 \ggg 13$
end function

The Rectangle cipher differs from the most common LBCs by a few aspects:

- It respects the three step but does not have a MixColumn function
- The rotation of its ShiftRow function is done at bit-level

PRESENT

PRESENT [18] is an SPN cipher, with a 64-bit block size, a 128-bit key size and 32 rounds. It respects the three step model. It uses a bit-level permutation as its Diffusion Step rather than ShiftRow, MixColumn functions. The first 31 rounds are the same, and the last round consists only of a key addition.

Let,

- *state* be the state,
- K_i be the i^{th} round key,
- *addRoundKey* be the XOR between the round key and the state, which corresponds to the Key Addition Step,
- *sBoxLayer* be the S-Box function, which corresponds to the Confusion Step,
- *pLayer* be the bit-level permutation function, which corresponds to the Diffusion Step.

The PRESENT *Round* function in pseudocode notation is shown in Alg. 21.

Algorithm 21 PRESENT Round Algorithm

function ROUND($state, K_i$)
  $addRoundKey(state, K_i)$
  $sBoxLayer(state)$
  $pLayer(state)$
end function

The *addRoundKey* function XORs the round key to the state.

The *sBoxLayer* function is sixteen 4×4 S-Boxes in parallel, for which the substitution table is shown in Table. A.6.

Table A.6: The S-Box Table of PRESENT

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

The *pLayer* function is a bit-level permutation, for which the permutation table is shown in Table. A.7.

Table A.7: The PRESENT Permutation Table

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(x)$	0	16	32	48	1	17	33	49	2	18	34	50	3	19	35	51
x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(x)$	4	20	36	52	5	21	37	53	6	22	38	54	7	23	39	55
x	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(x)$	8	24	40	56	9	25	41	57	10	26	42	58	11	27	43	59
x	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(x)$	12	28	44	60	13	29	45	61	14	30	46	62	15	31	47	63

The PRESENT cipher differs from the most common LBCs by the fact that:

- It uses a bit-level permutation as its Diffusion Step

GIFT

GIFT [9] is an SPN cipher, with a 64-bit block size (it also supports a 128-bit block size), a 128-bit key size and 28 rounds (40 rounds for the 128-bit block size). The GIFT algorithm is strongly inspired by the PRESENT algorithm. It respects the three step model. It uses a bit-level permutation as its Diffusion Step rather than ShiftRow, MixColumn functions. All the rounds of this cipher are the same. All the following information will be given for the GIFT-64 version, with a 64-bit block size.

Let,

- *state* be the state, such that $state = b_{63}b_{62}...b_1b_0$,
- RK_i be the 32-bit i^{th} round key,
- U_i and V_i be vectors such that $RK_i = U_i|V_i = u_{15}...u_0|v_{15}...v_0$,
- *addRoundKey* be the XOR between the round key and some constants to part of the state, which corresponds to the Key Addition Step,
- *sBoxLayer* be the S-Box function, which corresponds to the Confusion Step,
- *pLayer* be the bit-level permutation funtion, which corresponds to the Diffusion Step.

The GIFT *Round* function in pseudocode notation is shown in Alg. 22.

Algorithm 22 GIFT Round Algorithm

```
function ROUND(state, K_i)
    sBoxLayer(state)
    pLayer(state)
    addRoundKey(state, K_i)
end function
```

The *addRoundKey* function XORs the round key to specific bits of the state. The round key is XORed according to the following rule:

$$b_{4j+1} \leftarrow b_{4j+1} \oplus u_j, b_{4j} \leftarrow b_{4j} \oplus v_j, \forall j \in 0, ..., 15$$

It also XORs a 6-bit round constant $C = c_5c_4c_3c_2c_1c_0$ to specific bits of the state.

$$b_{63} \leftarrow b_{63} \oplus 1$$
$$b_{23} \leftarrow b_{23} \oplus c_5$$
$$b_{19} \leftarrow b_{19} \oplus c_4$$
$$b_{15} \leftarrow b_{15} \oplus c_3$$
$$b_{11} \leftarrow b_{11} \oplus c_2$$
$$b_7 \leftarrow b_7 \oplus c_1$$
$$b_3 \leftarrow b_3 \oplus c_0$$

The initial value of the C constant is zero, and it is updated at each round using a 6-bit affine LFSR. The update function is:

$$(c_5, c_4, c_3, c_2, c_1, c_0) \leftarrow (c_4, c_3, c_2, c_1, c_0, c_5 \oplus c_4 \oplus 1)$$

The *sBoxLayer* function is sixteen 4×4 S-Boxes in parallel, for which the substitution table is shown in Table. A.8.

Table A.8: The S-Box Table of GIFT

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

The *pLayer* function is a bit-level permutation, for which the permutation table is shown in Table. A.9.

Table A.9: The GIFT Permutation Table

x	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$P(x)$	0	17	34	51	48	1	18	35	32	49	2	19	16	33	50	3
x	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
$P(x)$	4	21	38	55	52	5	22	39	36	53	6	23	20	37	54	7
x	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
$P(x)$	8	25	42	59	56	9	26	43	40	57	10	27	24	41	58	11
x	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
$P(x)$	12	29	46	63	60	13	30	47	44	61	14	31	28	45	62	15

The GIFT cipher differs from the most common LBCs by the fact that:

- It uses a bit-level permutation as its Diffusion Step

PRINCE

PRINCE [19] is an α-reflective cipher based on the SPN round structure, it uses a 64-bit block size, a 128-bit key size and 10 rounds. The algorithm is such that the number of rounds can also be seen as 11 when considering the middle operations as a round without key addition, or 13 when also considering the whitening keys which are added before and after the rest of the rounds. As an α-reflective algorithm, it has a three part overall structure. It was design with the intent of being

implemented as a fully unrolled cipher. This means that the encryption/decryption should be executed in a single clock cycle. PRINCE does not use key schedule, but instead a key extension. The state is manipulated as a 4×4 nibble matrix and each nibble is arranged in lines, like with LED (*Cf.* A). If m is the state and $m_0|m_1|...|m_{14}|m_{15}$ its 16 4-bit nibbles, then the matrix is:

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

Let,

- $PRINCE_{core}$ be the core algorithm of PRINCE,
- *k* be the secret 128-bit key,
- *state* be the state,
- R_i be the round function of round i,
- SL be the S-Box function, which corresponds to the Confusion Step,
- fM/fM' be the M/M'-Layer, which corresponds to the Diffusion Step,
- SR be the Shift Row function,
- RC be the round constants and RC_i be the round constant for round i.

The key extension of PRINCE is defined as follows: The 128-bit k key

$$k = k_0|k_1$$

is extended to a 192-bit key,

$$(k_0|k_0'|k_1) \leftarrow (k_0|(k_0 \ggg 1) \oplus (k_0 \lll 63)|k_1)$$

The k_0 and k_0' are XORed to the state before and after the $PRINCE_{core}$ as whitening keys. The k_1 subkey is used during each round of the $PRINCE_{core}$ during the Key Addition Step.

The $PRINCE_{core}$ function in pseudocode notation is shown in Alg. 23.

The R_i function in pseudocode notation is shown in Alg. 24.

The R_i^{-1} function in pseudocode notation is shown in Alg. 25.

The SL function is sixteen 4×4 S-Boxes in parallel, for which the substitution table is shown in Table. A.10. The SL^{-1} function is the inverse function of SL, for which the substitution table is shown in Table. A.10.

The fM, fM^{-1} and fM' function each multiply the state to the corresponding 64×64 matrix **M**, $\mathbf{M}^{-1}$ and **M'**. The function fM is defined as:

$$fM = SR \circ fM'$$

Algorithm 23 PRINCE$_{core}$ Algorithm

function $PRINCE_{core}(state, k_1, RC)$
  $state \leftarrow state \oplus k_1 \oplus RC_0$
  for $i = 1; i < 6; i++$ **do**
    $state \leftarrow R_i(state, RC_i, k_1)$
  end for
  $state \leftarrow SL(state)$
  $state \leftarrow fM'(state)$
  $state \leftarrow SL^{-1}(state)$
  for $i = 6; i < 11; i++$ **do**
    $state \leftarrow R_i^{-1}(state, RC_i, k_1)$
  end for
  $state \leftarrow state \oplus RC_1 1 \oplus k_1$
end function

Algorithm 24 PRINCE Round Algorithm

function $R_i(state, RC_i, k_1)$
  $state \leftarrow SL(state)$
  $state \leftarrow fM(state)$
  $state \leftarrow state \oplus RC_i \oplus k_i$
end function

Algorithm 25 PRINCE Inverse Round Algorithm

function $R_i^{-1}(state, RC_i, k_1)$
  $state \leftarrow state \oplus k_i \oplus RC_i$
  $state \leftarrow fM^{-1}(state)$
  $state \leftarrow SL^{-1}(state)$
end function

where the SR function is a permutation function, which will apply a rotation to each line of the Matrix such that:

$$SR \circ \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{pmatrix} \leftarrow \begin{pmatrix} m_0 & m_1 & m_2 & m_3 \\ m_5 & m_6 & m_7 & m_4 \\ m_{10} & m_{11} & m_8 & m_9 \\ m_{15} & m_{12} & m_{13} & m_{14} \end{pmatrix}$$

Table A.10: The S-Box Tables of PRINCE

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

TWINE

TWINE [77] is a GFN cipher, with a 64-bit block size, a 128-bit key size and 36 rounds. Its implementation corresponds to the three step model. It uses nibble level permutation as its Diffusion Step, 8 identical 4×4 S-Boxes as its Confusion Step and a key scheduling to have a different round key at each round. Fig. A.3 represents a round of TWINE. Each round is the same except for the first and last rounds which omits the Diffusion Step.

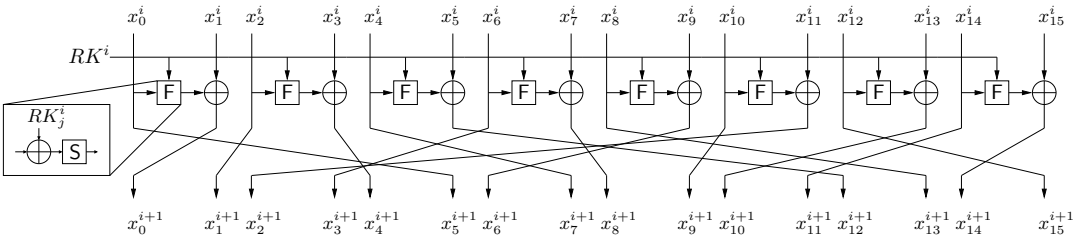


Figure A.3: TWINE Round Function

Let,

- *state* be the state, such that $state = X_0^i|X_1^i|...|X_{14}^i|X_{15}^i$ during the i^{th} round,
- RK^i be the 32-bit i^{th} round key,
- π be the permutation function.

The TWINE *Round* function in pseudocode notation is shown in Alg. 26. where,

Algorithm 26 TWINE Round Algorithm

function ROUND($state, RK^i$)
  for j = 0; j < 8; j++ **do**
    $X_{2j+1}^i \leftarrow S(X_{2j}^i \oplus RK_j^i) \oplus X_{2j+1}^i$
  end for
  for h = 0; h < 15; h++ **do**
    $X_{\pi[h]}^{i+1} \leftarrow X_h^i$
  end for
end function

$S()$ and $\pi[]$ are detailed in Tab. A.11

Table A.11: The S-Box and Permutation Tables of TWINE

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$S(x)$	C	0	F	A	2	B	9	5	8	3	D	7	1	E	6	4
h	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$\pi[h]$	5	0	1	4	7	12	3	8	13	6	9	2	15	10	11	14

SKINNY

SKINNY [14] is an SPN cipher, witha 64-bit block size (it also supports 128-bit block size), a 128-bit key size and 36 rounds (48 rounds for 128-bit block size). Its algorithm respects the three step model. Unlike most ciphers, it uses a *tweakey* framework [41], which modifies the key into a tweakey and lets the user choose which tweakey to have as an input. It uses a nibble-level Diffusion Step, which incorporates a Shift Row and a Mix Column function. It uses 16 identical 4×4 S-Boxes as its Confusion Step. All the rounds of the cipher are the same. The state **S** is manipulated as a 4×4 matrix of nibbles:

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

Let,

- $state$ be the state,
- $sBox$ be the S-Box function, which corresponds to the Confusion Step,
- RTk_i be the tweakey of the i^{th} round, which corresponds to part of the AddKey Step,
- C_i be the constant of the i^{th} round, which corresponds to part of the AddKey Step,
- SR be the Shift Row function, which corresponds to part of the Diffusion Step,
- MxC be the Mix Column function, which corresponds to part of the Diffusion Step.

The SKINNY round function in pseudocode is shown in Alg. 28

Algorithm 27 SKINNY Round Algorithm

function SKINNYROUND($state, RTk_i$)
  $state \leftarrow sBox(state)$
  $state \leftarrow state \oplus C_i \oplus RTk_i$
  $state \leftarrow SR(state)$
  $state \leftarrow MxC(state)$
end function

The $sBox$ function is 16 identical 4×4 S-Boxes in parallel for which the table is shown in Tab. A.12.

Table A.12: The S-Box Table of SKINNY

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

The C_i constant is initialized to zero and updated before use in a given round. It uses a 6-bit LFSR denoted $(rc_5, rc_4, rc_3, rc_2, rc_1, rc_0)$ and updated as follows:

$$(rc_5|rc_4|rc_3|rc_2|rc_1|rc_0) \leftarrow (rc_4|rc_3|rc_2|rc_1|rc_0|rc_5 \oplus rc_4 \oplus 1)$$

Those bits are then rearranged as such:

$$\begin{bmatrix} c_0 & 0 & 0 & 0 \\ c_1 & 0 & 0 & 0 \\ c_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

where $c_2 = 0x2$ and $(c_0, c_1) = (rc_3|rc_2|rc_1|rc_0, 0|0|rc_5|rc_4)$.

The SR function is the same used in AES, its permutation function sets $S_i \leftarrow S_{P[i]}$, with P:

$$P = [0, 1, 2, 3, 7, 4, 5, 6, 10, 11, 8, 9, 13, 14, 15, 12]$$

The MxC function multiplies the state by the following matrix:

$$\begin{pmatrix} 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 \end{pmatrix}$$

Midori

Midori [8] is an SPN cipher, with a 64-bit block size (it also supports 128 bit block size), a 128-bit key size and 16 rounds (20 for the 128-bit block size). It respects the three step model. It uses a nibble-level Diffusion Step, which incorporates a Shift Row and a Mix Column function. It uses 16 identical 4×4 S-Boxes as its Confusion Step. Midori also uses whitening keys before the first round and after the last. All the rounds of the cipher are the same, except for the last which is only the Confusion Step. The state **S** is manipulated as a 4×4 matrix of nibbles:

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

Let,

- $state$ be the state,
- K be the 128-bit secret key, such that $K = K_0|K_1$,
- WK be the whitening key, such that $WK = K_0 \oplus K_1$,
- α_i be the round constant of the i^{th} round,
- RK_i be the round key of the i^{th} round, such that $RK_i = K_{(imod2)} \oplus \alpha_i$,
- $sBox$ be the non-linear S-Box function, which corresponds to the Confusion Step,
- SR and MxC be two linear functions which correspond to the Diffusion Step.

The Midori algorithm in pseudocode notation is shown in Alg. **??**.

Algorithm 28 Midori Algorithm

$state \leftarrow state \oplus WK$
for i = 0; i < 14; i++ **do**
  $state \leftarrow sBox(state)$
  $state \leftarrow SR(state)$
  $state \leftarrow MxC(state)$
  $state \leftarrow state \oplus C_i \oplus RTk_i$
end for
$state \leftarrow sBox(state)$
$state \leftarrow state \oplus KW$

The $sBox$ function is 16 identical 4×4 S-Boxes in parallel for which the table is shown in Tab A.13.

Table A.13: The S-Box Table of Midori

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

The SR function is a permutation, which transforms the state as follows:

$$(s_0, s_1, ..., s_15) \leftarrow (s_0, s10, s_5, s_15, s_14, s_4, s_11, s_1, s_9, s_3, s_12, s_6, s_7, s_13, s_2, s_8)$$

The MxC is a matrix multiplication between four nibbles of the state (x_0, x_1, x_2, x_3) and the involutive binary matrix M:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

such that:

$$^t(x_0, x_1, x_2, x_3) \leftarrow M \cdot^t (x_0, x_1, x_2, x_3)$$

MANTIS

MANTIS [14] is ans $\alpha - reflective$ cipher based on the SPN round structure, it uses a 64-bit block size, a 128-bit key size and 12 rounds (or round assimilated-functions). The algorithm is such that the number of rounds can also be seen as 13 when considering the middle operations as a round without key addition, or 15 when also considering the whitening keys which are added before and after the rest of the rounds. Like PRINCE, described in Section A it has the symmetrical same three parts as any $\alpha - reflective$ algorithm. Also like SKINNY, described in Section A, it uses a *tweakey* framework [41], which modifies the key into a tweakey and lets the user choose which tweakey to have as an input. The state is manipulated as a 4×4 nibble matrix and each nibble is arranged in lines, like with LED (*Cf.* A). If m is the state and $m_0|m_1|...|m_{14}|m_{15}$ its 16 4-bit nibbles, then the matrix is:

$$\begin{bmatrix} m_0 & m_1 & m_2 & m_3 \\ m_4 & m_5 & m_6 & m_7 \\ m_8 & m_9 & m_{10} & m_{11} \\ m_{12} & m_{13} & m_{14} & m_{15} \end{bmatrix}$$

Let,

- $MANTIS_{core}$ be the core algorithm of MANTIS,
- k be the secret 128-bit key, such that $k = k_0|k_1$,
- R_i be the round function of round i,
- RK_i be a round key, it is defined as a XOR between k_1 and a permutation of the tweakey TK,
- RK_i' be a round key, it is defined as a XOR between k_1, a permutation of the tweakey TK and a constant $\alpha =$ `0x243f6a8885a308d3`,
- $state$ be the state,
- RC_i be the round constant used in RK_i and RK_i^{-1},
- $sBox$ be the S-Box function, which corresponds to the Confusion Step,
- PC be the permutation function, which corresponds to part of the Diffusion Step,
- MxC be the MixColumn function, which corresponds to part of the Diffusion Step,
- RC_i be the rounds constant of round i.

Before and after the $MANTIS_{core}$, a whitening key is applied to the state. The whitening key XORed before the rounds is k_0 and the one after the rounds is k_0', defined by:

$$k_0' = (k_0 \ggg 1) \oplus (k_1 \lll 63)$$

These are the same whitening keys used in PRINCE [19].

The tweakey TK is modified after being used at each round by permuting the tweakey used in the round before with h:

$$h = [6, 5, 14, 15, 0, 1, 2, 3, 7, 12, 13, 4, 8, 9, 10, 11]$$

The $MANTIS_{core}$ function in pseudocode notation is shown in Alg. 29.

Algorithm 29 $MANTIS_{core}$ Algorithm

function $MANTIS_{core}(state, RK)$
  $state \leftarrow state \oplus k_1 \oplus TK$
  for $i = 1; i < 7; i++$ **do**
    $state \leftarrow R_i(state, RK_i, RC_i)$
  end for
  $state \leftarrow sBox(state)$
  $state \leftarrow MxC(state)$
  $state \leftarrow sBox(state)$
  for $i = 6; i > 0; i--$ **do**
    $state \leftarrow R_i^{-1}(state, RK_i', RC_i)$
  end for
  $state \leftarrow state \oplus k_1 \oplus TK \oplus \alpha$
end function

The R_i function in pseudocode notation is shown in Alg. 30.

Algorithm 30 MANTIS Round Algorithm

function $R_i(state, RK_i, RC_i)$
  $state \leftarrow sBox(state)$
  $state \leftarrow state \oplus RK_i \oplus RC_i$
  $state \leftarrow PC(state)$
  $state \leftarrow MxC(state)$
end function

The R_i^{-1} function in pseudocode notation is shown in Alg. 31.

Algorithm 31 MANTIS Inverse Round Algorithm

function $R_i^{-1}(state, RK_i', RC_i)$
  $state \leftarrow MxC(state)$
  $state \leftarrow PC^{-1}(state)$
  $state \leftarrow state \oplus RK_i' \oplus RC_i$
  $state \leftarrow sBox(state)$
end function

The $sBox$ function is sixteen involutory 4×4 S-Boxes in parallel, for which the substitution table is shown in Table. A.14, they are the same S-Boxes as in Midori [8]. The fact that the S-Box used is involutory means that the same S-Box function can be used in R_i and R_i^{-1}.

The permutation function PC, permutes the state according to the Midori [8] permutation:

$$P = [0, 11, 6, 13, 10, 1, 12, 7, 5, 14, 3, 8, 15, 4, 9, 2]$$

Table A.14: The S-Box Table of MANTIS

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

The MxC function multiplies each column of the state with the following matrix, once again used by Midori [8]:

$$\begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Appendix B

Configurable S-boxes implementation for Xilinx FPGAs

We present here an optimisation of the S-Boxes implementation for Xilinx FPGA targets.

For the configurable S-Box instruction we had the following constraints:

- the S-Box lookup table must be dynamically reconfigurable,
- when the input of the S-Box changes, its output should be available in the same clock cycle.

For ASIC targets, this can be achieved using Latches to store the configuration (the lookup table values) and a combinatorial multiplexor tree to select the output value.

For FPGA targets, it is not that simple, especially if we want to keep hardware resources usage low. This is important, because used hardware resources usage is directly related to the power consumption, and thus the visible power activity of the S-Box instruction.

For instance, a straightforward RTL implementation of a configurable 4×4 Sbox for an FPGA target would be synthesized as:

- 4×16 flip-flops to store the configuration,
- a certain number of combinatorial logic cells (the exact number will depend on the architecture of the FPGA) for the four $16 \rightarrow 4$ selection multiplexors.

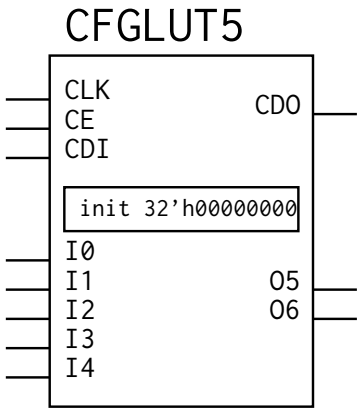
Also, this would use non-negligible amount of routing resources.

Commonly, for the purpose of implementing lookup tables, embedded SRAM block can be used. Those blocks can be used as dual port memories, and we can imagine using one port for configuration and the second port for the S-box lookup table. But this solution could not be used for our purpose as the SRAM blocks in the majority of FPGAs are synchronous. Which would have forced to stall the processor pipeline for one cycle each time the S-Box instruction is computed.

B.1 Xilinx Dynamically Reconfigurable Look-Up Table (LUT) primitives

For Xilinx FPGAs, the CFGLUT5 primitive allows to access, at run time, to the configuration of the combinatorial logic cells. More precisely, this primitive gives us access to a serial interface that allows to dynamically reconfigure a $5 \rightarrow 1$ look-up table LUT.

The following figure shows the interface of the CFGLUT5 primitive.



CLK : configuration clock

CE : configuration enable signal

CDI : configuration input bit

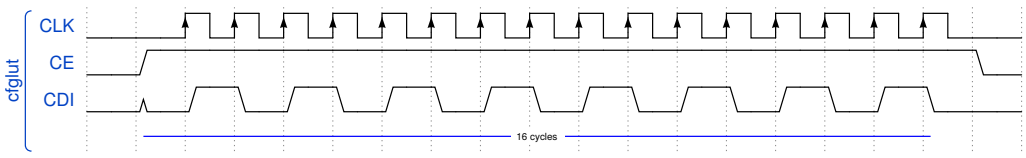
cdo : configuration output bit (when daisychaining cells)

In : the LUT functional inputs

On : the LUT functional outputs

This primitive can be used as either a 5-inputs LUT our a 4-inputs LUT. For the 4 input mode, only the lower 16-bits half of the configuration have to be defined, and the functional output is O5.

The following figure shows an example of configuration sequence for a CFGLUT5 cell for in 4-inputs mode.



The configuration is done sequentially through a serial 1-bit interface. 16 clock cycles are required and the configuration input bit `CDI` must be presented before each rising edge of the configuration clock. In the previous example, the 16-bits of the configuration value are `0xAAAA`.

ECOLE DOCTORALE

Titre: Primitives cryptographiques adaptés aux besoin des véhicules connectés

Mots clés: Cryptographie légère, RISC-V, Attaques par canaux auxiliaires

Résumé: La communication est une des fonctions clés des véhicules à venir, ce qui impose de la protéger. La cryptographie est une façon évidente d'en assurer la sécurité, spécifiquement, la cryptographie légère qui est mieux adaptée aux contraintes de ressources. Il est également essentiel de prendre en compte la résilience aux attaques par canaux auxiliaires sur des systèmes embarqués.
Les objectifs principaux de cette thèse sont d'étudier la possibilité d'implémenter une large variété d'algorithmes de chiffrement légers ainsi que leur protection. Une solution idéale est d'utiliser une implémentation agile, capable d'exécuter différents algorithmes, tout en utilisant un minimum de ressources et en garantissant la sécurité contre les attaques par canaux auxiliaires. Notre principale solution est une extension du jeu d'instruction du RISC-V permettant l'execution de multiples algorithmes tout en satifaisant les contraintes d'agilité.
Nous avons étudier de nombreux algoritmes de chiffrement et avons proposer plusieurs approches. La première est totalement matériel et la seconde et basée sur un processeur dédié afin d'implémenter ces algorithmes de chiffrement légers ainsi que leur protection dans un environement avec de fortes contraintes de ressources.

Title: Cryptographic Primitives Adapted to Connected Car Requirements

Keywords: Lightweight Cryptography, RISC-V, Side-Channel Attacks

Abstract: Communications are one of the key functions in future vehicles and require protection. Cryptography is an obvious answer to secure communications, specifically we studied lightweight cryptography to fit the constrained resources of the environment. A second emerging problem, specific to embedded systems, is resilience to side-channel attacks.
The main objectives of the thesis are to study the feasibility of implementing a wide variety of symmetric lightweight encryption algorithms and their protection. An optimal solution is to have an agile implementation, able to quickly execute different lightweight encryption algorithms, using few resources and guaranteeing protection against physical attacks. Our main architecture starts from a modification of the instruction set of a RISC-V processor to satisfy the agility property of lightweight cryptography algorithms.
We have studied many encryption algorithms and have proposed a first approach with a fully hardware architecture and a second approach with a dedicated processor in order to efficiently implement Lightweight Cryptography and their protection in a constrained embedded system.

Institut Polytechnique de Paris
91120 Palaiseau, France