Architectures Reconfigurables et Cryptographie: Une Analyse de Robustesse et Contremesures Face aux Attaques par Canaux Cachés

Daniel Gomes Mesquita

ACADEMIE DE MONTPELLIER

Universite Montpellier II

– Sciences et Techniques du Languedoc –

THESE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE DE MONTPELLIER II

Discipline : **Microelécronique**

Formation doctorale : **Systèmes Automatiques et Microélectroniques**

Ecole doctorale : **Information, Structures, Systèmes**

présentée et soutenue publiquement

par

Daniel Gomes MESQUITA

Le 06 Novembre 2006

TITRE :

Architectures reconfigurables et cryptographie : une analyse de robustesse et contremesures face aux attaques par canaux cachés

JURY

M. Michel Robert	**Professeur à l'Université Montpellier II**	**Directeur de thèse**
M. Lionel Torres	**Professeur à l'Université Montpellier II**	**co-Directeur de thèse**
M. Fernando Moraes	**Professeur à l'Université Catholique du RGS**	**Examinateur**
M. Didier Demigny	**Professeur à l'Université de Rennes**	**Rapporteur**
M. Viktor Fischer	**Professeur à l'Université de Saint-Etienne**	**Rapporteur**
M. Jean-Claude Bajard	**Professeur à l'Université Montpellier II**	**Président du Jury**
M. Gilles Sassatelli	**Chargé de Recherches au LIRMM**	**Membre Invité**

"Les ordinateurs de modèle courant n'étaient dotés que des capacités limitées parce qu'ils étaient hiérarchisés. Comme dans une monarchie, un microprocesseur central dirigeait des composants électroniques périphériques. Il était donc nécessaire de créer une démocratie au sein même des puces d'ordinateurs. (...) (il) proposait d'utiliser une multitude de petits microprocesseurs qui travailleraient simultanément, se concertaient en permanence et, à tour de rôle, prendraient des décisions. L'engin qu'il appelait de ses vœux, il nommait "*ordinateur à architecture démocratique*"."

(Edmond Wells - L'Encyclopédie du Savoir Relatif et Absolu

***apud* Bernard Werber *in* La révolution des fourmis)**

Je dédie cette thèse à mes familles : celle où je suis né, celle que j'ai choisie, et celle que dépasse les liens de sang.

Remerciements

La seule manière d'être juste envers tous ceux qui m'ont aidé pendant la période de ma thèse est de les remercier à tous, sans oublier personne.

Or, à ce moment à la fois et de joie et de stress, je risque d'oublier de citer quelques uns.

Donc, la seule manière que j'ai trouvé d'être équitable ce de remercier à Dieu de m'avoir permis de croiser avec autant de gens prêtes à m'aider sans demander de retour.

Mon grand merci aux personnes de l'administration et du service technique du LIRMM, aux membres permanents enseignants et chercheurs, aux collègues doctorants, actuels et passés, aux amis brésiliens et français, et à ma famille.

Sans votre soutien je n'aurais pas réussi cette épreuve.

Mon plus grand, sincère et inconditionnel merci à vous tous!!!

Table des matières

Introduction

"En science, la phrase la plus excitante que l'on peut entendre, celle qui annonce des nouvelles découvertes, ce n'est pas 'Eureka' mais c'est 'drôle'". ***Isaac Asimov***

Contexte

Dans une société où les échanges numériques se font de plus en plus fréquents, la sécurisation des informations est un enjeu majeur. S'il y a quelques années la nécessitée de sûreté était restreinte aux cercles militaires et gouvernementaux, aujourd'hui la sécurité s'impose à presque tous les niveaux de la communication d'une société moderne. Des transactions bancaires ont le besoin évident d'être sécurisées, mais aussi le transfert d'informations d'une chaîne de télévision numérique vers une *set-top box* chez son client doit préserver l'intégrité et la confidentialité des données.

Cacher un secret a toujours été un souci. Par exemple, à l'époque de l'Empire Romain, le niveau d'étude de la population et les mécanismes de calcul n'étaient pas très développés. Il suffisait d'écrire un message en décalant les caractères de quelques positions pour que le message devînt incompréhensible. C'était le principe du chiffrage de Cæsar. En effet, l'art et la science de cacher des informations, la cryptographie, existe depuis l'antiquité, mais les méthodes les plus avancées ne sont connues que depuis quelques dizaines d'années.

Ces méthodes modernes sont basées sur l'utilisation de clés cryptographiques, dont le secret est primordial pour la sécurité. Les clés cryptographiques sont souvent une large séquence de chiffres, pouvant varier de quelques centaines de bits jusqu'à des milliers de bits, selon l'algorithme utilisé

et le niveau de sécurité désiré. Certains algorithmes cryptographiques modernes, même s'ils ne sont pas mathématiquement prouvés sûres, ils offrent une sécurité suffisante en pratique, en raison de la difficulté de découvrir le secret d'une clé dans un temps de calcul raisonnable. Cela signifie que les méthodes de cryptanalyse à force brute[1] ne sont pas efficaces malgré la puissance croissante des ordinateurs modernes.

Cependant, les algorithmes de cryptographie, les attaques pour les casser, et les mesures pour contrecarrer ces attaques évoluent constamment. Cela implique une nécessité de flexibilité liée à la sécurité.

Jusqu'au début des années 80, la flexibilité était liée à des implantations des algorithmes en logiciel, exécutés par des processeurs génériques. Pourtant dans les années 60 une architecture a été proposée comprenant une partie fixe plus une partie variable (flexible) [36]. Ce concept a été à l'origine des architectures modernes de calcul aujourd'hui répandues tels que les FPGA (*Field Programmable Gate Arrays*) [84]. Les FPGAs ont permis une flexibilité matérielle, trouvant souvent le compromis entre la souplesse d'un processeur générique et la performance d'un circuit spécifique (ASIC - *Application Specific Integrated Circuit*).

En réalisant le lien entre la nécessité de flexibilité et performances demandées par les systèmes cryptographiques et les solutions architecturales disponibles, il est naturel d'envisager le rapprochement du domaine des architectures reconfigurables à celui de la cryptographie.

Motivation

La recherche au sujet des architectures reconfigurables pour la cryptographie est fondamentale non seulement pour l'augmentation du cycle de vie des dispositifs sécurisés, mais aussi vis à vis de la robustesse de ces systèmes.

Dans le contexte de ce travail, une architecture n'est considerée reconfigurable que si elle possède plusieurs éléments de calcul homogènes, ceux-ci pouvant être une simple table de vérité jusqu'à un processeur, interliés au travers d'un réseau de communication flexible, le tout implanté sur une seule et même puce.

Dans ce même contexte, la robustesse concerne la résistance d'un système cryptographique implanté en matériel, en particulier dans des architectures reconfigurables, contre certains types d'attaques.

Il est donc primordial de s'interroger si les architectures reconfigurables sont bien adaptées aux opérations de cryptographie, non seulement pour les apports en flexibilité, réduction de coût et de performance, mais aussi en ce qui concerne la sécurité.

[1] L'attaque par force brute est une méthode utilisée en cryptanalyse pour trouver un mot de passe ou une clé. Il s'agit de tester, une à une, toutes les combinaisons possibles de façon exhaustive.

Définition du problème

Les concepteurs de systèmes numériques se confrontent au compromis entre performance, réduction de coûts et flexibilité lors du choix architectural pour une application donnée. L'espace de décision varie du circuit spécifique, normalement appelé ASIC, jusqu'aux processeurs, en passant pour les dispositifs reconfigurables, tels que les FPGA.

Pourtant, dans le domaine d'application de la cryptographie, un nouveau paramètre emerge. Il n'est plus question seulement de performance versus flexibilité, mais aussi de sécurité.

Un système cryptographique est l'ensemble formé par l'algorithme de cryptographie et son implantation. La définition peut paraître redondante, mais très souvent un système cryptographique est cassé[2] non à cause d'une faiblesse algorithmique, mais parce que son implantation a laissé des points d'entrée aux attaquants. Donc le système cryptographique ne peut être dit sûr que s'il est basé sur un algorithme prouvé robuste, et implanté de façon soigneuse, évitant les fuites d'informations.

Il existe un certain nombre d'attaques qui profitent des fuites d'informations des circuits cryptographiques, tels que le temps de calcul, les émissions électromagnétiques, la consommation de courant, parmi d'autres. Ces attaques sont appelées attaques par canaux cachés ou *Side Channel Attacks* (SCA) en anglais. Ces attaques sont parmi les plus efficaces et dangereuses pour la sécurité des systèmes cryptographiques.

Dans cette thèse la question de robustesse s'est posée à propos des systèmes cryptographiques implantés sur des architectures reconfigurables concernant les attaques SCA. La réflexion se porte sur les besoins architecturaux nécessaires à la définition d'une architecture reconfigurable pour la cryptographie qui soit raisonnablement performante, flexible, et surtout resistante aux attaques par canaux cachés. Les niveaux de sécurité sont rappelés dans la Section 3.1, mais selon le standard FIPS PUB 140-2 [74], adopté par les institutions fédéraux aux États Unis et qui définit de façon croissante et qualitative des niveaux de sécurité, cette thèse et les contremesures ici proposées se situent entre le niveau 3 et le niveau 4 de sécurité (le plus élévé), ce qui signifie que le dispositif cible d'un attaque peut se trouver en milieu hostile, et souffrir des attaques physiques. Cette thèse apporte des solutions pour certains attaques physiques non invasives.

Plan du manuscrit

Cette thèse est organisée de la façon suivante. Le Chapitre 1 présente une discussion à propos des architectures reconfigurables, les concepts liés à ce domaine, les critères de classification et quelques métriques pour évaluer ces architectures.

Le Chapitre 2 introduit des concepts liés à la cryptographie et explique deux exemples d'algorithmes de cryptographie qui seront mentionnés par la suite du texte. Et en particulière ce Chapitre présente aussi l'arithmétique modulaire, car la solution architecturale proposée dans cette thèse

[2] "Casser" un algorithme de cryptographie signifie découvrir le secret permettant le déchiffrage des textes cryptés, par des personnes non autorisées.

est basée sur cette catégorie d'opérations.

Le Chapitre 3 amène une discussion au sujet des attaques par canaux cachés, en passant par les propositions de contremesures. Dans ce chapitre il est présenté une contremesure originale développée par l'auteur de cette thèse, basée sur la maîtrise de la consommation électrique.

Une architecture reconfigurable dédiée pour la cryptographie est présentée dans le Chapitre 4. Dans le même chapitre une méthode pour contrôler des architectures reconfigurables est proposée. Quelques résultats de performance et de robustesse sont aussi présents dans ce chapitre.

Finalement dans le Chapitre 4.6 les conclusions sont exposées, où les bénéfices de l'architecture développée sont montrés, ainsi que des perspectives d'amélioration.

Contributions

Cette thèse contribue à l'état de l'art sur les points suivants :

1. Proposition d'une architecture reconfigurable pour la cryptographie, robuste contre certains types d'attaques par canaux cachés [59]. Cette architecture permet d'implanter plusieurs algorithmes de cryptographie basés sur l'arithmétique modulaire ; les discussions au sujet de cette nouvelle architecture se trouvent dans le Chapitre 4.
2. Adaptation d'un modèle de gestion de configurations et acheminement de données pour des architectures reconfigurables [18] ; également dans le Chapitre 4, ce modèle est expliqué en détails.
3. Conception d'une structure analogique capable de contrecarrer un type d'attaque par canal caché [64], [63] [62] ; le CMG[3] est présenté dans le Chapitre 3, dans la partie concernant les contre-mesures ;
4. Mis en place d'une plate-forme d'attaque par analyse en courant, permettant de valider certaines attaques par canaux cachés ; cette plate-forme a permis la vérification des concepts développés pendant cette thèse.

Les références [59], [18], [64], [63], [62], [38], [5], [60] et [59] correspondent à quelques unes des communications scientifiques réalisées dans le cadre de cette thèse.

[3] *Current Mask Generator*

CHAPITRE 1

Architectures reconfigurables

"Toute technologie avancée est magique." ***Arthur C. Clark***

De nombreuses applications en télécommunications, multimedia et cryptographie nécessitent des fonctionnalités qui restent flexibles même après la fabrication du système. Une telle flexibilité est fondamentale, car les besoins des utilisateurs, les caractéristiques des systèmes, les standards et les protocoles peuvent changer pendant le cycle de vie du produit. Cette malléabilité peut aussi aboutir à des nouvelles approches d'implantation envisageant des gains de performance et des réductions de coûts du système.

La flexibilité fonctionnelle est normalement obtenue à partir de mises à jour logicielles, pourtant de cette manière le changement est limité à la partie programmable des systèmes. Afin de supplanter cette façon d'obtenir de la flexibilité, des développements ont été réalisés dans le domaine des architectures reconfigurables.

Ce chapitre présente des concepts liés à la reconfiguration, propose une classification des architectures reconfigurables, et exhibe les métriques proposées dans la littérature pour comparer ce type d'architecture.

Pourtant dans la littérature il existe peu de discussions en ce qui concerne l'utilisation de la reconfiguration dynamique pour la cryptographie. A la fin de ce Chapitre une réflexion sur ce sujet est abordée, et dans le Chapitre 4 nous mettons en évidence que la reconfiguration dynamique peut effectivement apporter de la robustesse aux systèmes cryptographiques.

1.1 Définitions

Le terme "architecture reconfigurable" suscite toujours d'intenses débats. Partant du principe que reconfiguration signifie le changement des fonctionnalités d'une architecture après le processus de fabrication, on peut considérer que le processeur générique est le plus reconfigurable des systèmes numériques, car il peut changer de comportement à chaque cycle d'horloge. Pourtant il est possible de dire qu'il est l'un des moins flexibles, car son architecture câblée ne se modifie pas après sa fabrication. De plus, les processeurs génériques n'exploitent que faiblement le parallélisme.

Dans ce document les architectures numériques ne seront considérées comme reconfigurables que si elles comportent plusieurs composants de base tels que éléments de calculs reconfigurables, interconnectés par un moyen d'interconnexion lui aussi reconfigurable. L'exemple classique d'un dispositif avec cette caractéristique est le FPGA[1], mais d'autres circuits en font partie.

Le FPGA est un exemple d'architecture reconfigurable à petit grain. Dans ce domaine le grain est l'unité de calcul de base. La taille du grain est donnée par la taille des données atomiques qui l'architecture en question est capable de manipuler à chaque opération. Dans un FPGA les données sont représentées bit à bit, et calculées à l'aide des tables de vérité (LUT - *look-up tables*). Un dispositif reconfigurable comprenant des opérateurs comme multiplieurs, additioneurs, multiplexeurs, par exemple, peut être considéré comme appartenant à la catégorie des architectures reconfigurables à grain moyen. Cependant, un dispositif composé par des unités logiques et arithmétiques interliées par un réseau intra-circuit peut être considéré comme étant une architecture reconfigurable à gros grain.

Cette définition de taille de grain peut être étendue par rapport au fichier de configuration des architectures reconfigurables. Une architecture à petit grain est configurée normalement par un *bitstream* (flot de bits). De l'autre côté, la configuration d'une architecture à gros grain est faite à l'aide d'un fichier comprenant la programmation de toutes les unités de calcul.

Une architecture reconfigurable peut être reconfigurable de manière statique ou dynamique, complète ou partielle. La reconfiguration dynamique est celle qui permet que le comportement d'un circuit soit modifié sans que l'application qu'il tourne soit complètement arrêtée. Au contraire, la reconfiguration statique demande que le circuit soit mis hors tension pour que la nouvelle configuration puisse être chargée. Une reconfiguration, qu'elle soit statique ou dynamique, peut être totale ou partielle. Dans le premier cas, tout le fichier de configuration est téléchargé vers le circuit.

[1] *Field Programmable Gate Array*

Dans le deuxième, seule la partie du fichier de configuration correspondant à des modifications de fonctionnalité nécessaires est envoyée au circuit.

Les systèmes dynamiquement reconfigurables (SDR) permettent la spécialisation de la logique implantée et/ou du routage des ressources de calcul en cours de fonctionnement. La Figure 1.1 illustre la reconfiguration dynamique au travers d'un graphique en trois axes. Les axes X et Z indiquent le plan spatial, dans lequel le routage et le placement de la logique sont réalisés. Les systèmes reconfigurables de façon statique ne peuvent qu'être vus dans ce plan. Pour les SDR il est nécessaire d'effectuer une analyse temporelle des reconfigurations, car le comportement du dispositif change dans le temps, sans l'arrêt du système. Pour cela, il a été introduit l'axe Y, qui indique les différentes implantations de modules fonctionnels (cœurs) dans le dispositif, par rapport au moment d'arrivée.

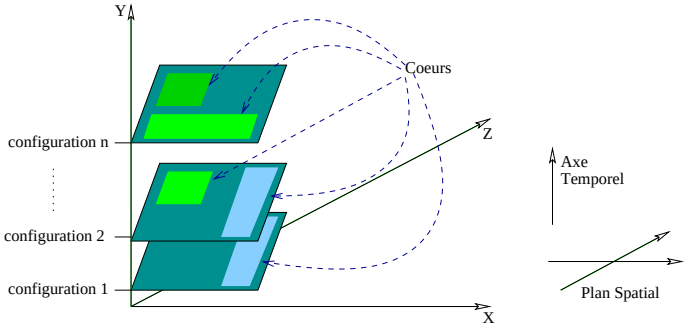


FIG. 1.1: Schéma indiquant la reconfiguration dynamique.

1.2 Classification

Le but de cette section est de se situer parmi la myriade des systèmes reconfigurables, en discutant des critères de classification proposés dans la littérature. Il est important de trouver des critères orthogonaux de comparaison afin de pouvoir comprendre ces architectures et les développer.

1.2.1 La Taxonomie de Radunovic [78]

Dans [78] les auteurs présentent une proposition de taxonomie pour les architectures reconfigurables basée sur les critères de l'objectif de l'architecture, taille du grain, intégration et finalement, la reconfigurabilité du réseau d'interconnexion. La Figure 1.2 montre cette proposition sous forme d'arbre. Il faut souligner qu'à l'époque de la publication de cette taxonomie il était normal de trouver des architectures composées de plusieurs dispositifs reconfigurables disposés sur une carte, avec des critères d'intégration et de reconfiguration du réseau d'interconnexion. Même si ces critères sont dépassés, cette taxonomie reste intéressante comme étant l'un des premiers essais de classification des architectures reconfigurables.

Les prochaines sous-sections discuteront les critères de cette classification.

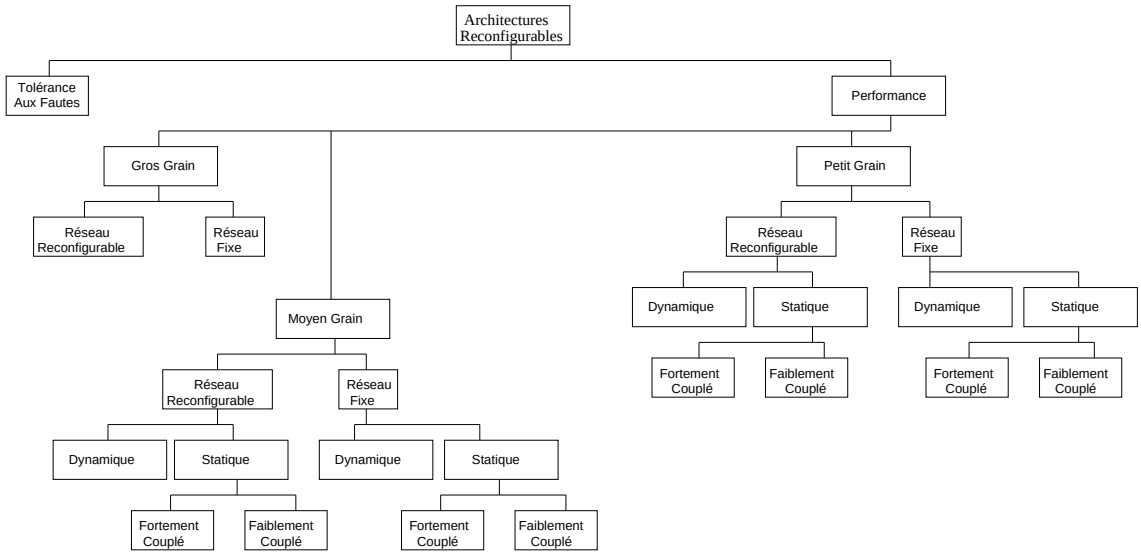


FIG. 1.2: Illustration de la taxonomie de Radunovic à propos des architectures reconfigurables [78].

L'objectif de l'architecture

Selon [78], une architecture reconfigurable peut avoir comme objectif, l'une des deux caractéristiques ci-dessous :

- **Tolérance aux fautes** :Il s'agit d'un des premiers buts du domaine de la reconfiguration. Durant la fabrication et l'utilisation d'un circuit, il y a une probabilité que ce circuit devienne défectueux. A l'époque de la publication de l'article de Radunovic [78], dans des architectures classiques, un problème de ce genre implique que le système entier soit inutilisable. Néanmoins, dans une architecture reconfigurable la partie défectueuse du système peut être détectée et remplacée.
- **Accélération** : L'utilisation des architectures reconfigurables pour accélérer les performances des systèmes a vu le jour dans les années 80, principalement due à sa capacité d'implanter des algorithmes d'une manière plus performante que sur des processeurs génériques, en exploitant par exemple le parallélisme.

Granularité

La granularité peut être définie de différentes façons, comme par exemple le nombre de fonctions booleènes qu'un bloc logique peut implanter, le nombre de portes NAND de deux entrées, le nombre total de transistors, le nombre d'entrées et de sorties, etc [84]. Pourtant l'auteur de l'Olimpo [78] utilise pour la granularité le même concept présenté dans la section 1.1, en bref, une architecture à petit grain est composée des tables de vérité, les moyens grains sont composées par des multiplexeurs (plus additioneurs, multiplieurs, etc.), alors que les architectures à gros grains

utilisent comme éléments de base des unités de calcul plus complexes, tels que des processeurs[2].

Intégration

En ce qui concerne la forme d'intégration, la classification donnée par [78] est la suivante :

- **Systèmes autonomes** : Après la charge initiale, le système s'adapte dynamiquement (et sans interférence externe) jusqu'à ce que le résultat des calculs soient finis. Un exemple d'utilisation d'une architecture autonome est la simulation des systèmes vivants, comme celui proposé par Eduardo Sanchez en [85].
- **Systèmes fortement couplés** : Les unités de reconfiguration jouent un rôle similaire aux unités de calcul d'un processeur et peuvent manipuler des données directement au sein des registres, ou à partir d'un bus.
- **Systèmes faiblement couplés** : Les unités de reconfiguration sont annexées au système comme co-processeurs, au départ dans la même carte, mais avec la capacité d'intégration actuelle on peut avoir des systèmes faiblement couplés dans le même circuit.

Reconfigurabilité du réseau d'interconnexion

Ce critère concerne le réseau d'interconnexion entre les unités reconfigurables, qui peut être fixe ou reconfigurable :

- **Réseau externe fixe** : Il n'est pas possible réaliser un changement du chemin entre les unités reconfigurables. C'était la solution la moins chère pour avoir une architecture reconfigurable sur carte dans les années 80.
- **Réseau externe reconfigurable** : Une architecture ayant un réseau externe reconfigurable peut fonctionner comme une grande unité reconfigurable, avec toute la flexibilité inhérente.

1.2.2 Les critères de Page [76]

Le matériel dynamiquement reconfigurable est la technologie qui donne impulsion au développement des systèmes que unissent les caractéristiques des processeurs génériques et la flexibilité matérielle des FPGAs. Sous cet angle, les SDR peuvent être classifiés selon les critères suivants [76] :

- Modèle d'opération ;
- Architecture mémoire ;
- Modèle d'opération de la mémoire locale ;
- Modèle de programmation.

[2]La différence c'est que Radunovic parle de granularité, quand en réalité le concept s'agit plutôt de la taille du grain et non de la granularité.

Le modèle d'opération

En considérant un dispositif reconfigurable et un processeur générique comme décrits ci-dessus, il existent plusieurs façons de les intégrer. Donc il est possible d'envisager une classification selon les différents niveaux d'interaction entre la partie reconfigurable et le processeur.

- **Co-processeur** : Le processeur expédie une instruction (ou une séquence d'instructions) qui va être détectée et interprétée par l'unité reconfigurable. Dans ce cas, la partie reconfigurable devient un co-processeur.
- **Appel distant à des fonctions** : Le processeur lance une instruction (ou une séquence d'instructions) qui est interprétée par le dispositif reconfigurable comme un appel distant à des fonctions (RPC - *Remote Procedure Call*). Cela est semblable à l'interaction co-processeur, excepté le fait que le RPC demande une interface utilisant une synchronization.
- **Modèle Client-Serveur** : L'algorithme implanté dans le dispositif reconfigurable est un process serveur qui est semblable au RPC, mais où les communications peuvent arriver de n'importe quel process exécuté par le processeur.
- **Processes parallèles** : Les process qui sont exécutés par le dispositif reconfigurable et par le processeur sont indépendants. La communication parmi les process est faite au travers du passage de messages.

L'architecture mémoire

Le programme exécuté dans le co-processeur reconfigurable a besoin d'une certaine quantité de variables stockées de façon temporaire afin de exécuter son application. Selon Page il peut exister trois modèles de mémoire :

- **Mémoire sans accès externe** : Dans certains cas, le dispositif reconfigurable n'as pas besoin d'une mémoire externe. C'est une situation qui n'est acceptable que lors qu'une application nécessite que de peu d'états pour son opération.
- **Mémoire partagée** : Le dispositif reconfigurable peut accéder à n'importe quel mémoire associée au bus qu'il partage. Pourtant cet approche génère une surcharge de temps, car nécessite d'un arbitrage complexe pour l'accès à la mémoire.
- **Mémoire interne** : Le dispositif reconfigurable possède suffisamment de mémoire interne, augmentant ainsi la performance du système. Cependant, pour éviter le risque de duplication de données par rapport au processeur, il est nécessaire d'avoir un contrôle de cohérence de cache.

Modèle d'opération de la mémoire locale

Les modèles suggérées par Page ne sont pas mutuellement exclusifs, i.e., ils peuvent être non-orthogonaux :

- **Mémoire de bloc** : La mémoire locale est suffisamment large pour contenir un ensemble de données pour le calcul.

- **Mémoire de file** : La mémoire agit comme une file quand il y a un échange de données avec le processeur.
- **Mémoire Cache** : La mémoire locale peut fonctionner comme un cache sur une structure de données plus importante gérée par le processeur.

Modèle de programmation

Il y plusieurs manières de programmer le dispositif reconfigurable. Ces différents modèles permettent d'exploiter plusieurs scénarios de compromis coût-performance :

- **Tout matériel** : L'application est transformée dans une description matérielle de l'algorithme, et est exécutée comme un matériel spécifique dans le dispositif reconfigurable.
- **Processeur à usage spécifique** : L'application est convertie vers un code de machine abstrait, pour un processeur abstrait (ASIP - *Application Specific Instruction-Set Processor*). Le deux sont alors co-optimisés pour produire la description d'un ASIP et son code machine. La description est donc compilée vers le dispositif reconfigurable.
- **Réutilisation séquentielle** : Si l'application est trop importante pour être implantée directement sur le dispositif reconfigurable, l'algorithme peut être divisé en parties, de façon qu'il soit exécuté partiellement, en utilisant la capacité de reconfiguration dynamique du dispositif. Les gains obtenus par la réutilisation du matériel doivent être balancés par le temps nécessaire pour exécuter la reconfiguration.
- **L'usage multiple simultané** : Si les ressources du dispositif reconfigurable sont suffisamment importantes, il est possible d'avoir plusieurs applications qui tournent en parallèle et qui communiquent indépendamment avec le processeur.
- **L'usage à la demande** : Critère qui correspond à la réutilisation séquentielle. Il est ici possible d'imaginer un "matériel virtuel", semblable à la "mémoire virtuelle" proposée pour les systèmes d'exploitation.

1.2.3 Statique versus Dynamique [85]

Selon Eduardo Sanchez [85] un système reconfigurable peut être vue selon deux angles par rapport à sa manière d'être reconfiguré : de manière statique ou dynamique. Dans la première le fichier de configuration est chargé une seule fois avant l'exécution d'une tâche, et une nouvelle configuration implique forcément la fin de l'ancienne. Dans la deuxième façon, dite dynamique, le fichier de configuration du dispositif reconfigurable peut être modifié au cours d'exécution d'une application, sans l'arrêter. Sanchez propose donc deux critères de classification pour les systèmes reconfigurables :

Statique

Les systèmes reconfigurables de façon statique possèdent peu de flexibilité. La configuration du système arrive avant l'exécution de la tâche qu'il est censé accomplir. Durant toute l'exécution de

cette tâche, le système reste figé. Après il peut être reconfiguré de façon complète. C'est le cas le plus répandu, car il demande peu de support logiciel et de contrôle.

Dynamique

Dans les systèmes dynamiquement reconfigurables un fichier de configuration peut entraîner la modification du comportement du circuit pendant l'exécution d'une application sans mettre le système hors tension. Son but est de s'adapter dynamiquement à des spécifications, mais aussi de prendre en compte la manipulation des spécifications incomplètes.

1.2.4 Réseau d'interconnexion pour les architectures reconfigurables à gros grain

En 2001 Reiner Hartenstein a réalisé une discussion à propos des architectures reconfigurables à gros grain. Dans [44] il parle des problématiques liées à la façon de programmer (configurer) une architecture reconfigurable, mais il montre aussi différentes manières de connecter les éléments qui la composent. Dans cette section ces caractéristiques sont prises comme des critères de classification.

Architectures en maille

Architectures *mesh-based* (en maille) organisent ses éléments de calcul dans une matrice rectangulaire 2-D, avec des connexions horizontales et verticales, qui supportent un riche réseau de communication, privilégiant les échanges voisin à voisin, de façon à privilégier le parallélisme. Typiquement sont ajoutés des lignes longues établissant les connexions de longueur supérieure à 1.

Architectures en vecteur linéaire

Quelques architectures reconfigurables sont basées sur des vecteurs linéaires (*linear arrays*), avec une communication voisin à voisin, envisageant le *mapping* de *pipelines*. Si les *pipelines* ont des divergences, alors des ressources de routage additionnelles sont nécessaires, comme des lignes plus longues, souvent nécessitant une segmentation.

Architectures en croisillon

La ressource de routage le plus performant est un commutateur en croix (*crossbar*), car il facilite le routage parmi tous les éléments de calcul.

Aux critères de Hartenstein il est possible d'ajouter celui du réseau sur puce, moyen de communication intra-circuit(NoC[3]) qui se développe de manière accéléré dernièrement.

[3]Network on Chip.

Architectures basées sur un réseau sur puce

Récemment les réseaux sur puce ont vu le jour comme une alternative pour connecter des cœurs hétérogènes. Pourtant ces réseaux peuvent servir de média de communication pour une matrice homogène d'éléments de calcul, permettant la construction d'architectures à gros grain très flexibles.

1.2.5 Modèles de reconfiguration

Complétant les critères proposés en 1.2.3, Katherine Compton [23] propose différentes manières de classiffier la façon d'amener des nouvelles fonctionnalités à une architecture reconfigurable dynamiquement.

- **Contexte unique** : Un dispositif reconfigurable à contexte unique correspond à un système reconfigurable de façon statique, car due à un accès séquentiel pour le fichier de configuration, le changement de fonctionnalité implique une reconfiguration complète (et donc un arrêt complet) du système. Malgré le fait que ce genre de reconfiguration est simple à implanter, cela entraîne une surcharge quand il est nécessaire reconfigurer juste une partie du dispositif.
- **Multicontexte** : Un dispositif reconfigurable multicontexte inclut des multiples fichiers de configuration et de multiples mémoires pour les stocker. Ces mémoires de configuration peuvent être vues comme des plans d'information de configuration. Un plan peut être activé à un moment donné, mais le dispositif peut commuter rapidement entre différents plans (ou contextes). De cette façon le dispositif multicontexte peut être considéré comme un ensemble multiplexé de systèmes à contexte unique.
- **Reconfiguration partielle** : Dans certains cas la reconfiguration ne concerne pas la totalité du dispositif reconfigurable, ou seulement une partie du système a besoin d'être modifiée. Dans les deux situations la reconfiguration partielle du dispositif est envisageable. Les dispositifs permettant ce genre de reconfiguration autorisent la reconfiguration dynamique dans son sens le plus complet.
- **Reconfiguration en *pipeline*** : Dans les dispositifs implantant la reconfiguration en *pipeline* la reconfiguration arrive en étapes incrémentales. Chaque étage est configuré entièrement. Cela est utilisé dans des calculs du genre chemin de données, où il y a plus d'étages de configuration que de matériel disponible pour les implanter simultanément.

1.2.6 Niveaux de reconfiguration

Conformément à ce qui est avancé dans la Section 1.1, il existe différents niveaux d'abstraction selon lesquels il est possible de classiffier le degré de reconfiguration d'une architecture. Raphaël David [30] propose une classification selon le potentiel de reconfiguration qui reflète notamment les caractéristiques des microprocesseurs, des FPGAs et des architectures reconfigurables à gros grain.

Niveau système

Ce niveau de reconfiguration traite le changement de fonctionnalité des processeurs via la modification de l'orientation du chemin de données. Les données sont acheminées vers des unités de calcul (ou vers les registres, ou la mémoire) selon les besoins de l'application, en modifiant des bits dans les commutateurs et multiplexeurs dans l'architecture. Cela permet une très grande flexibilité logicielle, pourtant, les unités de calcul ne se modifient pas, ce que entraîne une certaine rigidité matérielle.

Niveau fonctionnel

Pour assouplir cet effet, des nouvelles architectures orientées flot de données ont été proposées. Une architecture reconfigurable au niveau fonctionnel suppose une multitude d'unités de calcul interconnectés par un réseau d'interconnexion. Étant donné la taille du chemin de données de ces architectures, elles sont souvent appelées architectures reconfigurables à gros grain. Le problème de cet approche est la difficulté de trouver un modèle fiable de programmation, au contraire des architectures du type processeur.

Niveau logique

Une architecture est dite reconfigurable au niveau logique quand elle est composée de plusieurs unités de calcul et avec un riche réseau d'interconnexion, avec un chemin de données de faible largeur. Les unités de calcul opèrent au niveau logique, implantant des fonctions booléenes. Normalement, pour implanter une application, il est nécessaire de disposer d'une grande quantité de ressources, aussi bien que l'utilisation d'un complexe réseau de communication. Cela induit une faible fréquence de reconfiguration, malgré son apport en flexibilité matérielle.

1.3 Métriques

Les métriques sont un ensemble de paramètres et/ou manières d'appréciation quantitative d'un système. Une métrique définie ce qui va être mesuré. L'analyse d'une métrique peut inférer une tendance, ou la productivité, par exemple. Dans le domaine des architectures reconfigurables, les métriques proposées concernent fréquemment l'intérêt de réaliser ou non une reconfiguration. Les sous-sections suivantes montrent deux métriques répandues pour comparer les architectures reconfigurables.

1.3.1 La densité fonctionnelle

La flexibilité apportée par les architectures reconfigurables est liée à un coût. Un certain temps de reconfiguration et une largeur de bande passante sont nécessaires pour transférer les données de la mémoire de stockage de configurations vers la mémoire de configuration du dispositif. En outre,

la reconfiguration d'une architecture occasionne une certaine période d'inactivité (notamment pour les architectures reconfigurables de façon statique (Section 1.2.3). En 1998 a été proposé la métrique appelée densité fonctionnelle [98], qui compare des systèmes reconfigurables par rapport à des systèmes classiques. Une métrique similaire a été présentée quelques mois plus tard dans [31].

La densité fonctionnelle est définie par le coût d'implantation d'une certaine application dans le dispositif reconfigurable. Ce coût (C) est régulièrement mesuré comme le produit de la surface totale nécessaire pour implanter l'application en matériel (A) et le temps total d'opération (T) (Équation 1.1) :

$$C = A \cdot T \tag{1.1}$$

Il s'applique aux systèmes où le débit (nombre d'opérations réalisées par seconde) est plus importante que la latence. (T) inclue le temps nécessaire pour l'exécution, le contrôle, l'initialisation et le transfert de données. La Densité Fonctionnelle (D) mesure donc le débit de calcul d'une unité reconfigurable, et est définie comme l'inverse de C (1.2) :

$$D = \frac{1}{C} = \frac{1}{T \cdot A} \tag{1.2}$$

Cette métrique sera utilisée afin de comparer les circuits reconfigurables statiques à ceux dynamiquement reconfigurables. L'incrément de la densité fonctionnelle (I), calculé dans l'équation 1.3, des systèmes dynamiquement reconfigurables (D_r) sur les alternatives statiques (D_s), est calculé comme la différence entre D_r et D_s :

$$I = \frac{\delta D}{D_s} = \frac{D_r - D_s}{D_s} = \frac{D_r}{D_s} - 1 \tag{1.3}$$

La différence la plus importante entre systèmes reconfigurables dynamiquement et statiquement, réside dans le coût additionnel de reconfiguration. Le temps de configuration doit, donc, être ajouté au calcul du temps total d'opération d'un circuit. Pour un SDR, le temps total T est donné par l'addition du temps total d'exécution (T_e) et le temps total de configuration (T_c), conforme l'équation 1.4 :

$$T = T_c + T_e \tag{1.4}$$

En substituant T dans l'équation 1.2, il est obtenu la densité fonctionnelle augmentée du coût de configuration 1.5 :

$$D_r = \frac{1}{A \cdot (T_c + T_e)} \tag{1.5}$$

Néanmoins, même si le temps absolu est un paramètre important, le temps de configuration relatif au temps d'exécution est plus informatif. Le rapport du changement de configurations (f) exprime ce paramètre 1.6 :

$$f = \frac{T_c}{T_e} \tag{1.6}$$

Donc, le temps total d'opération d'un dispositif reconfigurable peut être réecrit comme 1.7 :

$$T = T_e \cdot (1 + f) \tag{1.7}$$

En substituant ce temps en 1.2, la métrique de la densité fonctionnelle est obtenue 1.8 :

$$D = \frac{1}{A \cdot T_e \cdot (1 + f)} \tag{1.8}$$

Comme suggéré en 1.8, les temps les plus longs de reconfiguration ne peuvent être tolérés que si les temps d'exécution correspondants soient suffisamment longs (i.e. f petit). A la limite, quand $f \to 0$, la surcharge imposée par la reconfiguration est négligeable. De tels systèmes s'approchent de la densité fonctionnelle maximale (D_{max}) possible. Cette valeur est donnée par l'équation 1.9 :

$$Dmax = \lim_{f \to 0} D_r = \frac{1}{A \cdot T_e} \tag{1.9}$$

Utilisant D_{max}, la limite supérieure de l'incrément (I_{max}) sur un système statique peut être trouvée. Ce paramètre suggère le bénéfice maximal d'un SDR, et c'est par ailleurs une bonne indication de la pertinence d'utilisation d'un SDR pour une application donnée. L'équation 1.10 calcule ce I_{max} :

$$I_{max} = \frac{D_{max}}{D_s} - 1 \tag{1.10}$$

La densité fonctionnelle est un bon paramètre à prendre en compte lors de la conception d'un système dynamiquement reconfigurable. Un cas d'étude a été réalisé dans la thèse [11] Pourtant, cette métrique est orientée applications. Pour bien évaluer l'intérêt et les gains d'une architecture reconfigurable, il faut prendre en compte aussi des caractéristiques intrinsèques de ces architectures. La sous-section suivante traitera des métriques inhérentes aux SDR.

1.3.2 La Rémanence

La métrique appelée rémanence a été introduite en [33]. Elle consiste à caractériser le dynamisme d'une architecture, tenant en compte sa capacité d'être reconfigurée et la fréquence à laquelle cela se produit. La rémanence est calculée comme montre l'équation 1.11 :

$$R = \frac{N_a \cdot F_e}{N_c \cdot F_c} \tag{1.11}$$

La rémanence suppose qu'une architecture est composée par des unités de calcul (N_a) qui opèrent à une fréquence F_e. L'architecture est donc capable d'être reconfigurable à travers du

changement de N_c opérateurs à une fréquence (F_c) que peut être différente de F_e (Fréquence de Configuration). Le cycle de calcul est défini de façon relative à la fréquence d'exécution F_e, donc une période de F_e correspond à un cycle. La rémanence est donc calculée en fonction du nombre de cycles d'exécution nécessaires à la reconfiguration d'une architecture.

La rémanence indique donc qu'un certain volume de données doit être calculé entre deux configurations pour que l'architecture soit efficace. Par ailleurs, la rémanence (ou son inverse, pour être plus précis) informe le degré de dynamisme d'une architecture. Si une architecture possède un R égal à 1[4], cela signifie que cette architecture est très dynamique. Dans le cas contraire, une architecture est dite statique si sa rémanence est trop importante.

Les points les plus intéressants de cette métrique sont :

- **Affranchissement de la taille du grain** : La rémanence permet de comparer des architectures dynamiquement reconfigurables indépendamment du fait qu'il s'agisse d'une architecture à gros grain où d'un FPGA. Le rapport d'opérateurs dans le calcul de R neutralise la notion de grain.
- **Indépendance du mode de reconfiguration** : Cette métrique ne prend pas en compte si la reconfiguration est partielle ou complète.

1.4 État de l'art

Ce Chapitre décrit brièvement quelques architectures reconfigurables développées récemment (depuis 2000) afin de fournir des informations à propos des architectures reconfigurables de façon représentative, mais non exhaustive.Ces architectures ont inspiré de manière plus ou moins évidente l'architecture développée dans le cadre de cette thèse et présentée Dans le Chapitre 4.

1.4.1 Architectures à Gros Grain

Les architectures reconfigurables à petit grain pourvoient souvent une flexibilité très importante lors de l'implantation d'un algorithme en matériel. Néanmoins, il y a une pénalité associée en ce qui concerne la quantité de ressources nécessaires pour calcul.

Une alternative au petit grain sont les architectures à gros grain, taillés pour réaliser des opérations qui ont besoin de chemins de données dites *orientées mots*. Comme ses blocs fonctionnelles sont optimisés pour des calculs de la même largeur que les données traitées, ces opérations sont réalisées de façon performante. Autre aspect positif des architectures reconfigurables à gros grain, c'est qu'elles ont des réseaux d'interconnexion parmi ses éléments de calculs plus simples que celles des FPGAs. Par conséquent, les architectures à gros grain sont plus efficaces aussi en termes de consommation par rapport aux FPGAs.

Les sous-sections suivantes décrivent deux architectures reconfigurables à gros grains représentatives de l'état de l'art.

[4]Par exemple, un microprocesseur.

Systolic Ring

Le Systolic Ring [89] est une architecture à gros grain dédiée pour les applications du traitement d'image et du signal. Cette architecture peut être vue à partir de deux niveaux d'abstraction. Le premier est composé par une couche de configuration constituée d'un ensemble de cellules de mémoires contenant la configuration du composant. Le deuxième est constitué d'une couche opérative : un ensemble d'unités fonctionnelles semblables à des petits processeurs du type RISC. La Figure 1.3 illustre ce concept. Conformément à cette illustration, le Systolic Ring n'a pas été conçu pour être une solution isolée, au contraire, il doit être un composant dans un système sur puce, avec un processeur hôte et une hiérarchie mémoire.

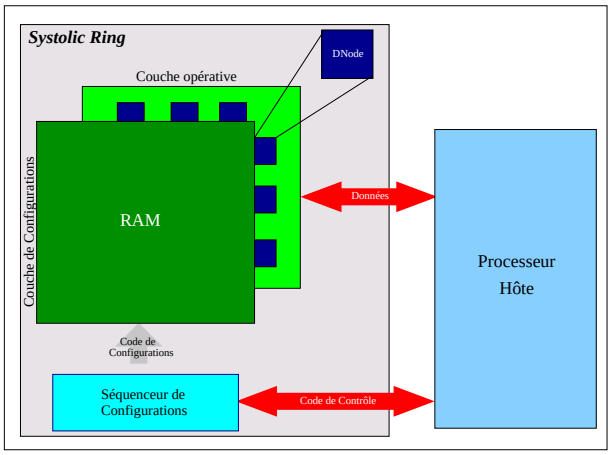


FIG. 1.3: Vue d'ensemble du Systolic Ring

Le Systolic Ring, différemment des FPGAs qui ont pour briques de base des CLBs (*Configurable Logic Blocs*), possède un composant à gros grain, appelé DNode (*Data Node*). Il s'agit d'un composant du genre chemin de données avec une ULA (*Unité Logique et Arithmétique*) et quelques registres (voir Figure 1.4). Le DNode est configuré au travers d'un code du type microinstruction.

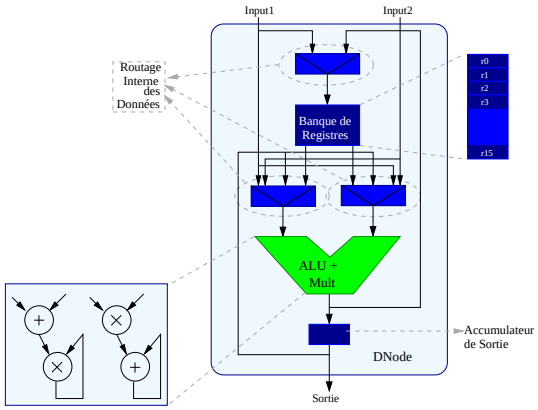


FIG. 1.4: L'architecture du DNode

La couche de configuration suit le même principe que la configuration des FPGAs : il s'agit

d'une RAM (*Random Access Memory*) laquelle contient la configuration de tous les DNodes et ses interconnexions. Le tout est contrôlé par un processeur RISC avec un jeu d'instruction dédié au contrôle de configurations. Sa tâche est de gérer dynamiquement la configuration et le réseau de communication, mais aussi de contrôler l'échange de données entre le Systolic Ring et le processeur hôte.

D'un point de vue fonctionnel, un système d'exploitation implanté dans le processeur hôte charge des applications taillées pour la co-exécution. L'application est constituée par un code exécuté par le hôte et le code de gestion du Systolic Ring. Le hôte donc télécharge ce code de gestion vers la mémoire de contrôle. Le contrôleur de configurations est donc est capable de changer la fonctionnalité du Systolic Ring, de manière partielle ou complète, à chaque cycle d'horloge. Lorsque le code de gestion est chargé, l'architecture est prête à exécuter des calculs. L'hôte envoie donc les données vers la couche opérative du Systolic Ring via un schéma spécifique, et récupère les données traitées. Grâce à la gestion dynamique des configurations, il est possible de multiplexer l'envoie de données, et les calculer par des noeuds processeurs de façon séquentielle (multiplexage matérielle) ou concurrent (statique).

La principale différence du Systolic Ring à l'égard des autres architectures à gros grain, c'est son réseau d'interconnection en double anneau. Dans le Systolic Ring, tous les DNodes forment un anneau dont nombre de couches, et le nombre de DNodes par couche déterminent respectivement la longueur et la largeur de l'anneau. La Figure 1.5 indique que les DNodes sont donc organisés par couches. Une couche est connectée aux deux couches adjacentes, mais aussi à un des commutateurs reconfigurables capables de connecter une couche à n'importe quel autre couche de l'architecture. Le commutateur gère également les communications arrivants du processeur hôte, et l'usage (optionnel) d'un bus partagé.

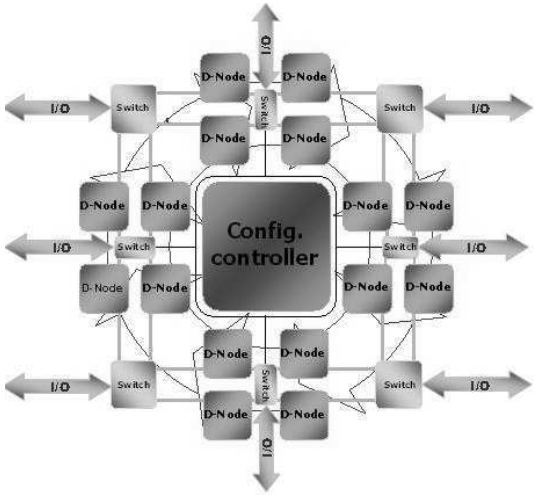


FIG. 1.5: Le schéma en anneau

En outre, le Systolic Ring a un deuxième flot de données, cette fois-ci dans le sens inverse, afin de permettre la re-injection des données traitées, ce qui est un problème recourant dans le traitement d'image. Ce flot de données inverse évite des structures de routage plus complexes.

La dernière tâche réalisée par chaque commutateur est d'écrire de façon inconditionnelle (sans besoin de contrôle) le résultat calculé dans un pipe-line (chaque commutateur a son propre pipe-line), qui permet la rétro-alimentation de chaque donnée dans l'étage précédent. Ensuite ces étages ont le choix de prendre ou pas ces informations. Cette technique assure une bonne scalabilité de l'architecture, évitant le problème de routage. De plus, une réflexion a été menée dans le cadre de la thèse de Pascal Benoit sur les méthodes d'assignation de tâches [11].

Un prototype de cette architecture a été validé dans une carte possédant un FPGA et un contrôleur RISC ; et elle a été aussi synthétisée dans une technologie CMOS $0,18\mu$, atteignant 200MHz de fréquence d'horloge, pour une capacité de calcul de 1600MIPS pour 8 noeuds de calcul.

Dart

DART [28] est une architecture reconfigurable développée dans le but d'apporter à la fois puissance de calcul et faible consommation aux applications mobiles de troisième génération. Cette architecture est composée par des grappes d'unités de calcul, comme décrit la Figure 1.6. Ces grappes peuvent fonctionner indépendamment où en combinaison avec les autres. Cela diminue la complexité du contrôle de l'ensemble du système. Donc la priorité du contrôleur du système n'est pas de gérer chaque grappe, mais d'agencer des tâches qui seront exécutées par les grappes, selon sa priorité et la disponibilité des ressources. Ainsi le contrôleur n'a pas à ordonnancer les instructions de chaque tâche, mais simplement spécifier aux grappes quel tâche elles doivent exécuter.

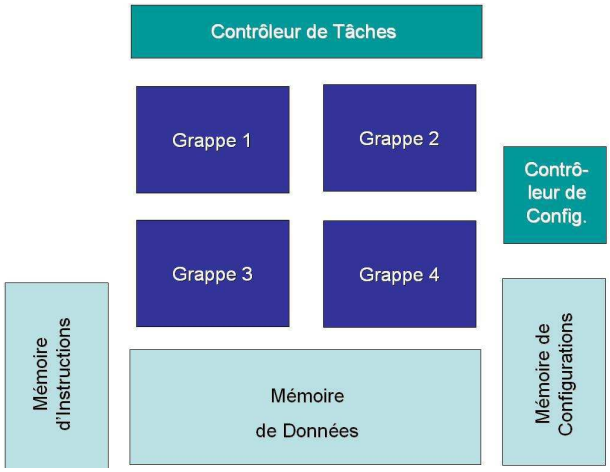


FIG. 1.6: L'architecture DART, niveau système

L'organisation hiérarchique de DART permet, au delà de la distribution du contrôle, la répartition du calcul. Par conséquent, il est possible de joindre un grand nombre de ressources sans augmenter le coût d'interconnexion. En effet, cela entraîne une amélioration de la performance et de la consommation. De plus, cette organisation facilite le développement d'outils de programmation.

Pour donner plus de flexibilité, tout en restant simple à programmer, les grappes comprennent des unités de calcul appelées *DataPath Reconfigurable (DPR)*, et un petit cœur FPGA (voir Figure

1.7). Les DPRs peuvent être connectés les uns avec les autres à travers d'un réseau du type maille segmenté, ce qui permet un calcul massivement parallèle ; mais ils peuvent aussi agir comme des unités indépendantes, opérant sans liaison les uns avec les autres, sur différentes flots de tâches.

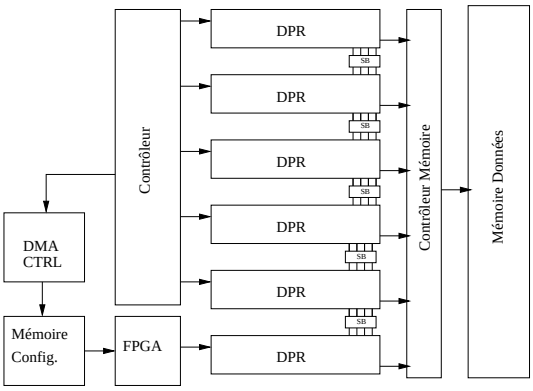


FIG. 1.7: Structure d'une grappe du DART

Toutes les primitives de calcul accèdent à la même mémoire, et la reconfiguration est gérée par un contrôleur local. La reconfiguration de la zone FPGA est faite de façon série, pendant que la reconfiguration des DPR arrive de manière semblable à celle des microprocesseurs. La différence c'est que le schéma de configuration du DART évite des lectures excessives de la mémoire d'instructions, en stockant les microinstructions dans des registres dans chaque DPR. Par conséquent, l'architecture devient efficace du point de vue consommation d'énergie.

Les primitives de calcul arithmétiques du DART sont les DPRs. Chaque DPR est organisé autour de ressources fonctionnelles et de mémoires, liés selon un flexible réseau de communication. Chaque DPR possède quatre unités fonctionnelles suivies d'un registre et supportant des calculs *SWP* (*Sub-Word Parallelism*). Ce concept est justifié par la nécessité d'opérer sur des différents tailles de données, notamment codage d'audio et de video sur 8 et 16 bits. Les concepteurs de DART ont donc opté pour implanter des opérateurs arithmétiques optimisés pour les formats de données les plus ordinaires (16 bits), mais supportant SWP pour pouvoir traiter des données plus petits.

Les résultats présentés en [29] montrent que les DPRs ont été synthétisés, atteignant une fréquence de 130MHz. Basés sur ce résultat, les auteurs estiment que DART peut atteindre $10,9$ $GOPS$ par grappe. Par rapport à la consommation, DART réalise $32M0PS/mW$ pour des opérations sur 16 bits.

PipeRench

En analysant la nature statique d'une configuration, se vérifie l'existence de deux problèmes majeurs :

- Un calcul peut demander plus de ressources matérielles que la quantité disponible ;

- L'implantation d'un cœur d'IP[5] actuel n'exploite pas des ressources matérielles additionnelles qui seront disponibles dans les versions futures du dispositif cible.

Afin de répondre à ces problèmes, la technique de reconfiguration par *pipeline*, envisageant la virtualisation matérielle a été proposée. Cette technique permet l'implantation des functions logiques plus grandes que les ressources du dispositif cible, au travers d'une succession de reconfigurations [41].

Un *pipeline* de configurations entraîne la virtualisation du matériel par la division de la logique statique en des morceaux qui correspondront aux étages du *pipeline* dans l'application. La Figure 1.8 illustre le procédé de virtualisation d'une logique à cinq étages dans un matériel ne disposant que des ressources suffisants pour trois étages. Dans la première partie de la Figure 1.8-(a) l'application en 5 étages est implantée dans les ressources disponibles, sur 7 cycles d'horloge. La Figure 1.8-(b) montre l'état des étages implantés dans un dispositif ayant moins de ressources. Dans cet exemple, le premier étage du *pipe* virtuel est configuré dans le cycle 1, et est prêt à exécuter dans le cycle suivant, étant exécuté pendant deux cycles consécutifs. Malgré l'absence d'un quatrième étage physique dans le *pipe*, dans le cycle 4 le quatrième étage du *pipe* virtuel est configuré dans l'étage physique 1, en substitution du premier étage virtuel.

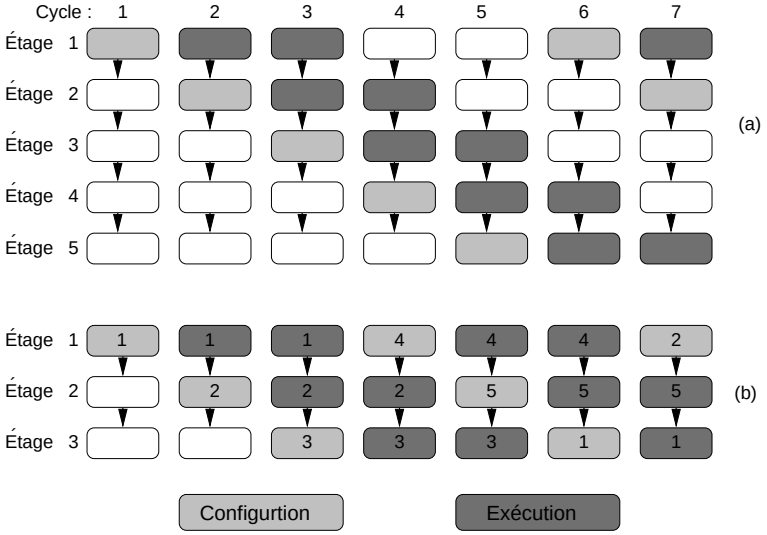


FIG. 1.8: Exemple d'application du PipeRench : virtualisation d'un *pipeline* à 5 étages vers un dispositif ne supportant que 3 étages.

En général, quand une application est "virtualisée" avec ν étages virtuels, dans un dispositif ayant une capacité d'implanter ρ étages, avec $\rho < \nu$, le débit sera proportionnel à $\frac{(\rho-1)}{\nu}$. Le débit est une fonction linéaire de la capacité du dispositif.

Comme certains de ces étages sont configurés pendant que d'autres exécutent, la reconfiguration ne diminue pas la performance. Une reconfiguration par *pipeline* est considérée réussie quand est réalisée une configuration d'un étage du *pipe* par cycle d'horloge. Pour aboutir à cela, il a été ajouté au dispositif un *buffer* de configuration dans le circuit, ainsi qu'un petit contrôleur capable

[5]Un cœur d'IP (*IP Core* en anglais) est un module pré-projeté et pré-validé d'un matériel numérique.

de gérer le processus de configuration.

Cependant la reconfiguration par pipeline impose quelques restrictions relativement aux types d'applications qui peuvent être implantées. La contrainte la plus importante concerne le fait que l'état de n'importe quel étage du *pipeline* doit être impérieusement une fonction de l'étage précédent et de l'étage suivant ; donc ne sont permises que les connexions physiques entre étages consécutifs.

Le PipeRench est un dispositif composé par un réseau d'éléments logiques et de stockage interconnectés. Il est différent des FPGAs car cette architecture propose la virtualisation du matériel au travers d'un *pipeline* de configurations. Cette architecture est particulièrement intéressante pour les applications basées sur des flots uniformes (comme le traitement d'images ou de son), ou n'importe quel calcul simple sur des grandes quantités de petits ensembles de données [41].

Avec PipeRench existe un gain considérable de performance par rapport aux processeurs conventionnels. Pendant que ces dernières forcent une mise en série des opérations intrinsèquement parallèles, le PipeRench exploite le parallélisme inhérent de telles opérations, en outre d'augmenter la flexibilité matérielle.

Le PipeRench se positionne par rapport au processeur comme co-processeur fortement couplé. La Figure 1.9 montre une vue d'ensemble d'un étage du PipeRench, alors que la Figure 1.10 illustre d'une façon plus détaillée son élément de calcul (PE - *Processing Element*). PipeRench contient un ensemble d'étages physiques de *pipeline*. Chaque étage possède un réseau d'interconnexion et un ensemble de PEs.

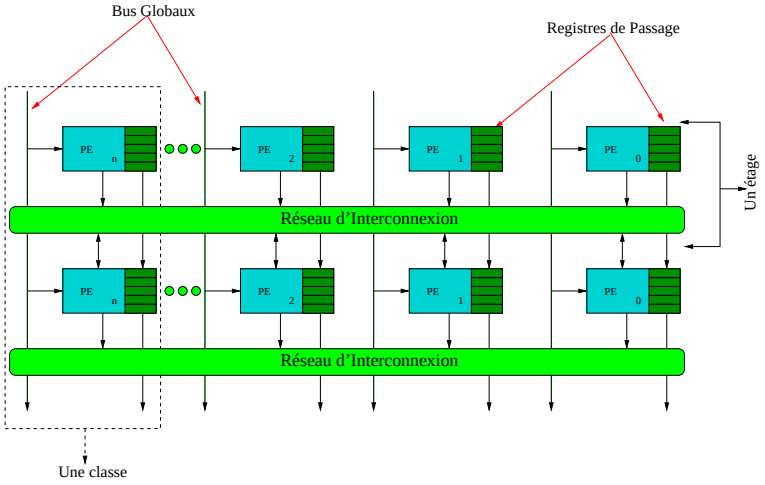


FIG. 1.9: L'architecture PipeRench et ses moyens de communication.

Chaque PE contient une unité logique et arithmétique (ULA) et une banque de registres utilisés pour la communication entre les étages (registres de passage). Chaque ULA contient des LUTs et un contrôle additionnel pour la propagation de retenue, détection de zéro, etc.

La Figure 1.9 montre aussi comme les PEs communiquent.

A partir des réseaux d'interconnexion les PEs peuvent accéder aux opérandes des étages précédents qui sont stockés dans les registres de sortie, aussi bien que les sorties des autres PEs du même

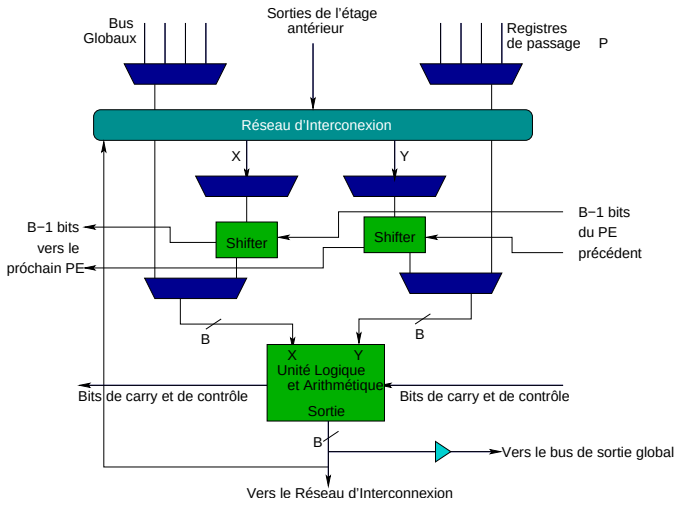


FIG. 1.10: L'élément de calcul du PipeRench.

étage. En fonction de la virtualisation du matériel, les bus des PEs ne peuvent pas se connecter à plus d'un étage consécutif. Cependant, les PEs peuvent accéder aux bus d'entrée et de sortie globaux. Pour que les sorties des applications arrivent à ses destinations, elles doivent faire usage des ces bus globaux.

D'une façon abstraite, les PEs d'un même étage sont classés selon ses positions dans l'étage. Le PE de la position zéro est appartient à la classe zéro, de même pour les positions suivantes. Le registre de passage pourvoit l'interconnexion d'un PE d'une classe vers le PE de la classe subséquent. C'est de cette manière que, par example, le PE_0 de l'étage 1 peut se communiquer avec le PE_0 de l'étage 2. L'ULA peut écrire ses sorties sur n'importe lequel des registres de passage. Si l'ULA n'écrit pas sur le registre de passage actuel, la valeur de ce registre est reçue du registre du PE équivalent de la classe antérieur.

Pendant que le registre de passage réalise les connexions inter-classes, la liaison parmi les PEs du même étage est assurée horizontalement via le réseau d'interconnexion. Dans chaque étage le réseau accepte des entrées de n'importe lequel des PEs de cet étage, en plus des valeurs des registres de passage des autres classes.

Comme PipeRench est placé comme un co-processeur, la bande passante entre lui, la mémoire principale et le processeur est limitée. Cela empêche l'implantation de certaines applications. A l'époque de la publication [41], les auteurs proposaient l'intégration du PipeRench au processeur, mais apparemment le projet a été interrompu avant d'atteindre cette étape.

1.4.2 Architectures à Grain Moyen

En fait, assez peu d'architectures reconfigurables à grain moyen existent. Pourtant, l'une des plus représentatives est la *Medium-Grain Reconfigurable Cell Array* - MGRCA.

MGRCA

Le MGRCA est d'une architecture reconfigurable à moyen grain taillée pour le traitement d'image [32]. Elle est composée par des cellules reconfigurables qui réalisent des opérations à 4 bits. Seize bus connectent chaque cellule à ses voisines.

Chaque cellule est composée par un commutateur, une interface et un cœur de calcul, comme présenté sur la Figure 1.11. Le cœur consiste en une matrice 4×4 d'éléments reconfigurables. Ces éléments sont organisés comme des mémoires 16×2bit d'accès aléatoire. Le cœur peut être configuré de deux manières différentes : la première optimisée pour des opérations mémoire ; la deuxième pour réaliser des opérations mathématiques. Les deux modes réalisent une opération par cycle d'horloge.

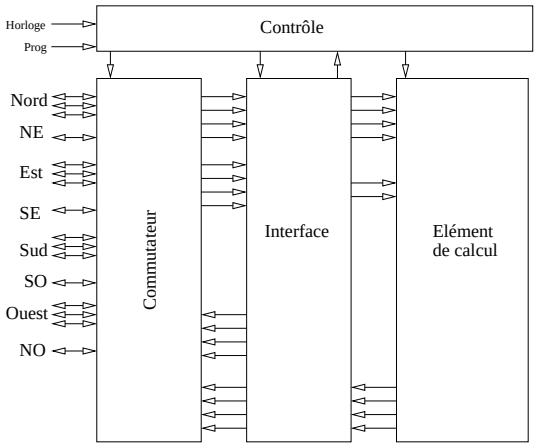


Fig. 1.11: Une cellule du MGRCA

Dans le mode mathématique (Figure 1.12), les cellules assemblées représentent une structure semblable à des multiplieurs du genre *carry-save*, mais peuvent réaliser d'autres opérations, tels que l'addition, la soustraction, le multiplexage, des opérations logiques et encore jouer le rôle d'une LUT 64 entrées $\times 4$ bits ou d'une machine à états.

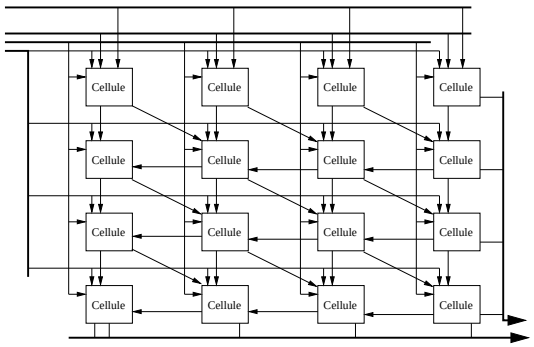


Fig. 1.12: Structure du MGRCA en mode mathématique

L'avantage de cette structure est la flexibilité située entre les architectures à gros grain et les

FPGAs. Par exemple, dans une architecture à gros grain, chaque élément de calcul possède un nombre fixe d'unités fonctionnelles. Cela limite son usage, mais aussi augmente la consommation. Pour réaliser une multiplication à 8 ou 12 bits dans une architecture à gros grain qui a un chemin de données de 16 bits, la consommation sera toujours la même que pour un multiplieur à 16 bits. D'un autre coté, la MGRCA implante un multiplieur selon la taille désirée.

Une autre caractéristique intéressante de l'architecture MGRCA c'est son schéma de correction de fautes, inclus dans la matrice reconfigurable. Cette correction de fautes est réalisée à partir d'un schéma de parité croisé (*cross-parity*), qui divise la mémoire 16×2 bits en deux éléments de 16 bits arrangés dans une configuration 4×4. Le schéma peut corriger jusqu'à une faute par unité de calcul et par cycle d'horloge. Des bits de parité sont stockés pour les lignes et pour les colonnes dans les deux unités de mémoire pendant les opérations d'écriture. Lors de la lecture de la mémoire, il est réalisé un XOR entre les bits de parité des lignes et des colonnes et les données dans les lignes et les colonnes, ce qui permet la détection d'une faute dans chaque ligne et dans chaque colonne. Si une faute est détectée dans la ligne et dans la colonne du bit de données réquisitionné, le système donc détermine que le bit a été corrompu. Alors le bit de donnée est inversé, envoyé vers la sortie, et stocké à nouveau dans la mémoire. Ce nouveau stockage corrige de manière permanente les bits fautifs.

Cet architecture montre qu'il y a encore intérêt à des approches à grain moyen, même si ce genre d'architecture n'est pas très répandu ni dans la communauté académique, ni dans l'industrie.

1.4.3 Architectures à Petit Grain

Les architectures reconfigurables à petit grain sont des dispositifs contenant des composants logiques et un réseau d'interconnexion programmable. Les unités logiques sont programmables afin d'émuler les fonctions logiques tels que AND, OR, XOR, NOT, ou même des fonctions plus complexes comme des decodeurs. Dans la plupart des dispositifs à petit grain, notamment les FPGAs, ces éléments programmables (blocs logiques) incluent des éléments de mémoire, qui vont des simples bascules jusqu'à des blocs complets de mémoire.

La hiérarchie d'interconnexion permet au concepteur de circuits une flexibilité très importante. Dans le cas des FPGAs, cette flexibilité reste au-delà de la manufacture du dispositif, et pour cela il est utilisé le terme "reconfigurable" à ce genre de dispositifs.

Les dispositifs reconfigurables à petit grain, comme les FPGAs sont généralement plus lents que les ASICs ou que les architectures à gros grain, et aussi ils consomment plus d'énergie. Cependant, ils offrent des avantages intéressants. Par rapport aux ASICs, les FPGAs ont un temps plus court pour arriver sur le marché, coûtent moins cher pour faibles quantités, et restent flexibles après la fabrication. En comparaison avec les architectures à gros grain, les FPGAs offrent un ensemble d'outils de configuration et de programmation beaucoup plus mûr que les premiers.

Les FPGAs ont été conçus au départ pour substituer les CPLDs comme "logique de colle"[6] pour les PCBs. Toutefois, due à l'évolution des technologies de synthèse de silicium, la taille et

[6] *Glue Logic.*

la complexité des FPGAs ont augmenté, les menant au rôle de compétiteurs sur le marché des systèmes sur puce. Aujourd'hui le domaine d'applications des architectures à petit grain s'étend aux algorithmes pour l'industrie spatial, imagerie médicale, vision par ordinateur, cryptographie, au delà d'une vaste étendue d'autres domains de recherche et de développement.

Dans la suite deux architectures à petit grain représentatives du marché industriel actuel sont présentées. Les FPGAs Virtex et FPSLIC ont été choisis pour ses capacités de reconfiguration dynamique, implantées de manière différent en chacun.

FPSLIC

Le FPSLIC [4] est une combinaison d'un FPGA de la famille AT40k fabriquée par ATMEL avec le processeur AVR (architecture RISC) et quelques ressources d'entrées et sortie et de mémoire, comme il peut être observé sur la Figure 1.13. Il s'agit donc d'une approche SoC (*System-on-Chip*), dont la flexibilité matérielle est garantie par un FPGA dynamiquement reconfigurable.

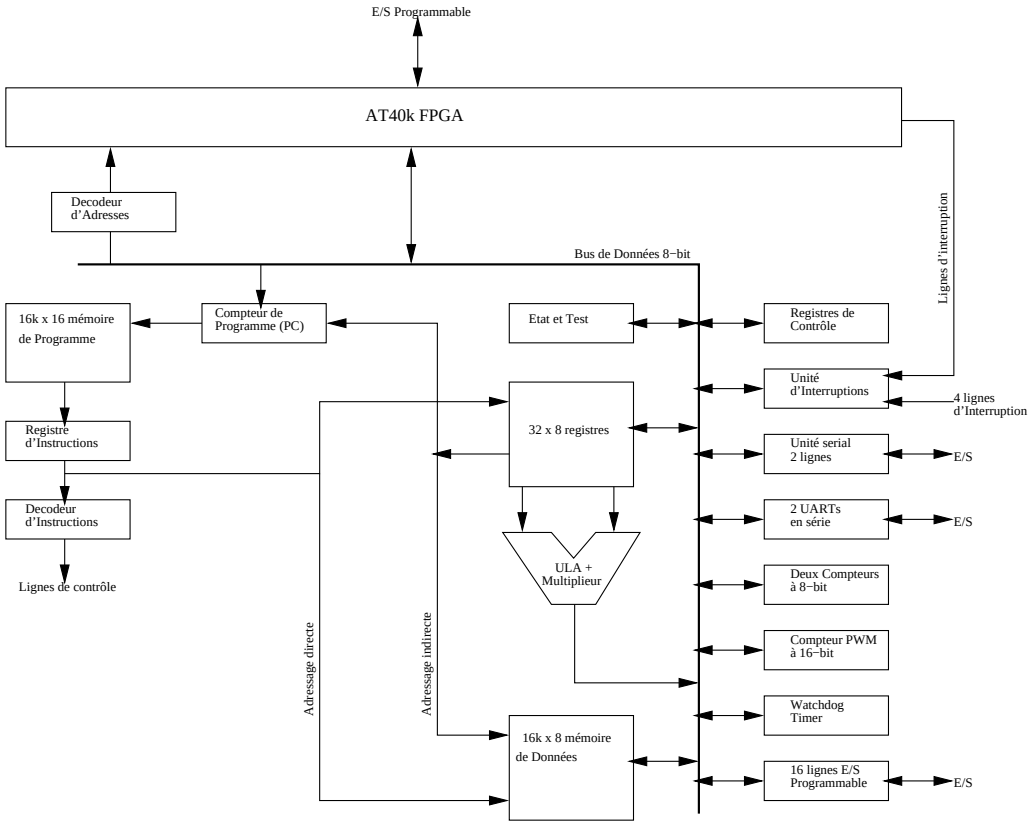
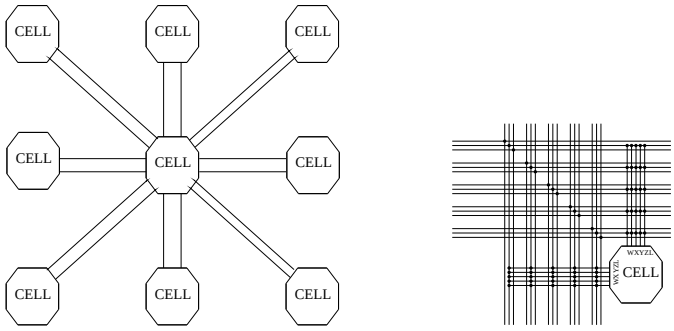


FIG. 1.13: Vue d'ensemble du SoC FPSLIC

La famille AT40K est composée par une matrice symétrique de cellules basées sur des LUTs à petit grain. Cette symétrie facilite la ré-allocation de ressources en cas de reconfiguration dynamique. La Figure 1.14 montre que chaque cellule peut communiquer soit par des liaisons directes (orthogonaux et diagonaux) aux cellules voisines (a) ; soit à travers de cinq bus horizontaux et cinq

bus verticaux (b).



(a) Connexions cellule à cellule (b) Connexions bus à bus

FIG. 1.14: Ressources de routage du FPSLIC

Des unités de connexion, dites répéteurs, sont placées après chaque quatre cellules, et ont pour but diviser les bus locaux dans des segments. Ces bus et cellules fonctionnent en deux dimensions. Une matrice 4×4 de cellules encadrée par ces répéteurs est appelée secteur. Dix bus locaux (cinq horizontaux et cinq verticaux) servent chaque ensemble de quatre cellules. Les deux bus par plan séparent deux secteurs. Ainsi, il existent quarante bus qui passent à travers des secteurs, fournissant des vastes ressources de routage.

LES FPGAs AT40k contiennent des blocs de mémoire SRAM *Static Random Access Memory* qui sont distribués au sein de la matrice. Les blocs de SRAM 32×4-bit sont localisés à l'intersection des secteurs et sont programmables pour fonctionner de façon synchrone ou asynchrone. De plus, ces blocs peuvent être utilisés comme des mémoires simples ou double ports.

Par rapport à la reconfiguration dynamique, les FPGAs de la famille AT40k sont reconfigurables via la commutation de contexte. Dans ce genre de reconfiguration l'application est découpée en des modules logiques que sont stockés dans une mémoire de configuration. En supposant qu'à chaque instant il y a juste une portion du FPGA qui réalise des opérations, la portion non-utilisée peut servir à recevoir des nouveaux modules de l'application (Figure 1.15).

La commutation de contexte dans la famille AT40k est implantée comme le cache dans les systèmes numériques tels que les ordinateurs. Dans ces systèmes, les données les plus utilisées sont stockées dans des mémoires plus rapides (*cache*), pendant que les données moins souvent nécessaires sont gardées dans des mémoires plus lentes, tels que la DRAM. Dans le cas de la famille AT40k, seulement une partie de la logique est implantée dans le FPGA, pendant que les fonctions les moins utilisées restent dans une mémoire auxiliaire (dite *cache logic*). Au fur et à mesure que nouvelles fonctions se font nécessaires, elles sont chargées dans la *cache logic*, pouvant compléter, modifier ou substituer les fonctions déjà implantées dans le FPGA.

La possibilité de réaliser la reconfiguration par la commutation de contexte permet l'utilisation de ce FPGA pour des applications nécessitant un matériel "évolutif", comme des réseaux de neurones et algorithmes génétiques.

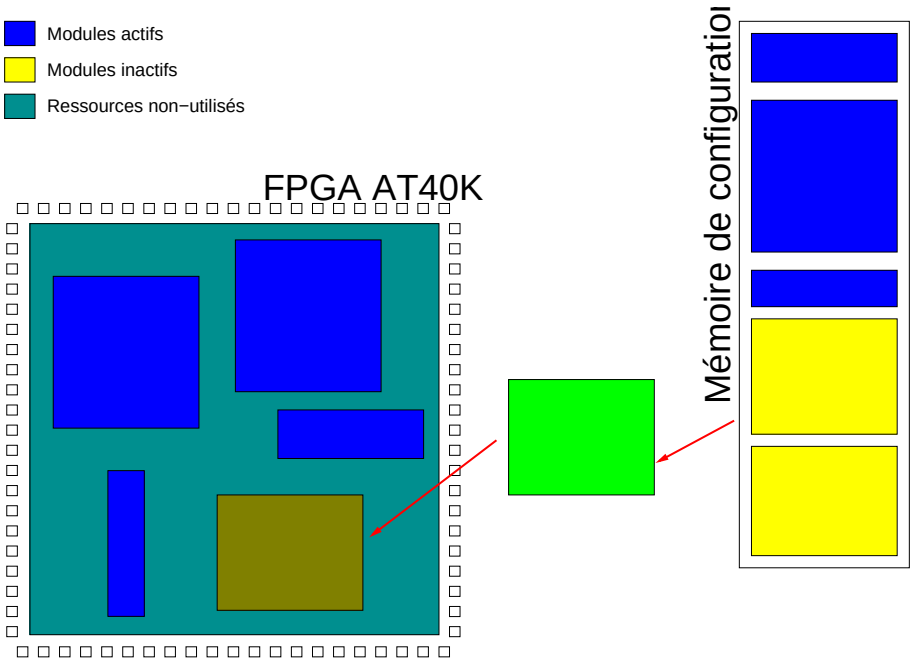


FIG. 1.15: Schéma de reconfiguration par commutation de contexte du FPGA AT40k

Virtex

Les dispositifs Virtex-4 [101] sont des FPGAs à petit grain (LUTs à 4 entrées organisés dans des CLBs), avec des éléments de calcul à gros grain embarqués (DSP, processeur PowerPC), en plus des blocs de mémoire et ressources d'entrée et de sortie (avec un transceiver à 11-Gigabit inclus), ce qui fait de cette architecture aussi un SoC, conforme la Figure 1.16. Ce dispositif comporte aussi un système de sécurisation du vecteur de configuration basé sur l'algorithme AES (*Advanced Encryption Standard*).

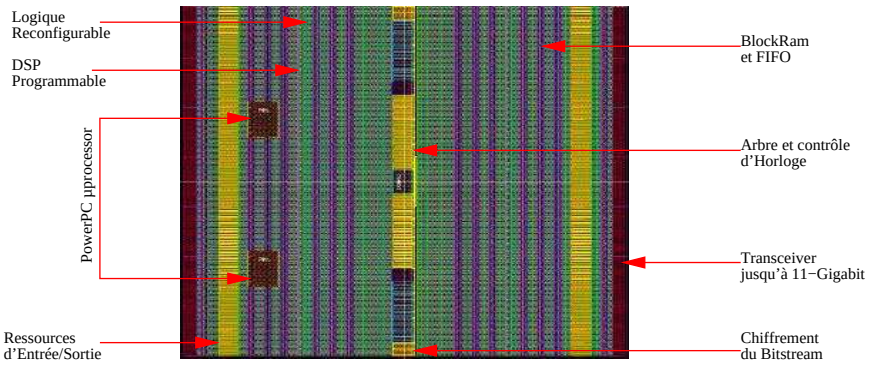


FIG. 1.16: Vue d'ensemble du FPGA Virtex-4

La fonctionnalité des circuits Virtex est déterminée à travers un vecteur de configuration, appelé *bitstream*. Les fichiers de configuration contiennent un mélange de commandes et de données qui peuvent être lues et écrites par l'interface de configuration.

L'organisation interne des FPGAs Virtex est en colonnes, qui peuvent être lues ou écrites individuellement. Cette caractéristique permet la reconfiguration partielle du dispositif, puisque la modification d'une colonne n'entraîne pas forcément le changement dans les colonnes adjacentes.

Cette régularité dans la distribution des éléments de calcul qui composent l'architecture du FPGA Virtex permet des actions de ré-allocation et defragmentation, opérations essentielles pour la reconfiguration dynamique [23].

Étant l'élément reconfigurable de base, les CLBs pourvoient la logique synchrone et combinatoire, mais aussi peuvent fonctionner comme des mémoires (*LUTRAM*) ou encore comme des registres à décalage. Chaque CLB possède deux ensembles de fonctions identiques, dénommés *slices*. La Figure 1.17 exhibe la structure interne d'une CLB, où chaque *slice* contient deux LUTs (F et G), deux bascules, un *buffer* à trois états, un circuit de propagation rapide (*carry*), en plus de ressources de routage. Chaque LUT à quatre entrées est composée par 16 bits de configuration.

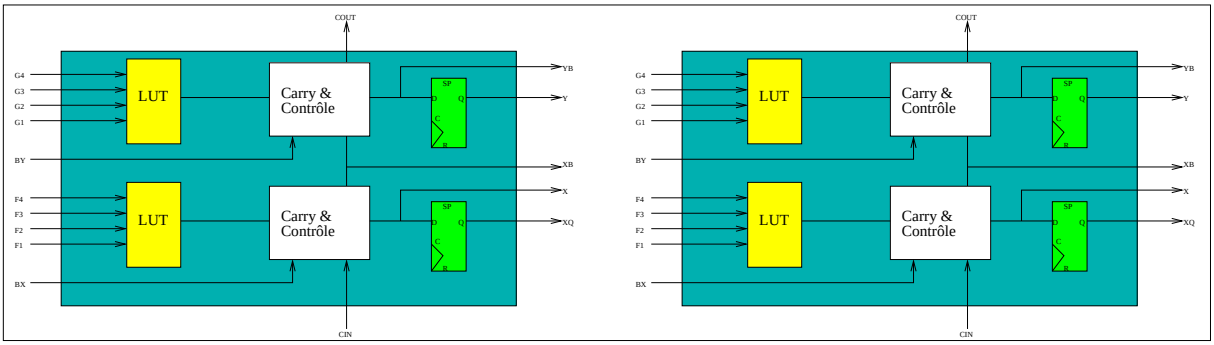


FIG. 1.17: Architecture interne simplifiée d'une CLB

Les blocs d'entrée et sortie (plus souvent appelés *IOBs - Input / Output Blocks*) sont l'interface entre les broches et la logique configurable interne. Les IOBs sont optimisées pour des applications synchrones. Ces optimisations comprennent *deskewing*, sérialisation et de-sérialisation de données, division d'horloge, et ressources pour les horloges locaux dédiés.

En ce qui concerne la mémoire embarquée (*BlockRAM*), il existent des modules de 18K-bit double-ports. De plus, les *BlockRAMs* contiennent des FIFOs *Firs In-First Out* - queues programmables.

Un système de gestion d'horloge facilite les solutions pour compenser les retards dans la distribution de l'horloge. Le FPGA contient aussi des ressources pour multiplier ou diviser l'horloge.

Puisqu s'agit d'un Soc, le circuit possède aussi des éléments de calcul avec différents tailles de grain. En plus des LUTs (petit grain), les Virtex ont :

- **grain moyen** : Multiplieurs à 18-bit, additioneur intégré, et un accumulateur à 48-bit.
- **gros grain** : DSPs et deux processeurs PowerPC.

Pour conclure, tous les composants dans la Virtex-4 utilisent le même schéma d'interconnexion et le même accès à la matrice de routage globale. Les modèles de timing sont partagés, augmentant la possibilité de prévision de performance pour les systèmes opérant à grande vitesse.

ARDOISE

Le projet ARDOISE (Architecture Reconfigurable Dynamiquement Orientée Image Signal Embarquable) vise à démontrer l'intérêt des architectures matérielles à reconfiguration dynamique (RD) dans les systèmes associant contraintes temps réel fortes et versatilité des traitements. À l'heure actuelle, ARDOISE est la seule architecture développée en France, dont un prototype a effectivement été réalisé et testé en situation réelle, qui mette en oeuvre ce concept [34].

L'idée de base de cette architecture est d'enchaîner des algorithmes nécessaires au traitement d'images dans la même structure matérielle, en utilisant la reconfiguration à petit grain à quelques reprises, pendant le calcul d'une seule image. Cela équivaut à s'en servir d'un FPGA afin d'exécuter une séquence d'algorithmes selon un agenda pre-établi. Ce concept permet l'implantation d'applications sur FPGA avec une performance raisonnable, tout en gardant la flexibilité d'un microprocesseur.

L'architecture d'ARDOISE est basée sur trois modules identiques (Figure 1.18). Chacun contient un FPGA à $45k$ portes connectés à deux mémoires locales qui stockent les résultats intermédiaires. Ces blocs identiques sont appelés MARD (*Modules ARDoises*) et sont contrôlés par le module GCC (*Gestion des Configurations et des Communications*). Le GCC permet à ARDOISE de fonctionner soit en *stand alone*, soit en liaison avec un DSP. Le FPGA, la mémoire et la connexion à une liaison en série autorisent la configuration du FPGA et le chargement de la mémoire flash à partir d'un PC.

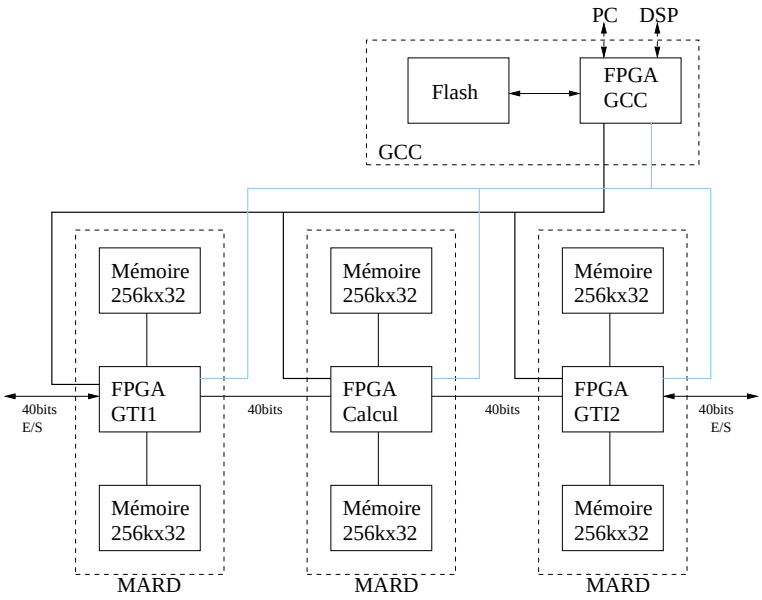


FIG. 1.18: Vue d'ensemble de l'architecture ARDOISE

Chaque module MARD possède les composants suivants :

- Deux bus de 40 bits qui assurent les échanges entre modules MARD et les entrées et sorties ;
- Deux bus d'accès aux blocs mémoires de 32 bits de données ;

- Un bus de configuration connecté au module GCC ;
- Un bus connecté à l'ensemble des modules.

Les MARDS GTI1 et GTI2 pourvoient une interface avec les entrées et sorties du système, ce qui permet la de-synchronisation du module central de calcul (TGV) et le système d'acquisition d'images. Les différents traitements sont changés dans le module central. Les résultats partiels sont stockés dans les mémoires locales dans différents modèles, selon l'action de reconfiguration ou de calcul.

Bien que cette approche permet l'extension des modules pour augmenter la puissance de calcul, l'intérêt de ses auteurs est de plutôt exécuter l'ensemble des traitements dans un module unique, afin qu'ils puissent étudier la validité d'utilisation de la reconfiguration dynamique.

1.5 Tendances

Le concept d'architecture reconfigurable a vu le jour dans les années 60, quand Gerald Estrin a écrit un papier proposant un ordinateur qui aurait été constitué d'un processeur et d'une partie "flexible" [37]. Le processeur contrôlerait le comportement de la partie flexible. Le matériel reconfigurable pourrait donc réaliser des tâches spécialisées. Une fois la tâche accomplie, le matériel reconfigurable aurait pu être modifié pour réaliser une autre tâche. L'idée de base était de combiner la flexibilité logicielle et la performance matérielle. Malgré le caractère visionnaire de cette idée, elle n'a pas pu aboutir, car les technologies de l'époque ne permettaient pas d'implanter une telle machine.

Une vingtaine d'années après la première machine reconfigurable a été commercialisé. Il s'agissait de l'Algotronix CHS2X4, en 1991 (Figure 1.19).

Fig. 1.19: Algotronix CHS2x4, la première machine reconfigurable

La deuxième moitié des années 90 a vu l'explosion de l'utilisation des circuits reconfigurables notamment les FPGAs. Basées sur ces circuits, plusieurs architectures reconfigurables ont été proposés. Comme dans le CHS2X4, le plus souvent ces architectures étaient composés par des circuits reconfigurables au sein d'une carte PCB, avec un réseau fixe d'interconnexion. Les exemples les plus notables de cette première génération des architectures reconfigurables sont le PRISM [2], le DECPerle [12] et le Splash, qui peuvent être vus dans la Figure 1.20 (a).

Cette façon de réaliser un système reconfigurable entraînait des problèmes tels que le manque

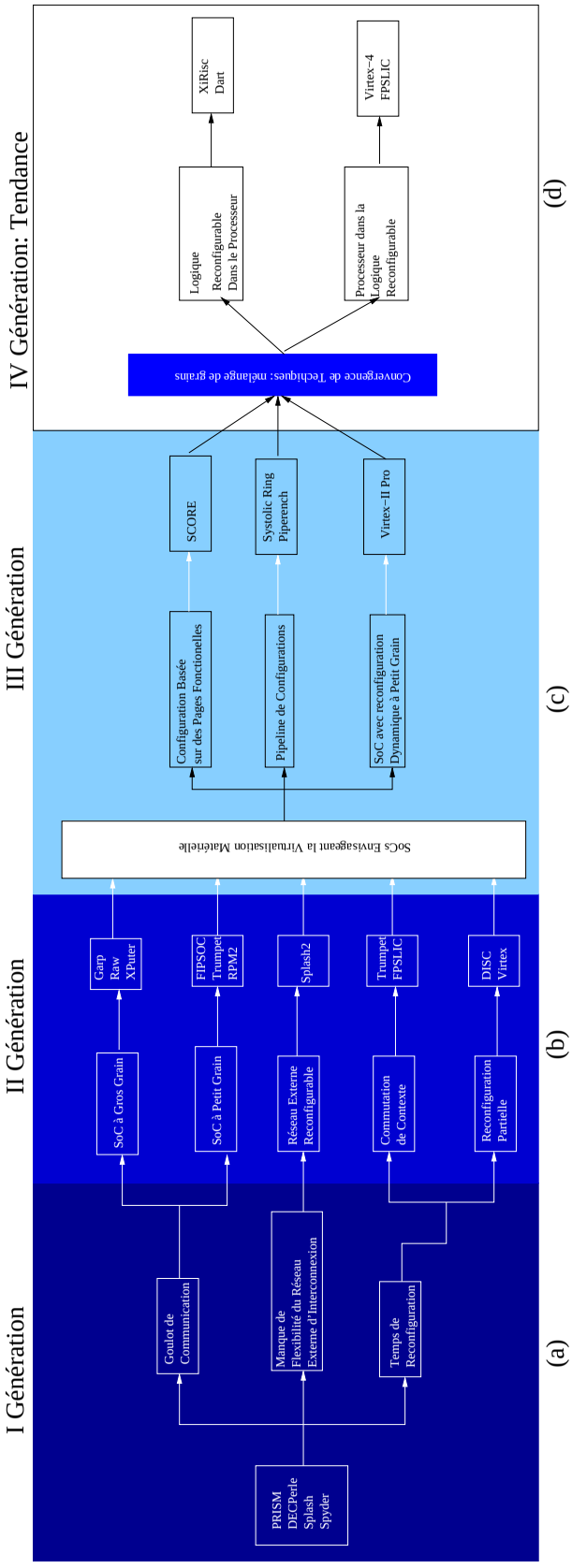


FIG. 1.20: Les différents générations d'architectures reconfigurables et tendances pour un futur proche

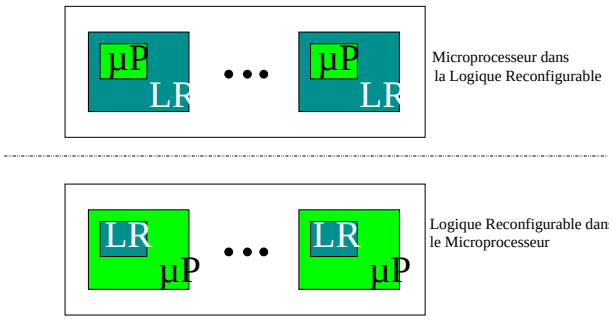


FIG. 1.21: Deux combinaisons possibles au sein d'une architecture reconfigurable.

de souplesse du réseau de communication, le temps de reconfiguration et principalement, le goulot d'étranglement entre la partie reconfigurable et le processeur hôte.

Avec la croissance de la capacité d'intégration, à la fin des années 90 le concept de SoC a été mis en place. A ce moment, plusieurs options de reconfiguration sont apparues, notamment celles qui opposaient les approches à petit grain de celles à gros grain ; et celles qui confrontaient la reconfiguration par commutation de contexte à la reconfiguration partielle/dynamique. A cette époque sont nés les architectures Garp [17], Raw [96], KressArray [45], FIPSOC [93], AT40K [3], Virtex[100], Disc, parmi tant d'autres. La Figure 1.20 (b) place ces architectures dans une deuxième génération d'architectures reconfigurables.

Après cette profusion d'idées dans le domaine du reconfigurable, il a eu lieu une convergence vers un objectif commun : la virtualisation du matériel, tout en restant raisonnable par rapport à la densité fonctionnelle. La troisième génération (Figure 1.20 (c)) était dans sa majeure partie composée par des architectures à gros grain, comme le Systolic Ring [89], le Dart [29] et le Score [19]. Pourtant, dispositifs à petit grain, comportant des éléments à grain variable comme le Virtex et le FPSLIC continuent à se développer.

Étant donné qu'il existe des applications orientées flots de données qui demandent un traitement au niveau mot, et que cependant restent aussi des applications comme la cryptographie où les calcul exigent beaucoup de contrôle et où il y a des opérations au niveau bit, la tendance logique c'est le projet d'architectures reconfigurables à grain mixte.

Cette tendance ouvre deux voies pour l'intégration entre petit grain et gros grain dans une même puce, afin de ne pas générer des goulots d'étranglement dans la communication entre FPGA et microprocesseur. La Figure 1.21 montre qu'il est possible soit d'intégrer de la logique reconfigurable dans le chemin de données d'un processeur générique (a), soit de submerger le processeur dans une mer d'éléments logiques reconfigurables (b). Un exemple de la première approche c'est l'architecture XiRisc ; pour le deuxième il est possible de citer le Virtex-4. Ce genre d'approche influence une quatrième génération d'architectures reconfigurables (Figure 1.20 (d)), caractérisée par la convergence de taille de grain et intégration de systèmes sur puce.

Pourtant, ce que les concepteurs d'architectures reconfigurables ne sont pas encore parvenus à réaliser, c'est qui en plus d'un dispositif matériel fiable et performant, il faut avoir un modèle de programmation associé et des outils de CAO pour supporter le développement des applications

pour ces systèmes. A défaut de cela, l'histoire montre que les architectures reconfigurables sont censées disparaître, comme le cas des FPGA Clay (National [71]) et la famille XC6200 (Xilinx [99]), et les systèmes Splash et Prism, pour citer aussi des systèmes académiques.

1.6 Le Reconfigurable pour la cryptographie

L'idée d'utiliser une architecture reconfigurable pour la cryptographie est née d'abord de la nécessité de flexibilité de ce domaine d'application. Des protocoles de cryptographie, ou même des algorithmes peuvent changer dans le temps. Donc, afin de conserver le matériel en conformité aux nouveaux besoins protocolaires, la reconfigurabilité est nécessaire.

Jusqu'à ce moment l'association entre la cryptographie et le reconfigurable se faisait au travers des implantations dans des FPGAs, et dans la majorité des cas, les possibilités de reconfiguration dynamique n'étaient pas envisagées. De plus, Au delà d'être souple[7], un circuit cryptographique doit être robuste contre certaines types d'attaques, notamment contre celles basées sur l'analyse de consommation.

Des dispositifs reconfigurables issus du milieu industriel n'adressent pas ce problème. Les FPGAs très récemment arrivés sur le marché tels que le Virtex-5 (Xilinx) et le ProAsic3 (Actel) se contentent de fournir des solutions pour la sécurité du bitstream, au travers d'un IP câblé implantant l'algorithme de cryptage AES. Dans le cas du ProAsic3, la possibilité de stocker la clé dans une mémoire non volatile au sein du circuit (par le biais d'une mémoire *flash*) accroît la sécurité.

Pourtant ces avancées dans le monde industriel ne concernent pas ni la sécurité face aux attaques par canaux cachés, ni ne fournissent pas des ressources supplémentaires à l'implantation de systèmes cryptographiques. Il manque encore aux dispositifs reconfigurables des caractéristiques capables d'assurer un certain niveau de sûreté.

Dans [38] Fischer et alii proposent un dispositif reconfigurable susceptible d'atteindre plusieurs niveaux de sécurité. Le terme Crypto-FPGAs est alors défini comme étant un dispositif reconfigurable possédant quelques caractéristiques additionnelles de sécurité et des opérandes cryptographiques accessibles par l'utilisateur. La différence fondamentale entre les Crypto-FPGAs et les dispositifs reconfigurables "classiques" est que les FPGAs existants possèdent des ressources cryptographiques afin de protéger les fichiers de configuration, mais ces ressources ne sont pas disponibles aux utilisateurs.

La macro-architecture du Crypto-FPGA proposé peut être analysée à partir de la Figure 1.22. Les tâches sont distribuées soit dans des zones dites "vertes" (gris clair dans la Figure)[8] ; soit dans des zones rouges (gris foncé dans la Figure 1.22)[9]. Les tâches concernant des données sensibles sont réalisées dans les zones rouges.

La macro-architecture représente un ensemble de blocs sécurisés et de blocs standards. Les blocs standards (zone verte) peuvent implanter les fonctions cryptographiques les plus répandues.

[7] Ici la souplesse signifie capacité d'être modifié dans le temps.
[8] Une zone verte équivaut à une zone non protégée
[9] Une zone rouge équivaut à une zone hautement protégée

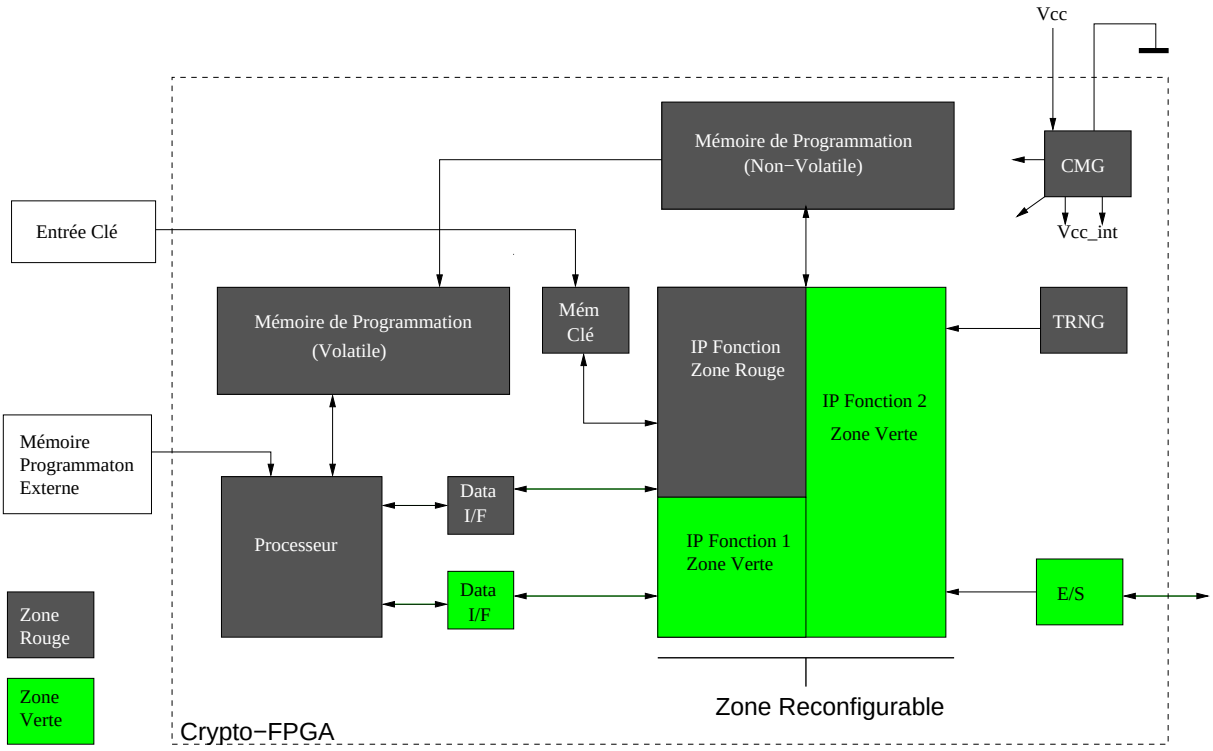


FIG. 1.22: Macro-architecture d'un dispositif reconfigurable pour la cryptographie.

Des fonctions de sécurité supplémentaires sont inclues dans la zone rouge, et peuvent être, par exemple, une mémoire spécifique pour la clé cryptographique, un générateur de nombres aléatoires, un co-processeur pour les opérations arithmétiques modulaires, capteurs de température, détecteurs d'intrusion, et cætera.

Le circuit CMG[10] [64], [63] est un exemple d'un module de la zone rouge. C'est un circuit analogique capable de masquer la signature de consommation d'un circuit cryptographique, empêchant de cette manière les attaques par analyse de courant.

Il est possible donc de conclure que les dispositifs actuellement dans le marché ne proposent pas des fonctions cryptographiques accessibles par l'utilisateur, ni sont sécurisés contre les attaques les plus répandues. Les idées proposées en [38] amènent à une réflexion au sujet d'une architecture résistante aux attaques par canaux cachés.

Cependant, un autre approche envisage d'associer la reconfiguration dynamique à une arithmétique spécifique, afin d'accroître la sécurité contre les attaques par canaux cachés. Cette association permet aux systèmes cryptographiques d'être flexibles tout en restant sécurisés. Cette idée est complémentaire à celle proposé en [38], car Fischer propose une approche à petit grain, alors qu'il est proposé, dans le cadre de cette thèse, une solution à gros grain. La décision pour une architecture à gros grain suit les justificatifs donnés par les auteurs du Systolic Ring et du DART, ainsi que les tendances montrées dans la Section 1.5. Le projet d'une architecture à gros grain pour la crypto-

[10]Le Current Mask Generator est exposé en détails dans la Section 3.2.2

graphie, résistante aux attaques par analyse de consommation est une étape dans la direction vers la proposition d'une future architecture multi-grain. Les prochains chapitres montrent les bases et l'intérêt de cette approche.

CHAPITRE 2

Introduction à la Cryptographie

"It is insufficient to protect ourselves with laws ; we need to protect ourselves with mathematics". ***Bruce Schneier***

Étant donnée que le sujet cette thèse concerne la cryptographie et les attaques contre des circuits cryptographiques, avant de discuter des attaques et sur les méthodes de sécurisation des dispositifs implantant ces algorithmes, il faut réviser les concepts mathématiques qui font la base de la cryptographie.

Le premier but de ce chapitre est de donner un aperçu des techniques de cryptographie modernes, tout en focalisant sur les algorithmes de cryptographie DES [73] et RSA [82], les plus répandus au monde.

L'arithmétique nécessaire à la cryptographie moderne est basée sur la théorie des nombres, qui est, à son tour une branche de la mathématique qui s'occupe des propriétés des nombres entiers positifs, c'est dans ce contexte que a été mené l'exploration au cour de cette thèse. Un axe de la théorie des nombre particulièrement intéressant à la cryptographie concerne l'arithmétique modulaire. C'est dans ce domaine où se trouvent la plupart des algorithmes modernes de cryptographie.

2.1 Introduction à la cryptographie

La cryptographie c'est une discipline qui étudie les méthodes pour assurer le secret et l'authenticité des messages. Le terme "cryptographie" vient du grec "*kriptós*" (caché) et "*gráphein*" (écrite).

La cryptographie concerne la transformation d'un message (texte, image, chiffres) intelligible vers un message codé, incompréhensible à tous sauf pour les détenteurs d'une code de déchiffrement.

Crypter est ce processus de transformation, et décrypter son inverse. Il est d'usage appeler ce processus de cryptage. Les fonctions de cryptage sont des algorithmes cryptographiques. Le code permettant chiffrer et déchiffrer un message codé se pose sur une clé cryptographique. Cette clé est un paramètre qui doit être variable et maintenu en secret (sauf dans certains algorithmes où part de la clé reste exposé - voir Section 2.2.2). Si auparavant tout la sécurité d'un système cryptographique résidait dans la clé, aujourd'hui il existe d'autres éléments qui composent la robustesse de tels systèmes. Plusieurs algorithmes, dont ceux de hachage, protocoles de authentification et procédures d'opérations de cryptage et décryptage aident à assurer la fiabilité des systèmes de sécurité.

La cryptographie a pour principales fonctions les quatre points suivants :

1. **Confidentialité** : Seulement le destinataire autorisé doit être capable d'extraire le contenu du message de son état crypté. Par ailleurs, l'obtention de l'information à propos du contenu du message (comme par exemple la distribution statistique des caractères) ne doit pas être possible.
2. **Intégrité** : Par l'analyse d'intégrité le destinataire peut déterminer si le message a été modifié pendant la transmission.
3. **Authentification** : Le destinataire doit avoir la capacité d'identifier l'auteur du message, aussi bien que de savoir si c'est l'auteur présumé qui a en effet envoyé le message.
4. **Non-reniement** : L'expéditeur du message ne doit pas pouvoir nier que c'est lui l'auteur.

Au-delà de ses buts inhérents, l'implantation d'un algorithme de cryptographie est jugé selon le niveau de sécurité qu'il peut atteindre. Selon Claude Shannon [92], un algorithme de cryptographie peut être classifié selon trois catégories :

1. **Sécurité Calculatoire** : Un algorithme est dit sûr dans le sens de la théorie de la complexité[1] si la meilleure méthode pour le casser nécessite N opérations, où N s'agit d'un très grand nombre. Malheureusement il n'existe pas un système cryptographique qui permet ce niveau de sécurité, car sa sécurité calculatoire est étudié vis-à-vis d'un certain type spécifique d'attaque, ce qui ne garantit pas celle par rapport à d'autres types.
2. **Sécurité inconditionnelle** : Un procédé est inconditionnellement sûr s'il ne peut être cassé, même avec une puissance de calcul infinie. Actuellement il n'exist qu'un algorithme à sécurité

[1] La théorie de la complexité s'intéresse à l'étude formelle de la difficulté des problèmes en informatique. Elle se distingue de la théorie de la calculabilité qui s'attache à savoir si un problème peut être résolu par un ordinateur. La théorie de la complexité se concentre donc sur les problèmes qui peuvent effectivement être résolus, la question étant de savoir s'ils peuvent être résolus efficacement ou pas en se basant sur une estimation (théorique) des temps de calcul et des besoins en mémoire informatique.

prouvée (inconditionnelle). Il s'agit du chiffre de Vernan[2], aussi appelé masque jetable. Ce chiffrement est le seul qui soit théoriquement impossible à casser, même s'il présente d'importantes difficultés d'utilisation pratique. La preuve de sa sécurité est démontrée para Shannon dans le même article [92].

Soit M le message en clair. Si E est la fonction de cryptage, en appliquant E sur M le message chiffré C est obtenu :

$$E(M) = C \tag{2.1}$$

Dépendant du système utilisé, la fonction de décryptage peut être la même fonction $D = E$, mais en tout cas, le décryptage est donné pour :

$$D(C) = M \tag{2.2}$$

Et comme le but de la cryptographie est de crypter et de décrypter un message, l'identité suivante doit être satisfaite :

$$D(E(M)) = M \tag{2.3}$$

D'une façon plus formelle, un système de cryptographie peut être précisé comme dans la définition 2.1 [94] :

Définition 2.1: *Un système cryptographique est un quintuplet* (M, C, K, E, D) *tel que :*

1. M *est un ensemble fini de blocs de textes non-cryptés.*
2. C *est un ensemble fini de blocs de textes cryptés.*
3. K *est un ensemble fini de blocs de textes non-cryptés.*
4. *Pour tout* $k \in K$ *il existe une règle de cryptage* $e_k \in E$ *et une règle de décryptage correspondante* $d_k \in D$. *Chaque* $e_k : P \to C$ *et* $d_k : C \to P$ *sont des fonctions telles que* $d_k(e_k(m)) = m$ *pour tout texte clair* $m \in M$. ∇

Le quatrième item de la définition 2.1 précise que si un texte en clair m est crypté en utilisant la fonction e_k, ayant comme résultat un texte crypté c, est ensuite décrypté en utilisant d_k, il est retrouvé à la fin le texte m.

Le schéma montré dans la figure 2.1 illustre le procès de cryptage et décryptage.

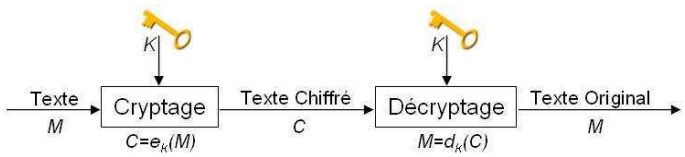


FIG. 2.1: Schéma basique de cryptage et décryptage

[2]Inventé par Gilbert Vernam en 1917

2.2 Types d'algorithmes de Cryptographie

Dans la cryptographie moderne toute sécurité est basée sur la clé (où les clés), et non dans les détails des algorithmes. Cela signifie qu'un algorithme peut être publié et analysé, mais la clé doit être protégée [90].

Il existe deux axes majeurs d'utilisation de clés : les systèmes cryptographiques symétriques (Sous-Section 2.2.1) ; et ceux asymétriques (Sous-Section 2.2.2).

2.2.1 Cryptographie à Clé Symétrique

Les algorithmes à clé symétrique concernent les méthodes de cryptage dans lesquels l'expéditeur et le récepteur d'un message utilisent la même clé, laquelle a été transmise via un canal sécurisé. Ceci était la seule méthode de cryptage connue jusqu'à 1976.

En effet, la plupart des algorithmes symétriques utilisent des clés identiques, mais certains algorithmes utilisent des clés de cryptage et de décryptage différentes, mais la deuxième peut être calculé de façon simples à partir de la première et vice-versa. Ces systèmes sont aussi connus comme étant à clé secrète, à clé privée où à clé unique.

Le fonctionnement de ce genre d'algorithme dépend du secret de la clé, et par conséquent, présume l'existence d'un canal sécurisé pour le transfert de la clé, à défaut de quoi la sécurité de tout système est compromise. Le cryptage et le décryptage d'un algorithme cryptographique à clé privée est donné par :

$$E_k(M) = C$$

$$D_k(C) = M$$

La figure 2.2 illustre ce procédé.

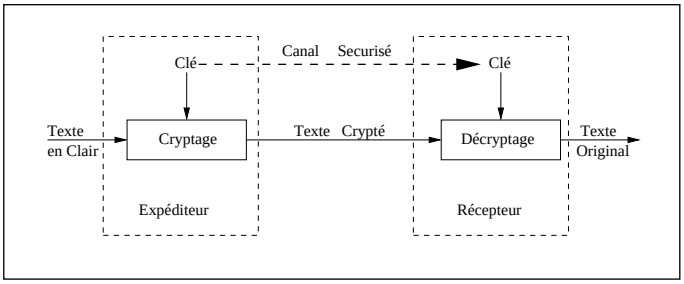


FIG. 2.2: Schéma basique de cryptage et décryptage

Catégories

Algorithmes symétriques sont divisés en deux catégories principales. Quelques uns opèrent sur un seul bit (ou octet) à chaque fois ; il s'agit des algorithmes de cryptage par flot. D'autres agissent

sur le texte en clair par ensembles de bits. Ces ensembles sont appelés blocs, d'où la dénomination cryptage par bloc. Dans les algorithme de cryptage par blocs les plus utilisés, ces blocs peuvent varier de 64 à 128bits. Quelques exemples d'algorithmes symétriques par flot sont le RC4[49] et le SEAL[83]. Les algorithmes à clé secrète les plus répandus sont le DES [73], le RC6[81] et le AES[26].

Principe de fonctionnement

La plupart des algorithmes symétriques modernes dérivent d'une méthode attribuée à Horst Feistel, utilisant des itérations répétées (*rounds*). Un *round* c'est la permutation des bits du bloc qui va être crypté et qui sont mixées avec quelques bits de la clé. Soit B le bloc de bits qui sera crypté. B est divisé dans deux parties P_1 et P_2. Pendant que P_1 reste inchangé, P_2 est additionné (où 'XORé') à une fonction de hachage à sens unique appliquée à P1. Les deux résultats sont alors echangés à nouveau : $P_1, P_2 \mapsto P_2', P_1$; de tel façon que :

$$P_2' = P_2 + f(P_1, k)$$

Étant donné que la sortie de l'itération accède encore à la valeur P_1, et que l'addition est une opération réversible, le *round* peut être défait, quelque soit la fonction f. Même si un *round* seul consiste en une opération non sécurisée, la répétition des *round* avec des sous-clés différentes incrémente de façon considérable la sécurité du système. Pour décrypter il suffit d'appliquer les itérations dans le sens inverse, avec les sous-clés aussi dans le reverse.

L'exemple classique d'un algorithme à clé privée utilisant cette méthode est le DES [73].

Pourtant même les algorithmes que n'utilisent pas la structure de Feistel utilisent des tours afin de promouvoir la diffusion et la confusion[3], comme c'est le cas pour le AES [26].

Performance

Par rapport à la performance, les algorithmes symétriques nécessitent d'une capacité de calcul moins intensive que les algorithmes asymétriques. Cela occasionne une rapidité de cryptage et de décryptage atteignant centaines ou milliers de fois supérieure à celle des algorithmes à clé publique.

Pourtant les algorithmes symétriques ont le désavantage de nécessiter le partage de la clé secrète entre les deux points qui veulent se communiquer. Par exemple, dans une population de n individus, pour assurer une communication entre chacun des membres, il est nécessaire d'avoir $n \times \frac{n-1}{2}$ clés. Actuellement, l'échange de clés secrètes se donne à travers d'un protocole d'échange basé sur un algorithme de cryptographie asymétrique. Donc, dans une session de communication, il est utilisé un algorithme à asymétrique (lent) pour réaliser le partage de la clé privée.

Les références [90] et [58] présentent plus dans le détail les algorithmes symétriques.

[3]Comme décrit Shannon en [92], les principes de sécurité pour un algorithme sont la confusion, la diffusion et l'addition d'une clé.

2.2.2 Cryptographie Asymétrique

Le principal inconvénient des algorithmes privée c'est qu'ils usent la même clé pour crypter et pour décrypter. Cela implique dans la nécessité d'un canal de communication sûr entre l'expéditeur et récepteur du message. L'algorithme est donc vulnerable durant la phase de transport de la clé.

En 1976 a été publié un article proposant un schéma de cryptographie faisant usage de deux clés distinctes (pourtant intrinsèquement liées) : l'une pour crypter et l'autre pour décrypter. Même s'il y a un rapport entre les clés, la possession d'une clé ne permet pas de calculer l'autre.

Aussi appelés algorithmes à clé publique, son nom provient du fait qu'une des clés générées doit être effectivement rendue publique : n'importe qui peut s'emparer de la clé publique et crypter un message ; mais seulement une personne spécifique peut le décrypter. Dans ces systèmes, la clé publique est la clé de cryptage et la clé privée est la clé de decryptage. Le procédé de cryptage est dénoté pour :

$$E_{kpu}(M) = C$$

$$D_{kpr}(C) = M$$

La figure 2.3 illustre ce procédé.

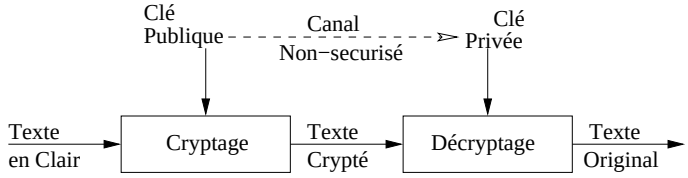


FIG. 2.3: Schéma basique de cryptage et décryptage

Fonctionnement

Du point de vue conceptuel, un algorithme à clé publique peut être imaginé comme une fonction à sens unique avec une trappe [4]. La fonction E_{kpu} doit être facile à appliquer. Pourtant de décryptage ne doit être possible que pour la personne ayant la clé privée *kpr* pour utiliser la fonction D_{kpr}. Dans le contexte de la cryptographie, il est désirable que E_{kpu} soit un fonction à sens unique injective, afin que le décryptage puisse avoir lieu. Toutefois, il n'existe que des fonctions injectives *supposées* être à sens unique : aucune n'est prouvée comme tel. Par exemple, en supposant que n soit le produit de deux grands nombres premiers p et q, et que b soit un entier naturel supérieur à 2, il est possible définir $f : \mathbb{Z}n \rightarrow \mathbb{Z}n$ par :

$$f(M) = [M^b]_n$$

[4]Une trappe s'agissant d'un "Piège à bascule installé au-dessus d'une fosse", emprunte son image à une fonction dont seuls ceux qui connaissent le secret de la trappe peuvent "sortir du piège". Ce concept est traité dans la Section 2.5.1

[5] Considérant la nécessité d'une trappe pour pouvoir décrypter le message, dans le cas proposé cette fonction est l'inverse f_{-1}, tel que $f(M) = [M^a]_n$, pour une valeur correcte de a. La trappe ici c'est une méthode pour retrouver a en connaissant b, qui utilise la factorisation de n.

Sécurité

Il est important de noter qu'un système cryptographique à clé publique ne peut jamais être considéré comme étant inconditionnellement sûr, car un attaquant qui observe un texte crypté C peut crypter chaque texte clair possible M avec la règle de cryptage E_{kpu} jusqu'à ce qu'il trouve l'unique X tel que $C = E_{kpu}(M)$. Ce M est le résultat du décryptage de C. Donc la seule sécurité possible dans ce genre d'algorithme c'est la sécurité calculatoire. Pourtant restent largement employés, car les algorithmes à clé publique permettent d'accomplir les buts de la cryptographie en ce qui concerne l'authentification et la non-répudiation, à travers des protocoles de signature numérique.

2.3 Exemples d'algorithmes de Cryptographie

Dans le cadre de cette thèse quelques algorithmes de cryptographie ont été implantés en logiciel et en matériel, afin de fournir des éléments pour les preuves de concepts nécessaires. Ces algorithmes sont donc présentés dans les sous-sections qui suivent.

2.3.1 Data Encryption Standard

Le *Data Encryption Standard* (DES) est un algorithme de cryptographie qui a été sélectionné comme un standard pour la *Federal Information Processing Standard* (FIPS) pour les États Unis en 1976, mais qui a connu un succès international par la suite.

Le DES est un algorithme symétrique de bloc, qui prend une chaîne de caractères (ou nombres) d'une taille fixée à 64-bits du texte en clair et le transforme à l'aide d'opérations compliquées vers un texte crypté de la même taille. Cette transformation dépend d'une clé, et donc seulement celui qui connaît la clé est - a priori - capable de décrypter le message.

En apparence la clé est constituée de 64-bits ; en fait seulement 56 de ces 64-bits sont vraiment utilisés dans l'algorithme. Huit bits sont employés pour faire de la vérification de parité.

La vue d'ensemble de la structure du algorithme DES est montrée dans la Figure 2.4. Il existent 16 étages de calcul identiques, dites *rounds*. Il y a aussi des permutations, l'une initiale (PI) et l'autre finale (PF), où PF défait l'opération réalisée par PI et vice-versa. Il semblerait que PI et PF ne sont pas de signification cryptographique, mais qui ont été inclus dans l'algorithme pour faciliter le chargement de données dans le matériel des années 70. Avant les itérations principales,

[5] Les opérations modulaires, du genre *a (mod b)* sont représentées comme décrit au troisième paragraphe de la Section 2.6.1.

le bloc est divisé en deux moitiés de 32-bits chacune, qui sont calculées de façon alternée. Ce croisement est connu comme le schéma de Feistel.

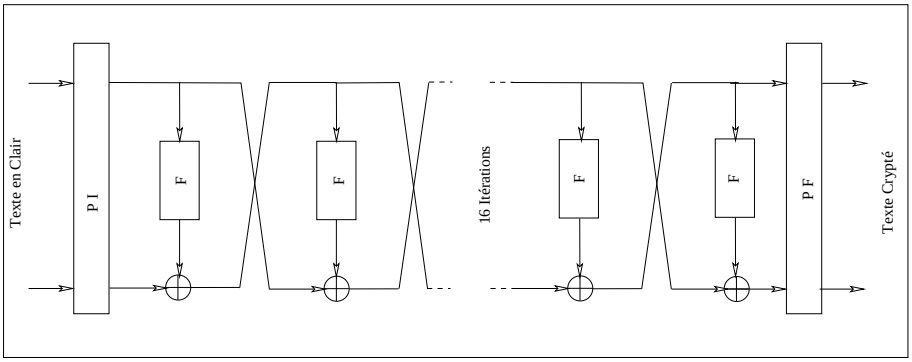


FIG. 2.4: Schéma de cryptage du DES

La structure de Feistel garanti que le cryptage et que le décryptage soient des procédés similaires. La seule différence c'est que les sous-clés sont appliqués dans l'ordre inverse pendant le décryptage. Cela simplifie l'implantation, notamment en matériel, car il n'est point question de séparer les modules de cryptage et de décryptage. Un exemple complet de cet algorithme, avec le source code en Maple, se trouve dans l'Annexe C.

La fonction de Feistel

La fonction F observée dans la Figure 2.5 opère dans une moitié d'un bloc (donc 32-bits). Cette fonction consiste de quatre étages :

1. **Expansion :** Le demi-bloc de 32-bits est étendu à 48 bits en faisant usage de la permutation d'expansion, dénoté E dans le diagramme ;
2. **Mixage de la clé :** Le résultat est combiné avec une sous-clé utilisant le XOR. Seize sous-clés de 48-bits (une pour chaque itération) sont dérivées de la clé principale à travers d'un ordonnancement de clés.
3. **Substitution :** Après avoir mixé à la sous-clé, le bloc est alors divisé dans huit parties de six-bits ; avant de passer par les boîtes de substitution (S-boxes - *Substitution Boxes*). Chacune de ces 8 S-boxes transforment les six bits d'entrées vers 4 bits de sortie, avec une transformation non-linéaire. Les S-Boxes fournissent le cœur de la sécurité du DES. Sans les S-box les transformations seraient linéaires, et par conséquent l'algorithme serait facilement cassé.
4. **Permutation :**Finalement, les sorties à 32-bits à partir des S-boxes sont disposées en accord avec un tableau fixe de permutation, appelée $P - Box$.

L'alternance de substitutions, permutations et expansions provoquent le concept de "confusion et diffusion" respectivement ; concept identifié par Shannon dans les années 40 comme étant des conditions nécessaires pour un algorithme de cryptographie sûr.

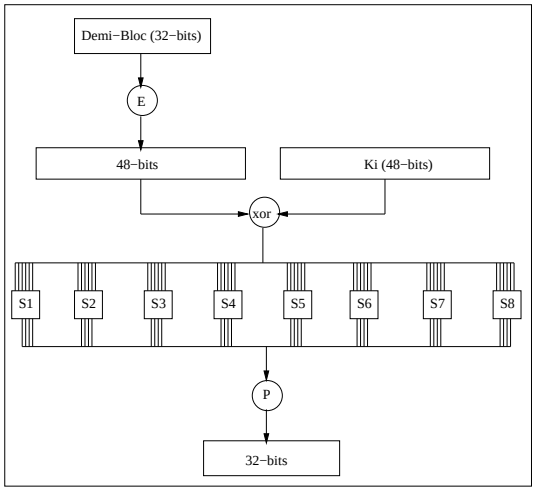


FIG. 2.5: La fonction Feistel du DES

L'ordonnancement de la clé

La Figure 2.6 illustre l'ordonnancement de clés pour le cryptage, I.e., l'algorithme qui génère les sous-clés.

D'abord 56-bits de la clé sont sélectionnés des 64-bits du départ, à travers du choix permuté 1 (PC1) ; les huit bits restant sont ignorés, ou alors utilisés comme bits de parité. Les 56-bits sont alors divisés en deux parties de 28-bits chacune. A partir de ce moment, chaque moitié est traitée séparément. Dans des itérations (*rounds*) successives, les deux parties de la clé sont rotationées d'un ou de deux bits (spécifique à chaque itération) ; et alors une sous-clé de 48-bits est choisie par le PC-2 - 24-bits de la moitié gauche et 24-bits de la moitié droite. Les rotations (dénotées par "$<<<$" dans la Figure 2.6) signifient que un différent ensemble de bits est utilisé à chaque sous-clé, chaque bit étant utilisé dans environ 14 des 16 sous-clés.

L'ordonnancement pour le décryptage est similaire, mais il doit générer la clé dans l'ordre inverse. Donc, les rotations sont à droite ("$>>>$") et non plus à gauche.

Sécurité Malgré le fait que il existe plus d'informations publiés à propos du DES que de n'importe quel autre algorithme de cryptographie, l'attaque logiciel la plus efficace et pratique contre cet algorithme est toujours la force brute. Plusieurs propriétés sont connues, et trois autres attaques sont possibles du point de vue théorique, avec une complexité de calcul plus faible que la force brute, mais ils demandent une quantité non-réaliste de textes choisis, et ne sont donc pas utilisés en pratique. Une machine proposée par le EFF (*Electrical Frontier Fondation*) 1998, équipée avec 1856 processeurs (29 cartes possédant 64 circuits "*DES-crackers*" chacune) et coûtant environ U$250.000, a pu casser une clé DES par force brute en moins de trois jours de calculs [39].

Cependant, depuis la fin des années 90, il existe une catégorie d'attaques en matériel assez efficace contre le DES, et qui sera montrée dans la Section 3.1.1.

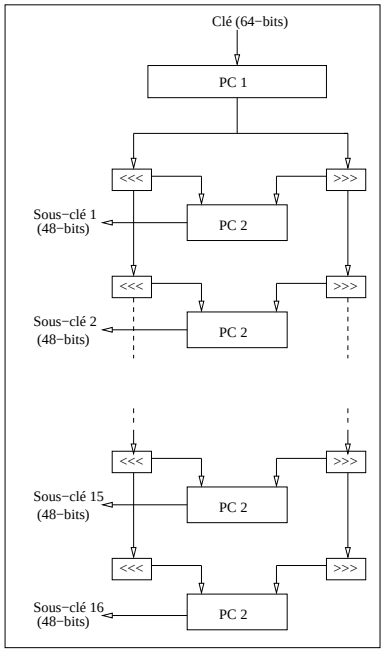


FIG. 2.6: L'ordonnancement de clés

2.3.2 Algorithme RSA

Le système RSA, nommé d'après les noms de ses inventeurs (Rivest, Shamir et Adleman), est le premier algorithme à clé publique publié ; et il est le plus cryptanalysé depuis les années 80. Malgré ces années d'intense cryptanalyse, personne n'a pas pu prouver la sécurité du RSA, et ni la désapprouver. Cela suggère un haut dégrée de fiabilité de l'algorithme. Sa sécurité vient de la difficulté de factoriser des grands nombres. Les clés publique et privée sont une fonction d'un couple de grands nombres (1024 bits ou plus) premiers. Découvrir le texte en clair à partir de la clé publique et du texte crypté est conjecturé comme equivalent à factoriser le produit des deux grands premiers.

L'algorithme RSA utilise l'arithmétique de $\mathbb{Z}n$, où n est le produit de deux nombres premiers distincts p et q, choisis de façon aléatoire :

$$n = p \cdot q$$

Ensuite il faut choisir une clé publique e de tel manière que e et $\phi(n) = (p-1)(q-1)$[6] soient premiers entre eux. Finalement, à l'aide de l'algorithme d'Euclide étendu, se calcule la clé de décryptage d, tel que :

$$e \cdot d \equiv [1]_{\phi(n)}$$

D'une autre façon :

$$d = [e^{-1}]_{\phi(n)}$$

[6]La fonction ϕ est définie dans la Section 2.6.1.

Il faut noter que d et e sont relativement premiers. Les nombres e et n forment la clé publique, pendant que les nombres d et n sont la clé privée. Les nombres p et q ne sont plus alors nécessaires et ne doivent être jamais révélés.

Pour crypter une message m, il est tout d'abord nécessaire la découper en blocs numériques de taille plus petit que n (normalement la plus grand puissance de 2 plus petite que n), composés de digits binaires. I.e. si p et q ont 1024-bits chacun, donc n aura 2048-bits et chaque bloc m_i doit être légèrement au-dessous de 2048-bits. Le message crypté c sera aussi divisé en blocs c_i de la même taille que les m_i. La formule de cryptage est la suivante :

$$c_i = [m_i^e]_n$$

et le décryptage est obtenu simplement par :

$$m_i = [c_i^d]_n$$

puisque :

$$c_i^d = (m_i^e)^d = m_i^{ed} = m_i^{k(p-1)(q-1)+1} = m_i \times 1 = m_i;$$

le tout $mod\ n$. Ceci est résumé dans l'algorithme 1.

Algorithme 1: Cryptage RSA

Entrée: deux grands nombres premiers p et q
Sortie: c ou m dépendant s'il s'agit d'une opération de cryptage où de décryptage

1: **Clé publique :**
2: $n \leftarrow p \cdot q$ et $\phi(n) \leftarrow (p-1)(q-1)$.
3: Choisir un nombre e tel que $(1 < e < \phi(n))$ avec $pgcd(e, \phi(n)) = 1$
4: **Clé privée :**
5: $d \leftarrow [e^{-1}]_{\phi(n)}$
6: **Cryptage :**
7: $c \leftarrow [m^e]_n$
8: **Décryptage :**
9: $m \leftarrow [c^d]_n$

Un exemple de cet algorithme, avec le source code en Maple, se trouve dans l'Annexe B.

Sécurité

Une attaque évidente à ce système consiste à tenter de factoriser n. Si cela est possible, il est trivial de calculer $\phi(n) = (p-1)(q-1)$ et alors de calculer l'exposant de décryptage d à partir de e. Ainsi, pour que le système RSA soit sûr, il est nécessaire que n soit suffisamment grand pour que sa factorisation soit calculatoiremment impossible. Aujourd'hui la taille recommandée est de au moins 1024-bits.

Opérations

Laissant de côté pour l'instant les aspects de sécurité, il faut faire attention aussi aux opérations nécessaires au cryptage et au décryptage. Ces procédés sont basés sur l'exponentiation *modulo* n. Comme n est un grand nombre, is est primordial d'appliquer des règles arithmétiques pour calculer dans $(Z)_n$. Le calcul de $c = [m^x]_n$ pourrait se faire par $e - 1$ multiplications modulaires, mais ce calcul est inefficace si x est grand. Il est important de considérer que x peut être aussi grand que $\phi(n) - 1$; soit légèrement plus petit que n, et donc exponentiellement grand. La méthode d'exponentiation rapide la plus utilisée c'est l'exponentiation binaire. L'algorithme d'exponentiation binaire utilise la décomposition de l'exposant x dans son expression binaire :

$$x \leftarrow \sum_{i=0}^{l-1} x_i \cdot 2^i,$$

où l est la longueur en bits de x, et $c_i = 0$ *ou* $1, 0 \leq i \leq l - 1$. L'algorithme 2 permet de calculer $z = [y^x]_n$:

Algorithme 2: exponentiation binaire

Entrée: des entiers y, x et n
Sortie: z tel que $z = [y^x]_n$
ExpBin(y, x, n)
1: $z \leftarrow 1$
2: **for** $i \leftarrow l - 1$ **to** 0
3: $z \leftarrow [z^2]_n$
4: **if** $x_i = 1$ **then** $z \leftarrow [z \times x]_n$
5: **return** (z)

L'algorithme RSA est étudié et cryptanalysé depuis presque trente ans, et même si sa sécurité n'est pas prouvé (ni d'ailleurs son insécurité), il reste désormais comme un des algorithmes à clé publique les plus fiables et répandus. Pourtant, une nouvelle catégorie d'attaques, ne reposant plus seulement sur les aspects mathématiques des algorithmes de cryptographie, a vu le jour à la fin des années 90. Ce genre de cryptanalyse est spécialement intéressant dans le domaine de la micro-électronique, car elle s'attaque aux implantations matérielles des systèmes cryptographiques. Par conséquent, ce sujet est abordé dans la Section 3.1.

Mais avant de venir aux attaques, il est important revoir quelques concepts mathématiques, car une contremesure basée sur l'arithmétique modulaire est au cœur de ce travail.

2.4 Nombres Premiers

Un nombre premier est un nombre entier naturel strictement supérieur à 1 ne possédant que deux entiers naturels que le divisent : lui même et 1. L'ensemble des nombres premiers peut être

noté comme $\mathbb{P}$. La propriété d'un nombre d'être premier ou pas est appelée *primalité*. Un entier supérieur à 1 qui n'est pas premier est appelé nombre composé car il est factorisable. En effet, tout nombre supérieur à 1 et non premier peut être décomposé en un produit de puissances de facteurs premiers de façon unique, ainsi un nombre qui peut être décomposé en un produit de facteurs premiers n'est pas premier.

Les nombres premiers ont de nombreuses utilisations pratiques, notamment dans la cryptographie asymétrique. Les nombres premiers extrêmement grands (c'est à dire plus grand que 2^{1024}) peuvent être utilisées comme des clés publiques cryptographiques (2.3.2), rendant ainsi difficile (voir impossible) la cryptanalyse en temps réel. La cryptanalyse anticipée reste pourtant possible, conforme la discussion dans la Section (3.1).

Les nombres premiers sont aussi utilisés pour construire des tables de hachage et pour constituer des générateurs de nombres pseudo-aléatoires.

Voici quelques propriétés qui rendent les nombres premiers intéressants pour la cryptographie :

Lemme 2.1: ***(Gauss)*** *: Si p est un nombre premier et si p divise un produit $a \cdot b$ d'entiers, alors p divise a ou p divise b.* ∇

Théorème 2.1: ***(Petit théorème de Fermat)*** *: Si p est un nombre premier et a est un entier quelconque, alors $a^p - a$ est divisible par p.* ∇

Théorème 2.2: ***(Wilson)*** *: Un entier p est premier si et seulement si la factorielle $(p-1)!+1$ est divisible par p. Inversement, un entier $n > 4$ est composé si et seulement si $(n-1)!$ est divisible par n.* ∇

Théorème 2.3: ***(Postulat de Bertrand)*** *: Si n est un entier strictement positif, alors il existe toujours un nombre premier p tel que $n < p \leq 2 \cdot n$.* ∇

Théorème 2.4: ***(Euclide)*** *: La séquence de nombres premiers est infinie.* ∇

Théorème 2.5: ***(Théorème Fondamentale de l'Arithmétique)*** *: Tout entier plus grand que 1 peut être représenté de manière unique comme un produit de nombres premiers.* ∇

Dans le domaine de la cryptographie il est primordial de tester la primalité d'un nombre, afin de pouvoir choisir des grands nombres premiers. Il existe plusieurs façons de générer un nombre premier (ou vérifier la primalité d'un nombre). Ces méthodes sont en général de trois types :

- Formules polynomials ;
- Formules exponentielles ;
- Formules factorielles.

Au delà de ces formules, il y a une autre forme pour calculer les nombres premiers. Il s'agit du Cribe d'Érastotenes, qui est à l'origine des cribes modernes. La littérature dans ce domaine est riche, mais pour cette thèse ont été consultés principalement les références [25] et [86].

2.5 Fonctions à sens unique

Une fonction à sens unique (*one-way function*) est une fonction qui peut être facilement calculée, mais qui est difficile à inverser. Cela signifie qu'il est difficile de calculer la donnée d'entrée à partir de la seule donnée de sortie. Par exemple, étant donnée une fonction ϕ, il est facile de calculer $y = \phi(x)$; pourtant, avec ϕ et y, il est difficile d'obtenir x. Il faut souligner l'emploi du mot "difficile" au lieu de "impossible", car il n'est pas encore prouvé qu'il existe des fonctions réellement à sens unique. Les fonctions à sens unique pour la cryptographie peuvent être classées fonctions à sens unique avec une trappe (voir Section 2.5.1) et en fonctions de hachage à sens unique (Section 2.5.2).

2.5.1 Fonctions à sens unique avec une trappe

Cette catégorie de fonctions à sens unique se caractérise pour posséder une trappe, c'est à dire, il est toujours difficile de calculer la donnée d'entrée à partir de la fonction et de sa sortie sauf, si un secret (la trappe) est à disposition. Un exemple de ce genre de fonction c'est le problème de factorisation des grandes nombres entiers, problème sur lequel repose la sécurité du système de cryptographie RSA. Dans cet exemple, la fonction ϕ est la multiplication entre deux grandes nombres premiers p et q. Calculer $y = \phi(p, q) = p \times q$ est facile. Par contre, si p et q sont suffisamment grands, il est irréalisable de les retrouver connaissant seulement la fonction ϕ et y.

2.5.2 Fonctions de hachage à sens unique

Une fonction de hachage est une manière de créer une "empreinte" à partir d'une certaine quantité de données. La fonction hache et mélange des données pour fabriquer l'empreinte, souvent appelée valeur de *hash*. Une caractéristique intéressante des fonctions de hachage c'est qu'elles acceptent comme entrée un message X, de longueur variable, mais sortent une valeur de hachage ($H(X)$)de taille fixe. Normalement $H(X)$ est beaucoup plus petit que X.

Dans le domaine de la cryptographie, les fonctions de hachage fournissent une sécurité additionnelle pour des applications comme par exemple garantir l'authenticité et l'intégrité d'un message.

Deux algorithmes qu'implantent les fonctions de hachage habituellement utilisés pour la cryptographie sont le MD-5 et le SHA-2.

Les propriétés des fonctions de hachage à sens unique peuvent être résumées comme suit :

Propriété 2.1: *H peut être appliqué à n'importe quel taille de données d'entrée.* ∇

Propriété 2.2: *La sortie de H est de taille fixe (128-bits pour le MD5, par exemple).* ∇

Propriété 2.3: *$H(X)$ est facile à calculer pour n'importe quel X.* ∇

Propriété 2.4: *Pour un Y quelconque, c'est infaisable trouver X, pour $Y = H(X)$* ∇

Propriété 2.5: *Pour un X fixé, c'est infaisable trouver $X' \neq X$, pour $H(X) = H(X')$* ∇

Si les fonctions à sens unique sont fondamentales pour assurer l'intégrité des messages, les principes des méthodes cryptographiques résident ailleurs. La Section 2.6 traite des concepts mathématiques indispensables à compréhension des algorithmes de cryptographie.

2.6 Arithmétique Modulaire

L'arithmétique modulaire est un système d'arithmétique d'entiers, où les nombres ne sont pas traités directement, mais calculés à partir de ses restes à partir d'une division euclidienne, par rapport à une valeur donnée - le module. La façon la plus simples d'illustrer l'arithmétique modulaire c'est de se poser la question : "Quand est-ce que $13 + 18 = 7$? Cela arrive quand on se réfère aux heures, car il s'agit d'un système cyclique : tout ce qui dépasse 12 (le module) est re-calculé de façon à obtenir un résultat plus petit que 12.

La notion de l'arithmétique modulaire est semblable à celle du reste dans les divisions. L'opération pour trouver le reste d'une division est connue comme module, et souvent est écrit comme *mod*, de la façon suivante : $14 \ mod \ 11 = 3$. La différence entre *mod* et ce qui a été écrit dans le paragraphe précédent est subtile mais significative. Cette distinction est expliquée par le concept de congruence.

2.6.1 Congruence

Deux entiers a et b sont dites congruentes *modulo* n si a et b ont le même reste quand divisés par n, ou, de manière équivalent, sa différence $(a - b)$ est un multiple de n. Dans ce cas, il est exprimé comme 2.4 :

$$a \equiv b(mod\ n) \tag{2.4}$$

Par exemple : $36 \equiv 14(mod\ 11)$, car $36 - 14 = 22$ qui est un multiple de 11. Congruence peut aussi être vue comme deux nombres qui ont le même reste après une division par le module n. Donc, $36 \equiv 14(mod 11)$ car quand 36 et 14 sont divisés par 11, le reste est 3.

Pour reprendre l'exemple du début de cette Section, c'est correcte de dire $36 \equiv 14(mod\ 11)$, mais c'est incorrecte de dire $36 = 14\ mod\ 11$. Pour un souci de rigueur, dorénavant les opérations de l'arithmétique modulaire seront écrites comme $[a]_n$ pour dénoter $a(mod\ n)$.

Propriétés

La congruence *modulo* n possède les propriétés suivantes :

Propriété 2.6: ***Réflexivité*** *:*$a \equiv a(mod\ n)$ ∇

Propriété 2.7: ***Symétrie*** *:*$a \equiv b(mod\ n) \iff b \equiv a(mod\ n)$ ∇

Propriété 2.8: ***Transitivité*** *: Si* $a \equiv b(mod\ n)$ *et* $b \equiv c(mod\ n)$*, alors* $a \equiv c(mod\ n)$ ∇

De plus, il existe une propriété algébrique remarquable :

Propriété 2.9: *Si* $a_1 \equiv [b_1]_n$ *et* $a_2 \equiv [b_2]_n$, *alors* $a_1 + a_2 \equiv [b_1 + b_2]_n$ *et* $a_1 \times a_2 \equiv [b_1 \times b_2]_n$. ∇

Il y a donc une compatibilité avec les opérations d'addition et de multiplication des entiers. Ces propriétés permettent de définir le domaine de l'arithmétique modulaire : les ensembles de quotients $\mathbb{Z}/n\mathbb{Z}$. Alors il est possible de définir une addition et une multiplication en $\mathbb{Z}/n\mathbb{Z}$ par les règles ci-dessous :

$$[a]_n + [b]_n = [a + b]_n$$

$$[a]_n - [b]_n = [a - b]_n$$

$$[a]_n \times [b]_n = [a \times b]_n$$

De cette manière $\mathbb{Z}/n\mathbb{Z}$ devient un anneau commutatif à n éléments.

Congruences linéaires

Une congruence dont la forme est $a \times x \equiv [b]_m$ et où x est l'inconnue, est appelée congruence linéaire [70]. Les systèmes de congruences linéaires sont la base pour le Théorème du Reste Chinois (CRT - *Chinese Remainder Theorem*), étudié dans la Section 2.6.4.

La congruence $a \times x \equiv [b]_m$ a une solution si et seulement si le plus grand commun diviseur $pgcd(a, n)$ divise b.

A ce moment, il faut rappeler le petit théorème de Fermat (2.4) : si p est un nombre premier et a est un entier quelconque, alors il est possible de vérifier que :

$$a^p \equiv [a]_p \tag{2.5}$$

La fonction ϕ de Euler

Le théorème 2.4, à partir de la congruence 2.5, fut généralisé par Euler : pour tout entier strictement positif n et tout entier a premier avec n,

$$a^{\phi(n)} \equiv [1]_n \tag{2.6}$$

où ϕ désigne la fonction ϕ d'Euler comptant les entiers compris entre 1 et n et premiers avec n. En guise d'exemple, pour $n = 8$, $\phi(8) = 4$, car les quatre nombres $1, 3, 5, 7$ sont premiers avec 8. L'importance de cette fonction c'est qu'elle indique la taille du groupe multiplicatif des entiers *module n*. En d'autres mots, la valeur $\phi(n)$ est égale à l'ordre du groupe des éléments inversibles de l'anneau $\mathbb{Z}/n\mathbb{Z}$.

Le calcul de la fonction $\phi(n)$ se donne à partir de la définition 2.2.

Définition 2.2: $\phi(1) = 1$ *et* $\phi(n)$ *est* $(p-1) \cdot p^{k-1}$ *quand n est la* k^{ime} *puissance d'un nombre premier* p^k. ∇

Définition 2.3: $\phi(n \cdot m) = \phi(n) \cdot \phi(m)$ *; et dans ce cas,* ϕ *est une fonction multiplicative.* ∇

Une propriété à souligner c'est 2.10, car c'est la base de l'algorithme de cryptographie RSA :

Lemme 2.2: *Le nombre entier n est premier si, et seulement si, $\phi(n) = n - 1$.* ∇

Propriété 2.10: *Si n est un nombre entier naturel et a est premier avec n, alors $a^{\phi(n)} \equiv [1]_n$.* ∇

2.6.2 L'Algorithme d'Euclide

Le but de l'algorithme Euclidian a pour but déterminer le plus grand commun diviseur de deux nombres entiers. L'avantage de cet algorithme est qu'il n'exige pas la factorisation, tâche qui demande beaucoup de ressources de calcul.

Le pseudo-code (Algorithme 3) illustre l'algorithme d'Euclide de façon récursive, qui est l'implantation la plus naturelle de la description suivante. Étant donné deux entiers naturels a et b, on commence par tester si b est nul. Si oui, alors le PGCD est égal à a. Sinon, on calcule c, le reste de la division de a par b. On remplace a par b, et b par c, et recommence le procédé. Le dernier reste non nul est le plus grand commun diviseur entre a et b.

Algorithme 3: Algorithme d'Euclide récursif pour calculer le PGCD

Entrée: Deux entiers naturels a et b
Sortie: Le plus grand commun diviseur entre a et b
PGCD(a, b)
1: **if** $b = 0$ **then return** (a)
2: **else return** $c \leftarrow [a]_b$
3: PGCD(b, c)

L'algorithme original décrit par Euclide traitait d'un problème géométrique, et alors utilisait des soustractions répétées du nombre plus petit du nombre plus grand, à la place de réaliser des divisions. Cela équivaut au pseudo-code 4, qui est nettement moins efficace que l'algorithme 3.

Algorithme 4: Algorithme Original d'Euclide pour calculer le PGCD

Entrée: Deux entiers naturels a et b
Sortie: Le plus grand commun diviseur entre a et b
PGCD(a, b)
1: **while** $(b > 0)$
2: $c \leftarrow a \ mod \ b$
3: $a \leftarrow b$
4: $b \leftarrow c$
5: **return** a

En analysant les quotients calculés pendant l'exécution de l'algorithme, il est possible de déterminer des nombres entiers α et β de tel façon que :

$$a \cdot \alpha + b \cdot \beta = PGCD(a, b) \tag{2.7}$$

Exemple :

$$a = 2322; b = 654; \alpha = 20; \beta = -71$$

$$\begin{aligned} a \cdot \alpha + b \cdot \beta &= PGCD(a, b) \\ 2322 \cdot 20 + 654 \cdot (-71) &= PGCD(2322, 654) \\ 46440 + -46434 &= 6 \\ 6 &= 6 \end{aligned}$$

Algorithme d'Euclide étendu

L'équation 2.7 est calculée par l'algorithme d'Euclide modifié par D. Knuth [52]. L'avantage de l'algorithme d'Euclide étendu c'est qu'il est possible de calculer explicitement α et β.

L'implantation suggérée par Knuth consiste à exprimer chacun des restes obtenus lors des divisions en termes de a et de b, de façon semblable à la formule 2.7, mais sans espérer arriver au *dernier* reste. L'avantage réside dans le fait qu'il est simple de réaliser une division en utilisant les données des divisions immédiatement précédentes. Tout le reste peut être écarté, libérant de l'espace mémoire.

Supposant qu'en calculant le plus grand commun diviseur entre a et b, il est obtenu la séquence de divisions suivante :

$$\begin{aligned} a = b \cdot w_1 + r_1 \quad et \quad r_1 = a \cdot p_1 + b \cdot q_1 \\ b = r_1 \cdot w_2 + r_2 \quad et \quad r_2 = a \cdot p_2 + b \cdot q_2 \\ r_1 = r_2 \cdot w_3 + r_3 \quad et \quad r_3 = a \cdot p_3 + b \cdot q_3 \\ \vdots \qquad \vdots \\ r_{n-3} = r_{n-2} \cdot w_{n-1} + r_{n-1} \quad et \quad r_{n-1} = a \cdot p_{n-1} + b \cdot q_{n-1} \\ r_{n-2} = r_{n-1} \cdot w_n \quad et \quad r_n = 0 \end{aligned} \tag{2.8}$$

Les nombres $p_1, \ldots, p_{n-1}$ et $q_1, \ldots, q_{n-1}$ sont les nombres entiers à déterminer. Pour cela, il est intéressent de condenser les informations dans un tableau :

Il est possible d'observer qu'il y a deux lignes "illégales" dans le tableau 2.1, car elles ne correspondent à aucune division, car a et b ne sont pas des restes. Mais ces lignes seront importantes pour la suite des calculs.

TAB. 2.1: Informations concernant l'exécution du algorithme d'Euclide étendu

restes	quotients	p	q
a	$\varnothing$	p_{-1}	q_{-1}
b	$\varnothing$	p_0	q_0
r_1	w_1	p_1	q_1
r_2	w_2	p_2	q_2
$\vdots$	$\vdots$	$\vdots$	$\vdots$
r_{n-2}	w_{n-2}	p_{n-2}	q_{n-2}
r_{n-1}	w_{n-1}	p_{n-1}	q_{-1}

Le but c'est de déterminer comment remplir les colonnes p et q. En supposant que le tableau 2.1 a été rempli jusqu'à $(j-1)^{me}$ ligne. Pour remplir la ligne j, il est nécessaire diviser r_{j-2} par r_{j-1}, afin de trouver r_j et w_j, de façon que $r_{j-2} = r_{j-1} \cdot w_j + r_j$ et $0 \leq r_j < r_{j-1}$. Ainsi :

$$r_j = r_{j-2} - r_{j-1} \cdot w_j \tag{2.9}$$

En lisant les lignes $j-1$ et $j-2$ les valeurs de p_{j-2}, p_{j-1} et q_{j-2}, q_{j-1}, il est possible écrire :

$$r_{j-2} = a \cdot p_{j-2} + b \cdot q_{j-2} \quad et \quad r_{j-1} = a \cdot p_{j-1} + b \cdot q_{j-1} \tag{2.10}$$

En faisant la substitution de 2.10 dans 2.9, il est obtenu :

$$r_j = (a \cdot p_{j-2} + b \cdot q_{j-2}) - (a \cdot p_{j-1} + b \cdot q_{j-1}) \cdot w_j = a \cdot (p_{j-2} - w_j \cdot p_{j-1}) + b \cdot (q_{j-2} - w_j \cdot q_{j-1}) \tag{2.11}$$

donc

$$p_j = p_{j-2} - w_j \cdot p_{j-1} \quad et \quad q_j = q_{j-2} - w_j \cdot q_{j-1} \tag{2.12}$$

Pour calculer p_j et q_j ont été utilisés seulement les valeurs contenues dans les deux lignes immédiatement antérieures à la ligne j, en plus du quotient w_j. Donc, il est possible de calculer n'importe quel ligne du tableau 2.1, dès que les deux lignes précédentes soient connues. Il s'agit d'un processus récursif, et c'est à ce moment qui rentrent les deux lignes auparavant dites "illégales". Elles sont la car il est facile de calculer p et q dans ces deux cas. Interprétant cette ligne comme les autres il s'obtient :

$$a = a \cdot p_{-1} + b \cdot q_{-1} \quad et \quad b = a \cdot p_0 + b \cdot q_0$$

ce qui suggère le choix de :

$$p_{-1} = 1, q_{-1} = 0, x_0 = 0 \quad et \quad y_0 = 1$$

et à partir de ce moment la récursion peut démarrer. A la fin de l'algorithme il est calculé le PGCD qui correspond au reste r_{n-1}, il est ainsi possible de calculer :

$$d = r_{n-1} = a \cdot pn - 1 + b \cdot q_{n-1}$$

L'algorithme 5 synthétise les idées discutées à propos du calcul des inverses[7].

Algorithme 5: Algorithme d'Euclide étendu pour calculer le PGCD

Entrée: Deux entiers naturels a et b
Sortie: Le plus grand commun diviseur entre a et b
PGCDETENDU(a, b)
1: **if** $([a]_b = 0)$ **then return** $(0, 1)$
2: **else** $temp \leftarrow PGCDetendu(b, [a]_b)$
3: $p \leftarrow temp_{-1}$
4: $q \leftarrow temp$
5: **return** $(q, p - q \cdot \frac{a}{b})$

L'une des applications de cet algorithme est l'unicité de la factorisation d'un entier dans des produits de premiers. L'algorithme RSA repose essentiellement sur une manière efficace de calculer α et β.

2.6.3 L'inverse modulaire

En supposant que a et b sont des nombres réels($\mathbb{R}$), "*diviser a par b*" à multiplier a par $1/b$. Le nombre réel $1/b$ est connu comme étant l'inverse de b - avec $b \neq 0$. En outre, un nombre réel n'a q'un inverse que s'il est différent de 0. Maintenant il faut transposer cette idée vers $n\mathbb{Z}$.

Pour la classe $\bar{a}$, la classe $\bar{\alpha} \in n\mathbb{Z}$ est l'inverse de $\bar{a}$ si l'équation $\bar{a} \cdot \bar{\alpha} = \bar{1}$ est vérifiée dans $n\mathbb{Z}$. Il est évident que si $\bar{a} = \bar{0}$, donc $\bar{a}$ n'a pas d'inverse. Pourtant, dans $n\mathbb{Z}$ il peut y avoir d'autres éléments qui n'ont pas d'inverse.

Si $\bar{a} \in n\mathbb{Z}$ a l'inverse $\bar{\alpha}$, donc $\bar{a} \cdot \bar{\alpha} = \bar{1}$; ce qui revient à dire que $a \cdot \alpha - 1$ est divisible par n. Donc :

$$a \cdot \alpha + k \cdot n = 1 \tag{2.13}$$

pour un k quiconque. Cette équation implique que $pgcd(a, n) = 1$. Pour conclure, si $\bar{a}$ a un inverse dans $n\mathbb{N}$, donc $pgcd(a, n) = 1$.

En supposant a comme un entier et que le $pgcd(a, n) = 1$, l'équation 2.6.3 suggère que il est possible d'appliquer l'algorithme d'Euclide étendu aux nombres a et n afin d'obtenir α et β tel que :

$$a \cdot \alpha + n \cdot \beta = 1$$

Cette équation est équivalente à :

$$\bar{a} \cdot \bar{\alpha} = \bar{1}$$

[7]Une version itérative de cet algorithme se trouve dans l'adresse suivant : http://en.wikipedia.org/wiki/Extended_Euclidean_algorithm.

Alors la classe $\bar{\alpha}$ calculée par l'algorithme d'Euclide étendu est l'inverse de a dans $n\mathbb{Z}$. Finalement, si $pgcd(a, n) = 1$, donc $\bar{a}$ a un inverse dans $n\mathbb{Z}$. Le tout peut être résumé dans le théorème suivant :

Théorème 2.6: ***(Théorème de l'inversion)*** *: La classe $\bar{a}$ possède un inverse dans $n\mathbb{Z}$ si et seulement si a et n sont premiers entre eux.* ∇

L'inverse modulaire est dorénavant représentée par $[X^{-1}]_n$, précisant que c'est l'inverse modulaire de X *modulo* n. Le théorème de l'inversion est utilisé dans le calcul de la multiplication modulaire qui est discutée dans la Section 2.7.

2.6.4 Residue Number System

Le RNS (*Residue Number System*) permet de représenter un grand nombre entier par un ensemble de petits entiers, de façon à permettre que les calculs soient réalisés plus efficacement. Son intérêt est évident pour la cryptographie, où il est nécessaire de manipuler des grands nombres. Ce système est basé sur le Théorème des Restes Chinois.

Théorème des Restes Chinois

Le théorème original date du troisième siècle, et a été développé par le mathématicien chinois Sun Tzu, et concerne des congruences simultanées. Il est connu par la sigle CRT, qui provient de l'expression en anglais *Chinese Remainder Theorem*.

Théorème 2.7: ***(Théorème des Restes Chinois)*** *: Soient $n_1, ..., n_k$ des entiers premiers entre eux (I.e. $pgcd(n_i, n_k) = 1$ pour tout $i \neq j$). Alors, pour des entiers quelconques $a_1, ..., a_k$, il existe un entier x résolvant le système de congruences simultanées*

$$x_i \equiv [a_i]_{n_i}$$

de façon unique, pour $i = 1, ..., k$. ∇

En outre, toutes les solutions x_i pour ce système sont congruentes modulo le produit :

$$N = \prod_{i=1}^{k} n_i$$

Une solution x peut être obtenue comme suit. Pour chaque i, les entiers n_i et n/n_i sont premiers entre eux, et en utilisant l'algorithme d'Euclide étendu c'est possible de trouver r et s de manière que $r \cdot n_i + s \cdot (n/n_i) = 1$. Si e_i est défini comme $e_i = s \cdot n/n_i$, pour chaque $i \neq j$, alors :

$$ei \equiv [1]n_i \quad et \quad ei \equiv [0]n_j \tag{2.14}$$

La solution pour le système de congruences simultanées est donc :

$$X = \sum_{i=1}^{k} \left[a_i \cdot e_i \cdot \left[e_i^{-1} \right]_{n_i} \right]_N \tag{2.15}$$

Après cette introduction au CRT, il est possible définir le RNS.

Définition du RNS

Un système de numération RNS est défini par un ensemble de $k+1$ entiers $m_0, m_1, m_2, ...m_k$, dits *moduli*. Tous les moduli doivent être premiers entre eux, donc aucun modulus peut être facteur d'un autre modulo. Soit M le produit de tous les moduli m_i. N'importe quel entier $X < M$ peut être représenté dans le système RNS comme un ensemble de $k+1$ petits entiers $x_0, x_1, ...x_k$ avec $x_i = [X]_{m_i}$ représentant la classe residue de X pour ce modulus. L'entier X peut alors être récupéré de l'ensemble des x_i à l'aide du Théorème des Restes Chinois (théorème 2.7).

Opérations en RNS

Une fois représentés en RNS, plusieurs opérations peuvent être efficacement accomplies dans les entiers codés. Pour les opérations suivantes sont considérés deux entiers A et B, représentés pour a_i et b_i dans le système RNS défini par m_i, pour $0 < i \leq k$.

Addition et soustraction L'addition peut être réalisée tout simplement en additionnant les petites valeurs entières a_i et b_i modulo ses respectifs moduli m_i. Par exemple, une addition $Rsum$ peut être calculé à travers d'un ensemble de $Rsum_i$ tels que :

$$Rsum_i = [a_i + b_i]_{m_i}$$

De façon similaire pour la soustraction :

$$Rdiff_i = [a_i - b_i]_{m_i}$$

Multiplication et division Comme dans l'addition, pour obtenir le produit $Rprod$ entre A e B, il faut d'abord les représenter en RNS, et ensuite réaliser les calculs de chaque $Rprod_i$:

$$Rprod_i = [a_i \times b_i]_{m_i}$$

Par contre, la réalisation de la division est complexe, mais il ne s'agit pas d'une opération nécessaire pour les systèmes cryptographiques dont s'intéresse cette thèse. Par exemple, le papier [67] décrit une méthode pour réaliser la division en RNS.

Exemples

Afin de mettre en évidence le degré de parallélisme permis par le RNS, voici quelques exemples pratiques de son utilisation.

Conversion Decimal - RNS. En utilisant le Théorème 2.7, un nombre quelconque peut être représenté par un ensemble de moduli plus petits que lui même. L'intérêt de cette approche est de pouvoir représenter des grandes nombres au travers des petits moduli. Pourtant, par un souci pédagogique l'exemple sera montré avec un petit nombre entier.

S'il faut représenter, donc, le nombre $X = 335$ en RNS, en prenant une base $\beta_1 = 3, 4, 5, 7$ il s'obtient :

$$N = \prod_{i=1}^{k} n_i$$

$$N = 3 \times 4 \times 5 \times 7 = 420$$

$$\vdots$$

$$x_i \equiv [a_i]_{n_i}$$

$$x_1 \equiv [335]_3 \equiv 2$$

$$x_1 \equiv [335]_4 \equiv 3$$

$$x_1 \equiv [335]_5 \equiv 0$$

$$x_1 \equiv [335]_7 \equiv 6$$

Finalement :

$$\{335\}_{(10)} = \{2, 3, 0, 6\}_{RNS_{(3,4,5,7)}}$$

Conversion RNS - Decimal. Ensuite, pour retrouver le nombre décimal d'origine l'équation 2.15 est utilisée, où l'inverse modulaire est pre-calculé à l'aide de l'Algorithme 4, comme montré dans la Section 2.6.3 :

$$X = \sum_{i=1}^{k} \left[a_i \cdot e_i \cdot \left[e_i^{-1} \right]_{n_i} \right]_N$$

$$E = \{140, 105, 84, 60\}$$

$$E_N^{-1} = \{2, 1, 4, 2\}$$

$$\vdots$$

$$X_0 = 0;$$

$$X_1 = (0 + 2 \times 140 \times 2) \ mod \ 420 = 140$$

$$X_2 = (140 + 3 \times 1 \times 105) \ mod \ 420 = 35$$

$$X_3 = (35 + 0 \times 4 \times 84) \ mod \ 420 = 35$$

$$X_4 = X = (35 + 6 \times 2 \times 60) \ mod \ 420 = 335$$

Exemple d'opération RNS : une addition . Dans le cas de l'opération $Z = X + Y$ sur RNS, l'addition se fait au travers de la somme des moduli de X et Y modulo N. Pour la même base et le même nombre X de l'exemple précédent, que soit $Y = 28$ représenté en RNS comme $Y_{RNS_{(3,4,5,7)}} = \{1, 0, 3, 0\}$. L'opération $(X + Y)_{RNS}$ se réalise de la manière suivante :

$$z_1 = 2 + 1 \ mod \ 3 = 0$$

$$z_2 = 3 + 0 \ mod \ 4 = 3$$

$$z_3 = 0 + 3 \ mod \ 5 = 3$$

$$z_4 = 6 + 0 \ mod \ 7 = 6$$

En convertissant $Z_{RNS_{(3,4,5,7)}} = \{0, 3, 3, 6\}$ selon l'équation 2.15 il est obtenu le décimal $Z_{(10)} = 363 = 335 + 28$.

2.7 Multiplication Modulaire - L'algorithme de Montgomery

Considérant que les algorithmes de cryptographie les plus répandus utilisent l'exponentiation modulaire, il est important de réaliser la multiplication modulaire de manière efficace. Il existent plusieurs méthodes pour implanter cette opération, comme celles proposés para Barret [9], Booth [15] et Blakley [13]. Cependant, l'une des plus répandues est celle de la méthode de Montgomery ([68], [97]).

En 1985 Peter Montgomery a publié une manière de calculer le produit modulaire sans avoir besoin de réaliser la division par le module [68]. Cette caractéristique facilite l'implantation matérielle, puisque la division par un module grand est trop coûteuse du point de vue de la surface.

Avant d'expliquer la technique, il faut faire quelques suppositions préliminaires. Tout d'abord, tous les nombres en question sont représentés dans une base (ou racine) β , qui est normalement une puissance de 2 : $\beta = 2^t$, où t représente la taille en bits du chemin de données de l'architecture cible. Soit k le nombre de digits de t bits nécessaires pour représenter un nombre donné, la forme suivante symbolise les nombres :

$$A = \sum_{i=0}^{k-1} a_i \times \beta_i; \ B = \sum_{i=0}^{k-1} b_i \times \beta_i; \ M = \sum_{i=0}^{k-1} m_i \times \beta_i \tag{2.16}$$

Par exemple, si pour un système cryptographique donné il est nécessaire de manipuler des nombres de 1024bits, dans une architecture qui a un chemin de données de 16bits (t), k sera donc de 64, puisqu'il aura 64 mots de 16bits pour composer le nombre de $1024 - bits$. Sous cette forme, les digits x_i (a_i, b_i et m_i) satisfont la condition $0 \leq x_i < b$. L'algorithme de multiplication modulaire accumule les digits produits par $a_i \times \beta$, et intercale avec des réductions modulaires pour maintenir les résultats plus petits que M. La contribution de Montgomery est de parcourir les digits de A en ordre inverse (Voir l'algorithme 6), et par conséquent, il n'est pas nécessaire de réaliser des comparaisons avec la taille totale des nombres dans les réductions modulaires. Par rapport aux algorithmes de produit modulaire classiques, Montgomery réalise un décalage à droite au lieu d'un

décalage à gauche, et une addition au contraire d'une soustraction. Tout cela permet simplifier la logique combinatoire.

La multiplication modulaire de Montgomery part du principe que :

$$\frac{U + (U \times M' \ mod \ R) \times N}{R} \equiv U \times \beta^{-1} \ mod \ M \tag{2.17}$$

Pour un M' égal à :

$M' = -M^{-1} \ mod \ \beta$

Comme β est choisi de façon que $\beta = 2^n$, la division par β n'est qu'un décalage à droit de n bits ; et la réduction par β ($mod \ \beta$) équivaut à prendre les n bits de poids forts. Enfin, si U de l'équation 2.17 par $A \times B$, il est possible de calculer la multiplication modulaire, comme montre l'algorithme 6.

Algorithme 6: Algorithme de Montgomery pour la Multiplication Modulaire

Entrée: A, B et M dans la forme décrite en 2.16
Sortie: $R = [A \cdot B \cdot -\beta_{-1}]_M$
MMM(A, B, M)
1: $R \leftarrow 0$
2: **for** $i = 0$ **to** $k - 1$
3: $q_i \leftarrow \left[(r_0 + a_i \times b_0) \times [-m]^{-1}\right]_\beta$
4: $R \leftarrow R + a_i \times B + q_i \times M$
5: $R \leftarrow \frac{R}{\beta}$
6: **if** $(R > M)$ **then** $R \leftarrow R - M$
7: **return** (R)

Pour que l'algorithme soit réalisable, il faut aussi qui M soit premier avec b, condition facilement satisfaite, puisque b est une puissance de 2 et M est la multiplication de deux grands nombres premiers (donc, il est *impair*). Par conséquent, $pgcd(M, b)$ est toujours égal à 1.

La valeur $[-m]_{\beta^{-1}}$ est l'inverse modulaire de m_0 par rapport à b . $[-m_0]_\beta^{-1}$ est donc un résidu qui satisfait la congruence $([-m]_\beta^{-1} \cdot m_0 \equiv [1]_\beta$ et cela est calculé à l'aide de l'algorithme d'Euclide étendu. Le digit est calculé de telle façon qui $R + a_i \cdot B + q_i \dot{M}$ soit divisible par β.

Par contre, l'algorithme de Montgomery ne calcule pas exactement $[A \cdot B]_M$ mais, $[A \cdot B \cdot \beta_k]_M$. Par conséquent, il faut des étapes additionnelles pour enlever la valeur résiduelle β^{-k}. Cela peut être obtenu soit avec un pré-calcul, soit avec un post-calcul. Dans le premier cas A et B sont multipliés par $[\beta]_M$ avant que le calcul de la multiplication modulaire ne commence ; et à la fin le résultat doit être quand même multiplié par la valeur 1, afin de s'obtenir $[A \times B]_M$. La seconde méthode consiste en réaliser un deuxième passage par l'algorithme, en multipliant le résultat du premier passage par $\beta^{2 \times k}$.

Un exemple de cet algorithme, avec le source code en Maple, se trouve dans l'Annexe D.

Comme il est mentionné auparavant, les systèmes cryptographiques doivent manipuler des données de l'ordre de $1024-bits$ ou plus. Il est donc impossible de prendre l'algorithme de Montgomery pour réaliser des calculs sur des grands nombres. Il faut l'adapter. En représentant A, B, M et le résultat R dans la forme proposée en 2.16, il est nécessaire d'adapter l'algorithme 6. Comme conséquence, il faut compter sur une deuxième boucle enchaîné, utilisée pour réaliser les opérations de multiplication, addition et division (décalage) sur des grands nombres découpés. L'algorithme 7 montre cette adaptation.

Le seul problème de cet algorithme c'est qu'il a une complexité $O(2^k)$ car il a une telle dépendance de données qui ne permet pas de rendre les opérations parallèles. Dans l'algorithme 7 il est évident que pour calculer q il est nécessaire avoir d'abord r (ligne 3). Ensuite, pour calculer r, il faut absolument avoir q (lignes 6 et 7).

Le Tableau 2.2 montre les performances de cet algorithme pour des différentes cibles technologiques. L'implantation de cet algorithme sur des plate-formes diverses a contribué pour la maîtrise du problème, aussi bien qu'aide à positionner l'architecture proposée dans le cadre de cette thèse par rapport à d'autres architectures. Ce tableau montre aussi les résultats de trois implantations de l'état de l'art. Il est remarquable que la performance sur GPP est très faible par rapport aux autres. Cela est dû au fait que l'implantation sur GPP a été faite d'une façon simples, voir pédagogique, afin d'aider à la compréhension de l'algorithme.

Algorithme 7: Algorithme de Montgomery pour la Multiplication Modulaire au niveau Mot

Entrée: A, B et M dans la forme décrite en 2.16
Sortie: $R = [A \cdot B \cdot -\beta_{-1}]_M$
MMM2(A, B, M)

1: $R \leftarrow 0$
2: **for** $i = 0$ **to** $k - 1$
3: $q_i \leftarrow \left[(a_i \times b_0 + r_0) \times (-m_0)^{-1}\right]_\beta$
4: $c \leftarrow 0$
5: **for** $j = 1$ **to** $k - 1$
6: $r_{j-1} \leftarrow [r_j + a_i \times b_j + q \times m_j + c]_b \, eta$
7: $c \leftarrow \frac{(r_j + a_i \times b_j + q \times m_j + c)}{\beta}$
8: $r_j \leftarrow c$
9: **return** (R)

Dans la prochaine Section il est démontré qu'il est possible de combiner l'algorithme de Montgomery avec le système de numération residual (RNS) afin de se paralléliser le calcul de l'exponentiation modulaire.

Tab. 2.2: Implantation de l'algorithme de Montgomery sur différentes cibles technologiques

Architecture	Fréquence	Taille des mots	Cycles	Temps
GPP (UltraSparc)	1000MHz	1024	1.048.576	1.048,58μs
DSP (TMS320C62)	300MHz	32	21.409	71,35μs
Asic (CMOS 0,35μm)	50MHz	16	2.364	47,28μs
FPGA (Badrignans et al.) [5]	32,6MHz	32	1164	35,71μs
Nedjah et al. [72]	25MHz	16	2051	82,04μs
Örs et al. [75]	95,41MHz	32	3076	32,24μs
McIvor et al. [57]	71,04MHz	32	1025	14,43μs
Daly et al. [27]	54,61MHz	32	1083	19,83μs
Deschamps et al. [35]	24,4MHz	32	1024	41,97μs

2.8 Leak Resistant Arithmetic

Le but de la Leak Resistant Arithmetic (LRA) [7] est de protéger les circuits cryptographiques des attaques du genre *Side Channel*, qui sont expliqués dans le Chapitre 3, Section 3.1.1. La LRA fait usage du RNS et de l'algorithme de Montgomery pour implanter de manière sure l'exponentiation modulaire. De plus, grâce au parallélisme intrinsèque de cet approche, la LRA permet une implantation matérielle rapide et avec une bonne modularité pour supporter la croissance des tailles de clés cryptographiques, comme il est montré dans le Chapitre 4

L'une des bases du LRA c'est l'adaptation de l'algorithme de Montgomery à la notation RNS. Dans la version RNS de la multiplication, la valeur :

$$M_1 = \prod_{i=1}^{k} m_i \tag{2.18}$$

est choisie comme la constante de Montgomery, au lieu de β^k de la représentation classique. Alors, la multiplication produit :

$$R = \left[A \cdot B \cdot M_1^{-1}\right]_N \tag{2.19}$$

où R, A, B et N sont représentés en RNS à partir d'une base β_1 pré-definie. Comme dans la version classique, il faut trouver un Q tel que $(A \cdot B + Q \cdot N)$ soit multiple de M_1. Ensuite la division résultante $\frac{(A \cdot B + Q \cdot N)}{M_1}$ est exacte, mais la multiplication par M_1^{-1} ne peut pas être réalisée dans la base β_1. Il est donc nécessaire d'utiliser une deuxième base, dite β_2, contenant d'autres moduli premiers entre eux. Les valeurs intermédiaires du calcul sont alors étendues vers la base β_2, et la multiplication par M_1^{-1} est calculée dans cette nouvelle base. Dans le cadre de ce document, les deux bases β_1 et β_2 ont une taille k. Pour la suite des calculs, il faut définir $\beta_1 = (m_1, ...m_k)$ et $\beta_2 = (m_{k+1}, ...m_{2 \cdot k})$, avec $M_2 = \prod_{i=1}^{k} m_{k+i}$ et $pgcd(M_1, M_2) = 1$. Par la suite, pour calculer Q il est utilisée la caractéristique que $(A \cdot B + Q \cdot N)$ est un multiple de M_1. Par conséquent

$Q = [-A \cdot +B \cdot N_{-1}]_{M_1}$, ce qui implique :

$$q_i = \left[-a_i \cdot b_i \cdot n_i^{-1}\right]_{m_i} \tag{2.20}$$

Comme résultat la valeur $Q < M_i$ a été calculée de tel forme que $Q = [-A \cdot +B \cdot N_{-1}]_{M_1}$. Comme mentionné précédemment, $(A \cdot B + Q \cdot N)$ est calculé dans la base β_2. Avant de pouvoir évaluer $(A \cdot B + Q \cdot N)$, il est primordial connaître la valeur de $A \cdot B$ dans la base β_2, et étendre Q, qui vient d'être calculé dans la base β_1, en utilisant 2.20 en β_2. Ensuite se calcule $R = (A \cdot B + Q \cdot N) \cdot M_1^{-1}$ aussi dans la base β_2. Le résultat doit être étendu vers β_1 pour être utilisé dans le prochain appel à la procédure de multiplication. L'algorithme 8 décrit les calculs de la multiplication modulaire de Montgomery en RNS. L'issue de la fonction c'est le produit $\left[A \cdot B \cdot M_1^{-1}\right]_N$.

Algorithme 8: Montgomery pour la Multiplication Modulaire en RNS

Entrée: β_1, β_2 ; N dans les deux bases tel que $0 < 4 \cdot N < M_1, M_2$ et $PGCD(N, M_1) = 1$; A, B dans les deux bases, tel que $A \cdot B < M_1 \cdot N$
Sortie: R tel que $R \equiv \left[A \cdot B \cdot M_1^{-1}\right]_N$
MMMRNS(A, B, M)
1: $T \leftarrow A \otimes_{rns} B$ *in* $\beta_1 \cup \beta_2$
2: $Q \leftarrow T \otimes_{rns} (-N_1^{-1})$ *in* β_1
3: *Etendre* Q *de* β_1 *vers* β_2
4: $R \leftarrow (T \oplus_{rns} Q \otimes_{rns} N) \otimes_{rns} (M_1^{-1})$ *in* β_2
5: *Etendre* R *de* β_2 *vers* β_1

Les étapes 1, 2 et 4 sont des opérations parallélisables, grâce la représentation RNS, car dans le RNS il n'existe pas de dépendance de données explicite. Par conséquent, la complexité de l'algorithme réside essentiellement dans l'extension de base.

Extension de base La méthode adoptée pour réaliser l'extension de base dans le cadre de la LRA est celle proposée par Garner en [40], en utilisant un système à racine mixte (MRS - *Mixed Radix System*). En considérant l'étape 3 de l'algorithme 8, où la valeur de Q en RNS est étendue de β_1 vers β_2, alors pour chaque m_j en β_2 il est évalué :

$$|q|_{m_j} = |t_1 + t_2 m_1 + ... + t_k m_1 ... m_{k-1}|_{m_j} \tag{2.21}$$

où

$$\begin{aligned}
t_1 &= |q|_{m1} = q_1 \\
t_2 &= \left| \left| (q_2 - t_1) \right|_{m_1} \right|_{m_2}^{-1} \Big|_{m_2} \\
&\vdots \\
t_k &= \left| \left| \left| ... \left(q_k - t_1\right) |m_1|_{m_k}^{-1} - ... - t_{k-1} \right|_{m_{k-1}} \right|_{m_k}^{-1} \right|_{m_k}
\end{aligned}$$

Dans ce cas, les valeurs pré-calculées sont les $|m_i|_{m_j}^{-1}$ pour $j > i$, ce que signifie que il y a $\frac{k(k-1)}{2}$ valeurs à stocker. Une fois les calculs des t_i terminés, l'équation 2.21 peut être évaluée sans d'autres pré-calculs :

$$|q|_{m_j} = |t_1 + m_1[t_2 + m_2(t_3 + ... + m_{k-1}t_k)...)|_{m_j} \tag{2.22}$$

Le nombre de multiplications où additions requises est donc de $\frac{k(k-1)}{2} + k(k-1)$. Certaines améliorations sont envisageables ; par exemple, il est possible de définir des bases RNS telles que la taille de $|m_i - m_j|$ soit plus petite que la moitié de la taille de m_i pour tous les i, j. Cela signifie que le coût de 2.22 équivaut à $\frac{k(\frac{1}{2}1)}{2}$ multiplications pour chaque m_j, ce qui entraîne une complexité égale à $k(k-1)$ multiplications pour une extension de base MRS.

Un exemple de cet algorithme, l'extension de base incluse, avec le source code en Maple, se trouve dans l'Annexe E.

Exponentiation modulaire. Considérant l'usage de la multiplication modulaire de Montgomery en RNS (algorithme 8, il est envisageable de l'utiliser pour réaliser l'exponentiation modulaire. Comme tout algorithme d'exponentiation modulaire basé sur l'algorithme de Montgomery, le premier pas pour le calculer est l'évaluation de $[X^E]_N$, ce qui implique de transformer X vers la forme de représentation de Montgomery : $[X' = X \cdot M_1]_N$. X' est aussi appelé N-résidue de x par rapport à M_1. Cela est réalisé à l'aide de l'algorithme 8 ayant comme entrées X et $[M_1^2]_N$. Cette représentation a l'avantage de rester stable pour toute la multiplication de Montgomery :

$$MMRNS(X', Y', N, \beta_1, \beta_2) \equiv [XYM_1]_N$$

A la fin de l'exponentiation, la valeur $[Z' = X^E]_N$ est convertie vers le résultat espéré $Z = [X^E]_N$ à travers d'un appel à la multiplication de Montgomery, en passant comme paramètres d'entrées Z' et 1.

En guise d'exemple l'algorithme 9 calcule l'exponentiation modulaire avec la méthode de multiplication et élévation au carré.

Algorithme 9: Exponentiation Modulaire de Montgomery en RNS

Entrée: β_1, β_2 ; N dans les deux bases tel que $0 < 4 \cdot N < M_1, M_2$ et $PGCD(N, M_1) = 1$; la valeur pré-calculée $[M_{1^2}]_N$ aussi dans les deux bases ; un exposant positif représenté en binaire $E = \sum_{i=0}^{l-1} e_i S$
Sortie: un entier positif $Z = [X^E]_N$
MMERNS$(\beta_1, \beta_2, N, [M_{1^2}]_N)$
1: $U_0 \leftarrow MMMRNS(1, [M_{1^2}]_N, N, \beta_1, \beta_2)$
2: $U_1 \leftarrow MMMRNS(X, [M_{1^2}]_N, N, \beta_1, \beta_2)$
3: **for** $i = l - 1$ **to** 0
4: $b \leftarrow \bar{e_i}$
5: $U_b \leftarrow MMMRNS(U_b, U_{e_i}, N, \beta_1, \beta_2)$
6: $U_{e_i} \leftarrow MMMRNS(U_{e_i}, U_{e_i}, N, \beta_1, \beta_2)$
7: **return** $Z \leftarrow MMMRNS(U_0, 1, N, \beta_1, \beta_2)$

Il est possible de vérifier que dans une étape i il se rencontrent les calculs $U_0 = [X^{E_i} M_1]_N$ et $U_1 = [X^{E_i+1} M_1]_N$, avec $E_i = \sum_{t=i}^{l-1} e_t \cdot 2^{t-1}$.

Comparaisons. Cet approche peut être comparée à celle de Kawamura [50], qui a lui même démontré être plus efficace que la solution proposée para Posch [77]. Les algorithmes proposés différent essentiellement dans la multiplication modulaire, plus particulièrement en ce qui concerne l'extension de base. Donc, les auteurs de la LRA comparent les deux méthodes dans le papier [6], dont les résultats sont montrés au Tableau 2.3.

Tab. 2.3: Deux différentes méthodes d'implantation de l'exponentiation modulaire en RNS

	Bajard et al.	Kawamura et al.
Lignes 1 et 3 (Alg. 9)	$5 \cdot k$	$5 \cdot k$
1^{re} **extension de base**	$k^2 + k$	$2 \cdot k^2 + 2 \cdot k$
2^{me} **extension de base**	$k^2 + 2 \cdot k$	$2 \cdot k^2 + 2 \cdot k$
Total	$2 \cdot k^2 + 8 \cdot k$	$4 \cdot k^2 + 9 \cdot k$
Clé de 1024-bits	$2584k$	$4930k$

Même si les valeurs rapportées dans le Tableau 2.3 ne prennent pas en compte que des opérations élémentaires, telles que les multiplications modulaires à simple précision, elles peuvent être calculés en parallèle, la méthode proposée par Bajard est visiblement plus efficace, réalisant la même opération de multiplication modulaire avec approximativement la moitié d'itérations employées par la méthode de Kawamura. Une autre constatation tirée du Tableau 2.3 c'est que la complexité de l'exponentiation modulaires réalisée au travers du RNS réside fondamentalement dans les extensions de base. L'intérêt dans la réalisation de l'extension de base c'est le changement

d'une base β_1 vers une base β_2 complètement en RNS sans avoir besoin de recomposer le nombre en question dans sa forme décimale, ce qui serait encore plus coûteux.

2.9 Conclusion du Chapitre

Dans ce Chapitre sont montrés non seulement les bases mathématiques pour la compréhension des algorithmes cryptographiques, mais aussi l'arithmétique qui permette l'implantation d'une architecture matérielle sécurisée contre l'attaque par analyse de consommation (décrit dans le Chapitre suivant).

Ce chapitre-ci montre les algorithmes cryptographiques qui sont utilisés par la suite, soit pour démontrer une attaque matérielle, soit parce qu'ils sont implantés dans l'architecture proposée. L'algorithme DES est utilisé comme cible pour les expérimentations d'attaques et validations de contremesures, car il est amplement connu et utilisé, s'avérant de cette façon un instrument assez pédagogique pour la compréhension des concepts traités. D'un autre côté, l'algorithme RSA se présente comme le plus répandu algorithme de cryptographie à clé publique, se situant comme la première application cible pour l'architecture décrite dans le Chapitre 4.

Ensuite l'étude de l'arithmétique modulaire, en particulier de la multiplication modulaire, a permis la consolidation de l'intérêt pour la LRA. La *Leak Resistant Arithmetic* possède des caractéristiques menant à une implantation matérielle efficace et robuste. L'efficacité se donne du point de vue de la performance, car elle est basée sur le RNS, ce qui permet une parallélisation au niveau des opérations mathématiques de base ; et la robustesse est obtenue grâce à la possibilité du changement de base au cours d'une exponentiation modulaire.

L'entendement de la robustesse de la LRA est possible après la lecture du Chapitre 3, qui explique les attaques du type *Side Channel* ; et où sont montrées des considérations à propos de l'application de la LRA pour la sécurité contre les attaques par canaux cachés. Ce chapitre se focalise sur les attaques par analyse de consommation contre les circuits cryptographiques, montre un état de l'art à propos des contremesures matérielles et propose une contremesure originale.

CHAPITRE 3

Attaques et contremesures

"Il n'est point de secrets que le temps ne révèle". ***Jean Racine***

Ce chapitre a pour objectif discuter à propos des attaques contre les circuits cryptographiques, notamment les attaques du type canaux cachés. Dans cette partie sont aussi présentées quelques contre-mesures actuelles. Ensuite il est proposée une contre-mesure analogique originale contre les attaques pour analyse de consommation.

3.1 Attaques Matérielles

Les requis de sécurité pour des modules cryptographiques ont été normalisés par le standard FIPS PUB 140-2 [74], adopté par les institutions fédéraux aux États Unis. Le FIPS 140 définit de façon croissante et qualitative des niveaux de sécurité, comme montré ci-dessous :

- **Niveau de sécurité 1 :** C'est le niveau plus bas de sécurité. Spécifique les requis basiques pour la cryptographie concernant les implantations logicielles.
- **Niveau de sécurité 2 :** Ce niveau commence à définir la sécurité matérielle, en ajoutant des notions pour l'identification d'invasion, comme des sceaux[1].

[1]Historiquement, un sceau authentifiait les actes royaux ou passés au nom du monarque. Toute décision de justice

- **Niveau de sécurité 3 :** Augmentation de la sécurité physique, ayant pour but d'éviter l'intrus afin d'accéder à des données critiques dans le dispositif.
- **Niveau de sécurité 4 :**Il s'agit du niveau plus élevé de sécurité. Le dispositif doit être sécurisé même dans un milieu hostile, d'attaques logiciels où physiques.

Le Niveau 1 demande juste qu'il soit utilisé un algorithme approuvé, normalement implanté dans un processeur à caractère général. Le Niveau 2 augmente la sécurité physique, exigeant l'identification de l'utilisateur, et mettant en évidence d'éventuelles tentatives d'intrusion. Dans le Niveau 3, les mécanismes de sécurité doivent détecter l'attaque, mais aussi contre-carrer les tentatives d'intrusion, utilisation, ou de modification du module cryptographique. Le Niveau 4, le plus élevé en sécurité, implante les mesures des autres niveaux, et inclut la possibilité de détecter de variations d'environnement, tels que la température ou la puissance.

Il existe d'ailleurs une classification publiée en [1] à propos du niveau d'expertise et de ressources des ataquants, ce qui aide à situer les travaux originaux présentés dans cette thèse. Les adversaires peuvent, donc, être réunis en trois différentes classes, selon ses capacités et moyens (les dénominations en anglais sont gardés par un souci historique) :

- **Classe 1 :** Étrangers malins (*Clever Outsiders*). Personnes intelligentes, mais par forcément initiées dans la cryptanalyse, n'ayant que peu de ressources et utilisant des outils et services de bas coût.
- **Classe 2 :** Internes savants (*Knowledgeable Insiders*). Individus ayant un savoir-faire d'intrusion et de cryptanalyse où même des équipes accédant à des instruments avancés pour réaliser des attaques.
- **Classe 3 :** Organisations (*Founded Organizations*). Équipes hautement qualifiés, supportés par des grandes groups financiers, industriels ou même des agences d'état. Ils utilisent des équipements sophistiqués qui n'existent pas dans le marché conventionnel, et sont capables d'analyser finement les systèmes à casser.

Les niveaux de sécurité et les classes d'attaquants sont rappelés dans la suite du texte pour contextualiser certaines contre-mesures. Mais en ce qui concerne les attaques SCA, il existe aussi une classification quant à la manière d'accéder aux informations. Selon ce découpage, un attaque peut être invasive, semi-invasive où non-invasive :

- **Attaques invasives :** Ce genre d'attaque implique l'enlèvement du package dispositif cryptographique et l'accès à ses composants internes. Un exemple typique de cette attaque est quand un attaquant perce un trou dans la couche passive du dispositif et place une sonde dans un bus de données pour analyser les informations qui transitent. Les dispositifs sécurisés doivent pouvoir identifier une invasion et mettre à zéro toutes les informations (*zeroization*) afin d'atteindre le Niveau 3 de sécurité mentionné auparavant. Ces attaques nécessitent d'énormes ressources et des outils sophistiqués, pratiqués par des attaquants du type 3.
- **Attaques semi-invasives :**Ces attaques concernent l'accès au dispositif, mais sans avoir recours à endommager le package. Par exemple, dans une attaque par injection de fautes, l'attaquant peut faire usage d'un faisceau laser pour ionizer le dispositif afin de changer quelques bits de mémoire, et par conséquent, modifier la sortie.

portait alors un sceau l'authentifiant.

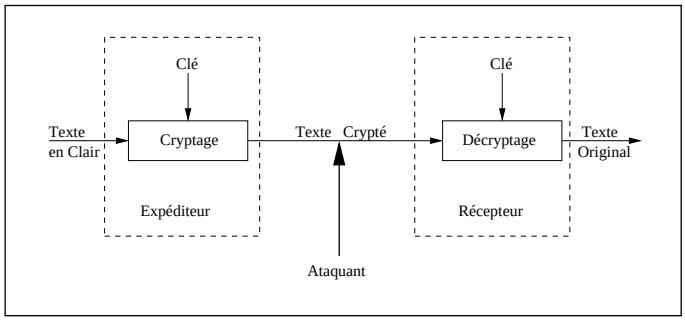


FIG. 3.1: Cryptanalyse Classique

- **Attaques non-invasives :** Une attaque non-invasive dépend de l'observation du dispositif pendant son opération. Cet attaque exploite les informations disponibles à l'extérieur du dispositif, qui s'enfuient de façon non-intentionnelle. La principale caractéristique des attaques non-invasives, c'est qu'elles sont indétectables. Comparé aux attaques invasives, ceux-ci sont moins coûteux, figurant comme la grande menace à l'industrie des circuits sécurisés.

Les attaques non-invasives sont donc l'objet d'étude de cette thèse. Des méthodes seront proposés pour empêcher les attaquants des classes I et II de réussir l'obtention de secrets des dispositifs cryptographiques.

La Section suivante donne un aperçu des attaques non-invasives.

3.1.1 Considérations à propos de sécurité et types d'attaques SCA

Après l'avènement des ordinateurs, les algorithmes de cryptographie sont toujours implantés en logiciel où en matériel, mais de tout façon dans des dispositifs physique qui sont influencés par leur environnement. Ces interactions peuvent être influencées par un attaquant, d'où peuvent ressortir des informations exploitables du point de vue cryptanalitique. Ce genre d'information est appelé informations de canaux-cachés, et les attaques qui les exploitent, attaques par canaux-cachés (en anglais SCA - *Side-Channel Attacks*). Les attaques SCA réussissent car il existe une relation entre les mesures physiques prises pendant un calcul (temps de calcul, puissance consommée, émissions electro-magnétiques, etc.) et l'état interne du dispositif, très souvent relié à la clé secrète de l'algorithme en question.

Un algorithme de cryptographie peut, donc, être considéré à partir de deux points de vue. D'un côté, come une abstraction d'un objet mathématique. D'un autre côté, comme un algorithme implanté dans un dispositif, dans un environnement donnée. Dans le premier cas c'est le point de vue de la cryptanalyse classique ; le seconde de la cryptanalyse des canaux-cachés.

Dans la cryptanalyse traditionnelle, conforme la Figure 3.1, l'attaquant peut avoir accès à la clé publique (si c'est le cas d'un algorithme de ce genre), aux messages échangés entre expéditeur et récepteur, à l'algorithme utilisé et une description détaillée du protocole en question. Alors l'attaquant exploite la spécification mathématique du protocole.

En plus des connaissances nécessaires à une cryptanalyse conventionnelle, les attaquants utilisant des techniques SCA prennent en compte toutes informations marginales d'un procédé de calcul. Quand un dispositif tel qu'un processeur calcule, il consomme de la puissance, il emmet de la chaleur et des radiations electro-magnétiques, il calcule dans un temps donné, dans une fréquence donnée, et il irradie même de la lumière ; tout cela selon les données qui sont calculés. Comme montre la Figure 3.2, ces informations par canaux-cachés sont exploitables afin de réaliser des attaques.

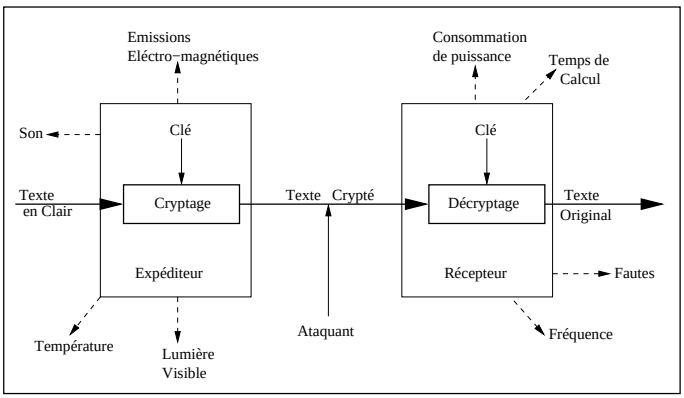


FIG. 3.2: Cryptanalyse SCA

Pour des informations détaillées à propos de touts les attaques SCA connus, la référence [103] fourni une intéressante analyse de dix ans d'attaques SCA, fournissant une riche référence bibliographique. En ce qui concerne cette thèse, puisqu'il n'existe pas de solution miracle pour contrecarrer tous les attaques existants, elle se concentre sur l'une des plus efficaces : les attaques par analyse de consommation.

Attaques par analyse de courant

Avant de poursuivre dans la discussion à propos des techniques de cryptanalyse, il est important de comprendre pourquoi ces attaques marchent. Presque la totalité des circuits numériques d'actualité sont fabriqués basés sur la technologie CMOS (*Complementary Metal Oxid Semiconductor*) ; il faut donc s'intéresser aux caractéristiques de consommation de cette technologie. Si une porte CMOS change d'état, ce changement peut être mesuré sur l'alimentation. Plus un circuit change d'état, plus il dissipe de la puissance. Dans un circuit synchrone toutes les portes changent d'état en même temps. Donc la consommation de puissance peut être mesurée à l'aide d'une resistance R_m placée en série avec le V_{dd} (ou le G_{nd}). Les deux parties principales de la consommation de puissance pendant un changement d'état sont la charge/discharge dynamique et le court circuit dynamique de courant. En considérant comme exemple le circuit inverseur présenté dans la Figure 3.3, la sortie de chaque porte a une charge capacitive, constitué de la capacité parasitique des fils connectés et des portes des étages suivantes (capacité de grille). Une transition d'entrée résulte dans une transition de sortie, produisant une charge ou une décharge de cette capacité, ce qui occasionne un flot de courant vers V_{dd}. Ce courant est celui de la charge/décharge dynamique. En

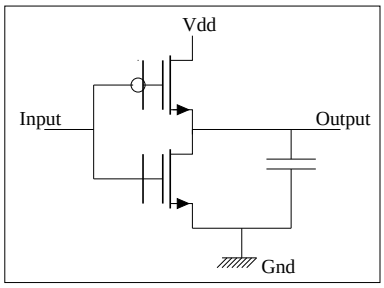


FIG. 3.3: Inverseur CMOS

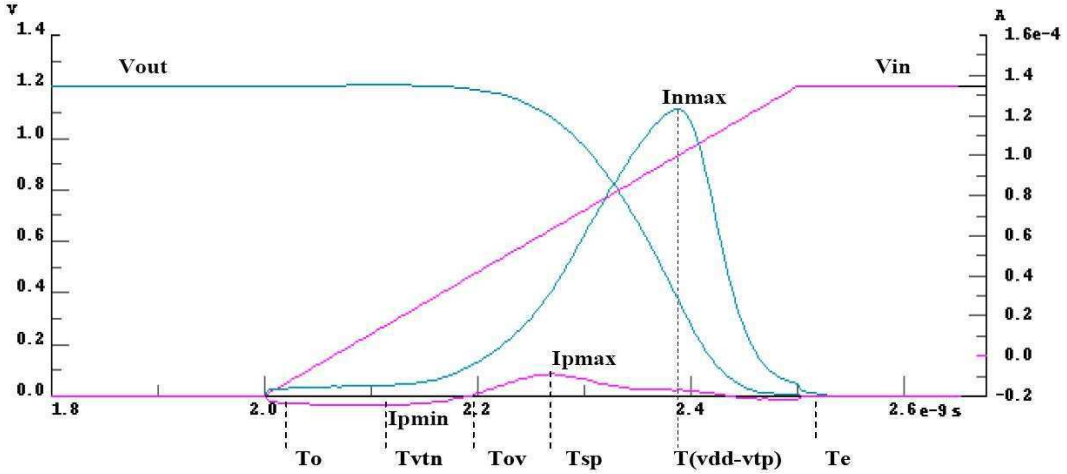


FIG. 3.4: Tensions d'entrée/sortie et courants des transistors P et N dans un inverseur CMOS sur front montant.[16]

mesurant le flot de courant dans V_{dd} (ou G_{nd})), il est possible de détecter si la sortie a changé de 0 vers 1 ou non.

La Figure 3.4 illustre ce phénomène. Les variations de courant circulant dans les transistors P et N de l'inverseur CMOS, ainsi que les tensions d'entrée et de sortie sont montrées dans cette figure. Il a été considéré pour cette image le front montant de l'entrée.

Lors de la commutation de la porte, les transistors passent successivement dans différents régimes de conduction qui permettent de déduire des zones de fonctionnement. A partir de l'analyse de ces zones est possible de tracer le profil de courant, et par la suite, de déduire les données traités à partir de la signature de consommation de la porte [16].

Dans la logique CMOS, chaque sortie apparaît aussi dans sa forme inversée, ce qui signifie qu'une transition va toujours causer charges et décharges dans la sortie, et dans la sortie inversée. En mesurant le courant en V_{dd}, il n'est pas possible de savoir s'il s'agit d'un 0 ou d'un 1, mas il est faisable détecter quelle transition est arrivée.

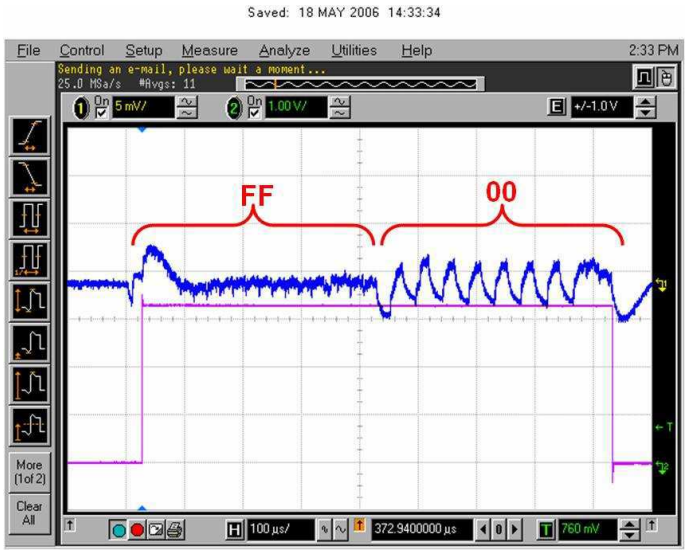


FIG. 3.5: Attaque SPA contre l'algorithme RSA.

Attaque SPA Une attaque nommée SPA (*Simple Power Analysis*) implique l'observation directe de la consommation du système. Si l'attaquant a un accès illimité au dispositif cryptographique et s'il connaît l'algorithme qui est implanté, il peut extraire de façon simple des informations à partir de l'analyse de la consommation. Par exemple, pendant la permutation P1 (Figure 2.5) dans l'algorithme DES, il est possible de déterminer le poids de Hamming de chaque octet en mesurant la valeur du pulse dans le cycle d'instruction qui accède à ces données. Dans un processeur 8-bits, en connaissant le poids de Hamming des huit octets de la clé DES, il est possible de réduire l'espace d'exploration d'une attaque à force brute, ainsi passé de 2^{56} à $\approx 2^{38}$ [65]

Afin d'illustrer la SPA, il est envisageable de la réaliser aussi sur des algorithmes de cryptographie à clé publique. Par exemple, lors d'une signature numérique utilisant l'algorithme RSA, le calcul effectué est une exponentiation modulaire (conforme l'algorithme 1). Pour cela, il est habituellement utilisé la méthode de l'exponentiation binaire (déjà vue dans l'algorithme 2). Dans le cas d'une signature électronique, l'exposant est la clé privée du signataire, et donc le secret véhiculé par le circuit. Réussir un attaque sur le RSA revient à retrouver la clé privée utilisée dans le calcul.

Pour mener l'attaque il faut remarquer que la multiplication située à la ligne 4 de l'algorithme 2 n'est exécutée que si le bit courant de l'exposant est 1. Autrement, le carré de la ligne 3 est effectué à chaque itération de la boucle for. La Figure 3.5 a été obtenue en calculant $C = [M^e]_N$ sur 16 bits[2], où l'exposant n'est pas connu, puis qu'il fait partie de la clé secrète.

Analysant la trace, il est possible de remarquer que dans un premier temps, la il y a peu d'oscillation dans la consommation. Dans un deuxième temps, sont visibles des oscillations. A partir de ces constatations, il est possible déduire que les bits de poids faible valent tous 1, car le multiplieur fonctionne à chaque itération ; ensuite, comme il y a une séquence de chutes de consommation, il est correcte de conclure que les bits de poids fort sont tous à 0, puisque la

[2]à titre pédagogique, afin de démontrer de façon évidente le concept.

chute de consommation signifie que la multiplication ne se réalise pas. Donc, l'exposant e dans la signature numérique $C = [M^e]_N$ vaut exactement $00FF$ (en hexadecimal). Il faut aussi noter que, pour bien réaliser un attaque SPA, il est nécessaire de connaître en détails l'algorithme attaqué. En outre, des contremesures simples existent, et sont abordées dans la Section 3.2

Attaque DPA L'attaque par analyse différentielle de puissance (DPA - *Differential Power Analysis*) est basée sur la mesure précise de la consommation de puissance d'un circuit cryptographique. L'attaquant faisant usage de la DPA n'a pas besoin de connaître les détails d'implantation de l'algorithme concerné, il suffit de savoir quel est cet algorithme. De plus, il est nécessaire à l'attaquant de pouvoir insérer une certaine quantité de textes en clair, afin de mesurer la consommation pendant le cryptage. La stratégie est de diviser ces mesures dans deux groupes selon une hypothèse. Ensuite, des méthodes statistiques sont utilisées pour vérifier l'hypothèse. Si, et seulement si une hypothèse est correcte, des pics sont visibles dans les courbes dérivées de la statistique. Afin d'illustrer cette attaque, l'algorithme DES implanté dans un FPGA jouera le rôle de victime. La Figure 3.6 illustre la mise en place physique de l'attaque. Comme vue pour la SPA, Il est nécessaire brancher le dispositif cible, qui dans notre cas s'agit d'une carte de prototypage équipée d'un FPGA Spartan3, à un oscilloscope (600MHz, résolution minimale de 6 bits) afin de prendre les mesures de courant à l'aide d'une sonde en courant. Ensuite un ordinateur (PC conventionnel) est utilisé pour le traitement informatique (soit avec des programmes en C, soit avec des logiciels comme Maple ou Matlab) des données obtenues.

Comme décrit dans la Section 2.3.1, le DES est composé par 16 étages dans lesquels sont réalisées 8 opérations de substitution dans les S-boxes. Les 8 S-Boxes ont chacune comme entrées 6-bits de la sous clé correspondante, qui sont XOR-é avec le 6-bits du registre R afin de produire les 4-bits de sortie. La sortie S de 32-bits est re-ordonnancée et XOR-ée avec le registre L. Les moitiés L et R sont alors échangées.

La fonction de sélection DPA $D(C, b, K_s)$ est définie comme le calcul de la valeur du bit $0 \leq b < 32$ du registre L intermédiaire dans le début du 16^{me} étage du DES, pour un texte crypté C, où les 6-bits de clé de l'entrée de la S-box correspondant au bit b est représenté par $0 \leq K_s < 2^6$. Il est important de remarquer que, si K_s est incorrect, évaluer $D(C, b, K_s)$ donne la valeur correcte pour le bit b avec une probabilité $P \approx \frac{1}{2}$ pour chaque texte crypté C.

La Figure 3.7 montre le schéma de l'attaque, qui est expliqué par la suite.

Pour implanter l'attaque DPA, l'attaquant observe d'abord m opérations de cryptage et capture les traces de courant $T_{1..m}[1..k]$ contenant k échantillons chacun. De plus, l'attaquant garde les textes cryptés $C_{1..m}$. Nulle connaissance des textes en clair n'est requise.

Donc, en analysant les mesures de consommation, l'attaquant essaye de vérifier les hypothèses de clé K_s. Il calcule k traces différentiels $\delta D[1..k]$ à travers de la différence entre la moyenne des traces pour lesquels $D(C, b, K_s)$ est un (1) et la moyenne des traces pour lesquels $D(C, b, K_s)$ est zéro (0). alors $\Delta D[j]$ est la moyenne pour $C_{1..m}$ de la valeur représentée par la fonction de sélection D dans les mesures de consommation dans un point j.

Si l'hypothèse de clé K_s est incorrecte, le bit calculé à l'aide de la fonction D va être différent

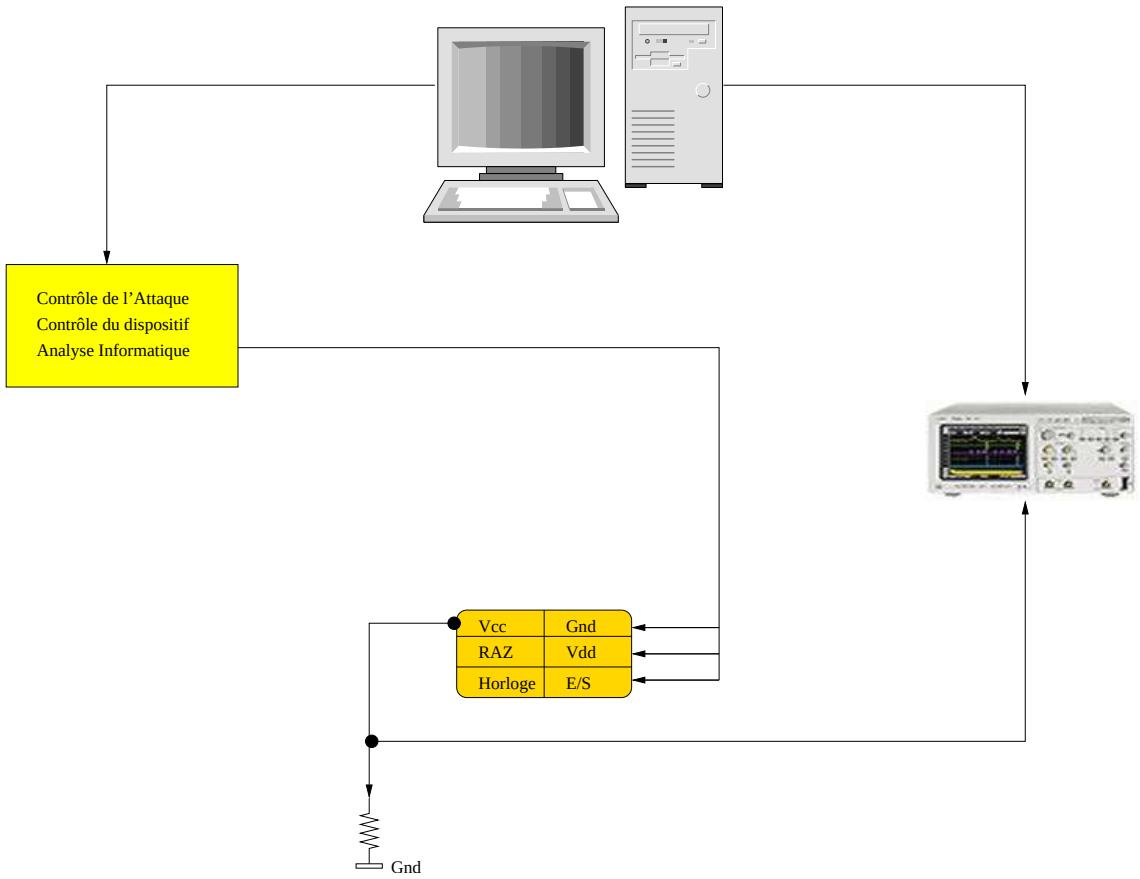


FIG. 3.6: Mise en place d'un attaque DPA

du bit cible réel pour approximativement la moitié des textes cryptés C_i. La fonction de sélection $D(C, b, K_s)$ est donc de-corrélée de ce qui était effectivement calculé dans le dispositif. Si une fonction aléatoire est utilisée pour diviser l'ensemble vers deux sous-ensembles, la différence dans les moyennes est approchée de zéro à l'infini. Donc, si K_s est incorrecte :

$$\lim_{m \to \infty} \Delta D[j] \approx 0$$

car les traces composantes de-corrélées à D diminuent en $\frac{1}{\sqrt[2]{m}}$, causant une trace différentielle tendant à zero.

Sinon, si l'hypothèse de clé K_s est incorrecte, la valeur calculé par $D(C, b, K_s)$ est égale au bit ciblé, avec une probabilité $P = 1$. La fonction de sélection est donc corrélée à la valeur du bit manipulé dans le 16^{eme} étage du DES. Comme résultat, $\Delta D[j]$ s'approche de l'effet du bit cible dans la consommation $m \to \infty$; les autres valeurs, erreurs de mesures, bruit, etc., ne sont pas corrélés à D proche de zéro. comme la consommation de puissance est corrélée aux valeurs des bits, le tracé de $\Delta D[j]$ sera plat avec pointes dans les régions où D est corrélé aux valeurs calculés. La Figure 3.8 montre que pour l'hypothèse correcte de clé, la courbe correspondant ressort comme prévu.

La valeur correcte de K_s peut donc être identifiée à partir de ces pointes dans la trace diffé-

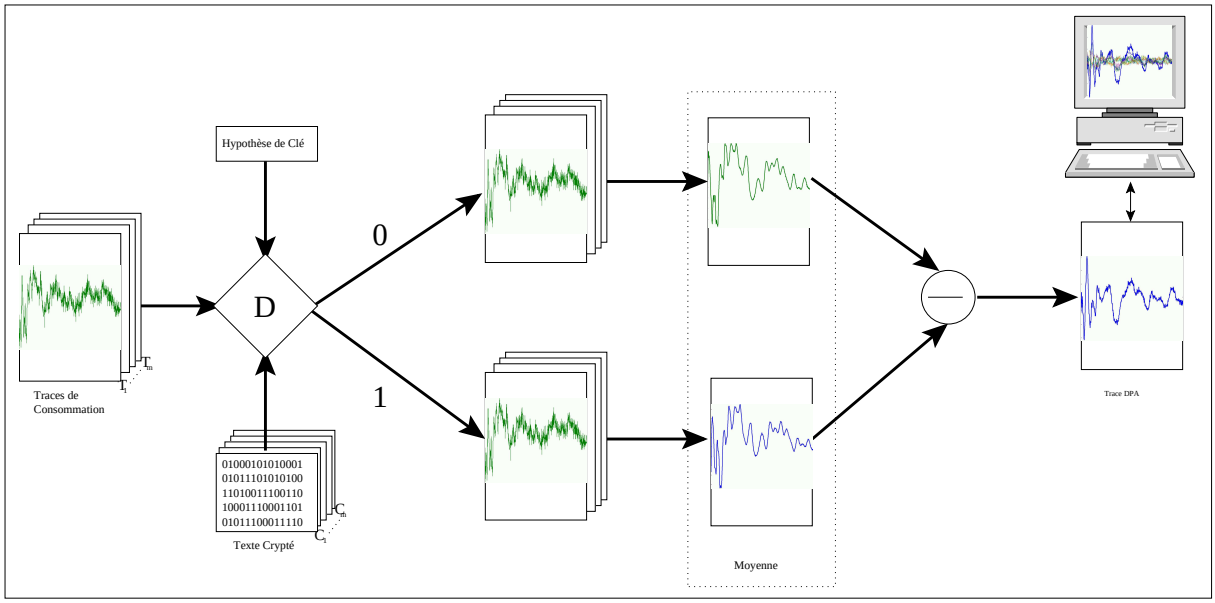


FIG. 3.7: Schéma d'un attaque DPA contre l'algorithme DES

rentielle. Quatre valeurs de b correspondant à chaque S-box fournit la confirmation de l'hypothèse de clé. Rencontrer les huit K_s permet de trouver la sous-clé d'étage à 48-bits. Les 8-bits restants peuvent être trouvés par recherche exhaustive où en applicant la DPA à un autre étage.

Que ce soit pour la DPA ou pour la SPA, des contremesures ont été proposées depuis l'apparition de ces attaques. Elles sont étudiées dans les Sections qui suivent.

3.2 Contremesures

Les contremesures développées pour contrecarrer les attaques DPA sont classées en deux familles. Le premier groupe est composé par les contremesures algorithmiques. L'idée de base donnée par les références [66] [43], [95] et [42] s'agit d'insérer un facteur d'hasard (*randomization*) dans les résultats des calculs intermédiaires, qui se produisent lors des calculs cryptographiques. Si ces contremesures sont bien implantés, la DPA est difficile à réaliser. Cependant, ces *randomisations* sont chères à implanter sur des algorithmes non linéaires tels que le DES ou le AES. Par ailleurs, les approches algorithmiques ne suffisent pas à protéger les circuits cryptographiques contre les attaques HO-DPA *High-Order Differential Power Analysis* [54]. Par conséquent, ce genre de méthode de protection doit être complété par des contremesures matérielles.

La prochaine sous-section montre un état de l'art succinct des contre-mesures afin de contrecarrer la DPA. La Section 3.2.2 propose une contremesure matérielle originale développée dans le cadre de cette thèse. Finalement, la Section 3.2.3 reprend la LRA vue lors du Chapitre 2 pour montrer une contremesure à la fois algorithmique et architecturale.

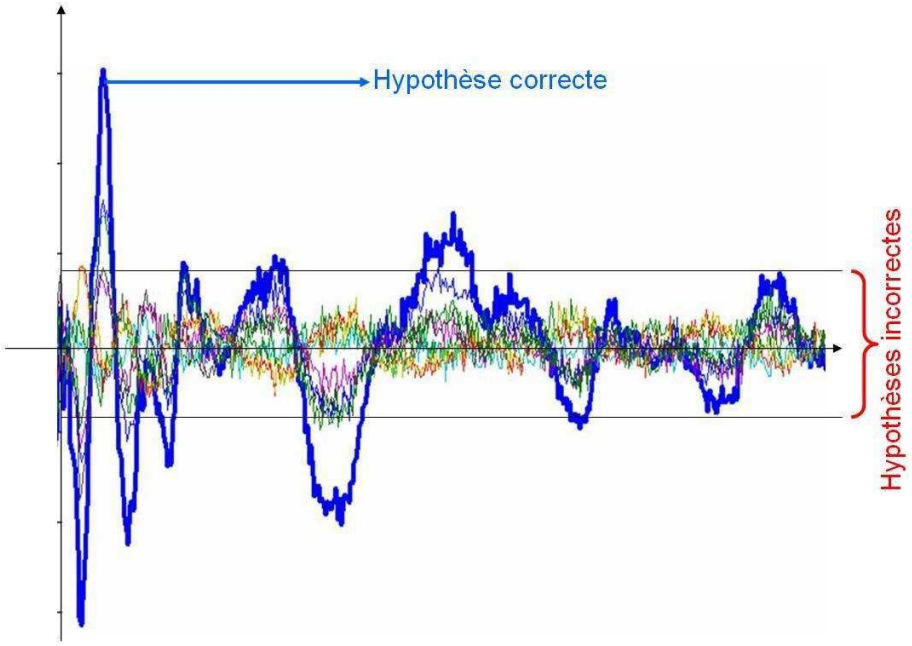


FIG. 3.8: Illustration d'un attaque réussi

3.2.1 État de l'art

Contremesures Algorithmiques. Il existe plusieurs contremesures logicielles (ou algorithmiques) pour rendre les attaques DPA plus difficiles à mener. L'une des premières, la méthode proposée en [43] est efficace contre la SPA et la DPA classique agissant dans des systèmes cryptographiques du type DES. La méthode consiste en trois étapes :

1. **Introduction de délais aléatoires ;**
2. **Substitutions d'instructions ;**
3. **Modification du mode d'opération de l'algorithme cryptographique ;**

Seuls, ces contremesures peuvent être contrecarrées par l'analyse statistique, notamment en augmentant le nombre d'échantillons pour attaque, mais ensemble elles s'avèrent efficaces.

Pour les systèmes à clé publique, la première méthode suggérée par [24] est applicable, et la deuxième est seulement une adaptation de la signature aveugle (*blind signature*[3]) mentionnée auparavant en [20]. La troisième méthode ne fonctionne que pour la cryptographie des courbes elliptiques (ECC - *Elliptic Curve Cryptography*). Pourtant l'attaque pour analyse de consommation raffinée (RPA - *Refined Power Analysis*) récemment proposé [47] outrepasse ces contremesures. D'un autre côté, la méthode BRIP[4] contrecarre la RPA, mais seulement si le système cryptographique cible est le ECC, ne fonctionnant pas pour le RSA.

[3]Une signature aveugle est un type de signature numérique dans laquelle le contenu du message est déguisé avant d'être signé, au travers d'une fonction qui va mêler le facteur d'aveuglement au message.

[4]Malheureusement les auteurs de cette méthode n'expliquent pas la signification du acronyme BRIP.

En ce qui concerne la protection du RSA, la contremesure d'aveuglement de message (*message blinding*) proposé par P. Kocher [53] semble être efficace pour contrecarrer l'attaque MRED (*Modular Reduction on Equidistant Data*) [14][5]. En général, les contremesures protégeant l'algorithme RSA contre les attaques pour analyse de consommation sont basées sur l'aveuglement de messages ou de l'exposant. Ces méthodes peuvent contribuer ou non pour la sécurité du système, selon la façon qu'elles ont été implantées, et selon le type d'attaque. Ce n'est pas rare qu'une contremesure contre une attaque s'avère inefficace contre d'autres, quand elles ne passent pas au champ adversaire. C'est à dire qu'une méthode pour contrecarrer une certaine attaque peut faciliter l'implantation d'une autre.

Néanmoins la manière la plus sûre de contrecarrer les attaques du type DPA reste celle de s'attaquer à son principe : briser la corrélation entre les données calculées et la consommation du circuit. D'une façon différente des travaux qui habituellement proposent l'usage du CRT pour accélérer le RSA (comme [51]), un autre approche suggère une implantation complète du RSA en RNS [6], [21]. Au delà d'accélérer le système, un usage intelligent de cette idée peut être utilisé comme contremesure afin d'empêcher les attaques DPA. Plus de détails sont donnés dans la Section 3.2.3.

Pourtant le problème de toutes ces approches c'est que cela demande le changement complet de l'algorithme en question afin de l'adapter à une arithmétique exotique. Au contraire, les contremesures matérielles ne demandent, a priori, des modifications dans l'algorithme cryptographique.

Contremesures Matérielles Les méthodes matérielles pour contrecarrer les attaques du genre DPA sont assez différentes des approches algorithmiques. Dans le cas des contremesures matérielles, les résultats intermédiaires des calculs cryptographiques ne sont pas pris en compte. Comme alternative, l'idée c'est de cacher une part ou la totalité des informations de consommation à l'aide de différents bruits. Le bruit a une action directe sur le nombre d'échantillons nécessaires a un attaque DPA. La plupart n'empêche pas les attaques DPA, mais rend sa mis en œuvre fort plus compliqué. L'efficacité des contremesures contre la DPA dépend de l'implantation combinée des contremesures algorithmiques et matérielles [10]. Une contremesure matérielle a pour but de diminuer la corrélation entre données d'entrée et la consommation de puissance d'un circuit donné, augmentant ainsi le nombre d'échantillons nécessaires à un attaque DPA. Il y a deux axes principaux de contremesures matérielles.

Le premier consiste en réduire le rapport signal sur bruit (SNR - *Signal to Noise Ratio*). Par définition du SNR, I_c est la consommation de courant du circuit attaqué, dans un instant t. Le bruit causé par la contremesure matérielle s'appelle I_n. La consommation totale de courant peut être écrite comme :

$$I_{total} = I_c + I_n$$

La variable k est l'atténuation du signal occasionnée par le courant I_n. Alors, le rapport signal sur bruit (SNR) est donné pour :

[5] Il s'agit d'un attaque envisageant les implantations du RSA utilisant le CRT.

$$SNR = 20 \times \log\left(\frac{I_c}{kR}\right) \tag{3.1}$$

Diminuer le SNR signifie diminuer la corrélation entre le courant hypothétique correct et la consommation réelle du circuit. Pour réduire le SNR il existe plusieurs méthodes. Les travaux [87] et [69] proposent l'usage d'une logique à double rail balancée. L'idée de base c'est qu'une porte logique devrait consommer une puissance équivalente si elle commute de zéro vers un, ou vice-versa. Le SNR est réduit par cette commutation indépendante des données d'entrée, dont les portes sont décrites dans des nouvelles cellules standards. Malheureusement les expériences montrent que cet objectif n'est que partiellement atteint. La logique à double rail n'est pas suffisante à elle seule pour garantir une signature de consommation complètement indépendante des données traitées [16]. Un problème potentiel c'est que la charge des portes peut différer à cause de différences de routage. La conception de portes à double-rail devrait assurer des charges égales et une consommation de puissance balancée. Pour y arriver, la procédure de groupage de cellules doit être fait très attentivement - ce qui implique un grand effort de développement. Au delà des coûts élevés de fabrication, le résultat final peut demander jusqu'à 15 fois plus de surface et 6 fois plus de consommation comparé à une procédé de synthèse conventionnel.

La deuxième approche matérielle pour prévenir les attaques DPA c'est la réduction de la corrélation entre les données d'entrée et la consommation de puissance à travers de la mise en désordre aléatoire du moment dans lequel les données intermédiaires sont attaquées. Si le temps T_c est différent à chaque trace, la corrélation entre la consommation de puissance hypothétique et la consommation réelle est grandement réduite. La contremesure proposée en [22] réside dans l'insertion de retards aléatoires. La méthode décrite en [10] contrecarre la DPA en utilisant blocs dont la consommation est gérée indépendamment afin de la masquer. Les approches [56] et [48] augmentent la difficulté de réaliser une DPA, mais comme est démontré en [55], même si le calcul direct de la probabilité maximale d'une consommation de puissance arrive dans un temps précis n'est pas pratique, c'est toujours faisable d'approcher cette valeur de façon empirique, basé dans un modèle logiciel de la contremesure.

Dans le cadre de cette thèse il est proposé une contremesure matérielle originale, qui n'essaye pas de "randomiser" la consommation, ni de créer du bruit pour la masquer. Il s'agit, au contraire, de compléter la consommation du circuit cryptographique afin de la rendre constante. L'idée est similaire à celle présentée par Adi Shamir en [91], en ce qui concerne le niveau d'abstraction. Pourtant le circuit décrit par Shamir considère seulement que l'attaquant sonde le V_{cc}, laissant la ligne G_{nd} vulnérable. Par ailleurs, les deux capacités proposées ne sont pas intégrables car trop grandes ($100nF$), sauf si un approche SiP (*System in Package*) est envisageable.

Comme expliqué dans la prochaine sous-section, le circuit proposé par l'auteur de cette thèse masque la consommation si l'attaque est contre le V_{cc} (ou le G_{nd}), et est facilement intégrable dans une technologie CMOS conventionnelle.

3.2.2 Une contremesure originale : le CMG

Basé sur le principe de diminuer le SNR, il a été conçu un circuit analogique capable de masquer la consommation réelle du circuit cryptographique. L'objectif principal de cet approche c'est d'augmenter la sécurité des dispositifs cryptographiques sans aucune modification de l'algorithme de cryptographie implanté. De plus, des nouvelles cellules standards ne sont pas requises, comme c'est le cas pour la logique à double-rail.

Cette technique-ci masque la consommation de puissance à travers la normalisation du courant consommé par le circuit cryptographique (CC). Cette tâche est accomplie par un circuit analogique appelé générateur de masquage de courant (CMG - *Current Mask Generator*), dont le rôle est de maintenir le courant total constant d'un point de vue externe au circuit. Pour projeter le CMG d'abord des mesures de consommation dans un circuit cryptographique conventionnel ont été prises, afin d'établir quel était le pic de consommation de courant de ce circuit. Une fois cette valeur détectée, l'intention est de rester au niveau de ce pic pendant tout le fonctionnement du CC, même quand il consomme moins.

Le CMG est composé fondamentalement d'un miroir de courant, d'un circuit suiveur, et d'une capacité. Comme peut être vu dans la Figure 3.9, le CMG agit à côté du CC, et cela souligne qu'aucun changement dans le circuit cryptographique est nécessaire. Il faut noter que le CMG et le CC, par un souci pédagogique, ne sont pas représentés à la même échelle. En effet, le CMG n'augment en moyenne que 30% la surface d'une implantation standard du algorithme DES, par exemple.

Dans la Figure 3.10, le miroir de courant agit en imposant un courant fixe (I_2). Ce courant I_2 est donné par le coefficient w des transistors P_1 et P_0, comme dans l'équation 3.2, et est égal au pic de consommation du CC :

$$I_2 = F\left(\frac{wP_1}{wP_0}\right) \tag{3.2}$$

Le circuit cryptographique consomme un courant I_c. Quand $I_c = I_2$, cela signifie que le CC consomme tout le courant fourni, et le CMG reste en mode d'attente. Autrement, quand le CC ne nécessite pas tout le courant I_2, alors le circuit suiveur réalise une boucle de rétroaction afin de consommer le courant restant I_L, de façon que $I_L = I_2 - I_c$.

Le suiveur joue le rôle d'un générateur de tension. L'amplificateur opérationnel reçoit une tension du miroir et la compare avec une tension de référence (par exemple V_{ext}). Si le circuit cryptographique consomme une quantité de courant plus petite que I_2, la tension dans l'entrée de l'amplificateur opérationnel est plus basse que la tension de référence. Donc la sortie de l'amplificateur opérationnel envoie 0 au transistor P_4. Alors il consomme un courant I_L, qui est la différence entre I_2 et I_c. Quand le CC consomme à son pic (i.e. $I_2 = I_c$), l'amplificateur opérationnel envoie un 1 vers le transistor P_4, interrompant son fonctionnement, puisqu'il n'est plus nécessaire drainer le courant. Finalement, la fonction de la capacité de $9,5pF$ c'est de donner du temps à la boucle de rétroaction de réagir. La capacité sert aussi à lisser la tension, ce qui a un effet bénéfique au masquage de consommation.

Afin de valider la méthode CMG, il a été utilisé une S-Box de l'algorithme DES dans le rôle du

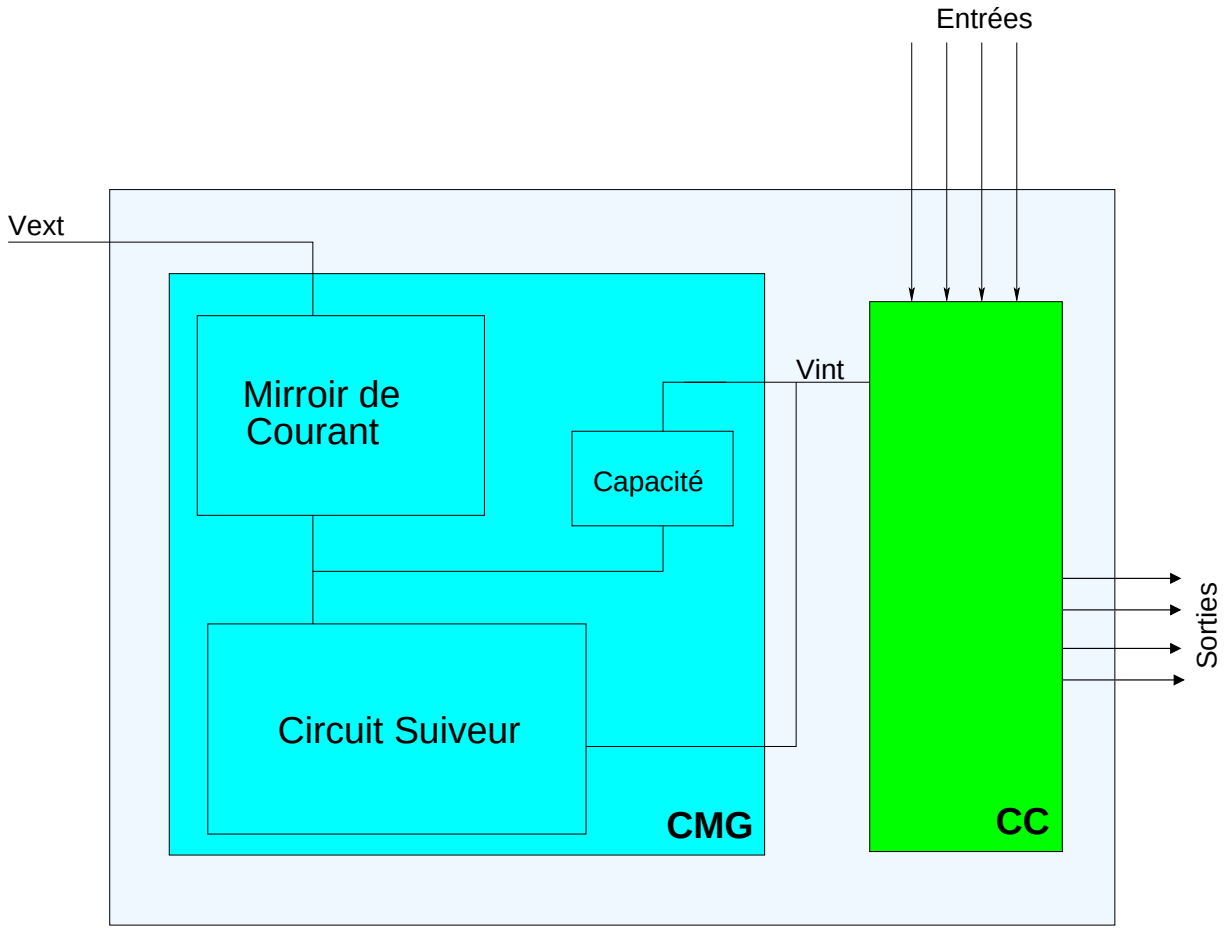


FIG. 3.9: Schéma de principe du CMG.

circuit cryptographique (CC). Une simulation électrique (à l'aide de l'outil SPICE[6]) de la S-box a été réalisée pour déterminer son pic de consommation dans le pire cas. La Figure 3.11 montre que cette consommation est de $6mA$. Plusieurs simulations ont été faites pour différents scénarios de données. Comme peut être vue dans la Figure 3.12, le CMG fonctionne de manière efficace, masquant la consommation du CC, rendant la DPA plus difficile à réaliser. Le signal $R8/Plus$ est le courant consommé par le CC, et le signal $R4/Plus$ est le signal masqué.

La Figure 3.13 montre le courant masqué qui peut être sondé dans le V_{dd} ou dans le G_{nd} ; le courant consommé par le circuit cryptographique, et deux signaux d'entrée a_0 et a_1. Analysant la consommation par rapport aux données d'entrée, la Figure 3.13 montre aussi que les commutations de zéro à un et vice-versa n'altèrent pas la consommation finale, c'est à dire, la consommation reste constante du point de vue d'un attaquant. Pour définir le degré de difficulté aux attaques d'analyse de courant apporté par le CMG, quelques paramètres doivent être considérés. Le premier c'est le rapport signal sur bruit. Au contraire des applications habituelles, tels que le multimedia, où les concepteurs essayent d'augmenter le SNR, l'approche CMG tente l'opposé : diminuer le plus

[6]Cet outil a été développé dans l'université de Berkeley (http://www.eecs.berkeley.edu).

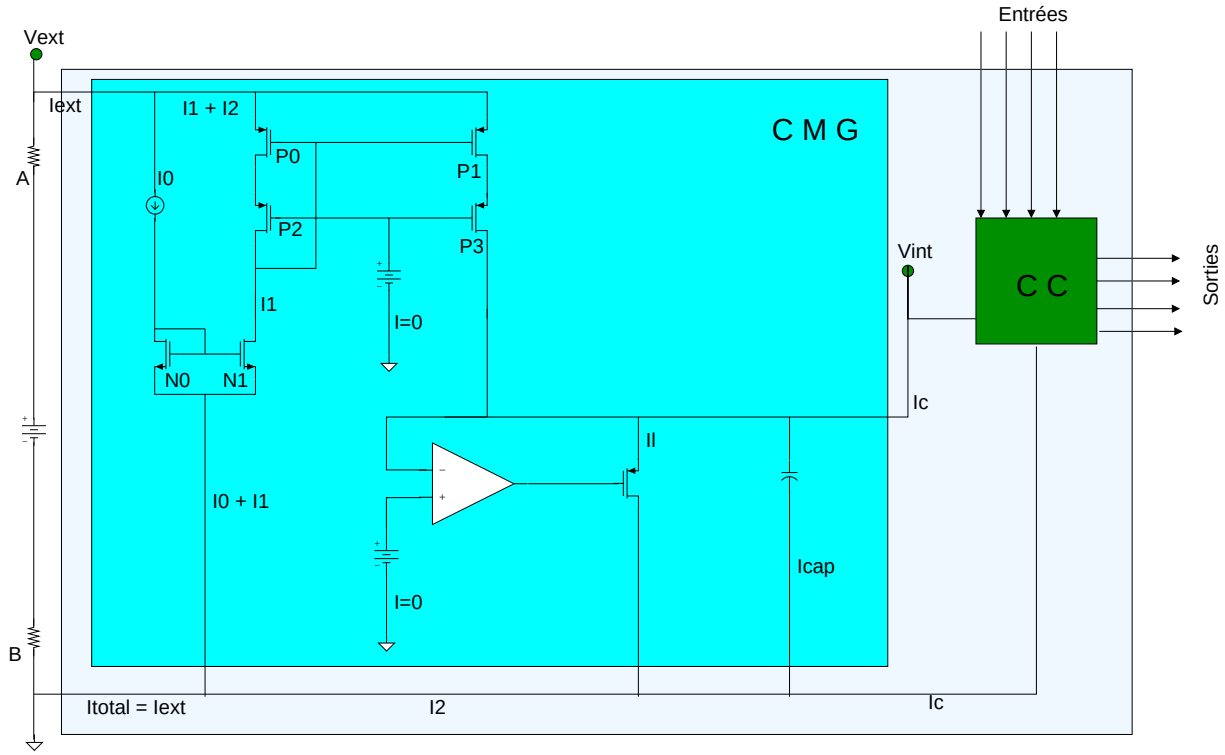


FIG. 3.10: Vue d'ensemble du CMG.

possible le SNR.

Les Figures 3.12 et 3.13 montrent des *glitches* dans le signal masqué $R4/Plus$. Si un zoom suffisamment puissant peut être réalisé, la même forme du signal $R8/Plus$ sera retrouvée. Cela signifie que le système n'est pas parfait. Mais, si les valeurs de chaque signal sont considérés (voir Figure 3.14), il est evident que le CMG atténue de façon non négligeable : le courant est réduit d'un facteur $k \approx 25$.

Ce facteur k est obtenue en mesurant la différence entre le pic et le point le plus faible du signal $R8/Plus$ (encore dans la Figure 3.14), ce qui donne une variation ΔCC. Ensuite, le procédé est répété pour le signal $R4/Plus$, d'où il est obtenu ΔCMG. Donc :

$$k = \left(\frac{\Delta CC}{\Delta CMG}\right) \tag{3.3}$$

Afin de voir l'atténuation générée par le CMG, le SNR montré dans l'équation 3.1 doit être étendu vers l'équation 3.4 :

$$SNR_{CMG} = 20 \times \log\left(\frac{I_c}{k \times N}\right) = 20 \times \log\left(\frac{I_c}{N}\right) - 20 \times \log(k) = SNR - 20 \times \log(k) \tag{3.4}$$

A partir de l'équation 3.4 et en regardant la Figure 3.14, pour l'exemple donné, le courant visualisé par l'attaquant est atténué de $\approx 25db$. Cela signifie que le signal est complètement noyé par le bruit, comme illustré par la Figure 3.15.

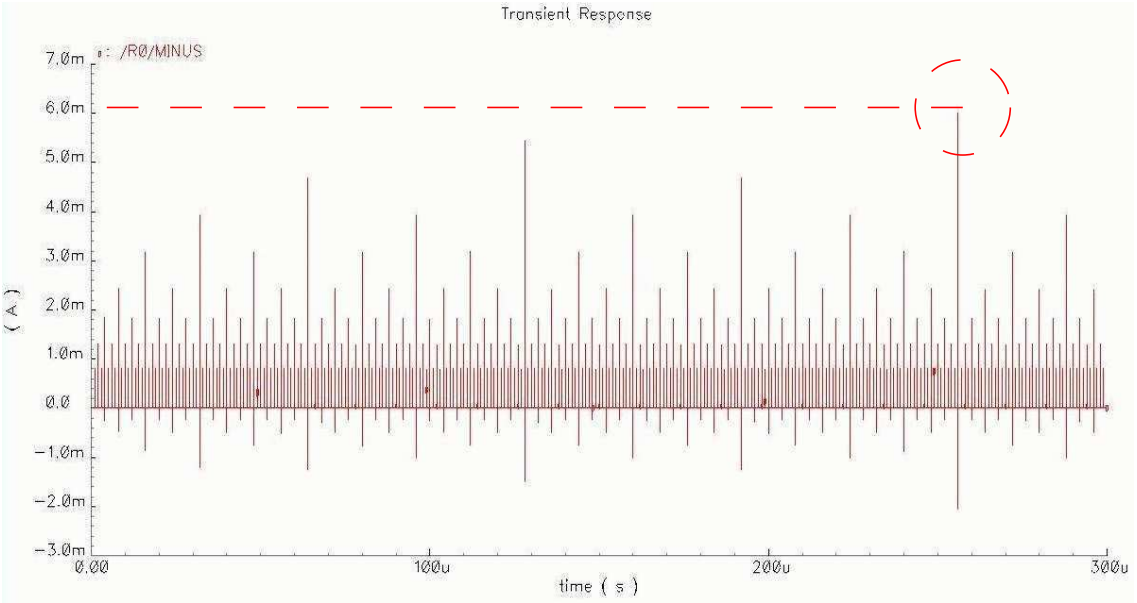


FIG. 3.11: La consommation de courant de la S-Box

Conclusion. Le CMG est une contremesure qui augmente la robustesse des circuits cryptographiques contre attaques par analyse de puissance. C'est une solution bas niveau qui a pour atout le fait de ne pas modifier l'algorithme ni les données du système cryptographique. Cette méthode est simple à implanter, et n'est pas coûteuse du point de vue de la surface requise. Un implantation classique du DES prend environ $16mm^2$ (synthèse avec technologie $CMOS0.35\mu$, pendant que le CMG requiert seulement $5mm^2$, ce qui est un coût tout à fait acceptable quand les enjeux sont la sécurité et la robustesse des systèmes de cryptographie.

Un autre coût évident de cet approche c'est l'augmentation de la consommation. Mais la méthode CMG reste intéressante pour des applications comme les cartes bancaires, *set-top boxes*, cartes téléphoniques, parmi d'autres, où la faible consommation n'est pas le principal sujet. Dans des opérations bancaires, comme la retraite d'argent dans des distributeurs automatiques, l'opération de cryptographie ne dure que quelques secondes, et la sécurité est plus important, donc la faible consommation n'est pas forcément le premier problème à régler. Le plus important dans ce cas c'est la sécurité. L'atténuation occasionnée par le CMG garantit cette sécurité.

Finalement, cette atténuation peut être augmentée. En modifiant la boucle de rétroaction et en améliorant le miroir de courant, le SNR peut encore être diminué. Un approche possible c'est l'inclusion d'une inductance en série avec le miroir de courant. Dans les derniers expériences, des résultats préliminaires ont montré une atténuation plus importante que celle des versions antérieures du CMG. De plus, il reste à vérifier si l'usage d'une inductance peut diminuer les émissions électromagnétiques, ce qui implique que le CMG peut aussi être efficace contre les attaques du type EMA.

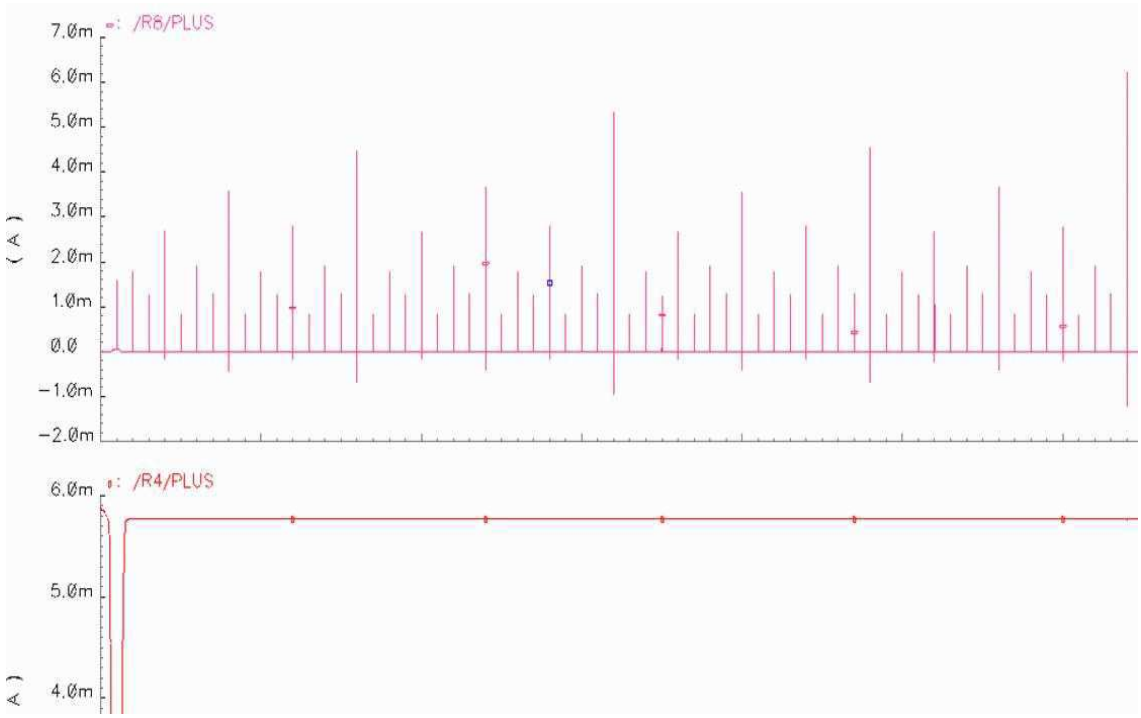


FIG. 3.12: La consommation de courant du circuit cryptographique (R8 Plus) et le courant généré par le CMG (R4 Plus)

3.2.3 Aspects de sécurité liés à la LRA

Au delà des gains de performance, l'utilisation du RNS pour réaliser des opérations telles que l'exponentiation modulaire, permet la possibilité d'additionner un facteur aléatoire dans les calculs à travers d'un choix aléatoire des bases. A chaque modification de base, même si les données d'entrée, notamment la clé cryptographique, restent les mêmes, tous les résultats intermédiaires seront modifiés, et par conséquent, la consommation associée à ces données change elle aussi.

La LRA propose deux approches pour apporter des aspects aléatoires aux calculs : le premier au niveau circuit (aléatoire spatial) et le deuxième au niveau données (masquage arithmétique). La combinaison des deux représente un bon compromis entre sécurité et coût d'implantation. Ces approches sont :

- **Choix aléatoire des bases initiales :** Le facteur d'hasard est donné par un choix aléatoire des éléments des bases β_1 et β_2 avant l'exponentiation modulaire.
- **Changement aléatoire des bases pendant le calcul :** L'algorithme proposé en [7] offre plusieurs degrés de liberté dans l'implantation et plusieurs niveaux de sécurité.

Le principal but de ces approches c'est de rendre les résultats intermédiaires aléatoires, rendant aussi aléatoire la consommation. Basée sur le principe de la corrélation entre données et consommation, la DPA n'a plus raison d'être. La sécurité de cette méthode a été démontrée en [7].

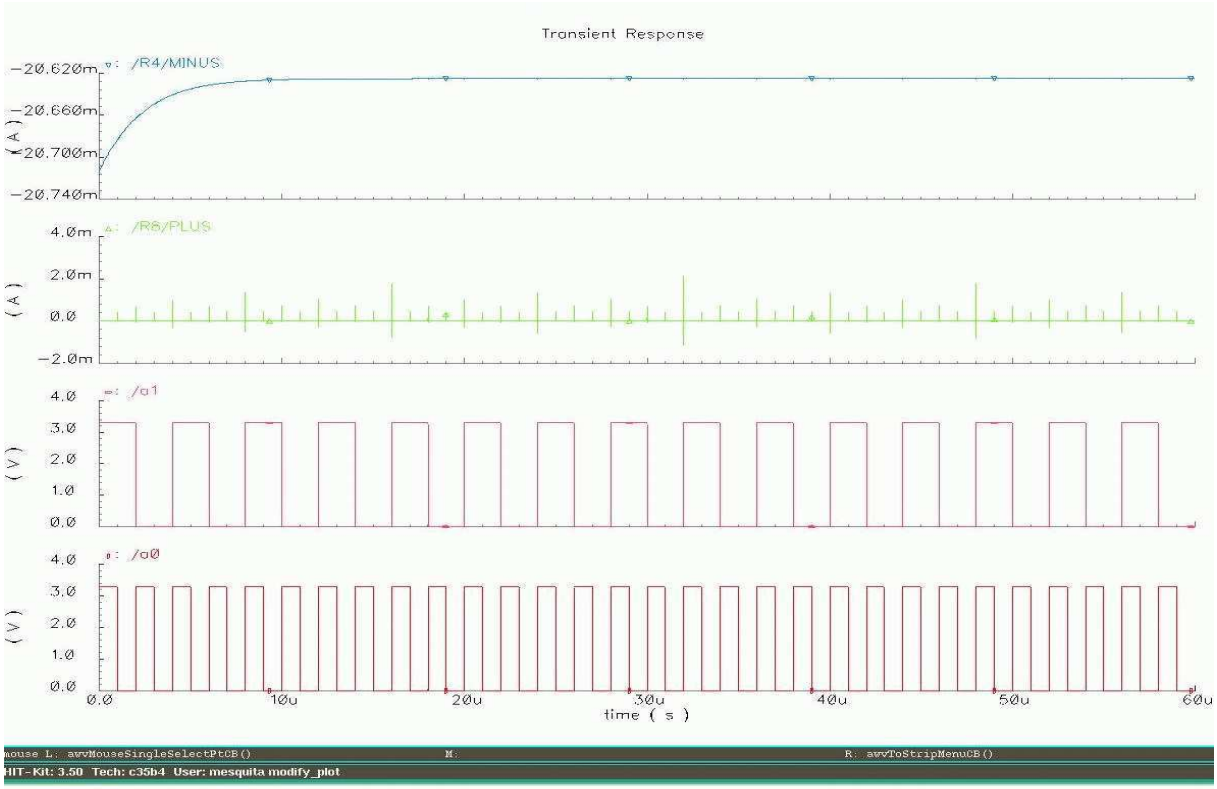


FIG. 3.13: Le courant du CC et du CMG par rapport à quelques entrées de données

3.3 Conclusion du Chapitre

Ce Chapitre a présenté les principes des attaques par canaux cachés, notamment les attaques par analyse de courant, ainsi que des contremesures algorithmiques et matérielles afin de contre-carrer ces attaques.

L'attaque par analyse différentielle de consommation est l'un des attaques para canaux cachés les plus efficaces, et peu coûteux à mettre en place. C'est pour cela qu'il a été détaillé. L'étude de l'attaque DPA e ses contremesures a permis d'envisager des méthodes originels afin d'accroître les défenses des systèmes cryptographiques matérielles.

La première contremesure est le CMG, un circuit analogique qui masque la consommation réelle d'un circuit cryptographique numérique. Le CMG s'est avéré une contremesure simple à implanter, de demandant aucun changement ni au circuit cryptographique, ni à l'algorithme de cryptographie implanté dans ce circuit. Le CMG a été co-simulé avec le circuit numérique et a été synthétisé. Il y a des indications claires de son efficacité, mais il reste la tâche de réaliser un attaque DPA sur l'ensemble afin de prouver sa robustesse.

Autre aspect à considérer c'est la possibilité d'approfondir l'étude du CMG sous un prisme automatique. Des questions comme la régulation de la précision, stabilité et dynamisme restent ouvertes. Il est possible qu'avec un système de contre-réaction correct, le CMG puisse fonctionner

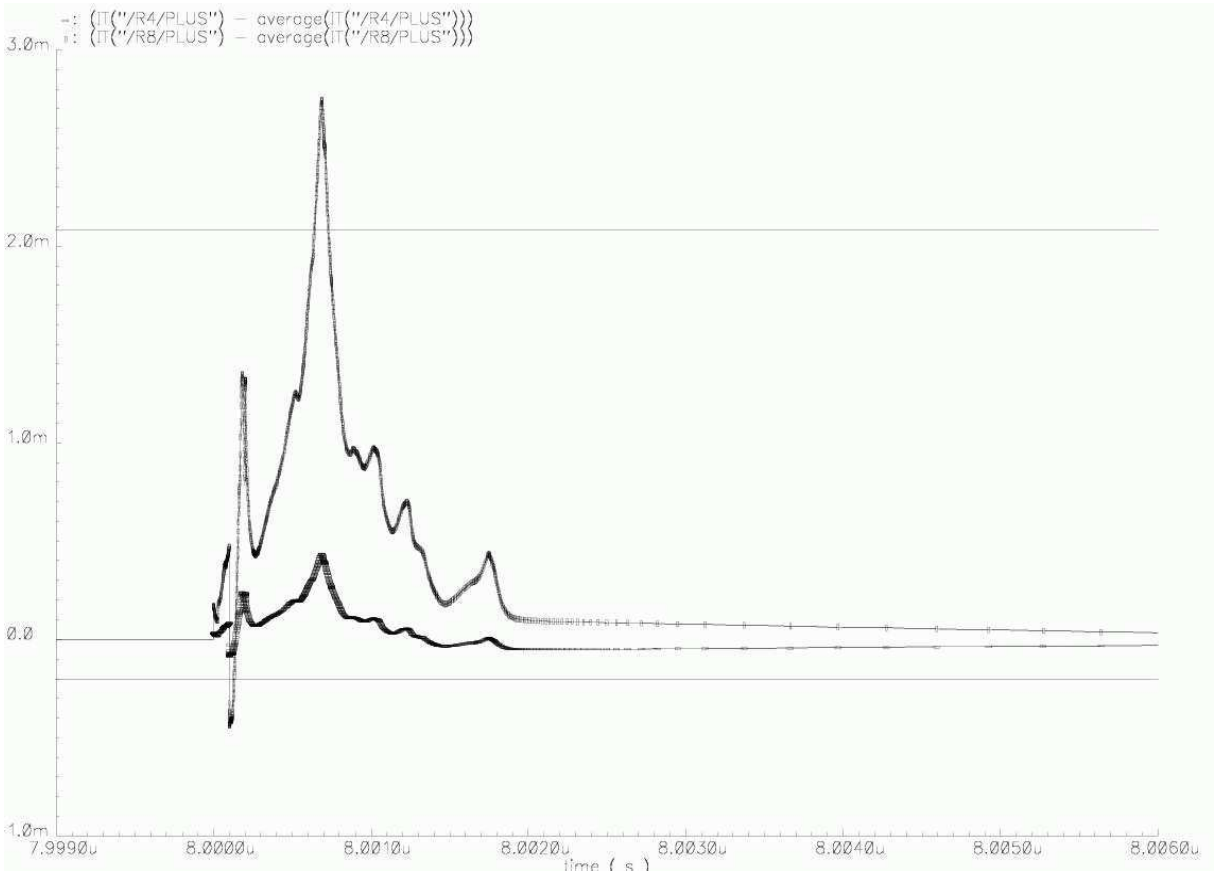


FIG. 3.14: L'atténuation du signal occasionnée par le CMG

sans avoir recours à la capacité afin de réduire son coût matériel.

Le CMG peut aussi être combiné avec la logique double rail proposée en [79]. Comme la logique double rail diminue considérablement les pics de consommation, le CMG peut s'avérer une contremesure complémentaire, éliminant les traces résiduels de signature de consommation.

La deuxième contremesure s'agit de la LRA. Cette méthode consiste en utiliser le changement de bases d'une représentation RNS afin de modifier la consommation intermédiaire du circuit. C'est une méthode algorithmique qui avait été proposée récemment par une équipe du LIRMM, mais qui n'avait pas encore été implantée en matériel. La suite de cette thèse traite de l'implantation de la LRA.

Possédant des connaissances des bases arithmétiques de la cryptographie, et étant à l'aise au sujet des attaques par analyse de courant et ses contremesures (tant algorithmiques que matérielles), il a été possible de concevoir une architecture reconfigurable - à la fois flexible et robuste - pour la cryptographie. Cette architecture et son modèle de programmation sont discutés dans le prochain chapitre.

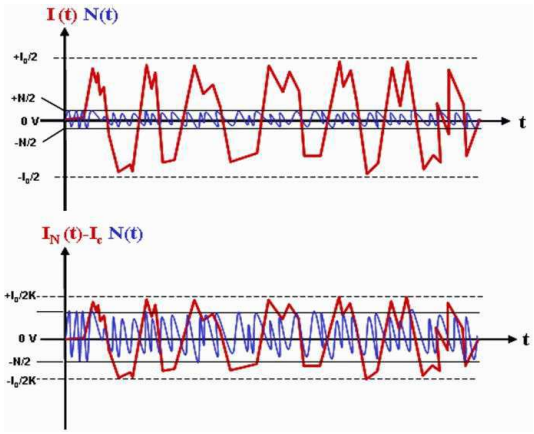


FIG. 3.15: Consommation normale et bruit versus la consommation noyé par le bruit à cause du CMG

CHAPITRE 4

Vers une Implantation Matérielle du LRA

"L'homme raisonnable s'adapte au monde ; l'homme déraisonnable s'obstine à essayer d'adapter le monde à lui-même. Tout progrès dépend donc de l'homme déraisonnable." ***Bernard Shaw***

Ce chapitre est consacré à l'architecture développée à partir des tendances observés dans le chapitre 1, tout en envisageant la robustesse du système selon les techniques discutées dans le chapitre 3. L'architecture a été développée en VHDL et synthétisée dans la technologie $CMOS$ $0.35\mu m$. La validation a été réalisé à l'aide d'outils tels que Cadence et un prototype (réduit) a été implanté sur FPGA.

4.1 Démarches architecturales

Le Chapitre 2 a mis en évidence les opérations arithmétiques nécessaires aux applications cryptographiques. Par ailleurs la Section 3.1 a guidé ce travail vers une approche préoccupée à propos des aspects liés à la sécurité, plutôt que à l'économie de la surface. Pour l'instant le domaine d'application de cette architecture est la cryptographie, surtout celle qu'utilise des opérateurs issus de l'arithmétique modulaire.

Le point de départ de la route vers une architecture reconfigurable pour la LRA était une implantation matérielle de cette méthode. Pour plus de clarté, l'Algorithme 8 qui calcule la multiplication modulaire de Montgomery en RNS, et qui est la base de la LRA, est rappelé ici :

Algorithme 10: Rappel de l'Algorithme de Montgomery pour la Multiplication Modulaire en RNS

Entrée: β_1, β_2 ; N dans les deux bases tel que $0 < 4 \cdot N < M_1, M_2$ et $PGCD(N, M_1) = 1$; A, B dans les deux bases, tel que $A \cdot B < M_1 \cdot N$
Sortie: R tel que $R \equiv \left[A \cdot B \cdot M_1^{-1}\right]_N$
MMMRNS(A, B, M)
1: $T \leftarrow A \otimes_{rns} B$ *in* $\beta_1 \cup \beta_2$
2: $Q \leftarrow T \otimes_{rns} (-N_1^{-1})$ *in* β_1
3: *Etendre* Q *de* β_1 *vers* β_2
4: $R \leftarrow (T \oplus_{rns} Q \otimes_{rns} N) \otimes_{rns} (M_1^{-1})$ *in* β_2
5: *Etendre* R *de* β_2 *vers* β_1

Dans l'Algorithme 8, les opérations réalisées dans les lignes 1, 2 et 4 sont des additions et des multiplications en RNS et donc, conformément vue dans la Section 2.6.4, il s'agit d'opérations qui peuvent être exécutées totalement en parallèle. Cela induit une architecture dotée de plusieurs éléments de calcul homogènes. Si cette architecture possède autant d'éléments de calculs que le nombre de bases RNS, les opérations de multiplication se font en seulement 5 cycles d'horloge, conformément à l'Algorithme 11.

Pourtant les points critiques de l'Algorithme 8 sont les étapes 3 et 5, car ils s'agissent des extensions de base MRS. Comme exposé dans le Paragraphe 2.8, la complexité de cette opération est de $O(k \times (k-1))$, une fois qu'il y a une forte dépendance de données qui ne permet pas la complète parallélisation de l'opération. Voici un exemple d'extension de base pour un $k = 4$:

$$
\begin{aligned}
t_1 &= |q|_{m1} = q_1 \\
t_2 &= \left| |(q_2 - t_1)|_{m_1} |_{m_2}^{-1} \right|_{m_2} \\
t_3 &= \left| \left(\left((q_3 - t_1) \, |m_1|_{m_3}^{-1} \right) - t_2 \right) |m_2|_{m_3}^{-1} \right|_{m_3} \\
t_4 &= \left| \left(\left(\left(\left((q_4 - t_1) \, |m_1|_{m_4}^{-1} \right) - t_2 \right) |m_2|_{m_4}^{-1} \right) - t_3 \right) |m_3|_{m_4}^{-1} \right|_{m_4}
\end{aligned}
\tag{4.1}
$$

Chaque étape de l'extension de base dépend des étapes précédentes, mais à partir du moment où le premier calcul est fini, tous les autres peuvent donc commencer. Cela indique la possibilité d'implémenter le MRS dans un pipeline, afin d'accélérer son exécution.

L'autre constatation possible est que dans les lignes 1, 2 et 4 de l'Algorithme 8, les opérations réalisées sont de nature différente de celles exécutées dans les lignes 3 et 5. Comme résultat, le comportement du circuit implantant la *LRA* doit changer pendant le cours de son exécution. Cela met en évidence l'intérêt pour une architecture capable d'être reconfiguré dynamiquement.

Considérant cette analyse il est possible de proposer une première approche architecturale afin d'implanter la LRA. La Figure 4.1 donne un premier aperçu du circuit qui calcule la LRA. Il faut noter que les inverses modulaires nécessaires aux différents opérations sur RNS sont pre-calculées et ensuite stockés dans le circuit. Les bases RNS sont aussi générées en avance.

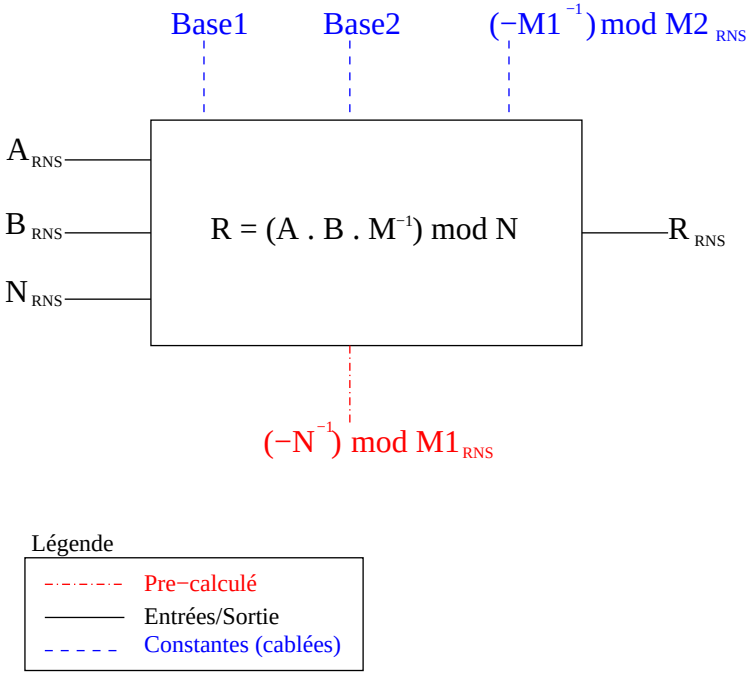


FIG. 4.1: Aperçu de la première implantation de la *LRA*.

Cet approche permet l'implantation de la LRA de façon séquentielle ou combinatoire, comme peut être vue dans la Figure 4.2. Cette Figure prend en compte l'exécution de l'algorithme 11 [8]. Dans le deuxième cas les blocs logiques sont re-utilisés à l'aide d'un bloc de contrôle.

Les variables a, b et m sont sur t bits, et m est construit de manière que $m = 2^t - c$ - où c est une valeur comprise entre 0 et $2^{\frac{t}{2}}$. Agissant de cette façon, les calculs sont simplifiés, et par conséquent la surface des PEs[1] est réduite, ce qui abouti dans les résultats d'implantation montrés dans le Tableau 4.1. Ainsi, il est possible d'effectuer les 3 opérations élémentaires sans réaliser de comparaison avec les moduli m_i. Par exemple le calcul $a_i \times b_i \ mod \ m_i$ se calcule à l'aide de l'algorithme 11.

[1]Selon la dénomination classique, ce genre de bloc est appelé *Processing Element*, d'où l'abréviation PE.

Algorithme 11: Multiplication RNS

Entrée: $0 \leq a, b < m_i$, où $m_i = 2^t - c_i$ et $c_i < 2k$, avec $1 < k < (t-1)/2$
Sortie: $r = a \times b \ mod \ m$
MODPROD(a, b, m_i)
1: $p \leftarrow a \times b$
2: $p_1 \leftarrow a/2^t$
3: $p_0 \leftarrow p \ mod \ 2^t$
4: $p' \leftarrow c_i \times p_1 + p_0$
5: $p'_1 \leftarrow p'/2^t$
6: $p'_0 \leftarrow p' \ mod \ 2^t$
7: $p'' \leftarrow c_i \times p'_1 + p'_0$
8: $\rho \leftarrow p'' + c_i$
9: **if** $p \geq 2^k$ **then** $r \leftarrow \rho \ mod \ 2^k$
10: **else** $r \leftarrow p''$
11: **return** (r)

Il est possible de remarquer que le produit modulaire est réalisé sans comparaison, et qu'il implique trois multiplications (lignes 1, 4 et 7 de l'algorithme 11). La valeur c_i peut être choisie de façon à simplifier les calculs de $c_i \times p_1$ et $c_i \times p'_1$. La méthode pour les simplifier c'est de regarder la composition binaire des c_i. Si les bits qui les constituent sont dans sa majorité des 0's, avec peu de 1's, ces multiplications peuvent être implantées comme des additions-décalages. Dans les résultats présentés dans le Tableau 4.1, les c_i's comprenaient seulement trois 1's.

Le circuit montré dans la Figure 4.1 est composé par plusieurs éléments de calculs, appelés PEs. Les PEs sont capables de réaliser les opérations suivantes :

- $a \times b \ mod \ m$
- $a + b \ mod \ m$
- $a - b \ mod \ m$
- *nop*

Chaque élément de calcul (PE) opère en parallèle, puisqu'ils implantent une application basée sur le RNS, et par conséquent, sans propagation de retenue. La Figure 4.3 illustre cette notion, dans une implantation à 4 PEs.

Ces démarches ont permis l'analyse de la faisabilité de l'implantation de la LRA en matériel, et ont guidé vers une architecture plus flexible, capable d'implanter d'autres algorithmes de cryptographie que le RSA. La prochaine Section aborde ce sujet.

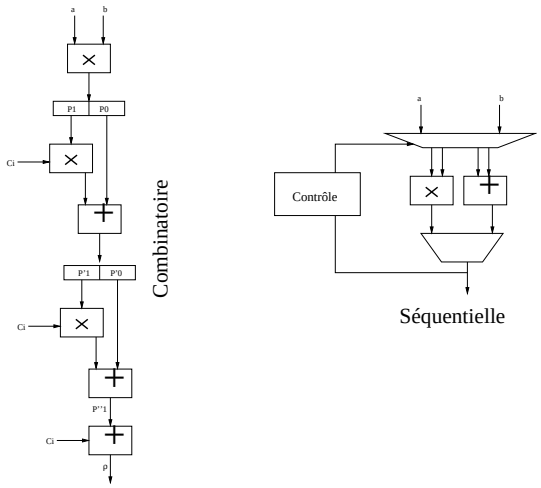


FIG. 4.2: La *LRA* peut être implantée de façon combinatoire ou séquentielle.

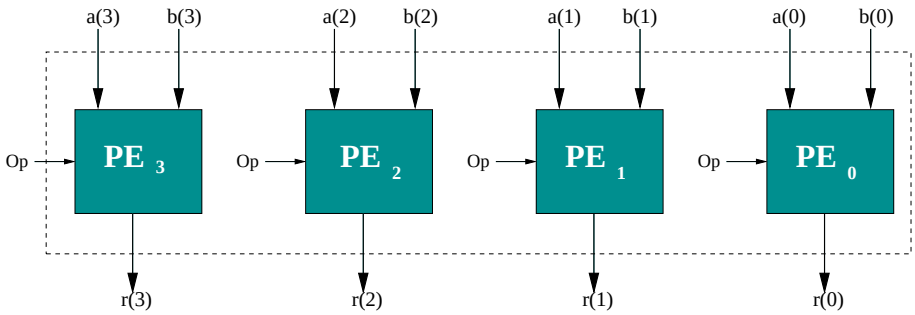


FIG. 4.3: Une version du circuit pour la LRA à 4 PEs

4.2 Leak Resistant Reconfigurable Architecture : Vue d'ensemble

L'idée de base de cette architecture est d'implanter la LRA pour le système cryptographique RSA, tout en restant suffisamment flexible pour être capable d'exécuter d'autres algorithmes cryptographiques basés sur l'arithmétique modulaire, par exemple la cryptographie par courbes elliptiques (ECC - *Elliptic Curve Cryptography*). Afin de pourvoir à cette caractéristique, ce travail s'inspire dans les architectures reconfigurables étudiées dans le Chapitre 1, spécialement sur celles à gros grain. La *Leak Resistant Reconfigurable Architecture* (dorénavant appelée LR^2A) a comme éléments principaux les modules suivantes :

TAB. 4.1: Résultats d'Implantation de la LRA sur FPGA

t (bits)	Version	Multiplieurs	LUTs	Fréquence
32	Combinatoire	12	828	13MHz
32	Séquentielle	4	931	44MHz

Contrôleur de configurations et injecteur de données :

Le but d'un contrôleur de configurations est de réduire le fossé entre l'architecture reconfigurable et le processeur qui la contrôle. Implanter ce contrôleur dans la seule et même puce que l'architecture reconfigurable diminue le coût de communication vis à vis d'un approche où le contrôleur serait implanté dans une puce à part, notamment dans un ordinateur hôte.

Éléments de calcul homogènes à gros grain :

La flexibilité matérielle fournie par les architectures reconfigurables à petit grain (FPGA) est limitée par un faible débit et un traitement des données au niveau bit. De plus, les architectures reconfigurables à gros grain permettent que la reconfiguration dynamique soit réalisée à chaque cycle d'horloge, ce qui est important dans une application cryptographique, où il est habituel de changer d'un algorithme de cryptographie asymétrique vers un algorithme symétrique pendant une même session de communication[2].

Réseau d'interconnexion :

L'utilisation du seul RNS dans les calculs permettrait d'avoir un schéma de communication simple, car il n'y a pas de propagation de retenue, et donc, pas de nécessité d'échanger d'informations parmi les éléments de calcul. Pourtant la LRA requiert aussi des extensions de base, réalisées au travers de la technique du *Mixed Radix System*, qui possède des opérations complexes du point de vue de la communication. C'est pour cela qu'un réseau de communication flexible se fait nécessaire.

Ressources de mémoire :

La *Leak Resistant Arithmetic* utilise plusieurs valeurs pré-calculées, tels que les inverses modulaires et les moduli pour les bases RNS. Toutes ces données ont besoin d'être dans la puce au moment des calculs. Comme les PEs doivent constamment lire ces données et écrire des résultats intermédiaires dans une mémoire, cela induit que cette mémoire soit placée à côté de chaque PE. Les mémoires locales ont aussi l'avantage d'éviter la gestion complexe des conflits d'accès inhérents à une mémoire centralisée.

Un modèle de gestion de configurations :

Souvent l'efficacité d'une architecture reconfigurable est préjudiciée par l'absence d'une méthode de contrôle de configurations et d'injection de données, car cet absence conduit à un impor-

[2]Normalement un algorithme à asymétrique (par exemple le RSA) est utilisé pour réaliser la signature numérique et ensuite pour chiffrer une clé privée d'un algorithme symétrique. Dans un deuxième moment, la communication entre les deux parties est cryptée à l'aide d'un algorithme à clé privé (par exemple le RC6), car ces dernières sont plus efficaces en termes de débit. Cela induit à l'utilisation de la reconfiguration.

tant effort de programmation manuelle. Pour cela la LR^2A est proposée avec un modèle de gestion de configurations.

Le contrôleur est un processeur intégré à l'architecture (et non seulement un processeur hôte). Il a pour but gérer le schéma de reconfiguration et le flot de données. Le contrôleur utilisé est un processeur du type RISC (microprocesseur à jeu d'instruction réduit - *Reduced Instruction-Set Computer*) doté d'un compilateur *C*.

Le bloc de base est un élément de calcul à gros grain [3] capable de gérer des boucles et des sauts, aussi bien que les accès mémoire. Chaque PE incorpore des opérateurs classiques, habituellement identiques à ceux trouvés dans des processeurs du type RISC, augmenté d'instructions spécifiques à la cryptographie. Par rapport à des architectures à petit grain, les PEs remplacent les CLBs, et en comparaison à des architectures à gros grain classiques, la contribution de ces nouveaux PEs (en plus des opérateurs pour la cryptographie) c'est leur capacité de gérer des structures de contrôle plus complexes.

L'ensemble de blocs est lié à travers un réseau d'interconnexion reconfigurable. Ce réseau permet la communication point-à-point ou par *broadcast*. Une vue d'ensemble de l'architecture (PEs, mémoire, processeur hôte, réseau d'interconnexion), telle qu'elle a été implantée, peut être observée dans la Figure 4.4.

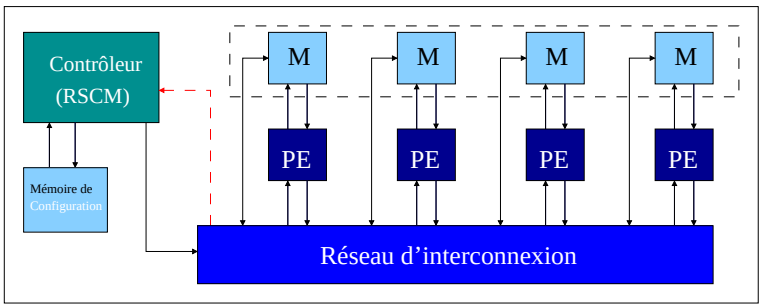


FIG. 4.4: Vue d'ensemble de l'architecture LR^2A

La mémoire est distribuée, et chaque PE accède à sa portion de mémoire de façon locale. Le Contrôleur peut écrire dans les mémoires distribuées pour mener une nouvelle configuration ou pour injecter des nouvelles données.

L'architecture est gérée à travers d'un modèle de contrôle pour les architectures reconfigurables appelé RSCM (*Reconfigurable Systems Configuration Model*). Une version de ce modèle, adapté aux FPGAs a été publiée en [18]. Dans le cadre de cette thèse le modèle à évolué afin d'apporter le contrôle nécessaire à une architecture comme la LR^2A.

Des détails architecturaux sont exposés par la suite ; aussi bien que le fonctionnement de la gestion de cette architecture.

[3] En effet, dans le future sera un bloc à grain mixte petit / gros, car il existe des algorithmes symétriques de cryptographie très répandus, comme le AES, qu'utilisent des opérations à petit grain.

4.3 La composition de la LR^2A

Comme exposé dans la Section précédente, la LR^2A est composé de cinq éléments disposés selon différents niveaux d'abstraction. Le niveau matériel consiste en des éléments architecturaux tels que les blocs de base, le processeur chargé de contrôler le système, des ressources de mémoire et des ressources d'interconnexion. Cependant le niveau logiciel traite l'injection des données et contrôle le schéma de configuration. Tous ces éléments sont présentés en détails ci-dessous.

4.3.1 Le contrôleur

La densité fonctionnelle (Section 1.3.1) étant l'une des principales métriques pour analyser la performance d'une architecture reconfigurable, la façon de gérer les configurations devient un enjeu majeur. Comme précise Benoît en [11], la reconfiguration dynamique ne veut pas dire reconfiguration systématique. Vis à vis de la densité fonctionnelle, cela signifie qu'un contrôleur servant d'interface entre le processeur host et le dispositif reconfigurable doit gérer le flot de configurations et l'injection de données, afin que le temps dépensé pour reconfigurer le dispositif soit faible par rapport aux quantités de données à traiter. Dans cette thèse, le rôle du contrôleur est semblable à celui décrit en [28], où le processeur chargé du contrôle réalise la gestion des configurations, tout en minimisant les accès à la mémoire et le décodage des instructions.

Le contrôleur ici utilisé est un processeur RISC, appelé Plasma [80], proche de l'architecture du processeur MIPS [46] et disponible sur internet sous forme de code source VHDL, ainsi qu'avec des outils de compilation.

Le Plasma a été synthétise, et fonctionne à une fréquence de 25MHz dans une technologie $CMOS$ $0.35\mu m$. Ce processeur permet l'implantation en C du modèle de gestion de configurations discuté dans la Section 4.3.4.

4.3.2 Les éléments de calcul

L'élément de calcul de la LR^2A est un processeur *load-store* similaire au MIPs : les instructions logiques et arithmétiques sont exécutées soit au travers d'une lecture (*load*), soit d'une écriture (*store*) de mémoire. Le processeur a été adapté, notamment sont jeu d'instructions a été réduit d'une façon générale, mais aussi il a été spécialisé pour opérer certains instructions cryptographiques.

Comme toute architecture du type *load-store*, le processeur doit avoir un ensemble de registres de propos général pour la manipulation de données relativement grand, afin de diminuer les accès à la mémoire, car cela représente une pénalité sévère de temps en comparaison aux opérations internes au processeur.

Les instructions ont toutes la même taille, occupant un mot de mémoire chacune. L'instruction contient le code de l'opération et la spécification des opérandes, dans les cas où ils existent. Les caractéristiques essentielles des PEs sont :

- La taille des addresses et des données est de 32-bits ;

- La banque de registres contient 16 registres à propos général, chacun de 32-bits
- Il existent quatre drapeaux appelés : *negative, zero, carry et overflow.*

Ces signaux, la banque de registres et la mémoire locale peuvent être vus dans la Figure 4.5.

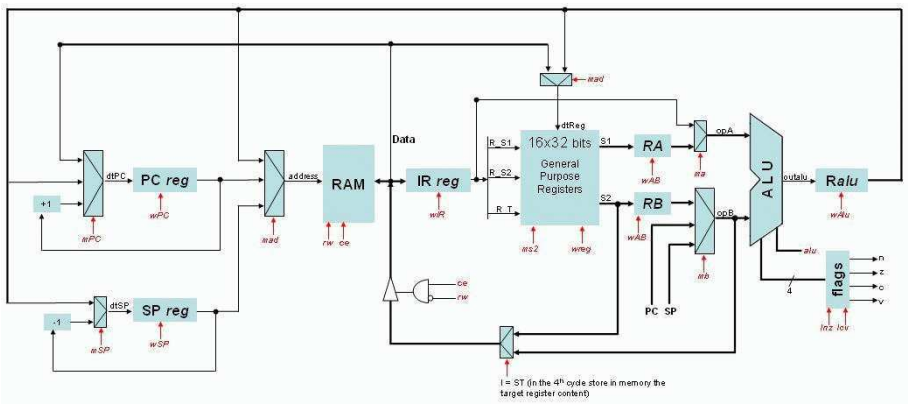


FIG. 4.5: Chemin de données du PE de la LR^2A

En effet, la mémoire a RAM n'est pas intégré comme représenté dans la Figure 4.5. La mémoire est externe au processeur, comme dans la Figure 4.6. Cela donne la possibilité de placer un multiplexeur à l'entrée de la RAM, ce qui est important pour pouvoir injecter des données et amener des configurations à partir du contrôleur. Les mémoires RAMs peuvent ainsi stocker les configurations provenant du contrôleur, les données provenant du processeur hôte, mais aussi les données des résultats des autres éléments de calcul (PEs).

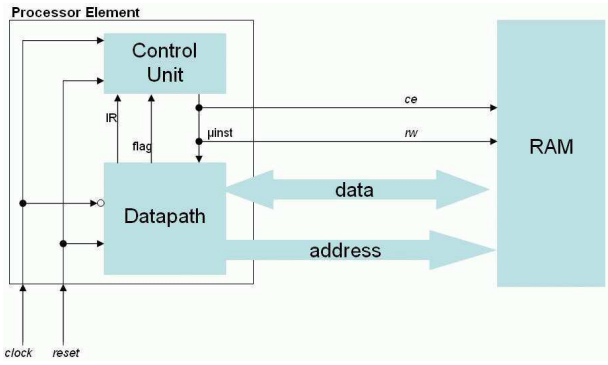


FIG. 4.6: Rapport du PE avec sa mémoire locale

Le chemin de données du processeur inclut des opérateurs spécifiques pour la cryptographie. Il faut souligner que l'objectif de l'architecture LR^2A est celui d'implanter les algorithmes de cryptographie basées sur l'arithmétique modulaire, plus spécifiquement ceux qui peuvent faire usage de la *Leak Resistant Arithmetic* (Chapitre 2, Section 2.8). Les opérations suivantes, nécessaires à la LRA sont implantées de façon câblée, incorporées dans le chemin de données de l'unité logique et arithmétique (ULA) :

- Réduction modulaire (X *mod* M)

- Réduction modulaire par un facteur 2^k ($X \ mod \ 2^k$)
- Addition modulaire ($X + Y \ mod \ M$)
- Soustraction modulaire ($X - Y \ mod \ M$)
- Multiplication modulaire ($X \times Y \ mod \ M$)
- Multiplication modulaire par un facteur 2^k ($X \ mod \ 2^k$)
- Décalages et rotations, réalisées à travers d'un *Barrel Shifter*.

Le Tableau 4.2 montre les instructions réalisées par chaque PE de la LR^2A.

TAB. 4.2: Instructions de la LR^2A

Instructions	**Instructions**
ADD	JMPNR
SUD	JMPZR
AND	JMPCR
OR	JMPVR
XOR	JMP
ADDI	JMPN
SUBI	JMPZ
LDL	JMPC
LDH	JMPV
LD	JSRR
ST	JSR
SL0	JMPD
SL1	JMPND
SR0	JMPZD
SR1	JMPCD
NOT	JMPVD
NOP	JSRD
HALT	MOD2
LDSP	RL
RTS	RR
POP	mulModN
PUSH	mulModB
JMPR	addMod

Il faut rappeler que l'un des objectifs de cette thèse c'est la proposition d'une architecture reconfigurable résistante aux attaques DPA. Par conséquent il ne faut pas oublier que cette architecture est strictement liée à la *Leak Resistant Arithmetic* (LRA) en que concerne la robustesse face aux attaques. Donc c'est la LRA qui détermine le nombre de PEs nécessaires à cette application. Dans la Section 4.4, qui traite des validations, les calculs menant au nombre requis de PEs sont montrés. Le nombre de PEs est aussi la fonction du niveau de parallélisme désiré, et bien entendu

du compromis entre le temps, la surface et la consommation.

4.3.3 Réseau d'Interconnexion

Combiner flexibilité et efficacité énergétique est un problème à addresser dans la conception d'un système sur puce. Dans ce domaine, les réseaux d'interconnexion jouent un rôle primordial. Parmi tous les choix possibles, à partir des réseaux globaux, en passant par les mailles et allant jusqu'aux réseaux hiérarchiques, la décision d'un réseau d'interconnexion pour la LR^2A est passé par deux étapes.

La première était de valider le principe de la LRA par rapport à la robustesse aux attaques DPA et SPA. Pour cela, il était plus important de mettre quelques PEs connectés afin de mesurer la sécurité du système. Donc, dans un premier temps, le choix d'interconnexion est finalement basé sur un bus multiple semblable à un réseau du type *crossbar*, comme montré dans la Figure 4.5.

La deuxième étape est la définition d'un réseau plus évolutif[4] pour l'architecture. Pourtant, dû à des contraintes de temps, cette étape a été avortée. Cependant des pistes sérieuses ont été soulevées.

Il se trouve dans la littérature des états de l'art qui indiquent que les réseaux d'interconnexion hiérarchiques sont des options intéressantes du point de vue de la flexibilité et de la consommation [102]. Une alternative intéressante est le réseau *fat-tree*, notamment utilisé dans les projets des architectures reconfigurables à l'université de Berkeley [31]. Ce réseau a un placement irrégulier qui limite les problèmes de routage en hiérarchisant les connexions. Selon les besoins d'une application donnée, les PEs sont configurés de manière à privilégier le principe de la localité, en plaçant dans des clusters liés directement, si nécessaire.

De tout façon, le schéma de principe reste inchangé. Un PE doit être capable d'envoyer ses résultats à n'importe quel autre élément de calcul de l'architecture. Le contrôleur a un bus à part pour envoyer les configurations et un autre pour injecter et récupérer les données.

4.3.4 Le modèle de configuration

Souvent l'efficacité d'une architecture reconfigurable est préjudiciée du à l'absence des méthodes pour contrôler le schéma de reconfiguration et l'injection des données. Habituellement un contrôleur de configurations commande quelle application va être insérée dans l'architecture reconfigurable, et à quel moment. Il exécute des tâches de manière similaire à un *loader* d'un système d'exploitation, étant responsable de charger les configurations afin d'exécuter des applications, selon un agencement de tâches pre-défini.

La plupart des modèles de configuration trouvés dans la littérature concernent les FPGAs. Un bref état de l'art dans ce domaine peut être lu dans [18]. Il manque donc un modèle taillé pour les architectures à gros grain.

[4] En anglais "*scalable*".

Pour cela, un modèle est proposé associé à la LR^2A. Ce modèle est composé de quelques modules :

- Mémoire de Configuration (CM) ;
- Moniteur de Reconfiguration (RM) ;
- Expéditeur de Reconfiguration (RD) ;
- Scheduler de Configuration (SC) ;
- Contrôle Central de Configuration (CCC).

Ce modèle de reconfiguration est basé sur le *Reconfigurable System Configuration Manager* (RSCM) [18]. Le RCSM peut être représenté comme sur la Figure 4.7.

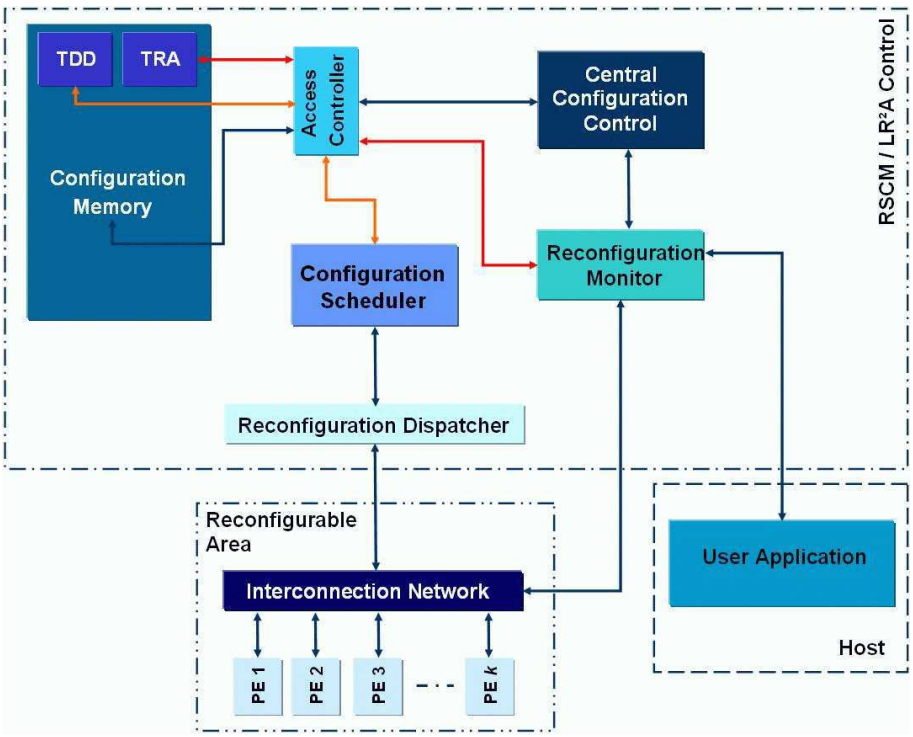


FIG. 4.7: Le Contrôle de la LR^2A basé sur le modèle RSCM

Comme son nom l'indique, la mémoire de configuration (CM) est une partie de la mémoire du contrôleur destinée à garder l'ensemble de configurations pouvant être utilisées par la LR^2A dans le cadre d'une application donnée.

Le Moniteur de Reconfiguration (RM) détecte les situations où des nouvelles configurations sont nécessaires d'être implantés (dites événements de reconfiguration), et notifie au Contrôle Central de Configuration (CCC), pour que le CCC puisse agir selon ce besoin.

Toutes les informations à propos de l'allocation des PEs et ses tâches correspondantes sont stockées dans une structure de données appelée Table d'Allocation de Ressources (TRA), associée au CCC. Le CC reçoit des requêtes du Moniteur de Reconfiguration (RM) et envoie les services nécessaires au Scheduler de Configurations (CS) et à l'Expéditeur de Reconfiguration (RD).

Le module CS est responsable de déterminer laquelle des configurations est la prochaine à être chargée dans le dispositif. Ce module reçoit des requêtes du CCC. Il garde une structure de données avec des informations à propos des dépendances de configuration, appelée Table de Dépendances et de Descriptions (TDD).

Finalement, le module Expéditeur de Reconfiguration gère la reconfiguration elle même, en envoyant le code approprié à chaque PE, où vers un groupe de PEs[5].

Comme mentionné en [18], le RSCM peut être implanté en logiciel, en matériel où dans une solution hybride. Dans le cas de la LR^2A une la solution logicielle a été adoptée, car l'idée est de ne pas avoir à reconfigurer souvent l'architecture, afin d'augmenter sa densité fonctionnelle. Donc le RSCM tourne en permanence dans le contrôleur (le processeur Plasma).

Ce modèle de configuration centralisé permet la parfaite synchronization des calculs, évitant des problèmes tel que la cohérence de la cache. Le contrôleur "connaît" à chaque instant ce qui arrive sur chaque PE, quand ils finissent un calcul, quand des nouvelles données doivent être envoyées, etc. Chaque modification dans l'état de l'architecture est inscrite dans les structures de données du RSCM, et ces structures sont constamment mises à jour.

4.3.5 La gestion de la mémoire

Le RSCM a été développé en langage C et compilé pour le processeur Plasma. Toutes les structures de données sont donc décrites en langage de haut niveau, ainsi que les modules logiciels chargées d'envoyer les données vers les PEs. Dans la LR^2A tous les temps d'exécution d'une tâche sont connus au moment de la compilation, de même que la suite des configurations requises.

De son côté, les PEs sont configurés (par le contrôleur de configurations[6] à travers d'un microcode écrit en langage bas niveau (assembleur). Le microcode est chargé dans une zone protégée de sa mémoire locale, et n'est pas accessible par les autres PEs. Un PE peut bien entendu lire la zone destinée à sa programmation, mais il lui est interdit d'écrire dans cet endroit. En travaillant avec le concept de zones protégées, la mémoire peut opérer selon le modèle de mémoire Harvard[7] ([46]), afin d'éviter des conflits de mémoire.

Donc, la mémoire a une zone destiné à la programmation dont l'écriture n'est permise qu'au contrôleur, une zone de cache local, dont le PE peut stocker les résultats intermédiaires[8] de ses calculs, une zone destinée à l'échange de données avec d'autres PEs. Dans ce dernier cas, la mémoire peut aussi être vue comme une mémoire partagée, mais avec un schéma de sécurité pour éviter tout conflit.

[5] Un broadcast de configurations est possible.
[6] Le processeur Plasma qui exécute le RSCM joue le rôle de contrôleur de configurations.
[7] Selon le modèle Harvard, le mémoire de données est différent du mémoire programme.
[8] ...et finaux, dont l'occurrence est signalisée par un drapeau.

4.4 Validations

L'architecture LR^2A est conçue pour implanter les algorithmes de cryptographie basés sur les opérations modulaires, notamment le RSA. Il est connu que pour certaines applications, cet algorithme requiert une taille de clé au minimum de 1024-bits, allant jusqu'à 4096-bits. Donc la LR^2A a été synthétisé dans plusieurs scénarios de tailles de chemin de données pour différents tailles de clés.

La taille de la clé et la largeur du chemin de données déterminent la quantité de PEs nécessaires aux calculs cryptographiques. Par exemple, pour réaliser une exponentiation modulaire de 1024-bits, ayant un chemin de données de 32-bits, la LR^2A aura besoin de 32 PEs. Si le chemin de données est de 16-bits, pour la même taille de clé, 64 PEs sont nécessaires . Il faut souligner qu'il existe aussi une limitation de performance en augmentant la largeur du chemin de données, car cela entraîne une baisse de fréquence de fonctionnement. Le Tableau montre des résultats après synthèse (toujours dans la technologie $CMOS$ 0.35μ).

Tab. 4.3: Résultats après Synthèse d'un PE pour différents tailles de chemin de données

Chemin de Données	**Portes Logiques (Milliers)**	**Fréquence (MHz)**
16	4,3	60
32	8,5	40
64	15,8	30
128	32	20

Prenant comme exemple la configuration avec un chemin de données ($t = 32$) de 32-bits, la LR^2A nécessite de 32 PEs pour réaliser une exponentiation modulaire de 1024-bits (car $32 \times 32 = 1024$) ; et considérant que le Plasma a environ 40k portes pour ce chemin de données(32-bits), l'architecture occupe environ $325k$ portes(Plasma $(40k)$ + PEs $(32 \times 8,5k = 272Kgates) = 312kgates$). En comparaison à d'autres implantations du RSA dans l'état de l'art, la LR^2A coûte de cinq à huit fois plus de surface en silicium [57], [75]. Il faut souligner que pour garder la performance, le nombre de PEs doit accroître selon la taille des clés à traiter.

Cependant, en termes de performance, comme montre la Figure 4.8, pour des clés cryptographiques plus grandes que 1024-bits, la LR^2A dépasse les implantations classiques du RSA. La comparaison a été faite prenant en considération l'algorithme d'exponentiation modulaire avec la méthode binaire (décrit dans le Chapitre 2). L'implantation de référence est une implantation de l'exponentiation modulaire faisant usage de l'algorithme de Montgomery pour la multiplication modulaire, avec un chemin de données de 32-bits, et fonctionnant à 40MHz[9] [5]. D'un autre côté, la LR^2A implantée possède 32 PEs de 32-bits, et tourne aussi à 40MHz.

Encore dans la Figure 4.8, en ce qui concerne la performance, la LR^2A est équivalente à une

[9]Implantation réalisé dans le cadre de cette thèse, synthétisée sur un FPGA de la famille Spartan III.

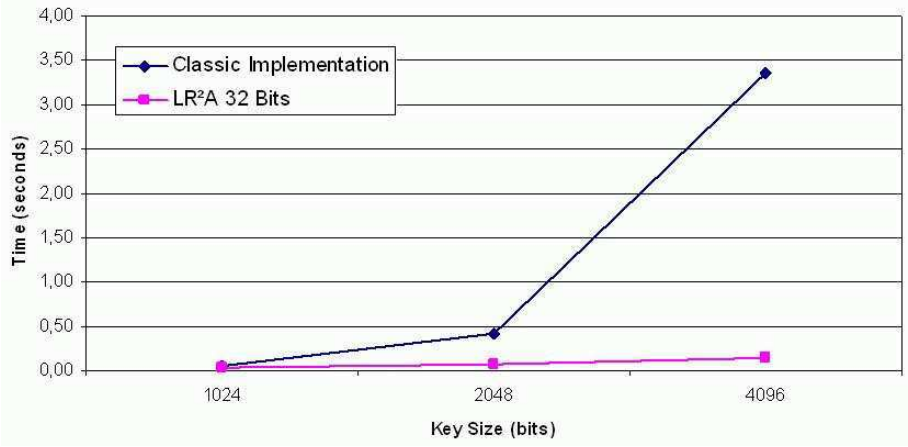


FIG. 4.8: LR^2A versus une implantation classique de l'exponentiation modulaire

implantation classique de l'exponentiation modulaire pour des clés cryptographiques jusqu'à 2048-bits. Au-delà de 4096 bits la LR^2A est nettement plus intéressant en performance, malgré une pénalité par rapport à la surface.

Par rapport à l'état de l'art, il est parfois difficile de comparer exactement les implantations étant donnée les différents manières qui les auteurs choisissent pour présenter ses résultats. Pourtant en restreignant les points de comparaison au temps de calcul pour une exponentiation modulaire, il a été possible de générer le graphique montré dans la Figure 4.9. Dans cette image il est clair que pour une exponentiation de 1024 bits, la LRA est efficace du point de vue de la performance, égalant le temps de calcul le plus rapide de l'état de l'art.

En revenant sur la comparaison entre la LR^2A et une implantation classique, la Figure 4.10 montre un rapport $Surface \times Temps$, où le coût de l'approche LR^2A est démontré. Pour des clés plus petites que 2048-bits, l'implantation classique a un rapport entre performance et coût de silicium plus intéressant. Cependant ce résultat est inversé pour des tailles plus grandes que 2048-bits.

Selon les expériences réalisées, les données de synthèse montrent que le rapport de surface est de cinq fois pour clés de 2048-bits et de dix fois pour clés de 4096-bits. Ceci est le coût de la sécurité. L'architecture de référence ne présente pas des contremesures afin de contrecarrer la DPA. Comme montré sur la Figure 4.11, la LR^2A entraîne un changement dans la consommation quand il arrive un changement de base dans la LRA, pour un même ensemble de données. Pour cela il se peut que dans une exponentiation modulaire $M^e \ mod \ N$, les nombres M, e et N ne changent pas. Ce qui change c'est la base dont ils sont représentés. Les traces de la Figure 4.11 illustrent un changement de base avant l'exponentiation, mais comme discuté dans la Section 3.2.3, il est possible aussi de réaliser ce changement pendant l'exponentiation, augmentant encore le niveau de sécurité.

Afin de prouver le concept de la LRA, des mesures de consommation (à l'aide d'un oscilloscope et d'une sonde en courant) ont été réalisées sur l'implantation FPGA décrite dans la Section 4.1. On peut voir la signature de consommation, au cours d'une exponentiation modulaire ce que

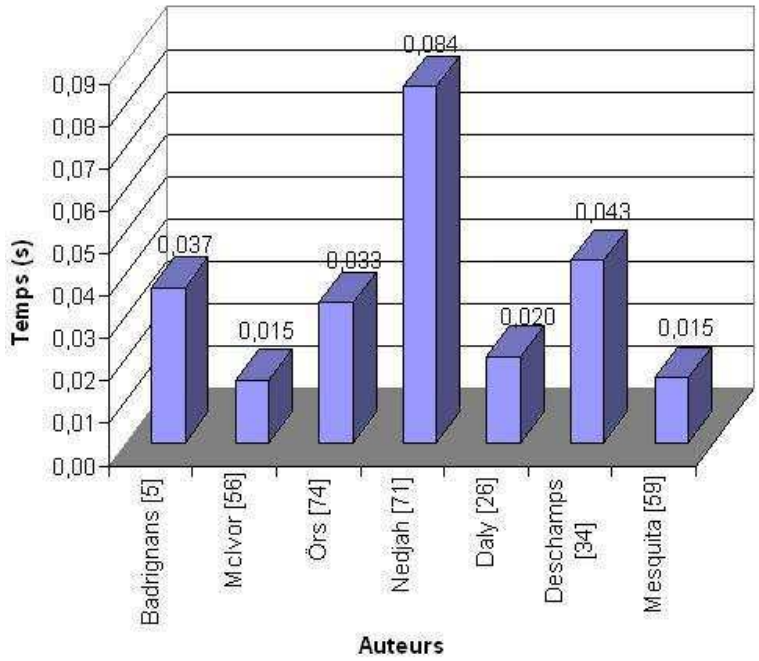


FIG. 4.9: Comparaison entre la LR^2A [60] et d'autres implantations de l'exponentiation modulaire (1024 bits) de l'état de l'art.

permet un attaque SPA. Cela a été laissé sous cette forme afin de mettre en évidence les différentes consommations dans des points spécifiques (indiqués dans la Figure 4.11). Mais en appliquant une contremesure basique contre la SPA il n'est plus possible tirer aucune information des traces de consommation, comme montre la Figure 4.12.

La contremesure en question s'agit tout simplement de ne pas inhiber le multiplieur, mais seulement écrire où non le résultat de la multiplication suivant la valeur du bit de l'exposant. Le résultat c'est une consommation quasiment constante, où il n'est plus possible vérifier si ce sont des zéros ou des uns dans l'exposant. Retrouver la clé, que ce soit par SPA ou par DPA devient nettement plus difficile.

Pourtant, c'est dans la Figure 4.13 que l'efficacité de la LR^2A est plus évidente. Dans la Figure 4.13-(a) a été calculée la différence entre les courbes de consommation de deux exponentiations modulaires réalisées avec une modification de bases RNS. Cela signifie que la consommation quand les calculs sont réalisées dans la base β_1 est différente de la consommation réalisée dans la base β_2. Cependant, quand le calcul est effectué deux fois de suite avec la même base (situation classique), la Figure 4.13-(b) montre que les courbes de consommation sont très proches, indiquant que si le changement de base n'est pas possible, l'attaque DPA devient faisable.

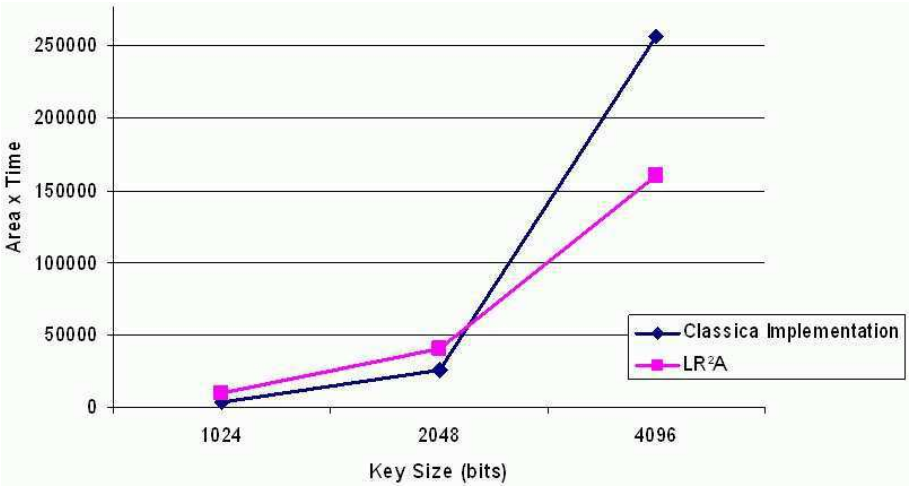


FIG. 4.10: Rapport Performance × Surface : LR^2A versus une implantation classique de l'exponentiation modulaire

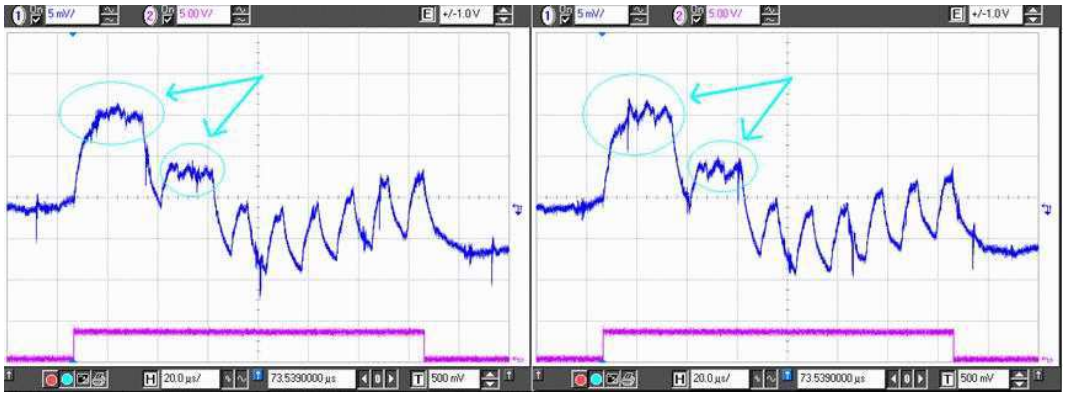


FIG. 4.11: Différentes consommations pour le même ensemble de données calculé.

4.5 L'intérêt de la reconfiguration

L'Algorithme 10 (la multiplication modulaire de Montgomery selon la LRA) montre, en effet, deux catégories de calculs. Les lignes 1, 2 et 4 de cet algorithme traitent des opérations de multiplication et d'addition en RNS. Par conséquent, au moment de ces opérations, la LR^2A est configurée de façon à permettre l'exécution des opérations RNS en parallèle. Pourtant des changements de base sont nécessaires, et donc le moment est venu pour reconfigurer l'architecture. L'équation 4.1 montre que les opérations nécessaires au MRS sont différents des opérations RNS, justifiant ainsi l'utilisation de la reconfiguration.

Pourtant, même si la Section 4.1 indique la possibilité d'utilisation de la reconfiguration dynamique au cours d'une multiplication modulaire, le vrai intérêt de cette technique pour la robustesse des systèmes cryptographiques basées sur l'arithmétique modulaire réside ailleurs.

La Section 3.2.3 explicite que la LRA tant que contremesure contre la DPA a deux démarches.

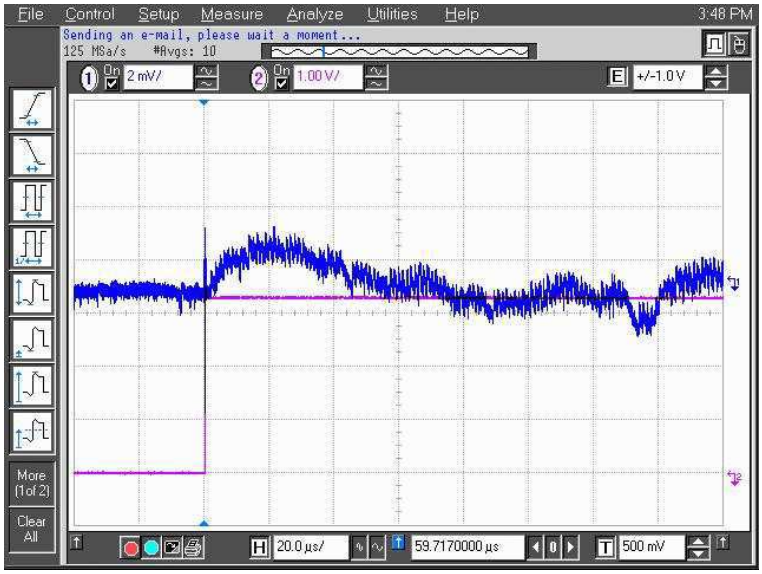


FIG. 4.12: Consommation d'une exponentiation modulaire LRA en presence d'une contremesure simple.

La première consiste dans le choix aléatoire des bases initiales. La deuxième conduit à un changement de bases pendant le calcul d'une exponentiation modulaire. C'est dans la deuxième option que réside l'intérêt pour la reconfiguration dynamique.

Selon l'article à l'origine de la LRA [7], la sécurité de la LRA est basée surtout dans la conversion à la volée entre deux représentations différentes de Montgomery. Considérant que les bases initiales ont été choisies et l'algorithme d'exponentiation modulaire exécuté jusqu'à, par exemple, $R = |A^B \cdot M_1|_N$, dans les deux bases (β_1, β_2). Envisageant de continuer l'exponentiation avec deux nouvelles bases aléatoires (β_3, β_4), il est nécessaire de changer la première représentation de Montgomery (en accord avec M_1), vers la nouvelle représentation (en accord avec M_2). Cela signifie calculer $R = |A^B \cdot M_2|_N$ à partir de $R = |A^B \cdot M_1|_N$.

La solution proposée par [7] consiste à réaliser l'inversion des bases dans l'appel à la fonction de multiplication modulaire (MM). Il faut remarquer l'ordre des bases (β_1 et de β_2) :

$$MM(|A^B \cdot M_1|_N, |M|_N, N, \beta_4, \beta_3)$$

Comme résultat il est obtenu :

$$R = |A^B \cdot M_1 \cdot M_2|_N$$

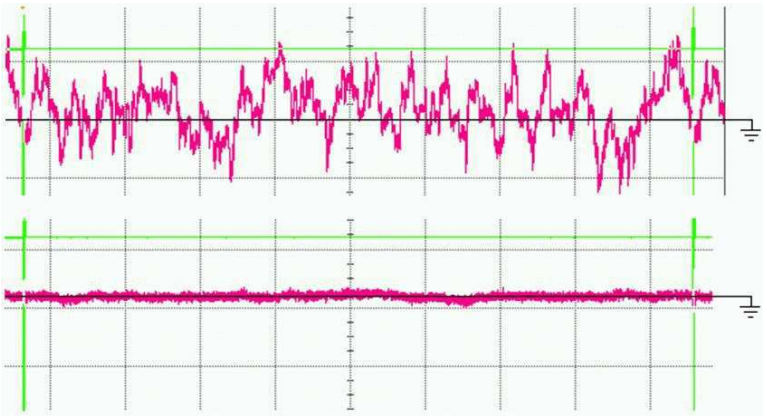
.

Ensuit il est fait un nouveau appel à la fonction MM :

$$MM(|A^B \cdot M_1 \cdot M_2|_N, 1, N, \beta_1, \beta_2)$$

Et donc le résultat espéré est obtenu :

$$R = |A^B \cdot M_2|_N$$



(a) Différence de consommation lors de l'utilisation de deux bases différentes
(b) Différence de consommation lorsque seulement une base est utilisée

FIG. 4.13: Différences de consommation pour deux exponentiations modulaires, entre une implantation qu'utilize la LRA et une implantation classique.

.

Comme résultat, la valeur $|A^B|_N$ est toujours masquée par une quantité aléatoire. Dans son article [7] les auteurs remarquent que la sécurité pourrait être accrue avec un changement de base fréquente. L'architecture LR^2A réponde à ce besoin permettant les changements de base au travers de la reconfiguration dynamique.

Le schéma de reconfiguration de la LR^2A est géré par le RSCM présente au contrôleur de configurations, et suit l'exemple du Systolic Ring [88] en ce qui concerne la possibilité de modification de la configuration de chaque PE dans un cycle d'instruction. Cela permet une changement de base à chaque multiplication modulaire, si c'est nécessaire.

Il faut rappeler que le but est de rendre les calculs intermédiaires les plus aléatoires possibles, afin de masquer la consommation en conséquence des données initiales de l'exponentiation modulaire ($|A^B|_N$). La LR^2A, à cause de sa capacité de reconfiguration dynamique, permet une implantation efficace de la LRA, contribuant ainsi à augmenter la robustesse des systèmes cryptographiques qui utilisent la multiplication modulaire.

4.6 Conclusion du Chapitre

Nous avons proposé dans ce Chapitre une implantation matérielle innovante de la *LRA* [10]. L'architecture proposée n'a pas pu être implanté dans sa totalité, mais la version réduite synthétisée a permis la réalisation des mesures réelles afin de démontrer la robustesse de la LRA.

La LR^2A démontre que l'alliance entre la cryptographie et les architectures reconfigurables est possible et envisageable, non seulement des points de vue de performance et de flexibilité, mais

[10] En effet, l'architecture LR^2A est la toute première implantation matérielle de la *Leak Resistant Arithmetic*

aussi en ce qui concerne la robustesse, qui est l'un des enjeux les plus importants des système sécurisés.

Il restent encore des tâches à accomplir, comme l'intégration des PEs à un NoC, le développement des outils de programmation, et aussi la réalisation d'attaques réelles contre l'architecture. Un aperçu des travaux futurs est donné dans le Chapitre suivant.

Considérations Finales

"Say not, 'I have found the truth,'
but rather, 'I have found a truth.'".
Kahlil Gibran

Les considérations finales de ce travail s'articulent sur trois points. Tout d'abord il est établi une synthèse des apports scientifiques des travaux réalisés dans le cadre de cette thèse. Ensuite les perspectives à moyen terme de ces travaux de recherche sont présentées. Enfin, la bibliographie des communications scientifiques réalisées pendant la durée de cette thèse est présentée.

Conclusions

Dans ce mémoire il a été souligné l'intérêt de la sécurisation des circuits cryptographiques face aux attaques par canaux cachés, notamment les attaques par analyse de consommation, dites SPA et DPA.

Toutefois, le point de départ a été l'étude de viabilité d'utilisation d'une architecture reconfigurable à gros grain pour la cryptographie [61]. Dans cette analyse il a été soulignée l'importance de l'algorithme de Montgomery pour la multiplication modulaire, et son implantation sur différentes cibles technologiques, dont le Systolic Ring. Ce travail a permis la vérification de la complexité liée aux implantations cryptographiques sur une architecture à gros grain, ainsi que des besoins en terme de ressources de mémoire et de contrôle.

Après ce premier travail des questions au sujet des métriques pour évaluer l'efficacité d'une architecture pour la cryptographie se posèrent. Au delà des métriques classiques, tels que le débit, la surface et la consommation, la robustesse a été mise en évidence.

La robustesse est la capacité d'un circuit à résister à certains types d'attaques, notamment les attaques par canaux cachés. Afin de mieux comprendre le fonctionnement de ces attaques, il a été mis en œuvre une plate-forme d'attaque SPA/DPA[11]. Cette plate-forme a contribué non seulement à la compréhension de ces attaques, mais aussi à la validation postérieure de la robustesse de la LR^2A.

Comme conséquence de l'étude des attaques, une nouvelle méthode pour contrecarrer les attaques DPA a été proposée. Cette contremesure originale est basée sur un circuit analogique qui "lisse" le courant, ne permettant pas à un attaquant d'analyser les courbes de consommation, car celles-la sont constantes. Le CMG [64] s'avère efficace et simple à implanter.

Revenant aux aspects architecturaux, les études réalisées sur le Systolic Ring [61] concernant la multiplication modulaire ont convergé vers la proposition d'une architecture reconfigurable pour implanter la LRA. La LRA est une arithmétique créée à partir du RNS et de la multiplication modulaire de Montgomery, et peut être utilisée afin de contrecarrer les attaques DPA.

L'implantation d'une architecture pour exécuter la LRA est passée par deux étapes. Une architecture reconfigurable à gros grain a été proposée. Cette architecture dérive de l'analyse de l'état de l'art dans le domaine des architectures, mais aussi des besoins en terme de performance et de robustesse que requièrent les systèmes cryptographiques. L'adéquation algorithme-architecture a pris en compte les caractéristiques du système de représentation RNS de la LRA, ce qui a guidé vers une architecture parallèle et reconfigurable appelée LR^2A [59]. La LR^2A est composé d'un contrôleur, de plusieurs éléments de calcul (PEs) et d'un réseau d'interconnexion.

La LR^2A est gérée par un modèle de contrôle de configurations adapté à celui publié en [18]. Ce système permet la gestion des configurations, ainsi que l'injection des données du processeur hôte vers la LR^2A. Son schéma de gestion est centralisé, ce qui permet au contrôleur de savoir quelles sont les données attendues dans un endroit précis, dans un temps donné.

Afin de valider l'approche LR^2A, un prototype matériel a été réalisé. Ce prototype garde le principe des PEs de la LR^2A, mais ne réalise que les opérations nécessaires à l'exécution de la LRA. Ce prototype a été implanté sur un FPGA et des attaques par analyse de consommation contre lui ont été menés.

Au travers d'une extrapolation des résultats obtenues au travers du prototype, la LR^2A a été validée en ce qui concerne la robustesse.

En résume, cette thèse a contribué par l'état de l'art sur cinq points précis :

1. Étude de la viabilité d'une architecture reconfigurable pour des applications cryptographiques ;
2. Mis en œuvre d'une plate-forme d'attaques DPA/SPA ;
3. Développement d'une contremesure analogique originale afin de contrecarrer les attaques DPA/SPA ;
4. Proposition d'une architecture reconfigurable pour la cryptographie résistante aux attaques

[11] Au fait cette plate-forme a été développée dans le cadre d'un master deuxième année, et suivie par l'auteur de cette thèse.

DPA/SPA ;

5. Adaptation d'un modèle de contrôle de reconfiguration pour les architectures reconfigurables.

Travaux Futurs

Tout d'abord, quelques travaux à cour terme sont envisagés. Des études se font nécessaires afin de raffiner nos méthodes d'attaque. Il est possible de combiner la DPA avec des attaques par fautes, afin de réduire le nombre d'acquisitions nécessaires aux attaques. Une connaissance plus précise au sujet de différents attaques aidera à améliorer les différents contremesures proposées.

Cependant, le CMG reste une idée à explorer. Un étude automatique s'avère une perspective intéressante.

Autre sujet d'intérêt c'est une réorganisation de l'approche afin de produire des solutions reconfigurables sécurisées à partir d'un étude approfondi des exigences des algorithmes cryptographiques peut aboutir à une architectures plus générique toute en restant performante et robuste.

D'autres améliorations sont possibles dans la LR^2A, sur tous les éléments qui la composent :

- **Brique de base** : Sont état actuel est celui d'une unité reconfigurable à gros grain, mais les besoins guident vers une unité reconfigurable à grain mixte, incorporant du petit grain dans le chemin de données. Cela permettrait l'implantation d'algorithmes nécessitant des opérations niveau bit, tels que le AES. De plus, la flexibilité d'une telle unité élargirait le domaine d'applications de la LR^2A.
- **Réseau d'Interconnexion** : Le prototype implanté comporte un réseau du type crossbar simplifié, utile pour valider le principe de la LR^2A. Pourtant il est envisageable d'utiliser un réseau plus élaboré pour connecter plusieurs dizaines de PEs. Ce réseau peut être un NoC hiérarchisé, s'inspirant des réseaux et sous-réseaux informatiques, afin d'améliorer le principe de la localité et mieux gérer la consommation. Un étude des topologies existantes, en commençant par la *fat-tree*, est envisageable.
- **Le côté logiciel** : En cherchant encore des solutions à partir de l'univers informatique, il faudrait réaliser un étude à propos des modèles de programmation distribués. Un nouveau paradigme architectural impose la réflexion sur des nouvelles formes de programmation.

Les travaux futurs mènent vers des voies de recherche riches, dans des domaines pluridisciplinaires, comme l'informatique, la microélectronique et les mathématiques.

Communications

Voici la liste des communications réalisées pendant la période du doctorat (2002 - 2006) :

Revue

1. MORAES, Fernando Gehm ; CALAZANS, Ney Laert Vilar ; MARCON, C. A. M. ; **MESQUITA, Daniel G.** ; PALMA, José Carlos Sant'anna ; BLAUTH, V. H. . Design and Prototyping of an E1 Drop_Insert Soft Core. IEE Proceedings on Communications, Londres, Anglaterre, v. 150, n. 4, p. 239-243, 2003.

Chapitre de livre

1. **MESQUITA, Daniel G.** ; TECHER, Jean Denis ; TORRES, Lionel ; ROBERT, Michel ; CATHEBRAS, G. ; SASSATELLI, Gilles ; MORAES, Fernando Gehm. Current Mask Generation : an Analogical Circuit to Thwart DPA Attacks. IFIP International Federation for Information Processing. Berlin : Springer, 2006. (*in press*)

Conférence Internationales (avec comité de lecture)

1. **MESQUITA, Daniel G.** ; TORRES, Lionel ; BADRIGNANS, B. ; MORAES, Fernando Gehm ; SASSATELLI, Gilles ; ROBERT, Michel. A Leak Resistant SoC to Counteract Side Channel Attacks. The International Symposium on system-on-Chip. Finlande, 2006.
2. **MESQUITA, Daniel G.** ; TORRES, Lionel ; BADRIGNANS, B. ; MORAES, Fernando Gehm ; SASSATELLI, Gilles ; ROBERT, Michel . A Leak Resistant Architecture Against Side Channel Attacks. In : Field Programmable Logic, 2006, Madrid. Proceedings of FPL 2006, 2006.
3. BADRIGNANS, B., **MESQUITA, Daniel G.**, BAJARD, Jean Claude, TORRES, Lionel, SASSATELLI, Gilles, ROBERT, Michel A Parallel and Secure Architecture for Asymetric Cryptography In : ReCoSoC, 2006, Montpellier. Proceedings Of ReCoSoC 2006. , 2006.
4. FISCHER, V. ; TORRES, Lionel ; **MESQUITA, Daniel G.**Flexible security and its technology limits. In : Reconfigurable Communication-Centric SoCs, 2006, Montpellier. Proceedings Of ReCoSoC 2006.
5. **MESQUITA, Daniel G.** ; TECHER, Jean Denis ; TORRES, Lionel ; SASSATELLI, Gilles ; CAMBON, Gaston ; ROBERT, Michel ; MORAES, Fernando Gehm . Current Mask Generation : A New Hardware Countermeasure for Masking Signatures of Cryptographic Cores. In : IFIP VLSI SoC - International Conference on Very Large Scale Integration, 2005, Perth, 2005.
6. **MESQUITA, Daniel G.** ; TECHER, Jean Denis ; TORRES, Lionel ; SASSATELLI, Gilles ; CAMBON, Gaston ; ROBERT, Michel ; MORAES, Fernando Gehm . Current Mask Generation : A Transistor Level Security Against DPA Attacks. In : IEEE/ACM 18th SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEMS DESIGN, 2005, Florianópolis, 2005.

7. **MESQUITA, Daniel G.** ; TECHER, Jean Denis ; TORRES, Lionel ; SASSATELLI, Gilles ; CAMBON, Gaston ; ROBERT, Michel ; MORAES, Fernando Gehm . A New Hardware Countermeasure for Masking Power Signatures . In : Reconfigurable Communication-Centric SoCs, 2005, Montpellier, 2005.
8. CARVALHO, E. ; CALAZANS, Ney Laert Vilar ; MORAES, Fernando Gehm ; **MESQUITA, Daniel G.** . Reconfiguration Control for Dynamically Reconfigurable Systems. In : XIX Conference on Design of Circuits and Integrated Systems, 2004, Bordeaux. Proceedings of XIX Conference on Design of Circuits and Integrated Systems, 2004.
9. **MESQUITA, Daniel G.** ; MORAES, Fernando Gehm ; PALMA, Jose Carlos Sant'anna ; MOLLER, Leandro Heleno ; CALAZANS, Ney Laert Vilar . Remote and Partial Reconfiguration of FPGAs : tools and trends. In : IEEE-ACM-IPDPS-RAW - International Parallel and Distributed Processing Symposium - Reconfigurable Architectures Workshop, 2003, Nice. Proceedings of the 10th RAW. Los Alamitos : IEEE Computer Society Press, 2003. v. 1. p. 1-8.
10. **MESQUITA, Daniel G.** ; TORRES, Lionel ; ROBERT, Michel ; SASSATELLI, Gilles ; MORAES, Fernando Gehm . Are coarse grain reconfigurable architectures suitable for cryptography ?. In : IFIP VLSI SoC - International Conference on Very Large Scale Integration, 2003, Darmstadt. Proceedings of the 12th VLSI-SoC, 2003. p. 276-281.
11. MORAES, Fernando Gehm ; **MESQUITA, Daniel G.** ; MOLLER, Leandro Heleno ; PALMA, Jose Carlos Sant'anna ; CALAZANS, Ney Laert Vilar . Development of a Tool-Set for Remote and Partial Reconfiguration on Virtex Devices. In : IEEE/ACM Design, Automation and Test EUROPE, 2003, Munique. Proceedings of the DATE 2003. Los Alamitos : IEEE Press, 2003. p. 1122-1123.

Conférence Nationales (avec comité de lecture)

1. BADRIGNANS, B. ; **MESQUITA, Daniel G.** ; BAJARD, Jean Claude ; TORRES, Lionel ; SASSATELLI, Gilles ; ROBERT, Michel . Implémentation Matérielle d'une Arithmétique Résistante aux Fuites. In : Symposium en Architecture de Machines, 2006, Perpignan. Proceedings of Sympa 2006. Perpignan, 2006.
2. **MESQUITA, Daniel G.** ; TORRES, Lionel ; SASSATELLI, Gilles ; ROBERT, Michel . La reconfiguration dynamique comme technique pour éviter les attaques DPA. In : XIII Journées Doctorants de l'Ecole Doctorale I2S (DOCTISS'05), 2005, Montpellier, 2005.
3. **MESQUITA, Daniel G.** ; TORRES, Lionel ; SASSATELLI, Gilles ; ROBERT, Michel . Adéquation Algorithme Architecture : Des Opérateurs Arithmétiques Pour La Cryptographie Dans Les Architectures Reconfigurables. In : Journées Nationales du Réseau Doctoral de Microélectronique, 2004, Marseille. Proceedings of JNRDM. Marseille, 2004. p. 310-312.
4. **MESQUITA, Daniel G.** ; TORRES, Lionel ; ROBERT, Michel ; SASSATELLI, Gilles ; MORAES, Fernando Gehm . A coarse grain reconfigurable architecture to compute modular product. In : 18th South Symposium on Microelectronics, 2003, Novo Hamburgo, RS, Brasil. Anais do 18ž Simpósio Sul de Microeletrônica, 2003. v. 1. p. 9-16.

Bibliographie

[1] D. Abraham, G. Dolan, G. Double, and J. Stevens. Transaction security system. *IBM Systems Journal*, 30(2) :206–209, 1991.

[2] P. Athanas and H. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer Magazine*, 26(3) :11–18, 1993.

[3] Atmel. At40k product overview. ftp ://www.atmel.com/pub/atmel/at40k.hlp, 2000.

[4] Atmel. At94k05 / FPSLIC. http ://www.atmel.com/dyn/resources/prod_documents/doc1138.pdf, Juin 2005.

[5] B. Badrignans, D. Mesquita, J-C. Bajard, L. Torres, G. Sassatelli, and M. Robert. A parallel and secure architecture for asymetric cryptography. In *Proceedings of ReCoSoC*, page in press, Montpellier, France, 2006.

[6] J-C. Bajard and L. Imbert. A full rns implementation of rsa. *IEEE Transactions on Computers*, 53(6) :769–774, 2004.

[7] J-C. Bajard, L. Imbert, P-Y. Liardet, and Y. Teglia. Leak resistant arithmetic. In *Proceedings of CHES '04*, pages 62–75, Cambridge, USA, 2004. Springer-Verlag.

[8] J-C. Bajard, N. Meloni, and T. Plantard. Efficient rns bases for cryptography. In *Proceedings of IMACS*, page CD, Paris, France, 2005.

[9] P. Barret. Implementing the rivest, shamir and adleman public-key encryption algorithm on a standard digital signal processor. In *Proceedings of Crypto 86*, pages 311–323, Santa Barbara, USA, 1986.

[10] L. Benini, A. Macii, E. Macii, E. Omerbegovic, F. Pro, and M. Poncino. Energy-aware design techniques for differential power analysis protection. In *Proceedings of DAC '03*, pages 36–41, Anaheim, CA, USA, 2003.

[11] P. Benoit. *Architectures des Accélerateurs de Traitement Flexibles pour les Systèmes sur Puce*. PhD thesis, Université Montpellier II, LIRMM, Département de Microélectronique, 2004.

[12] P. Bertin, J. Vuillemin, D. Roncin, M. Shand, H. Shand, and P. Boucard. Programmable active memories : Reconfigurable systems come of age. *IEEE Transactions on VLSI systems*, 4(1) :56–69, 1996.

[13] G. Blakley. A computer algorithm for calculating the product A *modulo* B. *IEEE Transactions Computers*, 32(5) :497–500, 1983.

[14] B. Boer, K. Lemke, and G. Wicke. A dpa attack against the modular reduction within a crt implementation of rsa. In *Revised Papers from the CHES '02*, pages 228–243, San Francisco, USA, 2002. Springer-Verlag.

[15] A. Booth. A signed binary multiplication technique. *Quarterly Journal of Mechanics and Applied Mathematics*, 4(1) :236–240, 1951.

[16] G. Bouesse. *Contribution à la conception de circuits intégrés sécurisés : l'alternative asynchrone.* PhD thesis, Institute National Polytechnique de Grenoble, INPG, 2005.

[17] T. Callahan, J. hauser, and J. Wawrzinek. The Garp architecture and C compiler. *IEEE Computer Magazine*, 33(4) :62–69, 2000.

[18] E. Carvalho, N. Calazans, F. Moraes, and D. Mesquita. Reconfiguration control for dynamically reconfigurable systems. In *Proceedings of DCIS*, 2004.

[19] E. Caspi, M. Chu, R. Huang, J. Yeh, Y. Markovskiy, J. Wawrzynek, and A. DeHon. Stream computations organized for reconfigurable execution (SCORE). In *Proceedings of FPL*, pages 605–614, Villach, Austria, 2000.

[20] D. Chaum. Security without identification : transaction systems to make big brother obsolete. *Communication of the ACM*, 8(10) :103–144, 1985.

[21] M. Ciet, M. Neve, E. Peeters, and J-J. Quisquater. Parallel fpga implementation of rsa with residue number systems - can side-channel threats be avoided ? In *Proceedings of MWSCAS 03*, pages 806–810, Cairo, Egypt, 2003. IEEE Computer Society.

[22] C. Clavier, J-S. Coron, and N. Dabbous. Differential power analysis in the presence of hardware countermeasures. In *Proceedings of CHES '00*, pages 252–263, Worcester, USA, 2000.

[23] K. Compton and S. Hauck. Reconfigurable computing : a survey of systems and software. *ACM Computing Surveys*, 34(2) :171–210, 2002.

[24] J-S. Coron. Resistance against differential power analysis for elliptic curve cryptosystems. In *Proceedings of the CHES '99*, Worcester, USA.

[25] S. Coutinho. *Números Interios e Criptografia RSA*. Editora do IMPA, 2000.

[26] J. Daemen and V. Rijmen. Annexe to aes proposal rijndael. http ://www.iaik.tugraz.ac.at/research/krypto/AES/old/ rijmen/rijndael/PropCorr.PDF, Juin 1998.

[27] A. Daly and W. Marnane. Efficient architectures for implementing montgomery modular multiplication and rsa modular exponentiation on reconfigurable logic. In *Proceedings of the FPGA '02*, pages 40–49, 2002.

[28] R. David. *Architectures des Accélerateurs de Traitement Flexibles pour les Systèmes sur Puce.* PhD thesis, École Nationale Supérieure de Sciences Appliquées et de Technologie, LASTI, 2003.

[29] R. David, D. Chillet, S. Pillement, and O. Sentieys. Dart : A dynamically reconfigurable architecture dealing with future mobile telecommunications constraints. In *IPDPS*, pages CD–ROM/Abstracts Proceedings, Nice, USA, 2002. IEEE Computer Society.

[30] R. David, D. Lavenier, and S. Pillement. Du microprocesseur au circuit fpga. une analyse sous l'angle de la reconfiguration. *Technique et Science Informatiques*, 24(4) :395–422, 2005.

[31] A. DeHon. Comparing custom computing machines. In *volume 3526 of Proceedings of SPIE*, pages 124–133, Boston, USA, 1998.

[32] J. Delgado-Frias, M. Myjak, F. Anderson, and D. Blum. A medium-grain reconfigurable cell array for dsp applications. In *Proceedings of the IASTED CSS*, pages 231–236, Cancun, Mexique, 2003. IASTED.

[33] D. Demigny, N. Boudouani, N. Abel, and L. Kessal. La rémanence des architectures reconfigurables, un critère significatif de classification des architectures. In *Proceedings of JFAAA*, pages 49–52, Monastir, Tunisie, 2002.

[34] D. Demigny, L. Kessal, R. Bourguiba, and N. Boudouani. How to use high speed reconfigurable fpga for real time image processing ? In *Proceedings of CAMP '00*, page 240, 2000.

[35] J-P. Deschamps and G. Sutter. Fpga implementation of modular multipliers. In *Proceedings of the DCIS '02*, pages 107–112, 2002.

[36] G. Estrin. Reconfigurable computer origins : The ucla fixed-plus-variable (f+v) structure computer. *IEEE Annals of the History of Computing*, 24(4) :3–9, 2002.

[37] G. Estrin and C. Viswanathan. Organization of a "fixed-plus-variable" structure computer for computation of eigenvalues and eigenvectors of real symmetric matrices. *Journal of ACM*, 9(4) :522–, 1962.

[38] V. Fischer, L. Torres, and D. Mesquita. Flexible security and its technology limits. In *Proceedings of ReCoSoC*, page in press, Montpellier, France, 2006.

[39] Electronic Frontier Foundation. *Cracking DES - Secrets of Encryption Research, Wiretap Politics and Chip Design.* O'Reilly, 1998.

[40] H. Garner. The residue number system. *IRE Transactions on Electronic Computers*, 8 :140–147, 1959.

[41] Seth Copen Goldstein, Herman Schmit, Srihari Cadambi, Matt Moe, and R Reed Taylor. Piperench : a reconfigurable architecture and compiler. *IEEE Computer Magazine*, 33(4) :70–77, 2000.

[42] J. Golic and C. Tymen. Multiplicative masking and power analysis of aes. In *Revised Papers from the CHES '02*, pages 198–212, San Francisco, USA, 2002. Springer-Verlag.

[43] L. Goubin and J. Patarin. Des and differential power analysis (the "duplication" method). In *Proceedings of the CHES '99*, Worcester, USA. Springer-Verlag.

[44] R. Hartenstein. A decade of reconfigurable computing : a visionary retrospective. In *Prodeedings of DATE*, pages 642–649, 2001.

[45] R. Hartenstein and R. Kress. A datapath synthesis system for the reconfigurable datapath architecture. In *Proceedings of ASP-DAC '95*, page 77, Makuhari, Japan, 1995.

[46] J. Hennessy and D. Patterson. *Computer Architecture : a quantitative approach.* Morgan Kaufmann, 2003.

[47] M. Hideyo and M. Atsuko. Efficient countermeasures against rpa, dpa, and spa. In *Proceedings of CHES '04*, pages 343–356, Cambridge, USA, 2004. Springer-Verlag.

[48] J. Irwin, D. Page, and N. P. Smart. Instruction stream mutation for non-deterministic processors. In *Proceedings of ASAP '02*, page 286, San Jose, USA, 2002. IEEE Computer Society.

[49] K. Kaukonen and R. Thayser. A stream cipher encryption algorithm "arcfour". http ://www.mozilla.org/projects/security/pki/nss/draft-kaukonen-cipher-arcfour-03.txt, Juillet 1999.

[50] S. Kawamura, M. Moike, F. Sano, and A. Shimbo. Cox-rower architecture for fast parallel Montgomery multiplication. In *Proceedings of Eurocrypt*, pages 523–538, 2000.

[51] C. Kim, J. Ha, S. Moon, S-M. Yen, and S-H. Kim. A crt-based rsa countermeasure against physical cryptanalysis. In *Proceedings of HPCC 2005*, pages 549–554, Sorrento, Italie, 2005. Springer-Verlag.

[52] D. Knuth. *The art of computer programming - vol 2 - Seminumerical algorithms.* Addison-Wesley, 1951.

[53] P. Kocher. Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In *Proceedings of CRYPTO '96*, pages 104–113, Santa Barbara, USA, 1996. Springer-Verlag.

[54] P. Kocher, J. Jaffe, and B. Jun. Introduction to differential power analysis and related attacks. Technical Report S/N, Cryptography Research Inc, 1998.

[55] S. Mangard. Hardware countermeasures against dpa - a statistical analysis of their effectiveness. In *Proceedings of CT-RSA '04*, pages 222–235, San Francisco, USA, 2004. Springer-Verlag.

[56] D. May, H. Muller, and N. Smart. Non-deterministic processors. In *Proceedings of ACISP '01*, pages 115–129, Sidney, Australia, 2001. Springer-Verlag.

[57] C. McIvor, M. McLoone, J. McCanny, and W. Marnane. Fast montgomery modular multiplication and rsa cryptographic processor architectures. In *Proceedings of the Asilomar Conference*, Pacific Groove, USA, 2003. IEEE Computer Society.

[58] A. Menezes. *Handbook of Applied Cryptography - fifth printing.* CRC Press, Boca Raton, USA, 2001.

[59] D. Mesquita, B. Badrignans, L. Torres, G. Sassatelli, M. Robert, and F. Moraes. A leak resistant architecture against side channel attacks. In *Proceedings of FPL*, page in press, Madrid, Spain, 2006.

[60] D. Mesquita, B. Badrignans, L. Torres, G. Sassatelli, M. Robert, and F. Moraes. A leak resistant soc against side channel attacks. In *Proceedings of ISSoC*, page in press, Tampere, Finlande, 2006.

[61] D. Mesquita, F. Moraes, J.Palma, L.Möller, and N. Calazans. Remote and partial reconfiguration of fpgas : Tools and trends. In *IPDPS*, page 177, Nice, France, 2003. IEEE Computer Society.

[62] D. Mesquita, J-D. Techer, L. Torres, M. Robert, G. Cathebras, G. Sassatelli, and F. Moraes. Current mask generation : an analog circuit to thwart dpa attacks. *IFIP Series : VLSI-SOC : From Systems to Chips*, 200 :in press, 2006.

[63] D. Mesquita, J-D. Techer, L. Torres, G. Sassatelli, G. Cambon, M. Robert, and F. Moraes. Current mask generation : A new hardware countermeasure for masking signatures of cryptographic cores. In *Proceedings of VLSI-SoC*, page in press, Perth, Australia, 2005.

[64] D. Mesquita, J-D. Techer, L. Torres, G. Sassatelli, G. Cambon, M. Robert, and F. Moraes. Current mask generation : A transistor level security against dpa attacks. In *Proceedings of SBCCI*, page in press, Florianópolis, Brasil, 2005.

[65] T. Messerges, E. Dabbish, and R. Sloan. Investigations of power analysis attacks on smartcards. In *Proceedings of USENIX'99*, pages 151–162, 1999.

[66] T. Messerges, E. Dabbish, and R. Sloan. Power analysis attacks of modular exponentiation in smartcards. In *Proceedings of the CHES '99*, pages 144–157, London, UK, 1999. Springer-Verlag.

[67] M.Hitz and E. Kaltofen. Integer division in residue number systems. *IEEE Transactions on Computers*, 44(8) :983–989, 1995.

[68] P. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44, 1985.

[69] S. Moore, R. Anderson, R. Mullins, G. Taylor, and J. Fournier. Balanced self-checking asynchronous logic for smart card applications. *Journal of Microprocessors and Microsystems*, 27(9) :421–430, 2003.

[70] T. Nagell. *Linear Congruences*. Wiley, 1951.

[71] National. Navigator magazine. http ://www.national.com/appinfo/milaero/files/f61.pdf, 1998.

[72] N. Nedjah and L. Mourelle. A review of modular multiplication methods and respective hardware implementations. *Informatica*, 30 :111–130, 2006.

[73] National Bureau of Standards. Announcing the data encryption standard. Technical Report FIPS Publication 46, National Bureau of Standards, Jan 1977.

[74] National Bureau of Standards. Security requirements for cryptographic modules. Technical Report FIPS Publication 140-2, National Bureau of Standards, Decembre 2002.

[75] S. Örs, L. Batina, B. Preneel, and J. Vandewalle. Hardware implementation of a montgomery modular multiplier in a systolic array. In *IPDPS*, page 184, Nice, France, 2003. IEEE Computer Society.

[76] I. Page. Reconfigurable processor architectures. *Elsevier Microprocessors & Microsystems*, 20(1) :185–196, 1996.

[77] K. Posch and R. Posch. Modulo reduction in residue number systems. *IEEE Transactions on Parallel and Distributed Systems*, 6(5) :449–454, 1995.

[78] B. Radunovic and V. Milutinovic. A survey of reconfigurable computing architectures. In *Proceedings of the FPL*, pages 376–385, Tallin, Estonia, 1998.

[79] A. Razafindraibe, P. Maurine, M. Robert, F. Bouesse, B. Folco, and M. Renaudin. Secured structures for secured asynchronous qdi circuits. In *Proceedings of DCIS*, 2004.

[80] S. Rhoads. Plasma - most mips i(tm) opcodes : Overview. http ://www.opencores.org/projects.cgi/web/mips/overview, Juillet 2006.

[81] R. Rivest, M. Robshaw, R. Sidney, and Y. Yin. The RC6 block cipher. ftp ://ftp.rsasecurity.com/pub/rsalabs/rc6/rc6v11.pdf, August 1998.

[82] R. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2) :120–126, 1978.

[83] P. Rogaway and D. Coppersmith. A software-optimized encryption algorithm. *Journal of Cryptology*, 11(4) :273–287, 1998.

[84] J. Rose, A. El Gamal, and A. Sangiovanni-Vincentelli. Architecture of field programmable gate arrays. In *Proceedings of IEEE*, volume 81, pages 1013–1029, 1993.

[85] E. Sanchez, M. Sipper, J-O. Haenni, J-L. Beuchat, A. Stauffer, and A. Perez-Uribe. Static and dynamic configurable systems. *IEEE Transactions on Computers*, pages 556–564, 1999.

[86] J. Santos. *Introdução à Teoria dos Números*. Editora do IMPA, 2000.

[87] H. Saputra, N. Vijaykrishnan, M. Kandemir, M. J. Irwin, R. Brooks, S. Kim, and W. Zhang. Masking the energy behavior of des encryption. In *Proceedings of the DATE '03*, page 10084, Munich, Germany, 2003. IEEE Computer Society.

[88] G. Sassatelli. *Architectures Reconfigurables Dynamiquement pour les Systèmes sur Puce*. PhD thesis, Université Montpellier II, LIRMM, Département de Microélectronique, 2002.

[89] G. Sassatelli, L. Torres, J. Galy, G. Cambon, and C. Diou. The systolic ring : A dynamically reconfigurable architecture for embedded systems. In *Proceedings of FPL*, pages 409–419, Belfast, Northern Ireland, UK, 2001.

[90] B. Schneier. *Applied Cryptography : Protocols, Algorithms, and Source Code in C - second edition*. John Wiley & Sons, New York, USA, 1996.

[91] A. Shamir. Protecting smart cards from passive power analysis with detached power supplies. In *Proceedings of CHES '00*, pages 71–77, Worcester, USA, 2000. Springer-Verlag.

[92] C. Shannon. Communication theory of secrecy systems. *Bell System Technical Journal*, 28(4) :656–715, 1949.

[93] Sidsa. Fipsoc mixed signal system-on-chip. http ://www.sidsa.com/FIPSOC/fipsoc1.pdf, 1999.

[94] D. Stinson. *Cryptographie : Théorie et pratique - Deuxième edition*. Vuibert, 2003.

[95] E. Trichina, D. De Seta, and L. Germani. Simplified adaptive multiplicative masking for aes. In *Revised Papers from the CHES '02*, pages 187–197, San Francisco, USA, 2002. Springer-Verlag.

[96] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. KIM, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software : Raw Machines. *IEEE Computer Magazine*, 30(9) :86–93, 1997.

[97] C. Walter. Montgomery's multiplication technique : How to make it smaller and faster. In *Proceedings of the CHES '99*, Worcester, USA.

[98] M. Wirthlin and B. Hutchings. Improving functional density using run-time circuit reconfiguration. *IEEE Transactions on VLSI systems*, 6(2) :247–256, 1998.

[99] Xilinx. Xc6200 field programmable gate arrays (datasheet). http ://www.dcs.gla.ac.uk/ nicholas/ca4/dsheet.pdf, April 1997.

[100] Xilinx. Virtex series configuration architecture user guide. http ://www.xilinx.com/xapp/xapp151.pdf, September 2000.

[101] Xilinx. Virtex-4 user guide. http ://www.xilinx.com/bvdocs/userguides/ug070.pdf, Mars 2006.

[102] H. Zhang, M. Wan, V. George, and J. Rabaey. Interconnect architecture exploration for low-energy reconfigurable single-chip dsps. In *Proceedings of WVLSI '99*, pages 2–9, Orlando, USA, 1999. IEEE Computer Society.

[103] Y. Zhou and D. Feng. Side-channel attacks : Ten years after its publication and the impacts on cryptographic module security testing. http ://eprint.iacr.org/2005/388.pdf, 2005.

Table des figures

Liste des tableaux

Annexes

ANNEXE A

Abréviations

Quelques abréviations, sigles et acronymes

ACM	Association for Computing Machinery
AES	Advanced Encryption Standard
ASIC	Application Specific Integrated Circuit
CLB	Configurable Logic Block
DPA	Differential Power Analysis
CISC	Complex Instruction-Set Computer
CMG	Current Mask Generator
CMOS	Complementary Metal Oxid Semiconductor
CPU	Central Processing Unit
CRT	Chinese Remainder Theorem
CTR	Compile-Time Reconfiguration
DCT	Discrete Cosine Transform
DES	Data Encryption Standard
DRAM	Dynamic Random Access Memory
DPA	Differential Power Analysis

DPR	DataPath Reconfigurable
DSP	Digital Signal Processing
ECC	Elliptic Curve Cryptography
EFF	Electrical Frontier Fondation
E/S	Entrée / Sortie
FIPS	Federal Information Processing Standard
FPGA	Field Programmable Gate Arrays
HO-DPA	High-Order Differential Power Analysis
IOB	Input/Output Blocks
LRA	Leak Resistant Arithmetic
LR^2A	Leak Resistant Reconfigurable Architecture
LUT	Look-Up Table
MRED	Modular Reduction on Equidistant Data
MRS	Mixed Radix System
NoC	Network on Chip
PGCD	Plus Grand Commun Diviseur
PE	Processing Element
RNS	Residue Number System
RPA	Refined Power Analysis
RPC	Remote Procedure Call
RISC	Reduced Instruction-Set Computer
RSA	Rivest Shamir and Adlemann
RSCM	Reconfigurable Systems Configuration Model
RTR	Run-Time Reconfiguration
SCA	Side Channel Attacks
SiP	System in Package
SNR	Signal to Noise Ratio
SoC	System on Chip
SPA	Single (or simple) Power Analysis
SPICE	Simulation Program with Integrated Circuit Emphasis
ULA	Unité Logique et Arithmétique
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XOR	eXclusive-OR

ANNEXE B

RSA

Exemple pratique du RSA

Daniel Gomes Mesquita, LIRMM, France

L'algorithme de cryptographie RSA a été développé en 1978 par Ron Rivest, Adi Shamir et Leonard Adleman.

Voici un exemple pour faciliter sa compréhension. Cet exemple peut être copié et éxecuté sur xmaple, version 8.00.

B.1 La description succinte de l'algorithme :

1. Trouver p et q, deux grands (e.g., 1024-bits) nombres premiers.

2. Choisir e tel que e soit plus grand que *1*, que e soit moins grand que $p.q$ et e et *(p-1).(q-1)* doivent être premiers entre eux, ce qui signifie qu'ils ne doivent pas avoir des facteurs premiers en commun ; e peut ne pas être premier, mais il doit être impair.

3. Calculer d de façon que $(d.e)=1$ mod $(p\text{-}1).(q\text{-}1)$. d est l'inverse multiplicative de e.

4. La fonction de cryptage est $C = M\hat{}e \ mod \ p.q$, où C c'est le message crypté (un nombre entier positive) ; M est le message en clair (un nombre entier positive) et '^' indique une exponentiation. Le message M doit être plus peti que $p.q$.

5. La fonction de decryptag est $M = C\hat{}d \ mod \ p.q$.

```
> restart:
```

B.2 Le choix des nombres

La première étape est le choix de deux grands nombres premiers. Pour un souci pédagogique la taille est limitée dans cet exemple à 80 digits décimaux. Cela est réalisé à l'aide des fonctions 'rand' et 'nextprime' disponibles sur xmaple.

```
\mapleinline{active}{1d}{M1 := rand(10^80)();
M2 := rand(10^80)();
P := nextprime(M1);
Q := nextprime(M2);}{%
}
```

$M1 :=$ 735094736764707883422813387791917924959003937512095393006283\
63443011313746086538

$M2 :=$ 626649130748136562206438424438441319057545656720753583911355\
37108795991638155474

$P :=$ 735094736764707883422813387791917924959003937512095393006283\
443011313746086647

$Q :=$ 626649130748136562206438424438441319057545656720753583911355\
3\710879599163815553

Maintenant il faut calculer le produit des deux premiers (p et q), et le produit de 1 moins chacun des premiers.

```
> n := P*Q;
> n2 := (P-1)*(Q-1);
```

$n :=$ 460646477811134459110242789721095449310742498732087650896866\
52\81176024132727926340735112611427061529513565809109424346544\
63617545402274655172704112260280220079\1

$$n2 := 460646477811134459110242789721095449310742498732087650896866 5\backslash$$
$$2811760241327279262045607258601426169665883845860734999448 8\backslash$$
$$9676752169125054787826489315297417958592$$

L'étape suivante est le choix de e. Comme e fait partie de la clé publique, il n'y a pas besoin de le choisir de façon aléatoire. Pourtant une bonne pratique est de prendre un e de façon qu'il aye le format *2^x+w*, où x est un exposant quiconque et w est un nombre entier de petite taille. Cela conduit à un e composé (en binaire) dans la plupart par des zéros.

```
> e:= 2^16+1;  isprime(e); gcd(e,n2);
```

$$e := 65537$$
$$true$$
$$1$$

Il faut noter que e doit être premier avec *n2*. Cela est garanti car il a été choisi un e premier. Ensuite il faut calculer *d,* l'inverse multiplicative de *e mod n2*. Cela peut être calculé à l'aide de l'algorithme d'Euclide, mais ici sont utilisées quelques fonctions d'arithmétique modulaire du xMaple.

```
> d := eval(1/e mod n2);
```

$$d := 3864715494448543128266107626230177180051248233865619023896929 8\backslash$$
$$5367323055547114293348134566266508462839153055648969774279 8\backslash$$
$$9377991251838200169957686270704064732 17$$

Finalement, il est possible de crypter et de decrypter le message.

```
> encoder := (a,e,n) -> Power(a,e) mod n:
> decoder := (a,d,n) -> Power(a,d) mod n:
```

Pour vérifier l'éxactitude, il est possible de refaire l'example pour un message simples -> m=11 :

```
> m1 := encoder(11,e,n);p1 :=decoder(m1,d,n);
```

$$m1 := 2690756479270719794232489449278224023805507511996314046418 88\backslash$$
$$3516970278149147397386672803631701229210683211357716596151 7\backslash$$
$$0783176652131145919166423883971936611602 5$$
$$p1 := 11$$

B.3 Un example avec un message textuel

Suivant l'exemple précedent, pour coder un message il faut réaliser les conversions suivantes 1. Convertir le message du format ASCII vers une séquence de décimaux

2. Convertir chaque décimal vers un nombre hexa de deux digits

3. Concaténer les nombres hexa dans un seul nombre 4. Convertir ce nombre en décimal

Le message peut alors être crypté et decrypté.

Procédure pour la conversion vers le nombre hexa de deux digits.

```
> twodigithex := a -> substring(convert(a+256,hex),2..3):
```

Un message court.

```
> mess1 := "Perde-se o amigo,
> mas jamais a oportunidade.
> (Lema do GAPH)";
```

$$\begin{aligned} &\textit{mess1} := \text{“Perde-se o amigo,\textbackslash} \\ &\text{mas jamais a oportunidade.\textbackslash} \\ &\text{(Lema do GAPH)”} \end{aligned}$$

Conversion du message vers un nombre.

```
> shortconverter := proc(messagestring)
> local stringofhex, lengthofmess, hexstring, i:
> #First we convert the ASCII string to a list of decimal
> equivalents
> #Then we convert thedecimal numbers to hex equivalents
> stringofhex := map(twodigithex, convert(messagestring,bytes)):
> print(stringofhex);
> lengthofmess := nops(stringofhex):
> print(lengthofmess);
> #Now we concatenate the hex numbers
> hexstring := cat(seq(stringofhex[i],i=1..lengthofmess)):
> #Finally we convert the big number back to decimal
> convert(hexstring,decimal,hex):
> end:
> messnum1 := shortconverter(mess1);
```

$[50, 65, 72, 64, 65, 2D, 73, 65, 20, 6F, 20, 61, 6D, 69, 67, 6F, 2C, 0A, 6D, 61, 73, 20, 6A, 61, 6D, 61, 69, 73, 20, 61, 20, 6F, 70, 6F, 72, 74, 75, 6E, 69, 64, 61, 64, 65, 2E, 0A, 28, 4C, 65, 6D, 61, 20, 64, 6F, 20, 47, 41, 50, 48, 29]$

59

messnum1 := 382960449754841266138386121933851098195061024452026794\
715514777570809063113279908453484806054467198196049265583553\
377859661549406923803622298 01

Conversion vers un format hexa pour manipulation postérieure.

```
> sechex1 := convert(encoder(messnum1,e,n),hex);
```

sechex1 := *8119468CE3ED829E64F482392DD2C91ECECD3101789622E52A7\
C165F6C9AE536DF5A0DCF914FF468ED130E55A5D0F810019B3859B\
0004A12A95B622FA91A1CB04FD9*

Toujours afin de faciliter la manipulation le message est divisé dans blocs de 10 digits.

```
> length(sechex1);
```

132

```
> sechex2 := cat(seq("0",counter =
> 1..(140-length(sechex1))),sechex1);
```

sechex2 := "000000008119468CE3ED829E64F482392DD2C91ECECD3101789\
622E52A7C165F6C9AE536DF5A0DCF914FF468ED130E55A5D0F810 \
019B3859B0004A12A95B622FA91A1CB04FD9"

```
> secmess := [seq(substring(sechex2,10*i-9..10*i),i=1..14)];
```

secmess := ["0000000081", "19468CE3ED", "829E64F482", "392DD2C91E", "CECD310178", "9622E52A7C", "165F6C9AE5", "36DF5A0DCF", "914FF468ED", "130E55A5D0", "F810019B38", "59B0004A12", "A95B622FA9", "1A1CB04FD9"]

Maintenant il faut vérifier si c'est possible de retrouver le message et le decoder.

```
> hexvectodec := (hexvec,size) ->
> convert(cat(seq(hexvec[i],i=1..size)),decimal,hex):
```

Le resultat doit être égal à *messnum1* ci-dessus.

```
> newnum1 := decoder(hexvectodec(secmess,14),d,n);
```

newnum1 := 382960449754841266138386121933851098195061024452026794 7\
155147775708090631132799084534848060544671981960492655835 53\
778596615494069238036222980 1

```
> convert(newnum1,hex);
```

*50657264652D7365206F20616D69676F2C0A6D6173206A616D6169732061206\
F706F7274756E69646164652E0A284C656D6120646F204741504829*

Finalement, une procédire pour decoder le nombre et retrouver le message original.

```
> numtostring := proc(bignum)
> local hexstr1, listlength, declist, i:
> #convert to a hex string
> hexstr1 := convert(bignum,hex):
> #make sure the hex string has even length
> if (length(hexstr1) mod 2) = 1 then hexstr1 := cat(‘0‘,hexstr1) fi;
> #compute the number of characters
> listlength := length(hexstr1)/2:
> #convert to a vector of decimal numbers
> declist := linalg[vector](listlength);
> for i from 1 to listlength do
> declist[i] := convert(substring(hexstr1,2*i-1..2*i),decimal,hex);
> od;
> #convert the vector to a list, then to an ASCII string
> convert(convert(convert(declist,list),bytes),name);
> end:
> numtostring(newnum1);
```

Perde − se o amigo, \
mas jamais a oportunidade. \
(*Lema do GAPH*)

ANNEXE C

DES

Constantes pour le DES

Daniel Gomes Mesquita, LIRMM, France

mesquita@lirmm.fr

```
> restart;
```

L'algorithme de cryptographie DES utilise beaucoup de constantes. La première étape pour l'implanter à l'aide du xMaple est donc créer une feuille de travail pour définir et stocker ces constantes. Avant d'éxecuter l'exemple du DES, ces constantes doivent être déjà en mémoire.

C.4 S boxes

D'abord les boîtes de substitutions sont définies tant que matrices, après elles seront converties en tableaux.

C.4.1 S boxes comme matrices

```
> S1 := [
> [14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7],
> [0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8],
> [4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0],
> [15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13]]: S2 := [
> [15,1,8,14,6,11,3,4,9,7,2,13,12,0,5,10],
> [3,13,4,7,15,2,8,14,12,0,1,10,6,9,11,5],
> [0,14,7,11,10,4,13,1,5,8,12,6,9,3,2,15],
> [13,8,10,1,3,15,4,2,11,6,7,12,0,5,14,9]]: S3 := [
> [10,0,9,14,6,3,15,5,1,13,12,7,11,4,2,8],
> [13,7,0,9,3,4,6,10,2,8,5,14,12,11,15,1],
> [13,6,4,9,8,15,3,0,11,1,2,12,5,10,14,7],
> [1,10,13,0,6,9,8,7,4,15,14,3,11,5,2,12]]: S4 := [
> [7,13,14,3,0,6,9,10,1,2,8,5,11,12,4,15],
> [13,8,11,5,6,15,0,3,4,7,2,12,1,10,14,9],
> [10,6,9,0,12,11,7,13,15,1,3,14,5,2,8,4],
> [3,15,0,6,10,1,13,8,9,4,5,11,12,7,2,14]]: S5 := [
> [2,12,4,1,7,10,11,6,8,5,3,15,13,0,14,9],
> [14,11,2,12,4,7,13,1,5,0,15,10,3,9,8,6],
> [4,2,1,11,10,13,7,8,15,9,12,5,6,3,0,14],
> [11,8,12,7,1,14,2,13,6,15,0,9,10,4,5,3]]: S6 := [
> [12,1,10,15,9,2,6,8,0,13,3,4,14,7,5,11],
> [10,15,4,2,7,12,9,5,6,1,13,14,0,11,3,8],
> [9,14,15,5,2,8,12,3,7,0,4,10,1,13,11,6],
> [4,3,2,12,9,5,15,10,11,14,1,7,6,0,8,13]]: S7 := [
> [4,11,2,14,15,0,8,13,3,12,9,7,5,10,6,1],
> [13,0,11,7,4,9,1,10,14,3,5,12,2,15,8,6],
> [1,4,11,13,12,3,7,14,10,15,6,8,0,5,9,2],
> [6,11,13,8,1,4,10,7,9,5,0,15,14,2,3,12]]: S8 := [
> [13,2,8,4,6,15,11,1,10,9,3,14,5,0,12,7],
> [1,15,13,8,10,3,7,4,12,5,6,11,0,14,9,2],
> [7,11,4,1,9,12,14,2,0,6,10,13,15,3,5,8],
> [2,1,14,7,4,10,8,13,15,12,9,0,3,5,6,11]]:
> sbox :=[S1,S2,S3,S4,S5,S6,S7,S8]:
```

C.4.2 S boxes comme tableaux

Cela facilite le traitement informatique. Il est nécessaire une liste de tableaux, où chaque tableau réprésente une S-Box. L'index doit avoir des vecteurs de 6 bits de largeur, et la valeur stocké doit garder 4 bits. Les fonctions nécessaires à la conversion sont définies ci-dessous.

```
> indexer := (i,j,k) ->
> substring(convert(convert(
> (i-1)*32+(j-1)*2+(k-1)+64,binary),string),2..7):
> fourbitbin := decnum ->
> substring(convert(convert(decnum+16,binary),string),2..5):

> SBox := [seq(table(
> [seq(seq(seq(
> indexer(i,j,k)=fourbitbin(sbox[m][2*(i-1)+k,j]),j=1..16),k=1..2),i=1..
> 2)]),
> m=1..8)]:
```

C'est intéressant de visualizer au moins une de ces S-Boxes comme une matrice :

```
> showAsMatrix := Table ->
> matrix(4,16,[seq(seq(
> [seq(
> Table[indexer(i,j,k)],j=1..16)],
> k=1..2),i=1..2)]):
> showAsMatrix(SBox[8]);
```

[“1101” , “0010” , “1000” , “0100” , “0110” , “1111” , “1011” , “0001” , “1010” , “1001” , “0011” , “1110” , “0101” , “0000” , “1100” , “0111”]
[“0001” , “1111” , “1101” , “1000” , “1010” , “0011” , “0111” , “0100” , “1100” , “0101” , “0110” , “1011” , “0000” , “1110” , “1001” , “0010”]
[“0111” , “1011” , “0100” , “0001” , “1001” , “1100” , “1110” , “0010” , “0000” , “0110” , “1010” , “1101” , “1111” , “0011” , “0101” , “1000”]
[“0010” , “0001” , “1110” , “0111” , “0100” , “1010” , “1000” , “1101” , “1111” , “1100” , “1001” , “0000” , “0011” , “0101” , “0110” , “1011”]

C.5 Constantes de Permutation

Maintenant les constantes de Permutation.

```
> PC1 :=[
57,49,41,33,25,17,9,
1,58,50,42,34,26,18,
10,2,59,51,43,35,27,
19,11,3,60,52,44,36,
63,55,47,39,31,23,15,
7,62,54,46,38,30,22,
14,6,61,53,45,37,29,
21,13,5,28,20,12,4]:
PC2 :=[
14,17,11,24,1,5,
3,28,15,6,21,10,
23,19,12,4,26,8,
16,7,27,20,13,2,
41,52,31,37,47,55,
30,40,51,45,33,48,
44,49,39,56,34,53,
46,42,50,36,29,32]:
keyshifts := [1,1,2,2,2,2,2,2,1,2,2,2,2,2,2,1]:
IP := [
58,50,42,34,26,18,10,2,
60,52,44,36,28,20,12,4,
62,54,46,38,30,22,14,6,
64,56,48,40,32,24,16,8,
57,49,41,33,25,17,9,1,
59,51,43,35,27,19,11,3,
61,53,45,37,29,21,13,5,
63,55,47,39,31,23,15,7]:
E := [
32, 1, 2, 3, 4, 5,
4, 5, 6, 7, 8, 9,
8, 9,10,11,12,13,
12,13,14,15,16,17,
16,17,18,19,20,21,
20,21,22,23,24,25,
24,25,26,27,28,29,
28,29,30,31,32, 1]:
P := [
16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25]:
```

```
> IPI := [ 40,8,48,16,56,24,
64,32,39,7,47,15, 55,23,63,31,38,6, 46,14,54,22,62,30,
37,5,45,13,53,21, 61,29,36,4,44,12, 52,20,60,28,35,3,
43,11,51,19,59,27, 34,2,42,10,50,18, 58,26,33,1,41, 9,
49,17,57,25]:
```

Il est intéressante de voir que IP et IPI sont les inverses l'une de l'autre.

```
> seq(IP[IPI[i]],i=1..64);
```

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64

Après avoir éxecuté cette feuille de travail, il faut utiliser la feuille ds fonctions DES.

Une colléction de procédures utilisées par le DES

Afin d'éxecuter le DES sont nécessaires plusieures fonctions. Pour cela il est intéressant de les définir auparavant dans une feuille séparée.

Les quatre premièrese sections de fonctions sont utilisées dans chaque étape du DES. Elles concernent conversion de données, permutations, XOR et lecture des s-boxes.

La section suivante montre les fonctions qui produisent la Fonction de Feistell (F function). Le but de ces 5 sections c'est de focaliser sur le fonctionnement du DES sur un bloc de message.

Les trois dernières sections de fonctions sont projetées afin de réaliser des opérations focalisent les modes d'opérations du DES, plutôt que le fonctionnement d'un seul bloc. De ces 3 sections, la première aide à l'expansion de clé, la deuxième traite de crypter et de decrypter, et la troisième fait la conversion d'un message long en ASCII vers un vecteur à huit octets de mots en hexa.

C.6 Conversion entre différents types de données

Cette première section de codes contient des fonctions pour convertir vers hexa où binaire, tout en gardant les tailles correctes.

```
> #When we convert a decimal equivalent to hex, we want 2-digit hex
> hex2digit := x -> if x < 16 then cat(‘0‘,convert(x,hex))
> else convert(x,hex) fi:
> #We want to convert an ASCII string of up to 8 characters to a
> #64 bit binary string.
> bin64 := proc(shortstring)
> local temp,l1,missing,i, binstr:
> temp := convert(shortstring,bytes):
> l1 := max(1,length(shortstring)):
> binstr := convert(convert( cat(seq(hex2digit(temp[i]),i=1..l1)),
> decimal,hex),binary):
> missing := 64 -length(binstr):
> cat(seq(‘0‘,i=1..missing),binstr):
> end:
> #We want to convert an ASCII string of up to 8 characters to a
> #16 character hex string.
> ASCII2hex16 := proc(shortstring)
> local temp,l1,missing,i, hexstr:
> temp := convert(shortstring,bytes):
> l1 := max(1,length(shortstring)):
> hexstr := cat(seq(hex2digit(temp[i]),i=1..l1)):
> missing := 16 -length(hexstr):
> cat(seq(‘0‘,i=1..missing),hexstr):
> end:
> #We want to convert an hex string of up to 16 characters to a
> #64 bit binary string.
> bin64hex := proc(hexstring)
> local temp,l1,missing,i, binstr:
> l1 := 16 - max(1,length(hexstring)):
> temp := cat(seq("0",i=1..l1),hexstring):
> binstr := convert(convert( temp,decimal,hex),binary):
> missing := 64 -max(1,length(binstr)):
> cat(seq("0",i=1..missing),binstr):
> end:
> #The contents of s-boxes should be converted to 4 bit binary
> fourbitbin := decnum ->
> substring(convert(convert(decnum+16,binary),string),2..5):
```

C.7 Permutation de vecteurs et XOR

Fonctions pour les permutations et expansions.

```
> PC1onKey := binstr -> cat(seq(substring(binstr,PC1[i]),i=1..56)):
> PC2onKey := binstr -> cat(seq(substring(binstr,PC2[i]),i=1..48)):
> expander := binstr -> cat(seq(substring(binstr,E[i]),i=1..48)):
> InitPerm := binstr -> cat(seq(substring(binstr,IP[i]),i=1..64)):
> FinalPerm := binstr -> cat(seq(substring(binstr,IPI[i]),i=1..64)):
> PPerm := binstr -> cat(seq(substring(binstr,P[i]),i=1..32)):
```

L'opération XOR est nécessaire dans des différents largeurs de mot : 64, 48, et 32 . Il est aussi importante de réaliser un XOR entre deux mots de 64bits réçus en hexa.

```
> XOR := (a,b) -> if (a=b) then "0" else "1" fi:
> xorNbits := proc(a,b,N)
> local aString, bString:
> aString := convert(a,string):
> bString := convert(b,string):
> cat(seq(XOR(substring(aString,i), substring(bString,i)),i=1..N)):
> end:
> xor64 := (a,b) -> xorNbits(a,b,64):
> xor48 := (a,b) -> xorNbits(a,b,48):
> xor32 := (a,b) -> xorNbits(a,b,32):
> xor64hex := proc(a,b)
> local binab, hexab, i, l:
> binab := xor64(bin64hex(a),bin64hex(b)):
> hexab := convert(convert(parse(binab),decimal,binary),hex):
> l := 16 - length(hexab):
> cat(seq("0",i=1..l),hexab):
> end:
```

C.8 S-Boxes et la Fonction de Feistell

Afin de réaliser les substitutions des s-boxes, il faut découper les mots de 48 bits dans des vecteurs de 6 bits.

```
> prodbs := proc(binstr)
> local bvec,i:
> bvec := linalg[vector](8):
> for i from 1 to 8 do
> bvec[i] := substring(binstr,i*6-5..i*6):
> od:
> convert(bvec,list):
> end:
```

Il fault maintenant une fonction correspondant à la Fonction F. Elle doit prendre une sous-clé d'une étape du DES et une demi-mot, faire l'expantion du demi-mot et le XOR avec la sous-clé, et ensuite découper le résultat de 48 bits dans 8 mots de 6 bits, qui seront utilisées ensuite par les S-Boxes. Les résultats sont concaténés et permutés en accord avec la définition de l'algorithme.

```
> fcomb := proc(ri, ki)
> local t1,t2, t3, vec1, vec2, i:
> t1 := xor48(ki,expander(ri)):
> vec1 := prodbs(t1):
> vec2 := linalg[vector](8,[seq(SBox[i][vec1[i]],i=1..8)]):
> t2 := cat(seq(vec2[i],i=1..8)):
> t3 := PPerm(t2):
> end:
```

C.9 Commandes en ligne

Des fonctions pour faciliter l'utilisation de cet exemple quand il y a plus de 64 bits d'informations à traiter.

```
> keyexpander := proc(hexstring)
> local keybin, binlength, keybin64, PC1key, c, d, key, i:
> keybin:= convert(convert(hexstring,decimal,hex),binary):
> binlength := length(keybin):
> keybin64 := cat(seq(‘0‘,i=1..(64-binlength)),keybin):
> PC1key := PC1onKey(keybin64):
> c := linalg[vector](17): d := linalg[vector](17):
> key := linalg[vector](16):
> c[1] := substring(PC1key,1..28): d[1]:=substring(PC1key,29..56):
> for i from 2 to 17 do
> c[i] := cat(substring(c[i-1],keyshifts[i-1]+1..28),
> substring(c[i-1],1..keyshifts[i-1])):
> d[i] := cat(substring(d[i-1],keyshifts[i-1]+1..28),
> substring(d[i-1],1..keyshifts[i-1])):
> od:
> for i from 1 to 16 do
> key[i] := convert(PC2onKey(cat(c[i+1],d[i+1])),string);
> od:
> key := convert(key,list):
> end:
> #When we convert a decimal equivalent to hex, we want 2-digit hex
> hex2digit := x -> if x < 16 then cat(‘0‘,convert(x,hex))
> else convert(x,hex) fi:
> asciistrtohexword := proc(messagestring)
> local stringofhex, lenofstr, roundlen, padding, wordlen, temp, j, i,
> k:
> #First we convert the ASCII string to a list of decimal
> equivalents
> #Then we convert thedecimal numbers to 2 place hex equivalents
> lenofstr := length(messagestring);
> stringofhex := map(hex2digit, convert(messagestring,bytes)):
> #Time to compute the appropriate lengths
> roundlen := lenofstr + 7 - (lenofstr + 7 mod 8);
> padding := roundlen - lenofstr:
> wordlen := roundlen/8:
```

```
> #Now we put togehter hex words
> temp := linalg[vector](wordlen):
> for i from 1 to (wordlen -1) do
> temp[i] := cat(seq(stringofhex[j],j=(i*8-7)..i*8)):
> od:
> temp[wordlen] :=cat(seq(stringofhex[j], j=(roundlen-7)..lenofstr),
> seq(‘00‘,k=1..padding)):
> #Finally we convert the vector to a list
> convert(temp,list):
> end:
> hexwordtoasciistr := proc(wordlist)
> local l1, temp, i, j, vec1:
> vec1 := convert(wordlist,array):
> l1 := linalg[vectdim](wordlist):
> temp := linalg[vector](8*l1):
> for i from 1 to l1 do
> for j from 1 to 8 do
> temp[j+8*(i-1)] := substring(wordlist[i],2*j-1..2*j):
> od:
> od:
> temp := convert(temp,list):
> convert(map(convert,temp,decimal,hex),bytes):
> end:
```

C.10 Sauvegarde des fonctions et des constantes

Pour éviter le besoin de charger à la main les feuilles de travaux contenant les fonctions et constantes, xMaple permet de sauvegarder un ficher avec ces définitions, qui chargerà automatiquement les procédures et valeurs nécessaires à l'éxecution des exemples..

```
> save
> SBox, PC1, PC2, keyshifts, IP, E, P, IPI,
> hex2digit, bin64, ASCII2hex16, bin64hex, fourbitbin,
> PC1onKey, PC2onKey, expander, InitPerm, FinalPerm, PPerm,
> XOR, xorNbits, xor64, xor48, xor32, xor64hex,
> prodbs, fcomb, keyexpander, qdDEShex, unDEShex,
> qdDESASCII,
> hex2digit, asciistrtohexword, hexwordtoasciistr,
> ‘DES.m‘;
```

Ensuite il faut éxecuter la feuille de travail Expansion de Clés du DES

Expansion des clés du DES

D'abord il faut charger le fichier contenant les définitions du DES : DES.m.

```
> read ‘DES.m‘:
```

La clé est convertie vers le format utilisé par le DES.

```
> keytest := "133457799BBCDFF1";
> keybin:= convert(convert(keytest,decimal,hex),binary);
> binlength := length(keybin);
```

$$keytest := \text{“133457799BBCDFF1”}$$

$$keybin := 1001100110100010101110111100110011011101111001101111111110001$$

$$binlength := 61$$

Comme la clé avait 61 bits et le DES a besoin initialement de 64 bits, sont insérées des zéros pour compbler la clé.

```
> keybin64 := cat(seq("0",i=1..(64-binlength)),keybin);
```

$$keybin64 := \text{“0001001100110100010101110111100110011011101111001101111111110001”}$$

Ensuite les bits de parité sont enlévés.

```
> keybin56 :=cat(seq(substring(keybin64,i*8-7..i*8-1),i=1..8));
```

$$keybin56 := \text{“00010010011010010101110111100100110110111101101111111000”}$$

La première permutation est apliquée.

```
> PC1key := PC1onKey(keybin64);
```

$$PC1key := \text{“11110000110011001010101011111010101010110011001111000111 1”}$$

La clé est donc divisée dans deux blocs.

```
> c0 := substring(PC1key,1..28): d0:=substring(PC1key,29..56):
```

Les moitiées sont permutées par des décalages à gauche, selon la formule.

```
> c := linalg[vector](16): d := linalg[vector](16):
> key := linalg[vector](16):
> c[1] := cat(substring(c0,2..28),substring(c0,1)):
> d[1] := cat(substring(d0,2..28),substring(d0,1)):
> for i from 2 to 16 do
> c[i] := cat(substring(c[i-1],keyshifts[i]+1..28),
> substring(c[i-1],1..keyshifts[i])):
> d[i] := cat(substring(d[i-1],keyshifts[i]+1..28),
> substring(d[i-1],1..keyshifts[i])):
> od:
```

Les clés sont rassemblées au travers du PC2.

```
> for i from 1 to 16 do
> key[i] := convert(PC2onKey(cat(c[i],d[i])),string);
> od;
```

$$key_1 := \text{“}000110110000001011101111111111000111000001110010\text{”}$$
$$key_2 := \text{“}011110011010111011011001110110111100100111100101\text{”}$$
$$key_3 := \text{“}010101011111111001000101001000010110011111100111001\text{”}$$
$$key_4 := \text{“}011100101010110111010110110110110011010100011101\text{”}$$
$$key_5 := \text{“}011111001110110000000111111010110101001110101000\text{”}$$
$$key_6 := \text{“}011000111010010100111110010100000111101100101111\text{”}$$
$$key_7 := \text{“}111011001000010010110111111110110000110001011100\text{”}$$
$$key_8 := \text{“}111101111000101000111010110000010011101111111011\text{”}$$
$$key_9 := \text{“}111000001101101111101011111011011110011110000001\text{”}$$
$$key_{10} := \text{“}101100011111001101000111101110100100011001001111\text{”}$$
$$key_{11} := \text{“}001000010101111111010011110111101101001110000110\text{”}$$
$$key_{12} := \text{“}011101010111000111110101100101000110011111101001\text{”}$$
$$key_{13} := \text{“}100101111100010111010001111110101011101001000001\text{”}$$
$$key_{14} := \text{“}010111110100001110110111111100101110011100111010\text{”}$$
$$key_{15} := \text{“}101111111001000110001101001111010011111100001010\text{”}$$
$$key_{16} := \text{“}110010110011110110001011000011100001011111110101\text{”}$$

Maintenant c'est possible de regarder un exemple complét du DES, et du DES selon l'un des modes d'éxecution qu'il possède..

Un exemple du DES

```
> restart:
```

Cet example utilise les fonctions et constantes définies auparavant.

```
> read ‘DES.m‘:
```

C.11 Setup Initial

Au départ une clé est étendue à toutes les étapes du DES.

```
> keytest := "133457799BBCDFF1";
> key := keyexpander(keytest):
```

keytest := “133457799BBCDFF1”

Alors un message en hexa est converti en binaire.

```
> mess1 := "0123456789ABCDEF";
> bin1 := bin64hex(mess1);
```

mess1 := “0123456789ABCDEF”

bin1 := “0000000100100011010001010110011111000100110101011110011010111101111”

La permutation IP est apliquée.

```
> bin2 := InitPerm(bin1);
```

bin2 := “1100110000000000110011001111111111110000101010101111000010101010”

Ensuite est possible de définir les registres R0 et L0.

```
> R0 := substring(bin2,33..64);
> L0 := substring(bin2,1..32);
```

R0 := “11110000101010101111000010101010”

L0 := “11001100000000001100110011111111”

C.12 Une étape de cryptage

Le registre R0 est étendu et il est fait un xor avec la clé key[1].

```
> expander(R0);
> key[1];
> bin3 := xor48(key[1],expander(R0));
```

“011110100001010101010101011110100001010101010101”

“000110110000001011101111111111000111000001110010”

bin3 := “011000010001011110111010100001100110010100100111”

Ensuite il est nécessaire prendre le résultat sur 48 bits et le découper dans 8 mots de 6 bits. Chaque mot est utilisée par une S-Box.

```
> vec1 := prodbs(bin3);
> vec2 := linalg[vector](8,[seq(SBox[i][vec1[i]],i=1..8)]);
```

vec1 := [“011000”, “010001”, “011110”, “111010”, “100001”, “100110”, “010100”, “100111”]

vec2 := [“0101”, “1100”, “1000”, “0010”, “1011”, “0101” , “1001”, “0111”]

Après, les valeurs sont converties dans des mots de 4 bits et concaténées ensemble.

```
> bin4 := cat(seq(vec2[i],i=1..8));
```

bin4 := “01011100100000101011010110010111”

Le résultat de la concaténation est permuté, afin de générer la valeur F..

```
> f1 := PPerm(bin4);
```

f1 := “00100011010010101010100110111011”

Le résultat est XORé avec L0 pour former R1. L1 est tout simplement R0.

```
> R1 := xor32(f1,L0);
> L1 := R0;
```

R1 := “11101111010010100110010101000100”

L1 := “11110000101010101111000010101010”

C.13 Les 15 étapes suivantes de cryptage

A l’aide d’une boucle, les étapes suivantes sont calculées.

```
> L := linalg[vector](16):
> R := linalg[vector](16):
> L[1] := L1;
> R[1] := R1;
```

L_1 := “11110000101010101111000010101010”

R_1 := “11101111010010100110010101000100”

```
> fcomb := proc(ri, ki)
> local t1,t2, t3, vec1, vec2, i:
> t1 := xor48(ki,expander(ri)):
> vec1 := prodbs(t1):
> vec2 := linalg[vector](8,[seq(SBox[i][vec1[i]],i=1..8)]):
> t2 := cat(seq(vec2[i],i=1..8)):
> t3 := PPerm(t2):
> end:
```

```
> for i from 2 to 16 do
> L[i] := R[i-1];
> R[i] := xor32(L[i-1],fcomb(R[i-1],key[i]));
> od;
```

$$L_2 := \text{“}11101111010010100110010101000100\text{”}$$
$$R_2 := \text{“}11001100000000010111011100001001\text{”}$$
$$L_3 := \text{“}11001100000000010111011100001001\text{”}$$
$$R_3 := \text{“}10100010010111000000101111110100\text{”}$$
$$L_4 := \text{“}10100010010111000000101111110100\text{”}$$
$$R_4 := \text{“}01110111001000100000000001000101\text{”}$$
$$L_5 := \text{“}01110111001000100000000001000101\text{”}$$
$$R_5 := \text{“}10001010010011111010011000110111\text{”}$$
$$L_6 := \text{“}10001010010011111010011000110111\text{”}$$
$$R_6 := \text{“}11101001011001111100110101101001\text{”}$$
$$L_7 := \text{“}11101001011001111100110101101001\text{”}$$
$$R_7 := \text{“}00000110010010101011101000010000\text{”}$$
$$L_8 := \text{“}00000110010010101011101000010000\text{”}$$
$$R_8 := \text{“}11010101011010010100101110010000\text{”}$$
$$L_9 := \text{“}11010101011010010100101110010000\text{”}$$
$$R_9 := \text{“}00100100011111001100011001111010\text{”}$$
$$L_{10} := \text{“}00100100011111001100011001111010\text{”}$$
$$R_{10} := \text{“}10110111110101011101011110110010\text{”}$$
$$L_{11} := \text{“}10110111110101011101011110110010\text{”}$$
$$R_{11} := \text{“}11000101011110000011110001111000\text{”}$$
$$L_{12} := \text{“}11000101011110000011110001111000\text{”}$$
$$R_{12} := \text{“}01110101101111010001100001011000\text{”}$$
$$L_{13} := \text{“}01110101101111010001100001011000\text{”}$$
$$R_{13} := \text{“}00011000110000110001010101011010\text{”}$$
$$L_{14} := \text{“}00011000110000110001010101011010\text{”}$$
$$R_{14} := \text{“}11000010100011001001011000001101\text{”}$$
$$L_{15} := \text{“}11000010100011001001011000001101\text{”}$$
$$R_{15} := \text{“}01000011010000100011001000110100\text{”}$$
$$L_{16} := \text{“}01000011010000100011001000110100\text{”}$$
$$R_{16} := \text{“}00001010010011001101100110010101\text{”}$$

Maintenant les moitiées droite et gauche sont rassemblés, mais dans le désordre. Alors il est apliqué l'inverse de la permutation initiale, et ensuite la conversion vers le hexadecimal.

```
> puttogether := cat(R[16],L[16]);
> bincodetext := FinalPerm(puttogether);
> ciphertext :=
> convert(convert(parse(bincodetext),decimal,binary),hex);
```

puttogether :=
"0000101001001100110110011001010101000011010000100011001000110100"
bincodetext :=
"10000101111010000001001101010100000011110000101010110100000000101"
ciphertext := *85E813540F0AB405*

C.14 Pour decrypter

La procédure est essentiellement l'inverse ducryptage..

```
> R2 := linalg[vector](17):
> L2 := linalg[vector](17):
> R2[17] := R[16];
> L2[17] := L[16];
```

$R2_{17}$:= "00001010010011001101100110010101"
$L2_{17}$:= "01000011010000100011001000110100"

```
> for i from 1 to 16 do
> j := 17-i:
> R2[j] := L2[j+1];
> L2[j] := xor32(R2[j+1],fcomb(L2[j+1],key[j]));
> od;
```

$j := 16$
$R2_{16}$:= "01000011010000100011001000110100"
$L2_{16}$:= "11000010100011001001011000001101"
$j := 15$
$R2_{15}$:= "11000010100011001001011000001101"
$L2_{15}$:= "00011000110000110001010101011010"
$j := 14$
$R2_{14}$:= "00011000110000110001010101011010"
$L2_{14}$:= "01110101101111010001100001011000"
$j := 13$
$R2_{13}$:= "01110101101111010001100001011000"
$L2_{13}$:= "11000101011110000011110001111000"
$j := 12$
$R2_{12}$:= "11000101011110000011110001111000"
$L2_{12}$:= "10110111110101011101011110110010"

$$j := 11$$
$$R2_{11} := \text{“}10110111110101011101011110110010\text{”}$$
$$L2_{11} := \text{“}00100100011111001100011001111010\text{”}$$
$$j := 10$$
$$R2_{10} := \text{“}00100100011111001100011001111010\text{”}$$
$$L2_{10} := \text{“}11010101011010010100101110010000\text{”}$$
$$j := 9$$
$$R2_{9} := \text{“}11010101011010010100101110010000\text{”}$$
$$L2_{9} := \text{“}00000110010010101011101000010000\text{”}$$
$$j := 8$$
$$R2_{8} := \text{“}00000110010010101011101000010000\text{”}$$
$$L2_{8} := \text{“}11101001011001111100110101101001\text{”}$$
$$j := 7$$
$$R2_{7} := \text{“}11101001011001111100110101101001\text{”}$$
$$L2_{7} := \text{“}10001010010011111010011000110111\text{”}$$
$$j := 6$$
$$R2_{6} := \text{“}10001010010011111010011000110111\text{”}$$
$$L2_{6} := \text{“}01110111001000100000000001000101\text{”}$$
$$j := 5$$
$$R2_{5} := \text{“}01110111001000100000000001000101\text{”}$$
$$L2_{5} := \text{“}10100010010111000000101111110100\text{”}$$
$$j := 4$$
$$R2_{4} := \text{“}10100010010111000000101111110100\text{”}$$
$$L2_{4} := \text{“}11001100000000010111011100001001\text{”}$$
$$j := 3$$
$$R2_{3} := \text{“}11001100000000010111011100001001\text{”}$$
$$L2_{3} := \text{“}11101111010010100110010101000100\text{”}$$
$$j := 2$$
$$R2_{2} := \text{“}11101111010010100110010101000100\text{”}$$
$$L2_{2} := \text{“}11110000101010101111000010101010\text{”}$$
$$j := 1$$
$$R2_{1} := \text{“}11110000101010101111000010101010\text{”}$$
$$L2_{1} := \text{“}11001100000000001100110011111111\text{”}$$

Après les 16 étapes, les deux moitiées sont rassemblées, de-mixées, et ensuite reconverties vers le texte ASCII d'origine.

```
> almost := cat(L2[1],R2[1]);
> unscrambled := FinalPerm(almost);
> code3 := convert(convert(parse(unscrambled),decimal,binary),hex);
```

almost :=
"1100110000000000011001100111111111111000010101010111100001010101010"

unscrambled :=
"0000000100100011010001010110011111000100110101011110011011110111 1"

code3 := *123456789ABCDEF*

Et voici le message original :

```
> mess1;
```

"0123456789ABCDEF"

ANNEXE D

Algorithme de Montgomery

Un exemple de l'algorithme de Montgomery

Daniel Gomes Mesquita, LIRMM, France

mesquita@lirmm.fr

```
> restart;
```

L'algorithme de Montgomery est l'un des plus répandus pour réaliser la multiplication modulaire. Il est décrit dans la Section 2.7 de cette thèse. Ici il est presenté une implanatation sur xMaple en guise d'exemple pratique.

```
> mont := proc (A,B,M,n,base)
> local q,R,T,ai,i,im,b0, r0, y;
> b0 := B mod base;
> T := A;
> R := 0;
> #L'inverse modulaire peut être calculée à l'aide de l'algorithme d'Euclide Etendu, #mais ici sont utilisées des fonctions xMaple
> im := (M^(-1) mod base);
> printf("im = %a^-1 mod %a = %a \n",M,base,im);
> for i from 0 to n-1
> do
> ai := T mod base;
> #L'opération suivante, en sert à découper un grand nombre vers des mots de k bits, k étant l'exposant de la base.
> T:= (T - ai)/ base;
> r0 := R mod base;
> q := (-(r0+ai*b0)* im) mod base;
> R := (R+ai*B+q*M)/base;
> printf(" i = %a, q = %a; R = %a, \n",i,q,R);
> od;
> return R
> end:
```

Tout d'abord il faut choisir les valeurs selon les caractéristiques décrites dans l'Algorithme 6.

```
> a := 1729;
> b := 1256;
> m := 3057;
> n := 4;
> k := 16;
> base := 2^k;
```

$$a := 1729$$
$$b := 1256$$
$$m := 3057$$
$$n := 4$$
$$k := 16$$
$$base := 65536$$

Ensuite le calcul peut être réalisé.

```
> r := mont(a,b,m,n,base);
```

```
im = 3057^-1 mod 65536 = 34065

 i = 0, q = 9880; R = 494,

 i = 1, q = 14642; R = 683,

 i = 2, q = 64421; R = 3005,

 i = 3, q = 1907; R = 89,
```

$$r := 89$$

Pourtant le résultat calculé ne correspond pas au résultat espéré pour l'opération "a * b mod m".

```
> espere := (a*b) mod m;
```

$$espere := 1154$$

Cela est normal, car l'algorithme de Montgomery calcule au fait "(a*b*base^(-n)) mod m". Donc, le resultat calculé est correct.

```
> calcule := r;  espere := (a*b*base^(-n)) mod m;
```

$$calcule := 89$$
$$espere := 89$$

```
Finalement, pour retrouver la valeur correcte de la multiplication
modulaire "a*b mod m", il faut un deuxième passage par l'algorithme,
multipliant le résultat précédent par base^(2*n+1).
> finalCalcule := mont(r,(base^(2*n+1) mod m),m,n+1,base); finalEspere
> := a * b mod m;


im = 3057^-1 mod 65536 = 34065

 i = 0, q = 37197; R = 1739,

 i = 1, q = 5509; R = 257,

 i = 2, q = 27119; R = 1265,

 i = 3, q = 30463; R = 1421,

 i = 4, q = 24739; R = 1154,
```

$$finalCalcule := 1154$$
$$finalEspere := 1154$$

ANNEXE E

LRA

Program : Leak Resistant Arithmetic(LRA) Author : Daniel Mesquita Date : 12/12/2005 Last modified : 15/02/06

Goal : To implement the functions set needed to perform the LRA proposed by J-C Bajard and L. Imbert. Calculate A x B mod N without trial division

```
>  restart;
>  geraBase := proc (tamanho,inicio)
```

> This procedure generates the basis that will compose Beta1 and Beta2 It receives the number of basis (tamanho) and the initial base.

```
>  local i, j, ehPrimo, flag, candidato, baseRNS : aray(0..tamanho-1);

>  candidato := inicio;

>  baseRNS[0] := inicio;

>  i := 0;

>  while (i < tamanho)
>  do

>  j := 0;

>  flag := 0;

>  while (j < i) and (flag = 0)

>  do

>  ehPrimo := gcd(candidato,baseRNS[j]);
```

> this condition tests the relative primality between two basis

```
>  if (ehPrimo = 1) then

>  j := j+1;

>  else

>  flag := 1;

>  end if;

>  od;

>  if (flag = 0) then
```

> All chosen basis MUST be relative primes

```
>  baseRNS[i] := candidato;

>  i := i + 1;

>  end if;

>  candidato := candidato + 1;

>  od;

>  return [seq(baseRNS[i],i=0..tamanho-1)];
```

> The function returns an array of 2k basis

```
>  end;
```

geraBase := **proc**(*tamanho*, *inicio*)
local *i*, *j*, *ehPrimo*, *flag*, *candidato*, *baseRNS*;
array(0..*tamanho* − 1) ;
candidato := *inicio* ;
$baseRNS_0$:= *inicio* ;
i := 0 ;
while *i* < *tamanho* **do**
j := 0 ;
flag := 0 ;
while *j* < *i* **and** *flag* = 0 **do**
ehPrimo := gcd(*candidato*, $baseRNS_j$) ;
if *ehPrimo* = 1 **then** *j* := *j* + 1 **else** *flag* := 1 **end if**
end do;
if *flag* = 0 **then** $baseRNS_i$:= *candidato* ; *i* := *i* + 1 **end if**;
candidato := *candidato* + 1
end do;
return [seq($baseRNS_i$, *i* = 0..*tamanho* − 1)]
end proc

```
> geraM := proc(vetorBase)
> This function returs M, that is the product of mi (generated before)
> local i, Num;
> Num := 1;
> for i from 0 to k-1
> do
> Num := Num * vetorBase[i+1];
> od;
> end;
```

geraM := **proc**(*vetorBase*)
local *i*, *Num*;
Num := 1 ; **for** *i* **from** 0 **to** *k* − 1 **do** *Num* := *Num* ∗ $vetorBase_{i+1}$ **end do**
end proc

```
> convDECtoRNS:=proc (T,vetorBase)
> A simple function to convert a number from decimal to RNS representation
> local i, base, tRNS;
> for i from 0 to k-1
> do
> tRNS[i] := T mod vetorBase[i+1];
> od;
> return [seq(tRNS[i],i=0..k-1)];
> end;
```

$convDECtoRNS := \mathbf{proc}(T,\ vetorBase)$
$\mathbf{local}\, i,\ base,\ tRNS;$
$\quad \mathbf{for}\, i\, \mathbf{from}\, 0\, \mathbf{to}\, k-1\, \mathbf{do}\, tRNS_i := T \bmod vetorBase_{i+1}\, \mathbf{end\ do};$
$\quad \mathbf{return}\ [\mathrm{seq}(tRNS_i,\ i = 0..k-1)]$
$\mathbf{end\ proc}$

```
> geraMi := proc (vetorBase, M)
> This procedure generates the Mi values, that are the division of M by each mi
> local i, vetorMi;
> for i from 0 to k-1
> do
> vetorMi[i] := M / vetorBase[i+1];
> od;
> return [seq(vetorMi[i],i=0..k-1)];
> end;
```

$geraMi := \mathbf{proc}(vetorBase,\ M)$
$\mathbf{local}\, i,\ vetorMi;$
$\quad \mathbf{for}\, i\, \mathbf{from}\, 0\, \mathbf{to}\, k-1\, \mathbf{do}\, vetorMi_i := M/vetorBase_{i+1}\, \mathbf{end\ do};$
$\quad \mathbf{return}\ [\mathrm{seq}(vetorMi_i,\ i = 0..k-1)]$
$\mathbf{end\ proc}$

```
> geraIm := proc (vetorM, vetorMi)
> Computes the modular inverses of Mi concerning each mi
> local i, invMi;
> for i from 0 to k-1
> do
> invMi[i] := vetorMi[i+1]^(-1) mod vetorM[i+1];
> od;
> return [seq(invMi[i],i=0..k-1)];
> end;
```

$geraIm := \mathbf{proc}(vetorM,\ vetorMi)$
$\mathbf{local}\, i,\ invMi;$
$\quad \mathbf{for}\, i\, \mathbf{from}\, 0\, \mathbf{to}\, k-1\, \mathbf{do}\, invMi_i := 1/vetorMi_{i+1} \bmod vetorM_{i+1}\ \mathbf{end\ do};$
$\quad \mathbf{return}\ [\mathrm{seq}(invMi_i,\ i = 0..k-1)]$
$\mathbf{end\ proc}$

```
> convRNStoDEC := proc (vetorX, vetorMi,vetorInvMi, M)
> A function to convert a number from the RNS format to the decimal number system
> It needs to receive the array of basis and the corresponding modular inverses, besides the RNS number to convert.
> local i,mTemp;
> mTemp := 0;
> for i from 0 to k-1
> do
> mTemp := mTemp + (vetorX[i+1]*vetorMi[i+1]*vetorInvMi[i+1]);
> od;
> mTemp := mTemp mod M;
> return mTemp;
> end;
```

$$
\begin{array}{l}
convRNStoDEC := \mathbf{proc}(vetorX,\ vetorMi,\ vetorInvMi,\ M) \\
\mathbf{local}\, i,\ mTemp; \\
\quad mTemp := 0\,; \\
\quad \mathbf{for}\, i\, \mathbf{from}\, 0\, \mathbf{to}\, k-1\, \mathbf{do} \\
\qquad mTemp := mTemp + vetorX_{i+1} * vetorMi_{i+1} * vetorInvMi_{i+1} \\
\quad \mathbf{end\ do}; \\
\quad mTemp := mTemp \bmod M\,; \\
\quad \mathbf{return}\ mTemp \\
\mathbf{end\ proc}
\end{array}
$$

```
> opRNS := proc(vetorT, vetorU, vetorBase,op) #op1=+, op2=-, op3=*
> local i, rRNS;
> Function to perfor RNS operations +, - and *.
> if ((op=1 )or(op=2))then
> if it is a sum or subtraction, ititialize with zero
> for i from 0 to k-1
> do
> rRNS[i] := 0;
> od;
> end if;
> if (op=3 )then
> if it is a mutiplication, initialize with one's
> for i from 0 to k-1
> do
> rRNS[i] := 1;
> od;
> end if;
> for i from 0 to k-1
> do
> if (op=1 )then
> rRNS[i] := (vetorT[i+1] + vetorU[i+1]) mod vetorBase[i+1];
> else if (op=2) then
> rRNS[i] := (vetorT[i+1] - vetorU[i+1]) mod vetorBase[i+1];
> else if (op=3) then
> rRNS[i] := (vetorT[i+1] * vetorU[i+1]) mod vetorBase[i+1];
> end if;
> end if;end if;  od;
> op = 1 => addition +
> op = 2 => subtraction - op = 3 => multiplication *
> return [seq(rRNS[i],i=0..k-1)];
> end;
```

$opRNS := \mathbf{proc}(vetorT,\ vetorU,\ vetorBase,\ op)$
local $i,\ rRNS$;
if $op = 1$ **or** $op = 2$ **then for** i **from** 0 **to** $k-1$ **do** $rRNS_i := 0$ **end do end if**;
if $op = 3$ **then for** i **from** 0 **to** $k-1$ **do** $rRNS_i := 1$ **end do end if**;
for i **from** 0 **to** $k-1$ **do**
if $op = 1$ **then** $rRNS_i := (vetorT_{i+1} + vetorU_{i+1}) \bmod vetorBase_{i+1}$
else
if $op = 2$ **then** $rRNS_i := (vetorT_{i+1} - vetorU_{i+1}) \bmod vetorBase_{i+1}$
else
if $op = 3$ **then** $rRNS_i := vetorT_{i+1} * vetorU_{i+1} \bmod vetorBase_{i+1}$ **end if**
end if
end if
end do;
return $[\mathrm{seq}(rRNS_i,\ i = 0..k-1)]$
end proc

```
> geraImM1B2 := proc (vetorBase2, X1)
> Computes the modular inverse of M1 concerning M2 – needed to calculate the Montgomery Modular Multiplication
> local i, invM1B2;
> for i from 1 to k
> do
> invM1B2[i] := X1^(-1) mod vetorBase2[i];
> od;
> return [seq(invM1B2[i],i=1..k)];
> end;
```

$geraImM1B2 := \mathbf{proc}(vetorBase2,\ X1)$
local $i,\ invM1B2$;
for i **to** k **do** $invM1B2_i := 1/X1 \bmod vetorBase2_i$ **end do**;
return $[\mathrm{seq}(invM1B2_i,\ i = 1..k)]$
end proc

```
> geraImMRS := proc (vetorM1)
> Modular inverses between Mi[i] and Mi[j] – needed to perform the MRS base extension
> local i, j, l, invM12;
> l := 1;
> for i from 1 to k
> do
> for j from 1 to k
> do
> if (j > i) then
> invM12[l] := vetorM1[i]^(-1) mod vetorM1[j];
> l := l + 1;
> end if;
> od;
> od;
> return [seq(invM12[i],i=1..l-1)];
> This function returs an array with the modular inverses
> end;
```

geraImMRS := **proc**(*vetorM1*)
local *i*, *j*, *l*, *invM12*;
l := 1 ;
for *i* **to** *k* **dofor** *j* **to** *k* **do**
if *i* < *j* **then** $invM12_l := 1/vetorM1_i \bmod vetorM1_j$; *l* := *l* + 1 **end if**
end do
end do;
return [seq($invM12_i$, *i* = 1..*l* − 1)]
end proc

```
> baseExtMRS := proc(vetInvM12, vetM1, vetM2, vetQ)
```

> Procedure to compute the Base extention from Beta1 to Beta2 without re-compute the cencerned number. In this version I performed a loop unrolling

```
> local i, j, alpha, extQ;
> alpha[1] := (vetQ[1]);
> alpha[2] := (vetQ[2] - alpha[1])*vetInvM12[1] mod vetM1[2];
> alpha[3] := ((vetQ[3] - alpha[1])*vetInvM12[2] -
> alpha[2])*vetInvM12[8]  mod vetM1[3];
> alpha[4] := (((vetQ[4] - alpha[1])*vetInvM12[3] -
> alpha[2])*vetInvM12[9]  - alpha[3])*vetInvM12[14] mod vetM1[4];
> alpha[5] := (((((vetQ[5] - alpha[1])*vetInvM12[4] -
> alpha[2])*vetInvM12[10] - alpha[3])*vetInvM12[15] -
> alpha[4])*vetInvM12[19]) mod vetM1[5];
> alpha[6] := (((((vetQ[6] - alpha[1])*vetInvM12[5] -
> alpha[2])*vetInvM12[11] - alpha[3])*vetInvM12[16] -
> alpha[4])*vetInvM12[20] - alpha[5])*vetInvM12[23] mod vetM1[6];
> alpha[7] := ((((((vetQ[7] - alpha[1])*vetInvM12[6] -
> alpha[2])*vetInvM12[12] - alpha[3])
> *vetInvM12[17] - alpha[4])*vetInvM12[21]- alpha[5])*vetInvM12[24] -
> alpha[6])*vetInvM12[26] mod vetM1[7];
> alpha[8] := (((((((vetQ[8] - alpha[1])*vetInvM12[7] -
> alpha[2])*vetInvM12[13] - alpha[3])*vetInvM12[18] -
> alpha[4])*vetInvM12[22] - alpha[5])*vetInvM12[25]-
> alpha[6])*vetInvM12[27] - alpha[7])*vetInvM12[28] mod vetM1[8];
> for i from 1 to 8 do
> extQ[i]:= (alpha[1] + vetM1[1]*(alpha[2] + vetM1[2]*(alpha[3] +
> vetM1[3]*(alpha[4] + vetM1[4]*(alpha[5] + vetM1[5]*(alpha[6] +
> vetM1[6]*(alpha[7] + vetM1[7]*(alpha[8])))))))) mod vetM2[i];
> od;
> return [seq(extQ[i],i=1..8)];
> end;
```

$baseExtMRS := \mathbf{proc}(vetInvM12,\ vetM1,\ vetM2,\ vetQ)$
local $i,\ j,\ \alpha,\ extQ;$
$\alpha_1 := vetQ_1\,;$
$\alpha_2 := (vetQ_2 - \alpha_1) * vetInvM12_1 \bmod vetM1_2\,;$
$\alpha_3 := ((vetQ_3 - \alpha_1) * vetInvM12_2 - \alpha_2) * vetInvM12_8 \bmod vetM1_3\,;$
$\alpha_4 :=$
$(((vetQ_4 - \alpha_1) * vetInvM12_3 - \alpha_2) * vetInvM12_9 - \alpha_3) * vetInvM12_{14}$
$\bmod vetM1_4;$
$\alpha_5 := ($
$(((vetQ_5 - \alpha_1) * vetInvM12_4 - \alpha_2) * vetInvM12_{10} - \alpha_3) * vetInvM12_{15}$
$- \alpha_4) * vetInvM12_{19} \bmod vetM1_5;$
$\alpha_6 := ($(
$(((vetQ_6 - \alpha_1) * vetInvM12_5 - \alpha_2) * vetInvM12_{11} - \alpha_3) * vetInvM12_{16}$
$- \alpha_4) * vetInvM12_{20} - \alpha_5) * vetInvM12_{23} \bmod vetM1_6;$
$\alpha_7 := ($((
$(((vetQ_7 - \alpha_1) * vetInvM12_6 - \alpha_2) * vetInvM12_{12} - \alpha_3) * vetInvM12_{17}$
$- \alpha_4) * vetInvM12_{21} - \alpha_5) * vetInvM12_{24} - \alpha_6) * vetInvM12_{26} \bmod$
$vetM1_7;$
$\alpha_8 := ($(((
$(((vetQ_8 - \alpha_1) * vetInvM12_7 - \alpha_2) * vetInvM12_{13} - \alpha_3) * vetInvM12_{18}$
$- \alpha_4) * vetInvM12_{22} - \alpha_5) * vetInvM12_{25} - \alpha_6) * vetInvM12_{27} - \alpha_7) *$
$vetInvM12_{28} \bmod vetM1_8;$
for i **to** 8 **do** $extQ_i := (\alpha_1 + vetM1_1 * (\alpha_2 + vetM1_2 * (\alpha_3 + vetM1_3 *$
$(\alpha_4 + vetM1_4 * (\alpha_5 + vetM1_5 * (\alpha_6 + vetM1_6 * (\alpha_7 + vetM1_7 * \alpha_8))))$
$)))\bmod vetM2_i$
end do;
return $[\mathrm{seq}(extQ_i,\ i = 1..8)]$
end proc

```
>  MM := proc(A1, A2, B1, B2, Beta1, Beta2, N1, N2, iN1, iM12,
>  vetInvBeta1,vetInvBeta2)
> Montgomery's algorithm to comute modular multiplication in RNS with base extention
>  local i, R2, R13, R23, R33, T1, T2, Q1, Q2, R1 : aray(1..tamanho);
> Compute T = A Xrns B in Beta1 and Beta2
>  T1 := opRNS(A1,B1,Beta1,3);
>  T2 := opRNS(A2,B2,Beta2,3);
> Compute Q = T x -N^-1 in Beta1
>  Q1 := opRNS(T1,iN1,Beta1,3);
> Extend Q from Beta to Beta2
>  Q2 := baseExtMRS(vetInvBeta1, Beta1, Beta2, Q1);
> Compute R = (T +rns Q Xrns N)Xrns M1^-1 in Beta2
> Partial result : R13 = Q2 Xrns N2
>  R13 := opRNS(Q2,N2,Beta2,3);
> Partial result : R23 = (Q2 Xrns N2) +rns T2
>  R23 := opRNS(T2,R13,Beta2,1);
> Partial result : R13 = ((Q2 Xrns N2) +rns T2) Xrns M1^-1
>  R33 := opRNS(R23,iM12,Beta2,3);
> Final result : R2 = (T2 +rns Q2 Xrns N2)Xrns M12^-1
>  R2 := R33;
> Final base extention, from Beta2 to Beta1
>  R1 := baseExtMRS(vetInvBeta2, Beta2, Beta1, R2);
>  return R1;
>  end; #MM
```

MM := **proc**(*A1*, *A2*, *B1*, *B2*, *B*1, *B*2, *N1*, *N2*, *iN1*, *iM12*, *vetInvBeta1*, *vetInvBeta2*)
local *i*, *R2*, *R13*, *R23*, *R33*, *T1*, *T2*, *Q1*, *Q2*, *R1*;
aray(1..*tamanho*);
T1 := opRNS(*A1*, *B1*, *B*1, 3);
T2 := opRNS(*A2*, *B2*, *B*2, 3);
Q1 := opRNS(*T1*, *iN1*, *B*1, 3);
Q2 := baseExtMRS(*vetInvBeta1*, *B*1, *B*2, *Q1*);
R13 := opRNS(*Q2*, *N2*, *B*2, 3);
R23 := opRNS(*T2*, *R13*, *B*2, 1);
R33 := opRNS(*R23*, *iM12*, *B*2, 3);
R2 := *R33*;
R1 := baseExtMRS(*vetInvBeta2*, *B*2, *B*1, *R2*);
return *R1*
end proc

-Tests-

Global variables

number of basis
```
> k    := 8;
```

Initial base

```
> inicio1 := 128;
```

Numbers will be mutiplicated
```
> X    := 225;
```

```
> Y    := 28;
```

Do not forget : X (+,-,*)Y must be less than M

Initial basis
```
> Bases := geraBase(k*2,inicio1);
```

Beta1
```
> mBase1 := [seq(Bases[i],i=1..k)];
```

Beta2
```
> mBase2 := [seq(Bases[i],i=k+1..2*k)];
```

M1 = product of al mi in Beta1
```
> M1 := geraM(mBase1);
```

M2 = product of al mi in Beta2
```
> M2 := geraM(mBase2);
```

$$k := 8$$
$$inicio1 := 128$$
$$X := 225$$
$$Y := 28$$
$$Bases := [128, 129, 131, 133, 137, 139, 143, 145, 149, 151, 157, 163, 167, 173, 179, 181]$$
$$mBase1 := [128, 129, 131, 133, 137, 139, 143, 145]$$
$$mBase2 := [149, 151, 157, 163, 167, 173, 179, 181]$$
$$M1 := 113595734416644480$$
$$M2 := 538945254996352681$$

Convertion of the numbers that will be multiplicated from Decimal to RNS in both Beta1 and Beta2
```
> xRNS1 := convDECtoRNS(X,mBase1);
```

```
> yRNS1 := convDECtoRNS(Y,mBase1);
```

```
> xRNS2 := convDECtoRNS(X,mBase2);
```

```
> yRNS2 := convDECtoRNS(Y,mBase2);
```

$$xRNS1 := [97, 96, 94, 92, 88, 86, 82, 80]$$
$$yRNS1 := [28, 28, 28, 28, 28, 28, 28, 28]$$
$$xRNS2 := [76, 74, 68, 62, 58, 52, 46, 44]$$
$$yRNS2 := [28, 28, 28, 28, 28, 28, 28, 28]$$

Addition test in RNS, conversion from RNS to decimal

```
> testADDrns := opRNS(xRNS1,yRNS1,mBase1,1);

> testADDcomputed := convRNStoDEC(totoRNS,vetMi1,vetInvMi1,M1);

> testADDexpected := X + Y;
```

$$testADDrns := [125, 124, 122, 120, 116, 114, 110, 108]$$

$$\begin{aligned} testADDcomputed := {} & totoRNS_1\, vetMi1_1\, vetInvMi1_1 + totoRNS_2\, vetMi1_2\, vetInvMi1_2 \\ & + totoRNS_3\, vetMi1_3\, vetInvMi1_3 + totoRNS_4\, vetMi1_4\, vetInvMi1_4 \\ & + totoRNS_5\, vetMi1_5\, vetInvMi1_5 + totoRNS_6\, vetMi1_6\, vetInvMi1_6 \\ & + totoRNS_7\, vetMi1_7\, vetInvMi1_7 + totoRNS_8\, vetMi1_8\, vetInvMi1_8 \end{aligned}$$

$$testADDexpected := 253$$

Compute the Mi = M/mi in Beta1 and Beta2

```
> vetMi1 := geraMi(mBase1,M1);

> vetMi2 := geraMi(mBase2,M2);
```

Compute the modular inverses of Mi in both basis Beta1 and Beta2

```
> vetInvMi1 := geraIm(mBase1,vetMi1);

> vetInvMi2 := geraIm(mBase2,vetMi2);
```

$$vetMi1 := [887466675130035, 880587088501120, 867143010814080, 854103266290560, 829165944647040, 817235499400320, 794375765151360, 783418858045824]$$

$$vetMi2 := [3617082248297669, 3569173874148031, 3432772324817533, 3306412607339587, 3227217095786543, 3115290491308397, 3010867346348339, 2977598093902501]$$

$$vetInvMi1 := [123, 22, 115, 6, 36, 42, 85, 114]$$

$$vetInvMi2 := [90, 101, 135, 110, 102, 102, 139, 39]$$

Compute the modular Inverses needed to the base extensions in MRS

```
> vetInvBase1 := geraImMRS(mBase1);

> vetInvBase2 := geraImMRS(mBase2);
```

$$vetInvBase1 := [128, 87, 53, 76, 101, 19, 17, 65, 33, 17, 125, 51, 9, 66, 114, 52, 131, 31, 34, 23, 100, 12, 69, 119, 18, 107, 24, 72]$$

$$vetInvBase2 := [75, 98, 128, 102, 36, 173, 164, 26, 95, 73, 55, 147, 6, 27, 50, 54, 122, 98, 125, 121, 123, 10, 144, 164, 168, 149, 113, 90]$$

Tests the multiplication in RNS and the convertion to decimal from both basis

```
> X, xRNS1, xRNS2;

> Y, yRNS1, yRNS2;

> sRNS1 := opRNS(xRNS1,yRNS1,mBase1,3);

> sRNS2 := opRNS(xRNS2,yRNS2,mBase2,3);

> sExpected := X * Y;

> sFinal1 := convRNStoDEC(sRNS1,vetMi1,vetInvMi1,M1);

> sFinal2 := convRNStoDEC(sRNS2,vetMi2,vetInvMi2,M2);
```

$$225, [97, 96, 94, 92, 88, 86, 82, 80], [76, 74, 68, 62, 58, 52, 46, 44]$$
$$28, [28, 28, 28, 28, 28, 28, 28, 28], [28, 28, 28, 28, 28, 28, 28, 28]$$
$$sRNS1 := [28, 108, 12, 49, 135, 45, 8, 65]$$
$$sRNS2 := [42, 109, 20, 106, 121, 72, 35, 146]$$
$$sExpected := 6300$$
$$sFinal1 := 6300$$
$$sFinal2 := 6300$$

Test the base extention from Beta1 to Beta2

```
> testMRS := baseExtMRS(vetInvBase1, mBase1, mBase2, xRNS1);
```

$$testMRS := [76, 74, 68, 62, 58, 52, 46, 44]$$

Initialization of the values needed to perform a montgomery multiplication

```
> xRNS1,xRNS2, yRNS1,yRNS2;M1, M2;
```

4xN must be less than M

```
> N:= nextprime(20726);

> A := X;

> B := Y;
```

Modular inverse of N regarding M1

```
> iMN := (-N^(-1))mod M1;
```

Modular inverse of M1 regarding M2

```
> invM1B2 := M1^(-1) mod M2;
```

$$[97, 96, 94, 92, 88, 86, 82, 80], [76, 74, 68, 62, 58, 52, 46, 44],$$
$$[28, 28, 28, 28, 28, 28, 28, 28], [28, 28, 28, 28, 28, 28, 28, 28]$$
$$113595734416644480, 538945254996352681$$
$$N := 20731$$
$$A := 225$$
$$B := 28$$

$$iMN := 95190058284016589$$
$$invM1B2 := 296094129031463721$$

Basis generation.
```
> Bases;
```

Choising Beta1 and Beta2
```
> mBase1, mBase2;
```

Inversa modular de M1 em relaçao à M2, representada na base Beta2
```
> iMm1b2RNSb2 := convDECtoRNS(invM1B2,mBase2);
```

A e B no sistema RNS base Beta1
```
> aRNSb1 := convDECtoRNS(A,mBase1);

> bRNSb1 := convDECtoRNS(B,mBase1);
```

A e B no sistema RNS base Beta2
```
> aRNSb2 := convDECtoRNS(A,mBase2);

> bRNSb2 := convDECtoRNS(B,mBase2);
```

A X B < 2N para ambas as bases
```
> nRNSb1 := convDECtoRNS(N,mBase1);

> nRNSb2 := convDECtoRNS(N,mBase2);

> iMnRNSb1 := convDECtoRNS(iMN,mBase1);
```

0 <4N<M1,M2
```
> invM1 := vetInvMi1;
```

[128, 129, 131, 133, 137, 139, 143, 145, 149, 151, 157, 163, 167, 173, 179, 181]
[128, 129, 131, 133, 137, 139, 143, 145], [149, 151, 157, 163, 167, 173, 179, 181]
$$iMm1b2RNSb2 := [24, 124, 13, 16, 104, 129, 112, 12]$$
$$aRNSb1 := [97, 96, 94, 92, 88, 86, 82, 80]$$
$$bRNSb1 := [28, 28, 28, 28, 28, 28, 28, 28]$$
$$aRNSb2 := [76, 74, 68, 62, 58, 52, 46, 44]$$
$$bRNSb2 := [28, 28, 28, 28, 28, 28, 28, 28]$$
$$nRNSb1 := [123, 91, 33, 116, 44, 20, 139, 141]$$
$$nRNSb2 := [20, 44, 7, 30, 23, 144, 146, 97]$$
$$iMnRNSb1 := [77, 17, 127, 47, 28, 132, 36, 109]$$
$$invM1 := [123, 22, 115, 6, 36, 42, 85, 114]$$

End of tests and pre-computationss

Begin of the Montgomery algorithm. The returned value is A x B x M1^-1 mod N in Beta1

```
> R1 := MM
> (aRNSb1,aRNSb2,bRNSb1,bRNSb2,mBase1,mBase2,nRNSb1,nRNSb2,iMnRNSb1,iMm1
> b2RNSb2,vetInvBase1,vetInvBase2);
```

$$R1 := [43, 7, 66, 129, 130, 64, 75, 11]$$

To confirm the calcul :

```
> A; B; N;
> rComputed := convRNStoDEC(R1,vetMi1,vetInvMi1,M1);
> rExpected := (A * B * (M1^(-1))) mod N;
```

$$225$$
$$28$$
$$20731$$
$$rComputed := 4651$$
$$rExpected := 4651$$

But the real value expected is A x B mod N. So, a post computation is needed to remove the M1^-1 value. This is performed by post mutiplicating the result by M1^2 mod N.

value needed to remove the M1^-1factor

```
> mFinal := M1^2 mod N;
```

We need this value in both basis

```
> mFin1 := convDECtoRNS(mFinal,mBase1);
> mFin2 := convDECtoRNS(mFinal,mBase2);
```

We need also the result of the first montgomery multiplication in Beta2

```
> R2 := baseExtMRS(vetInvBase1,mBase1,mBase2, R1);
```

$$mFinal := 16948$$
$$mFin1 := [52, 49, 49, 57, 97, 129, 74, 128]$$
$$mFin2 := [111, 36, 149, 159, 81, 167, 122, 115]$$
$$R2 := [32, 121, 98, 87, 142, 153, 176, 126]$$

Then the final result is obtained as follows : resultFinal = rComputed x mFinal mod N ;

```
> resultFinal :=
> MM(R1,R2,mFin1,mFin2,mBase1,mBase2,nRNSb1,nRNSb2,iMnRNSb1,iMm1b2RNSb2,
> vetInvBase1,vetInvBase2);
```

$$resultFinal := [28, 108, 12, 49, 135, 45, 8, 65]$$

To confirm, we compare de corresponding decimal values

```
> finalResultComputedDec :=
```

```
> convRNStoDEC(resultFinal,vetMi1,vetInvMi1,M1);
> finalExpected:= A * B mod N;
```

$$finalResultComputedDec := 6300$$
$$finalExpected := 6300$$

ANNEXE F

Current Mask Generator

Current Mask Generator - Schéma Détaillé

TITRE (en français) :

Architectures Reconfigurables et Cryptographie : Une Analyse de Robustesse et Contremesures Face aux Attaques par Canaux Cachés

RÉSUMÉ :

Ce travail constitue une étude sur la conception d'une architecture reconfigurable pour la cryptographie. Divers aspects sont étudiés, tels que les principes de base de la cryptographie, l'arithmétique modulaire, les attaques matériaux et les architectures reconfigurables. Des méthodes originales pour contrecarrer les attaques par canaux cachés, notamment la DPA, sont proposés. L'architecture proposée est efficace du point de vue de la performance et surtout est robuste contre la DPA.

TITRE (en anglais) :

Reconfigurable Architectures and Cryptography : A Robustness Analysis and Countermeasures Against Side Channel Attacks

RÉSUMÉ (en anglais) :

This work addresses the reconfigurable architectures for cryptographic applications theme, emphasizing the robustness issue. Some mathematical background is reviewed, as well the state of art of reconfigurable architectures. Side channel attacks, specially the DPA and SPA attacks, are studied. As consequence, algorithmic, hardware and architectural countermeasures are proposed. A new parallel reconfigurable architecture is proposed to implement the Leak Resistant Arithmetic. This new architecture outperforms most of state of art circuits for modular exponentiation, but the main feature of this architecture is the robustness against DPA attacks.

MOTS-CLÉS :

Architectures Reconfigurables, Cryptographie, Attaques par Canaux Cachés et Contremesures.

DISCIPLINE :

Systèmes Automatiques et Microélectroniques.

INTITULE ET ADRESSE DE L'U.F.R. OU DU LABORATOIRE :

Laboratoire d'Informatique, de Robotique et de Micro-électronique de Montpellier (LIRMM)

UMR CNRS/Université Montpellier II, no. C55060

161, rue ADA. 34392 - Montpellier - Cedex 5 - France