TABLE OF CONTENTS

TABLE DES FIGURES

LISTE DES TABLEAUX

1

Introduction

Depuis une décennie, nous assistons à la multiplication dans nos vies des objets connectés mus par des processeurs. Ainsi, au-delà des exigences sur la consommation énergétique et du coût de ces dispositifs, de nouvelles contraintes de sécurité émergent avec ces usages. Ces composants peuvent être amenés à stocker, exécuter, collecter des données sensibles et les cas où des acteurs malveillants ont accès physiquement aux dispositifs sont de plus en plus probables. Lorsque l'on parle de sécurité, les principaux objectifs à respecter sont les suivants :

- Confidentialité : Le processeur doit maintenir les instructions et les données utilisées secrètes.
- Intégrité : Les instructions, les données manipulées ou l'état du processeur ne doivent pas avoir subi de modifications entre le stockage et leur manipulation, ces modifications peuvent être fortuites, illicites ou malveillantes. En clair, les éléments considérés doivent être exacts et sans altérations.
- Authenticité : Le processeur doit pouvoir s'assurer de l'origine des données. Le niveau de granularité de l'origine peut varier en fonction des cas d'applications. L'authenticité est souvent assurée par une signature sur les données qui permet conjointement de garantir leur intégrité.
- Disponibilité : le processeur doit garantir son fonctionnement à tout moment.

Au-delà de ces notions de sécurité, il est essentiel de les articuler convenablement et en particulier d'envisager les conséquences de pertes d'intégrité, de confidentialité ou d'authenticité sur la disponibilité. Le processeur doit pouvoir se rétablir après ces perturbations

et retrouver un fonctionnement nominale tout en ayant réparé ses fautes. On perle alors de résilience. Celle-ci passe par trois stades : la protection, la détection et le rétablissement (ref : https ://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-193.pdf).

Les travaux réalisés durant cette thèse visent à assurer ces quatre propriétés face aux attaques par canaux auxiliaires et par injection de fautes dans le cadre d'un pipeline de l'unité centrale de traitement (CPU).

1.1 Cible de la thèse : Architecture des processeurs

La microarchitecture d'un CPU est généralement séparée en chemin de données qui s'occupe de les charger dans les registres et effectuer les opérations. Et le chemin de contrôle qui charge, décode les instructions et génère les signaux de contrôle.

Des machines avec la même ISA et donc capables d'exécuter le même code compilé, peuvent avoir des micro-architectures complètement différentes. Les nouvelles micro-architectures et/ou optimisations, ainsi que les progrès réalisés dans la fabrication des semi-conducteurs permettent aux nouvelles générations de processeurs d'atteindre des performances plus élevées tout en utilisant la même ISA. Nous allons présenter dans un premier temps les principes fondamentaux de l'architecture des processeurs puis présenter les différentes optimisations architecturales les plus répandues pour augmenter les performances.

1.1.1 Fonctionnement général des processeurs

Un processeur consiste à exécuter un programme compilé selon un certain jeu d'instructions. Le programme compilé se situe dans la mémoire principale. Ainsi le processeur doit avoir un accès et pouvoir faire des requêtes a cette mémoire principale. Une fois l'instruction chargée celui-ci doit exécuter les instructions qu'il a appelées de la mémoire.

1.1.2 Principales fonctions réalisées

Le processeur de chaque ordinateur peut avoir des cycles différents basés sur des jeux d'instructions différents, mais les fonctions réalisées seront assimilables aux étapes suivantes :

- Phase de chargement de l'instruction : L'instruction est extraite de l'adresse mémoire qui est actuellement stockée dans le compteur de programme (PC). À la fin de l'opération d'extraction, le PC pointe vers l'instruction suivante qui sera lue au cycle suivant.
- Phase de décodage : Au cours de cette étape, l'instruction est chargée depuis la mémoire principale, puis interprétée par le décodeur pour savoir quelle est l'opération à réaliser et les données à utiliser pour cette instruction.
- Phase d'exécution : L'unité de décodage transmet les informations décodées sous la forme d'une séquence de signaux de commande aux unités fonctionnelles pertinentes pour qu'elles effectuent les actions requises par l'instruction, comme la lecture de valeurs dans les registres, leur transmission à l'unité arithmétique et logique (ALU) et la réécriture du résultat dans un registre. Le résultat généré par l'opération peut aussi être stocké dans la mémoire principale ou envoyé à un dispositif de sortie. En fonction du retour de l'ALU, le PC peut être mis à jour à une adresse différente à partir de laquelle l'instruction suivante sera extraite.
- Répétition des étapes précédentes

1.1.3 Caches

Un cache de processeur est une mémoire matérielle utilisée par le CPU d'un ordinateur pour réduire le coût moyen (temps et énergie) d'accès aux données de la mémoire principale. L'unité centrale comprend un contrôleur de cache qui automatise la lecture et l'écriture dans le cache. Si les données se trouvent déjà dans le cache, elles sont accessibles à partir de celui-ci, ce qui permet un gain de temps important, alors que si elles n'y sont pas, le processeur est « bloqué » pendant que le contrôleur de cache les lit. La plupart des processeurs possèdent une hiérarchie de plusieurs niveaux de cache (L1, L2, souvent L3, et même rarement L4), avec des caches spécifiques aux instructions et aux données au niveau 1.

1.1.4 Interruptions

En outre, sur la plupart des processeurs, des interruptions dues aux processus utilisateur, aux périphériques ou au système d'exploitation, peuvent se produire. Dans ce cas, le processeur saute à une routine d'interruption, l'exécute et revient. Dans certains cas,

une instruction peut être interrompue au milieu, l'instruction n'aura aucun effet, mais sera réexécutée après le retour de l'interruption.

1.1.5 Optimisation

Cette série d'étapes d'apparence simple est compliquée par le fait que la hiérarchie de la mémoire, qui comprend la mémoire cache, la mémoire principale et le stockage non volatile comme les disques durs (où résident les instructions du programme et les données), a toujours été plus lente que le processeur lui-même. Différentes solutions ont été apportées tout d'abord sur les superordinateurs jusque de nos jours dans des contrôleurs embarques les principales innovations seront ici présentées.

Pipeline

La première optimisions était de découper le processeur en plusieurs étages pour obtenir du parallélisme au niveau des instructions. Le pipeline tente de faire en sorte que chaque partie du processeur ainsi découpé soit occupée par une instruction différente et ainsi traiter plusieurs instructions en parallèle. Dans le cas le plus classique d'une architecture d'ordinateur à jeu d'instructions réduit (RISC) il y a 5 étages :

- La récupération des instructions
- Décodage des instructions et récupération des registres
- Exécution
- Accès à la mémoire
- Réécriture du registre

Cependant malgré le gain en performance apporté, cela apporte de nouveaux problèmes dont notamment des risques structurels se produisent lorsque deux instructions peuvent tenter d'utiliser les mêmes ressources au même moment. Mais aussi des risques liés aux données se produisent lorsqu'une instruction tente d'utiliser des données avant que celles-ci ne soient disponibles dans le fichier de registre. Et enfin des risques de contrôle causés par les branchements conditionnels et inconditionnels.

Prédiction de branchement

L'un des obstacles à l'obtention de performances plus élevées grâce au parallélisme au niveau des instructions provient des blocages et des vidages du pipeline dus aux branchements. Normalement, on ne sait pas si un branchement conditionnel sera effectué avant la fin du pipeline, car les branchements conditionnels dépendent des résultats provenant d'un registre. Entre le moment où le décodeur d'instructions du processeur décode une instruction de branchement conditionnel et le moment où il est possible de savoir si le branchement est pris ou non, le pipeline doit être bloqué pendant plusieurs cycles, ou s'il ne l'est pas et que le branchement est effectué, le pipeline doit être purgé. Plus la fréquence d'horloge augmente, plus la profondeur du pipeline augmente, et certains processeurs modernes peuvent avoir 20 étages ou plus. En moyenne, une instruction exécutée sur cinq est un branchement, ce qui, sans aucune intervention, représente une quantité élevée de blocage.

Des techniques telles que la prédiction de branchement et l'exécution spéculative sont utilisées pour atténuer ces pénalités de branchement. La prédiction de branchement consiste pour le matériel à faire des suppositions sur l'opportunité d'un branchement particulier. En réalité, l'un ou l'autre côté de la branche sera appelé beaucoup plus souvent que l'autre. Les conceptions modernes ont des systèmes de prédiction statistique assez complexes, qui observent les résultats des branches passées pour prédire l'avenir avec une plus grande précision. Cette prédiction permet au matériel de préempter des instructions sans attendre la lecture du registre. L'exécution spéculative est une autre amélioration dans laquelle le code le long du chemin prédit n'est pas seulement préextrait, mais également exécutée avant que l'on sache si la branche doit être prise ou non.

Superscalaire

Même avec toute la complexité ajoutée et les portes nécessaires pour supporter les concepts décrits ci-dessus, les améliorations dans la fabrication de semi-conducteurs ont bientôt permis d'utiliser encore plus de portes logiques.

Le processeur traite des parties d'une seule instruction à la fois. Les programmes informatiques pourraient être exécutés plus rapidement si plusieurs instructions étaient traitées simultanément. C'est ce que réalisent les processeurs superscalaires, en répliquant des unités fonctionnelles telles que les ALU.

Dans les conceptions modernes, il est courant de trouver deux unités de chargement, une

unité de stockage (de nombreuses instructions n'ont pas de résultats à stocker), deux unités mathématiques entières ou plus, deux unités à virgule flottante ou plus, et souvent une unité SIMD. La logique d'émission des instructions devient de plus en plus complexe en lisant une énorme liste d'instructions dans la mémoire et en les transmettant aux différentes unités d'exécution qui sont inactives à ce moment-là. Les résultats sont ensuite collectés et réorganisés à la fin.

Exécution dans le désordre

L'ajout de caches réduit la fréquence ou la durée des blocages dus à l'attente de données à extraire de la hiérarchie de la mémoire, mais n'élimine pas complètement ces blocages. Dans les premières conceptions, un manque de cache obligeait le contrôleur de cache à bloquer le processeur et à attendre. Bien sûr, il peut y avoir une autre instruction dans le programme dont les données sont disponibles dans le cache à ce moment-là. L'exécution hors ordre permet à cette instruction près d'être traitée pendant qu'une instruction plus ancienne attend dans le cache, puis réorganise les résultats pour faire croire que tout s'est passé dans l'ordre programmé. Cette technique est également utilisée pour éviter d'autres blocages liés à la dépendance des opérandes, comme une instruction qui attend le résultat d'une opération en virgule flottante à longue latence ou d'autres opérations à cycles multiples.

Renommage de registre

Le renommage de registre fait référence à une technique utilisée pour éviter l'exécution sérielle inutile d'instructions de programme en raison de la réutilisation des mêmes registres par ces instructions. Supposons que nous ayons deux groupes d'instructions qui vont utiliser le même registre. Un groupe d'instructions est exécuté en premier pour laisser le registre à l'autre groupe, mais si l'autre groupe est affecté à un autre registre similaire, les deux groupes d'instructions peuvent être exécutés.

Multiprocessing et multithreading

Les architectes d'ordinateurs se sont retrouvés bloqués par le décalage croissant entre les fréquences de fonctionnement des CPU et les temps d'accès aux DRAM. Aucune des techniques qui exploitaient le parallélisme au niveau des instructions (ILP) dans un programme ne pouvait compenser les longs délais qui se produisaient lorsque les données

devaient être extraites de la mémoire principale. Pour ces raisons, les nouvelles générations d'ordinateurs ont commencé à exploiter des niveaux plus élevés de parallélisme qui existent en dehors d'un seul programme ou d'un seul fil de programme.

L'une des techniques permettant d'obtenir ce parallélisme est celle des systèmes multiprocesseurs, c'est-à-dire des systèmes informatiques dotés de plusieurs unités centrales. Conceptuellement, le multithreading est équivalent à un changement de contexte au niveau du système d'exploitation. La différence réside dans le fait qu'une unité centrale multithread peut effectuer un changement de thread en un seul cycle d'unité centrale au lieu des centaines ou milliers de cycles d'unité centrale qu'un changement de contexte nécessite normalement. Ceci est réalisé en répliquant le matériel d'état (tel que le fichier de registre et le compteur de programme) pour chaque thread actif.

1.2 Description du RISCV et de l'architecture CV32e40p

Contrairement à la plupart des autres conceptions ISA, RISC-V est fourni sous des licences open source dont l'utilisation est gratuite. En tant qu'architecture RISC, l'ISA du RISC-V est une architecture load-store.

Le RISC-V a été lancé dans le but de créer une ISA pratique, en libre accès, utilisable dans le monde universitaire et pouvant être déployé dans n'importe quelle conception matérielle ou logicielle sans redevance. De plus, les justifications de chaque décision de conception du projet sont expliquées, au moins en termes généraux. Les auteurs du RISC-V sont

rajouter des références ou mettre des noms des créateurs

des universitaires qui ont une grande expérience de la conception d'ordinateurs, et le RISC-V ISA est un développement direct d'une série de projets universitaires de conception d'ordinateurs.

L'affirmation principale des concepteurs est que le jeu d'instructions est l'interface clé d'un ordinateur, car il se situe à l'interface entre le matériel et le logiciel. Si un bon jeu d'instructions était ouvert et disponible pour tous, il pourrait réduire considérablement le coût des logiciels en permettant une réutilisation beaucoup plus importante. Cela devrait également déclencher une concurrence accrue entre les fournisseurs de matériel, qui pourraient alors consacrer plus de ressources à la conception et moins au support logiciel. L'ISA à longueur variable permet d'étendre le jeu d'instructions, et le jeu d'instructions privilégiées séparé

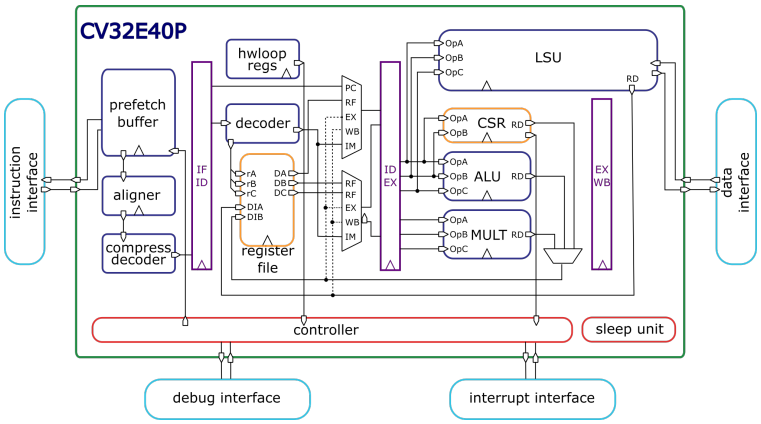

Figure 1.1 : Microarchitecture du CPU CV32E40P.

permet de faire des recherches sur le support du système d'exploitation sans avoir à reconcevoir les compilateurs. Le paradigme de propriété intellectuelle ouvert du RISC-V permet de publier, réutiliser et modifier les conceptions dérivées.

1.3 Menace et solutions proposées

Cette thèse se concentre sur les attaques par canaux auxiliaires et par injection de faute. Une attaque par canal auxiliaire est une attaque basée sur des informations qui peuvent être recueillies en raison de l'implémentions physique d'un protocole ou d'un algorithme, plutôt que sur des failles dans leurs conceptions elle-même, par exemple, des failles trouvées par cryptanalyse d'un algorithme cryptographique ou sur des erreurs dans la conception. Les informations relatives au temps, à la consommation d'énergie, aux fuites électromagnétiques, sont des exemples d'informations qui peuvent être exploitées pour des attaques par canaux auxiliaires. L'attaque par injection de fautes consiste à stresser les transistors responsables des tâches de chiffrement, afin de générer une faute qui sera ensuite utilisée. L'élément perturbateur peut être une impulsion électromagnétique (impulsion EM ou impulsion laser) ou une modification de la tension d'alimentations, de la température ou de la fréquence d'horloge. Ces deux types d'attaques peuvent être menés de concert, l'utilisation de canaux auxiliaire pour trouver des points d'intérêt d'une injection de faute. Cependant leurs contremesures classiques sont assez antinomiques. Les canaux auxiliaires nécessitent de masquer les données. Le principe du masquage est d'éviter de manipuler directement une valeur sensible, mais plutôt un partage de celle-ci : un ensemble de variables (appelées « share »). Un attaquant doit récupérer toutes les valeurs des shares pour

obtenir des informations significatives. Cette division en share offre plus de possibilités d'attaque d'un point de vue de l'injection de faute, car on peut cibler n'importe quel share. Les contremesures contre l'injection de faute, elle se base sur le principe de redondance que celle-ci soit temporelle, physique ou informationnelle. Cependant, cette redondance entraine de nouvelles fuites qui peuvent être exploitées par canaux auxiliaires.

Des méthodes mixtes ont été proposées dans l'état de l'art, notamment M&M qui utilise les propriétés spécifiques des opérations de l'AES pour créer de la redondance pouvant vérifier les calculs arithmétiques, mais étant aussi compatible avec le masquage.

1.4 Sécurité recherchée et organisation de la thèse

Les problématiques soulevées par cette thèse sont d'identifier les différentes attaques pouvant cibler le pipeline d'un processeur généraliste et ainsi les adresser en tirant parti des spécificités de la partie attaquée. La question essentielle était de savoir s'il était possible d'obtenir un résultat similaire à M&M dans le cadre d'un processeur généraliste, mais aussi d'identifier les masquages les plus aptes à être implémentés dans ce type de composant. Au-delà de la protection des données, les instructions sont une cible de la part des attaquants. Ainsi une recherche a été menée sur des mécanismes qui permettraient de protéger efficacement et a à moindre coût les instructions. Mais aussi de trouver des moyens de réponse à une attaque ainsi en assurant la sécurité et la fiabilité nous convergeons vers un design résilient aux attaques par canaux auxiliaires et injections de fautes.

Le chapitre 2 sera un état de l'art sur les principales attaques par canaux auxiliaires et les attaques par injections et leurs contremesures. Nous nous concentrerons dans le chapitre 3 au chemin de données avec les attaques et les contremesures proposées par l'état de l'art pour ensuite présenter les différentes contremesures proposées sur cette partie de l'architecture des processeurs Le chapitre 4 lui traitera le chemin d'instruction et du contrôle avec la aussi un état de l'art des attaques et des contremesures pour ensuite expliquer l'apport de cette thèse et des contremesures, développer pour sécuriser le chemin d'instruction et la logique de contrôle. Dans le chapitre 5, nous verrons le processeur dans qui est ensemble avec une contremesure impactant le fonctionnement général et empêchant les attaques au sens large, mais aussi l'interaction et l'intégration de toutes ces contremesures dans le même circuit. Le chapitre 6 est la conclusion générale de ce manuscrit et synthétise les différents travaux et leur apporte dans la sécurisation d'une architecture de processeur.

1.5 Contribution

Les travaux présentés dans le chapitre 3 font l'objet de deux brevets. Le premier sur le principe de la permutation dépendante d'une clé et l'autre sur la permutation choisie afin de facilité les opérations arithmétique et est en cours de soumission Le chapitre 4 avec la sécurisation des instructions avec la génération de masque à partir de l'instruction précédent a été publie a DSD2022 et est présente dans un brevet. L'insertion de cycle factice pour créer de la désynchronisation temporelle a aussi été brevetée et a été publiée à HOST2022.

2

État de l'art

En introduction, on a présenté les deux types d'attaques contre lesquelles nous cherchons à nous protéger. Dans un premier temps, nous étudierons les attaques par canaux auxiliaires avec leur vecteur de fuite, mais aussi les deux grands types d'exploitations. Ensuite, nous passerons rapidement sur les différents types de contremesure pour finir sur l'évaluation des circuits. Dans un second temps, nous verrons les attaques par injections de fautes. Avec dans un premier temps les différentes manières d'injecter des fautes que nous définirons notre modèle de fautes que nous utilisons pour le reste de la thèse. Ensuite, nous étudierons les différentes méthodes de tests et finirons par définir les différents types de contremesure contre les attaques par injection de faute.

2.1 Attaque canaux auxiliaires

Tout d'abord, une attaque par canaux auxiliaire est une attaque qui cherche à exploiter les failles de l'implémentation matérielle. Cela ne remet pas en cause la robustesse théorique de l'implémentation, mais elle permet de récupérer de l'information par des moyens indirects. En effet, lors du traitement des données, il faut du temps et dissiper une quantité minimum d'énergie pour changer d'un état à un autre. De ce fait, des fuites par consommation de courant, temps d'exécution, température ou rayonnement électromagnétique sont inévitables.

2.1.1 Méthode d'analyse par canaux auxiliaires

Les canaux auxiliaires sont multiples, du fait que tout calcul est le résultat de composants électroniques ceux ci produise de la chaleur, une consommation électrique, une émanation électromagnétique mais aussi un temps d'exécution qui leur est propre. Toute ces fuites difficilement contrôlé par le concepteur peuvent dépendre de la donnée manipulé et ainsi permettre a un attaquant d'en extraire de l'information.

Fuite par consommation

Les premières fuites par canaux auxiliaire observées l'ont été sur la consommation du circuit. Cette consommation se divise en deux composantes une dynamique et une statique.

La consommation dynamique (Dynamic Power Dissipation DPD) est la principale source de fuite par side channel. Elle a lieu lors d'un changement de l'état logique, elle se compose de deux composants le chargement et déchargement de la capacité de charge, l'on remarque ces changements lors de la transitions de 0-1 ou de 1-0, et le court-circuit dû à un signal d'entrée non nul [**korkikian_side-channel_2016**]

La consommation statique (Static power dissipation SPD) se compose de 6 mécanismes qui sont illustré en Figure 2.1 :

- I_1 Consommation dans la jonction p-n,
- I_2 Fuite par subthreshold
- I_3 Tunneling dans et à travers la porte oxyde
- I_4 Hot carriers injection dans le substrat de la porte oxyde

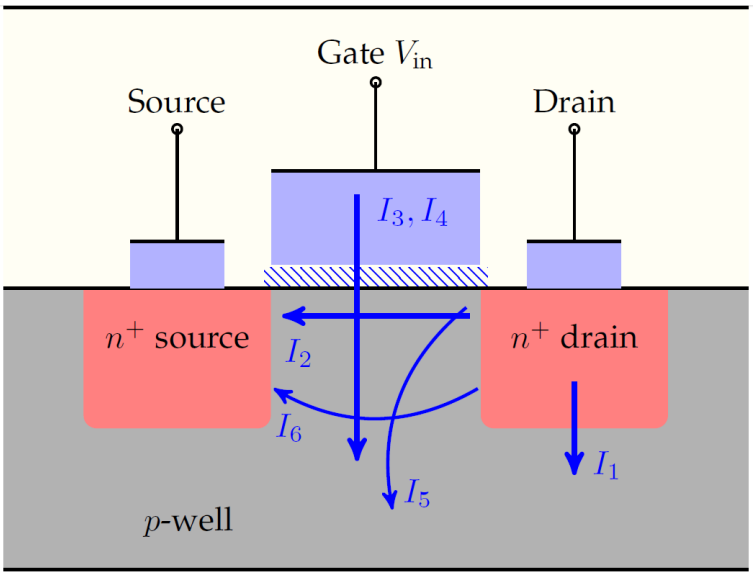

FIGURE 2.1 : Résume des SPD.

- I_5 Fuite du drain induit par la gate
- I_6 courant punch through

Les fuites I_2, I_5, I_4, I_6 ne se produisent quand dans l'état off ce qui fait que le SPD permet d'extraire de la donnée de ces fuites, I_1 et I_3 sont eux indépendants.

Fuite électromagnétique

Les émissions électromagnétiques viennent de la circulation du courant lors du traitement des données, du contrôle ou toute autre partie du circuit. Ces courants peuvent être volontaires ou involontaires. De plus, un courant peut aussi influencer les émanations d'autre composant par des phénomènes de couplage ou en fonction de la géométrie du circuit. La principale différence avec les fuites en consommation est que les fuites électromagnétiques sont locales alors qu'en consommations c'est l'agrégation totale de toutes les consommations. Elle permet aussi de différencier le passage des bits de 0-1 et de 1-0. Il faut différencier deux types d'émanations, celle direct et indirect :

– Les émanations directes qui dépendent directement de la circulation du courant. Ce sont souvent les composants avec les plus hautes fréquences qui sont les plus utiles à l'attaquant, car le bruit et les interférences sont plutôt en base de fréquence. Dans les circuits complexes, il peut être difficile d'isoler les émanations directes, il faut donc des sondes très précises et très proches de la source du signal. Il est même préférable

de décapsuler le circuit pour obtenir des résultats assez précis.

– Avec la miniaturisation et la complexification des circuits, les phénomènes de couplage électromagnétique et électrique sont de plus en plus significatifs du fait de la proximité des composants. Ces émanations se manifestent comme une modulation de signaux porteurs. Le signal d'horloge est un signal porteur très modulé par le circuit et est donc un important vecteur d'attaque. Cette modulation peut être une modulation d'amplitude s'il y a un couplage non linéaire entre la porteuse et le signal de donnée. La donnée peut être extraite grâce à une démodulation d'amplitude. Le couplage de circuit peut aussi entraîner une modulation de phase. Dans ce cas, une démodulation de phase s'impose. Étudier les émanations indirectes peuvent permettre de meilleurs résultats que les signaux directs, car ils se propagent mieux dans le circuit.

Autres fuites

Le temps d'exécution est un canal très important pour les fuites que ce soit au niveau des temps d'accès des caches [**bernstein_cache-timing_2005**] par exemple ce qui permet les attaques types spectre et Meltdown. Mais aussi en temps d'exécution ce qui a permis de casser certaine implémentation de RSA [**kocher_timing_1996**].

La température du circuit peut, de la même manière que la consommation fournir des informations sur l'état et les données qui sont en train d'être exécute sur le circuit. De même que les fuites EM la température permet d'extraire des informations locales. Cependant l'élévation de température n'est pas instantanée et il faut plusieurs milisecondes voir secondes pour avoir des fuites significatives ce qui limite les attaques à des systèmes qui répète en boucle un opération critique [**hutter_temperature_2013**]. Pour des raisons de précision il faut décapsuler le circuit pour le sonder. Les fuites acoustiques sont assez peu exploité, cependant dans [**genkin_acoustic_2017**] il a été mis en évidence que ces fuites, dans le contexte de CPU Intel pour laptop, permettait de déterminer les instructions qui était en train d'être exécutée. Toujours d'après cette article les fuites sont bien des vibrations des composants électriques du processeur même s'il est difficile de les caractériser précisément. Cette attaque a permis d'extraire les clés d'un chiffrement RSA.

	Time	SPD	DPD	EM	PSD	TSA
Hamming Weight	√	√	√	√	√	√
Hamming distance	√	×	√	√	√	×
Temporal Location	×	×	√	√	×	×
Spatial location	×	×	×	√	×	√

Table 2.1 : tableau récapitulatif des fuites extrait de [**korkikian_side-channel_2016**].

2.1.2 Modèle de fuite

Nous avons vu les différentes fuites qui sont possibles dans un circuit dans la section précédente, il est maintenant légitime de voir comment est-il possible d'évaluer ces fuites et comment les prendre en compte dans des modèles statistiques.

Il est possible de caractériser les fuites selon 2 groupes. Le premier est celui dont les fuites dépendent de l'état actuel du matériel, c'est-à-dire le nombre d'états à 1. C'est le modèle en poids de Hamming. Le second s'intéresse à la transition entre deux états pour en quantifier la différence. Plus précisément, le nombre de transitions entre ces deux états. C'est le modèle en distance de Hamming.

Il faut cependant prendre en compte de nombreux facteurs qui peuvent bruiter la mesure notamment les activités qui s'effectuent en parallèle de la donnée cible. Le bruit de l'environnent, mais aussi celui de la mesure et enfin les paramètres extérieurs tels que la température, les jitters d'horloge ou d'alimentation, etc. Même si on ne peut pas éliminer physiquement ces bruits selon la loi des grands nombres, à mesure que le nombre de variables distribuées de manière identique et aléatoire augmente, leur moyenne d'échantillon se rapproche de leur moyenne théorique. Ce qui permet de distinguer deux valeurs, même si le bruit est significatif, cela demandera juste plus de données.

La Table 2.2 présente les informations qui peuvent être obtenues à partir des fuites des canaux auxiliaires. Outre les données binaires l'information, c'est-à-dire le poids de Hamming ou la distance de Hamming, la fuite du canal auxiliaire peut caractériser le temps et l'emplacement spatial d'une donnée ciblée. Ainsi, les canaux auxiliaires peuvent être utilisés pour la rétro-ingénierie [**tang_power_2012**].

2.1.3 Exploitation des fuites

Attaques simples

Les attaques par analyse de courant les plus simple (Simple power attack SPA) sont des attaques qui fonctionnent en exploitant des dispositions ou des schémas dépendant de la clé dans une seule trace de fuite, par exemple l'enregistrement de la consommation électrique d'un appareil pendant un chiffrement ou une sous-séquence de la procédure de chiffrement. Cela ne signifie pas que pour une attaque SPA, une seule trace de puissance est utilisée. Des traces multiples pourraient être utilisées pour réduire le rapport signal/bruit entre le signal de fuite réel et le bruit électronique et de mesure, par exemple. Cependant, le terme "simple" fait référence au fait que dans une SPA, la relation entre les traces multiples et les changements à l'intérieur des valeurs intermédiaires qui produisent la trace de fuite enregistrée n'est pas exploitée.

Un exemple classique d'une attaque SPA est la reconnaissance des modèles de fuite qui sont causés par les commandes individuelles d'un processeur. Un carré et une multiplication tels qu'ils sont utilisés dans l'exponentiation binaire des implémentations non sécurisées de l'algorithme de multiplication par un scalaire d'un point d'une courbe elliptique ou également de la multiplication de nombres premiers dans le chiffrement RSA, montre que selon qu'il s'agit d'un simple mise au carré (opération par groupe) ou d'un mise au carré plus une multiplication les traces différeront.. Comme ces modèles peuvent même apparaître dans la simple inspection visuelle d'une trace de puissance, l'exposant qui a été utilisé pour la l'exponentiation (qui peut être la clé elle-même ou liée à la clé) peut être directement extrait de la trace de puissance.

Attaques non supervisé

En plus des variations de puissance, il y a des effets corrélés aux valeurs des données manipulées. Ces variations tendent à être plus petits et sont parfois éclipsés par des erreurs de mesure et d'autres bruits. Dans ces cas, il est encore souvent possible d'exploiter les fuites en utilisant des fonctions statistiques adaptées à l'algorithme cible. Une attaque classique peut être divisée en trois phases : la phase de collecte des fuites, la phase de construction des hypothèses et la phase de correspondance des hypothèses. Dans ce qui suit, nous ne résumons que brièvement le fonctionnement d'une attaque DPA typique.

Dans la première phase, les traces de fuite sont collectées sous différentes entrées, par

exemple différents textes en clair qui sont chiffrés à l'aide de la même clé secrète. L'exploitation des différences dans les traces de fuite nécessite un alignement, aussi exactes que possible, des traces individuelles les unes par rapport aux autres. Pour les attaques pratiques, il n'existe souvent pas de point de référence dans le temps pour chaque chiffrement. Cette phase pourrait donc comporter une phase de post-traitement au cours de laquelle les traces de fuite sont d'abord alignées. Les attaques nécessitent un post-traitement sont plus complexe car elles impliquent le filtrage ou la combinaison de points de fuite. Etant donné qu'aucune connaissance de la clé secrète utilisée n'est supposée, l'étape suivante consiste à générer des hypothèses sur la base de la clé secrète inconnue. Cette phase est généralement réalisée de manière à diviser pour mieux régner, car la construction d'hypothèses basées sur un espace clé de 128 bits, tel qu'il est requis pour un AES-128 complet, par exemple, serait trop complexe. Par conséquent, il faut d'abord trouver un point approprié dans l'algorithme attaqué pour lequel il existe une relation avec la clé secrète attaquée et les données d'entrée connues et qu'a cet instant la clé et la donnée soient processées par paquet de taille inférieure à 128 bits. Un point approprié pour la construction de cette hypothèse est par exemple le résultat de l'addition de la clé et du texte en clair dans le premier tour d'un AES qui est suivie par la recherche de la S-box 8 bits. On peut ainsi calculer des hypothèses individuellement pour tous les morceaux de 8 bits de la clé de 128 bits. Les hypothèses contiennent les valeurs potentielles du résultat intermédiaire attaqué de l'algorithme qui sont calculé pour chaque trace de fuite de l'octet clé ciblé. Comme la puissance dynamique d'un circuit CMOS dépend des modifications d'un signal plutôt que des valeurs absolues de la valeur intermédiaire, les valeurs intermédiaires hypothétiques sont ensuite mises en correspondance avec un modèle de fuites. Cette mise en correspondance peut à nouveau être assez simple, par exemple en calculant simplement le nombre de bits que sont non nuls (modèle de Hamming-weight) ou le nombre de bits qui ont changé (modèle de Hamming-distance), mais peut aussi devenir plus complexe et utiliser les caractéristiques de consommation de l'appareil attaqué.

Dans la phase de correspondance, le modèle de puissance hypothétique pour les principales suppositions est évalué statistiquement par rapport aux observations réelles dans les traces de fuite. Dans la pratique, il existe un grand nombre de d'outils mathématiques permettant la distinction des clés utilisées sur les fuites par canaux auxiliaires.x Pour sélectionner le candidat clé le plus probable parmi l'ensemble des clés hypothétiques. Ces distinctions sont basées sur des méthodes statistiques différentes, comme La corrélation de Pearson, la différence de moyennes ou l'analyse d'informations mutuelles, et ont des

propriétés et des implications pratiques variables. L'objectif de toutes ces outils est cependant de trouver et de quantifier les dépendances entre les hypothétique modèle de fuite afin de déterminer la clé utilisée.

Attaques supervisé

Essentiellement, nous avons un dispositif effectuant l'une des K séquences d'opérations possibles, $O_1, \ldots, O_k$: il peut s'agir, par exemple, d'exécuter le même code pour différentes valeurs de bits clés. Un adversaire qui peut échantillonner le canal auxiliaire pendant cette opération souhaite identifier laquelle des opérations est exécutée ou alors réduire de manière significative l'ensemble des hypothèses possibles pour l'opération. Dans le traitement du signal, il est habituel de modéliser l'échantillon observé comme une combinaison d'un signal intrinsèque généré par l'opération et de bruit qui est soit intrinsèquement généré, soit ambiant. Alors que la composante du signal est la même pour les invocations répétées de l'opération, le bruit est mieux modélisé comme un échantillon aléatoire tiré d'une distribution de probabilité du bruit qui dépend des conditions de fonctionnement et d'autres conditions ambiantes. L'approche optimale pour l'adversaire, qui tente de trouver la bonne hypothèse à partir d'un petit nombre d'échantillons S, est d'utiliser l'approche du maximum de vraisemblance : La meilleure hypothèse consiste à choisir l'opération de telle sorte que la correspondance entre les traces et le modèle soit maximal. Pour calculer cette correspondance, l'adversaire doit modéliser avec précision à la fois le signal intrinsèque et la distribution de probabilité du bruit pour chaque opération.

L'adversaire utilise un dispositif expérimental, identique au dispositif testé, pour identifier une petite partie de l'échantillon S qui ne dépend que de quelques bits de clés inconnus. Avec l'expérimentation, il construit des modèles correspondant à chaque valeur possible des bits clés inconnus. Le modèle est constitué des distributions de probabilité moyennes du signal et du bruit. Il utilise ensuite ces modèles pour classer cette partie de S et limiter les choix pour les bits clés à un petit ensemble. Cette opération est ensuite répétée avec un préfixe de S plus long impliquant plus de bits clés. Nous ne retiendrons qu'un petit nombre de possibilités pour la partie de la clé considérée jusqu'à présent. Ainsi, les attaques par template utilisent essentiellement une stratégie d'extension et de réduction dirigée par l'échantillon unique S à attaquer : nous utilisons des préfixes de S de plus en plus longs et les templates correspondants pour réduire l'espace des clés possibles. Le succès dépend essentiellement de l'efficacité avec laquelle la stratégie de réduction réduit l'explosion combinatoire dans le processus d'extension.

Les attaques par template sont particulièrement efficaces sur les implémentations d'algorithmes cryptographiques sur des dispositifs CMOS en raison de leur contamination et de leur diffusion sur plusieurs cycles dans une section de calcul. Dans les dispositifs CMOS, la manipulation directe des bits clés les fait entrer dans l'état du dispositif et ces fuites d'état peuvent persister pendant plusieurs cycles. En outre, d'autres variables affectées par la clé, telles que les indices et les valeurs des tables dépendantes de la clé, provoquent une contamination supplémentaire lors d'autres cycles. L'étendue de la contamination contrôle le succès de la réduction des nouveaux bits clés introduits dans la phase d'expansion. Il faut s'attendre à ce que si deux clés sont presque identiques, même avec les effets de la contamination, la réduction ne puisse pas éliminer l'une d'entre elles. La diffusion est la propriété cryptographique bien connue par laquelle de petites différences dans les bits clés sont amplifiées dans les parties suivantes du calcul. Même si certains candidats pour les bits clés n'ont pas été éliminés en raison des effets de la contamination, la diffusion garantira que les clés très rapprochées seront élaguées rapidement.

L'implémentation d'un algorithme sur un dispositif particulier impose par nature des limites théoriques au succès de l'attaque du modèle. Le mieux qu'un adversaire puisse faire pour approcher cette limite théorique est de disposer de caractérisations extrêmement bonnes et précises du bruit. Bien que de telles caractérisations soient très sophistiquées, en pratique, des approximations telles qu'un modèle gaussien multivarié pour les distributions du bruit donnent de très bons résultats. Et les progres en machine learning permet des attaques supervisé encore plus précise si les données d'entraînement sont en nombre suffisant.

2.1.4 Méthode de test

Deux méthodes de test sont principalement utilisées pour évaluer la résistance des implémentation face aux attaques par canaux auxiliaires la première empirique où on évalue la résistance avec l'analyse des fuites et des méthodes statistiques. La seconde est formelle c'est-à-dire que l'on vérifie si notre design respecte des propriétés mathématiques. Les deux méthodes sont souvent complémentaires, on vérifie d'abord formellement si l'implémentation est formellement résistante puis on vérifie par l'expérience si c'est effectivement le cas.

Méthode empirique

Analyser la résistance contre les attaques par canaux auxiliaires des implémentations de manière empirique, est souvent effectué par l'intermédiaire d'un t-test selon la méthode de Goodwill et al. [14] . Nous notons que les t-tests ne sont pas adaptés pour prouver des déclarations générales sur la sécurité d'un modèle (pour toutes les conditions et tous les temps de signal possibles) comme il serait nécessaire pour une vérification complète de la sécurité. Les t-tests n'autorisent des déclarations que pour les dispositifs testés et dans les limites du dispositif de mesure. De nombreux ouvrages testent les circuits masqués sur un FPGA (Field Programmable Gate Array) et effectuent le t-test sur les traces recueillies à partir des mesures de puissance. Cette approche présente l'inconvénient qu'en raison des niveaux de bruit relativement élevés, l'évaluation est généralement limitée à des t-tests multivariés du premier et du deuxième ordre. Cependant, dans la pratique, les t-tests se sont avérés très sensibles et utiles pour tester la résistance des canaux auxiliaires de circuit. Les traces de signaux sont enregistrées lors des simulations post-synthèse des netlists, qui sont exemptes de bruit et nous permettent d'évaluer les conceptions jusqu'au troisième ordre. L'utilisation de traces de fuites post-synthèse sur des traces collectées à partir d'une conception de FPGA ou de circuit intégré spécifique à une application (ASIC) montre quelques différences qui sont dans certains cas très bénéfiques, mais il y a aussi des inconvénients. Tout d'abord, les traces de fuite après synthèse sont totalement exemptes de bruit environnemental et de variations des conditions de fonctionnement comme la température ou la tension d'alimentation. En conséquence, les violations de la sécurité d'ordre D sont constatées avec beaucoup moins de traces de fuite. Un autre grand avantage est que les t-tests peuvent être effectués soit à un niveau assez grossier, en prenant en compte tous les signaux ensemble, soit à une granularité très fine en utilisant des signaux individuels. Cette dernière méthode permet de localiser directement la source de la fuite au niveau du signal, ce qui facilite grandement la conception des tests. L'un des inconvénients de cette approche est que les traces post-synthèse n'utilisent pas une source de fuite réelle existante. Cependant, un t-test effectué sur une puce ASIC ou sur la conception d'un FPGA ne permet de donner qu'une fuite existante sur ce dispositif et ne donne même pas de garantie sur son comportement à l'avenir, car les retards de signal peuvent changer dans les conditions environnementales et au cours du cycle de vie d'un dispositif. Dans le cas de la netlist synthétisée simulée, les retards de signal sont basés sur les retards de la porte unifié qui entraînent également des glitchs de signal qui apparaissent à partir des portes logiques en cascade. Les glitchs qui pourraient résulter des longueurs de fils de différentes, et autres ef-

fets parasites, ne sont cependant pas modélisés et sont donc plus susceptibles d'apparaître sur les t-tests basés sur FPGA ou ASIC. Pour vérifier qu'un appareil ne présente aucune fuite exploitable, deux séries de traces sont collectées par t-test :

1. un ensemble avec des entrées choisies au hasard
2. la valeur t est calculée selon l'équation ci-dessous où X désigne la moyenne de l'ensemble de traces respectif, S2 est la variance et N est la taille de l'ensemble.

$$t = \frac{X_1 - X_2}{\sqrt{\frac{S_1^2}{N_1} + \frac{S_2^2}{N_2}}} \tag{2.1}$$

L'hypothèse nulle est que les moyennes des deux ensembles de traces sont égales, ce qui est accepté si la valeur t calculée est inférieure au seuil de $\pm$4,5. Si la valeur t dépasse ce seuil, alors l'hypothèse nulle est rejetée avec une confiance supérieure à 99,999 % pour les ensembles de traces suffisamment importants. Une étape dite de prétraitement centré du produit, avec des points de trace à l'intérieur d'une fenêtre de six cycles, est effectuée pour les t-tests d'ordre supérieur. Au-delà de cette fenêtre de temps, on s'assure que les produits intermédiaires n'ont aucun lien avec les entrées. Nous combinons donc plusieurs points de trace en normalisant d'abord les moyennes des points de trace, puis en multipliant les valeurs résultantes par d'autres points normalisés à l'intérieur de la fenêtre temporelle.

Méthode formelle

Le masquage booléen d'ordre d (d share) d'une variable $a \in GF(2^m)$ est représenté par $s_a = a_{i(i=1)^n}$ où chaque part $a_i \in GF(2^m)$ est un nombre tiré aléatoirement. La donnée est ensuite combiné avec les différentes parts. Pour retrouver la donnée il faut ainsi combiner les differnet share via l'operation inverse.

Le nombre de parts n dépend du schéma de masquage, et est toujours supérieur à l'ordre de sécurité d. Toute combinaison des parts d au maximum ne doit pas donner d'informations sur a. Toute fonction $F(a) = x$ peut être mise en œuvre en utilisant un ensemble d'opérations affines et de multiplications dans le champ correspondant. Le calcul masqué d'une fonction affine A est trivial, car la fonction peut simplement opérer sur chaque action individuellement : $A(a_i) = x_i$. En revanche, le AND logique masquée est plus difficile. Nous donnons ci-dessous un exemple de partage pour la fonction $F(a, b) = ab$ qui utilise trois parts et une variable aléatoire r_i, tirée de [**ishai_private_2003**].

$$t_1 = (a_1 b_2 \oplus r_1) \oplus a_2 b_1$$

$$t_2 = (a_1 b_3 \oplus r_2) \oplus a_3 b_1$$

$$t_3 = (a_2 b_3 \oplus r_3) \oplus a_3 b_2$$

$$x_1 = a_1 b_1 \oplus r_1 \oplus r_2$$

$$x_2 = a_2 b_2 \oplus t_1 \oplus r_3$$

$$x_3 = a_3 b_3 \oplus t_2 \oplus t_3$$

Definition 2.1.1 (d-probing security [**ishai_private_2003**])**.** Un circuit est sécurisé d-probing si et seulement si chaque d-tuple de ses variables intermédiaires est indépendant de toute variable sensible.

Notez que l'ensemble des équations présentées est sûr car chaque variable intermédiaire (en plus des variables d'entrée et de sortie) est indépendante des variables sensibles non masquées. Il a été montré que pour un circuit sans aucun glitchs, la sécurité d-probing implique la sécurité du modèle bruité. Ce modèle suppose que chaque opération fuit indépendamment, lorsqu'il n'y a pas de bruit entre les actions, et que la somme des fuites de bruit de chaque action est fournie à l'adversaire. Il a été démontré que ce modèle correspond à des fuites physiques réelles. Toutefois, il a également été démontré qu'il présente deux inconvénients majeurs.

1. La sécurité face au attaque par canaux auxiliaires d'un gadget tel qu'une porte ET masquée n'implique pas la sécurité d'un circuit où ces gadgets sont composés arbitrairement
2. Il a été démontré que l'hypothèse selon laquelle un circuit ne présente pas de glitchs est irréaliste, en particulier sur les circuits matériels

Pour repondre aux deux lacunes de ce modèle de nouveau modèle ont émergés :

Definition 2.1.2 (Gadget composable)**.** Un gadget sécurisé d-probing est composable si la combinaison arbitraire de ces gadgets aboutit à un circuit sécurisé d-probing.

Plus tard, il a été démontré dans [**barthe_strong_2016**] qu'un gadget satisfaisant à la propriété d-SNI telle que décrite ci-dessous est composable. De plus, tout circuit composé de gadgets affines (où les limites de partage ne sont pas violées, c'est-à-dire $A(a_i) = x_i$) et de gadgets d-SNI est d-probing sécurisé.

Definition 2.1.3 (d-Strong Non-Interference [**barthe_strong_2016**])**.** Un gadget est un d-SNI (d-Strong Non Interfering) si et seulement si, pour un ensemble de p_1 sondes sur ses valeurs intermédiaires et chaque ensemble de p_2 sondes sur ses sorties, la totalité des sondes peut être simulée avec $p_1 + p_2 \leqslant d$, avec p_1 parts de chaque entrée. Ici, la simulation implique une fonction qui prend des parts p_1 pour chaque entrée et calcule une distribution conjointe qui est exactement égale à la distribution produite sur ses d sondes par le gadget ou l'algorithme étudié [**barthe_improved_2020**]

Donc si un modèle de circuit est sécurisé dans le cadre du modèle d-probing, il peut ne pas l'être dans la pratique. En effet, le modèle ne tient pas compte des effets physiques tels que les glitchs. Les glitchs sont des artefacts matériels involontaires et indésirables qui provoquent des consommations d'énergies involontaires et les concepteurs de matériel font de grands efforts pour les minimiser pour des raisons qui vont au-delà de la sécurité. Toutefois, la réduction des glitchs nécessite un processus de cheminement minutieux, qui est un grand défi compte tenu de facteurs tels que l'architecture, l'environnement de travail et l'âge de l'appareil. Plus important encore dans notre contexte, les glitchs peuvent momentanément démasquer des valeurs, invalidant ainsi théoriquement les garanties de sécurité et en le rendant dangereux pour la sécurité. Le modèle de extended d-probing a été créé pour remédier aux défauts du modèle d-probing en ce qui concerne les défauts physiques tels que les glitchs [**meyer_multiplicative_2018**]. Dans ce modèle, chaque sonde glitch-extended ne donne pas seulement des informations sur le fil sondé, mais aussi toutes les variables utilisées pour calculer la valeur de ce fil jusqu'au dernier point de synchronisation. Notez qu'il s'agit d'un modèle très solide couvrant les fuites dans le pire des cas qui pourraient ne pas se produire en pratique. Toutefois, il fournit un modèle théorique que l'on peut utiliser avec un degré de confiance élevé. Un gadget est considéré comme composable sous le modèle d glitch-extended probing s'il satisfait à la propriété d glitch-extended SNI (d-GSNI) comme décrites ci-dessous.

Definition 2.1.4. [**meyer_consolidating_2019**]] Considérons un gadget avec d + 1 parts d'entrée a_i, où $i \in 1, ..., d+1$. Soit O tout ensemble d'observations d'au plus d sondes glitch-extended dans $GF(2 \wedge m)$. Soit p_1 et p_2 le nombre de sondes intermédiaires et de sortie respectivement tel que $p1 + p2 \leqslant d$. Le gadget est d-GSNI si, pour l'un quelconque de ces O, la condition suivante est remplie :

$\exists P \subset \{1, ...d+1\}$ et son complément $\overline{P}$ avec $|P| = p_1$ tel que $I(O; a'_P | a_P) = 0$.

Notez que cette définition suppose un point de vue de théorie de l'information et est équi-

valente à la définition 2.1.1 si des sondes régulières sont utilisées à la place des sondes glitch-extended.

2.1.5 Classifications des contremesure

De nombreuses contremesures ont été développé depuis la première attaque par canaux auxiliaires en 1999 [**kocher_differential_1999**]. Ces contremesures peuvent être repartie en plusieurs groupes :

- Ajouter du bruit temporel pour éviter les attaques par analyse de consommation, de timing ou d'émission électromagnétique.
  - Instabilité dans le rapport cyclique ou la fréquence d'horloge.
  - Circuits asynchrones, le circuits n'est plus soumis a une horloge pour effectuer les opérations.
  - Ajout de opération factice aléatoire, qui sont des opérations qui n'ont pas d'influence sur le fonctionnement du circuit.
- - L'homomorphisme utilise les propriétés arithmétiques pour obtenir le résultat en fonctions d'un ou plusieurs chemins de calcul.

  La randomisation consiste à changer la représentation de la donnée sensible, elle n'est donc plus directement accessible à l'attaquant. Ces contre-mesures sont efficaces contre tous les types de canaux auxiliaires.
  - Le masquage consiste à combiné la donnée sensible avec une donnée aléatoire pendant tout le processus à effectuer sur cette donnée. A la fin de ce processus on peut démasquer pour obtenir le résultat.
  - Ajouter du bruit dans la consommation globales ou local du circuit :
    * Préchargement de données dans les registres ou le chemin de données
    * Instabilité dans l'alimentation.
- Les détecteurs qui sont capable de reconnaître si un circuit est en train d'être attaqué. Il est possible de détecter le depackaging, les sondes EM, les modifications d'horloges ou d'alimentation etc... Cependant il est toujours possible de passer outre ces protections.

item La réduction des émissions par canaux auxiliaires consiste à diminuer au niveau des portes du circuit les dépendances entre les données. On peut ainsi équilibrer consommation d'énergie ou avoir une exécution en temps constant. De nombreuses solutions ont été proposé notamment pour équilibré ou réduire la consommation : Dual-Rail Random Switching Logic (DRSL)[**chen_dual-rail_2006**], Masked Dual-Rail Pre-charge Logic (MDPL)[**popp_masked_2005**], Dual-Rail Pre-charge Logic (TDPL)[**bucci_three-phase_2006**], Random Switching Logic (RSL)[**suzuki_random_2004**], Sense Amplifier Based Logic (SABL), Wave Dynamic Differential Logic (WDDL)

2.2 Attaque par injection de fautes

Après l'observation du circuit et de ses fuites par canaux auxiliaires, il est possible d'interférer directement avec le circuit. Le fonctionnement optimal d'un circuit est donné pour un milieu avec des contraintes physiques données (température, alimentation, hydrométrie, etc..). De plus dans les années 70, avec l'avènement de l'ère spatiale, les premières fautes sur les circuits ont été observées dues aux rayons cosmiques hors de l'atmosphère terrestre. Avec la réduction de taille des transistors et donc de leur énergie d'activation, on a commencé à observer ces effets dans des contextes terrestres ou aéronautiques à des échelles certes inférieures, mais tout de même présentes. On a appeler ces erreurs des soft errors.

Les fautes peuvent également être intentionnelles, on parle alors d'attaque par injection de fautes. Elle consiste à venir perturber un circuit de manière volontaire et contrôlée en changeant de manière parfois brutale ses conditions de fonctionnement. De telles attaques sont généralement invasives, modification du composant pour effectuer l'attaques, ou semi-invasives avec la non modification du composant, on considere que ce genre d'attaque est toujours invasives car il y a modification du fonctionnement du circuit (en opposition aux attaques par canaux auxiliaires qui sont plutôt non invasives). Ces attaques sont redoutables et peuvent mettre en défaut un système,déni de service, modifier son comportement à des fins profitables,contournement d'authentification, élévation de privilèges, ou encore permettre de révéler des données sensibles, des clés cryptographiques par exemple.

Les attaques par injections se développent à la fin des années 90 [**biham_differential_1997**], comme dit précédemment les fautes dans le domaine du spatial datent du début des années 70. Cette différence est que le sujet des fautes

dans le domaine spatial est bien plus traité. Dans le domaine de l'injection de faute, ce sont principalement les crypto processeurs qui cherchent à éviter les fautes. En effet de nombreuses exploitations de l'injection de fautes sont possibles dans les algorithmes de chiffrements.

Il est important d'établir une différence claire entre sécurité et sureté. En sécurité, on est en présence d'un adversaire malveillant envers notre système. Cette différence de nature du risque induit une différence dans la gestion de celui-ci. Dans le cas de la sureté, le risque est statique et probabiliste, c'est-à-dire qu'il est connu lors de la création du système et peut donc être anticipé de manière probabiliste. Dans le cas de la sécurité, l'adversaire cherche à mettre à mal les contremesures implémentées et toutes les contremesures peuvent devenir de nouveaux vecteurs d'attaques.Ces deux cas de figure sont très différents et il peut être difficiles de les concilier, en effet le sûreté se base souvent la correction de l'erreur ce qui offre plus de possibilité pour un attaquant. La sécurité elle se base principalement sur la détection et la réponse a incident via des routines spécifiques.

2.2.1 Fautes stochastiques

Différents types de fautes qui peuvent intervenir, quelle que soit indifféremment de la méthode d'injection de fautes. Il y a de nombreux single-event effets (SEEs) transitoires possible :

- Single-event transients (SETs) qui causent un changement temporaire de tension à la sortie d'une porte.
- Single-event upsets (SEUs) qui causent une inversion de la valeur d'une mémoire.
- Single-event functional interrups (SEFIs) qui causent une dysfonction jusqu'au redémarrage du système [**koga_single_1997**].

En cas de concentration trop importante d'énergie, les SEEs vont causer des dommages permanents :

- Single-event latchups (SELs) qui cause un thyristor parasite dans le CMOS qui active celui-ci en permanence avec une tension haute.
- Single-event burnouts (SEBs) qui cause une polarisation directe dans le transistor parasite d'un power MOSFET.

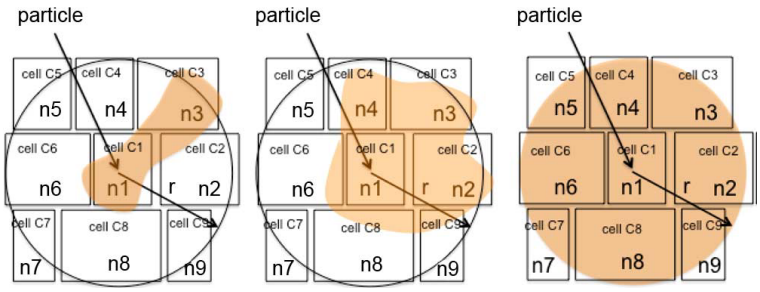

Figure 2.2 : Influence d'une faute.

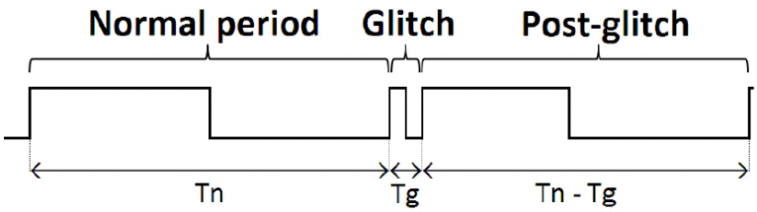

Figure 2.3 : Glitch sur le signal d'horloge.

- Single-event gate ruptures (SEGRs) qui causent un champ électrique transitoire à travers la gate oxide d'un power MOSFET

Plusieurs de ses fautes peuvent se réaliser au même instant on parle alors de multiple-event effets (MEEs). De plus, avec la réduction des tailles de gravures une particule peut causer plusieurs softs errors[**pagliarini_analyzing_2011**], [**zoutendyk_characterization_1989**]. Il en résulte qu'une soft erreur peut affecter plusieurs bits d'une même variable.

Si l'on se réfère aux données que l'on peut trouver dans des articles de la NASA, un processeur sera soumis en moyenne à 5-10 fautes par heures et plus de 99,99% d'entre elles seront des fautes transitoires et 99,9% sont des SEUs [**springer_analysis_2001**].

2.2.2 Faute non stochastique

Glitch d'horloge

Le glitch d'horloge est décrit en Figure 2.4, elle explique comment des une violation des contraintestemporelle peut induire des fautes. Il consiste en la modification de la fréquence d'horloge nominale T_n par l'injection d'un glitch de période T_g. le principe est d'overclocker temporairement le circuit c'est-à-dire $T_g \ll T_n$ Ce qui peut potentiellement causer des violations de contrainte temporelles. Si $T_g \ll T_n$ Le post-glitch ne modifie pas le fonctionnement normal du circuit [**balasch_-depth_2011**]. Il y a deux caractéristiques aux erreurs

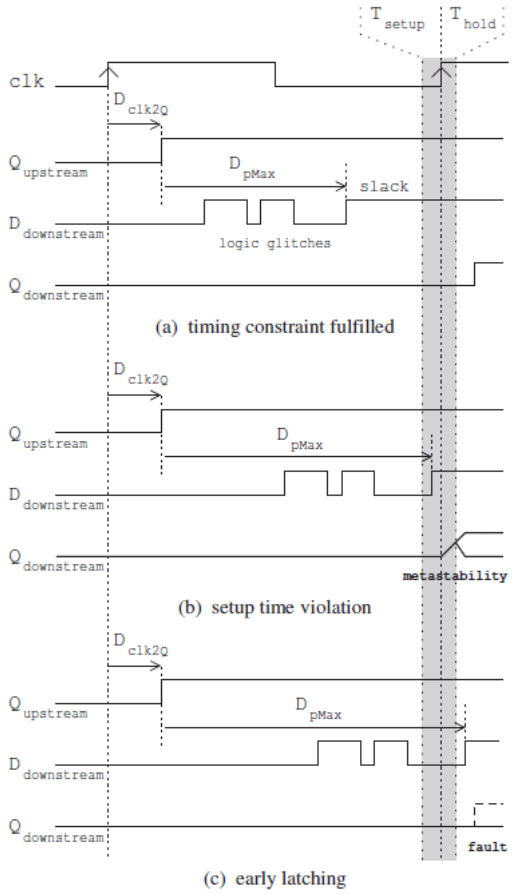

FIGURE 2.4 : Violation de contrainte temporelle.

par violation de contrainte temporelle :

- Les fautes dépendent des données en cours d'exécution. En effet, les données peuvent modifier les chemins critiques notamment dans le cas des instructions d'un processeur d'application.
- Le processus d'injection peut amener à des états métastables quand le stress applique au circuit est trop faible. Ils peuvent devenir déterministes si on augmente ce stress. Il faut donc trouver les meilleurs paramètres pour permettre la faute.

Ces deux propriétés sont communes à toutes les injections qui modifient les contraintes temporelles du circuit (glitch d'alimentation, changement de température, injection EM).

Glitch d'alimentation

Le glitch d'alimentation consiste en faire varier l'alimentation du circuit pendant un bref instant. Cette injection est la plupart du temps est sous-alimentation. L'explication la plus probable est la modification des caractéristiques physiques des transistors s'en trouve modifié et des erreurs de temps de SETUP et de HOLD sont observées. [**djellid-ouar_supply_2006**] Il est difficile de définir si les erreurs de timings sont la seule cause des fautes, mais c'est actuellement la piste la plus sérieuse [**zussa_power_2013**]. Dans ce même article, les auteurs arrivent à la conclusion de l'équivalence entre les glitchs d'alimentations et les glitchs d'horloges. Il est possible d'augmenter la reproductibilité des glitchs d'alimentations en injectant non plus des signaux carrés en entrée, mais des signaux de forme aléatoire [**bozzato_shaping_2019**]. Les fautes ont entre 2 et 10 fois plus de chance de se produire avec ce type d'injection.

Changement de température

Les équipements électroniques ont une certaine plage de température d'utilisation. De nombreuses publications mentionnent seulement la possibilité de réaliser des attaques par changement de température, mais beaucoup moins l'ont effectivement réalisé. Le refroidissement est principalement utilisé pour récupérer des données dans les SRAMs après l'extinction [**halderman_lest_2009**]. Pour l'injection de faute, on se concentre plus sur l'augmentation de température. Il est par exemple possible de produire des fautes avec une probabilité de 71,4% avant que la machine s'arrête de fonctionner en chauffant à 100 °C avec une lampe de bureau de 50W. Une exposition prolonge à des températures élevées peut provoquer une instabilité de l'ordre de 30% des bits de la mémoire. [**hutter_temperature_2013-1**] Le changement de température affecte tout le circuit. Il ne semble pas qu'il y ait des injections de température précise et la gestion temporelle de celle-ci. Enfin les fautes produites peuvent être permanente ce qui diminue grandement l'intérêt de ces attaques.

Injection combinée

Il est possible de combiner les différentes méthodes d'injections de fautes. En effet, il est possible de combiner glitch temporel, d'alimentation et les changements de température [**kumar_precise_2014**]. En effet, en combinant les attaques on arrive à fixer les paramètres physiques et ainsi affiner la précision des techniques d'injection. Cela permet d'attaquer

des parties très localisées du circuit à moindre coût. Cependant ces injections combinées ne permettent pas l'émergence de nouvelles fautes, mais permettent d'augmenter la reproductibilité de certaines.

Injection laser

Le laser (Light Amplification by Stimulated Emission of Radiation) est une émission électromagnétique monochromatique unidirectionnelle et cohérente. Ce laser peut être de diamètre très faible de l'ordre du µm et peut traverser de nombreux matériaux sur une période très courte. Il semble donc parfaitement adapté pour l'injection de faute. La première utilisation de laser pour induire des fautes sur les circuits électroniques a été reportée par [**skorobogatov_optical_2003**]. La meilleure explication du pourquoi le laser est donné par JM Dutertre [**dutertre_laser_2018**]. Différents paramètres sont à prendre en compte lors d'une attaque par laser. Le diamètre, la longueur d'onde, les coordonnées, la quantité d'énergie et la durée d'exposition. Il faut notamment faire attention à la durée d'exposition et la quantité d'énergie quand on fait des fautes par laser, car celle-ci peut créer des erreurs permanentes[**darracq_single-event_2001**]. L'avantage est la reproductibilité des fautes par laser, mais aussi sa grande précision de l'ordre de l'octet ou du bit même sur des technologies récentes. [**agoyan_how_2010**] [**selmke_precise_2016**]. De plus, de nouveaux bancs d'injections permettent de faire des injections multispots, des bancs 2 et 4 spots sont actuellement commercialisés. Lors d'une injection laser bien que le diamètre du laser soit de l'ordre du µm la précision de celui-ci est plus élevée, car le centre du laser possède plus d'énergie et donc il y a plus de chance de faute. [**godlewski_electrical_2009**] L'injection laser peut être effectuée sur face avant ou face arrière bien que les caractéristiques soient différentes le principe reste identique. Pour l'attaque par face avant le positionnement est facilité par la visibilité des composants électroniques. Cependant, à cause de la couche métallique d'interconnexion et de sa réflexivité, il est difficile d'avoir une précision élevée. On utilise souvent des longueurs d'onde de 523nm. Pour les attaques par face arrière on préfère utiliser des longueurs d'onde de 1064 nm pour une meilleure pénétration dans l'épaisseur de silicium. Cependant le positionnement est plus difficile, mais la précision des fautes est plus élevée, car on évite la couche métallique. Il faut prendre en compte les coefficients d'absorption et de réflexion du silicium qui dépendent grandement de la longueur d'onde [**breier_testings_2015**]. Les résultats d'une campagne d'injection réalisée sur une puce de 28nm sont données en figure 2.2.

Ces résultats montrent le fait que les fautes peuvent modifier un nombre important de bits

Energie[nJ]	0.4	0.5	0.8	1	1.5	2	3	4	5
nb de fautes	1	8	21	23	24	24	26	30	31
nb de fautes 1-bit	1	8	15	17	10	7	7	9	9
nb de fautes 2-bit	-	-	6	6	7	5	4	5	6
nb de fautes 3-bit	-	-	-	-	4	7	8	4	4
nb de fautes 4-bit	-	-	-	-	3	3	3	5	1
nb de fautes 5-bit	-	-	-	-	-	1	1	2	4
nb de fautes 6-bit	-	-	-	-	-	1	1	2	2
nb de fautes 7-bit	-	-	-	-	-	-	1	2	4
nb de fautes 8-bit	-	-	-	-	-	-	-	1	1

Table 2.2 : tableau récapitulatif du nombre et taille de faute en fonction de la puissance du laser.

consécutif, il est ainsi possible de réaliser des faute multiple avec une unique source de faute. Ainsi il est possible de procéder à un grand nombre de faute consécutive sur une donnée particulière avec une puissance de laser suffisante.

Injection EM

En 2002, Quisquater et Samyde mettent en évidence que le champ magnétique d'une sonde électromagnétique peut perturber les calculs sur un circuit [**quisquater_eddy_2002**]. C'est en 2007 que cette technique est réalisée pour effectuer une attaque, l'attaque Bellcore [**Schmidt_opticaland**]. Contrairement à ce qu'il a été longtemps cru les fautes induites par l'injection EM ne sont pas seulement des erreurs de timing [**dehbaoui_electromagnetic_2012**], mais aussi des samplings fault c'est-à-dire des fautes au niveau de la porte et non pas au niveau du circuit (bits-set et des bits-reset) [**dumont_electromagnetic_2019**] [**ordas_electromagnetic_2017**]. L'avantage de l'injection de faute par électromagnétisme est qu'il n'est pas nécessaire de décapsuler le circuit. Cependant dans la pratique il est souvent nécessaire de décapsuler le circuit pour obtenir de meilleure performance d'injection. Quand à la précision temporelle elle reste inférieure aux injections par glitch et par laser il faut charger le générateur d'impulsion avant d'effectuer l'attaque.

2.2.3 Méthode de test

On peut classer les méthodes de test en quatre catégories en fonction du niveau selon lequel on se place :

- effectuer les tests directement sur le système physique cette technique
- logiciel
- simulation
- émulation

Pour chacune de ces méthodes on peut se placer à différent niveau d'abstraction : assembleur/jeux d'instructions, RTL, porte logique, post-implémentation. Nous ne détaillerons pas l'injection physique de faute, car cela a déjà été fait en partie I. On peut cependant noter que pour l'injection de faute physique dans un contexte de sureté, des outils ont été mis en place pour automatiser les campagnes de fautes [**arlat_fault_1990**],[**madeira_rifle_1994**]. Cette synthèse se base sur différentes bibliographies déjà existantes dont j'ai essayé de concilier les différents apports de chacune d'entre elles. Cependant elle rentre souvent beaucoup plus en détail sur les différents outils possibles pour chacune des catégories. [**eslami_survey_2020**], [**kooli_survey_2014**]

Niveau d'abstraction

On peut tester la résistance les fautes à plusieurs niveaux d'abstraction. Chaque niveau a ses avantages selon le contexte. Nous allons commencer par les modèles de plus hauts niveaux.

- Le niveau le plus abstrait est celui d'assembleur/jeux d'instruction. Il consiste à modifier le code source d'une application pour modifier le fonctionnement du processeur. Il tient plus compte des détails d'architectures et permet de simuler des fautes. L'avantage de cette solution est de permettre d'injecter facilement des fautes sur des applications et le système d'exploitation. Cependant elle ne permet d'injecter des fautes sur les ressources accessibles via le jeu d'instructions et modifie le code source donc le code exécuter pendant la phase de test est différent du code qui sera exécuté lors du fonctionnement nominal.
- Le niveau transactionnel, on simule le fonctionnement des composants en ne considérant que les transactions et on regarde comment réagit le système en cas de faute. On peut simuler des fautes dans les registres et les mémoires, les transactions sur le bus. mais aussi des détails plus centrés sur l'implémentation comme les rejeux, la corruption des registres ou de la mémoire, des erreurs de lectures sur le disque ou les accès que peut effectuer le processeur. L'avantage de cette solution est sa facilité

de mise en place.

- Pour avoir une vision réaliste de l'implémentation il faut se placer en Register Level Transfert (RTL) l'avantage est que l'on a un aperçu réaliste de l'implémentation et des signaux de notre circuit. Cependant cette représentation est seulement appropriée pour une analyse fonctionnelle on ne prend pas en compte les analyses temporelles et de consommation.
- Le niveau supérieur au RTL est celui de la synthèse. Les tests effectués sur celle-ci restent assez bien généralisables à toutes les technologies, car il s'agit d'une étape commune à toutes les implémentations matérielles. L'avantage de cette solution est la généralisation cependant c'est aussi son point faible, car il est difficile de déterminer des timings et des et les consommations instantanées. Et elle reste plus longue que la simulation RTL, mais elle demeure plus proche des implémentations matérielles.
- Il est ensuite possible de réaliser des simulations sur après le placement routage. Dans ce cas, nous avons un modèle correspondant à notre circuit physique. On a donc des valeurs réalistes de timing et de consommation cependant de telles simulations sont beaucoup plus lourdes et les temps de simulations deviennent prohibitifs.
- Il est sinon possible de travailler directement sur un circuit physique qu'il soit ASIC ou FPGA ce qui permet de réduire la durée des campagnes d'injections de fautes. De plus, on a des temps de propagation et de consommation qui sont par définition réalistes. Cependant comme il s'agit de circuit physique nous sommes limités sur les interfaces et sur la connaissance de l'état interne du circuit. Il y a donc un travail important d'instrumentation à réaliser. De plus, de fait de la différence de nature entre un FPGA, routage de LUTs pour simuler les portes logiques, et les ASICS, routage de portes logiques, les résultats d'une campagne de faute sur une technologies ne sont pas directement généralisable à l'autre technologie.

Injection Software

L'injection logicielle permet d'injecter des fautes pendant l'exécution nominale du système (simulation, émulation ou sur le système physique) en utilisant seulement les composants de ce système, dans le cas d'un processeur avec le JTAG, le code source, les mécanismes de DEBUG. La plupart des cas, il s'agit de modifier l'état des mémoires, registres ou RAM, le résultat d'un calcul. On peut injecter les fautes dynamiquement ou statiquement.

- L'injection de faute statique est effectuée lors de la compilation, on modifie le système pour simuler une faute. Cette technique est facilement intégrable dans un flot de conception, car elle ne modifie que la compilation et ne dépend donc pas du système physique.
- L'injection de faute dynamique consiste à ajouter un déclencheur avant l'instruction à fauter. Cette technique se base sur les mécanismes d'exception et d'interruption. À la différence du code statique, on ne modifie pas des instructions, mais on en ajoute.

Cette technique ne nécessite pas de matériel spécifique lors de la campagne d'injection, en effet on utilise seulement les capacités de la cible. L'injection de faute logicielle permettant seulement de se placer au niveau assembleur, elle est principalement utilisé pour caractériser les fautes au niveau de l'application et du système d'exploitation. De plus, la vitesse de fonctionnement pendant l'injection est presque identique au fonctionnement nominal ce qui permet de réaliser des campagnes de grande ampleur. Cependant avec les modifications que l'on apporte au système, on peut avoir des erreurs notamment sur le temps d'exécution.

Injection par simulation

L'injection par simulation implique la construction d'un modèle de simulation. Celui-ci peut être du jeu d'instruction, transactionnel, RTL, post-synthèse ou post ment/routage avec les limitations vues précédemment. Dans l'état de l'art, les principaux outils d'injections de fautes par simulation se basent sur un modèle RTL [**sieh_verify_1997**]. Une faute de la campagne d'injection correspond à une simulation. Il y a deux moyens d'effectuer une campagne d'injection en modifiant le code source ou en utilisant les outils des simulateurs. On peut changer le code VHDL en modifiant les composants pour qu'il soit en exécution fauté, on appelle cela des mutants. On peut aussi ajouter des composants qui seront en charge d'injecter des fautes à l'exécution, ce sont des saboteurs voir figure 2.5. Un mutant est un code source d'un composant qui a été modifié pour qu'à l'exécution celui-ci exécute la faute voulue. L'avantage de cette méthode est que n'importe quel niveau d'abstraction de faute peut être injecté et elle est complètement indépendante du simulateur. Cependant le coût de modification des sources pour chaque faute est très important et il faut ajouter à cela le temps de simulation. Un saboteur est un composant ajouté au RTL dans le seul but d'injecter des fautes. Il est de manière générale inactif, mais il peut modifier le circuit, au niveau des signaux ou des timings, quand un ou des signaux spécifiques sont

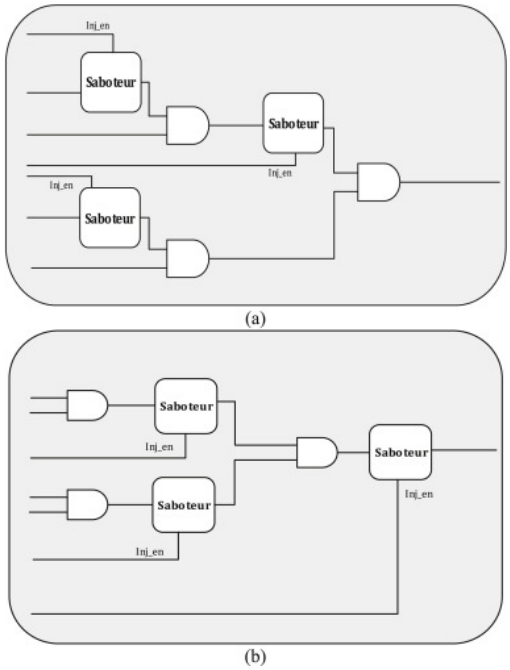

Figure 2.5 : Placement des saboteurs en entrée ou en sortie.

actifs. Ces saboteurs peuvent être ajoutés manuellement ou automatiquement dans le RTL en parallèle ou en séries des composants. Les saboteurs [**naviner_fifa_2011**] permettent de simuler une grande partie des fautes et des conditions environnementales telles que le bruit. Mais comme sa méthode de déclenchement est limitée, elle ne peut modéliser des fautes au niveau de la porte seulement.

La seconde famille se base sur l'injection de faute dans les traces. Il peut s'agir d'un simulateur SPLICE pour les niveaux transistors ou des simulations niveaux RTL, là aussi deux méthodes sont couramment utilisées. Il est soit possible de modifier les signaux, c'est-à-dire déconnecter le signal de son driver puis le forcer à une valeur fixe. Soit on modifier les variables à l'intérieur d'un process. L'injection par les outils de simulation dépend fortement comme son nom l'indique des outils de simulations. Les techniques de simulation sont classées en fonction de trois critères principaux : la capacité à simuler des fautes, l'effort de mise en place et overhead de temps de simulations.

- Les mutants offrent la meilleure capacité à simuler des fautes suivies des mutants
- La manipulation de signal et de variables nécessite peu d'effort à être mise en place, alors que les mutants et les saboteurs nécessitent la création et la génération de nouveaux modèles pour injecter des fautes et la recompilation du RTL. Cependant les

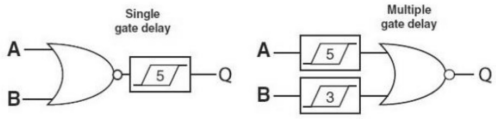

Figure 2.6 : Ajout des informations de delays en entrée ou en sortie de porte.

saboteurs restent plus simples que les mutants.

- L'overhead de temps de simulation pour la manipulation de variable et de signaux est principalement dû au fait d'arrêter et redémarrer la simulation pour effectuer les modifications.

D'autre part pour les mutants et les saboteurs cela dépend de la quantité d'évènements additionnels qui ont été ajoutés, le nombre de lignes de code à exécuter pour chaque évènement et enfin de la complexité du contrôle ajouté.

Injection par émulation

L'injection par simulation regroupe les injections de fautes qui sont spécifiques au FPGA. On ne considérera donc pas les solutions de mutant et de saboteur c'est-à-dire les méthodes de simulations placées sur cible FPGA. L'un des principaux problèmes lorsque l'on veut effectuer des vérifications de résistances aux fautes sur FPGA alors que notre cible est ASIC est le problème des délais. En effet, du fait de la différence entre les deux technologies les délais sont complètement différents ce qui pose des problèmes de généralisations des résultats FPGA sur les cibles ASICS. Pour résoudre ce problème on ajoute des informations de temps [**valderas_set_2007**], mais aussi de consommation [**entrena_set_2009**] pour éviter tous les effets de masquage qui ont lieu lors de l'injection de fautes dans la logique, voir figure **??**.

La reconfiguration dynamique des FPGAs peut être utilisée comme outil d'injection de faute. Pour la première fois introduite en 2000 [**antoni_using_2002**], elle est l'un des champs de simulation des fautes les plus actifs [**fibich_fiji_2019**]. L'avantage de cette technique est d'utiliser les mécanismes internes de reconfiguration pour injecter les fautes. De plus, comme cette reconfiguration ne peut être que partielle, il est d'insérer la faute à un position précise et laisse le reste du circuit tel qu'il est. Ce processus de reconfiguration est bien plus court que celui de la synthèse. Cependant comme on se trouve sur une cible matérielle il peut être difficile de remonter la chaine de conséquence de la faute. Ce que l'on gagne en temps de simulation on le perd en connaissance de notre système.

2.2.4 Modèle de faute

Dans cette partie, il est question du modèle de faute c'est à dire la représentation abstraite permettant de définir l'impact de la faute sur le fonctionnement global d'un circuit que l'on cherche à étudier. Dans cette thèse, la classification choisie est inspirée de celle présentée par [**otto_fault_2005**] dans cette thèse les principaux modèles de fautes sont présentés selon sa classification. Nous utiliserons cette classification qui a le mérite d'être assez claire et complète pour définir nos modèles de faute, car je n'ai pas trouvé de formalisme clair dans les définitions des modèles de fautes. La classification a été légèrement modifiée pour introduire des précisions sur les fautes multiples. Les définitions de faute transitoire, permanente ont été changée pour correspondre aux définitions données en 2.2.1. Différents paramètre sont pertinent à prendre en compte pour caractériser une faute :

- Localité spatiale : Les fautes peuvent être aléatoirement (répartie aléatoirement sur le circuit), localisées (une variable peut être ciblée précisément) et précisent (un octet ou un bit peut être ciblé)
- Localité temporelle : Les fautes peuvent être aléatoirement (répartie aléatoirement sur le circuit), localisées (une variable peut être ciblée précisément) et précisent (un octet ou un bit peut être ciblé).
- Nombre de bits affecté
- Durée : temps pendant laquelle la faute a lieu si cette faute est transitoire ou bien permanente.
- Type de fautes : plusieurs types de fautes peuvent intervenir selon le circuit et la méthode d'injections de fautes. Chacune de ces fautes peut être transitoire ou permanente.
  - Stuck-at fault : la valeur des bits affectés ne changera plus.
  - Bit flip : la valeur des bits affectés prend la valeur complémentaire à l'instant de la faute.
  - Random Fault : la valeur des bits affectés prend une valeur aléatoire.
  - Bit Set : la valeur des bits affectés prend une valeur 1.
  - Bit reset : la valeur des bits affectés prends une valeur 0.

Paramètres	Valeurs possibles
Localité spatiale	
Localité temporelle	
Fautes multiples	
Nombre de bits affecté	
Durée	
Type de fautes	
Probabilité	

Table 2.3 : tableau type d'un modèle de faute.

- La probabilité : avec quelle probabilité une faute peut avoir lieu. Par exemple certaines attaques ont plus de change de SET que de RESET sur les bits notamment les attaques par glitch.
- Localité spatiale des fautes multiples : Cette possibilité est ajouté pour introduire des fautes multiples induite par un évènement commun. Cet ajout caractérise le nombre de bits consécutif affecté lors d'une faute. En effet, lorsqu'une faute a lieu il y a des chances que les transistors proches subissent aussi cette faute.

Pour résumer les principales caractéristiques, des paramètres sont donnés dans le tableau 2.3.

Deux types grandes types de fautes se distinguent celui des soft errors et celui des injection de faute. La principale différence entre les deux est qu'il n'y a pas de soft errors multiple du fait de la nature aléatoire de celle ci. En effet, il est improbable que deux particules chargées tombe en même temps sur la même variable.

Sureté : Soft error

Dans le cas de la sûreté, on cherche à corriger le plus rapidement les fautes qui arrivent. Les fautes sont un cas d'usage normal du circuit et celle-ci se produit aléatoirement sur le circuit. Avec la diminution des circuits, il est devenu probable qu'un rayonnement cosmique produise des fautes multi bits [**zoutendyk_characterization_1989**] [**pagliarini_analyzing_2011**]. Les fautes multiples sont donc proches spatialement, mais cette localité spatiale est aléatoirement répartie sur l'ensemble du circuit. Les caractéristique de tels fautes sont données en tableau 2.4

Localité spatiale	Aléatoire
Localité temporelle	Cycle
Fautes multiples	Non
Nombre de bits affecté	Entre 1 - longueur de la variable affecté
Durée	99.99% Trans 0.01% Perm
Type de fautes	Bit-flip
Probabilité	100%

Table 2.4 : tableau type d'un modèle de faute.

Localité spatiale	Sur les variables sensibles
Localité temporelle	Cycle
Fautes multiples	Jusqu'à 4
Nombre de bits affecté	Entre 1 - longueur de la variable affecté
Durée	transitoire
Type de fautes	Bit-flip
Probabilité	100%

Table 2.5 : tableau type d'un modèle de faute.

Sécurité : Injection de faute

Pour les cas d'injection de faute nous nous plaçons dans le cas de l'attaquant avec des capacités à l'état de l'art. Celui ci possède des capacités d'injection laser multiple (jusque 4) avec une gestion de la puissance de ceux si. Ainsi il est précis au cycle et à la variable près et la variation de puissance du laser il peut plusieurs bits consécutifs. Le tableau 2.5 défini ce genre d'attaquant.

2.2.5 Solution

Les contremesures à implémenter sont destinées à être complètement transparentes pour le logiciel. Il n'est pas exclu d'introduire de nouvelles instructions RISCV pour modifier les niveaux de sécurités, mais cela doit explicitement activer ou désactiver des composants matériels. De plus, seules les contremesures au niveau de la microarchitecture sont étudiées, car facilement adaptable à d'autres designs indépendamment de la technologie d'implémentation. Les méthodes de détections à base de capteur et les solutions aux niveaux de la porte ne sont pas dans notre spectre de solution, mais ces solutions sontp complémentaires au travail microarchitectural à effectuer. Les principales solutions pour renforcer un système contre les attaques par injections de fautes sont basées sur de la redondance. Celle-ci

peut être spatiale, temporelle et informationnelle. Les contremesures sont différentes en fonction du type de circuit à protéger : mémoires ou logique combinatoire. Un point important à prendre en compte est le degré de redondance. En effet, une simple redondance permet de seulement détecter les fautes cependant les redondances de degrés supérieurs permettent à la fois de corriger et de détecter les fautes. Comme expliqué précédemment, il est difficile de classifier le pouvoir détecteur/correcteur d'une solution. Le degré de redondance s'applique donc bien pour la redondance spatiale et temporelle, mais est plus difficilement adaptable pour la redondance d'information.

Redondance par réplication

Les solutions classiques de sécurisations contre les fautes consistent en de la duplication des blocs fonctionnels et de comparer leurs résultats. Quand nous parlons de composants, cela peut très bien être une porte logique d'un étage de pipeline ou le processeur entier. Les solutions proposées peuvent se placer à n'importe quel niveau. La duplication matérielle consiste en la duplication à l'identique des composants. Un doublement avec un comparateur permet de détecter des erreurs. Un duplication, avec le plus souvent un comparateur, permet de détecter toute les erreurs si elle se produit sur une seule des unités, si les deux unités sont fautées des fautes peuvent ne pas être detecté. Si une correction des erreurs est désirée, il faut non plus une duplication, mais un triplement des unités fonctionnelles et il faut un vote pour décider si la donnée est valide, corrigible ou non corrigible. Cependant là aussi, bien que cette solution protège bien contre les fautes aléatoires, il est très peu probable de produire une faute valide, pour cela deux fautes touchent deux des redondances au même endroit. Mais dans le cas d'un attaquant, il devient réalisable d'effectuer une faute valide avec deux fautes bien placées. Il faut tenir compte du fait que chaque ajout matériel pour sécuriser contre les fautes est lui-même sensible contre les fautes pour résoudre ce problème. La solution la plus simple est de dupliquer aussi les voteurs. Ces solutions sont appelées Full TMR, car aussi bien le composant que la vérification est triplé. Ces solutions sont souvent utilisées dans le domaine de la fiabilité, car les fautes étant uniformément répartie les chances que deux fautes se produisent sur deux redondances différentes est très faibles. Cependant dans le cadre de la sécurité un attaquant peut injecter fautes identiques sur les redondances.

Vient ensuite la duplication temporelle qui consiste à effectuer plusieurs fois de suite la même opération et voir si le résultat diffère. De la même manière que le doublement matériel une duplication permet seulement la détection et le triplement la correction. Comme

dit précédemment, il est assez facile d'attaquer ce type de protection, par exemple sur des crypto processeurs. En effet, une faute assez longue va affecter les deux itérations successives de la même manière et entraîner une faute valide.

redondance information

On trouve enfin la redondance d'information, qui consiste à ajouter de l'information complémentaire à la donnée pour pouvoir détecter ou corriger les fautes. Les principales solutions sont les codes détecteurs et correcteurs d'erreurs. Cependant très peu sont adapté au contexte hardware et encore moins aux exigences du pipeline. Les plus utilisés sont les codes résidus, le code berger et enfin la parité. L'avantage de ces codes est qu'ils ne sont pas une simple duplication, il est donc plus difficile d'identifier la zone à fauter. De plus, avec certains codes correcteurs il m'est pas possible d'injecter des erreurs mono bit car elles seront nécessairement détecté. Le principal avantage des codes correcteurs est qu'ils sont en général plus légers qu'une duplication, de plus ils induisent moins de fuites par canaux auxiliaires. Les principaux codes détecteurs et correcteurs utilisés dans le matériel sont : la parité, Cyclic redundancy check (CRC), code de hamming, code BCH, code SEC-DED, code Red-Salomon, Reed-muller et les low density Parity Code (LDPC).

Offuscation

Enfin, une des dernières grandes catégories de protection contre les injections de faute est l'obfuscation. On trouve dans les cryptoprocesseurs l'ajout de cycles factices c'est-à-dire des cycles où le processeur effectuer des opérations sans rapport avec l'exécution en cours. On peut aussi mélanger la donnée pour éviter que l'attaquant ne sache où il doit injecter les fautes. Elles doivent donc obligatoirement être combinées avec d'autre technique. De plus, si ces contremesures ne sont pas aléatoires, la difficulté de l'attaque est rendue plus difficile seulement le temps d'identifier la constante ajoutée, si celle-ci est aléatoire la démarche devra être répétée à chaque attaque. Ces méthodes sont efficaces pour compliquer la tâche d'injection, mais elles ne sont pas une protection en soi.

2.3 Conclusion

Ce chapitre a présenté une vision d'ensemble des attaques par observation et des attaques par perturbation mais aussi les concepts généraux des contre-mesures. Les attaques par

canaux auxiliaires sont complexes et plurielles et de nombreux vecteur de fuite et d'exploitation de ces fuites sont possible. Cependant un travail théorique important a été produit pour évaluer les masquage et de nombreuses méthodes empiriques sont proposé pour évaluer les autres méthodes. Dans le cas de notre thèse nous nous concentrerons sur les contremesures de randomisation et d'ajout de bruit.

Les attaques par injections de fautes sont elles aussi pluriels, mais dans leur cas peu de vérifications formelles et de métriques permet l'évaluation des contremesure. De plus, il est difficile d'assurer à la fois la sécurité et de la sûreté d'un composant. Même si cette thèse se concentrera principalement sur la sécurité mais des piste de réflexions et d'implantation seront proposés pour combiner sûreté et sécurité dans les cas des fautes sur les circuits électroniques

Enfin, en regardant, en detail les contremesures possibles contre ces attaques on remarque qu'elle sont en partie antinomique. Les contremesures de masquages pour les canaux auxiliaires offre plus de possibilités de faute et les redondances des contremesures contre les fautes créées de nouvelles fuites.

3

Chemin de donnée

3.1 attaques possibles

3.1.1 Attaques en fautes

3.1.2 Attaques par canaux auxilaires

3.2 contre-mesures existantes

3.2.1 Masquage

3.2.2 processeur durci

3.3 Tag homomorphique

3.4 Présentation de la contre-mesure

Dans cette section nous allons présenter la réflexion qui nous a amené à choisir la permutation comme opération de redondance, et ensuite comment rendre une permutation dépendante d'une clé en facilitant les opérations complexes

3.4.1 Choix de la permutation

Pour montrer que les calculs sur les données ont bien été effectués sans faute dans le CPU, des calculs équivalents peuvent être faits sur leurs tags d'authenticité. En partant du principe que toute fonction de logique combinatoire ne peut être réalisée qu'avec des portes NAND (La porte NOR a la même propriété), nous avons cherché des fonctions qui présentent un homomorphisme par rapport à cette porte logique. Pour rappel, une fonction sera dite homomorphe par rapport à un certaine opération op s'il existe un opération op' qui vérifie pour tout x et y :

$$f(x\,op\,y) = f(x)\,op'\,f(y)$$

Pour l'opération logique NAND, on peut plus prosaïquement chercher $op = op' = NAND$. On réalise alors que la vérification des calculs entre $x\,op\,y$ et $f(x)\,op'\,f(y)$ devient avantageuse car il suffit d'appliquer f sur $x\,op\,y$ et de vérifier l'égalité avec $f(x)\,op'\,f(y)$. Seulement deux catégories de fonctions mathématiques permettre de réaliser cet homomorphisme sur des données de plusieurs bits : il s'agit soit des projections p définies par $p^2 = Id$ soit

des permutations bit à bit définies par $p^2 = p$. Les projections manquent d'intérêt pour générer des tags d'intégrité car elles font perdre par nature de l'information sur les données.

Mais la permutation malgré sa difficulté inhérente à être implémentée en logique binaire est-elle une bonne candidate pour faire un tag d'authenticité ? Il est à noter que toute attaque par injection de fautes est détectable par une permutation. Cependant il est a priori impossible de déterminer la bonne permutation à utiliser avant que l'attaque ne se produise. D'autre part, avec l'émergence des architectures open source, il est de plus en plus probable que l'attaquant ait une connaissance avancée du système et peut ainsi préparer des scenarii d'injections de fautes qui permettraient se passer outre les redondances. Même si l'attaquant connait parfaitement l'architecture, il ne pourra prédire l'aléatoire. Ainsi en ajoutant de l'aléatoire on complexifie l'attaque d'un facteur irréductible du point de vue de l'attaquant. Ce facteur irréductible nous permet de mettre en place une notion de résilience et de réponse à la menace dans le temps qui nous ai impartie.

La permutation utilisée a alors été conçue pour dépendre d'une clé secrète tout en conservant des propriétés intéressantes pour être également homomorphe par rapport aux opérations arithmétiques et pas seulement logiques. Des permutations entre 2 bits seulement (que nous appellerons transposition) ou entre deux blocs de 2^k bits ont été retenues. Celles-ci peuvent être réalisées à partir de la porte de Fredkin très utilisée en reversible computing et en quantum comuting, qui au triplet (a, b, c) associe le couple $(a.\bar{c} + b.c, a.c + b.\bar{c})$ où $\bar{c}$ est le complémentaire de c. En d'autre terme, cette porte renvoie (a, b) vers (a, b) si le bit de clé c vaut 0 et vers la permutation bit à bit (b, a) si le bit de clé c vaut 1. Le bit c joue alors le rôle de la clé de la permutation.

Cette porte a la propriété de fuir peu la clé c lors d'attaque side-channel du fait d'une dépendance symétrique de la porte par rapport à c et $\bar{c}$. Ce schéma de "chiffrement" n'est toutefois pas très robuste, car en ayant quelques valeurs de texte chiffré il est aisément possible de comprendre comment les bits sont permutés et donc de remonter à la clé secrète. Cette technique est donc réservée au fonctionnement interne du CPU et ne devra pas être utilisée dans la hiérarchie mémoire où des mémoires peuvent être externes avec des données chiffrées facilement extractibles. Néanmoins, tout comme pour le M&M, il est possible de masquer les données et la donnée permutée. Ainsi au lieu d'avoir $(x, p_\alpha(x))$ où α est la clé secrète de la permutation p_α, on travaille avec (x XOR $m, p_\alpha(x)$ XOR m'). Ainsi ces données masquées peuvent très bien se retrouver en mémoire et être récupérées par un attaquant, le masque empêchera ce dernier de retrouver la donnée, la permutation et

sa clé. Il faut bien sûr que les masques soient aléatoires. Il est également à remarquer que p_α étant homomorphe au XOR, on a $(x \text{ XOR } m, p_\alpha(x) \text{ XOR } m') = (x \text{ XOR } m, p_\alpha(x \text{ XOR } m'')$ où $m' = p_\alpha(m'')$. D'autre part, dans le cas d'une implémentation masquée l'opération ne s'effectue que sur un seul share du masque. Il n'y a donc pas de besoin de modifier les opérations sur une donnée masquée. Ainsi cette opération de permutation ne réduit pas la sécurité face aux attaques par canaux auxiliaire et un attaquant possédant des capacités d'exploitation des fuites par canaux auxiliaires ne peut pas extraire facilement la clé de permutation. Revoir ces 2 dernières phrases. opération ?

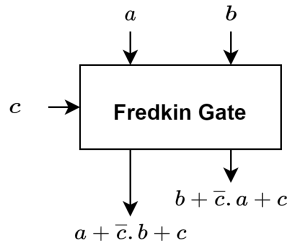

Figure 3.1 : Fredkin gate.

3.4.2 Permutation par bloc

Il y a de nombreuses manières d'effectuer des permutations dépendantes d'une clé. Cependant, certaines sont plus adaptées pour faciliter les opérations complexes telles que les décalages, additions et multiplications au prix de quelques concessions au niveau de l'entropie de la permutation. La solution retenue a été celle d'un arbre à structure dichotomique. Cet arbre se base sur une division en bloc de notre donnée, si on prend une donnée 32 bits on peut la diviser en 2 blocs de 16, ces deux blocs de 16 peuvent aussi diviser en 4 blocs de 8 bits. On peut continuer ainsi jusqu'à avoir des blocs de 1 bit. Avec cette division en bloc on peut utiliser la porte de Fredkin pour effectuer une transposition de deux blocs de même taille selon une clé comme montrer en figure 3.2. Une transposition T_K^n échange le contenu des blocs de taille n si $K = 1$. On ne transpose que deux blocs qui sont consécutifs sans chevauchement. C'est-à-dire pour un mot de 32 bits. On a besoin d'1 bit de clé pour transposer les 2 blocs de 16 bits, de 2 bits pour les 4 blocs de 16 bits, 4 bits pour les 8 blocs de 4 bits, etc. Ainsi la clé est la concaténation des différentes clés de permutation $Key = Key_0^{16}|Key_0^8|Key_1^8|Key_0^4|...|Key_3^4|Key_0^2|...|Key_7^2|Key_0^1|...|Key_{15}^1$ qui a une taille de 31 bits pour une permutation de 32 bits P_{Key}^{32}. Elle est composée de 5 niveaux de transpositions et au total de 31 transpositions qui sont composées en commençant par les blocs les plus grands $P_{Key}^{32} = T_{Key_0^{16}}^{16} o T_{Key_0^8}^8 o T_{Key_1^8}^8 o T_{Key_0^4}^4 o ... o T_{Key_3^4}^4 o T_{Key_0^2}^2 o ... o T_{Key_7^2}^2 o T_{Key_0^1}^1 o ... o T_{Key_{15}^1}^1$

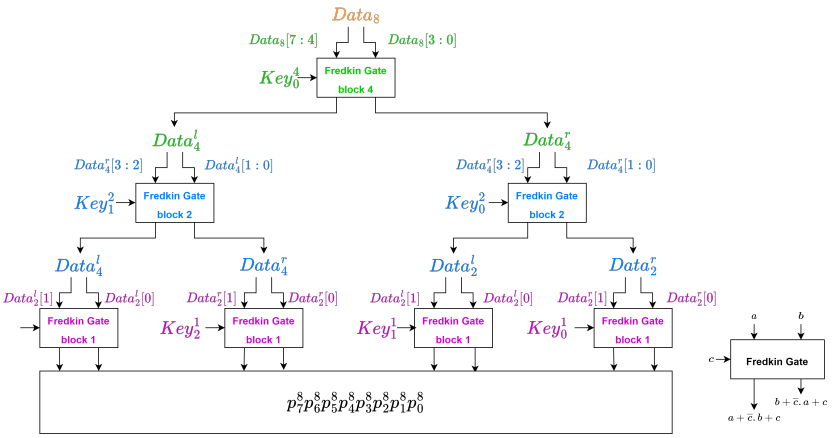

Figure 3.2 : Block permutation.

comme défini en Figure 3.3. Chaque sortie de transposition du niveau n est utilisée comme entrée de la transposition suivante de niveau $n/2$. Les transpositions d'un niveau donné peuvent être faite dans n'importe quel sens sachant que les chevauchement dans un niveau n'ont pas été permis. L'avantage et la faiblesse de cette permutation sont que cette dernière conserve une localité par blocs. Ainsi les n bits composants un bloc sont nécessairement dans le même bloc en sortie de permutation. Cette propriété offre plus de connaissances à l'attaquant, mais permet d'effectuer plus facilement les opérations complexes telles que les décalages et les additions.

3.5 Opération logiques et arithmétiques

3.5.1 Opérations booléennes

Dans un processeur généraliste, l'ALU réalise des opérations booléennes, logiques et arithmétiques. Pour avoir un processeur résistant face aux fautes, il faut que ces opérations puissent être effectuées avec des opérandes qui ont été permutés comme défini dans 3.4.2. En effet, pour protéger les opérations arithmétiques d'un processeur, il faut réussir à réaliser ces mêmes opérations dans le domaine de la redondance. La principale difficulté dans notre cas est que cette représentation est une permutation des bits selon une clé. Le choix de la permutation comme composée de transposition de blocs assure que toutes les opérations booléennes bit à bit comme AND, OR, NOT, NAND, NOR et XOR peuvent être effectuées à la fois sur les données et sur les tags d'authenticité sans changer d'opérateur. Reste alors à traiter les trois opérations les plus courantes qui sont les décalages, l'addition

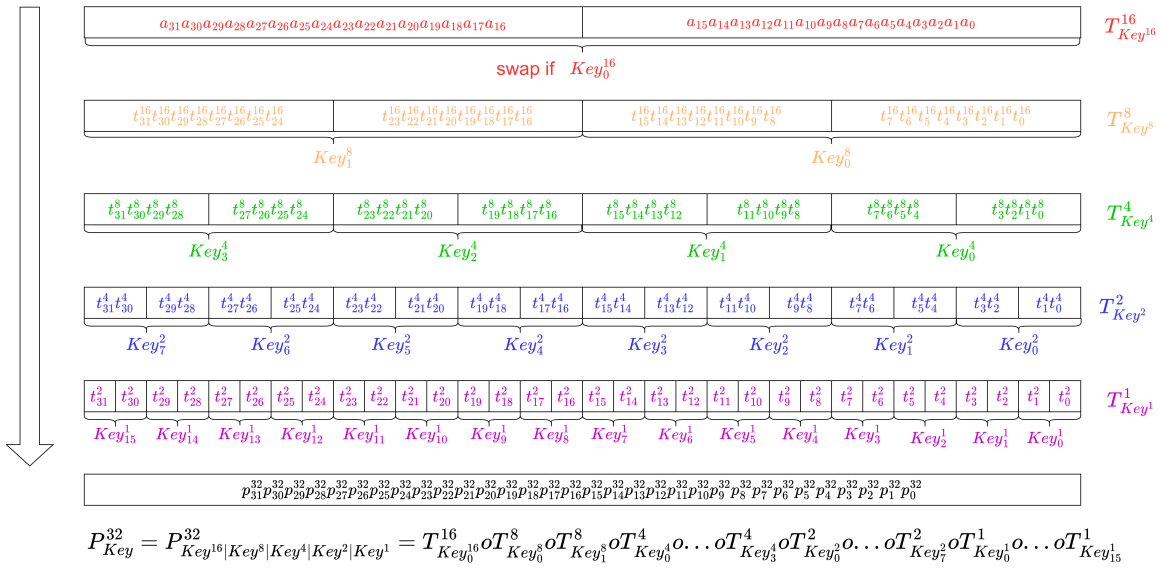

Figure 3.3 : Block Permutation.

et la multiplication.

3.5.2 Décalage

La première opération à réaliser est le décalage. La manière classique d'implémenter un décalage est de combiner des opérations de décalage. Cette combinaison se réalise avec une structure comme donnée en Figure 3.4. La sortie du décalage 2^n est propagé si et seulement un flag $Shift_{(2^n)}$ est présent. Ainsi pour réaliser un opérateur de décalages il faut savoir faire des décalages de 2^n.

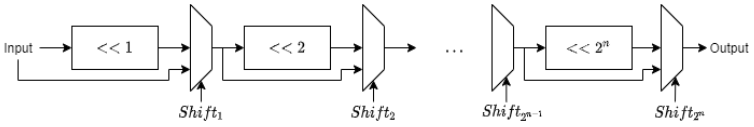

Figure 3.4 : opérateur de décalage.

Décalage à gauche de 1 bit Dans un premier temps, prenons le décalage à gauche de 1 bit. De par la division en bloc de notre permutation, il est possible d'effectuer le décalage à l'intérieur des blocs de taille 2. En effet, cette structure que tous les bits dans un bloc de 2 bits sont consécutifs, il faut donc inverse la position du bit de poids faible et propager le bit

de poids fort dans l'arborescence. Cette propagation se fait à l'aide de la clé. La Figure 3.5 présente la solution retenue pour effectuer un décalage de 1 bit à gauche sur un vecteur de 4 bits avec en sortie cette donnée décalée toujours permuter avec la même clé. On a donc en entrée une donnée $a_3a_2a_1a_0$ permute avec une clé de valeur 110 donc $Key^2 = 1$ et $Key^1 = 10$.

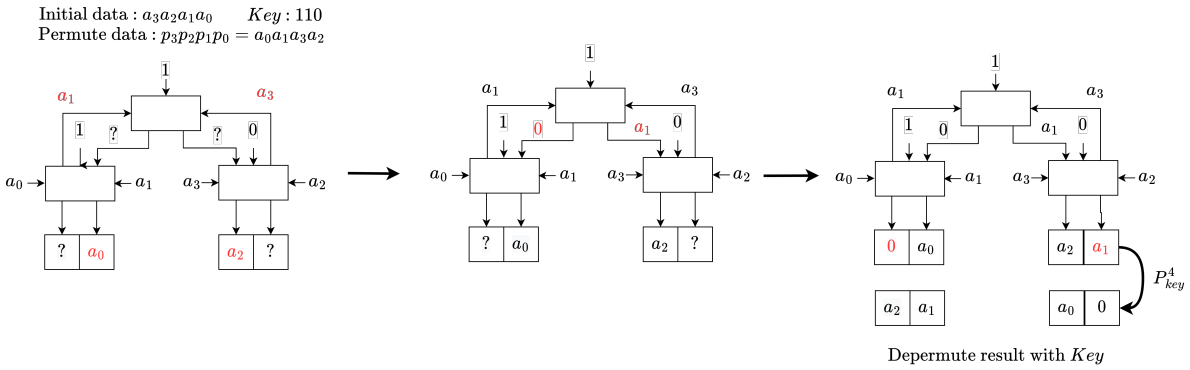

FIGURE 3.5 : Explication du décalage.

Cependant, il faut réussir à router les retenues pour remplir les zéros laissés libres par le décalage. Si on regarde plus en détail la Figure 3.5, elle montre la propagation des retenues. Dans un premier temps, le décalage dans les blocs de deux bits est réalisé et les retenues sont propagées aux étages supérieurs. Une fois aux étages supérieurs on répète l'opération d'inversion, si on arrive au dernier niveau il faut dans les cas de décalage logique propager un 0 à travers notre schéma. La propagation du zéro à droite ou à gauche est déterminée par la clé, la logique est que si la clé est à zéro les retenues sont dans le bon ordre et il faut donc propager le zéro vers la droite, dans le cas où la clé est égale à 1 les retenues sont inversées et il faut donc propage le zéro sur la gauche. Une fois les retenues propagées vers les blocs inférieurs, celle-ci sont routées vers la sortie laissée libre. La sortie est donc toujours bien permutée selon la même clé que la donnée initiale.

Décalage à gauche de 2^n bit Les décalages par les puissances de 2 sont plus difficiles à réaliser, car on peut facilement effectuer des décalages de blocs avec un schéma analogue au décalage de 1. Ce décalage déplaçait des blocs de 1 bit, le décalage de 2^n prends des blocs de 2^n bits. Dans le cas P_{key^8} à décalage de 2 aura une profondeur de 2 et un décalage de 4 aura une profondeur de 1. Cependant bien que les blocs permutés soient à la bonne place, les permutations à l'intérieur des blocs ne correspondent plus à la clé de permutation. Pour résoudre, il faut pouvoir comparer la permutation du bloc avec permutation en cours pour

remettre les blocs dans le bon ordre. Une solution est proposée en Figure 3.6, la clé en cours Key_n^m et la clé selon laquelle le bloc est permuté $K_{D_n}^m$. La partie supérieure (blocs blanc et gris clair) du schéma représente le décalage de la donnée, cette partie est analogue à la Figure 3.5. Au lieu d'être avec une donnée de 4 bits, nous avons ici une donnée de 32 bit et un décalage de 8 bits. Nous n'inversons plus deux bits entre eux, mais des blocs de 8bits. À la fin, de cette partie, nous avons bien décalé nos blocs de 8 bits. Il est à noter que les valeurs à l'intérieur des blocs ne correspondent pas à la permutation suivant la clé. Dans un deuxième temps (blocs noirs), il faut réordonner l'intérieur des blocs pour les faire correspondre avec la permutation en cours. Pour cela il faut donc comparer Key_n^m la clé de l'emplacement actuel du bloc et $K\_D_n^m$ la clé de l'emplacement précédent du bloc, s'ils sont différents, le bloc est inverse sinon il est laissé dans sa configuration d'entrée. L'opération est répétée pour tous les blocs constituant le bloc permuté.

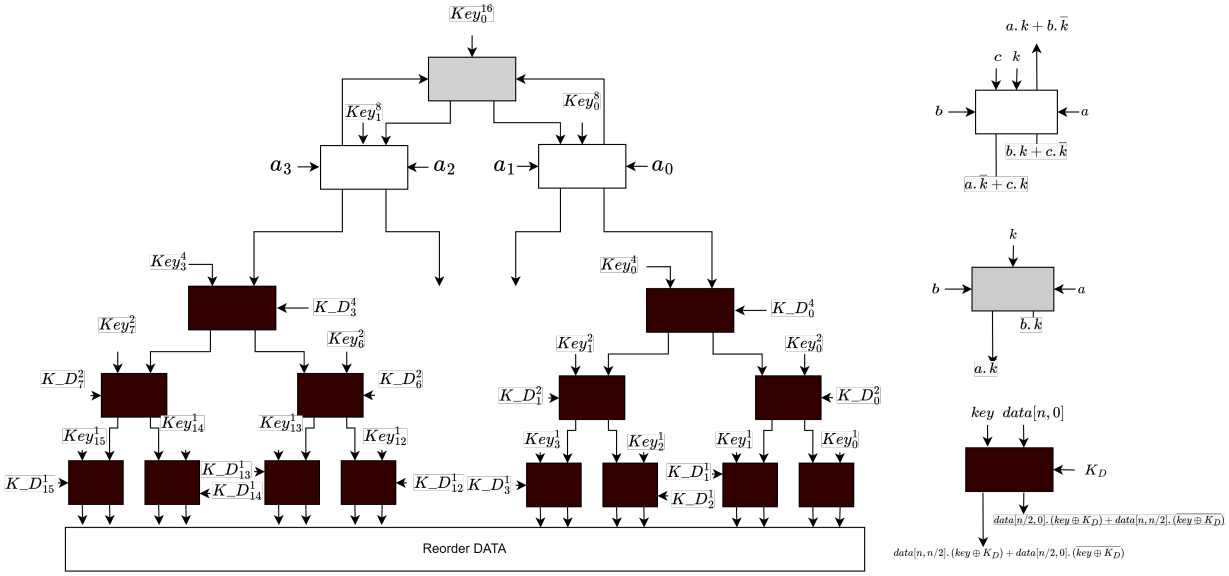

FIGURE 3.6 : Shift data.

Nous avons ainsi en fin de schéma un bloc décalé de 8 bits et permuté suivant la clé d'entrée. Cependant, pour effectuer le réordonnancement des blocs, il est nécessaire de récupérer les permutations de chacun des blocs. Pour cela, on utilise un schéma analogue au décalage de 1. Cette partie est représentée dans la partie supérieure de la Figure 3.7, qui représente un décalage de 8 dans un contexte de permutation P_{key^32}, cependant les entrées Key_i sont la clé contenant toutes les sous clés qui dépendent de ce bloc par exemple : $key_0^4 key_1^2 key_0^2 key_3^1 key_2^1 key_1^1 key_0^1$ La définition des blocs blanc et gris clair de ce schéma est assez proche du schéma précédent sauf qu'il ne faut pas prendre la sous-clé relative relative

à l'étage en cours en sortie de bloc. Ils ne sont pas explicitement écrits, car il complexifie inutilement le schéma. Enfin, a partie inférieure est la mise à jour des permutations internes au bloc en fonction de la clé en cours. En effet, chaque différence entre la clé en cours et la clé du bloc entraîne une permutation sur les clés des blocs le constituant. Donc s'il y a une différence entre la clé en cours et la clé du bloc on inverse les positions des clés des blocs de constituant le bloc.

On peut ensuite réaliser tous les décalages possibles avec la combinaison des décalages de 2^n. Pour chacun des décalages l'équilibrage du nombre d'étages entre la permutation des blocs et le réordonnancement interne des blocs sera différent, mais la profondeur globale du décalage sera toujours de $log_2(len)$ avec len la taille de la donnée.

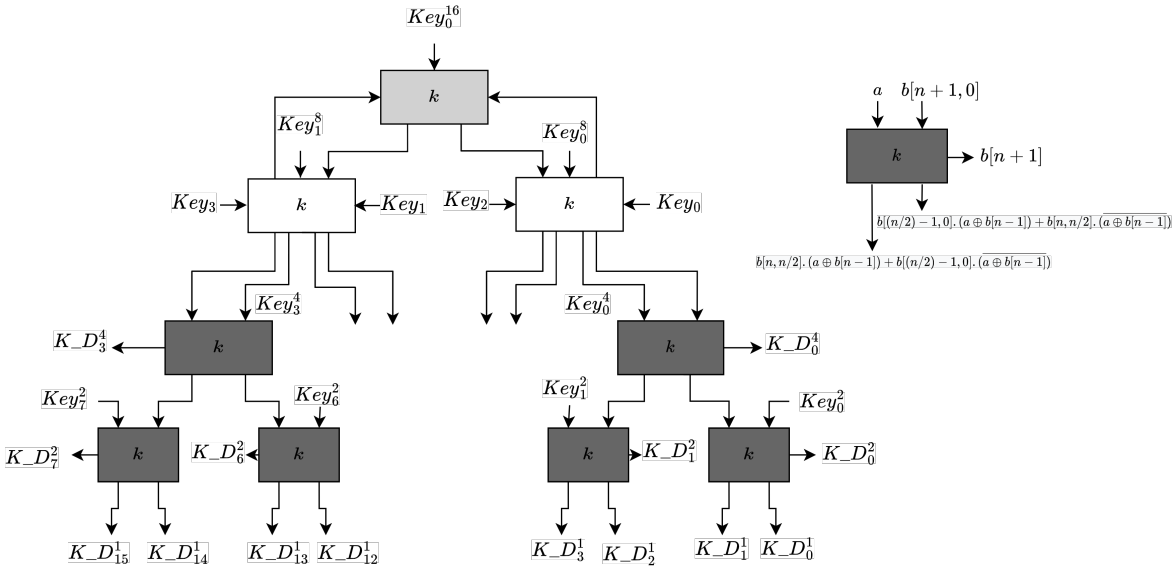

FIGURE 3.7 : Find old Key.

Autres décalages Le décalage à droite consiste à changer le sens de toutes les permutations effectuées lors du décalage à gauche. On remarque que ces changements sont seulement une prise du complémentaire de la clé. En effet, prendre le complémentaire de la clé effectue toutes les transpositions dans l'autre sens. Le dernier type de décalage restant est la rotation, là aussi seul le premier bloc doit être modifié. En effet, il n'y a plus de notion de mise à zéro ou de remplacement de bit, les bits arrivant en haut de l'architecture doivent obligatoirement être inverses, car ce qui sort de la rotation doit être remis dans son entrée. Ainsi l'inversion est inconditionnelle et ne dépend donc pas de la clé.

Si on veut maintenant réaliser un décalage arithmétique à droite, le décalage arithmétique à gauche est équivalent au décalage logique à gauche. Pour effectuer ce décalage, il faut d'abord retrouver le bit de poids de fort non permuté. Il est possible de le trouver rapidement par une sélection des sous-blocs qui le contient. Une fois que nous avons le bit de poids fort, nous pouvons réaliser ce décalage. La seule différence réside dans la duplication du bit de poids fort dans les places laissées libres par le décalage. Ces places libres sont créées dans le premier bloc l'architecture, le bloc gris clair, il faut donc à ce moment remplacer la mise à zéro par le bit de poids fort P_f. Tous les blocs sont identiques, car le bit de poids forts est celui qui se trouve tout à gauche donc tous les blocs internes ce bit doit aller à gauche quand la clé est à zéro. Ce schéma reprend simplement la séquence de bloc qui permet de retrouver l'emplacement du bit, pour le bit de poids fort il est toujours à gauche c'est pour cela que tous les blocs sont identiques.

Masquage Il n'y a pas de problème pour effectuer le masquage de ce procédé de décalage. En effet, il suffit de masquer l'opération de décalage, car elle s'effectue de manière indépendante entre les différents shares.

3.5.3 Addition

La seconde opération à modifier est l'addition, la principale difficulté est la propagation de la retenue. En effet, celle-ci se propage de proche en proche entre les bits, on remarque du fait de la division en bloc de 2^n bits que cette permutation est particulièrement adaptée aux structures en arbre dichotomique, comme le décalage présenté précédemment. Des structures de propagation dichotomique rapide de propagation de la retenue existent déjà et c'est le cas du carry look ahead.

Carry look ahead Le principe de cet additionneur est d'anticiper à chaque étage la retenue avant de calculer la somme. Le fait d'obtenir une retenue sortante à un étage d'addition est dû à la génération et la propagation de la retenue. On peut les retrouver dans l'expression pour la retenue d'indice i on a :

$$c_{i+1} = (a_i.b_i).(a_i + b_i).c_i$$

. Ainsi

- $G_i = (a_i.b_i)$ est appelé le générateur de la retenue

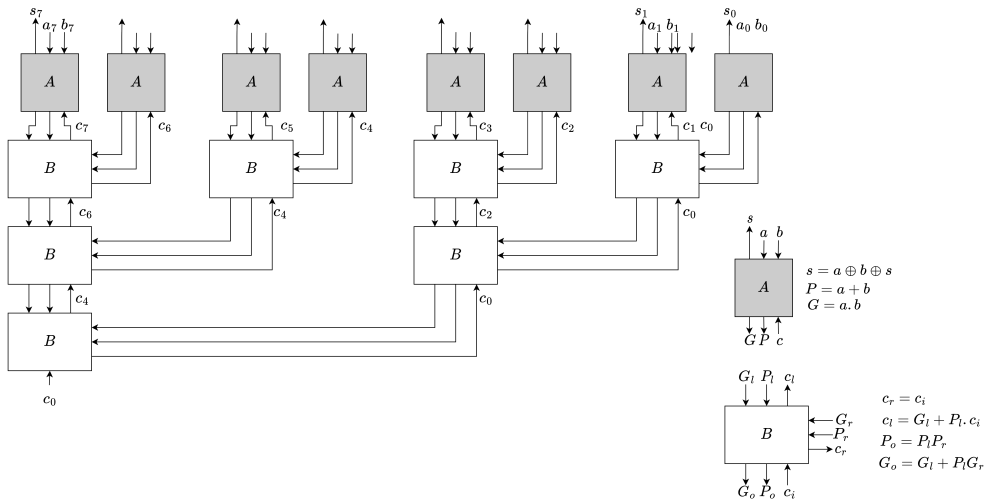

FIGURE 3.8 : Carry look ahead.

- $P_i = (a_i + b_i)$ est appelé le propagateur de la retenue

Calculer pour chaque position de chiffre si cette position va propager une retenue si elle arrive de la droite. Combiner ces valeurs calculées pour pouvoir déduire rapidement si, pour chaque groupe de chiffres, ce groupe va propager une retenue qui vient de la droite. Ainsi on peut construire la Figure 3.8 en décomposant la propagation sous forme d'un arbre dichotomique qui permet de réduire la profondeur logique de la propagation de la retenue

Gestion de la permutation Ce type d'additionneur permet une propagation rapide de la retenue en se basant sur l'architecture look ahead, il suffit donc d'ajouter à ce schéma la gestion de notre permutation. Cet additionneur modifié est donné en Figure 3.9. Ce type d'additionneur, de la même manière que le décalage possède une rétropropagation, des éléments P et G permettant de calculer la retenue. Seul le calcul de G est dépendant du fait que l'entrée vienne de droite ou de gauche, le choix s'effectuant toujours selon la clé K. Avec la propagation de P,G, il est possible de calculer les retenues. Les retenues doivent se propager vers la gauche ou la droite en fonction de la clé, quand $K = 0$ la retenue de poids le plus faible part à droite sinon si $K = 1$ elle part à gauche.

Gestion de la permutation Pour masquer l'additionneur il est a noter toute les operation avec une clé k n'ont pas besoin d'etre

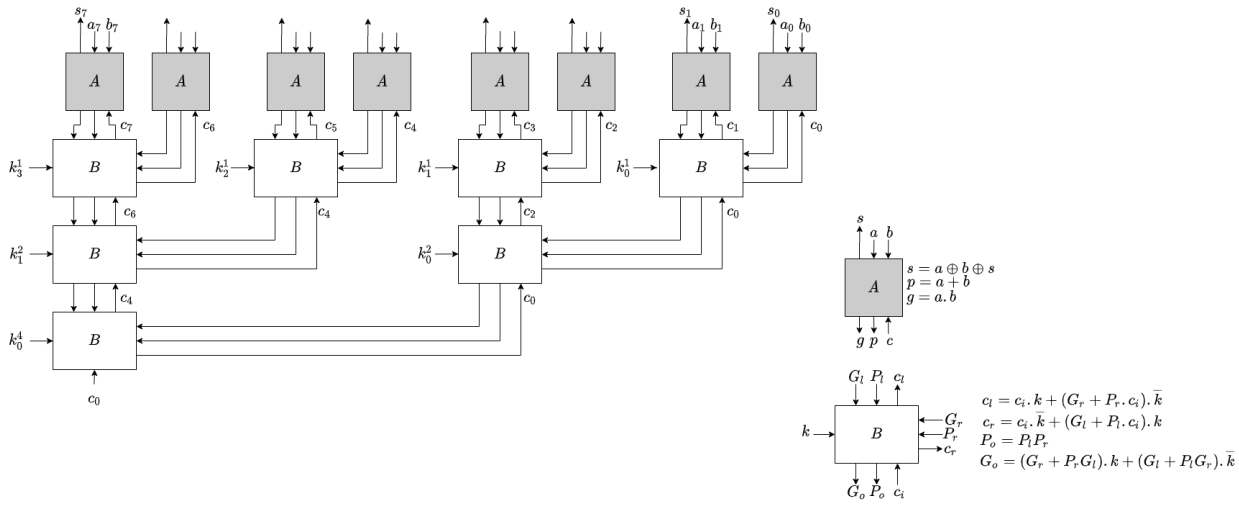

Figure 3.9 : Permute look ahead.

3.5.4 Multiplication

La dernière opération à réaliser est la multiplication. Plusieurs méthodes de multiplications sont possibles. La méthode itérative est la plus simple, mais demande un nombre important de cycles. Cependant, différentes optimisations sont possibles et permettent de réduire ce nombre de cycles

Méthode itérative La méthode itérative est la méthode analogue à une multiplication posée elle qui consiste en une succession d'addition avec le multiplicande qui est décalé à chaque étape, figure 3.10. À chaque étape, on ajoute le produit partiel a la somme finale. Ainsi il faut 32 cycles pour réaliser une multiplication sur 32 bits. Avec cette solution, il faut seulement réaliser des décalages et des additions. Pour une réalisation matérielle de ce type

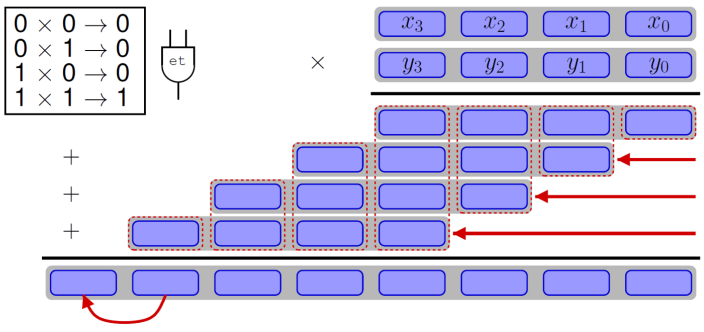

Figure 3.10 : Multiplication binaire.

de composant, très peu d'éléments sont nécessaires : 1 registre 32 bits, un additionneur et 1 registre à décalage de 64 bits. La structure générale est décrite en figure 3.11. On considère une multiplication sur n bits. À l'initialisation on ajoute le multiplicande dans le registre X et le multiplieur dans le registre Y_0 ensuite a chaque cycle d'horloge si le bit de poids faible $Y_0[0]$ du registre Y_0 :

- est égale a 1 on ajoute le multiplicande a $Y1$ on stock le résultat dans Y_1 et le débordement dans C
- est égale a 0 on ne fait rien

On réalise ensuite un décalage à droite de sur les 32 bits (C concaténer à Y_1 et Y_2). Après n cycle les bits de poids forts (most significant bits MSB) sont disponible dans le registre Y_1 et les bits de poids faibles (Less significant bits LSB) dans Y_0. Pour rendre cette descrip-

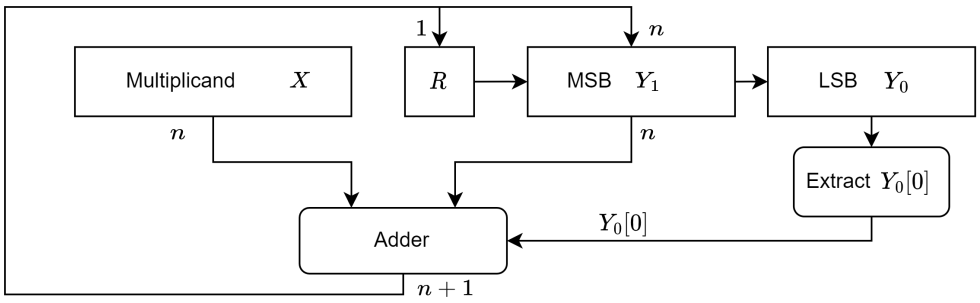

Figure 3.11 : Description matérielle d'une multiplication itérative sur n bits.

tion compatible avec les permutations, peu de changement doit être mené. Il faut d'abord remplacer l'additionneur par celui décrit en 3.5.3. De plus, le registre à décalage doit être remplacé par un décalage par celui présenté en 3.5.2. Cependant, il faut ajouter quelques subtilités à cette implémentation. En effet, il ne s'agit pas de décalage classique ou l'on met un zéro pendant le décalage et il faut aussi réussir à sortir le bit de poids faible de MSB. Il faut modifier le bloc au sommet de l'arbre. Pour insérer la retenue en début de décalage, il faut ajouter une entrée et la propager entre les différentes branches de l'arbre. Pour sortir le bit de poids faible, il faut utiliser une structure en arbre qui sélectionne à l'aide de la clé la branche où se situe le bit de poids faible. Le bloc ainsi modifié est donné en figure 3.12

Optimisation de Booth Il est possible de réduire le nombre de cycles nécessaires pour la multiplication en utilisant la méthode de Booth. L'algorithme de Booth donne une méthode pour multiplier les entiers binaires dans la représentation du complément à 2 signé. Il se

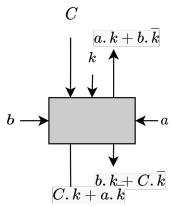

FIGURE 3.12 : Bloc de décalage pour la multiplication itérative.

base sur le fait que les 0 dans le multiplicateur ne nécessitent aucune addition, mais juste un décalage. Un 1 dans le multiplicateur du poids de bit 2^k au poids 2^m peut être traité comme $2^{(k+1)}$ à 2^m. Comme dans tous les schémas de multiplication, l'algorithme nécessite l'examen des bits multiplicateurs et le décalage du produit partiel. Avant le décalage, le multiplicande peut être ajouté ou soustrait au produit partiel ou laissé inchangé selon les règles suivantes :

- Le multiplicande est soustrait du produit partiel lorsqu'il rencontre le premier 1 le moins significatif d'une suite de 1 dans le multiplicateur.
- Le multiplicande est ajouté au produit partiel lors de la rencontre du premier 0 (à condition qu'il y ait eu un 1 précédent) dans une suite de 0 dans le multiplicateur.
- Le produit partiel ne change pas lorsque le bit multiplicateur est identique au bit multiplicateur précédent.

Cependant, si l'addition au cycle n est suivi d'une soustraction au cycle $n+1$ cela est équivalent à une soustraction au cycle n et ne rien faire au cycle $n+1$. Réciproquement quand une soustraction est suivie d'une addition, cela est équivalent à une addition au cycle n. Ainsi on est assuré que la moitié des bits sont à zéro. Ainsi la multiplication peut être réalisée en 16 cycles. Pour effectuer cette optimisation, il faut donc connaître les deux bits de poids faible ainsi que le bit précédent. L'implémentation matérielle est donnée en figure 3.13. La principale différence est le calcul du complément à deux sur la donnée permutée. On effectue une inversion et un additionneur permuté. Pour obtenir les deux bits de poids faible, il faut qu'on utilise là aussi une structure qui sélectionne les bonnes branches. Il faut cependant les réordonner une fois qu'on les a obtenus. De plus, le bit Y_{-1} est retenu en sortie de décalage du cycle précédent qui est maintenant de deux bits vers la droite. Enfin l'adder sélectionne s'il doit faire une addition, une soustraction ou ne rien faire.

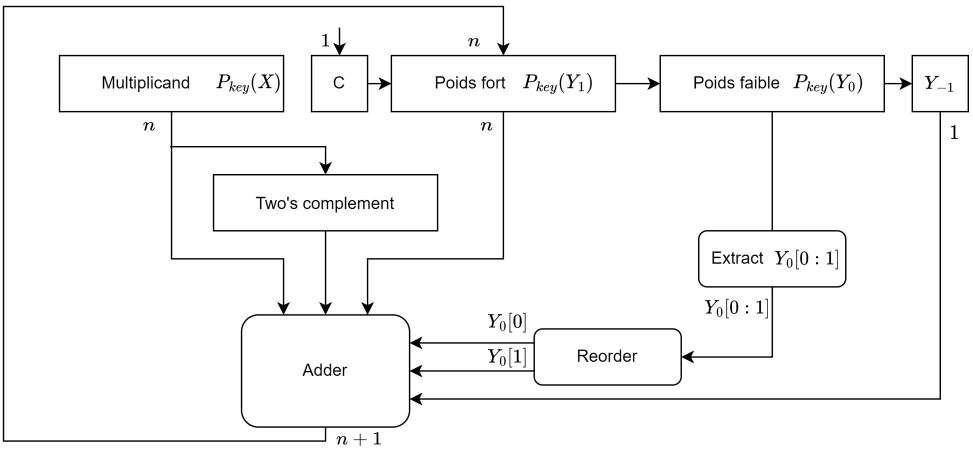

Figure 3.13 : Multiplication de booth avec des données permutées

Optimisation possible Il est possible de trouver d'autre méthode d'optimisation pour encore réduire le nombre de cycles, comme du Booth à radix plus élevé, mais aussi des méthodes de réduction rapide des sous-produits à l'aide d'arbre de dada. Les méthodes avec les arbres de dada posent le problème de calculer tous les sous-produits et donc réaliser beaucoup de décalage ce qui a un coup important avec notre solution. Cependant, des structures de décomposition dichotomique à l'aide de la méthode de Karatsuba peuvent être une piste intéressante pour faire des multiplications rapides avec des données permutées.

3.6 Mise en place dans un cœur RISCV

Un CPU comprend plusieurs étages de calculs qui sont pipelinés pour gagner en performances temporelles. Les données avant de subir des modifications sont chargées dans un banc de registres. La donnée x est stockée dans le banc de registres sous la forme $(x, p\alpha(x))$. Chaque registre à cette structure, ainsi un processeur 32 bits manipulera des données sur 64 bits sous la forme $(x, p\alpha(x))$. L'architecture où l'on a ajouté notre contremesure est un cœur RISC-V à 4 étages à exécution dans l'ordre (CV32E40P), un schéma de l'architecture est donné en figure 3.14.

Les instructions arrivent dans l'étage de FETCH sous la forme (x), mais on peut aussi choisir de les protéger même si d'autres solutions semblent plus indiquées comme expliquer plus tôt. Les données arrivent sous cette forme $(x, p\alpha(x))$ au niveau de l'interface mémoire pour être chargées dans des registres et éventuellement subir des calculs effectués par

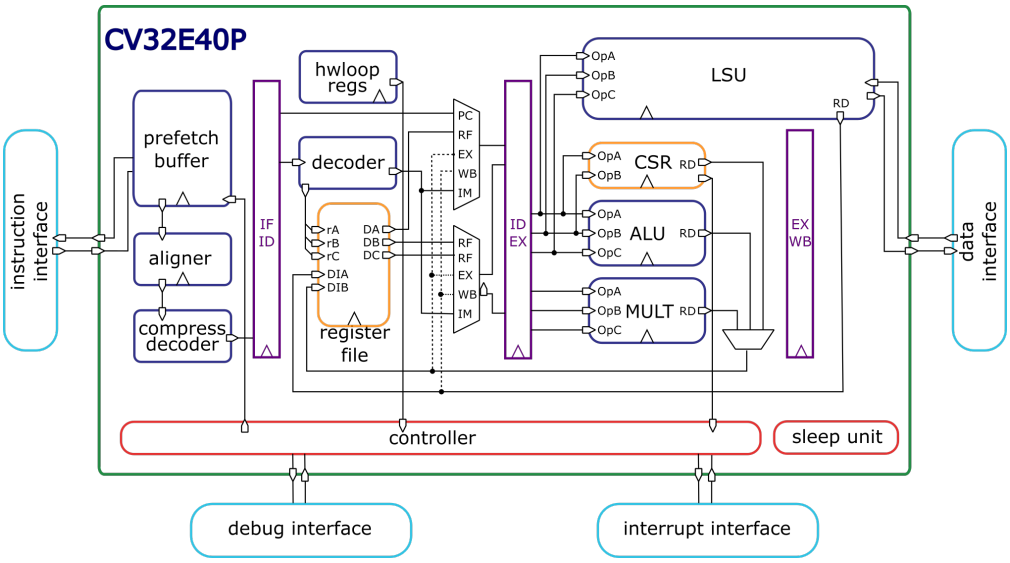

Figure 3.14 : Architecture du cœur CV32E40P

l'ALU. Il y a différentes sortes de calculs possibles définis par des instructions : des instructions purement logiques (AND, NAND, NOR, OR, XOR), des décalages (SHIFT), des additions et soustractions (ADD, SUB), des multiplications et divisions (MUL, DIV). Les calculs sur x et $p\alpha(x)$ sont effectués en parallèle de sorte qu'à la fin des calculs l'on puisse comparer les résultats en effectuant une permutation d'un côté ou de l'autre.

3.6.1 Modification du chemin de donnée

Cette contre-mesure a été implémentée dans un cœur RISC-V. La permutation a été ajoutée à l'entrée du cœur. Ensuite, l'ensemble des registres interétages ainsi que le banc de registre pour stocker les données permutées ont été permutés. Ensuite, nous ajoutons 4 points de vérifications.

- 3 ports de lectures du banc de registres (équivalent à la sortie de l'étage DECOD)
- 2 ports d'écritures du banc de registres (équivalent à la sortie de l'étage EXEC)
- Entré de l'étage EXEC
- Sortie de la LSU

Du fait que l'on ajoute le tag d'intégrité à l'entrée du processeur, il n'est pas nécessaire de vérifier les entrées de celle-ci. De manière générale, la donnée doit être vérifiée en entrée et en sortie de chaque étage où est manipulée la donnée. Dans le cas, des registres en cas

de lectures et d'écriture de ces registres. En cas de différence, une exception spécifique est levée. Il est important de noter que cette exception ne peut être levée exclusivement en cas de faute, que ce soit l'œuvre d'une particule chargée ou d'un attaquant. Au-delà des modifications du cœur en lui-même, il a aussi fallu apporte des modifications aux contre-mesures proposées. En effet, les décalages et additions dans le CV32E40P ont des modes vectoriels ainsi il faut que l'additionneur et le décalage puissent gérer des données de 32 bits, mais aussi des 2 blocs de 16 bits et 4 blocs de 8 bits. Notre solution séparant de manière dichotomique notre donnée permet de faire ce genre d'opérations sur les données permutées. Ainsi en empêchant la propagation à un certain point précis dans nos arbres d'addiction et de décalage on a pu implémenter ces opérateurs.

En terme de surcoût matériel induit par ces modifications Voir les comparaison pour les branchements/breanchement conditionnel

3.6.2 Masquage

Choix du masquage Le pipeline d'un processeur ayant des contraintes élevées de timing et dont le nombre d'étages est une variable importante, surtout dans les cas des processeurs moins évolués sans prédicteur de branchement efficace. Il est important de trouver des masquages à faible latence et n'allongeant pas le chemin critique. Le masquage à faible latence est un champ d'étude assez récente, la première occurrence date de 2016, même si des masquages à faible latence existaient antérieurement il ne mettait pas ce point en avant. Le masquage a faible latence est recherché en tant que tel pour la première fois par Moradi et al. dans [**DBLP:conf/asiacrypt/0001S16**], où les auteurs ont considéré des méthodologies de conception asynchrones pour réduire la latence des d'implémentation threshold de premier ordre. La première approche générique pour la conception de circuits S-box masqués à faible latence (d'ordre supérieur) a été présentée avec GLM par Gross et. al. en 2018 [**DBLP:journals/tches/GrossIB18**]. L'idée principale de GLM est de sauter l'étape de compression des parts après chaque opération non linéaire, ce qui élimine le besoin d'étapes de registre au prix d'une augmentation du nombre de shares, en particulier dans le cas d'une opération non linéaire. Cependant cette augmentation des shares. LMDPL [**DBLP:conf/ches/LeisersonMW14**] est une des premières méthodes a faible latence générique, mais il a fallu attendre pour [**DBLP:journals/tches/SasdrichBHM20**] pour en prouver la composabilité et la résistance au glitch, mais aussi a un cycle réduire la latence dans le cas de l'AES. Bien que cette technique puisse surpasser considérablement le GLM dans certains scénarios, elle ne peut offrir qu'une sécurité de premier ordre

et s'accompagne de cycles de précharge explicites Plus récemment, Arribas et al ont présenté une technique de masquage à faible latence basée sur des implémentations threshold, appelée LLTI, qui a des exigences de surface comparables à celles du GLM, mais qui peut éliminer le besoin d'aléatoire en ligne [**DBLP:journals/tifs/ArribasZN21**]. Enfin SESYM [**DBLP:journals/tches/NagpalGPM22**] propose d'applique des méthodes de logique asynchrone pour réduire la latence, pour cela ils suppriment les registres de synchronisation du masquage DOM à l'aide de porte de Muller Le choix des portes sur LMDPL, car il offre de nombreux avantages. Il est composable, il n'a besoin que d'un random frais par porte non linéaire, permet une implémentation légère. Comparé à SESYM, on peut ajouter facilement des étages pour attendre la fréquence voulue sans problème. De plus, le dual Rail dans LMDPL permet de dissimulation (hidng) elle n'est pas nécessaire au critère de sécurité se qui offre des perspectives de réduction du surcout matériel alors que LMDPL est déjà parmi les contremesures avec le surcout el plus faible. Enfin, se plaçant au niveau de la porte logique et ne nécessitant pas de contrainte de placement et de prote spécifique, elle offre la perspective de générer automatique la fonction masquée au niveau de la synthèse.

Nombre de masque Malgré notre choix du masquage LMDPL, il nous reste deux possibilités de masquages : utiliser un masque différent pour la donnée et sa duplication ou utiliser le même masque. Utiliser le même masque apporte plus d'avantages. Tout d'abord, cela permet de ne stocker qu'un seul masque, réduisant l'impact sur le surcout matériel de la solution, et d'autre part la vérification est facilitée. En effet, utiliser deux masques induit une vérification en plusieurs cycles pour éviter les glitchs et elle devient plus couteuse, car il faut dépermuter deux éléments au lieu d'un. Le problème lorsqu'un seul masque est utilisé est lors des opérations arithmétiques, l'utilisation de nombre aléatoire frais vont modifier le masque de la permutation et de la donnée différemment entrainant un masque de sortie de l'opération arithmétique différent. Pour les opérations booléennes vu que notre permutation est homomorphe sur ce type d'opérateur, il n'y a pas de logique supplémentaire entre une opération standard et une opération permutée. Il est donc possible de s'assurer que les nombres aléatoires frais soient les mêmes pour la donnée et sa permutation. En effet, on peut prévoir où doivent aller les bits d'aléa en utilisant une structure en arbre dépendant de la clé. Cependant dans les cas plus complexes, où la permutation ajoute des portes à masquer à l'opérateur comme cela est le cas de l'addition, alors il y a un nombre différant de nombres aléatoires frais entre l'opérateur standard et celui permuté. Inévitablement, les masques de sortie sont différents entre les deux versions. Dans ces cas, il

faut effectuer un transmasquage. Pour l'effectuer, il faut deux cycles si on veut éviter les glitchs. Le premier cycle permet de xorer les masques ensemble. Le deuxième cycle permet de xorer ce masque sur la donnée permutée. Cela ajoute un cycle aux opérations tel que l'addition ou la multiplication.

3.6.3 Changement de clé

Une de principale difficulté à utiliser ce genre de permutation dans un processeur est de gérer le changement de clé de permutation. En effet, pour assurer un niveau de sécurité suffisant, il faut la changer à intervalle régulier, mais aussi après qu'une attaque par injection de fautes ait été menée. La levée d'exception permet d'avoir une détection purement matérielle qui identifie de manière manifeste une faute physique. En effet, aucune autre explication ne peut expliquer cette levée d'exception. Il faut cependant distinguer deux cas.

le premier est qu'une faute est détectée en dehors des registres alors on peut suivre la procédure de changement de clé suivante

1. Sauvegarder les registres et le PC
2. Changer la clé de permutation
3. Vider les parties de la hiérarchie mémoire qui contiennent des données permutées
4. Recharger les registres et le PC

Pour les applications nécessitant des contraintes de sécurités forte, par exemple des algorithmes de cryptographie, il est possible de changer la clé à intervalles réguliers en respectant les étapes précédentes.

Cependant, un cas plus difficile à gérer est celui d'une faute détectée au niveau du banc de registres alors on ne peut pas savoir quand a eu lieu la faute. Elle a pu avoir lieu entre le cycle d'entrée et le cycle de sortie de la donnée. On est certes assuré que la hiérarchie mémoire n'est pas fautée, car la faute a eu lieu entre le cycle d'entrée et le cycle de sortie de la donnée dans le banc de registre. Il est cependant difficile de définir un état sain vers lequelle restaurer les valeurs du banc de registre. Le plus simple est de laisser le système d'exploitation gérer ces cas, en relançant par exemple le processus. Une autre solution est d'ajouter de la correction d'erreur pour retrouver l'état correct du registre. Il est cependant à prendre en compte deux cas de faute, s'il s'agit d'un évènement isolé, le système d'ex-

ploitation doit considérer qu'il s'agit d'un évènement aléatoire par exemple dû à l'impact d'une particule chargée et donc une correction d'erreur est la bonne réponse à incident. En revanche, s'il y a des erreurs successives, il s'agit plutôt d'une attaque et il faut mettre le système dans un état de sécurité. Au vu de la complexité de la réponse à incident entre les deux cas, c'est au système d'exploitation de gérer ces cas. L'avantage de la permutation est qu'elle permet de garder d'autres méthodes de détection et de corrections d'erreurs, car il s'agit seulement de permutation. Cette correction peut être obtenue facilement et à bas coût en ajoutant de la parité par blocs à la permutation. La permutation permet de détecter la faute, la parité elle de faire le choix de la redondance à sélectionner. Ce genre de solution permet de combiner la sécurité et la sureté, la sureté est assurée par le mécanisme de correction de l'erreur, la sécurité quant à elle par l'aléatoire ajouté par la permutation qui dépend d'une clé. Si aucune exception n'est levée, un processus en fin de calcul est assuré que le résultat n'est pas fauté et qu'il est à l'origine de ce résultat.

3.7 Securité

Dans cette section nous allons étudier la sécurité de notre solution face aux attaques par canaux auxiliaire et contre les attaques en fautes. Comme dit en introduisant notre modèle d'attaquant, dans le cas des fautes stochastiques notre solution n'apporte pas de sécurité supérieur à une duplication.

3.7.1 sécurité contre les side channel

Dans cette section nous allons etudier la securité de notre solution face aux attaques par canaux auxiliaire Comme dit en introduisant notre modèle d'attaquant, dans le cas des fautes stocastiques notre solution n'apporte pas

3.7.2 securité contre les side channel

En adhérant à des principes de sécurité tels que l'assurance de la non-complétude [BGN+14] et du rafraîchissement adéquat, la sécurité de l'ACS est satisfaite partout. l'actualisation correcte est satisfaite partout, M&M hérite de la sécurité SCA du mécanisme partagé de mécanisme de multiplication et d'inversion partagé utilisé. Un schéma de masquage booléen qui est sûr dans le modèle d'attaquant SCA considéré. modèle d'attaquant SCA considéré fournit donc une sécurité contre le d th-order SCA. Puisque le modèle peut

inclure ou exclure les défaillances matérielles si le mécanisme de multiplication et d'inversion partagé utilisé est également sûr en présence de défaillances, M&M en hérite. Le calcul des partages de balises suit les mêmes principes de conception que les calculs de partage de valeurs. Les deux chemins de données fonctionnent de manière totalement indépendante l'un de l'autre et reçoivent leur propre caractère aléatoire frais et distinct (voir la figure 7). Il est important de noter que les partages d'entrée ρ et τ p doivent également être indépendants, ce qui est facilement réalisable si les masquages initiaux de p et τ sont obtenus séparément. L'indépendance des deux chemins de données assure que leur fusion dans le bloc imparfait n'induit pas de fuite sur p ou τ Le gadget de rafraîchissement. Il est important que tout mécanisme de rafraîchissement utilisé (cf. 4.1.1) assure la même sécurité que celle fournie par le schéma de masquage utilisé. Le type de rafraîchissement dépend donc de l'ordre de sécurité visé d [BBP+16] et du modèle d'attaquant considéré. En général, on peut toujours utiliser le gadget de rafraîchissement basé sur la multiplication d'Ishai et al [ISW03]. Il a été montré dans [BBD+16, Gadget 4 b] que ce rafraîchissement garantit la composabilité à tout ordre. Pour un niveau de sécurité cible spécifique, l'aléatoire peut être consommé plus efficacement. Par exemple, l'approche de rafraîchissement en anneau de [CRB+16] utilise Lauren De Meyer, Victor Arribas, Svetla Nikova, Ventzislav Nikov et Vincent Rijmen13 d + 1 masques frais de manière circulaire pour rafraîchir d + 1 parts. Cette méthode suffit pour la sécurité de deuxième et troisième ordre. À certains ordres supérieurs, on peut utiliser sa variante, le rafraîchissement par décalage, qui n'utilise toujours que d + 1 unités d'aléa frais, mais qui effectue une rotation avec un décalage de plus de 1 [BBD+18, Alg. 2]. Enfin, le rafraîchissement additif utilisant seulement d masques frais est suffisant lorsque la sécurité du premier ordre est visée. Pour un traitement plus détaillé du rafraîchissement gadgets de rafraîchissement, nous renvoyons à [BBD+18].

3.7.3 securite contre les attaques en fautes

securité sur les registres

Il faut le code le plus long possible pour assurer une couverture de faute maximal, la difference avec les codes plus complexe est d'assurer un detection minimum en distance de hamming ce qui n'est pas utile. pour la securite on cherche plitot a minimiser le fait de tomber sur un autre mot valide du code. Les codes correcteurs classique ne cherche pas ce genre de propriete. ce qui pose probleme dans le cas de faute multiple. Il existe toujours une permutation pour un faute donné qui arrive sur un mot du code pour tomber

en dehors de ce mot du code Pour evaluer la securité contre les attaques par injections de fautes - Il existe toujours une permutation pour contrer les modèles d'attaquants - Au pire a l'équivalent d'une duplication

logique combinatoire

Étendre les résultats des registres

3.8 registres dynamiques

L'écriture dans le banc de registres est un point discriminant qui peut permettre d'identifier des instructions factices. Pour cela, on écrit seulement dans les registres qui ne sont pas utilisés par le flux normal d'exécution. Il n'y a pas de changement à effectuer pour les lectures car seul les écritures ont une influence sur l'exécution légitime. Pour resoudre le probleme des écritures on a choisi d'écrire dans les registres qui ne sont pas utilisé. Cependant tous les registres sont susceptible d'être utilisé il faut donc prévoir deux registre de destination supplémentaire, il y a deux voies d'écritures de registre dans le processeur CV32E40P, cependant écrire toujours dans les même registres est identifiable il est possible d'utiliser un banc de registre dynamique comme on peut en trouver dans [18]. Cette idée est venu naturellement dans le processus de reflexion, c'est apres une recherche d'antecedent dans la vue de breveté cette invention que nous sommes tombé sur cet article.

Pour déterminer quel registre est utilisé, il faut maintenir une table de validité des registres. Cette table de validité consiste à ajouter un bit de validité à chaque registre du banc de registre. Un registre est considéré comme valide quand celui-ci est écrit. Il est invalidé lorsqu'un registre est la destination d'une ecriture. Il est maintenant possible de déterminer la validité des registres, la structure en arbre décrit en Figure 2 permet de choisir dans quel registre invalide l'instruction factice va écrire. Elle permet de tirer un registre aléatoirement parmi les registres libres. Cependant, il doit toujours y avoir au moins un registre libre. C'est le nombre aléatoire R qui détermine la sortie Out. Si un des registres n'est pas invalide, la sortie est l'autre registre. Le cas où deux registres sont invalides n'est pas problématique, car il a toujours un registre invalide donc on restera dans les branches de l'arbre où au moins un registre est valide. Ce composant est appelé random select module (RS module). Pour notre structure en arbre, il faut généraliser ce composant en sélectionnant la bonne branche à chaque étage de l'arbre. Pour cela on place N/2 RS modules avec N le nombre d'entrées. Le choix entre les N/2 modules est donné par les RS modules des étages

plus hauts dans l'arbre. Le schéma de ce composant généralisé est donné en Figure 4. L'index du registre écrit par l'instruction factice est donné par la concaténation des sorties des différents RS module.

Dans notre cas nous avons jusqu'à deux écriture dans les registres par cycle grâce à cette structure en arbre nous pouvons prendre l'inverse de la sélection pour avoir un autre registre libre. Cependant pour gérer le cas où le banc de registre est complet, il nous faut en permanence 2 registres vides. Le deuxième avantage est un plus grand choix lors de l'écriture car quand le banc de registre est plein nous avons 2 choix parmi trois pour écrire dans un nouveau registre. Alors quand sans ces registres supplémentaires il n'y a plus de variabilité. Enfin dans notre cas d'instruction factice il faut empêcher l'invalidation du registre à écrire et écrire dans un registre libre sans le mettre dans l'état valide. Donc dans le cas d'une écriture dans les registres la seule différence perceptible du point de vue de l'attaquant est dans la gestion des bits de validité. Pour l'écriture en mémoire nous sommes resté sur une solutions plus simple car il est difficile de définir des espaces non utilise dans la mémoire, le plus simple est de les allouer nous même, pour des raisons de simplicité nous avons alloué une adresse spécifique pour les écritures et les lectures factice en mémoire.

3.9 masquage

Le masquage est la solution privilégié contre les attaques par canaux auxiliaires car il permet de rompre la dépendance des fuite avec les données manipulées en plusieurs parties. Ainsi il devient plus difficile de faire une attaques car toute il faut combiner toute les parties pour recuperer de l'information.

3.9.1 Exigence du masquage que l'on recherche dans un pipeline

Au dela du masquage il y a des contrainte propre a un pipeline de processeur. Il s'agit d'un architecture qui est deja pipeline et possède un latence assez importante et il serait judicieux d'augmenter un minimum cette latence. De plus, de nombreuse unité arithmétique différente sont présente

3.9.2 Masquage intéressant

3.9.3 LMDPL

Decomposition en polynome

Discussion

3.9.4 Proposition d'implémentation

3.10 Conclusion

Dans ce chapitre nous avons vu les differnentes solutions misent en place pour proteger les données dans le chemin de données d'un processeur generaliste avec l'aide de tag homomorphique mais aussi de masque au premier ordre a faible latence. De plus, le banc de registre a été rendu non deterministe. Avec ces contremesures nous couvrons la majorité des attaques identifié dans l'état de l'art. Cependant meme si les données sont protegées il reste le probleme du chemin de controle.

4

Chemin d'instruction et de contrôle

Ce chapitre se concentre sur les attaques puis les contremesures développées pour protéger le chemin de contrôle du pipeline. Les protections apportées se divisent en deux contributions. La première se concentre sur la protection contre les attaques par canaux auxiliaires et par injections de faute sue les instructions jusqu'à leur décodage en limitant le surcout à son maximum. La seconde est de proposer une nouvelle méthode de vérification d'intégrité des données de contrôles en proposant d'utiliser la parité pour tirer avantage des limites des méthodes d'injections de fautes.

4.1 Attaques possibles

Les objectifs de l'attaquant par canaux auxiliaires et par injection de faute sur le chemin d'instructions est de compromettre la confidentialité et l'intégrité du code exécuté par le dispositif. D'autres attaques sont possible mais sont en dehors de notre modèle d'attaquant, on peux par exemple citer les attaques par cold boot [**DBLP:conf/IEEEares/GruhnM13**] de même que les attaques logicielles, tels que l'injection de code avec par exemple l'écrasement de l'adresse de retour sur la pile. Pour répondre à ce type de menace des méthodes de chiffrement et de control flow sont les plus indiqués. Les applications embarquées nécessitent du matériel à la fois léger et sécurisé afin de préserver la confidentialité et l'intégrité du code exécuté. Les attaques physiques et par observation se développent contre les processeurs destinés à l'IOT et les solutions proposées ont souvent un surcout important. Le chemin de donnée, en particulier, est sensible aux attaques par injections de fautes, car c'est le centre de la gestion des instructions. En effet, les injections de fautes sont particulièrement efficaces sur les instructions, notamment les fautes par saut d'instruction [**DBLP:conf/dtis/MenuDPRD20**; **DBLP:conf/nordsec/DutertreRPR19**] qui permettent d'outrepasser de nombreuses sécurités algorithmiques. Ces attaques permettent aussi de modifier les instructions pour soit changer le flux d'exécution ou le résultat de calcul. Le changement dans le flux d'exécution se concentre notamment sur la modification de prise de branchement ou de modification d'instruction pour pouvoir effectuer des sauts dans le code. Mais ce ne sont pas les seules menaces qui sont identifiées. La propriété de code source peut aussi être remise en cause avec par exemple du désassemblage par canaux auxiliaire[**DBLP:conf/cardis/CristianiLH19**].

4.2 Contre-mesures existantes

Pour assurer une sécurité des instructions face aux menaces précédentes, quatre propriétés doivent être respectées :

- L'instruction en cours est bien précédée par l'instruction précédente dans le flow d'exécution. C'est-à-dire éviter les sauts et les modifications d'instructions.
- Protection de l'étage DECOD, le décodage doit se dérouler sans fautes.
- Protection des branchements, le processeur doit exécuter la bonne branche

- La valeur de chaque bit de l'instruction ne doit pas être directement accessible

Dans l'état de l'art, ces quatre propriétés ne sont jamais traitées efficacement en même temps. Les solutions les plus courantes pour protéger le flux d'exécution sont les unités d'intégrité du flux de contrôle (CFI) [**DBLP:journals/corr/ClercqV17**] qui peuvent garantir l'intégrité des instructions jusqu'aux premiers étages du pipeline [**DBLP:conf/eurosp/WernerUSM18**] et peuvent éviter les fuites par des canaux secondaires dans la hiérarchie de la mémoire si des méthodes de sont mises en œuvre [**DBLP:conf/dsd/SavryEH20**]. Cependant, ces contre-mesures ne protègent pas le décodage des instructions et donc le branchement. Ce type de contre-mesure présente un surcoût important, car il s'attaque à d'autres vecteurs d'attaque sur le flux d'exécution que les attaques physiques. Dans les processeurs fiables, comme Klessydra [**DBLP:conf/applepies/BlasiVCMMO19**], nous trouvons une duplication spatiale de l'étage DECOD pour sécuriser le décodage. De plus, la duplication temporelle permet d'éviter le saut d'instruction [**DBLP:journals/jce/MoroHER14**]. Le principal problème des solutions basées sur la duplication est le surcoût matériel ou temporel. Enfin, pour contrer les attaques par canaux auxiliaires, les solutions de masquage sont les plus utilisées. Cependant, les coûts d'implémentation sont importants, notamment avec la duplication de la taille des instructions pour stocker le masque.

4.3 SECDEC

Notre contribution est de donner deux solutions proches, mais avec des implications d'implémentation différentes. Elles assurent la sécurité du chemin d'instruction contre les attaques par perturbation et observation. La première solution a un surcoût matériel et de taille de mémoire d'instruction négligeable, mais induit une dépendance du code compilé avec des contraintes microarchitecturales. La deuxième solution est libre de ces contraintes, mais implique une redondance de décodage. Ces solutions conduisent à des modifications dans le backend du compilateur, mais elles sont simples à mettre en œuvre et sont adaptables à différents types de compilation.

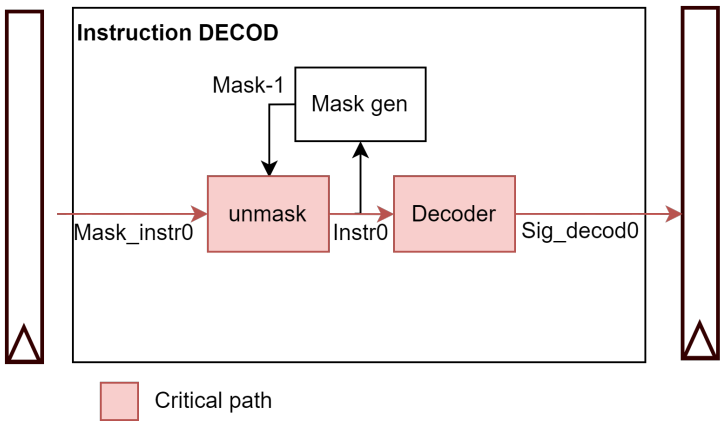

Figure 4.1 : Pré-décodeur.

4.3.1 Présentation de fonctionnement

Description de la contremesure

Cette contribution cherche a apporter à adresser les quatre propriétés de sécurité énoncées précédemment en utilisant les signaux générés par l'instruction précédente pour masquer l'instruction courante. Il s'agit d'un masquage booléen non aléatoire, car il dépend de l'instruction précédente. Pour ce faire, on génère un masque au cycle N qui sera utilisé dans l'étape de décodage (DECOD) du cycle N+1 pour démasquer l'instruction entrante. On peut choisir de prendre les signaux avant le décodeur, voir Figure 4.1, ce sera l'instruction elle-même qui servira à la génération de masque. Nous appelons cette solution pré-décodeur.

Il est également possible de prendre les signaux après le décodeur, voir Figure 4.2, cette solution est appelée post-décodeur.

L'instruction doit donc être masquée lorsqu'elle arrive dans l'étape DECOD. Pour appliquer ce masque, des passes de compilation supplémentaires doivent être ajoutées dans le backend du compilateur. En conséquence, toutes les instructions dépendent de l'instruction qui la précède. Ainsi, si un défaut survient au cycle N, le démasquage de l'instruction dans l'étape de décodage au cycle N+1 sera différent du décodage sans défaut. Cette solution permet donc de vérifier que l'instruction présente est exécutée après l'instruction qui la précède dans le code machine, on évite donc les sauts d'instructions et les modifications d'instructions. Lors d'un branchement, deux chemins sont possibles. Ainsi, il est courant d'essayer lors d'une campagne d'injection de fautes de faire prendre au processeur la mauvaise branche pour exécuter du code non désiré. La solution proposée permet

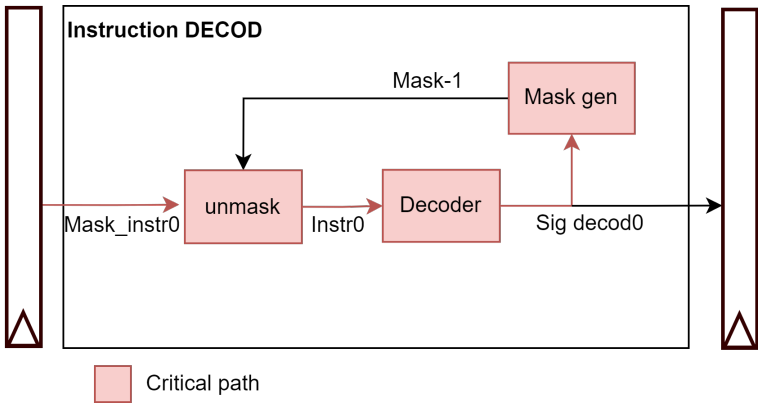

Figure 4.2 : Post-décoder.

de différencier les deux branches en leur attribuant des masques différents en fonction de la branche prise et ce en fonction des signaux de décodage. Ainsi, contrairement au CFI existant qui sont au moins sensible lors du décodage de l'instruction, la validité du branchement est assurée pendant l'ensemble du chemin d'instruction.

Le choix définitif de la branche est déterminé lorsque les conditions de branchement sont vérifiées. Le choix est donc effectué lorsque le résultat de la condition est disponible, ce qui est normalement le cas à la fin de l'étage EXEC. La seule façon de faire en sorte qu'une branche prenne la mauvaise branche est de fausser l'étage EXEC. Ce cas est géré par les tags d'intégrités présenté en 3.3.

La détection se fait sur les erreurs de décodage induites par cette faute. Une faute sur l'instruction à décoder, ou sur son masque, induit une modification du décodage. Si ce décodage est invalide, une exception "instruction invalide" est levée et l'erreur sera détectée. Cependant, même si cette instruction défectueuse est valide (c'est-à-dire qu'elle fait partie du jeu d'instructions), le masque généré par cette instruction sera différent de celui de l'instruction qui aurait du être exécuté sans la faute. Par conséquent, plus le jeu d'instructions est clairsemé,c'est a dire qu'il y a de nombreuses instructions invalides par rapport au nombre d'instruction valide, plus cette méthode de détection est efficace.

Si une grande partie des opcodes est invalide, la détection ne prend que quelques cycles. La figure 4.3 montre la propagation d'une faute à travers les différentes étapes, instruction d'extraction (IF), instruction décodée (DEC) et EXEC d'un processeur généraliste.

Ainsi, si une faute se produit au temps $t = -1$, elle génère un faux masque qui est propagé au cycle suivant, mais sans détection d'erreur à l'étape DECOD. Il en est de même pour

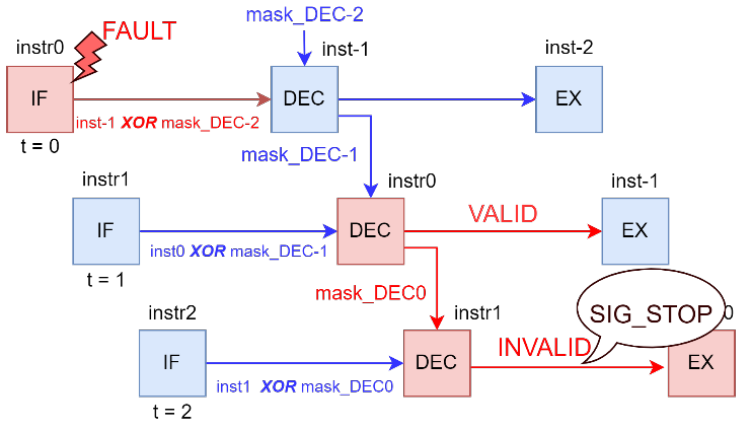

Figure 4.3 : Propagation et détection d'une instruction fautées.

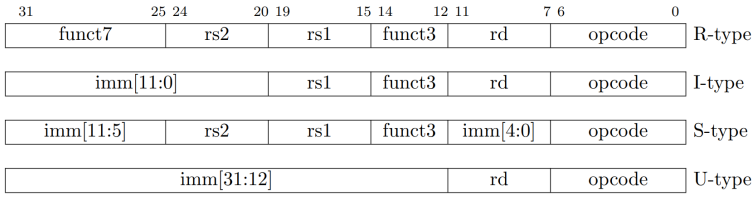

Figure 4.4 : Types d'instructions RISC-V.

l'étape DECOD au cycle 0. Cependant, au cycle 1, l'étape DECOD détecte une instruction invalide et lève une exception d'instruction invalide et le signal d'arrêt correspondant, ainsi dans l'exemple de la figure 4.3, le processeur prend 2 cycles pour détecter la faute.

Pour établir une stratégie de détection de faute rapide, il faut évaluer dans quelle partie de l'instruction les fautes ont des chances d'être détecté. Dans notre cas, il s'agit des bits d'opcode, funct3 et funct7. Il est cependant à noter que ces bits ne permettent pas de détecter les fautes de tous les types d'instruction. En effet comme on peut le voir sur la figure 4.4, il existe plusieurs types d'instruction, seul le type R possède les 3 champs opcodes, func3, func7. Ce sont des instructions arithmétiques et logiques. De plus, les bits de registres (rd, rs1, rs2) parce qu'ils correspondent aux registres ou aux immédiats sont toujours valides et ne permettent pas de détecter les fautes. Ainsi, les principales sources de détection de fautes sont les opcodes. Il faut donc maximisé la propagation des fautes vers les bits d'opcode et dans une moindre mesure les bits func. Cette contre-mesure est implémentée dans un processeur CV32E40P RISC-V, mais l'approche est bien transposable à une autre ISA ou architecture.

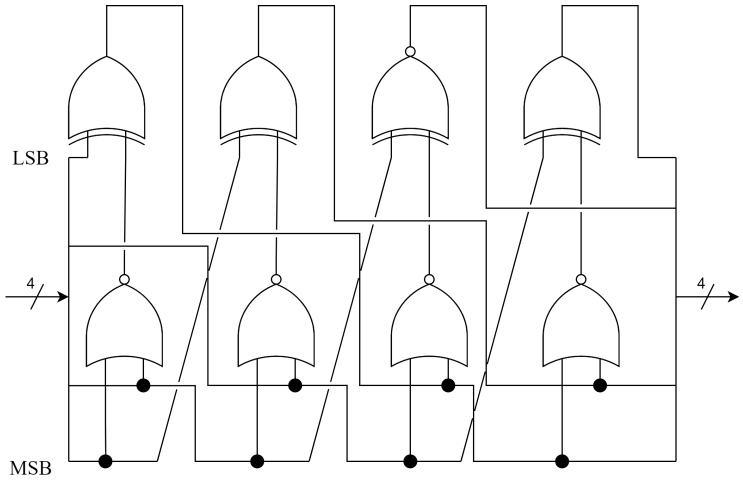

Figure 4.5 : Sbox de l'algorithme Piccolo.

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
S[x]	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

Table 4.1 : Tableau de la Sbox de Piccolo.

Choix du masque

Pour que cette détection soit la plus rapide possible, c'est-à-dire pour qu'une faute entraîne une instruction invalide le plus rapidement possible, deux approches complémentaires peuvent être suivies. La première consiste à rendre les bits interdépendants. Si une faute survient, elle doit entraîner un maximum de modifications des bits capables de provoquer une erreur de décodage. Il s'agit plus précisément des bits opcode et dans une moindre mesure func3 et func7. L'autre approche est la propagation des fautes à chaque nouveau cycle. Pour une détection efficace si une modification a lieu, le masque généré doit être très différent du masque légitime, mais aussi donner une place plus importante au bit opcode. Pour ce faire, nous allons générer deux masques en utilisant la sbox 4 bits du chiffrement léger Piccolo [**DBLP:conf/ches/ShibutaniIHMAS11**] présenté dans le tableau 4.1 et figure 4.5, pour sa légèreté d'implémentation en matériel. Le premier masque prend des blocs de 4 bits consécutifs pour créer la dépendance, le second masque prend 4 bits aux indices i,i+8,i+16,i+24 avec $i \in [\![0, 7]\!]$. En conséquence, presque tous les bits deviennent dépendants des bits de l'opcode.

La propagation des bits défectueux en cas de succession de décodage valide est assurée par une permutation du masque qui ne présente aucun surcoût matériel. En effet, il est prévu que les fautes se propagent sur l'ensemble du masque si les instructions restent valides

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

Table 4.2 : Permutation du DES.

après plusieurs cycles de décodage. Pour les propriétés de diffusion, la permutation 32 bits de l'algorithme de chiffrement DES, présentée dans le tableau 4.2, est choisie. Ainsi, si une faute se produit avant l'entrée de l'étape DECOD, elle provoquera des fautes multiples lors de la génération du masque grâce aux différentes sbox. La combinaison d'une permutation et d'une sbox assure la diffusion des fautes à tous les bits de l'instruction en quelques cycles.

4.3.2 Sécurisation du DECOD

Cette solution utilise les signaux du post-décodeur pour générer les masques. Les signaux utilisés sont l'index du registre, l'immédiat, le sélecteur de multiplexeur ainsi que les signaux d'activation. Il est difficile de faire des généralisations car ces signaux sont spécifiques à chaque architecture mais avec ceux-ci on peut s'assurer que le décodage est correct dans le cas de RISCY. Avec la solution de l'instruction précédente, on se protège contre les fautes seulement jusqu'au début de l'étape de décodage juste avant le décodeur, comme le montre la figure 4.1. Si l'on utilise les signaux de sortie du décodeur, on est assuré que le décodage a eu lieu sans erreur. En termes de taille de matériel ou de mémoire d'instructions, cette technique présente un surcoût relativement faible. Cependant, du fait que les masques sont générés à partir des signaux de décodage, le chemin critique de cette étape est allongé et peut donc conduire à une diminution de la fréquence du processeur. De plus, la compilation est rendue plus complexe, car les signaux de décodage doivent être générés de manière logicielle. Ainsi, un décodeur similaire à celui présent dans l'étage DECOD du processeur doit être ajouté au compilateur. Ces deux décodeurs doivent générer les mêmes signaux pour que le décodage soit possible. Le principal inconvénient de cette solution est de faire dépendre le code compilé non seulement de l'ISA mais aussi des spécificités de l'implémentation, dans notre cas le décodage des instructions. Pour rendre plus simple l'implémentation de la contre-mesure au niveau de la compilation, le choix a été fait d'utiliser directement les instructions précédentes et non leurs signaux de décodage.

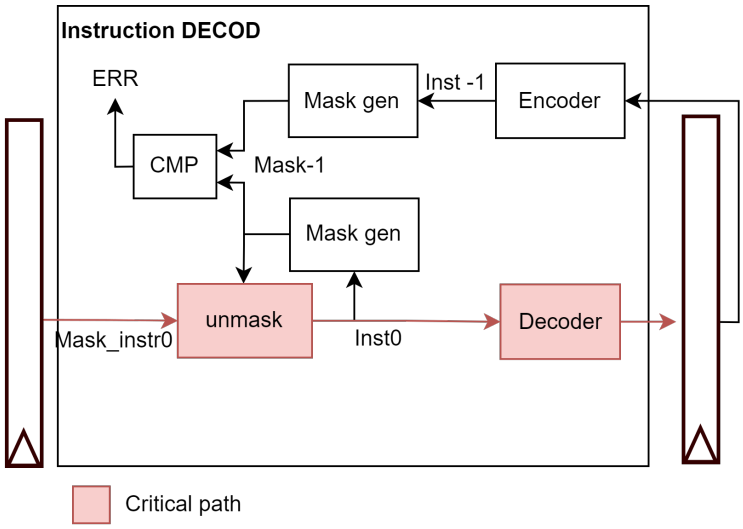

Figure 4.6 : Regénération du masque.

Regénération du masque

Si on utilise l'instruction précédente, il n'est plus possible d'assurer la sécurité du décodage. Il est donc nécessaire de profiter des différents éléments déjà ajoutés. Le principal d'entre eux est l'enregistrement du masque généré. Il serait donc intéressant de le vérifier au cycle suivant pour assurer le décodage. Les signaux d'entrée de l'étage EXEC peuvent être utilisés comme entrée pour un générateur d'instructions. En effet, il n'y a pas de perte d'information lors du décodage, seulement un changement de forme. Il est donc possible de coder les signaux de sortie du décodage sous la forme de l'instruction originale. Une fois l'instruction codée, le masque peut être généré et comparé à celui contenu dans le registre, figure 4.6.

Cette solution peut être comparée à une duplication de l'étage DECOD, mais elle présente des avantages par rapport à une simple duplication. En effet, le premier avantage est d'utiliser les signaux sortant du registre entre l'étage DECOD et l'étage EXEC. En utilisant ces signaux, on sécurise également ce registre. Au-delà de la sécurisation du registre, cela permet également d'utiliser une simple duplication qui n'est sensible qu'aux attaques par double injection. Les injections de fautes multiples sont toujours considérées comme plus difficiles à mettre en œuvre **??**. De plus, par rapport à la duplication simple, une génération de masque est utilisée, ce qui conduit à la diffusion des fautes dans l'intégralité du masque, ce qui complique encore la détection des fautes. Le surcoût du chiffrement est du même ordre que celui du décodage et est réalisé en parallèle, ce qui ne conduit pas à un al-

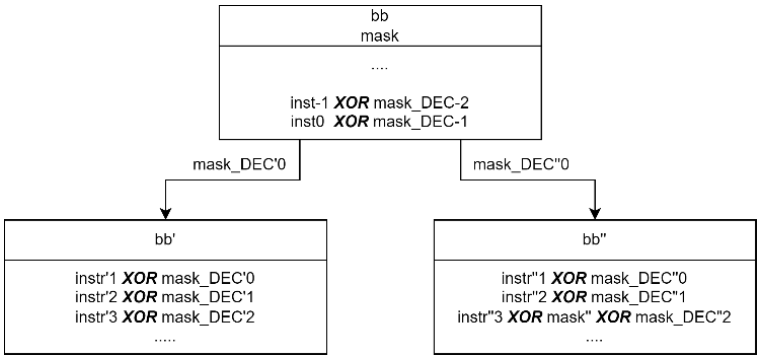

Figure 4.7 : Bloc de base avec deux successeurs.

longement du chemin critique par rapport à la duplication. Si l'on compare avec la solution des masques générés à partir des signaux de décodage, on peut noter plusieurs avantages. Tout d'abord, la détection n'est plus limitée par les erreurs de décodage et permet donc une détection plus rapide des fautes lors de celui-ci. Cependant, l'overhead matériel est plus important dans cette solution avec l'ajout d'un encodeur et d'une génération de masque, mais le chemin critique n'est pas allongé.

Cependant, d'autres problèmes apparaissent si un masque est créé à partir de l'instruction précédente ou des signaux de décodage. En effet, le code source d'un programme n'est pas parfaitement linéaire : des sauts sont possibles à différents indices du code. Par conséquent, l'instruction précédente dans le fil d'exécution n'est pas nécessairement l'instruction précédente dans le code compilé.

4.3.3 Problème à la compilation

Ce masquage ne peut se faire qu'au moment de la compilation, seul endroit où l'on connaît la signification du code assembleur et la séquence des instructions. La principale difficulté est alors de gérer les différentes branches possibles du flux d'exécution. Il y a deux cas, le premier est celui où un bloc de base comporte plusieurs blocs qui peuvent se succéder, figure **??**, c'est le cas du branchement.

Pour résoudre ce problème, il faut être capable de générer deux masques différents à partir d'une même instruction. Le deuxième cas se présente si plusieurs blocs de base pointent vers le même bloc de base suivant comme le montre la figure **??**. Ainsi une instruction peut avoir plusieurs instructions précédentes et donc avoir plusieurs masques possibles. Comme la connaissance du bloc de base qui est à l'origine de l'instruction précédente dans

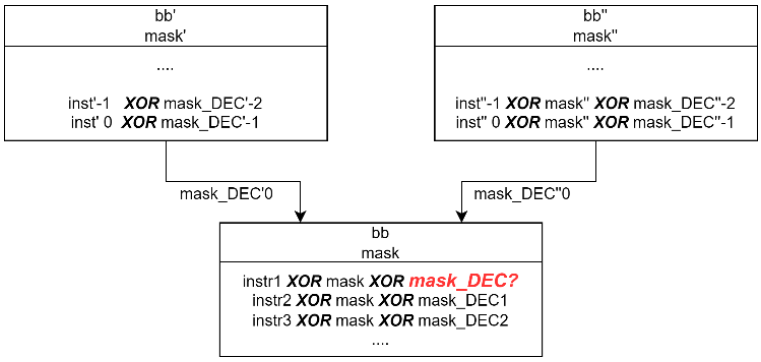

FIGURE 4.8 : Bloc de base avec deux prédécesseurs.

le pipeline du processeur est difficile, on ne peut pas choisir parmi les masques possibles. En général, un seul masque est utilisé par instruction. Ainsi, si une instruction a plusieurs antécédents, ils doivent tous générer le même masque ou cette instruction ne doit pas demander de masque.

Jump

Les premières instructions qui peuvent générer des antécédents multiples sont les sauts, qu'ils soient directs ou indirects. En effet, deux instructions de saut peuvent arriver à la même adresse. Pour ce cas particulier, le masque des instructions de saut doit être identique que le saut soit direct ou indirect. Mais il ne doit pas non plus dépendre de l'offset ou d'un registre. Des instructions de saut avec des offsets différents peuvent arriver à la même instruction. Un autre problème est qu'une instruction de saut peut également être atteinte par l'incrémentation standard du processeur. Ici, le problème est plus complexe, car n'importe quelle instruction peut précéder une destination de saut. Il est donc nécessaire que l'instruction précédant l'instruction masquée génère un masque de saut. Pour ce faire, la solution adoptée a été d'ajouter avant chaque destination de saut une instruction de saut avec l'instruction suivante comme destination. Ainsi, une destination de saut n'est accessible que par des sauts et le masque généré est toujours un mask_JUMP, Figure **??**.

Branch

Un problème similaire à celui soulevé pour les sauts subsiste pour les branches. En effet, lorsque la branche est prise, elle est comparable à un saut, tous les autres chemins possibles pour accéder à cette instruction doivent produire le même masque. Ajouter des branches avant chaque destination de branchement n'a pas de sens, car une branche signifie un saut

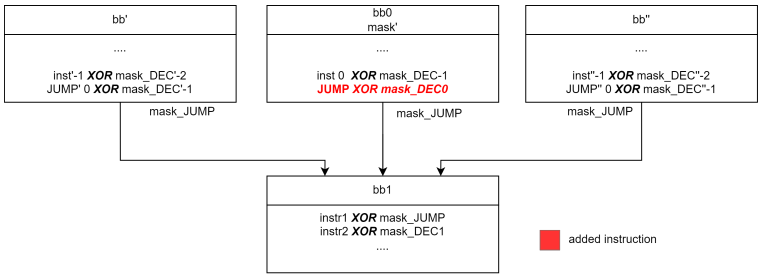

FIGURE 4.9 : Masque pendant un saut.

conditionnel, qui serait ici inconditionnel. L'ajout d'une instruction de saut est donc recommandé. Ainsi, avant chaque destination de branchement, une instruction de saut doit être ajoutée. De plus, le masque "mask_DEC0' ", voir Figure **??**, est remplacé par le masque utilisé pour les sauts, "mask_JUMP", car c'est cette branche qui peut être assimilée à un saut. Ainsi, toutes les instructions provoquant un saut dans le code assembleur produisent le même masque. De plus, chaque destination de saut est précédée d'un saut, ce qui permet d'avoir un masque unique, quel que soit le chemin pour y arriver. Enfin, il est possible de différencier si une branche est prise ou non en utilisant un masque différent dans chaque cas.

Résolutions des problèmes de commutation de contexte et des interruptions

Lors du changement de contexte, l'une des premières étapes consiste à sauvegarder le masque pour démasquer l'instruction suivante, sinon il ne sera pas possible de démasquer l'instruction à l'adresse de retour. Une étape de sauvegarde du masque est ajoutée au début du changement de contexte et une étape de restauration du masque à la fin du changement de contexte. Cette étape est analogue à celle du PC switch. Dans le cas d'une interruption, le masque qui aurait dû être utilisé dans le cycle suivant si l'interruption n'avait pas eu lieu est automatiquement sauvegardé. Dans le cas du RISC-V, cette sauvegarde est gérée automatiquement au niveau des registres de contrôle et d'état pendant une interruption.

Résolution des stall

Un processeur pour diverses raisons, dépendance de données, calcul de la branche à prendre, etc. peut geler son exécution le temps que les calculs avancent dans le pipeline pour résoudre ces dépendances. Pendant ces cycles, le processeur garde dans un registre interne pendant les cycles de gel le masque à utiliser pour l'instruction suivante.

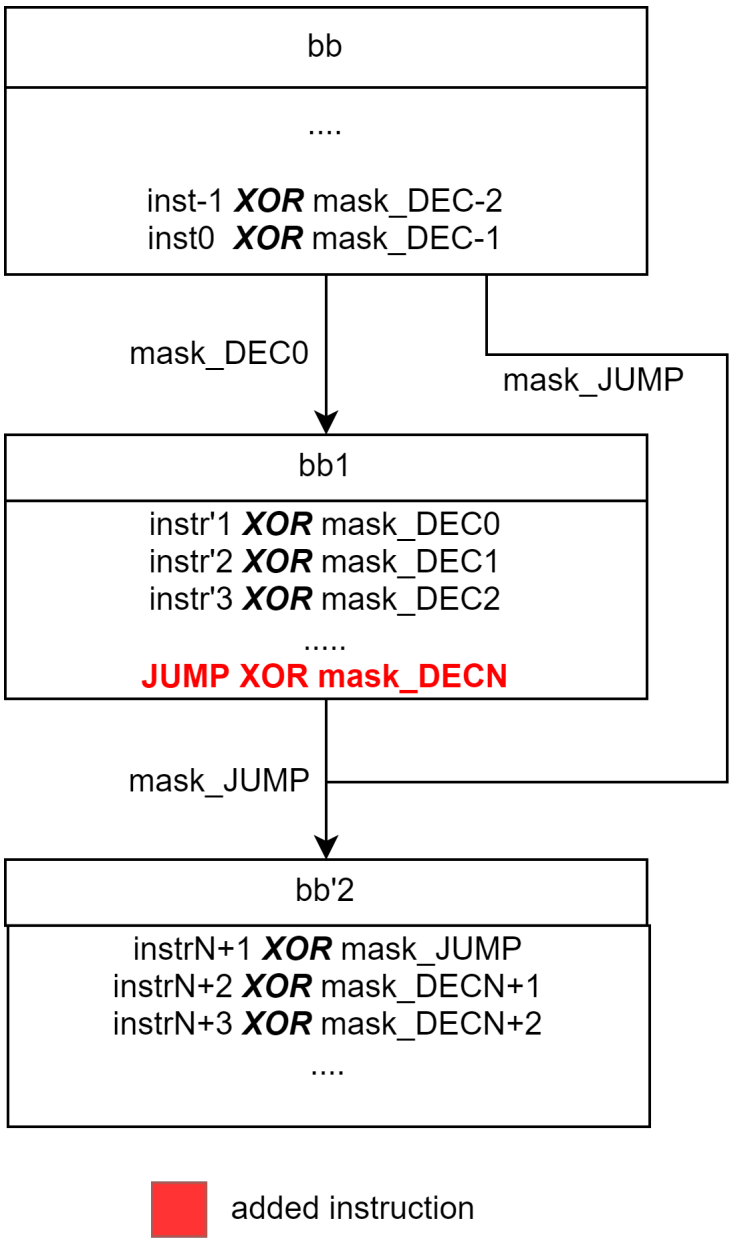

Figure 4.10 : Masque pendant un branchement.

4.3.4 Modification à la compilation

En effet, comme indiqué précédemment, toute instruction accessible par un saut ou une branche ne doit être accessible que par des instructions de saut. Une passe prend tous les blocs de base à antécédents multiples et vérifie que chacun d'eux se termine par une instruction de saut ou de branchement. Si ce n'est pas le cas, une instruction de saut est ajoutée. Ensuite, une passe ajoute les masques à toutes les instructions. Cette passe doit être la dernière, car elle fixe l'ordre d'exécution du programme, donc le code ne doit plus être modifié après cette passe. Cette passe est assez simple, car elle prend l'instruction précédente, génère le masque et l'applique ensuite à l'instruction courante. Le masque n'est qu'une composition de l'instruction précédente à l'aide de Sbox et de permutation, il est donc facile d'effectuer ces transformations en logiciel. L'utilisation d'un masque constant lors des sauts rend cette solution sensible au saut d'instruction. En effet, chaque instruction de destination JUMP possède le même masque. Ainsi, si lors d'une instruction à JUMP, un saut d'instruction est effectué et que celui-ci pointe sur une autre destination de JUMP alors le masque est valide. Il est impossible avec notre solution de détecter la faute. Bien que ce cas existe, il est en fait assez difficile à mettre en oeuvre car il faut cibler le cycle de saut précis, mais aussi déterminer une nouvelle destination de JUMP valide.

4.3.5 Implémentation

Pour réaliser notre solution, nous nous plaçons dans le cas d'un processeur RISC-V à 4 étages en ordre. Notre solution est donc ajoutée à l'étage de décodage du pipeline.

Réalisation hardware

Le registre "Decod_output" est ajouté pour stocker les signaux de sortie du décodeur, il est mis à jour à chaque cycle sauf pendant les cycles de gel. Les cycles de gel sont indiqués au processeur par le signal "freeze_sig". De plus, le masque utilisé pour tous les sauts et branches est enregistré dans le "jump_mask". Le choix du "jump_mask" ou du "DEC_mask" est déterminé par le "Jump_controller". Le "Jump_controller" utilise les signaux de décodage des instructions et les signaux de sortie de l'étape d'exécution pour déterminer si le "jump_mask" doit être utilisé. Cela inclut les cas de saut direct et indirect, mais aussi les branches prises. La solution peut être activée ou désactivée à l'aide du signal "ACT_DEC_mask". Dans le cas de notre implémentation, il s'agit d'un signal d'entrée du processeur. Ce signal est contrôlé par un registre, accessible par JTAG, à l'extérieur du

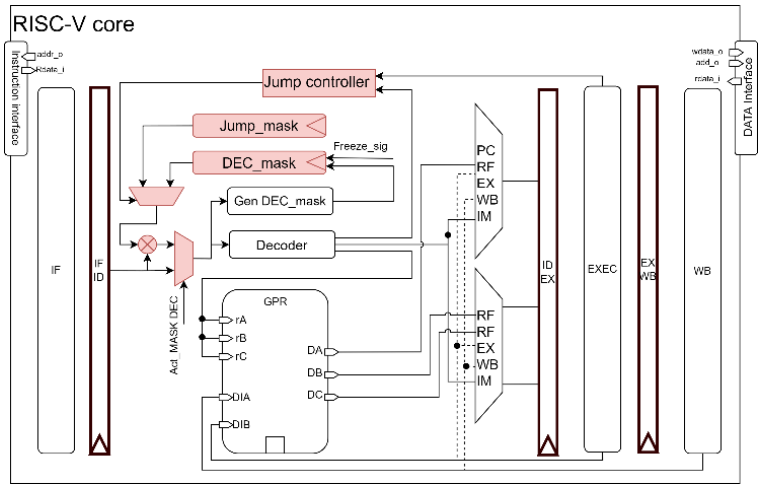

Figure 4.11 : Intégration dans le pipeline du CV32E40P.

cœur. Ce signal permet de prendre soit l'instruction entrante avec l'application du masque, soit sans l'application du masque. Le problème du premier masque à utiliser doit être résolu. En effet, la première instruction par définition n'a pas d'instruction précédente. Dans ce cas, c'est le masque de saut qui est utilisé. Il faut donc s'assurer que la première instruction exécutée est bien masquée par ce masque. Pour plus de possibilités d'activation, un signal de démarrage peut être ajouté. Cela ajoute un vecteur d'attaque ainsi qu'une complexité qui n'a pas trouvé d'avantage dans notre utilisation. Il faut noter que ce signal ne permet pas de désactiver la contre-mesure, il s'applique uniquement au masque ou non. Les deux solutions sont assez proches au niveau de l'implémentation, cependant, pour avoir la vérification de l'étage DECOD avec pré-décodeur il est nécessaire d'ajouter un encodeur et une vérification du masque, comme présenté Figure **??**, mais aussi d'ajouter les signaux manquants pour régénérer l'instruction.

Surcout matériel

Nous avons implémenté nos architectures RTL basées sur des processeurs RISCY CV32E40P et les avons synthétisées avec la bibliothèque RVT GF22FDX (GlobalFoundries 22nm FD-SOI) Standard Cells. L'overhead de cette solution est assez faible d'un point de vue matériel. Seuls le registre de masques, les applications de masques avec un xor et la génération de masques sont ajoutés. Les résultats montrent que 15 295 GE ont été nécessaires pour le post-décodeur et 17 729 GE pour le pré-décodeur. Il y a respectivement 3,71% et 16,93% de surface supplémentaire requise et une augmentation de la consommation d'énergie totale de 4,33% et 19,16% par rapport au noyau RISCY original qui a une surface

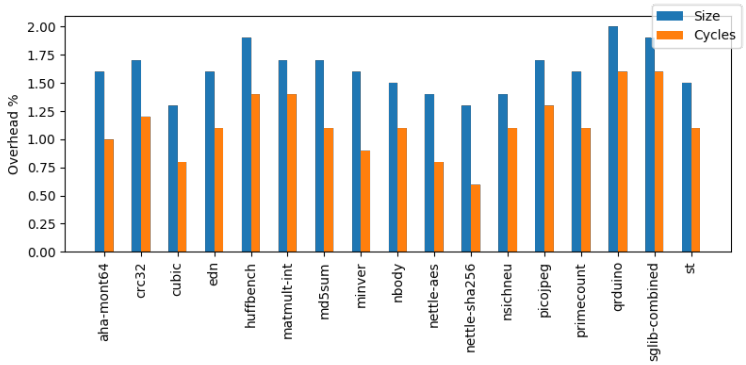

FIGURE 4.12 : Surcharge en temps d'exécution et en taille de code du benchmark Embench avec cette solution.

de 14 727 GEs.

Surcout logiciel

La figure donne le surcoût en taille de code mais aussi en temps d'exécution induit par les modifications apportées par le compilateur. Ces résultats ont été réalisés dans une simulation précise du cycle avec le benchmark Embench1.0. L'overhead de la taille du code est de 1,61% et de 1,12% pour le temps d'exécution. Ce surcoût est faible car il y a peu de modifications du compilateur et du flux d'instructions, seules les instructions de saut sont parfois ajoutées avant les destinations de saut.

4.3.6 Analyse de securité

Les avantages de notre solution sont, d'une part, sa grande légèreté, il suffit de stocker 32 bits de masques et de les xoriser à l'instruction entrante du cycle suivant. D'autre part, elle présente un surcoût très faible en termes de taille de code et de temps d'exécution. En effet, nous n'ajoutons une instruction de saut avant chaque destination de saut que lorsque cela est nécessaire. De plus, la simplicité de notre solution peut être adaptée aux compilations "just-in-time" ou aux codes auto-modifiants, la seule difficulté étant l'insertion des instructions de saut. Enfin, elle permet de se prémunir efficacement contre les sauts d'instructions. Comme l'ont souligné A. Menu et al. [1], des sauts non pas d'une, mais de plusieurs instructions peuvent se produire. Quel que soit le nombre d'instructions sautées, notre solution reste efficace, contrairement à la redondance temporelle, par exemple. L'une des principales difficultés des contre-mesures contre l'injection de fautes est de mesurer leur efficacité. En effet, aucune métrique ne permet de rendre compte de la sécurité

d'un circuit face à toutes les possibilités offertes à l'attaquant. Cependant, notre modèle d'attaquant étant limité au chemin d'instruction, nous pouvons assimiler toutes les fautes à une modification du code machine. En effet, une faute dans la hiérarchie mémoire est effectivement une corruption des binaires, et un saut d'instruction peut être assimilé à la suppression de son instruction. Pour les fautes au niveau de l'étape IFETCH et DECOD, elles peuvent également être perçues comme des corruptions des instructions. Ainsi, il est possible dans notre cas de tester exhaustivement toutes les fautes possibles sur les binaires. Comme les erreurs se propagent de cycle en cycle, nous sommes sûrs que la faute sera détectée dans un cycle, sauf en cas de saut valide. Le taux de détection n'est donc pas pertinent mais le nombre de cycles avant une erreur de décodage nous permet de comparer les différentes méthodes de création de masque. Avec ces présupposés, une campagne de test exhaustive du temps de détection d'une instruction défectueuse peut être réalisée. Il est à noter qu'il n'est pas nécessaire de disposer d'une plateforme matérielle ni même d'exécuter les instructions pour vérifier le bon fonctionnement de notre solution. Seule la séquence des instructions compte. Afin d'évaluer le masque, nous avons développé un outil logiciel permettant de simuler le décodage des instructions RISC-V. Pour simplifier la création du masque, nous utilisons la solution du pré-décodeur qui ne nécessite que l'instruction pour générer le masque. Pour gagner du temps de calcul, nous exécutons les binaires générés une première fois, puis dans un second temps nous modifions exhaustivement les instructions qui seront effectivement touchées dans le code. Nous exécutons 50 tests de vérification fonctionnelle de l'architecture RISC-V pour nous assurer que tous les types d'instructions sont bien utilisés. Nous avons retiré de nos résultats les fautes qui conduisent à un saut valide vers une autre destination de saut. En effet, du fait de l'exhaustivité de notre campagne d'injection de fautes, ces cas sont inévitables. Il n'est pas pertinent de les prendre en compte, car ce sont des cas irréalistes du point de vue de l'attaquant.

Faute simple

Le premier contrôle consiste à vérifier que le masque proposé est suffisant pour une simple injection de faute. La figure 4.13 compare la solution pré-décodeur (complète) avec un masque 32bit généré avec la fonction de hachage ASCON. Mais aussi avec les deux sous-masques qui la composent, celui des sboxes à blocs consécutifs (linear_sbox) et celui à blocs entrelacés (mix_sbox). Enfin, l'instruction sans transformation (identity). La première chose à prendre en compte est que la détection du cycle 0 est identique pour tous les types de masques, car celui-ci n'intervient pas encore et vu l'exhaustivité de notre mé-

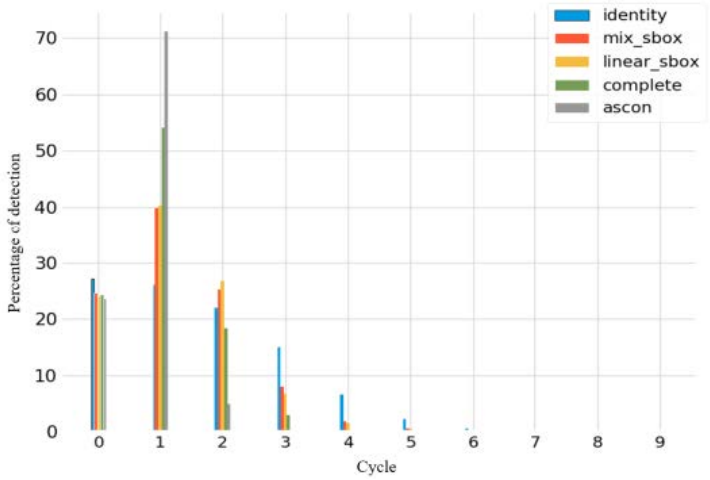

Figure 4.13 : Comparaison des différentes générations de masque pour les fautes simples

thode de test il n'y a pas de caractère aléatoire. Avec l'instruction seulement 35% des fautes restantes sont détectées au deuxième cycle, 45% avec seulement 1 type de sbox, 75% du masque complet et finalement 90% avec le hash généré par ASCON. Le tableau 4.3 montre le nombre moyen de cycles avant qu'une faute ne conduise à une erreur de décodage. Ainsi, un hash de l'instruction permet de détecter la faute en 2 cycles, notre solution qui consiste en seulement 16 sbox matérielles permet de détecter 97% des fautes en 2 cycles. Puisque 99% sont détectés en moins de 3 cycles, nous pouvons raisonnablement supposer que ces fautes n'ont pas corrompu la mémoire de données, donc en désactivant le pipeline et la banque de registres le processeur est dans un état sûr.

Faute multiple

Les fautes multiples sont des attaques de plus en plus utilisées [**DBLP:conf/cardis/ColombierGVCBLC21**] et permettent de contourner la sécurité mise en place contre les fautes simples. Cependant, notre solution, comme on peut le voir sur la figure 4.14, offre une meilleure protection lorsque le nombre de fautes augmente. Ainsi un attaquant n'a aucun intérêt à transmettre des fautes plus complexes sauf s'il veut modifier très précisément une instruction pour sauter vers une destination de saut valide, ce qui semble précisément hors de portée des attaques actuelles.

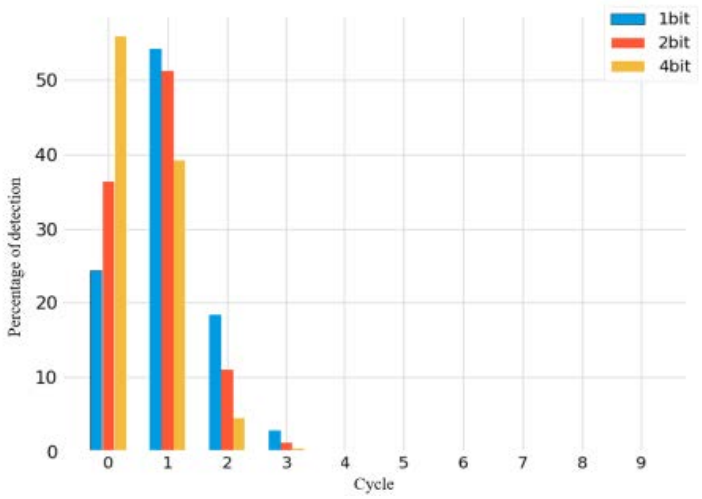

Figure 4.14 : Comparaison des différentes générations de masque pour les fautes multiples

hline	Identity	linear_sbox	Mix_sbox	Complete	ASCON
Average cycle detection	1.52	1.20	1.20	0.99	0.81

Table 4.3 : Comparaison du nombre moyen de détection pour les différentes générations.

Saut d'instruction

Au-delà de la modification des instructions, il est également courant que les injections de fautes provoquent des sauts d'instructions qui peuvent être uniques ou multiples, certaines solutions sont sensibles au nombre d'instructions sautées. Notre solution, comme on peut le voir sur la figure 4.15, détecte les sauts en moins de 2 cycles et ce, que le saut soit de 1, 2 ou 4 instructions. Par conséquent, notre solution reste efficace contre les attaques par fautes sur l'ensemble du chemin des instructions. Plus l'attaque est complexe, avec des fautes et des sauts, plus elle est efficace.

Attaque par canaux auxiliaires

Enfin, au-delà de la protection contre les attaques par fautes, le fait d'ajouter un masque aux instructions permet de renforcer la résistance aux attaques par canaux latéraux. Cependant, il ne s'agit pas d'une contre-mesure qui empêche formellement les fuites par canaux latéraux, mais qui rend seulement plus difficile le désassemblage, qui se fait avant l'étape DECOD [3], qui est à notre connaissance la seule attaque qui cible le chemin des instructions par canaux latéraux. Même si le masque n'est pas aléatoire, il permet de réduire de

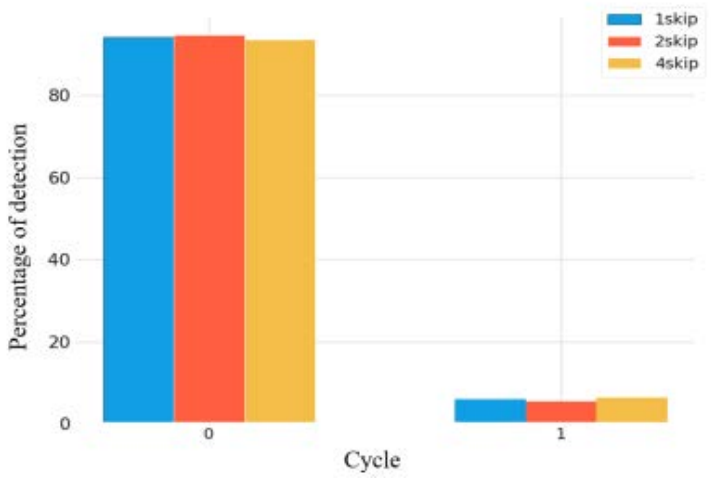

Figure 4.15 : Comparaison des différentes générations de masque pour les sauts d'instruction

nombreuses heuristiques comme les opcodes impossibles, les séquences d'instructions les plus probables et les dépendances entre opcodes, funct3 et funct7. Sans rendre ce type d'attaque impossible, notre contre-mesure les rend beaucoup plus complexes.

4.4 duplication/parité croisé sur les signaux de controle

Les registres et signaux de contrôle sont assez peut sensible aux attaques par canaux auxiliaires d'une part, car ils sont difficiles a ciblé précisément, mais aussi, car il ne contient des données ayant peu d'intégrer, car elle ne contient pas de donnée sensible. Cependant étant au centre du contrôle et de l'état du processeur elles sont une cible intéressante pour les injections de faute.

4.4.1 Description

Une des caractéristiques principales d'un code correcteur efficace contre les attaques par injections de faute sont ceux présentant une forte localité spatiale dans leur domaine de faute. Plus précisément il faut que deux fautes proches ne puissent pas être valides. On a donc cherché à maximiser cette distance spatiale à l'aide de la parité. En effet. La parité permet de détecter toutes les fautes impaires dans un mot ainsi en entrelaçant les parités il devient par exemple impossible d'injecter exactement deux fautes consécutives. Si on prend l'exemple de la longitudinal parity check (LPC), voir Figure 4.16, ainsi toutes fautes consécutives qui ne fait pas exactement 2 fois la longueur du LPC sera détecté. Il est ainsi

facile de combiner les codes détecteurs à base de parité. Si l'on combine la longitudinal et l'horizontal parity check(HPC) on obtient un code détecteur capable de détecter toutes les fautes qui n'ont pas un schéma bien défini. On peut encore aller plus loin en prenant aussi les diagonales cependant cela commence à couter cher en mémoire, mais aussi en logique combinatoire. En effet, il faut n xor pour calculer une parité et $p\sqrt{n}$ flip flop avec n le nombre de bits du mot.

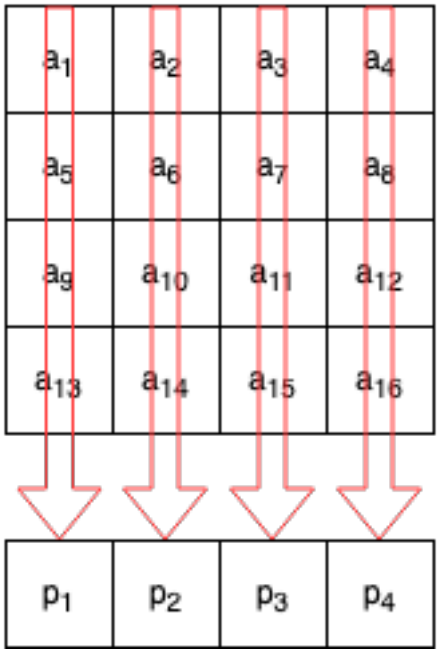

Figure 4.16 : Parité longitudinale

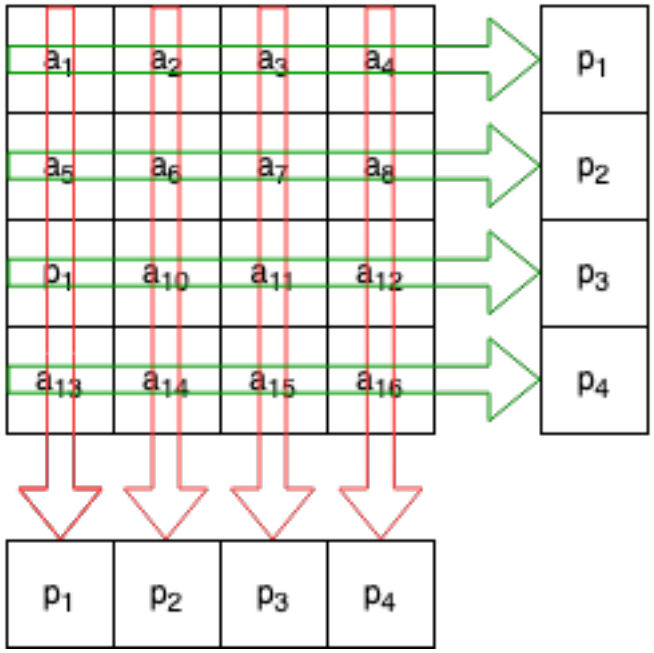

Figure 4.17 : Parité croisée

4.4.2 Avantage

Le premier avantage de cette solution est son coût restreint, du fait d'utiliser seulement de la parité, il s'agit donc en terme d'implémentation seulement de porte XOR. Nous cherchons avant tout la légèreté pour pouvoir effectuer des vérifications à toute les entrées et sorties d'étages mais aussi en au niveau des registres d'états. Enfin, si on cherche de la sureté on peut induire des la correction d'erreur monobit, du fait du croissent des deux types de parité il est possible d'obtenir les coordonnées dans la matrice de parité du bit fauté. En utilisant cette parité croisé on induit une dépendance spatiale entre les fautes ainsi pour faute un bit precis il faudra au minimum injecter 3 fautes a des endroits specifique pour induire une faute valide. Ainsi, contrairement au code detecteur d'erreurs qui ne sont capable que de détecter une distance minimal, ainsi le faute avec une source unique

peut induire plusieurs fautes consécutive peut plus facilement induire des fautes. Dans le cas de la parité croisé des schema d ; injections specifique avec de forte contrainte spatiale doivent être respecté.

Ainsi si on reprends la méthodologie présenté dans la section précédente on peut comparer plusieurs méthodes de sécurité. Nous comparerons ici le HPC, LPC, CPC ainsi qu'un code BCH de paramètre (à definir)

4.5 Conclusion

Cet article propose deux solutions pour protéger le chemin des instructions contre les attaques par défaut et par canal latéral. Ces solutions proposent de générer des masques soit directement avec les instructions précédentes, soit avec les signaux de décodage de l'instruction précédente. L'avantage de ce type de contre-mesure est qu'il permet de traiter tous les problèmes d'injection de fautes sur les instructions avec un surcoût très faible. En utilisant les signaux de décodage, nous avons un overhead de 3,25%, nous assurons la détection des fautes en 1 cycle en moyenne, mais rend la compilation dépendante des contraintes micro architecturales. En ne prenant que les informations du pré-décodeur, nous assurons la sécurité contre les attaques par injection de fautes sur le chemin des instructions au prix d'un doublement de l'étage DECOD. Mais le code source généré devient seulement dépendant du jeu d'instructions. L'overhead logiciel est très faible en raison des quelques modifications apportées au compilateur. En effet, nous avons un overhead de 1,61% en taille de code et de 1,12% en temps d'exécution. Cette solution propose une sécurité contre les attaques par observation et physique sur le chemin des instructions, elle est donc complémentaire à la solution de sécurité du flux de contrôle et du chemin des données.

5

Processeur global

En ayant proposé des contremesures pour le chemin de données et pour le chemin de contrôle, nous avons une protection à priori complète due pipeline de processeur contre les attaques en faute et par canaux auxiliaires. Cependant au-delà de regarder ces deux chemins de manières indépendantes il est intéressant de regarder l'architecture globale du processeur pour étudier si de nouvelles contremesures ne peuvent pas être misent en place. Dans ce chapitre une contremesure, exploitant les principes de bases des processeurs c'est à dire exécuter des instructions, est présentée pour complexifier tout type d'attaques physiques. Et en deuxième partie, nous verrons comment sont intégrées ces contremesures dans un pipeline d'un processeur RISC-V 32 bits. Avec les différentes interactions entre les contremesures qui renforcent la sécurité, mais aussi leurs adaptations pour cette architecture spécifique démontrant ainsi leur adaptabilité.

5.1 Désynchronisation

L'insertion de retards aléatoires dans l'exécution d'un algorithme cryptographique est une contre-mesure simple mais plutôt efficace ces attaques. À notre connaissance, les retards aléatoires sont largement utilisés pour la protection des implémentations cryptographiques dans les dispositifs embarqués, principalement les cartes à puce de par leur faible surcout. Il appartient à un groupe de contre-mesures de dissimulation qui introduisent un bruit supplémentaire (soit dans le domaine temporel, amplitude ou fréquentiel) aux fuites par canaux auxiliaires tout en n'éliminant pas d'information du signal en lui-même. Ceci est le contraire du masquage qui élimine la corrélation entre la fuite et les données sensibles. Le mélange de plusieurs contre-mesures de dissimulation et de masquage est souvent utilisé pour augmenter la complexité des attaques.

La plupart des attaques par canaux auxiliaires et par injections de fautes exigent que l'adversaire sache précisément quand les opérations cibles se produisent dans l'exécution. Cela permet de synchroniser plusieurs traces au moment de l'événement critique, comme dans le cas de l'analyse de puissance différentielle, ou d'injecter une perturbation dans les calculs au bon moment, comme dans le cas des attaques par injections de fautes. En introduisant des retards aléatoires dans l'exécution, la synchronisation est rompue, ce qui augmente la complexité de l'attaque. Parmi les différentes techniques de randomisation temporelle proposées dans la littérature, par exemple [**DBLP:conf/ches/CoronK09**; **DBLP:conf/ches/ClavierCD00**], on peut généralement distinguer les techniques logicielles, par exemple basées sur les délais aléatoire par interruption (RDI), des techniques matérielles, par exemple basées sur l'augmentation de la jitter de l'horloge. En général, plus les contre-mesures sont proches du matériel, plus les solutions pour les surmonter sont axées sur le traitement du signal [**DBLP:conf/wistp/GuilleyKLD11**; **DBLP:conf/ctrsa/WoudenbergWB11**]. Dans ce contexte, il est intéressant de noter que de nombreuses évaluations de l'impact des contre-mesures (par exemple [**DBLP:conf/ctrsa/Mangard04**]) prétraitent les traces de fuite en les intégrant. Quelque peu influencées par cette technique d'évaluation, des recherches sont menées pour augmenter au maximum la variabilité des injections de cycles factices pour améliorer la distribution statistique de l'échantillon aléatoire des retards, afin que leur intégration produise les traces les plus bruyantes possible. Mais d'autres évaluations se basant sur la reconnaissance de motif permettent d'éliminer les ajouts d'instructions [**DBLP:conf/cardis/DurvauxRSOV12**]. En résumé, pour avoir des injections de code fac-

tice efficace, il faut augmenter la distribution statistique de l'échantillon aléatoire des retards et éviter que ces injections suivent un motif particulier.D'autres évaluations basées sur la reconnaissance de formes, en revanche, peuvent éliminer les ajouts d'instructions [7]. Dans [7], les auteurs proposent des exigences pour une insertion efficace des instructions factices, ils devraient utiliser : - des retards sans motif régulier - des insertions qui ne sont pas prévisibles - des retards qui ressemblent à l'instruction environnante Les motifs réguliers sont difficiles à éviter pour les méthodes logicielles, car ces instructions nécessitent souvent des prologues. De plus, dans de nombreuses applications, nous ne pouvons pas nous permettre d'augmenter la taille du code. Seule l'insertion matérielle à l'exécution d'instructions factices permet d'éviter cette augmentation. Néanmoins, les solutions matérielles de l'état de l'art, telles que [8], [9], répondent aux deux premiers points en insérant aléatoirement des instructions sans motif, mais elles posent le problème de la distinction entre les instructions factices et les instructions réelles. Mais aussi dans la diversité et la cohérence des instructions insérées.

5.1.1 Présentation de la contremesure

La contre-mesure proposée est un système matériel capable d'insérer des instructions factices à des intervalles aléatoires pendant l'exécution du programme. Le but est de matérialiser le concept de cycle factice, qui est utilisé au moment de la compilation dans la grande majorité des propositions. Pour faciliter les explications de la réalisation, nous nous plaçons dans l'architecture RISC-V avec un processeur CV32E40P à quatre étages. Le choix de l'architecture du jeu d'instructions (ISA) et de l'architecture du processeur n'a pas d'incidence sur la pertinence de la solution. L'architecture de la contre-mesure proposée, présentée sur la Figure 1, est divisée en deux parties : la génération de l'instruction avec le registre "dummy opcode" et le bloc "Generate random instr". Ensuite, l'injection de l'instruction dans le pipeline du processeur avec un multiplexeur est contrôlée par un diviseur de fréquence programmable "Variable div clock" et un bit aléatoire "RNG". Il y a aussi la propagation d'un signal "flag dummy" pour avertir les étapes suivantes si une instruction dummy est en cours.

Génération de l'instruction factice

La génération d'instructions factices devrait permettre d'obtenir des instructions aussi semblables que possible aux instructions valides, mais c'est un défi. La solution la plus courante consiste à créer les instructions fictives à partir d'opcodes arithmétiques prédéfi-

nis. Cette solution permet de les générer en utilisant des instructions déjà traitées, ce qui permet d'utiliser la quasi-totalité de l'ISA comme instructions fictives. Seuls les éléments nécessaires à la détermination du type d'instruction doivent être sauvegardés. En observant les différents types d'instructions dans la figure 2, nous pouvons constater que seuls 17 bits du champ opcode, funct3 et funct7 sont utiles lors du décodage de l'instruction. Il est important de prêter attention aux instructions nécessitant un niveau de privilège particulier. Dans l'architecture RISC-V, ces instructions sont regroupées dans le SYSTEM opcode et sont ignorées dans notre solution pour éviter toute incohérence de privilège. Ainsi, si seuls ces 17 bits sont sauvegardés et que les autres bits peuvent être choisis au hasard, le décodage devrait toujours être valide. Mais générer 15 bits aléatoires nécessite une implémentation coûteuse du RNG. Il est donc préférable de choisir ces 15 bits dans un état interne du processeur (parmi le registre de contrôle/état (CSR), les registres, le compteur de programme (PC), etc.) Sa sélection a peu de contraintes, elle doit juste changer assez fréquemment parmi un grand ensemble de valeurs. Elle présente également peu de problèmes de sécurité car les données manipulées sont utilisées pour la sélection de registres ou d'immédiats. Elle permet des entrées différentes pour les instructions à chaque exécution, et ajoute de la variabilité en randomisant le poids de Hamming de l'instruction fictive et la distance de Hamming avec les instructions authentiques suivantes et précédentes.

Insertion de l'instruction factice

L'insertion des instructions ne doit pas être prévisible. Il est nécessaire d'ajouter du hasard dans la décision d'insérer des instructions factices. Dans notre solution, un diviseur d'horloge variable fournit un signal d'horloge à la fréquence F*2 lorsque F est la fréquence cible d'insertion des instructions fictives. F peut être défini par un registre de configuration, par exemple un CSR. Dans notre implémentation, pour des raisons pratiques, nous utilisons un registre de 32 bits, extérieur au cœur du processeur, modifiable par JTAG. L'insertion d'une instruction est conditionnée par un multiplexeur commandé par la sortie d'une porte ET avec comme entrées la sortie d'un générateur de nombres aléatoires et la sortie du diviseur d'horloge. Il y a une chance sur deux pour que le signal de sortie du diviseur d'horloge traverse la porte AND, ce qui entraîne une fréquence moyenne d'insertion d'instruction fictive égale à F. À chaque période du signal du diviseur d'horloge, le système récupère une instruction et la stocke dans le registre "dummy opcode". La variabilité a encore augmenté car les opcodes stockés ne sont utilisés qu'une seule fois. L'insertion des instructions factices se fait à l'étage DECOD. A ce moment, l'étape IF est gelée. L'exécution est reprise

au cycle suivant de manière transparente pour le noyau. Ainsi, le flux normal d'exécution n'est pas perturbé et les interruptions sont toujours fonctionnelles. Avec une instruction fictive et les signaux générés pendant son exécution, nous devons nous assurer qu'elle ne perturbe pas le flux normal d'exécution. Le signal "dummy flag" informe le processeur qu'il exécute une instruction fictive.

5.1.2 Modification de l'architecture

Il est difficile de définir toutes les modifications car elles dépendent fortement de l'architecture et de la mise en œuvre du processeur. Cependant, certaines considérations générales peuvent être soulevées. Tout d'abord, lorsque le processeur exécute une instruction fictive, aucune écriture en mémoire ou dans les registres légitimes n'est autorisée pour des raisons évidentes d'intégrité de la mémoire. En outre, il faut éviter que le processeur modifie les registres de contrôle, les drapeaux ou le PC ; par exemple, les instructions de saut et de branchement. Il est également nécessaire d'éviter que les résultats soient contournés. Comme proposé ci-dessous, il pourrait être intéressant de modifier l'architecture pour que ces cycles fictifs ressemblent davantage à des instructions légitimes.

Écriture dans les registres et la mémoire

L'écriture dans la banque de registres est un facteur distinctif qui peut être utilisé pour détecter les instructions factices. Pour éviter cela, nous n'écrivons que dans les registres qui ne sont pas utilisés par le flux d'exécution normal. Il n'est pas nécessaire de modifier quoi que ce soit pour les lectures dans les registres car seules les écritures influencent le véritable flux de contrôle. Comme tous les registres sont susceptibles d'être utilisés, il est nécessaire de prévoir deux registres de destination supplémentaires (shadow register). Dans le processeur CV32E40P, à chaque cycle, deux registres peuvent être écrits en même temps, un pour le résultat de l'ALU et un autre pour les accès à la mémoire. Cependant, l'écriture toujours dans les mêmes registres est identifiable. Il est alors possible d'implémenter un banc de registres dynamique, comme montré dans [10], pour utiliser les mêmes registres physiques pour les instructions factices et légitimes.

Le banc de registres dynamique, dans la figure 3, est rendu possible en séparant les index de registres ciblés par l'ISA des registres physiques. L'index fait maintenant référence à une table de consultation pointant vers un emplacement de registre physique. Une table de validité des registres physiques doit être conservée afin de déterminer quel registre

est utilisé. La table de validité consiste à ajouter un bit de validité à chaque registre de la banque. Lorsqu'un registre physique est écrit, il est considéré comme valide. Il y a deux façons d'invalider un registre, la première est lors du retour d'une fonction. En effet lorsqu'une instruction "RET" est exécutée une partie des registres est sauvegardée et l'autre non. Ces registres non sauvegardés peuvent être considérés comme invalides. L'autre façon de désactiver un registre physique est de maintenir la nature dynamique de la banque de registres. Lors de l'écriture dans la banque de registres, le registre physique associé au registre ISA à écrire est d'abord invalidé, puis un nouveau registre physique est choisi parmi les registres libres. Ainsi, à chaque écriture dans un registre ISA, celui-ci est écrit dans un registre physique différent. La figure 4 illustre une écriture dans un registre ISA déjà affecté à un registre physique.

Le "bloc Nxt$_{r}egister"tireunindicederegistreauhasardparmilesregistresnonvalides.Lalogiquecomplexe$

La gestion des instructions Load/Store est un problème en raison de la difficulté de déterminer les espaces mémoire libres. Nous résolvons ce problème en attribuant une adresse unique à la lecture et à l'écriture de la mémoire.

Sauts et branchements

Les branches et les sauts étant des instructions courantes, nous devons trouver un moyen de les exécuter tout en restant aussi proche que possible du comportement original. Nous utilisons la même opération pour les sauts, mais au lieu de sauter à l'adresse indiquée dans l'opérande ou le registre, nous passons à l'instruction suivante.

La même approche ne peut pas être utilisée pour les branchements ; en effet, si le branchement est effectué, le processeur doit annuler les instructions qui ont été exécutées avant le résultat du branchement. Par conséquent, les instructions seraient réexécutées, ce qui poserait un problème non seulement de détection mais aussi de fuite via des canaux secondaires. Par conséquent, nous avons été contraints de ne pas considérer les branches comme des instructions fictives.

5.1.3 conclusion

Nous avons présenté une solution hardware adapté au processeur généraliste avec un faible surcout. Elle consiste en l'injection de retard aléatoire basé sur l'injection d'instruction aléatoire factices dépendante du contexte d'exécution. Notre solution de par l'injec-

tion d'instruction randomisé déjà utilise sans prologue ni préambule, nous place dans un cadre optimal qui est impossible à atteindre par les solutions logicielles. Contrairement aux autres solutions matérielles nous proposons des instructions en cohérence avec le contexte d'exécution et nos instructions ont une grande variabilité et sont peu différentiable d'instruction légitime

5.2 combinaison des contre-mesures

5.3 Implémentions dans le VT2/résultats

5.4 Conclusion

6

Conclusion

Dans l'état de l'art, quelques solutions ont été apportées pour masquer le pipeline d'un processeur et aucune n'a proposé de solutions spécifiques aux attaques par injections de fautes. On ne trouve donc à fortiori aucun travail essayant de résoudre ces deux problèmes conjointement. C'est pourquoi cette thèse s'est concentrée sur l'adaptation aux contraintes spécifique d'un pipeline d'un CPU pour proposer des solutions innovantes avec toujours un attrait particulier au surcout matériel des solutions pour une intégration avec un coût raisonnable. Au-delà du surcout l'aspect facilement adaptable et généralisable à tout type d'architecture a été au centre de la démarche de recherche pour d'une part être intégré sur un large type de cible allant du micro contrôleur avec exécution dans l'ordre au processeur d'application avec exécution dans le désordre.

La solution a été de travailler sur des ensembles communs à tous les processeurs le chemin de contrôle et le chemin de donnée. Pour le chemin de données, 3 solutions ont été proposées. La première est l'ajout de tag d'intégrité. Ces tags sont générés à partir d'une permutation aléatoire. L'ajout de cette permutation aléatoire permet d'augmenter le niveau de sécurité face aux attaques par injections de fautes. En effet, de par la nature aléatoire de la permutation un attaquant même avec une connaissance parfaite du système ne peut pas s'assurer de sa capacité à injecter une faute valide. La deuxième, bien que connue, est le masquage avec une réflexion sur celui le plus adapté au pipeline de processeur. Après avoir analysé les différents masquages à faible latence, il a été choisi de partir sur le masquage LMDPL qui offre le surcout matériel le plus faible et l'augmentation du chemin critique le plus contenu. Cette thèse apporte une méthode analytique pour construire ce masquage et montre ainsi que l'impossibilité d'un masquage de ce type pour les ordres supérieurs.

Enfin, des registres dynamiques sont proposés, bien qu'existant déjà dans l'état de l'art, il permet d'apporter une sécurité supplémentaire sur les registres en rendant aléatoire le placement des données dans les registres. En effet, ceux-ci sont la partie la plus sensible du chemin de données, mais apportent aussi des interactions fondamentales avec la contre-mesure des cycles factices.

Le chemin de contrôle quand a lui a été sécurisé à l'aide de méthode plus légère contre les canaux auxiliaires, car celui si est moins sensible aux d'attaques de ce type même si elle reste possible. Les sécurités se sont principalement concentrées contre les attaques en fautes avec la création de dépendance entre les instructions. En effet, la solution proposée est de générer un masque à l'aide des signaux de décodage de l'instruction précédente pour masquer l'instruction en cours. Le fait d'utiliser un masque permet de réduire le surcout de la redondance à quelques pour cent que ce soit en termes de taille de circuit mais aussi de temps d'exécution. De plus ce masque apporte une résistance face désassemblage bits à bits par canaux auxiliaires. Enfin, les signaux de contrôle sont sécurisé à l'aide d'une parité croise qui offre une sécurité importante face aux injections de fautes multiples avec un surcout faible.

Enfin d'un point de vue global nous proposons un système de désynchronisation temporelle aléatoire pour complexifier les attaques par injection de faute et par canaux auxiliaires. Celui-ci consiste en l'insertion d'instruction aléatoire qui sont peu différentiable d'instruction légitime. Enfin nous avons implémenté ces différentes contre-mesures dans un véhicule de test qui a été fondu et qui va être testé par le CESTI du CEA-LETI.

De ces travaux de thèses nous pouvons en tirer quelques conclusions intéressantes. Tout d'abord l'ajout d'aléatoire contre les attaques en fautes est le niveau le plus avancé dans la prévention de ces attaques. En effet, il n'est plus possible pour un attaquant ayant une parfaite connaissance du système cible de pouvoir réaliser une attaque, mais cela permet aussi d'apporter de l'aléa dans les traces pour les canaux auxiliaires complexifiant ainsi l'analyse statistique. D'autre part, l'approche du masquage et détection par le décodage des instructions permet d'avoir des solutions très légères pour la détection de faute au niveau des instructions. Enfin la résilience passe par des indicateurs spécifiques a une attaque par injection de faute et c'est au logiciel de gérer ces cas là. En effet il est souvent trop complexe d'un point de vue matériel de gérer ces cas rares. La solution est la plus simple est la correction d'erreur, mais celle-ci offre de nouvelles opportunités d'attaques. Mais avec l'ajout de l'authenticité aux propriétés des contremesures pour les attaques par

injections de fautes, la combinaison de sécurité et sûreté est réalisable sans compromettre l'un ou l'autre. Ainsi cette thèse apporte une série de contributions pour une protection des principaux points clés d'un pipeline de processeurs contre les attaques par canaux auxiliaires et par injections de faute.

6.1 Perspective

Cependant certaines questions reste ouvertes ouvertes à la recherche notamment la résilience. En effet, même si nous avons mis en place des marqueurs spécifiques à une faute matérielle, il faut que le logiciel puisse gérer ces cas et avoir des procédures adapté à chacune des situations le cas le plus problématique soulevé dans cette thèse est celui d'une faute dans le banc de registre qui ne peut pas être adressé facilement. De plus, la partie de génération de nombre aléatoire n'as pas été traité dans cette thèse est le choix de la génération de l'aléatoire a été laissé en suspends. Une autre piste de recherche est l'améliorer de SECDEC pour en faire un CFI complet et plus léger que les solutions proposées dans l'état de l'art. D'autre part, bien qu'un masquage a été identifié il serait intéressant au vu de la nature généralisable et composable de LMDPL d'automatiser cette génération des fonctions masquées. Enfin ces contre-mesures soient intégrées dans un processeur 32 bits et qu'elle semble généralisable à tout type d'architecture, la continuité naturelle est de tester l'intégration sur des architectures plus complexes avec des optimisations qui pourraient interagir avec nos contre-mesure et ouvrir de nouvelle piste d'amélioration des contremesure, on pourrait apr exemple étudier les interaction entre les cycles factice et l'exécution dans le désordre.