AKHACIA : Arborescent Keyed Homomorphic tAgs for Confidentiality, Integrity and Authenticity of data in CPU pipeline

Gaëtan Leplus[1], Olivier Savry[1] and Lilian Bossuet[2]

[1] CEA Leti, Univ. Grenoble Alpes, Grenoble F-38000, France, firstname.lastname@cea.fr
[2] Laboratoire Hubert Curien, Jean Monnet University, Saint-Etienne F-42000, France, firstname.lastname@univ-st-etienne.fr

Abstract. Avec les évolutions des attaques par injection de faute, notamment les injections de fautes multiples, les duplications simples ne sont plus suffisantes pour assurer une sécurité suffisante face à ce type d'attaque. L'état de l'art actuel se concentre principalement sur de la redondance, mais en travaillant dans un autre domaine de représentation et une vérification. Ici, nous introduisons de l'aléatoire dans la redondance. Ainsi il reste difficile pour l'attaquant de reproduire des fautes même avec une connaissance parfaite du système qu'il est en train d'attaquer. Cette solution ne cherche pas a se substituer aux solutions existantes contre les fautes telles que les codes correcteurs, mais d'offrir la protection initiale dans le cadre d'opération arithmétique classique telle que les opérations booléennes, des décalages ou des additions. D'autre part cette solution reste compatible avec les opérations de masquages tels que LMDPL. Nous avons évalué cette contre-mesure sur un processeur RISC-V (cv32e40p).

Keywords: fault attack, side channel, countermesure, processor, RISCV

1 Introduction

L'essor des objets connectés de l'IoT et de tout système embarqués en général que ce soit pour le grand public ou pour l'industrie donne de plus en plus un accès physique privilégié aux attaquants en tout genre. Ceci est d'autant plus problématique que des processeurs open source tel que celles fondées sur l'ISA Risc V donnent une connaissance approfondie de leur architecture interne. Si les attaques invasives sur les puces semblent toujours difficiles, les attaques par observation et perturbation deviennent parfaitement réalisables. La première catégorie aussi connue sous les termes d'attaques side-channel (SCA) [BCO04], se base sur l'exploitation de fuite physique (émanation électromagnétique, température, temps, etc.). L'autre type d'attaque est les injections de faute (FA) [BS97] qui consiste à injecter (émission électromagnétique, lasers, glitch, etc.) et exploiter des erreurs dans la logique du système pour récupérer des données secrètes. Ces attaques peuvent être menées individuellement ou combinées [RLK11].

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 [Bar+06], temporelle et informationnelle [KKG03; MSY06]. On peut avoir de la redondance par réplication où on se contente de dupliquer tel quel le circuit, on parle de DMR (Dual modular redundancy) ou de TMR si le circuit est répliqué trois fois (un vote majoritaire permet alors de distinguer la bonne valeur, ce qui n'est pas possible lors d'une simple duplication). On voit que ces solutions n'empêchent pas un attaquant de cibler à la fois les bits des deux ou trois circuits [SHS16]. La redondance temporelle demande de recalculer, après un certain

temps, le calcul déjà effectué pour s'assurer de leur égalité. Elle rend difficile le travail de l'attaquant qui doit cibler deux fenêtres temporelles pour effectuer ses fautes. Néanmoins l'attaque reste possible une fois que la synchronisation est bien configurée [Col+21]. La redondance d'information est mise en œuvre avec des codes détecteurs d'erreurs. Certains sont simples à implémenter comme la parité, mais présentent des capacités de détection assez faibles. D'autres, tout en ayant une capacité de détection plus efficace, sont plus lourds et demandent plusieurs cycles d'horloge pour la vérification. Ils ne peuvent corriger qu'un nombre réduit de fautes (Hamming code [NBD06], Low Delay Single Error Correction [Sai+15], etc.). L'exécution de ces codes suit des schémas figés et connus qui permettent toujours d'entrevoir une possibilité d'injection de fautes.

Finalement, ces codes détecteurs/correcteurs se contentent d'assurer l'intégrité des données c'est-à-dire d'assurer que les données ne sont pas modifiées là où on veut s'assurer que les données ont bien aussi conservé leurs valeurs d'origine. Ainsi, conjointement à l'intégrité, l'authenticité des données doit être recherchée. Ceci est généralement obtenu en cryptographie par l'ajout d'un MAC (Message Authentication Code). Ainsi, on s'assure avec l'authenticité des données qu'un attaquant ne pourra pas forger un tag ou MAC même en connaissant la donnée associée. Alors que la génération d'un tag à base de code détecteur est évidente quand on connaît l'algorithme et la donnée, un vrai tag d'authenticité nécessite une clé secrète pour être généré.

Face à ce besoin d'intégrité et d'authenticité contre les injections de fautes, la prévention des attaques SCA revient à assurer la confidentialité des données. L'approche classique est alors de masquer les données. Celles-ci sont alors partagées entre plusieurs shares aléatoires [PR13] dans le but de rendre la fuite d'information indépendante de la donnée sensible. Chaque share est alors processé en garantissant une indépendance stochastique avec les autres. Plusieurs techniques de masquage sont maintenant possibles et résistantes aux glitches comme TI ou DOM [GMK17] et de plus récentes ont pris en compte à la fois le besoin de peu de bits aléatoires, mais aussi de faible latence dans les calculs. On peut citer LMDPL [LMW14], SESYM [Nag+22] etc. . .

La résolution conjointe des fuites et des fautes pour assurer à la fois confidentialité, intégrité et authenticité (CIA) dans le cadre de la protection des calculs des instructions d'un processeur a été jusqu'à maintenant peu étudiée. Une solution a été toutefois proposée avec le procédé M&M (Mask and MAC [Mey+19]) qui consiste à associer à la donnée un MAC lié à une clé secrète α et qui prend la valeur α.x où x est la donnée à protéger et la multiplication est effectuée sur le corps $GF(2^n)$. Cette technique permet de combiner le MAC avec du masquage. Les calculs sur la donnée sont aussi effectués avec le MAC dans le cadre de la logique d'un crypto-processeur. Cela assure des calculs authentiques sans faute moyennant plusieurs multiplications. Si cette technique peut être bien adaptée à la protection d'un algorithme cryptographique, elle pose problème dans l'ALU d'un CPU à cause de la difficulté de faire cette multiplication dans $GF(2^n)$ en peu de cycle d'horloge (idéalement un seul) et à cause d'autres types d'opérations comme le décalage, l'addition ou la multiplication arithmétique pour lesquelles elle n'est pas adaptée.

Problématique Nous avons vu que le chemin de données et l'exécution des instructions dans l'ALU d'un processeur est susceptible par des attaques SCA ou FA de dévoiler et/ou de corrompre les données manipulées. L'ajout d'un tag d'authenticité et d'un masquage sur ce dernier et sur la donnée permet alors d'atteindre une assurance CIA. En particulier, on veut garantir que les calculs sont effectués sur des opérandes authentiques et intègres et que les résultats fournis le soit également. Nous sommes également confrontés dans le cadre du pipeline d'un processeur généraliste au fait qu'il faille vérifier à chaque étage la validité des données manipulée pour éviter que celle-ci se propage à l'intérieur des registres ou dans les écritures en mémoire. Les contraintes sont néanmoins très fortes pour ne pas trop alourdir et ralentir le CPU. Ainsi, il est souhaitable que les calculs effectués sur les MAC

se fassent sur le même cycle d'horloge que les calculs sur les données et n'allongent que de façon très restreinte le chemin critique. On s'autorise par contre un étage supplémentaire sur le CPU et donc un cycle d'horloge pipeliné en plus pour traiter les problématiques de masquage et en particulier la gestion des glitches. On considerera q'un masquage au premier est suffisant.

Contribution Nous proposons dans cet article de dédoubler l'étage d'exécution du CPU en deux domaines parallèles, voir Figure 1. Le premier domaine est canonique, les opérations logiques et arithmétiques de l'ALU y sont effectuées normalement quoique de façon masquée.Le masquage low-latency et glitch-resistant LMDPL est alors utilisé pour sa légèreté et sa rapidité et l'utilisation modéré de bits aléatoires. Le deuxième domaine en parallèle est appelé domaine "permuté" car il effectue les mêmes opérations sur des données dont les bits ont été permutés. Ce domaine joue le rôle de tag d'authenticité ou MAC car la permutation utilisée dépend d'une clé secrète. Celle-ci a été choisie pour sa structure arborescente qui lui confère des propriétés d'homomorphisme par rapport aux opérations logiques et arithmétiques permettant des vérifications de calculs très rapides. Malgré la difficulté d'implémenter une permutation en hardware, nous montrons que son occupation spatiale est très limitée et son utilisation très efficientes pour la plupart des opérations. La dépendance de cette permutation avec une clé secrète interne au CPU et qui peut être changée à volonté procure une sorte de polymorphisme ou de randomisation à l'implémentation matérielle de l'ALU. L'injection de fautes ciblées par un attaquant se trouve grandement compromise. Les qualités homomorphes de cette permutation font qu'elle est également compatible avec le masquage à tous les ordres même si par la suite nous viserons qu'un masquage au premier ordre en utilisant la méthode LMDPL. La légèreté du schéma utilisé fait que celui-ci ne peut prétendre à une qualité cryptographique élevée car la seule connaissance de quelques couples entrée/sortie de la permutation peut permettre de retrouver la clé. Mais son utilisation interne au CPU couplé à du masquage et à un changement fréquent de la clé procure une assurance CIA totalement satisfaisante. Enfin, cette solution a été implémentée dans le pipeline d'un processeur Risc-V cv32e40p sur Asic en technologie FDSOI 22nm afin d'évaluer ses performances et sa sécurité. Les solutions de msquage n'ont toutefois pas été implémentées sur ce démonstrateur.

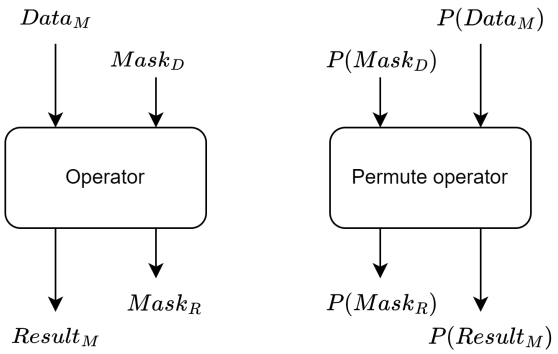

Figure 1: Datapath scheme.

Organisation Ce papier est structuré comme suit : la section II présente le modèle d'attaquant auquel nous cherchons à faire face. Dans la section III nous justifierons du choix d'utiliser une permutation et laquelle nous avons choisi. Dans la section IV, nous décrirons comment grâce à la permutation choisie nous arrivons à faire des opérations logiques et arithmétiques complexes telles que le décalage, l'addition et la multiplication.

Table 1: tableau récapitulatif du nombre et taille de faute en fonction de la puissance du laser.

Énergie[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

En Section V, il est question de l'intégration de cette contre-mesure dans le pipeline de processeur RISCV (cv32e40p) et voir la compatibilité de cette solution avec les méthodes de masquage telles que le masquage LMDPL. Enfin, en section V, nous présenterons une évaluation de la sécurité apportée par la permutation.

2 Modèle l'attaquant

Modèle d'attaquant On considère que notre attaquant possède des capacités d'injections de fautes. Cet attaquant a des capacités d'injection de faute élevée avec l'injection de fautes multiples non stochastiques (multi spot laser device)et additives (applique une fonction xor au bit fauté). On ne considère pas de nombre maximum de fautes cependant en pratique avec les technologies actuelles, de l'ordre de 4 fautes distinctes et précises peuvent être réalisées avec un banc laser. Le cas des erreurs stochastiques n'est pas étudié, car notre solution donne des résultats similaires à une duplication dans ce cas. Si on ajoute le fait qu'un laser est capable de provoquer un très grand nombre de fautes en augmentant la puissance du laser utilisé l'attaquant possède une large plage de faute possible. Un exemple de campagne d'injection réalisée sur une puce de 28 nm [God+09] les résultats de cette étude sont donné en table 1.

Dans le cas plus spécifique d'un processeur généraliste, on considère que l'attaquant ne cible que le chemin de donnée c'est-à-dire là où seules les données transitent, ainsi les chemins d'instructions et de contrôle ne sont pas dans le spectre de notre attaquant. Des solutions pour protéger ces parties sont déjà présentées avec un faible surcout (citer le papier DSD). Cependant, le chargement et l'enregistrement des données, les opérations logiques et arithmétiques ainsi que le banc de registres et les registres inter-étage sont des cibles potentielles. En plus, des attaques par injections de fautes, notre attaquant possède des capacités d'attaque par canaux auxiliaire. Nous nous plaçons dans le cas du d-probing [ISW03] au premier ordre, du fait du surcout prohibitif des contre-mesures des ordres supérieurs, pour le pipeline d'un processeur. Le modèle d'attaquant inclut également les glitchs matériels. On considère donc que les données ont pu traverser la hiérarchie mémoire avec une assurance CIA et sont arrivées sous forme masquées dans les registres généraux du CPU. La possibilité de chiffrer de façon authentifiée la DRAM [SEH20] mais aussi de protéger la hiérarchie de cache par du masquage léger [Tal+22] a déjà été démontrée. On considérera donc que les données sont masquées dans les registres généraux du CPU, associés à leur tags d'authenticité (eux-mêmes masqués) et aux masques utilisés.

3 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

3.1 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 XOR $m, p_\alpha(x)$ XOR m') = (x XOR $m, p_\alpha(x$ 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 ?

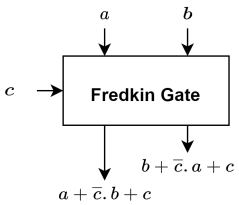

Figure 2: Fredkin gate.

3.2 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. Une transposition T_K^n échange le

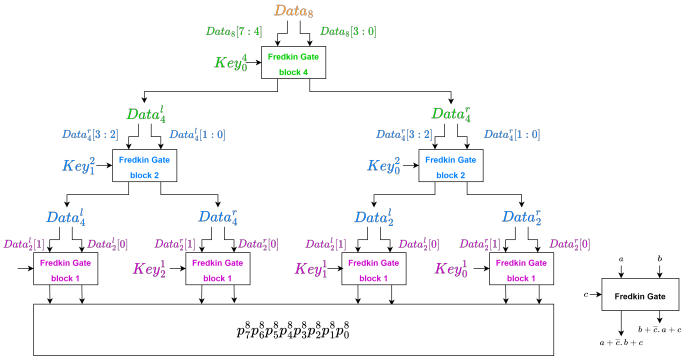

Figure 3: Block permutation.

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^{32}_{Key}. 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 $P^{32}_{Key} = T^{16}_{Key^{16}_0} o T^{8}_{Key^{8}_0} o T^{8}_{Key^{8}_1} o T^{4}_{Key^{4}_0} o ... o T^{4}_{Key^{4}_3} o T^{2}_{Key^{2}_0} o ... o T^{2}_{Key^{2}_7} o T^{1}_{Key^{1}_0} o ... o T^{1}_{Key^{1}_{15}}$ comme défini en Figure 4. 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.

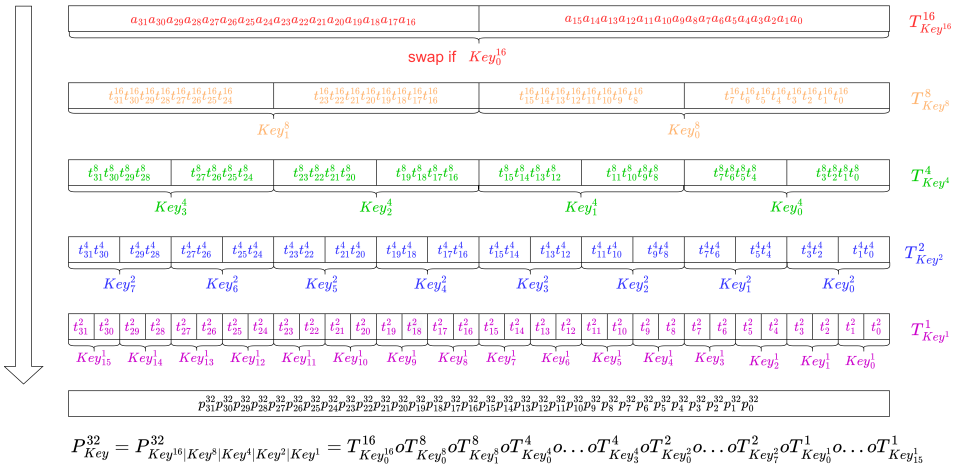

Figure 4: Block Permutation.

4 Opération logiques et arithmétiques

4.1 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 3.2. 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 et la multiplication.

4.2 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 5. 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.

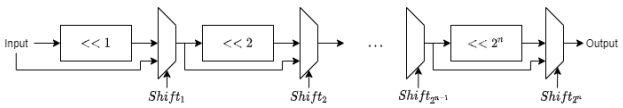

Figure 5: opérateur de décalage.

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 6 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 $Key^1 = 10$.

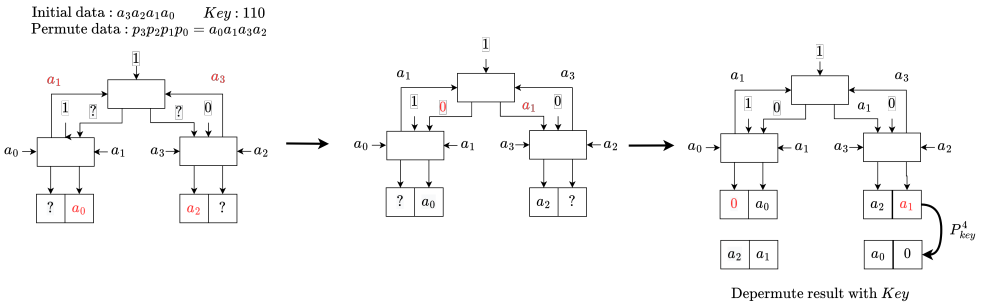

Figure 6: Explication du décalage.

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 6, 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 7, 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 6. 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.

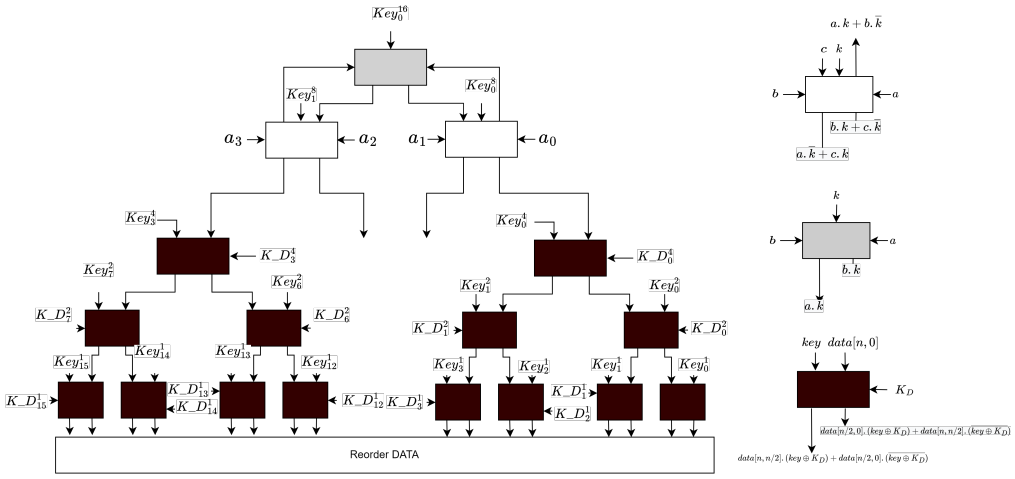

Figure 7: Shift data.

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 8, 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.

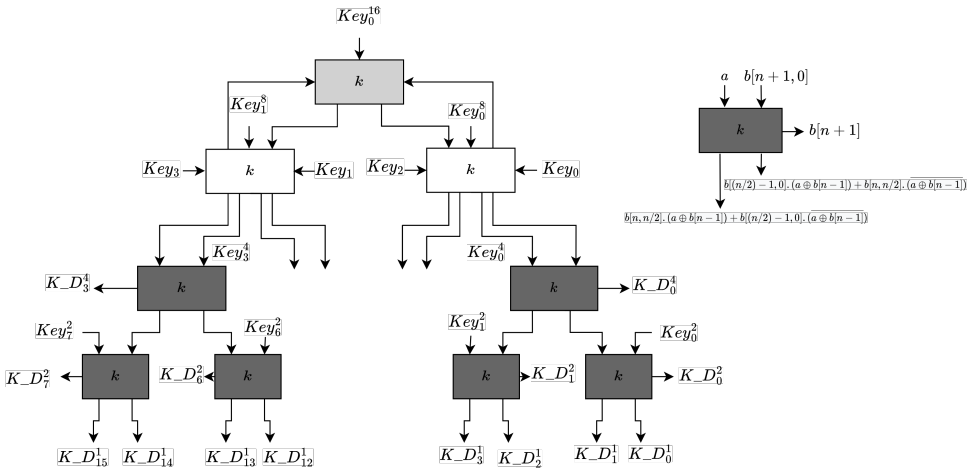

Figure 8: Find old Key.

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.

4.3 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.

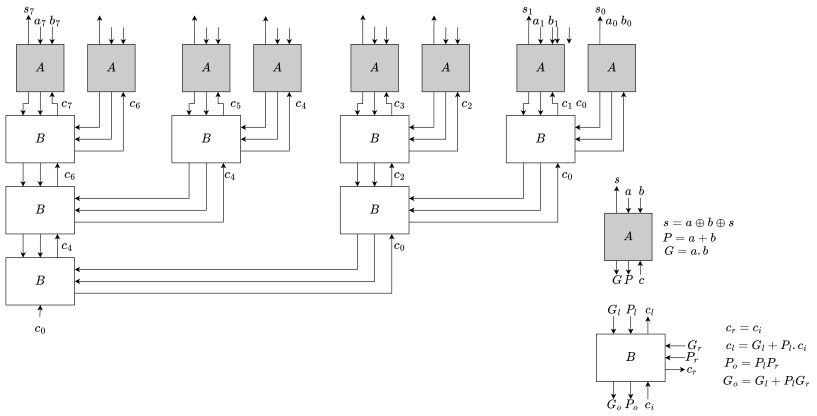

Figure 9: Carry look ahead.

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
- $P_i = (a_i + b_i)$ 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 9 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 10. 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

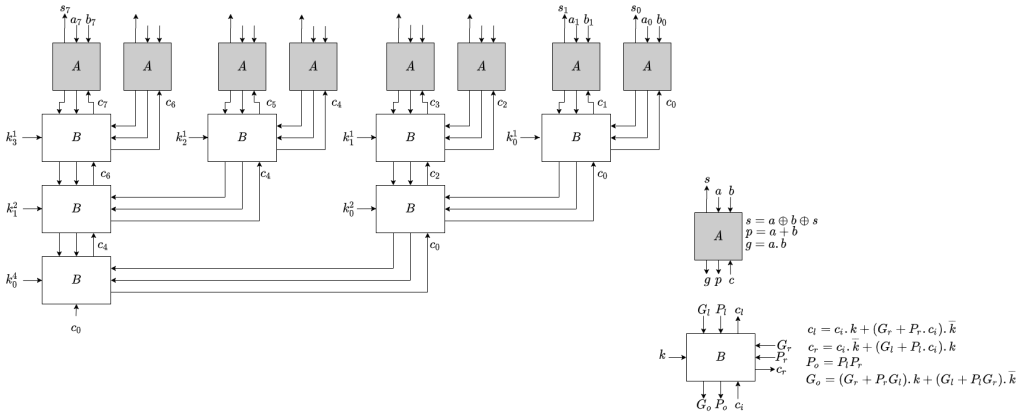

Figure 10: Permute look ahead.

4.4 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 11. 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

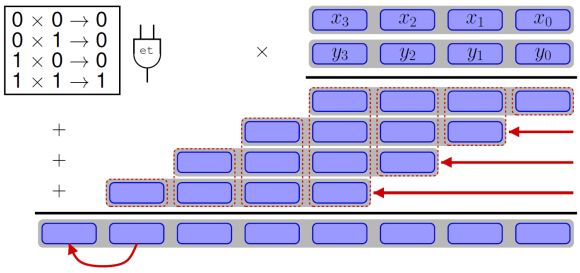

Figure 11: Multiplication binaire.

de ce type 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 12. 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

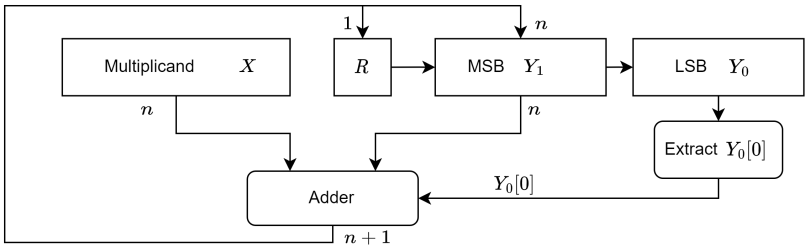

Figure 12: Description matérielle d'une multiplication itérative sur n bits.

description compatible avec les permutations, peu de changement doit être mené. Il faut d'abord remplacer l'additionneur par celui décrit en 4.3. De plus, le registre à décalage doit être remplacé par un décalage par celui présenté en 4.2. 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 13

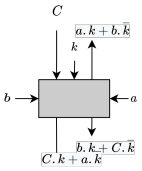

Figure 13: Bloc de décalage pour la multiplication itérative.

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 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 14. 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.

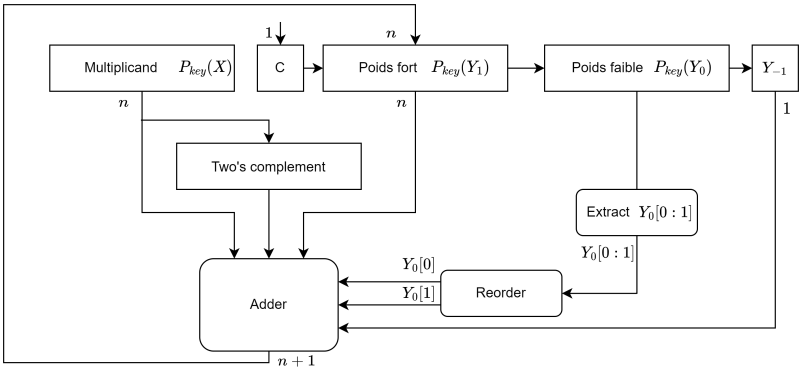

Figure 14: Multiplication de booth avec des données permutées

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.

5 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 15.

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 l'ALU. Il y a différentes sortes de calculs possibles définis par des instructions : des instructions

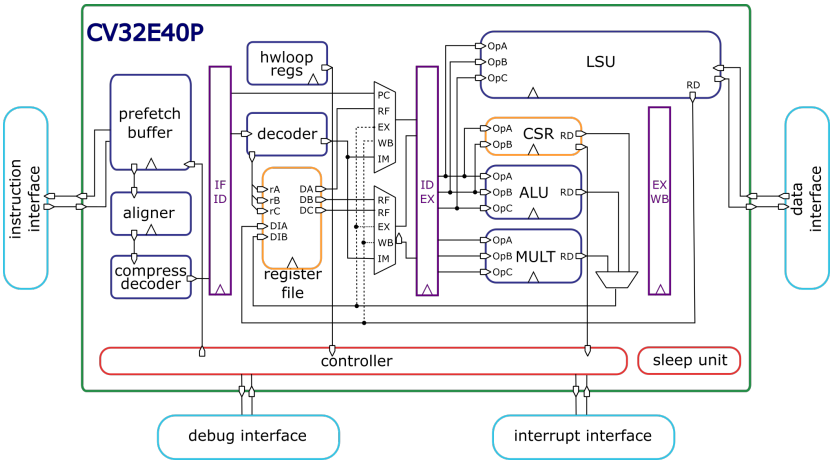

Figure 15: Architecture du cœur CV32E40P

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.

5.1 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

5.2 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 [MS16], 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 [GIB18]. 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 [LMW14] est une des premières méthodes a faible latence générique, mais il a fallu attendre pour [Sas+20] 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 thresold, appelée LLTI, qui a des exigences de surface comparables à celles du GLM, mais qui peut éliminer le besoin d'aléatoire en ligne [AZN21]. Enfin SESYM [Nag+22] 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.

5.3 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'exploitation 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.

6 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.

6.1 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

6.2 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].

6.3 securite contre les attaques en fautes

6.3.1 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

6.3.2 logique combinatoire

Étendre les résultats des registres

7 Conclusion

Cette contre-mesure se place donc comme la brique essentielle contre les attaques en fautes en ajoutant de l'aléatoire pour toutes les opérations arithmétiques du processeur. Cette solution est un supplément à toutes les autres solutions telles que les codes correcteurs. En effet, elle permet d'ajouter une notion d'aléa dans les opérations arithmétiques standard que l'on trouve dans un processeur standard avec un surcoût assez faible tout en ne limitant pas les contre-mesures standard contre d'autres types d'attaques tels que le masquage pour les attaques par canaux auxiliaires.

8 Perspective

Même si on assure l'authenticité des données celle-ci n'est pas exploitée dans la solution proposée, des recherches complémentaires pourraient être même pour tirer profit de cette nouvelle propriété apportée au processeur. Une piste de recherche serait l'isolation des processus. En assignant un pr Une autre piste de recherche serait le polymorphisme de calcul. En effet, du fait de l'utilisation de clé aléatoire pour permuter les bits a chaque exécution les calculs seront utilisés pour perspective assurer une authenticité des données et des calculs perspective : Utiliser comme calcul polymorphe dans le chiffrement vérifier def polymorphisme gestion d'isolation de processus. ajouter des capaciter de corrections et de detection

Acknowlegment This work was supported by the French National Research Agency in the framework of the "Investissements d'avenir" program (IRT Nanoelec, ANR-10-AIIX.RT-05).

References

[AZN21] Victor Arribas, Zhenda Zhang, and Svetla Nikova. "LLTI: Low-Latency Threshold Implementations". In: *IEEE Trans. Inf. Forensics Secur.* 16 (2021), pp. 5108–5123. DOI: 10.1109/TIFS.2021.3123527. URL: https://doi.org/10.1109/TIFS.2021.3123527.

[Bar+06] Hagai Bar-El et al. "The Sorcerer's Apprentice Guide to Fault Attacks". In: *Proc. IEEE* 94.2 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424. URL: https://doi.org/10.1109/JPROC.2005.862424.

[BCO04] Eric Brier, Christophe Clavier, and Francis Olivier. "Correlation Power Analysis with a Leakage Model". In: *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*. Ed. by Marc Joye and Jean-Jacques Quisquater. Vol. 3156. Lecture Notes in Computer Science. Springer, 2004, pp. 16–29. DOI: 10.1007/978-3-540-28632-5_2. URL: https://doi.org/10.1007/978-3-540-28632-5_2.

[BS97] Eli Biham and Adi Shamir. "Differential Fault Analysis of Secret Key Cryptosystems". In: *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*. Ed. by Burton S. Kaliski Jr. Vol. 1294. Lecture Notes in Computer Science. Springer, 1997, pp. 513–525. DOI: 10.1007/BFb0052259. URL: https://doi.org/10.1007/BFb0052259.

[Col+21] Brice Colombier et al. "Multi-Spot Laser Fault Injection Setup: New Possibilities for Fault Injection Attacks". In: *Smart Card Research and Advanced Applications - 20th International Conference, CARDIS 2021, Lübeck, Germany, November 11-12, 2021, Revised Selected Papers*. Ed. by Vincent Grosso and Thomas Pöppelmann. Vol. 13173. Lecture Notes in Computer Science. Springer, 2021, pp. 151–166. DOI: 10.1007/978-3-030-97348-3_9. URL: https://doi.org/10.1007/978-3-030-97348-3_9.

[GIB18] Hannes Groß, Rinat Iusupov, and Roderick Bloem. "Generic Low-Latency Masking in Hardware". In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2018.2 (2018), pp. 1–21. DOI: 10.13154/tches.v2018.i2.1-21. URL: https://doi.org/10.13154/tches.v2018.i2.1-21.

[GMK17] Hannes Groß, Stefan Mangard, and Thomas Korak. "An Efficient Side-Channel Protected AES Implementation with Arbitrary Protection Order". In: *Topics in Cryptology - CT-RSA 2017 - The Cryptographers' Track at the RSA Conference 2017, San Francisco, CA, USA, February 14-17, 2017, Proceedings*. Ed. by Helena Handschuh. Vol. 10159. Lecture Notes in Computer Science. Springer, 2017, pp. 95–112. DOI: 10.1007/978-3-319-52153-4_6. URL: https://doi.org/10.1007/978-3-319-52153-4_6.

[God+09] C. Godlewski et al. "Electrical modeling of the effect of beam profile for pulsed laser fault injection". In: *Microelectron. Reliab.* 49.9-11 (2009), pp. 1143–1147. DOI: 10.1016/j.microrel.2009.07.037. URL: https://doi.org/10.1016/j.microrel.2009.07.037.

[ISW03] Yuval Ishai, Amit Sahai, and David A. Wagner. "Private Circuits: Securing Hardware against Probing Attacks". In: *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*. Ed. by Dan Boneh. Vol. 2729. Lecture Notes in Computer Science. Springer, 2003, pp. 463–481. DOI: 10.1007/978-3-540-45146-4_27. URL: https://doi.org/10.1007/978-3-540-45146-4_27.

[KKG03] Ramesh Karri, Grigori Kuznetsov, and Michael Gössel. "Parity-Based Concurrent Error Detection of Substitution-Permutation Network Block Ciphers". In: *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*. Ed. by Colin D. Walter, Çetin Kaya Koç, and Christof Paar. Vol. 2779. Lecture Notes in Computer Science. Springer, 2003, pp. 113–124. DOI: 10.1007/978-3-540-45238-6_10. URL: https://doi.org/10.1007/978-3-540-45238-6_10.

[LMW14] Andrew J. Leiserson, Mark E. Marson, and Megan A. Wachs. "Gate-Level Masking under a Path-Based Leakage Metric". In: *Cryptographic Hardware and Embedded Systems - CHES 2014 - 16th International Workshop, Busan, South Korea, September 23-26, 2014. Proceedings*. Ed. by Lejla Batina and Matthew Robshaw. Vol. 8731. Lecture Notes in Computer Science. Springer, 2014, pp. 580–597. DOI: 10.1007/978-3-662-44709-3_32. URL: https://doi.org/10.1007/978-3-662-44709-3_32.

[Mey+19] Lauren De Meyer et al. “M&M: Masks and Macs against Physical Attacks”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2019.1 (2019), pp. 25–50. DOI: 10.13154/tches.v2019.i1.25-50. URL: https://doi.org/10.13154/tches.v2019.i1.25-50.

[MS16] Amir Moradi and Tobias Schneider. “Side-Channel Analysis Protection and Low-Latency in Action - - Case Study of PRINCE and Midori -”. In: *Advances in Cryptology - ASIACRYPT 2016 - 22nd International Conference on the Theory and Application of Cryptology and Information Security, Hanoi, Vietnam, December 4-8, 2016, Proceedings, Part I*. Ed. by Jung Hee Cheon and Tsuyoshi Takagi. Vol. 10031. Lecture Notes in Computer Science. 2016, pp. 517–547. DOI: 10.1007/978-3-662-53887-6_19. URL: https://doi.org/10.1007/978-3-662-53887-6_19.

[MSY06] Tal Malkin, François-Xavier Standaert, and Moti Yung. “A Comparative Cost/Security Analysis of Fault Attack Countermeasures”. In: *Fault Diagnosis and Tolerance in Cryptography, Third International Workshop, FDTC 2006, Yokohama, Japan, October 10, 2006, Proceedings*. Ed. by Luca Breveglieri et al. Vol. 4236. Lecture Notes in Computer Science. Springer, 2006, pp. 159–172. DOI: 10.1007/11889700_15. URL: https://doi.org/10.1007/11889700_15.

[Nag+22] Rishub Nagpal et al. “Riding the Waves Towards Generic Single-Cycle Masking in Hardware”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2022.4 (2022), pp. 693–717. DOI: 10.46586/tches.v2022.i4.693-717. URL: https://doi.org/10.46586/tches.v2022.i4.693-717.

[NBD06] Riaz Naseer, Rashed Zafar Bhatti, and Jeff Draper. “Analysis of Soft Error Mitigation Techniques for Register Files in IBM Cu-08 90nm Technology”. In: *2006 49th IEEE International Midwest Symposium on Circuits and Systems*. Vol. 1. 2006, pp. 515–519. DOI: 10.1109/MWSCAS.2006.382112.

[PR13] Emmanuel Prouff and Matthieu Rivain. “Masking against Side-Channel Attacks: A Formal Security Proof”. In: *Advances in Cryptology - EUROCRYPT 2013, 32nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Athens, Greece, May 26-30, 2013. Proceedings*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. Lecture Notes in Computer Science. Springer, 2013, pp. 142–159. DOI: 10.1007/978-3-642-38348-9_9. URL: https://doi.org/10.1007/978-3-642-38348-9_9.

[RLK11] Thomas Roche, Victor Lomné, and Karim Khalfallah. “Combined Fault and Side-Channel Attack on Protected Implementations of AES”. In: *Smart Card Research and Advanced Applications - 10th IFIP WG 8.8/11.2 International Conference, CARDIS 2011, Leuven, Belgium, September 14-16, 2011, Revised Selected Papers*. Ed. by Emmanuel Prouff. Vol. 7079. Lecture Notes in Computer Science. Springer, 2011, pp. 65–83. DOI: 10.1007/978-3-642-27257-8_5. URL: https://doi.org/10.1007/978-3-642-27257-8_5.

[Sai+15] Luis J. Saiz-Adalid et al. “Ultrafast Single Error Correction Codes for Protecting Processor Registers”. In: *11th European Dependable Computing Conference, EDCC 2015, Paris, France, September 7-11, 2015*. IEEE Computer Society, 2015, pp. 144–154. DOI: 10.1109/EDCC.2015.30. URL: https://doi.org/10.1109/EDCC.2015.30.

[Sas+20] Pascal Sasdrich et al. “Low-Latency Hardware Masking with Application to AES”. In: *IACR Trans. Cryptogr. Hardw. Embed. Syst.* 2020.2 (2020), pp. 300–326. DOI: 10.13154/tches.v2020.i2.300-326. URL: https://doi.org/10.13154/tches.v2020.i2.300-326.

[SEH20] Olivier Savry, Mustapha El-Majihi, and Thomas Hiscock. "Confidaent: Control FLow protection with Instruction and Data Authenticated Encryption". In: *23rd Euromicro Conference on Digital System Design, DSD 2020, Kranj, Slovenia, August 26-28, 2020*. IEEE, 2020, pp. 246–253. DOI: 10.1109/DSD51259.2020.00048. URL: https://doi.org/10.1109/DSD51259.2020.00048.

[SHS16] Bodo Selmke, Johann Heyszl, and Georg Sigl. "Attack on a DFA Protected AES by Simultaneous Laser Fault Injections". In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography, FDTC 2016, Santa Barbara, CA, USA, August 16, 2016*. IEEE Computer Society, 2016, pp. 36–46. DOI: 10.1109/FDTC.2016.16. URL: https://doi.org/10.1109/FDTC.2016.16.

[Tal+22] Ezinam Bertrand Talaki et al. "A Memory Hierarchy Protected against Side-Channel Attacks". In: *Cryptogr.* 6.2 (2022), p. 19. DOI: 10.3390/cryptography6020019. URL: https://doi.org/10.3390/cryptography6020019.