HAL open science

Sécurisation matérielle de processeurs embarqués face aux attaques par injection de fautes

Noura Ait Manssour

To cite this version:

Noura Ait Manssour. Sécurisation matérielle de processeurs embarqués face aux attaques par injection de fautes. Systèmes embarqués. Université de Bretagne Sud, 2023. Français. NNT : 2023LORIS638 . tel-04091758v2

HAL Id: tel-04091758

https://hal.science/tel-04091758v2

Submitted on 16 Oct 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE DE DOCTORAT DE

L'UNIVERSITÉ DE BRETAGNE SUD

ÉCOLE DOCTORALE N° 644
Mathématiques et Sciences et Technologies
de l'Information et de la Communication en Bretagne Océane
Spécialité : *Informatique et Architectures Numériques*

Par
Noura AIT MANSSOUR

Sécurisation matérielle de processeurs embarqués face aux attaques par injection de fautes

Thèse présentée et soutenue à Lorient à l'Université Bretagne Sud, le 13 janvier 2023
Unité de recherche : Lab-STICC, UMR CNRS 6285
Thèse N° : 638

Rapporteurs avant soutenance :

Dr Karine HEYDEMANN	Maître de Conférences HDR à Sorbonne Université
Dr Pascal BENOIT	Maître de Conférences HDR à Université de Montpellier

Composition du Jury :

Président :	Pr Vincent BEROULLE	Professeur à Grenoble INP
Dir. de thèse :	DR Arnaud TISSERAND	Directeur de Recherche à CNRS Lab-STICC UMR 6285
Co-dir. de thèse :	Pr Guy GOGNIAT	Professeur à Université Bretagne Sud
	Dr Vianney LAPOTRE	Maître de Conférences à Université Bretagne Sud

Acknowledgement

Throughout the grueling years of research and writing for my thesis, I was blessed with the guidance and support of many people who played a vital role in its successful completion. I am forever grateful for their unwavering encouragement and would like to take this moment to express my sincerest appreciation for each and every one of them.

I would like to express my sincerest gratitude to my thesis director Arnaud TISSERAND. It is difficult to describe in a few phrases the invaluable assistance he provided throughout this journey. He guided me with kindness, corrected me with rigor, and even pushed me out of my comfort zone in order to progress in a constructive way. His demanding pedagogy and constant availability were essential for me to deepen and clarify my research. I can not imagine having completed this thesis without his support, engagement and guidance.
I also warmly thank my co-director Guy Gogniat for the trust he placed in me by supervising my thesis. His kind demeanor and benevolent perspective created a peaceful working environment that greatly contributed to the successful completion of my thesis.
I am deeply thankful to my co-advisor, Vianney Lapôtre, for generously sharing his insights and ideas with me. His support enabled me to explore new methodologies, leading to valuable discoveries. His guidance and advice were crucial in the successful completion of my thesis, and I am eternally grateful for his invaluable contributions.

I would like to thank Karine Heydemann and Pascal Benoit for agreeing to read and comment on my research work as referees for this thesis. I am also grateful to Vincent Beroulle for serving as an examiner and taking the time to thoroughly review my thesis. Their contributions were instrumental in ensuring the quality and completeness of my work.
Additionally, I also want to thank Karim Bigou, Benoit Gerard, and Sebastien Pillement for being a part of my personal follow-up committee/Comité de Cuivi Individual (CSI). Their scientific discussions and guidance were invaluable in shaping my thesis, and I greatly appreciate their efforts in monitoring my progress throughout the process.

I would like to extend my sincere gratitude to all the individuals I had the privilege of meeting during the course of my thesis. My colleagues at the Lab-STICC deserve special recognition for their warm welcome and for fostering a positive and productive work environment. I also appreciate the support provided by my friends, who were always available to offer encouragement and assistance, helping me persevere through the challenges. Lastly, I am deeply thankful to Ayoub Drissi for his invaluable support and guidance throughout my research journey.

I would also like to thank Beatrice Guern for being one of the most kind-hearted and welcoming people I have ever met. She has consistently demonstrated a warm and welcoming demeanor and has made significant efforts to assist international researchers and facilitate opportunities for networking and community building. Her contributions are greatly appreciated.

I would also like to express my sincere thanks to my family, who has been my pillar of support throughout my academic journey. I could not have achieved this milestone without their unwavering love and support. Their presence in my life has been integral to my success and I am deeply grateful. My father, who has been my mentor and guide since my childhood. His unwavering support and encouragement have been a source of strength and motivation for me. My mother's love and encouragement have also been an integral part of my journey. She has always been there for me. Her sacrifices and unrelenting support have been a guiding light, and I am deeply grateful for everything she has done for me.

I extend my heartfelt gratitude to my grandmother and grandfather for their unwavering encouragement and support in my pursuit of education and my dreams. I am also grateful to my brothers Ayoub and Hamza, and my sister Doaae, for their valuable presence and support throughout my journey.

In particular, I would like to warmly thank my dear brother, Oussama AIT MANSSOUR, for his unwavering support, constant love, and presence by my side which has been precious. His patience, listening, and wise advice have been a great help to me and I have learned a lot from him. I am very grateful for his support and commitment to me and I sincerely thank him for everything he has done for me. He is a true embodiment of what it means to be a brother, and I am truly fortunate to have him in my life.

Table des matières

1. Introduction **11**

1.1. Contexte . . . 11

1.2. État de l'art . . . 13

1.2.1. Attaques physiques . . . 13

1.2.2. Attaques par injection de fautes . . . 15

1.2.3. Contre-mesures contre les FIA . . . 18

1.3. Résumé des travaux . . . 26

1.3.1. Présentation du processeur cible . . . 26

1.3.2. Première contribution : Extension de processeur pour le rejeu matériel d'instructions contre FIA [3] . . . 29

1.3.3. Seconde contribution : nouvelles protections par rejeu . . . 31

2. Contribution 1 : Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks **35**

2.1. Introduction . . . 35

2.2. State of the Art . . . 36

2.3. Target Processor . . . 36

2.4. Proposed Hardware Replay Protection . . . 37

2.4.1. Replay Instruction . . . 37

2.4.2. Detection Elements . . . 38

2.4.3. Processor Modifications for Replay Support . . . 39

2.5. Processor Versions . . . 39

2.6. Evaluation Environment . . . 40

2.7. Evaluation of our Hardware Protections . . . 42

2.7.1. FPGA Implementations Results . . . 42

2.7.2. Timing Performance and Code Size Results . . . 43

2.7.3. Fault Injection Simulations Results . . . 44

2.7.4. Additional Discussions . . . 45

2.8. Conclusion and Future Prospects . . . 46

3. Contribution 2 : New Replay Protections **47**

3.1. Introduction . . . 47

3.2. Concepts Presentation . . . 49
3.2.1. Replay with Comparison (RC) . . . 49
3.2.2. Random Replay with Comparison (RRC) . . . 50
3.2.3. Triplication with Majority Vote (TV) . . . 50
3.2.4. Opcode and rpl Mode Protections . . . 51
3.2.5. Required Hardware Modifications . . . 53
3.2.6. Detection (DE) and Voting (VE) Elements . . . 53
3.2.7. Security Policy . . . 56
3.3. Processor Versions Implementation . . . 57
3.3.1. Processor for RC and RRC Versions . . . 57
3.3.2. Processor Versions for TV . . . 58
3.3.3. Complete Version (supports RC, RRC and TV) . . . 58
3.3.4. Versions with Refetch Support . . . 58
3.4. Description of the Evaluation Environment . . . 59
3.5. Analysis of Implementation and Simulation Results . . . 62
3.5.1. FPGA Implementation Results . . . 62
3.5.2. Timing Performance and Code Size Results . . . 65
3.5.3. Fault Injection Simulations Results . . . 67

4. Conclusion **73**

A. Appendix **75**
A.1. Definitions Used in the Tables . . . 75
A.2. Simulations for Processor Versions with Instruction Protection . . . 77
A.3. Simulations for Processor Versions without Instruction Protection . . . 83

Table des figures

1.1. Attaques physiques passives (inspiré de [22]). 14
1.2. Attaques par injection de fautes (inspiré de [22]). 15
1.3. Différents types d'impulsions sur les rails d'alimentation (tiré de [57]). 16
1.4. Tailles comparées du faisceau laser par rapport à la technologie cible (tiré de [21]). 18
1.5. Les effets d'une faute aux différents niveaux (tiré de [60]). 19
1.6. Duplication simple avec comparaison (tiré de [5]). 24
1.7. Réplication avec vote (trois blocs pour la triplication) (tiré de [5]). 24
1.8. Duplication complémentaire avec comparaison (tiré de [5]). 25
1.9. Architecture du processeur cible. 28
1.10. Format des instructions : champs et utilisations. 28
1.11. Catégories d'instructions de notre processeur. 29

2.1. Simplified schematic of the processor architecture (not all signals and elements are represented). Red elements are for replay protection. 37
2.2. Illustration of a typical software protection and our hardware replay protection in a small 4-instruction code ('I1-4' where 'I1-3' are protected using one replay and 'I4' is not). 38
2.3. The `rpl` instruction format. 38
2.4. Detection element (used for `prd`, `pid`, `pir` in Fig. 2.1). 39
2.5. Execution time and code size results. Values above the columns are : the number of cycles (top) ; and the number of instructions (bottom). Values on the right are averages. 44

3.1. The new `rpl` instruction format. 49
3.2. Illustration of random replay sub-modes in a small 4-instruction code ('I1', 'I2 and 'I3' are protected while 'I4' is left unprotected). 51
3.3. Illustration of a typical TV protection in software (left) or in hardware (right) for a small 3-instruction code ('I1' and 'I2' are protected while 'I3' is left unprotected). 52
3.4. Voting element (used for `tvid`, `tvir` in Fig. 3.5). 55
3.5. Simplified schematic architecture of TV version (not all signals and elements are represented). Red elements are for replay protection and VEs. 55
3.6. Simplified schematic architecture of RRC version (not all signals and elements are represented). Red elements are for replay protection and DEs. 57
3.7. Merged detection and voting element (used for `tvid/pid`, `tvir/pir` in Fig. 3.8). 59

3.8. Simplified schematic architecture of the complete version (not all signals and elements are represented). Red elements are for replay protection (DEs and VEs). 59
3.9. Evolution of the simulation metrics over 5000 random simulations. 71

Liste des tableaux

2.1. FPGA (Zynq 7020) implementation results for various versions (V) of the processor and protection elements. 43
2.2. Fault injection simulation results. The decimal values are ratios of number of executions where the protection solution (SW or HW) detects a fault over the total number of executions (integers in gray) for each code. 45
2.3. Distribution of the widths of replay windows in source codes and their corresponding execution traces. 45

3.1. The evaluated replay modes. 48
3.2. Modes supported by `rpl`. 49
3.3. The coding of different `rpl` modes. 53
3.4. FPGA (Zynq 7020) implementation results for all implemented versions of the processor (V). Each line corresponds to a specific configuration of the protection elements and hardware parameters used in the HDL code of the version. 63
3.5. Detailed performance results. 66
3.6. Average performances results summary. 67
3.7. Statistics (average and standard deviation) for computation time (CT) and number of cycles (NC) obtained using the random replay versions of the processor. 67
3.8. Fault injection simulations results for one single fault. 69
3.9. Simulation results summary for one single fault. 70
3.10. Simulation results summary for two faults at distance 1. 70
3.11. Simulation results summary for two faults at distance 2. 71

A.1. Performance results summary. 77
A.2. Simulation results summary for one single fault. 77
A.3. Simulation results summary for two faults at distance 1 (identical to Tab. 3.10). . 78
A.4. Simulation results summary for two faults at distance 2 (identical to Tab. 3.11). . 78
A.5. Detailed performance results. 79
A.6. Fault injection simulations results for one single fault (identical to Tab. 3.8). . . . 80
A.7. Fault injection simulations results for two faults at distance 1. 81
A.8. Fault injection simulations results for two faults at distance 2. 82
A.9. Simulation results summary for one single fault for replay protections only (without instruction coding protection). 83

A.10.Fault injection simulations results for one single fault for replay protections only (without instruction coding protection). 84

1. Introduction

1.1. Contexte

De nombreuses avancées technologiques ont un très grand impact sur notre mode de vie. Avec l'augmentation rapide de la technologie, les gens se tournent vers elle pour se rendre la vie plus facile. Parmi toutes ces technologies, l'Internet des objets (IoT) nous a déjà impacté. Il a affecté notre style de vie de la façon dont nous réagissons à la façon dont nous nous comportons, des climatiseurs que nous pouvons contrôler avec notre téléphone aux voitures intelligentes fournissant le chemin le plus court ou notre *smart watch* qui suit nos activités quotidiennes. L'IoT est un réseau géant d'appareils connectés. Ces appareils recueillent et partagent des données sur la manière dont ils sont utilisés et sur l'environnement dans lequel ils fonctionnent. Pour cela, ils se reposent sur des capteurs intégrés dans chaque appareil physique comme les téléphones portables, les appareils électriques, les feux de circulation et presque tout ce que nous rencontrons dans la vie quotidienne. L'IoT est une infrastructure de service mondiale dotée de capacité de connectivité et d'auto-configuration, basée sur des protocoles standards et interopérables. Il se compose d'objets hétérogènes qui ont des identités et des attributs physiques et virtuels, et sont intégrés de manière transparente à l'Internet.

L'objectif de l'IoT est de permettre aux objets de se connecter à tout moment, n'importe où, avec n'importe quoi et n'importe qui idéalement en utilisant n'importe quel réseau. L'IoT a créé de nombreuses applications qui touchent différents aspects de la vie humaine. On cite à titre d'exemples : les appareils portables, la surveillance de la santé, les applications militaires comme la détection d'intrusion dans la surveillance d'environnements distants et hostiles, les maisons et les villes intelligentes, les réseaux intelligents et les voitures connectées.

L'évolution de ce domaine est due à la convergence de plusieurs technologies, y compris l'informatique ubiquitaire, les capteurs de commodité, le *machine learning* et surtout des systèmes embarqués qui deviennent de plus en plus performants.

Les systèmes embraqués sont des systèmes matériels informatiques basés sur un microprocesseur avec un logiciel conçu pour exécuter une fonction dédiée, soit en tant que système indépendant ou en faisant partie d'un système complexe. Au centre, se trouve un circuit intégré conçu pour effectuer des opérations éventuellement en temps réel. Les principaux composants de ces systèmes sont une combinaison de matériels et de logiciels, généralement : les microprocesseurs ou les microcontrôleurs, les unités de traitement graphique (GPU), la mémoire volatile ou non-volatile, les interfaces et les ports de communication d'entrée-sortie, le code du système et de l'application et la gestion d'énergie.

Ces systèmes ont attiré beaucoup d'attention, d'autant plus que leur forme devient de plus en plus diversifiée et omniprésente. Allant des routeurs, ordinateurs, téléphones, cartes à puce, aux cartes de débit et de crédit, véhicules intelligents et appareils portables, les systèmes embarqués actuels fournissent souvent des fonctionnalités critiques susceptibles d'être sabotées et de causer de graves dommages. L'évolution rapide de ces systèmes qui ne cessent de progresser et la sensibilité des informations manipulées font l'objet de plusieurs menaces, ce qui a fait de la sécurité des systèmes embarqués un domaine critique qu'il faut prendre en charge afin de protéger les données des utilisateurs et garantir le bon fonctionnement des applications basées sur les systèmes embarqués.

Avec une grande variété d'outils et de technologies, les attaquants peuvent s'introduire dans les systèmes embarqués via l'exploitation du matériel, du logiciel ou du réseau, et causer des conséquences indésirables telles qu'un dysfonctionnement du système, un abus de privilège, une fuite d'information, ou un déni de service.

Les attaques logicielles les plus répandues comprennent les logiciels malveillants, la force brute, le débordement de la mémoire (*Memory Overflow*), et l'exploitation des vulnérabilités de sécurité des applications Web. Quant au côté réseau, les attaques les plus courantes sont : personne au milieu (PITM), l'empoisonnement du système de noms de domaines (DNS), le déni de service distribué (DDoS), le détournement de session et le brouillage du signal (*Signal Jamming*). Au niveau matériel, les attaques les plus courantes comprennent : les attaques par injection de fautes et l'analyse de canaux auxiliaires (l'analyse de puissance, l'exploitation des variations de temps d'exécution, rayonnement électromagnétique, ...).

Afin de lutter contre ces attaques, des recherches sont menées afin de protéger la confidentialité, l'intégrité, l'authenticité et la disponibilité des données traitées à différents niveaux (protocoles de communication, chaîne de développement logiciel, systèmes d'exploitation, architectures matérielles...). Au fil des années, plusieurs solutions de protection ont été proposées telles que les algorithmes de chiffrement (RSA, AES ...) afin d'assurer la sécurité des données échangées et des calculs effectués. Néanmoins, ces algorithmes sont basés sur des structures et des opérations mathématiques complexes, nécessitant donc un temps de calcul important et beaucoup de ressources. De plus, malgré leur sécurité mathématique, ils peuvent être vulnérables à des attaques physiques lorsque l'attaquant a un accès direct à des grandeurs physiques du circuit comme sa tension d'alimentation. Ce type d'attaques analyse des mesures de ces grandeurs physiques durant le fonctionnement du circuit afin de révéler des données secrètes ou de contourner des primitives de sécurité. En outre, le processus de certification de sécurité exige une analyse de sécurité contre les attaques physiques, notamment les attaques par injection de fautes. En effet, [1] précise qu'à partir d'un certain niveau de certification *(Evaluation Assurance Level (EAL))*, l'évaluation des cartes à puce ou des circuits intégrés est réalisée en fonction de plusieurs critères : le temps d'attaque, le niveau d'expertise de l'attaquant, la connaissance et l'accès au circuit et l'instrumentation de l'attaquant. De plus, l'équipement standard utilisé pour une évaluation des menaces sur des circuits intégrés est, entre autres, basée sur des moyens d'attaques

par injection de fautes.

De nombreux travaux proposent des techniques pour protéger le flux de contrôle contre les attaques par injection de fautes à tous les niveaux du processeur. En revanche, peu d'entre eux proposent une protection du flux de données, en particulier au niveau de l'architecture. Pour cette raison, le sujet de cette thèse s'inscrit dans le cadre de ce thème de recherche, et se focalise essentiellement sur la proposition de protections matérielles face aux attaques par injection de fautes, afin de garantir l'intégrité du flux de données manipulé par un processeur.

1.2. État de l'art

Cette section résume l'état de l'art afin de pouvoir réaliser les objectifs généraux de la thèse. D'abord, nous présentons les grandes familles d'attaques physiques. Ensuite, nous décrivons les moyens d'injection et de modélisation de fautes. Puis, nous introduisons les contre-mesures logicielles et matérielles contre certaines attaques par injection de fautes. Nous terminons par une introduction du concept de rejeu d'instructions.

1.2.1. Attaques physiques

Les attaques physiques se composent de deux grandes familles à savoir les attaques passives et les attaques actives.

Les attaques passives consistent à exploiter des informations sur le comportement d'un circuit comme sa consommation d'énergie, son rayonnement électromagnétique ou le temps d'exécution d'une application comme illustré sur la figure 1.1 inspirée du site web [22]. L'observation et l'analyse de ces informations permettent à l'attaquant de dévoiler des données. Par exemple, la clé de chiffrement/déchiffrement d'un système cryptographique est souvent stockée sur une puce dans une mémoire non-volatile, telle qu'une *Read-Only Flash*. Cependant, le contenu de la clé peut être dévoilé lors de son utilisation par le circuit (p. ex. pendant l'exécution d'un algorithme de chiffrement) car sa valeur influencera le comportement du circuit. Une fois la clé révélée, tout le mécanisme d'authentification pourra être contourné.

En 1996, une première attaque par analyse de canaux auxiliaires (*Side Channel Attacks* (SCA)) consistant à observer le temps d'exécution des algorithmes afin d'extraire des données secrètes (p. ex. clés de chiffrement) a été introduite dans [32]. Ensuite, des méthodes plus sophistiquées de SCA ont été proposées pour améliorer leur efficacité : l'analyse différentielle de puissance (*Differential Power Analysis* (DPA)) proposée en 1999 dans [31] qui est basée sur la comparaison de la consommation électrique du circuit cible lors la manipulation des secrets avec un modèle pour différentes valeurs de ce secret afin de trouver la distance minimale entre la trace collectée et le modèle ; l'analyse de corrélation de puissance (*Correlation Power Analysis* (CPA)) proposée en 2004 dans [11] est basée sur le coefficient de corrélation de Pearson entre la consommation d'énergie modélisée et les mesures physiques.

Les attaques sur la microarchitecture sont un autre type d'attaques passives qui exploitent

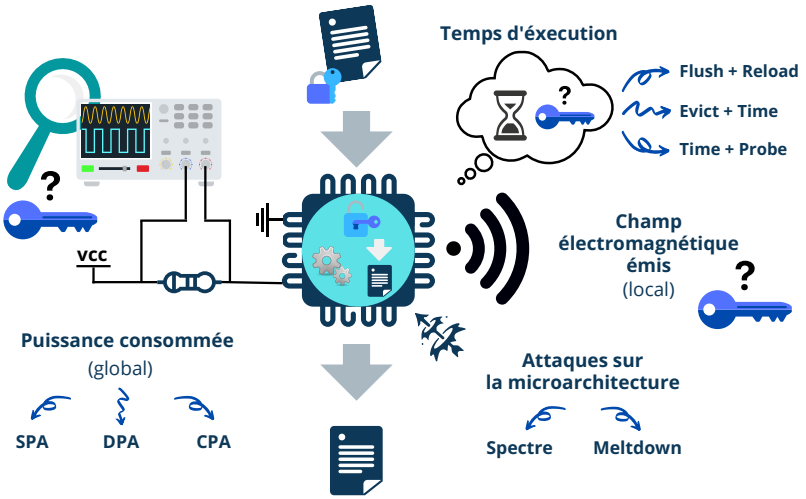


FIGURE 1.1. – Attaques physiques passives (inspiré de [22]).

les optimisations des processeurs modernes, telles que l'exécution spéculative, l'exécution non-ordonnée et la hiérarchie mémoire. Les attaques les plus populaires de ce type sont *Meltdown* [39] et *Spectre* [30] introduites respectivement en 2018 et en 2019.

Les attaques actives consistent à perturber le circuit cible lors de son fonctionnement et causer des erreurs d'exécution afin de l'amener à un état vulnérable et exploitable par l'attaquant. Cette attaque est appelée attaque par injection de fautes (*Fault Injection Attacks* (FIA)) ou par perturbation. En effet, afin qu'un circuit fonctionne correctement, il doit respecter des contraintes physiques liées à sa conception : notamment une tension nominale, une plage de température, une fréquence maximale . . . Ces contraintes peuvent être perturbées en modifiant des grandeurs physiques du circuit cible lors de son fonctionnement.

En 1997, l'article [7] a introduit théoriquement l'attaque par injection de fautes matérielles comme un nouveau moyen pour retrouver des secrets manipulés dans des cryptosystèmes asymétriques tels que RSA. L'idée consiste à éviter de casser les cryptosystèmes mathématiquement (p. ex. devoir factoriser le module ce qui est infaisable en pratique pour des tailles actuelles de clés). Elle est basée sur une méthode d'exploitation de calculs erronés ou des valeurs de registre corrompues. Cette attaque repose sur le fait que le matériel n'a aucun indicateur de fautes injectées par l'attaquant. Les attaques par injection de fautes peuvent être exploitées afin d'extraire des données sensibles [8, 16], de contourner des procédures d'authentification [17] ou d'obtenir plus de privilèges [53].

Les injections de fautes se présentent sous deux formes : *permanentes*, où les perturbations des grandeurs physiques endommagent ou dégradent le circuit en permanence et peuvent invalider une protection de manière définitive ; *transitoires*, où le système cible est momentanément perturbé pendant son fonctionnement et reprend sa fonctionnalité normale une fois réinitialisé.

C'est ce dernier type de fautes qui sera au cœur de la thèse.

1.2.2. Attaques par injection de fautes

Différentes méthodes d'injection ont été proposées dans l'état de l'art [5], notamment la perturbation d'horloge, la perturbation de la tension d'alimentation, la perturbation de la température, le rayonnement lumineux par laser et l'injection d'impulsions électromagnétiques (*Electromagnetic perturbations* (EM)). La figure 1.2 inspirée du site web [22] illustre ces différentes méthodes. Chacune requiert une instrumentation matérielle particulière et une configuration dont le coût dépend du type de technique d'injection. En outre, ces techniques nécessitent des niveaux de précision dans le temps (instant et durée d'injection) et l'espace selon la méthode utilisée. Dans le reste de cette section, nous résumons les principales méthodes d'injection de fautes et leur modélisation.

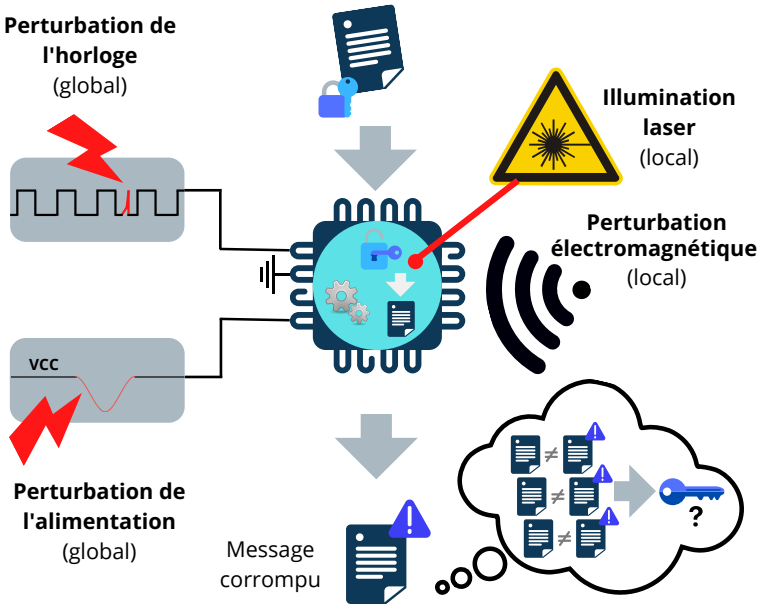


FIGURE 1.2. – Attaques par injection de fautes (inspiré de [22]).

Les perturbations de l'horloge sont des variations locales de la fréquence d'horloge du circuit cible. Afin qu'un circuit synchrone fonctionne dans de bonnes conditions, il faut respecter la contrainte de la fréquence maximale pour que les opérations logiques s'effectuent avant le front d'horloge. Et donc, une variation de la fréquence d'horloge peut provoquer des fautes à certains cycles d'horloge, et par conséquent, cela peut se traduire par des erreurs dans des calculs ou dans des transferts mémoires.

En pratique, l'attaquant optera pour une variation temporaire de la période durant un ou plusieurs cycles choisis par l'attaquant en utilisant des impulsions créées sur le signal d'horloge. L'idée est de réduire la période d'un cycle de manière progressive jusqu'à ce qu'une faute apparaisse.

Les articles [4, 2] mentionnent que le fait de raccourcir la période d'un dispositif synchrone provoque une modification d'un ou de plusieurs octets d'une clé de chiffrement. En revanche, pour pouvoir faire varier la période du système, il faut impérativement avoir un accès direct au signal d'horloge. Dans [2], les cartes à puce sont attaquées par ce type de perturbation à cause de leur horloge pilotée de l'extérieur.

Les perturbations de la tension d'alimentation consistent à faire varier la tension d'alimentation d'un circuit cible par rapport à sa tension nominale pendant un instant fixé. La figure 1.3 extraite de l'article [57] illustre différents types d'impulsions sur les rails de tension d'alimentation.

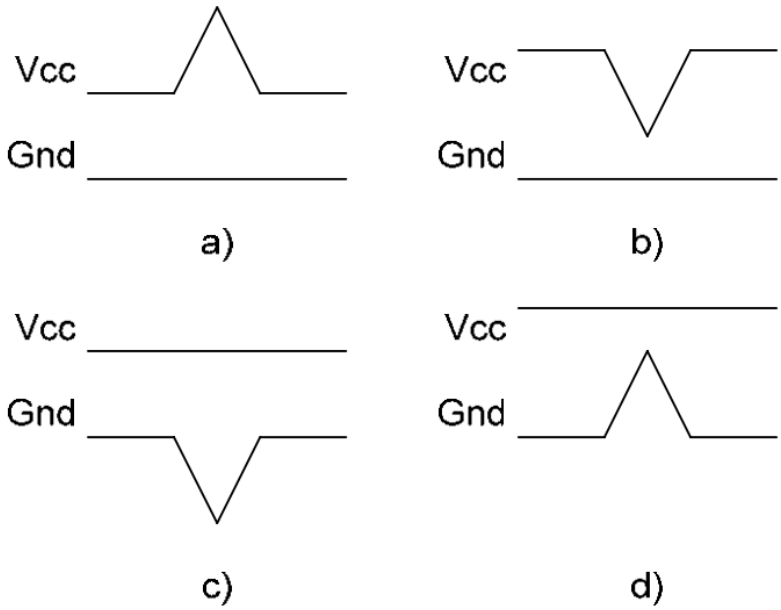


FIGURE 1.3. – Différents types d'impulsions sur les rails d'alimentation (tiré de [57]).

La tension d'alimentation d'un dispositif est contrainte afin de garantir le bon fonctionnement du circuit. Cette contrainte dépend directement de la technologie des composants. Et donc, une sous-alimentation ou une sur-alimentation a un impact direct sur la vitesse (contraintes temporelles) de transfert des données traitées par le circuit cible. Par conséquent, un attaquant est capable de modifier des données traitées par certaines portes logiques ou d'écraser une valeur stockée dans un registre. Et donc, des calculs peuvent être erronés durant l'exécution. [21] montre expérimentalement l'impact de la variation de tension sur un circuit implémenté sur FPGA. De plus, cette perturbation peut aussi fauter des calculs intermédiaires d'algorithmes de chiffrement comme l'*Advanced Encryption Standard* (AES) [61].

Les perturbations de tension nécessitent également un accès au dispositif afin de réaliser une attaque à faible coût. C'est pour cela que cette attaque est généralement exploitée dans un contexte où le circuit possède une alimentation externe, comme des anciennes cartes à puce [33] ou des dispositifs IoT [10].

Les perturbations de température s'appuient sur le même principe que celui de la pertur-

bation de la tension. En effet, des variations de température peuvent aussi provoquer des erreurs de synchronisation en modifiant le temps de propagation de la logique [34]. D'ailleurs, dans [50], une variation de température locale d'une mémoire EPROM peut modifier son contenu et effacer des bits de cette dernière.

En revanche, ce moyen d'attaque présente des limitations en précision à la fois temporelle et spatiale. Autrement dit, chauffer ou refroidir un dispositif prend un temps important en raison d'inertie thermique par rapport à la vitesse du calcul du dispositif.

Les injections d'impulsions électromagnétiques (EM) utilisent un champ électromagnétique à la surface du circuit cible afin de générer des courants sur des fils électriques spécifiques et de perturber le fonctionnement du dispositif. En 2002, [48] a introduit ce moyen d'attaque en décrivant l'usage d'une sonde induisant un fort champ magnétique transitoire sur un microprocesseur. Cette technique a été appliquée dans l'article [49] afin de contourner un algorithme de chiffrement RSA. Les travaux [14] ont réussi à perturber chacun des 16 octets du 10ème tour d'un AES exécuté dans un microcontrôleur 8 bits en technologie 0,35 μm par une injection d'une impulsion de 50 V d'amplitude durant 20 ns. En fait, de nombreuses attaques basées sur l'injection d'EM ont été présentées dans la littérature en ciblant des algorithmes cryptographiques dans divers contextes [16, 15, 18, 27].

Le rayonnement lumineux ou les injections de lumière focalisée, consistent généralement à injecter une faute avec un laser dans un circuit. Cette méthode a été adoptée pour la première fois dans [51]. Des précisions sur le phénomène photoélectrique expliquant la perturbation sont fournies dans [19]. Le rayonnement lumineux peut provoquer la conductivité de transistors cibles et ainsi perturber le fonctionnement du circuit. Afin de perturber un seul ou un ensemble de transistors, il faut choisir soigneusement le diamètre du faisceau lumineux adapté à la technologie cible, comme illustré dans la figure 1.4 tirée de la présentation [21].

Cette technique est plus précise et efficace que les injections électromagnétiques. En revanche, elle nécessite une préparation du circuit et une certaine expertise de la part de l'attaquant, car elle requiert un accès direct à la puce au niveau silicium [20] (décapsulation du circuit).

La modélisation d'injection de fautes

La modélisation de menaces précises est un processus nécessaire pour identifier les contextes d'attaque, les hypothèses des vecteurs d'attaque et les vulnérabilités du système. En fait, les attaquants doivent comprendre la nature de la modification induite par des injections de fautes afin de bien exploiter les effets de l'attaque. De plus, les concepteurs doivent comprendre ces effets afin de proposer des contre-mesures efficaces. Les effets des fautes peuvent être modélisés à différents niveaux d'abstraction : les effets au niveau du code source sont modélisés par la corruption du flux de données et du flux de contrôle du programme en cours d'exécution ; les effets au niveau ISA peuvent être modélisés comme une corruption du registre, des remplacements d'instruction ou des sauts d'instruction ; les effets au niveau logique se manifestent au niveau du circuit via la corruption de bits dans une bascule ou une porte logique ; les effets au niveau

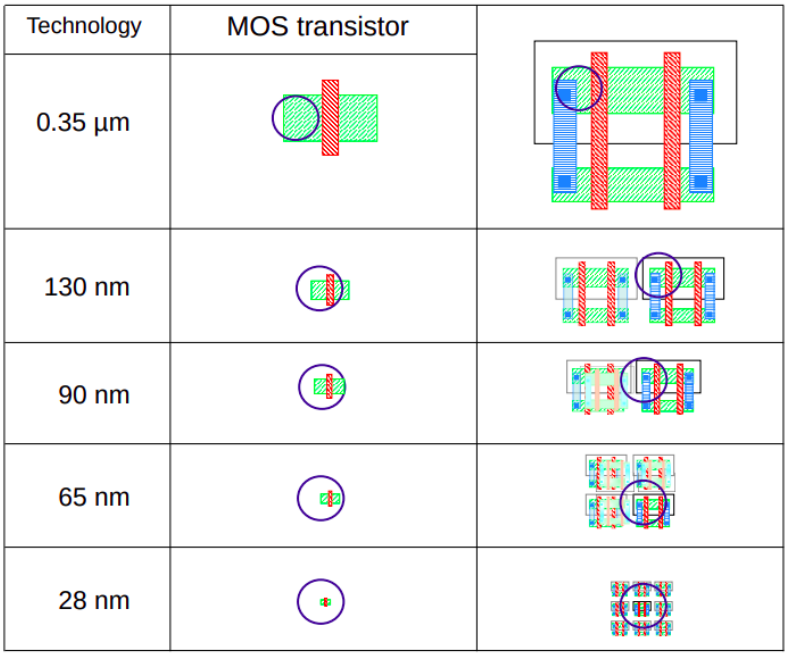


FIGURE 1.4. – Tailles comparées du faisceau laser par rapport à la technologie cible (tiré de [21]).

physique peuvent être modélisés comme une génération de courant induit, une violation de contraintes temporelles ou une altération de niveaux électriques. La figure 1.5 tirée de l'article [60] illustre les différents niveaux d'abstraction.

Le modèle de fautes a deux paramètres importants : l'emplacement et l'effet. L'emplacement comprend l'emplacement temporel et spatial de l'injection de faute par rapport à l'exécution du programme. L'impact dépend du type et de la granularité de la technique utilisée pour injecter la faute. Le type de faute peut prendre trois formes : à savoir le *bit-set* où la valeur est mise à 1, le *bit-reset* où la valeur est mise à 0, ou le *bit-flip* où la valeur est inversée. La granularité peut être un bit, un octet ou un mot de plusieurs octets.

Les résultats d'injection de fautes lors de l'exécution d'un programme sont généralement classés en cinq catégories : les sorties du programme sont erronées ; l'absence d'effet des fautes par rapport à une exécution de référence (p. ex. un *bit-reset* d'une bascule déjà à zéro) ; aucun effet sur les sorties du programme, en revanche l'état d'exécution du programme ne concorde pas avec la référence (p. ex. le registre fauté n'est pas utilisé lors de l'exécution du programme) ; la divergence de programme (p. ex. instruction invalide, division par zéro . . .) ; la non-terminaison du programme (p. ex. crash du système).

1.2.3. Contre-mesures contre les FIA

Les attaques FIA reportées dans l'état de l'art ciblent souvent une clé cryptographique, un micrologiciel, le stockage de données, la configuration et l'identification du système. Ces éléments ont besoin d'être protégés par les concepteurs de sécurité.

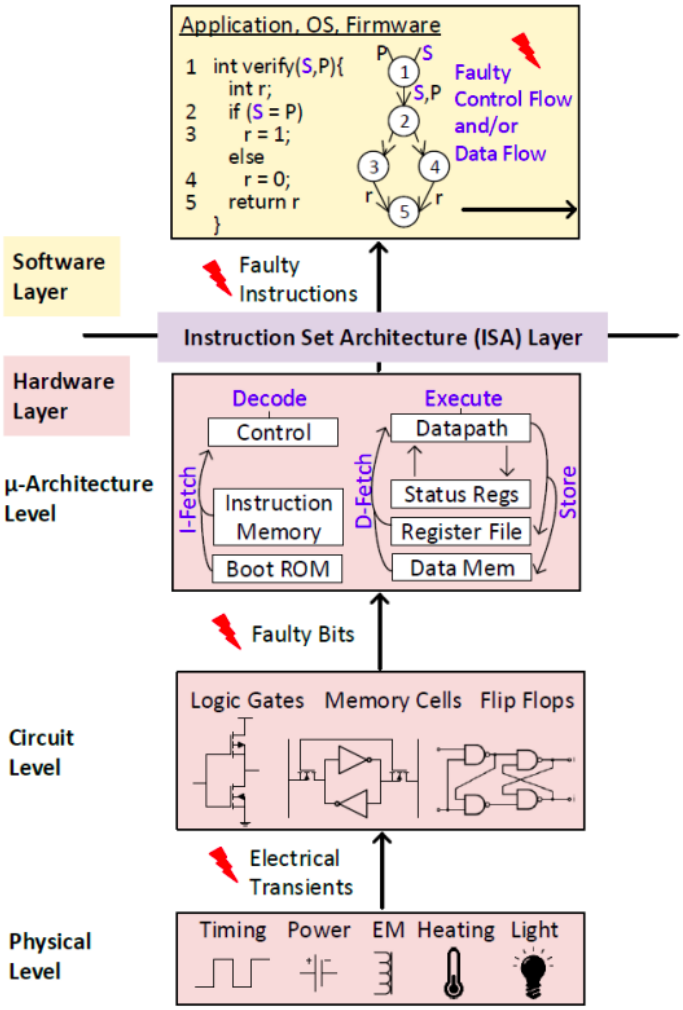


FIGURE 1.5. – Les effets d'une faute aux différents niveaux (tiré de [60]).

Les objectifs des contre-mesures sont : détecter les effets de fautes et faire réagir le processeur conformément à la politique de sécurité (p. ex. une réinitialisation du système) ; tolérer les erreurs ou les pannes même en présence d'une injection de faute et tenter de les corriger ; rendre les résultats erronés non-exploitables par l'attaquant. Une contre-mesure peut être implémentée au niveau logiciel, matériel ou hybride.

Les contre-mesures logicielles ciblent les parties vulnérables du code (boucles, accès mémoire...). Elles sont souvent génériques et relativement faciles à implémenter par rapport aux contre-mesures matérielles. Cependant, elles sont plus susceptibles d'être contournées, car leur implémentation ne tient pas compte de la microarchitecture du système comme le pipeline du processeur et ses optimisations (le *forwarding* et l'exécution spéculative) [59, 35]. En revanche, les contre-mesures matérielles consistent à rajouter des mécanismes matériels de protection à l'architecture du système ; de ce fait, elles sont plus efficaces.

L'ajout d'une contre-mesure introduit une pénalité de performance dans le système cible, car

leur mise en œuvre implique généralement une augmentation de la taille du matériel, une diminution de la fréquence maximale ou une augmentation de la taille du code en mémoire. Par conséquent, les concepteurs de sécurité sont obligés de proposer une protection sans que les performances ne soient trop pénalisées surtout dans un cadre applicatif à faible coût (p. ex. une protection avec une réduction substantielle de la fréquence). Par ailleurs, une fois que l'implémentation de la protection est mise en place, une évaluation des différents compromis entre les performances, la robustesse aux attaques, le coût du silicium et/ou des mémoires, et l'énergie consommée est nécessaire.

Dans l'état de l'art, il existe diverses méthodes de protection contre les FIA [28], notamment la randomisation, les codes correcteurs d'erreurs et la redondance temporelle/spatiale.

— **La randomisation des calculs** consiste à compliquer davantage l'exploitation des erreurs causées par l'attaquant grâce à des schémas aléatoires. En fait, l'attaquant n'a pas assez d'information sur l'exécution du programme afin de synchroniser ses injections. Par conséquent, il est peu probable qu'un attaquant réussisse à injecter des fautes aux bons moments, ce qui rend difficile le fait de cibler une instruction sensible (comme un branchement). Nous utiliserons cette protection dans le chapitre 3.

— **Les codes détecteurs/correcteurs d'erreurs** consistent à détecter ou à corriger des erreurs. Ils sont basés sur l'ajout d'un ou plusieurs bits de contrôle à chaque donnée traitée par le processeur. Il existe plusieurs codes de correction d'erreurs, notamment les codes de Golay et les codes cycliques, mais l'un des plus connus est le code de Hamming. Cette protection est généralement appliquée dans le banc de registres et les mémoires, mais elle peut également être utilisée pour protéger les registres de pipeline du processeur ou l'unité arithmétique et logique comme présenté dans le travail [23].

— **La redondance temporelle/spatiale** consiste à rejouer les calculs plusieurs fois séquentiellement ou en parallèle afin de protéger le flux de données. Dans cette thèse, nous nous intéressons aux contre-mesures basées sur la redondance temporelle.

Dans la suite de cette section, nous détaillons le principe de redondance temporelle et ses différentes implémentations en logiciel et en matériel dans des processeurs.

La redondance temporelle logicielle

La redondance temporelle permet de protéger le flux de données à travers des schémas de tolérance aux fautes [43] ou de détection de fautes [6]. La redondance temporelle avec détection de faute consiste principalement à re-exécuter plusieurs fois de suite les mêmes opérations sur un processeur et à comparer leurs résultats. Et si ces résultats sont différents, c'est qu'au moins une des exécutions a été fautée.

Cette contre-mesure est facile à mettre en place, car il suffit de rejouer les opérations plusieurs fois et de détecter les effets de fautes via des comparaisons. De plus, elle est efficace dans la mesure où elle est capable de détecter de nombreuses attaques ayant un impact sur le résultat final si l'architecture est simple. Cependant, elle engendre un surcoût important en terme de

temps d'exécution et de taille de code. Le temps d'exécution peut être multiplié au moins par le nombre d'exécutions rejouées.

La redondance temporelle logicielle peut s'appliquer à différents niveaux : au niveau du programme ; au niveau de l'algorithme dont l'exécution est effectuée plusieurs fois successivement ; au niveau des instructions où chaque instruction est exécutée plusieurs fois de suite.

L'article [6] indique qu'un attaquant peut facilement générer la même faute lors d'exécutions successives d'une même fonction ou d'un algorithme avec la même implémentation. Par conséquent, il peut provoquer les mêmes effets d'attaques lors des différentes exécutions, et donc la comparaison des résultats n'est plus en mesure de détecter les fautes. Ce niveau de redondance est donc un peu plus compliqué et nécessite un rejeu du même algorithme avec différentes implémentations. D'autre part, les auteurs de [6] constatent que les dispositifs d'injection de fautes à faible coût (en 2010) ne permettaient pas d'injecter des fautes successives très rapprochées dans le temps (certains documents de l'état de l'art cités dans [6] évoquent des délais d'au moins 3 à 10 cycles entre deux injections). Pour cette raison, la redondance au niveau des instructions pourrait être plus efficace et une meilleure méthode d'application de la redondance temporelle. Mais de nouveaux dispositifs d'injection (p. ex. lasers multi-têtes) peuvent remettre en cause cela, et le fait de pouvoir injecter plusieurs fautes ne doit plus être négligé. De plus, si une simple faute dans une architecture simple n'a qu'un effet très localisé (et donc détectable), dans le cas d'un processeur complexe, une simple faute peut impacter plusieurs instructions en cours de traitement dans le pipeline.

La redondance au niveau instruction fait l'objet principal de cette thèse. [6] présente deux catégories de cette forme de protection en logiciel à savoir la duplication d'instructions avec comparaison et la triplication d'instructions avec vote majoritaire.

La duplication d'instructions avec comparaison (ID) rejoue l'instruction cible deux fois avec des registres de destination différents, compare leur contenu et lève une interruption s'ils sont différents. L'exemple suivant illustre l'insertion de cette contre-mesure logicielle présentée dans l'article [6]. Dans la partie gauche de l'exemple, nous avons un code non protégé, consistant en une instruction d'addition `add` (une addition entre `r1` et `r2` avec un résultat stocké dans `r4`). De l'autre coté, l'instruction est rejouée deux fois (lignes 1 et 2) avec des registres de destination différents (`r4` et `r10`). Ensuite, les deux registres sont comparés et une interruption logicielle est levée si une différence est détectée (lignes 3 et 4).

```
1. add r4 r1 r2
```

```
1. add r4 r1 r2
2. add r10 r1 r2
3. cmp r4 r10
4. bneq <fault>
```

La duplication d'une instruction requiert parfois une transformation d'instructions dans le cas où le registre de destination est le même que l'un des registres sources (une instruction non-idempotente) comme mentionné dans [43]. L'exemple suivant illustre la duplication de l'ins-

truction non-idempotente “`add r1 r1 r2`”. La transformation nécessite l’utilisation d’une instruction supplémentaire `mov` afin de pouvoir produire le même résultat lors de la comparaison.

```
1. add r1 r1 r2
```

```
1. mov r10 r1
2. add r1 r1 r2
3. add r10 r10 r2
4. cmp r10 r1
5. bneq <fault>
```

La triplication d’instructions avec vote majoritaire (IT) rejoue l’instruction cible trois fois et corrige la faute lorsque le contenu des trois registres diffère à l’aide d’un système de vote majoritaire. L’objectif de la triplication avec vote est d’essayer de corriger une faute dans le cas où une seule différence est détectée.

L’exemple de code suivant tiré de l’article [6] illustre l’insertion de la triplication d’instructions avec vote majoritaire afin de protéger l’opération ou-exclusif (`eor r4 r1 r2`) présenté en partie gauche ci-dessous.

L’instruction est exécutée trois fois avec les trois registres de destination `r4`, `r10` et `r0` (lignes 2–4). De plus, un quatrième registre `r12` est utilisé afin de stocker le résultat des différentes comparaisons de vote (lignes 5–10). Les lignes de 11 à 14 déterminent s’il faut corriger une seule erreur (lignes 13–14), détecter deux erreurs (lignes 11–12) ou ne pas modifier le résultat de l’opération ou-exclusif (pas de différence détectée).

```
1. eor r4 r1 r2
```

```
1. eor r12 r12 r12
2. eor r10 r1 r2
3. eor r0 r1 r2
4. eor r4 r1 r2
5. cmp r4 r10
6. eoreq r12 r12 #1
7. cmp r4 r0
8. eoreq r12 r12 #2
9. cmp r10 r0
10. eoreq r12 r12 #4
11. cmp r12 #0
12. beq <fault>
13. cmp r12 #4
14. moveq r4 r0
```

Les deux types de redondance temporelle d’instructions présentent des limitations en terme de surcoût engendré. L’article [6] confirme à travers des résultats d’évaluation du temps d’exécution que la duplication des chargements seuls est la moins coûteuse. Elle engendre un surcoût d’un facteur de 1.5 à 2 environ selon la latence mémoire (2 à 64 cycles d’horloge de chargement

respectivement) et selon le nombre de rondes d'AES protégées (les trois dernières ou toutes respectivement). La triplication est recommandée pour de forts besoins de sécurité, car elle est bien plus coûteuse (surcoût d'un facteur de 3 à 5 fois par rapport à l'implémentation non protégée). De plus, le surcoût en taille par instruction cible est mentionné : 4 pour ID (duplication d'instructions), 14 pour IT (triplication d'instructions). Mais cela ne tient pas compte du code à ajouter lors d'éventuelles saturations du banc de registres par les registres additionnels de duplication/triplication. De plus, le surcoût dépend du nombre d'instructions protégées et donc de l'algorithme lui-même.

La robustesse "théorique" des protections est déduite directement des hypothèses de possibilité d'injection de fautes successives. Par exemple, ID demandant 4 instructions, alors si l'attaquant est incapable d'injecter deux fautes dans cette fenêtre de temps, alors 100% des fautes seront détectées. Cependant, les articles ultérieurs [59] et [42] montrent expérimentalement que ces hypothèses de robustesse des contre-mesures proposées dans [6] sont insuffisantes, car elles ne tiennent pas compte de certains aspects microarchitecturaux comme le pipeline.

L'article [59] montre qu'une seule faute peut impacter non pas une instruction (comme les hypothèses de [6] le suggèrent) mais plusieurs instructions en cours d'exécution dans les différents étages du pipeline. Dans l'article [59], une campagne expérimentale d'injection de fautes est effectuée sur un algorithme de chiffrement par blocs LED-64 exécuté sur un processeur LEON3 implanté en FPGA (Spartan-6 LX9). Les fautes injectées sont des perturbations d'horloge générées par un autre FPGA. L'analyse différentielle de l'intensité de fautes DFIA proposée dans [24] permet de dévoiler la clé secrète.

En particulier, les auteurs de [59] constatent qu'une injection de fautes dans l'étage de chargement d'instructions ou celui de décodage peut modifier le flot de contrôle, tandis qu'une injection dans les étages d'accès aux registres, d'exécution, d'accès à la mémoire ou de rangement du résultat peut modifier les données du banc de registres.

Protections par redondance matérielle

La redondance matérielle est basée sur le même concept que la redondance temporelle logicielle mais les opérations sont effectuées plusieurs fois en parallèle, et non en série dans le temps. Généralement, les blocs matériels qui exécutent les opérations cibles sont instanciés dans le circuit deux ou trois fois pour réaliser une duplication ou une triplication comme illustré dans les figures 1.6 et 1.7 tirées de [5]. La duplication nécessite l'ajout d'un circuit de comparaison pour détecter les erreurs (figure 1.6), mais dans le cas de la triplication, un circuit de vote majoritaire est mis en œuvre pour détecter les erreurs et les corriger si nécessaire (figure 1.7).

La redondance matérielle génère un surcoût important en terme de taille de circuit en la multipliant par le nombre souhaité de redondances au lieu de multiplier le temps d'exécution comme dans la redondance temporelle logicielle. De plus, l'ajout du composant de comparaison ou de vote peut diminuer la fréquence maximale du circuit. En outre, un attaquant sera capable de contourner cette contre-mesure si il est capable d'injecter la même faute dans deux blocs

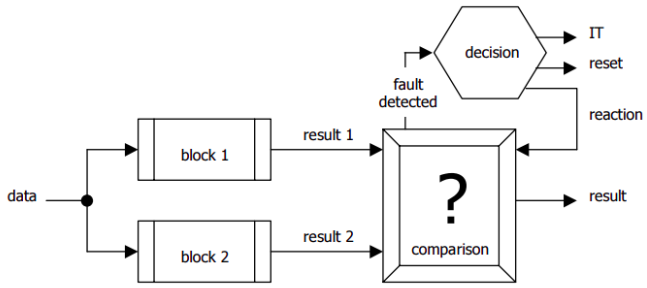


FIGURE 1.6. – Duplication simple avec comparaison (tiré de [5]).

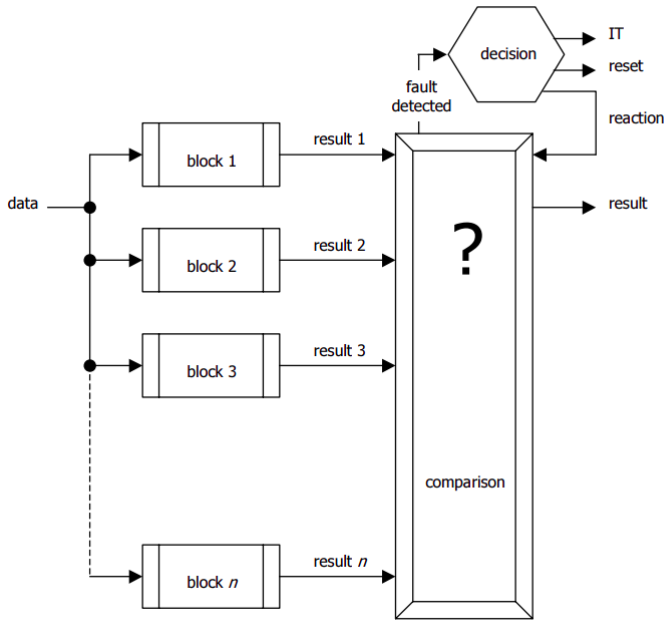


FIGURE 1.7. – Réplication avec vote (trois blocs pour la triplication) (tiré de [5]).

matériels identiques. C'est pour cela que dans l'état de l'art, nous trouvons plusieurs versions d'implémentation de la redondance matérielle, y compris l'utilisation des données complémentaires comme illustré sur la figure 1.8 tirée de l'article [5]. Cette approche permet une meilleure détection dans le cas où la même faute est introduite dans les deux blocs de calcul.

Comme nous l'avons mentionné précédemment, les contre-mesures matérielles sont plus efficaces que les contre-mesures logicielles, car elles tiennent compte de la microarchitecture du circuit. En revanche, dans certains cas, comme dans le domaine de l'IoT, le fait qu'un bloc de calcul soit dupliqué matériellement plusieurs fois peut poser de sérieuses limitations.

L'objectif de ces travaux de thèse est de proposer une solution d'implémentation de la redondance matérielle (un rejeu d'instructions) avec un surcoût acceptable et adapté aux applications à faible coût car la redondance matérielle est limitée à des petites parties de l'architecture.

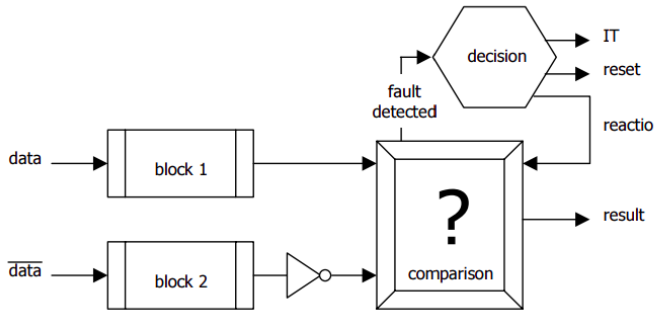


FIGURE 1.8. – Duplication complémentaire avec comparaison (tiré de [5]).

Le rejeu d'instructions

L'implémentation de mécanismes matériels de rejeu d'instructions dans les processeurs est utilisé dans l'état de l'art pour plusieurs raisons et depuis de nombreuses années.

Les travaux [29] introduisent ce mécanisme afin de résoudre des problèmes de spéculation à haute performance. Cet article propose un nouveau schéma de rejeu sélectif afin de corriger les cas où les instructions sont exécutées avec des opérandes incorrects en provenance des mauvaises prédictions de *hit/miss* de la mémoire cache L1 (p. ex. les instructions dépendantes du chargement sont émises après avoir prédit un *hit* L1, mais finalement le chargement entraîne un *miss* L1). Dans le cas où l'exécution d'une instruction avec un opérande incorrect est détectée, l'instruction est interrompue puis rejouée. Les auteurs ont fourni une évaluation détaillée des performances de différents schémas de rejeu et ils ont montré que ce schéma réduit la complexité de la logique d'ordonnancement.

Le rejeu d'instructions matériel est également utilisé pour des motifs de tolérance aux fautes où le composant de rejeu détecte et corrige des pannes. Les travaux [9] proposent une implémentation qui maintient la capacité de détecter des erreurs de synchronisation et d'éliminer la variation de la fréquence causée par les variations dynamiques de la tension d'alimentation et de la température. Le mécanisme de tolérance rejoue les instructions défaillantes à une fréquence d'horloge inférieure à la fréquence maximale afin de garantir un fonctionnement correct.

La technique de rejeu est aussi exploitée en tant que contre-mesure aux attaques par injection de fautes. Les travaux [47] implémentent un mécanisme matériel qui rejoue et ordonnance les instructions lors de leur exécution en exploitant ainsi les emplacements libres sous contraintes du nombre et du type des unités de traitement. Les premiers résultats de l'évaluation pour un processeur VLIW à 8 unités prouvent que la méthode proposée est capable de supporter jusqu'à 5 fautes simultanées avec une pénalité de performance significative, bien que logique, allant jusqu'à 350%.

Même si le rejeu d'instructions est connu dans l'état de l'art dans certains contextes, y compris en cybersécurité, nous ne connaissons pas de travail de l'état de l'art qui présente un support matériel de rejeu dans un processeur tel que celui présenté dans la suite de ce document de thèse.

1.3. Résumé des travaux

Ces travaux ont été réalisés dans le cadre d'un financement de la DGA et de la Région Bretagne que nous remercions pour leur soutien financier.

Comme nous l'avons mentionné dans les sections précédentes, la protection efficace d'un processeur contre les attaques par injection de fautes repose sur une certaine compréhension de son architecture. Ainsi, les contre-mesures matérielles sont généralement plus efficaces que les contre-mesures logicielles, cependant, elles engendrent une augmentation (légère ou importante) de la taille du circuit cible [35, 45].

L'idée de nos travaux est d'implémenter un support de rejeu matériel pour protéger le flux de données du processeur contre certaines FIA par un rejeu automatique de certaines instructions désignées par le programmeur dans le code assembleur. Nous implémentons une *nouvelle instruction* `rpl` dans le processeur, qui supporte plusieurs modes de rejeu, notamment le rejeu simple, le rejeu avec détection, le rejeu avec tolérance aux fautes et le rejeu aléatoire. L'instruction `rpl` automatise le rejeu d'instructions dans le processeur lors de son exécution. Le principe est de rejouer les instructions sans avoir besoin ni de les dupliquer dans le code assembleur ni d'ajouter les instructions de comparaison ou de vote majoritaire comme présenté dans l'état de l'art [43, 6]. En effet, afin de rejouer `n` fois en série toutes les instructions d'une séquence d'instructions de taille `w`, appelée *fenêtre de rejeu*, nous ajoutons dans le code assembleur l'instruction `rpl` immédiatement avant la fenêtre à protéger en indiquant la valeur des paramètres `w` et `n` à prendre en compte pour la fenêtre de rejeu courante. En outre, des éléments matériels de correction et de détection sont ajoutés au processeur pour compléter le dispositif.

Notre support de rejeu avec différents éléments de détection de fautes ou avec différents éléments de tolérances aux fautes permettent d'avoir un gain significatif en termes de temps d'exécution (il diminue le nombre d'instructions exécutées) et de taille de l'application logicielle par rapport à leurs équivalents logiciels ID et IT présentés dans l'état de l'art [43, 6].

Nous structurons notre travail en deux contributions principales. La première comprend une implémentation des solutions de *rejeu avec comparaison* (RC) pour un nombre de rejeu `n`, fixé par l'utilisateur. La deuxième contribution consiste à étendre le support du rejeu avec des mécanismes de randomisation matérielle et de triplication avec vote majoritaire en matériel. Dans la suite de cette section, nous présentons d'abord l'architecture de base du processeur utilisé, ensuite nous fournissons un résumé des contributions que nous détaillerons dans les chapitres 2 et 3.

1.3.1. Présentation du processeur cible

Nous avons développé un processeur élémentaire qui permet d'étudier des protections matérielles et logicielles contre des attaques par injection de fautes. Les objectifs d'implémentation de ce processeur sont d'abord : déterminer l'impact d'injections de fautes sur l'architecture ; implémenter des protections matérielles et logicielles contre des attaques physiques et les comparer.

Afin de concentrer nos efforts, mieux identifier les sources de vulnérabilité et proposer des

protections globalement plus robustes, nous avons opté pour l'implémentation d'un processeur avec un faible niveau d'optimisation et un fonctionnement interne simple. De plus, le cadre applicatif de notre processeur est celui d'applications embarquées à faible coût, ce qui justifie aussi l'utilisation d'un processeur très simple.

Par ailleurs, plusieurs versions de ce processeur sont considérées lors de sa réalisation matérielle, afin d'évaluer différents compromis entre performances, robustesse aux attaques et coût silicium et mémoire.

Bien évidemment, les connaissances acquises par l'étude des versions très simples de ce processeur ne sont pas directement applicables à des processeurs plus complexes en terme d'optimisation et d'architecture utilisée. Mais nous pensons que l'étude de la sécurité dans notre processeur constitue une étude préliminaire et nécessaire dans un cadre maîtrisé.

Dans l'avenir, nous souhaitons étudier l'impact de certaines optimisations utilisées dans des processeurs plus complexes et porter nos solutions sur des cœurs RISC-V.

Description de l'architecture de notre processeur

Le processeur cible est un petit processeur d'architecture RISC de 32 bits. Il est proche du cœur RISC-V RV32IM, mais avec un système de branchement légèrement différent avec moins de fonctionnalités, moins d'optimisations et avec une exécution ordonnée. La figure 1.9 illustre une représentation de base de l'architecture du processeur (les signaux et les éléments du processeur ne sont pas tous représentés). Le pipeline de processeur comporte deux étages : 1) chargement ; 2) décodage et exécution.

Le processeur se compose de :

— un banc de registres (*Register file* (RF)) de 32 registres généraux de 32 bits chacun avec deux ports de lecture et un port d'écriture accessibles à chaque instruction. Le registre `R0` est fixé à la valeur 0 ;
— deux unités d'exécution pour les nombres entiers de 32 bits : une unité arithmétique et logique (*Arithmetic Logic Unit* (ALU)) et un multiplieur ;
— une unité de chargement/stockage (*Load–Store Unit* (LSU)) qui accède à la mémoire de données (`D-MEM`) ;
— de données en mémoire ou registres de 32 bits (pas d'accès par octet ou autres) ;
— des compteurs de performance pour les cycles, les instructions, les branchements et l'accès à la mémoire de données ;
— des mémoires de données et de code séparées et adressées par mot (et pas par octet) ce qui implique qu'on doit incrémenter de 1 le compteur de programme (*Program Counter* (PC)) afin de passer à l'instruction suivante en dehors des sauts (et pas +4 comme dans d'autres systèmes).

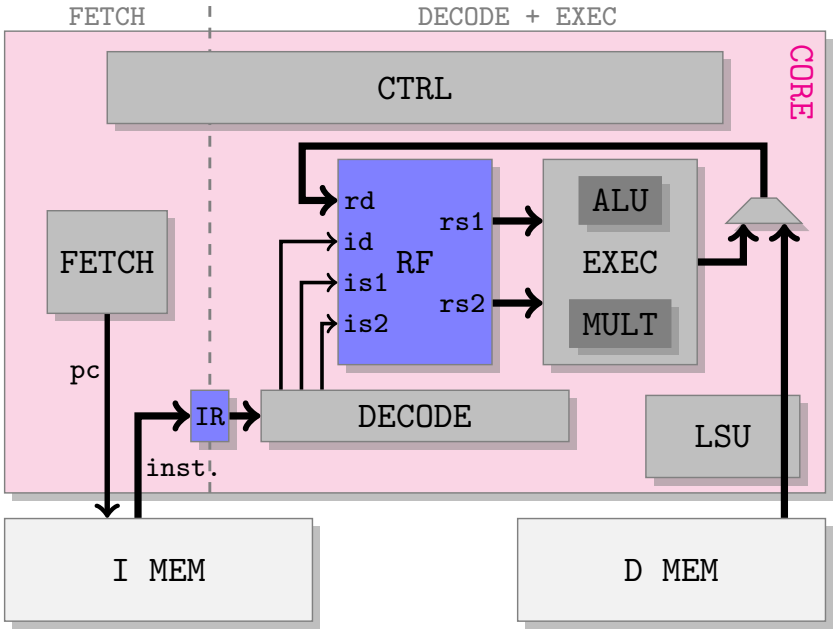


FIGURE 1.9. – Architecture du processeur cible.

Format des instructions

Une instruction comporte systématiquement les 5 champs A, B, C, D et E définis dans la figure 1.10. Elle contient les éléments suivants :

- le code opération, noté *opcode*, sur 8 bits ;
- un éventuel opérande immédiat (`imm`) de 14 bits qui est la concaténation des champs C et B utilisés conjointement ;
- de 1 à 3 indices de registres (`rid`) de 5 bits, désignés plus spécifiquement :
  - `rs` : l'unique registre source de l'instruction ;
  - `rs1` et `rs2` : les deux registres sources de l'instruction ;
  - `rd` : le registre destination de l'instruction ;
  - `r@` : le registre contenant une adresse pour la `LSU` (seuls les 14 bits de poids faibles sont considérés étant donnée que la taille des mémoires (`D-MEM` et `I-MEM`) est 2^{14} mots).

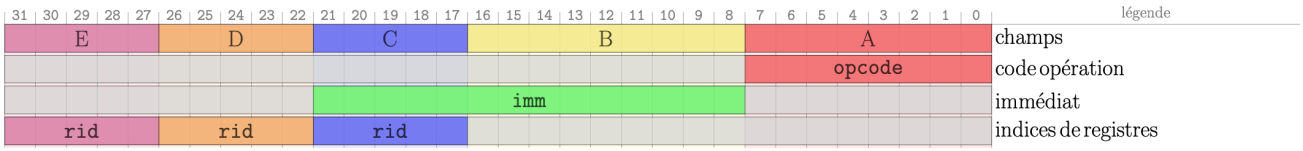


FIGURE 1.10. – Format des instructions : champs et utilisations.

Catégories d'instructions

Le processeur comporte les catégories d'instructions illustrées dans la figure 1.11, qui sont :

— opérations `ALU` ;
— lectures et écritures en mémoire de données désignées par `ld` et `st` ;
— sauts inconditionnels désignés par `jmp` ;
— branchements conditionnels désignés par `br` ;
— des protections internes désignées par `prot` (nous détaillerons cette catégorie dans les chapitres suivants).

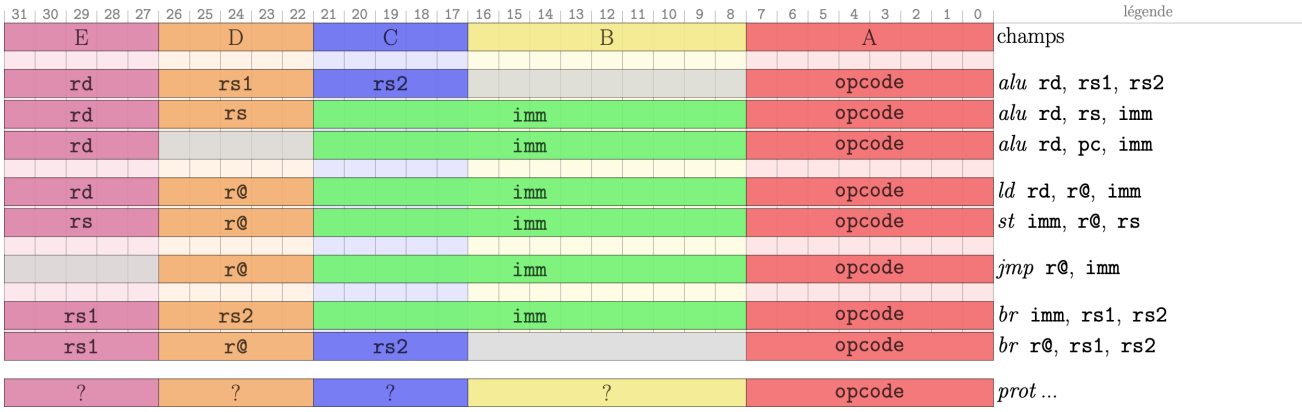


FIGURE 1.11. – Catégories d'instructions de notre processeur.

1.3.2. Première contribution : Extension de processeur pour le rejeu matériel d'instructions contre FIA [3]

La première contribution de la thèse a été publiée dans la conférence DDECS 2022. Elle consiste à implanter une contre-mesure matérielle par une extension du jeu d'instruction du processeur afin de lutter contre certaines FIA.

L'extension implémentée permet de rejouer en série des instructions désignées (dans des *fenêtres de rejeu*) sans devoir les dupliquer dans le code assembleur. L'idée est d'ajouter une instruction `rpl` à l'ISA du processeur, en implémentant un support matériel de rejeu à son architecture. L'instruction `rpl` prend 2 arguments `n` et `w` et spécifie que chaque instruction de la *fenêtre* des `w` instructions qui suivent immédiatement `rpl` sera exécutée une fois et *rejouée* `n` fois pour un total de 1+ `n` exécutions consécutives.

Les valeurs possibles de la fenêtre de rejeu `w` et du nombre de rejeux `n` sont fixées dans le code assembleur par l'utilisateur. En revanche, leurs tailles maximales sont limitées par des paramètres matériels, appelés respectivement Ω et Θ dans notre code HDL tels que $1 \leq \mathtt{w} \leq 2^{\Omega}$ et $1 \leq \mathtt{n} \leq 2^{\Theta}$. Les valeurs possibles de Ω et Θ appartiennent respectivement aux ensembles $\{1, 2, 3, 4\}$ et $\{1, 2\}$ dans nos implantations.

La figure 2.3 illustre le format de l'instruction `rpl` (elle fait partie de la catégorie `prot`). L'instruction comporte 3 champs qui contiennent :

— l'*opcode* sur 8 bits ;
— la valeur de `w` codée sur au plus Ω bits ;
— la valeur de `n` codée sur au plus Θ bits.

Rejouer l'instruction `n` fois lors de son exécution peut corriger certaines fautes mais ce mécanisme n'est pas suffisant, de ce fait divers éléments de détection de fautes sont ajoutés au processeur pour rendre le support du rejeu plus résistant aux FIA. À chaque cycle de rejeu, des éléments de détection comparent les données actuelles avec les valeurs originales stockées lors de la première exécution de l'instruction rejouée afin de détecter immédiatement les fautes et les signaler à la politique de sécurité.

L'idée de cette protection réside dans la manière dont la détection de fautes est implémentée dans le processeur. Nous implantons en effet des mécanismes dédiés à la comparaison des données rejouées de telle sorte que la détection de fautes se réalise en parallèle du rejeu et automatiquement sans intervention de l'utilisateur. Autrement dit, à chaque cycle de rejeu, un élément de détection matériel dédié effectue une comparaison entre la valeur courante et la valeur originale mémorisée lors de la première exécution de l'instruction. Le résultat de la comparaison alimente le contrôle du processeur pour signaler des valeurs différentes pendant les cycles de rejeu. Dans le cas d'une détection d'erreur lors de la comparaison, un signal d'interruption spécifique à l'erreur détectée est levé. Par conséquent, nous n'insérons pas d'instructions de branchement supplémentaires dans le code (le mécanisme logiciel de détection de faute) pour effectuer la comparaison et lever une exception comme présenté dans l'état d'art en section 1.2.3. Ainsi, notre mécanisme de détection consomme moins de ressources CPU que les protections logicielles présentées dans l'état de l'art (p. ex. moins de pression sur le banc de registres, pas d'instructions pour les comparaisons des résultats rejoués) [6, 43] mais nécessite quelques éléments matériels en plus.

L'implémentation du rejeu matériel nécessite quelques modifications dans le processeur, notamment dans le contrôle et le *fetch* pour gérer le support du rejeu et le décodage de l'instruction `rpl`. Dans le chapitre 2, nous examinons en détail les modifications requises dans le contrôle du processeur afin d'implémenter le support de rejeu. Puis, nous exposons le choix d'emplacement de chaque élément de détection.

Ce rejeu est appelé rejeu avec comparaison (*Replay with Comparison* (RC)), où le nombre de rejeux est fixé par l'utilisateur dans le code assembleur (c'est le paramètre `n`).

Nous concevons également une version avec un support de *refetch*. Durant les `n` exécutions du rejeu, l'instruction est rechargée depuis `I-MEM` afin de rafraîchir le registre d'instruction pendant chaque exécution. L'instruction courante est comparée à l'instruction originale (exécutée pour la première fois) à travers un élément de détection matériel dédié supplémentaire. Le support de *refetch* détecte plus de fautes dans l'interface entre le cœur du processeur et `I-MEM` pendant le rejeu, mais il consomme plus d'énergie. C'est pourquoi nous implémentons deux versions possibles du processeur : avec et sans support *refetch*. C'est donc à l'utilisateur de choisir la

version adaptée à ses contraintes.

Afin d'étudier l'impact de tous les paramètres de protection, nous avons implémenté plusieurs versions du processeur en faisant varier des paramètres de protection :

— la taille maximale de `w` (Ω) qui varie entre 1 et 4 bits ;
— la taille maximale de `n` (Θ) qui varie entre 1 et 2 bits ;
— différentes combinaisons des éléments de détection avec et sans support de *refetch*.

Les versions protégées de processeur sont implémentées dans un FPGA Zynq 7020 en utilisant Vivado 2018.3 de Xilinx. Après une validation fonctionnelle en simulation, nous avons comparé ces versions en termes de coût et performance à savoir la surface du processeur dans le FPGA et la fréquence maximale. Nous avons aussi évalué notre protection de rejeu matériel et la duplication logicielle d'instructions (présentée dans l'état d'art) en termes de temps d'exécution, de taille de code dans la mémoire `I-MEM` et de couverture de fautes. Nous avons obtenu une réduction significative du temps d'exécution et de la taille de l'application par rapport à une protection logicielle avec des réductions respectives de $\times 3 \rightarrow \times 2$ et $\times 2 \rightarrow \times 1.3$.

Notre nouvelle instruction conduit à des améliorations significatives par rapport aux protections logicielles pour un faible coût de silicium (avec une réduction très légère au niveau de la fréquence maximale). Nous constatons que lorsque la valeur de Ω augmente de 1 à 4 bits ou Θ de 1 à 2 bits, la surface du FPGA n'augmente que très légèrement et l'atténuation de la fréquence reste très faible.

Pour pouvoir évaluer le support de rejeu d'instructions contre les injections de fautes logiques et pour maîtriser le comportement du processeur face à ces injections, nous avons opté pour une méthodologie de simulation d'injections de fautes logiques dans le code RTL. À ce niveau d'abstraction, la simulation de fautes permet un meilleur suivi et une meilleure observabilité. C'est-à-dire que nous pouvons injecter précisément des fautes dans la structure analysée (PC, RF, mémoire, etc.) et observer leur propagation avec une grande précision. Dans l'avenir, nous aimerions pouvoir réaliser des attaques physiques sur un prototype FPGA complet.

Dans le chapitre 2, nous décrivons aussi l'environnement d'évaluation, notamment les codes de caractérisation (les applications logicielles), les versions du processeur évaluées et la méthode de simulation et d'analyse de fautes. Puis, nous rapportons les résultats post-synthèse et après placement et routage de la surface (*LUT*, *flip-flop*) et les performances à la fréquence d'horloge la plus rapide obtenue pour chaque version du processeur. Enfin, nous présentons les résultats du temps d'exécution, de la taille de chaque code de caractérisation et de simulations de fautes logiques.

1.3.3. Seconde contribution : nouvelles protections par rejeu

La seconde contribution est une extension de l'article [3] que nous espérons soumettre à un journal dans les prochains mois. Elle consiste à étendre le support du rejeu RC (présenté dans le chapitre 2) en introduisant des mécanismes de randomisation matérielle et de triplication avec vote majoritaire matériel. Nous proposons donc de modifier et d'étendre l'instruction `rpl` en

ajoutant un *nouvel argument* nommé `mode`.

Le chapitre 3 propose des mécanismes de sécurité supplémentaires dans le processeur, basés sur deux stratégies de rejeu : *rejeu avec comparaison* (RC) et *triplication avec vote majoritaire* (TV).

La première stratégie consiste à implémenter d'abord des versions du processeur qui supportent uniquement un nombre de rejeux `n` fixé dans le code assembleur (comme présenté dans chapitre 2). Ensuite, nous implémentons des versions du processeur qui supportent uniquement un nombre de rejeux aléatoire (*Random Replay with Comparison* (RRC)).

Concernant le RRC, le nombre de rejeux n'est plus fixé comme dans RC, mais il est généré par un générateur de nombres pseudo-aléatoire matériel (*Pseudo Random Number Generator* PRNG) (p. ex. Trivium [25, 12]) lui-même alimenté par un générateur TRNG. Ce mode de rejeu empêche l'attaquant de prédire le nombre de rejeux par fenêtre et donc l'empêche de prévoir le bon moment pour injecter des fautes. Le rejeu aléatoire permet de rejouer la fenêtre de `w` instructions qui suivent immédiatement `rpl` sans devoir fixer le nombre de rejeux `n` par l'utilisateur dans le code assembleur ni dans le code HDL. Le nombre de rejeux prend deux formes : aléatoire et *uniforme* pour la fenêtre de rejeu (RRCU), ce qui signifie que toutes les instructions de la fenêtre ont le même nombre aléatoire de rejeux `n` ; ou bien il est aléatoire et *variable* pour chaque instruction de la fenêtre (RRCV), ce qui signifie que les instructions protégées ont un nombre aléatoire de rejeux `n` différent les unes des autres au sein de la fenêtre.

La second stratégie consiste à implémenter des versions du processeur qui supportent uniquement la triplication avec vote majoritaire. Le nombre de rejeux est fixé dans le code HDL et égal à 2. Lors du dernier rejeu de l'instruction rejouée, le vote majoritaire se fait en se basant sur les données des 3 exécutions. Pour corriger les fautes présentes dans le flux de données, nous avons implanté les éléments de vote majoritaire à certains endroits spécifiques du processeur, à savoir le banc de registres RF et l'unité mémoire LSU.

Nous implémentons également une version complète qui supporte tous les modes de rejeu : RC fixe, les deux modes RC aléatoire (RRCU et RRCV) et le mode TV, afin d'offrir une certaine flexibilité. L'utilisateur peut choisir le mode approprié pour chaque fenêtre de rejeu dans l'application logicielle. L'étude des meilleures combinaisons des modes de protections constitue une perspective pour l'avenir.

L'instruction `rpl` a 3 paramètres précisés à l'exécution comme suit : `rpl mode w [n]` où `n` est obligatoire uniquement pour le mode RC. La figure 3.1 illustre son nouveau décodage. Elle comporte donc 4 champs qui contiennent respectivement :

— l'*opcode* sur 8 bits ;
— la valeur de `w` codée sur au plus Ω bits ;
— la valeur de `n` codée sur au plus Θ bits (obligatoire uniquement pour le cas de RC) ;
— le mode de rejeu, noté `mode` et codé sur 5 bits.

Afin de protéger les instructions du processeur en augmentant le nombre de fautes détectées (p. ex. des erreurs de décodage) et en fournissant une forte probabilité de se retrouver avec une

instruction invalide en cas d'injection de fautes, nous avons proposé une méthode spécifique pour coder le champ `opcode` des instructions du processeur et le champ `mode` de `rpl`. Le système de codage interdit d'utiliser les mots composés uniquement de zéros ou de uns pour les opcodes, afin de détecter automatiquement les fautes de type *bits-set* ou *bits-reset*. Afin de garantir la détection d'un seul *bit-flip*, le système de codage permet également de choisir la combinaison (un ensemble des mots) ayant la plus grande distance de Hamming entre ces éléments. En effet, nous avons codé la partie `mode` sur 5 bits alors que nous n'avons besoin que de 8 modes en choisissant une combinaison de 8 mots de 5 bits selon nos critères de robustesse. Le chapitre 3 présente cette méthode en détail.

Dans le chapitre 3, nous détaillons les modifications requises afin de mettre en place les trois types de rejeu. Ensuite, nous présentons le choix des emplacements des différentes comparaisons (les éléments de détection pour RC et RRC) et ceux de triplication avec vote majoritaire (les éléments de vote pour TV).

Nous avons implémenté plusieurs versions du processeur avec différentes valeurs de Ω et Θ et différentes combinaisons des éléments de détection ou de tolérance aux fautes (avec et sans support *refetch*), pour étudier l'impact de tous les paramètres de l'instruction `rpl` et de son matériel.

Après avoir validé le fonctionnement des versions du processeur en simulation, nous avons comparé les versions protégées du processeur et nous avons étudié l'impact de chaque élément de protection en terme de coût notamment la surface du processeur dans le FPGA et la fréquence maximale. Nous avons aussi évalué la duplication d'instructions avec comparaison logicielle (voir section 1.2.3), la triplication d'instructions avec vote majoritaire logicielle (voir section 1.2.3) et nos protections matérielles (RC, RRC, TV) en termes de temps d'exécution, de taille de code et de couverture de fautes. Comparativement à la redondance temporelle logicielle (ID et IT), nous avons énormément gagné en termes de temps d'exécution et de taille de l'application protégée.

2. Contribution 1 : Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks

This chapter presents the first contribution of this thesis. It consists of protecting processors against some fault injection attacks by implementing hardware support for instruction replay (fixed replay). A replay instruction is added to the instruction set of a small 32-bit RISC processor to allow the automatic and parametrized replay of sequences of instructions. Various detection elements are added to the processor and implemented on FPGA. We evaluated several protected versions of the processor by varying the parameters of the fixed replay support, the detection elements and the refetch support in terms of processor area costs and the impact on maximum frequency. We also evaluated and compared an ultimate version, which we consider in this chapter as a complete version, integrating all detection elements and the refetch support in terms of performances, cost and fault coverage. The proposed extension leads to significant improvements compared to software protections for a small silicon overhead.

This chapter is based on the paper published at the DDECS 2022 conference. The original paper is available on the following websites : HAL and DDECS

Chapter 3 extends this contribution and adds protection to the instruction replay support, including randomization and fault tolerance.

2.1. Introduction

Embedded processors can be subject to *physical attacks* due to some proximity between an attacker and the circuit. *Side channel attacks* [41] exploit correlations between observable parameters (e.g., computation time, power consumption, electromagnetic radiation) and secret data. *Fault injection attacks* (FIAs) [5, 60] exploit perturbations (e.g., glitch on power supply or clock, electromagnetic radiation, laser shot) in the circuit to reveal secret data [7] or bypass security features [54]. Below, we deal with hardware protections against FIA built in embedded processors.

Various protection methods exist against FIA (see [5] for an introduction) : error code correction/detection, algebraic/functional property check, information and temporal redundancy, randomization, etc. *Redundancy*-based solutions are popular in hardware and software implementations.

In software (SW), *instruction duplication* and *triplication* [6, 52] are easy to use to secure critical (parts of) codes but lead to important overheads in execution time and code size. Moreover, software protections rarely take into account hardware implementation details, such as the processor pipeline, and may not be as effective as intended. Sec. 2.2 quickly reviews some previous works on this topic.

Hardware (HW) support for *instruction replay* in processors is well-known to resolve speculation issues in high-performance processors [29], improve fault-tolerance [9] and security [47]. In this chapter, we propose *processor extensions* for *hardware instruction replay* in Sec. 2.4. One dedicated *replay instruction* is added to the instruction set of a small RISC processor described in Sec. 2.3. Several *protection elements* are also added to the processor, see Sec. 2.4, and implemented on FPGA, see Sec. 2.7.

We evaluate various protected versions of our processor and compare them to typical software protections in terms of cost, speed and robustness against FIA using logical simulations on typical critical benchmarks in Sec. 2.7.

In this chapter, we only protect the *data flow* but not the control flow (see [56, 13]). Our team works on this topic (see [55] for a recent PhD) and we plan to add protections of the control flow in the future.

2.2. State of the Art

Faults are identified as a *reliability* issue in processors since the 70s [38]. FIAs also lead to *cybersecurity* issues [5, 60]. Protections based on *redundant computation or information* are available in hardware and software. In hardware for instance, double data rate computation [40] allows high coverage but requires larger circuits with a lower frequency. In software, *temporal redundancy* is popular with methods from manual instruction duplication and triplication [6, 52] to compiler-assisted solutions [46].

Instruction duplication with comparison and instruction triplication with voting from [6] lead to important overheads in terms of execution time (resp. 2x to 4x) and code size (resp. 4x to 14x per protected instruction). These overheads are not the only concerns with such protections. [6] assumes that a single fault only impacts a single instruction during the execution, but this is not the case. [59] and [58] show that in pipelined processors, a single fault impacts several pipeline stages and then several instructions (not only one instruction). These software protections can be bypassed even with a single fault [42, 36, 37].

2.3. Target Processor

Our processor is a small homemade 32-bit RISC designed, in System Verilog, for exploring hardware protections at architecture level. It is close to a RV32IM RISC-V core [44] with a slightly different branching system and less features. In future, we plan to implement our protections in RISC-V. Fig. 2.1 presents a simplified schematic of the processor architecture. The

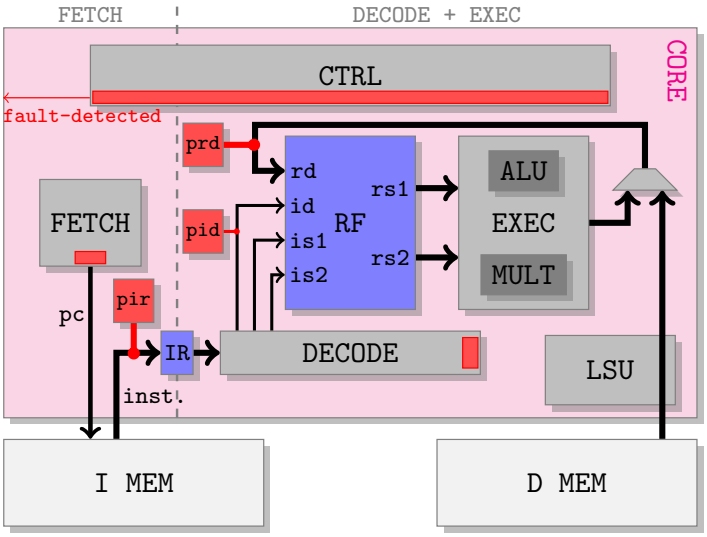


FIGURE 2.1. – Simplified schematic of the processor architecture (not all signals and elements are represented). Red elements are for replay protection.

pipeline has 2 stages : fetch | decode + execute. The register file (RF) has 32 registers of 32 bits each with 2 read ports and 1 write port. Register `R0` is fixed at the value 0. The processor contains 2 execution units for 32-bit integers : one arithmetic and logic unit (ALU) and one multiplier. The load/store unit (LSU) accesses the data-memory hierarchy (`D-MEM`) without cache for this chapter. The instruction memory `I-MEM` is also without cache. The processor also includes hardware performance counters for cycles, instructions, memory accesses, branches, and protections monitoring. It is fully implemented and validated on FPGA (see Sec. 2.6).

We also develop an assembler to generate binaries and scripts for cycle accurate and bit accurate HDL simulation for functional validation and fault injection evaluation.

2.4. Proposed Hardware Replay Protection

Our protection consists in a *replay instruction* added to the instruction set. Various *detection elements* added to the core and *modifications* in the processor control are detailed in this section. See Sec. 2.7 for implementation results and performance evaluations.

2.4.1. Replay Instruction

The replay instruction, named `rpl`, is intended to protect small critical sequences of instructions, called *replay windows*. It takes 2 arguments, `rpl n w`, and specifies that each instruction in the window of the `w` instructions immediately following `rpl` will be *executed once and replayed* `n` times for a total of `n`+1 consecutive executions.

This behavior is illustrated in the right side of Fig. 2.2 with a simple 4-instruction example : `I1`, `I2`, `I3`, `I4`. Instructions `I1`, `I2`, `I3` must be protected and executed twice while `I4` is kept

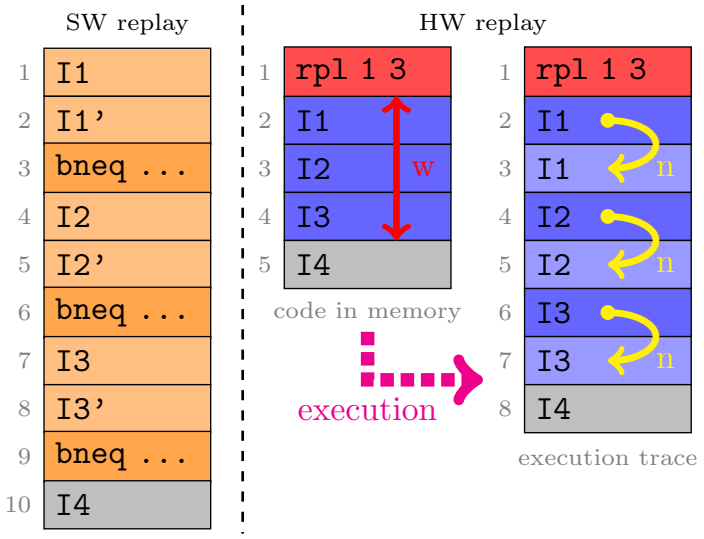


FIGURE 2.2. – Illustration of a typical software protection and our hardware replay protection in a small 4-instruction code ('I1-4' where 'I1-3' are protected using one replay and 'I4' is not).

unprotected. The window of instructions to be protected has a width `w=3` and the number of replay(s) is `n=1`. Thus, a replay instruction `rpl 1 3` is added just before the instructions window `I1-3`.

Left side of Fig. 2.2 presents the same example in case of a pure software replay protection for comparison.

Fig. 2.3 describes the 3 fields of the `rpl` instruction coding :

— 8-bit *opcode* ;

— value of `w` coded on a maximum of Ω bits (Ω defined later) ;

— value of `n` coded on a maximum of Θ bits (Θ defined later) ;

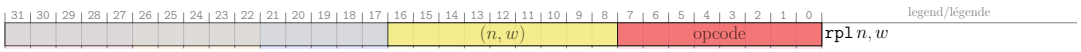


FIGURE 2.3. – The `rpl` instruction format.

2.4.2. Detection Elements

Just replaying instructions may not be sufficient. The result of each replayed instruction has to be *compared* to the original one. For this, we add *detection elements* (DE) depicted in Fig. 2.4. At each replay cycle, a DE compares the current `value` with the original one stored at the first execution of the instruction. The `enable` signal on the register is managed by the processor control. The comparison result `diff` feeds the processor control to indicate different values during replay cycles. In such a case, a specific `fault-detected` interruption signal is raised. The security policy related to this interruption is out of the scope of this thesis.

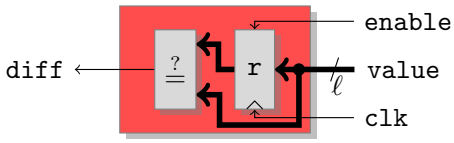


FIGURE 2.4. – Detection element (used for `prd`, `pid`, `pir` in Fig. 2.1).

To protect the data flow in the processor, we protect the result from the execution units and LSU. The 32-bit DE called `prd` in Fig. 2.1 protects the destination register (`rd`) written in RF. The index `id` of the destination register is protected by the 5-bit DE called `pid`. The `pir` DE protects the instruction register (see Sec. 2.5 for details).

Compared to software replay, a cheap `prd` DE (one register and comparator) avoids the use of additional registers in the register file and explicit detection instructions in the code. In software replay Fig. 2.2 (left side), instruction `I1` (line 1) is replayed by instruction `I1'` (line 2) with a different destination register (but similar opcode and sources), and the comparison in line 3 compares the two result registers and branches to the software error handler if they are different. Duplication with comparison in software leads to 3 executed instructions for each protected instruction. This can be 4 in processors where comparison and conditional branch are separated instructions. Our hardware replay leads to smaller codes in memory and fewer executed instructions. The number of original and replayed instructions is the same as for software replay. But there is no comparison instruction(s) in the code but one `rpl` instruction for each replay window. The window width `w` in the `rpl n w` instruction leads to fewer instructions as soon as `w`>1 compared to software replay.

2.4.3. Processor Modifications for Replay Support

The processor control is modified to handle our hardware replay. A few internal flip-flops and logic gates are added to control the replay in the pipeline. When a `rpl` is decoded, the processor stores `n` and `w` into small local registers. During the execution of an instruction in a replay window, small counters control the number of replays and the position in the window. For each DE, the control must enable its register during the first execution of an instruction. During replay cycles, the control checks for `diff` signals to raise the `fault-detected` interruption.

The behavior of the processor is not modified at the ISA level. The addition of the `rpl` instruction does not change the behavior of the other instructions. This allows users to protect their codes with small additions on the assembly code. The automation of this protection for higher level codes (e.g., C) is out of the scope of this thesis.

2.5. Processor Versions

Several *versions* of the processor with different protections solutions are evaluated in Sec. 2.7. The maximal width of a replay window `w` is limited by a hardware parameter, called Ω, in our

HDL code such that $1 \leq \mathtt{w} \leq 2^{\Omega}$. We study the impact of the replay window width on cost and performance for $\Omega \in \{1, 2, 3, 4\}$ (see Sec. 2.7 for Ω determination).

Possible values for `n` are in $\{1, 2, 3, 4\}$ leading to 2-bit registers and counters in the control for the number of replays. We use 2-bit values for `n` since we target single fault protection. Protection against injection with multiple faults close in time and space will be a challenge for future systems to avoid a prohibitive cost due to numerous replays.

The `prd` and `pid` DEs are intended to protect the destination register from execution units and LSU. To complement this type of data flow protection, we also study replay support with a DE called `pir` in Fig. 2.1 to protect the *instruction register* (`IR`) fed from the instruction memory. It protects the indexes for source registers `is1`, `is2` as well as the index of the destination register. Protecting the 3 indexes `id`, `is1`, `is2` using individual DEs would cost $3 \times 5 = 15$ bits (about half of the 32 bits in `pir`). Adding a specific DE for immediate operands in the instruction would also add up to the protection bill. Then adding `pir` may be a good way to protect all parts of the instruction directly for a small overhead.

We also design a version with a *refetch* support. During the replay cycles, the current protected instruction is refetched from the instruction memory and compared to the original one. This is simple in a 2-stage pipeline, but this would be more costly for deeper ones. `pir` and refetch are different. Refetch leads to detect more faults in the interface between the core and `I-MEM` but costs more energy. We plan to study energy aspects of our protections in a future work.

2.6. Evaluation Environment

HDL codes are implemented in a Zynq 7020 FPGA using Vivado V2018.3 from Xilinx after functional validation in simulation. In the next sections, we report post-synthesis and place&route results for area (LUTs, flip-flops) and performance at the fastest obtained clock frequency for each version of the processor. We evaluate various versions of the processor and protection elements proposed in this chapter :

— `wop` : base processor *without any protection* (implementation results at line V=1 in Tab. 2.7.1) ;

— `wphw_x` : base processor *with protection* for different DE selections and parameters where `x` $\in \{2, 3, \cdots, 11\}$ (see lines V=`x` in Tab. 2.7.1 for details).

All the processor versions require the exact same number of BRAMs (16) and DSP blocks (4). Since they are only used in `I-MEM`, `D-MEM` and execution units, they are not impacted by the replay support and DEs.

To compare to pure software replay, we also evaluate a version called `wpsw` with the base processor `wop` and using *software protection* from state of the art [6, 52]. This is not a hardware version, the processor is the one in `wop`, only the code is changed.

To evaluate and compare these versions over various representative binary codes, we protect and execute the following functions commonly used in parts of secure applications :

— `verify_pin` to authenticate users (tested with 4 digits and a constant time code) ;

— `memcpy` to copy a memory region (critical for sensitive data, tested for various lengths) ;
— `atoi` to convert a string to an integer (critical when reading some identifiers, tested for various lengths) ;
— `trivium` a stream cipher used in some cryptographic applications (to reduce simulation time, we only focus on its `generate_bit` function which computes a new result bit and update the internal state).

We plan to use benchmarks such as FISSC [17] in the future.

For evaluating our protections, we use logical fault injection simulations on the HDL description (at cycle accurate and bit accurate level) of the processor. We inject faults for numerous *injection times*, *locations* and *types* as illustrated in the pseudo algorithm below :

```
list_codes = [verify_pin, memcpy, atoi, trivium]
list_versions = [wop, wpsw, wphw_1, wphw_2, ...]
list_fault_types = [stuck0, stuck1, bit_flip]

for code in list_codes:
  for version in list_versions:
    trace = get_trace(code, version)
    for inst in get_inst(trace):
      for reg in get_reg(inst, version):
        for type in list_fault_types:
           ftrace = insert_fault(trace, inst, reg, type)
           state = simulate(version, ftrace)
           save(state)
```

For each `code` and `version`, the `get_trace` function provides the execution `trace` corresponding to `code` adapted for the processor `version`. For a version with software protection, explicit replayed instructions (e.g. `I1'` for `I1` in Fig. 2.2 left side) and comparison instructions (e.g. `bneq`) are added. For a version with hardware protection, `rpl` instructions are added. When the `version` is unprotected, there is no addition to base `code`.

We use a large selection of executed instructions related to the data flow as *injection times* : all instructions before and after a loop in the function, and all instructions in the first, intermediate and last iterations of a loop (to save simulation time). The `get_inst` function provides the list of all instructions where a fault will be injected.

Regarding *injection locations* we use all registers of the base processors and all registers of our protection elements (DEs and replay control). This is the purpose of the `get_reg` function. We only inject faults in registers used by the `inst` instruction (IR, RD in RF, registers in our hardware protections). In this work, we only inject faults in registers since several physical faults lead to setup/hold time violation in flip-flops [42, 36, 37]. We plan to extend this with other types of faults in the future (logical simulation may not be sufficient).

To limit simulation time, we only use 3 types of *injection types* : stuck at 0 or 1, or one bit flip at a random position of the faulted register.

When a `code`, `version`, instruction `inst`, faulted register `reg` and `type` are selected, the `insert_fault` function generates the HDL script for the simulation of this fault in the complete trace. There is one simulation for each fault time (`inst` instruction in `trace`), location (`reg` register) and `type`. Then, the simulation is performed and its state (memory, all registers contents and flags) is saved for analysis.

For each version of the processor and code, a reference trace is also required for the execution of the base code without any fault injection. This reference trace is used to compare the simulation `state` of each simulation with a fault injection. This reference simulation is not represented in the pseudo algorithm to simplify the presentation.

The efficiency of a protection solution is evaluated using its *detection rate*. For the software protection, there is a detection when the detection code branches to the error handler instead of continuing the nominal execution. For our hardware protection, the detection signals are provided by the DEs.

The various loops iterations in the pseudo-algorithm above correspond to dozens of thousands of simulations over a few weeks of CPU time. Actual numbers of simulations are reported in gray in Tab. 2.2.

As this chapter targets the protection of the data flow, we do not attack the control flow of the pure software protection and applications. However, we inject faults in all registers of our protections : both "data" registers in DEs and control registers for replay management.

2.7. Evaluation of our Hardware Protections

Below, we report and analyze results for FPGA implementation, execution time, code size and fault injection simulations. We close the section with additional discussions.

2.7.1. FPGA Implementations Results

Table 2.7.1 reports FPGA results for all versions of the processor and various protection elements configurations (see Sec. 2.5 and Sec. 2.6 for details). Each line in the table is a version denoted Vα with $\alpha \in [1, 11]$. "Protection" columns indicate the configuration of the protection elements in the version. Gray values are relative differences $(x - ref)/ref$ w.r.t. reference values in V1 expressed in %. Version 1 is the *reference processor* without hardware replay or detection support.

Version 2 adds replay support but no detection for a very small impact : +10% area and -3% clock frequency.

Versions 3 to 5 add various combinations of detection elements to V2. Between V2 and V3, the 5 additional flip-flops are for the 5 bits in `pid`. Between V2 and V4, the 32 additional flip-flops are for the 32 bits in `prd`. V5 costs 37 (32 + 5) flip-flops more than V2 due to `pid` and `pir`. The area penalty is +20% for V5 but those DEs do not change much the clock frequency.

V	Protection						Area				Freq.	
	HW	Ω	pid	prd	ref	pir	LUT	%	FF	%	MHz	%
1	✘	✘	✘	✘	✘	✘	731	+00	301	+00	304	+00
2	✓	4	✘	✘	✘	✘	756	+03	324	+08	294	-03
3	✓	4	✓	✘	✘	✘	759	+04	329	+09	301	-01
4	✓	4	✘	✓	✘	✘	799	+09	356	+18	297	-02
5	✓	4	✓	✓	✘	✘	801	+10	361	+20	294	-03
6	✓	3	✓	✓	✘	✘	799	+09	359	+19	300	-01
7	✓	2	✓	✓	✘	✘	797	+09	357	+19	303	-00
8	✓	1	✓	✓	✘	✘	797	+09	355	+18	305	+00
9	✓	4	✘	✘	✓	✘	757	+04	325	+08	295	-03
10	✓	4	✘	✓	✓	✘	797	+09	357	+19	287	-06
11	✓	4	✘	✓	✓	✓	809	+11	389	+29	283	-07

TABLE 2.1. – FPGA (Zynq 7020) implementation results for various versions (V) of the processor and protection elements.

Versions 5 to 8 only differ in the maximal window width supported ($\mathtt{w} \leq 2^{\Omega}$). Its impact is negligible for $1 \leq \Omega \leq 4$.

Versions 9 to 11 add refetch support (column "ref"). V10 and V11 add `prd` and `pir` support to V9 for a more complete protection. There is no `pid` in these versions since `id` is already in the refetched instruction (and in `pir` in V11). The impact is still limited : +30% area and -7% clock frequency.

In the following, we only use V11, the most advanced version of the processor, for the evaluations (performance, code size, fault injection simulations) due to limited benefit of the smaller versions.

Adding hardware support for replay and detection support does not cost much. The +30% area overhead in a very simple core should be even smaller in more complex ones. In cores with a deeper pipeline, hardware replay will require more internal registers and slightly more complex control than in our 2-stage one. But we think that the overall penalty should be still limited. We still have to evaluate if some internal resources can be shared between deeper pipeline elements and hardware replay support (and DEs).

2.7.2. Timing Performance and Code Size Results

Figure 2.5 presents the performance results in terms of execution time and code size for each evaluated code (`verifyPin`, `memcpy`, `atoi` and `trivium`) used in 3 cases :

— reference : unprotected code on the base processor V1 ;

— SW protection : protected code with software replay from state of the art [6, 52] on the base processor V1 ;

— HW protection : protected code with our hardware replay on the processor V11.

For both SW and HW protections, protected instructions are replayed once (`n=1` for our `rpl` instructions).

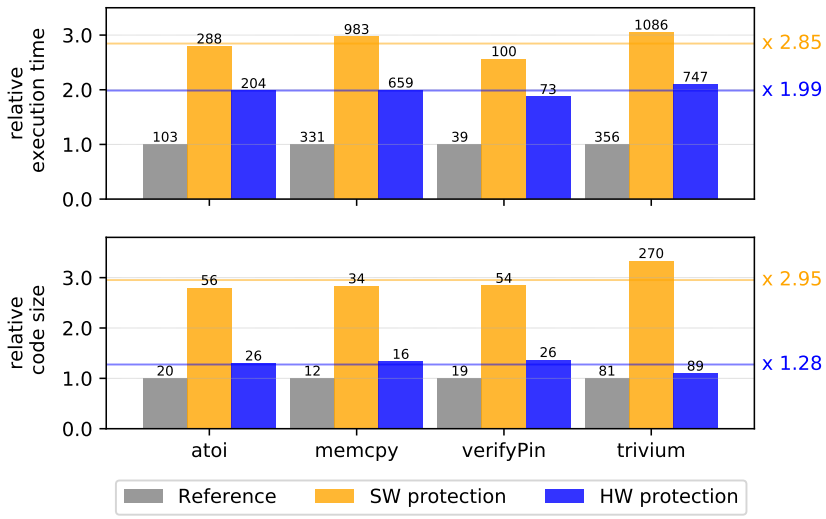


FIGURE 2.5. – Execution time and code size results. Values above the columns are : the number of cycles (top) ; and the number of instructions (bottom). Values on the right are averages.

The benefit of a hardware replay in comparison to a software one is significant as illustrated in Fig. 2.2. The execution time is reduced by a factor of almost 3 with SW replay to 2 with HW replay. It will be difficult to decrease the execution time below 2 for n=1 (more generally below n+1) since the processor must execute `n+1` times each protected instruction (unless using parallelism [47, 38]). The code size is also significantly reduced from a factor 3 with SW replay to only 1.3 with HW replay. This may constitute a potential interest in embedded systems.

2.7.3. Fault Injection Simulations Results

Using real fault injections would be interesting, but this is not simple in a FPGA implementation of softcore processors since one does not really know how close are the fault effects in the FPGA compared to a hardcore processor in an ASIC. Therfore, we use logical fault injection simulations at HDL level as explained in Sec. 2.6. Tab. 2.2 reports the corresponding results for 2 cases :

— SW protection : protected code with software replay from state of the art [6, 52] on the base processor V1 ;

— HW protection : protected code with our hardware replay on the processor V11.

The hardware replay leads to similar fault detection results than the software one (about 75% of faults detected). But this protection level is achieved with much faster execution times and smaller codes for a moderate hardware overhead. This shows that hardware replay can be an interesting solution for protecting the data flow.

codes	protection solutions			
	software		hardware	
`verifyPin`	0.757	(969)	0.760	(3036)
`memcpy`	0.743	(1341)	0.796	(3954)
`atoi`	0.764	(1317)	0.790	(6897)
`trivium`	0.737	(3222)	0.772	(9195)
mean	0.750		0.779	
std. dev.	0.011		0.014	

TABLE 2.2. – Fault injection simulation results. The decimal values are ratios of number of executions where the protection solution (SW or HW) detects a fault over the total number of executions (integers in gray) for each code.

One should keep in mind that this fault injection simulation campaign is in favor of the software protection since we do not inject faults in the control flow (e.g. all `bneq` instructions in Fig. 2.2) while we inject faults in all the control and detection registers of our hardware replay. We will complete those simulations when working on the control flow protection.

2.7.4. Additional Discussions

To help determine a good Ω value for the maximal width of replay windows ($\mathtt{w} \leq 2^{\Omega}$), Tab. 2.3 reports the distribution of `w` values in protected source codes and their corresponding execution traces. The count of the number of times an actual window of width `w` occurs is reported as $\mathtt{w}^{(\text{count})}$. Depending on the actual input data of the code, replay windows in the source code can be executed different numbers of times or not executed at all in the execution trace due to conditional jumps and loops. The "in source" column represents the effort provided by the user to protect the code. The "in trace" column shows what are the values `w` used during the actual executions (for given input data). Reported values in Tab. 2.3 ("in trace" column) are typical results. In most cases, only small `w` values are used. But in `trivium` example, there is a long 54-instruction sequence to be protected before the loop (4 `rpl` are required with $\Omega = 4$, since $54 = 3 \times 2^4 + 6$) and a 17-instruction one after the loop. As those sequences are only used outside

codes	replay windows distributions $\mathtt{w}^{\text{count}}$	
	in source	in trace
`atoi`	$1^{(1)}, 2^{(1)}, 3^{(3)}, 4^{(1)}$	$1^{(1)}, 3^{(10)}, 4^{(8)}$
`memcpy`	$2^{(1)}, 3^{(1)}, 4^{(1)}$	$2^{(1)}, 3^{(1)}, 4^{(8)}$
`verifPin`	$1^{(4)}, 3^{(1)}, 4^{(2)}$	$1^{(4)}, 3^{(1)}, 4^{(5)}$
`trivium`	$1^{(2)}, 6^{(2)}, 16^{(4)}$	$1^{(35)}, 6^{(36)}, 16^{(4)}$

TABLE 2.3. – Distribution of the widths of replay windows in source codes and their corresponding execution traces.

the internal loop, the impact of a Ω value smaller than $\lceil \log_2 54 \rceil$ is very small. Increasing Ω would not cost much (see lines V5-8 in Tab. 2.7.1), but this may not be very useful. For instance, using $\Omega = 6$ would reduce the total number of `rpl` executed for `trivium` to (35-1)+(36-1)+2=71 instead of 75 for $\Omega = 4$. In the future, we will evaluate how this parameter impacts protections of the control flow. For the `atoi` code in Tab. 2.3, one can notice that the replay window `w=2` in the source code "disappears" in the trace since it is not executed for the actual data set. We plan to evaluate more codes and data sets in the future.

Currently, we consecutively execute each instruction and its replay(s) before going to the next instruction in the window. To reduce the switching activity in the processor, we do not consider another solution : executing each instruction in the window, then replay them as a block of instructions.

Regarding energy aspect, we also have to investigate links between refetch and memory hierarchy.

2.8. Conclusion and Future Prospects

We propose a *hardware support for instruction replay* in a small RISC processor. It consists in a small extension of the instruction set (one new instruction), a few detection elements (mainly registers and comparators) and light modifications of the processor control. We explore various configurations of internal protection elements for hardware replay.

The core area overhead is about +30% while the clock frequency is reduced by less than 10% on FPGA.

The hardware replay allows to significantly reduce the execution time and code size compared to a pure software replay protection (respective reductions : $\times 3 \rightarrow \times 2$ and $\times 2 \rightarrow \times 1.3$).

The protection efficiency is evaluated using numerous logical fault injections in HDL simulation. It is similar to the one of software replay solutions but with much smaller execution times and code sizes.

In the future, we plan to study deeper pipelines, protection of the `rpl` instruction itself, porting to a RISC-V processor and energy optimizations. We also plan to study hardware protections of the control flow and their combination with hardware replay. Links to secure software development, compilation and security policy management should also be studied. Finally, we would like to perform more complete fault injection simulations and real attacks for security evaluation.

3. Contribution 2 : New Replay Protections

3.1. Introduction

This chapter presents the current version of an article that should be submitted to a journal in the near future. It is intended to be an extension of our DDECS article [3] presented in Chapter 2. Below, we present several additional replay techniques for triplication with majority vote (TV) and randomization that can be deployed in secure systems at a low cost. The replay instruction is extended and new small hardware blocs are added to the architecture of a small 32-bit RISC processor to provide the ability to automate and parameterize multiple types of replay behavior. Various fault detection and fault tolerance elements are added to the processor. Compared to Chapter 2, we also slightly modified the protection of the destination register inside the register file (RF) and not on datapath before RF to increase the protection efficiency.

In the previous chapter, we implemented replay with comparison (RC) solutions for a fixed number of replays. We also evaluated several versions of the processor by adding different combination of hardware detection elements. This chapter proposes additional security features in the processor based on two replay strategies :

1. hardware *replay with comparison* (RC), where the protection mechanism immediately detects faults and reports them to the security policy ;
2. hardware *triplication with majority vote* (TV), where the protection mechanism tries to absorb the effects of injected faults as much as possible ;

For the first strategy, we deal with various versions of RC. Firstly, we implement processor versions that only handle a number of replays `n` fixed in the assembly code. Secondly, we implement processor versions that only handle *random* number of replays (RRC) to lower the chance for an attacker injecting faults at the right times, such that we make it more complex for two related executions to be faulted in the same way. Random replay (RRC) can take two forms (or sub-modes) : The replay number `n` is random and *uniform* for all instructions in the window (RRCU) ; or the replay number `n` is random and *variable* for each instruction in the window (RRCV).

For the second strategy, we implement processor versions that only handle the replay with fault tolerance by adding triplication with majority vote (TV) to increase fault coverage and to achieve a significant gain in terms of execution time and code size compared to software TV (as presented in the state of the art [43, 6] for instance).

We also implement a complete processor version that supports all replay modes : fixed RC, random RC and TV, in order to provide flexibility. The user can choose the appropriate mode for each replay window in the software application, taking into consideration its sensitivity and security characteristics (the specification of the best combinations of replay modes is a future prospect).

To summarize, we have 5 main processor versions :

— the base processor (Fig. 1.9) ;
— a specific version for RC (Fig. 2.1) ;
— a specific version for RRC (Fig. 3.6) ;
— a specific version for TV (Fig. 3.5) ;
— a complete version (in Fig. 3.8).

We implemented two processor versions with or without *refetch* for each of these main protected versions, in order to evaluate the impact of *refetch*.

In this chapter, we first validate the functionality of the processor versions through intensive simulations. Second, we examine the impact of each protection element on implemented versions (RC, RRC, TV, and the complete one) in terms of silicon overhead and maximum frequency and we compare them to the base processor. Next, we execute various protected software applications using our replay modes (RC with `n = 1`, RRCU, RRCV, TV) on the main protected processor versions. We also execute these software applications using software protections (DC and TV) on the base processor. Finally, we evaluate and analyse them in terms of execution time, code size and fault coverage. As opposed to pure software protections, our proposed hardware replay support provides a shorter execution time and code size of the protected applications (reduced number of executed instructions) for a low silicon overhead.

Table 3.1 summarizes our two protection strategies (RC and TV) and the evaluated replay modes.

Strategy	Replay mode
RC	DC `n` = 1 in ASM code
	RRCU `n` random for all instructions in the window RRCV `n` random for each instruction in the window
TV	TV `n` = 2 fixed in HDL code

TABLE 3.1. – The evaluated replay modes.

This chapter is organized as follows. Section 3.2 provides a detailed description of the two protection strategies (RC and TV) and their modes, required hardware modifications and detection and voting elements. Section 3.3 presents the implementation of the processor versions studied in this chapter. Section 3.4 describes the evaluation environment, and Section 3.5 reports and analyzes the results of evaluations of our hardware protections in terms of FPGA implementations, fault injection simulations, and performance (time and code size).

3.2. Concepts Presentation

The `rpl` instruction is intended to secure short critical sequences of instructions (without branchs or jumps) located in a *replay window* (similar to a basic bloc in compilers). Our proposed approach consists of modifying and extending the replay instruction presented in Chapter 2. A new argument is added to `rpl`, named `mode`, in order to make the processor support multiple types of replay (when the processor version supports multiples modes).

`rpl` has three arguments (`rpl mode w [n]`), and indicates that the `w` instructions (each instruction in the replay window) immediately following `rpl` will be executed once and replayed `n` times with a total of `1 + n` subsequent executions (`n` is mandatory only for the fixed RC mode). The argument `mode` indicates the type of replay : RC for `n` fixed, RRC for `n` random, or TV.

Figure 3.1 describes the 4 fields of the `rpl` instruction coding :

— 8-bit *opcode* ;

— value of `w` coded on a maximum of Ω bits ;

— value of `n` coded on a maximum of Θ bits (mandatory only for the fixed RC mode) ;

— replay mode, noted `mode` and coded on 5 bits.

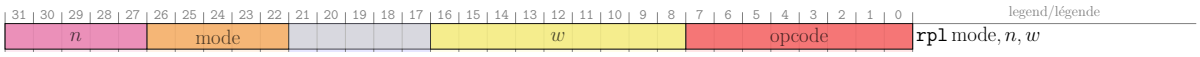


FIGURE 3.1. – The new `rpl` instruction format.

Table 3.2 summarizes the 4 modes that can be supported by the `rpl` instruction and specifies the sources of the parameter `n` according to the `mode` decoded by the processor when `rpl` is executed. The replay window width `w` always comes from the decode unit.

Modes	`n`
RC	from decode
TV	constant in HDL
RRC uniform in the replay window (`RRCU`)	from RNG
RRC variable in the replay window (`RRCV`)	from RNG

TABLE 3.2. – Modes supported by `rpl`.

We also implement versions with or without refetch support, as detailed in Sec. 3.3.4.

3.2.1. Replay with Comparison (RC)

This mode of replay is introduced in Chapter 2. The possible values of the replay window width `w` and the number of replays `n` are fixed by the user in the assembly code. Their maximum sizes are limited by hardware parameters, respectively labelled Ω and Θ in our HDL code such that $1 \leq \mathtt{w} \leq 2^{\Omega}$ and $1 \leq \mathtt{n} \leq 2^{\Theta}$. The values of Ω and Θ are fixed at design time in the HDL code. When $\mathtt{n} = 1$, this RC mode becomes duplication with comparison (DC), and it is similar

to the one proposed in the state of the art [6] (duplication with comparison in software). This behavior is illustrated in Fig. 2.2 presented in Chapter 2.

3.2.2. Random Replay with Comparison (RRC)

If the attacker has access to the assembly code of the software application, he/she can predict the execution trace even using RC mode since `n` is fixed for each replay window. Therefore, we added another mode (RRC) that introduces randomization for the replay number to ensure that the attacker cannot predict the replay number of protected instructions.

The number of replays is no longer fixed as in RC, but is generated by a hardware pseudo-random number generator (PRNG) (e.g. Trivium) [25, 12] and/or a hardware true random number generator (TRNG) [26]. RRC prevents, or at least makes guessing more complicated, an attacker from predicting the number of replays in each window and thus prevents him/her from predicting the right time to inject faults. Random replay allows the window of `w` instructions immediately following `rpl` to be replayed without having to set the number of replays `n` by the user in the assembly code. The number of replays has two forms (or sub-modes) :

1. random but uniform for a window (RRCU), meaning that all instructions in the window have the same random number of replays `n`.
2. random and variable for each instruction in the window (RRCV), which means that the protected instructions have different random number of replays `n`.

The major difference, at hardware level, between these two sub-modes is the throughput required on the RNG. For RRCV, we require a slightly faster RNG that should generate the random value of `n` at each instruction. However, for RRCU, the RNG only needs to generate the random value of `n` at the beginning of the replay window.

The behavior of the two sub-modes of random replay is illustrated in Fig. 3.2. The left side of the figure shows RRCU sub-mode while the right side shows RRCV sub-mode.

The instructions `I1`, `I2` and `I3` must be protected, while `I4` remains unprotected. The window of instructions to be protected has a width of `w=3`. Thus, a replay instruction `rpl rrcu 3` or `rpl rrcv 3` is added just before the instruction window `I1-3`.

The size of the code remains constant even if the mode changes, the only thing that changes is the execution time (in an unpredictable way).

3.2.3. Triplication with Majority Vote (TV)

This new mode of replay adds fault tolerance to the processor with hardware majority vote elements (see Sec. 3.2.6). The maximum size of the replay window width (Ω) and the replay number (`n=2`) are fixed in the HDL code.

This behavior is illustrated on the right side of Fig. 3.3 with 3 instructions : `I1`, `I2` and `I3`. The instructions `I1` and `I2` must be protected by TV and executed 3 times while `I3` remains unprotected. The window of the instructions to be protected has a width of `w=2` and the replays

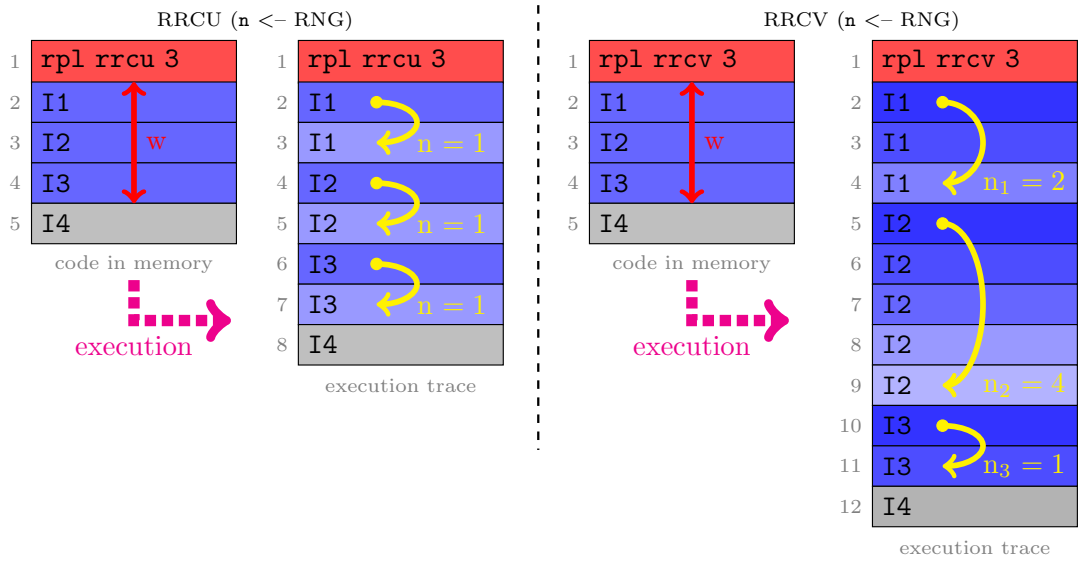


FIGURE 3.2. – Illustration of random replay sub-modes in a small 4-instruction code ('I1', 'I2 and 'I3' are protected while 'I4' is left unprotected).

number is constant in HDL and equal to 2 (for 1+`n` consecutive executions). In this way, an instruction `rpl tv 2` is added just before the replay window `I1-2`.

The left part of Fig. 3.3 illustrates the same example in the case of a pure software TV. In fact, to protect an instruction, software TV has to replace it with about 14 instructions to apply the protection [6]. This severely affects the code size and the execution time. Sec. 1.2.3 of Chapter 1 presents a detailed example of TV in software.

Our hardware TV leads to much smaller codes in memory and fewer executed instructions. The number of original and replayed instructions is the same as software TV. However, instead of inserting several instructions for majority vote in software in the code, in hardware we use only one `rpl` instruction for each replay window and the hardware elements added to the core. The width of the `w` window in the `rpl tv w` instruction leads to fewer instructions in the code as long as `w`>1 compared to software TV.

In this work, we do not consider fault tolerance schemes with more than triplication before majority vote. Using more than triplication, especially a random number of replays, then majority vote can be interesting to increase security policy solutions and should be not be very difficult to add to the processor in a future work.

3.2.4. Opcode and rpl Mode Protections

In order to make the processor capable of raising a decode error (invalid instruction) during a fault injection targeting instructions, we have protected the `opcode` of all processor instructions in the decode unit.

To protect against stuck at 0 or 1 faults in the instruction, we have adopted a coding system that typically prohibits words from being only composed of zeroes or ones. Indeed, such faults

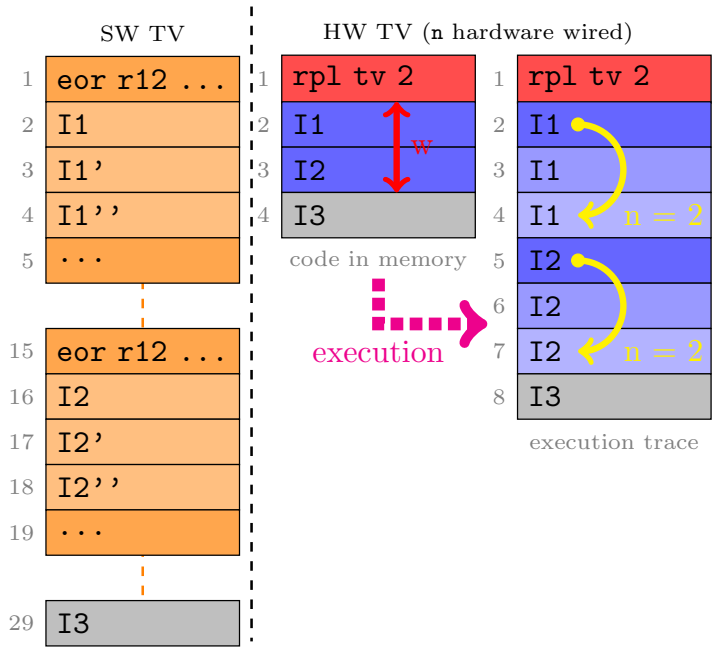


FIGURE 3.3. – Illustration of a typical TV protection in software (left) or in hardware (right) for a small 3-instruction code ('I1' and 'I2' are protected while 'I3' is left unprotected).

will automatically be detected as invalid instructions during decoding if all bits of the `opcode` are set to 0 or 1.

In order to systematically guarantee the detection of a single bit flip, the coding system also allows to choose the combination (a group of words coded on `m` bits) with the largest Hamming distance between its elements. The more the Hamming distance between the words of the combination rises, the more the probability that a fault will lead to an invalid instruction increases. In other words, when we have several possible words for an instruction category (`ALU`, branch ...), we have opted for a Hamming distance higher than at least 2. This justifies the choice of coding the `opcode` in 8 bits even if 5 bits would be sufficient. In practice, we are aware that our coding scheme has a cost in terms of silicon and energy and has an impact on immediate operands which are smaller compared to other processors. However, in hardware security, there is always a trade-off between area and adding protections. This coding system is applied to all processor versions including the base version. In other words, we have chosen to protect the opcodes of the instructions by default in the decode unit.

We have also protected all the fields of the `rpl` instruction using this coding system. Indeed, we have protected the different modes of the `rpl` in order to make the processor able to raise a decode error (invalid instruction) whenever the attacker tries to target the `mode` fields. We have therefore chosen to code the 8 modes on 5 bits whereas it would be sufficient to use only 3 bits. Choosing 5 bits allows us to have a combination with a Hamming distance of 2. Table 3.3 shows the coding of the different `rpl` modes.

rpl modes	coding
rc	00110
rc-ref	10100
tv	00101
tv-ref	10011
rrcu	01100
rrcu-ref	11010
rrcv	01011
rrcv-ref	11001

TABLE 3.3. – The coding of different `rpl` modes.

3.2.5. Required Hardware Modifications

In order to implement hardware replay, a modification of the decode unit is required to decode the replay instruction. Afterward, as mentioned in Chapter 2, we add small registers to the processor to store `n` and `w` (resp. on Θ and Ω bits specified in the HDL code) during replay cycles. Then, we add small counters (resp. on Θ and Ω bits) to keep track of the position inside the protection window and to control the number of replays. Thereafter, we modify the processor control to monitor replay in the pipeline. Then, we add a hardware PRNG (Trivium) in order to generate pseudo-random values for the replay number (`n`) in case of an RRC `mode`. In the future, we have to feed the PRNG from a good TRNG output (such as [26]), but we also can use a stronger PRNG such as Keccak (especialy if it can be shared with another usage). Finally, we also modify the fetch unit to add support for replay and refetch. To do so, we add a flip-flop to memorize the state of the refetch signal, and a few control signals that come specifically from the decoding unit.

For the complete version, we increase the number of FFs from 16 to 17 and the number of LUTs from 3 to 40 in the fetch unit compared to the base version of the processor.

The addition of the `rpl` instruction does not change the behavior of the other instructions. This allows users to protect their code with very small additions to the assembly code (or just a small intrinsic in higher level codes). Automating the insertion of the `rpl` instructions constitutes a future prospect.

In this subsection, we presented how the instruction replay works, described the different replay modes and detailed the modifications required to implement our replay support. In the following subsection we will describe the protection elements (for detection and voting) added to the replay support.

3.2.6. Detection (DE) and Voting (VE) Elements

We add hardware DEs in order to replace the software comparison of instruction duplication with comparison in software protection. We use the same DE implementation presented in Sec. 2.4 of Chapter 2 for the instruction register (`ir`) and for the index of destination register (`id`).

Compared to our solution in Chapter 2, we slightly modified the protection of the destination register (RD), the `prd` detection element. What is protected in Chapter 2 is the dataphath *up to* the register file (RF), *but not* the register inside RF. Then a fault injected on RD inside RF is not detected while faults injected on the datapath in the execution units (ALU, multiplier, LSU) are detected. In our new versions of the processor for RC modes, we decided to protect the destination register inside RF (without a `prd` on the datapath). For that, we added a register to memorize the value stored in RD during the original execution and a comparator which checks that the value stored in RD is the same than the one stored in the protection register. Then each time RD is written, it is automaticaly verified the next cycle. The amount of additional hardware is the same than the in `prd` DE in Chapter 2) (but it is now located inside RF).The benefit of the protection of RD datapath inside RF (instead of before) will be analyzed in Sec. 3.5. The major difference is that now, RF must has three read ports instead of two (where two ports are still used from sources while the third one is used to verify RD the cycle after each write in RF). Adding a third read port on our FPGA implementation has a very small impact when using BRAM for RF. As the fanout on the third port is very small (just one comparator), this new protection implementation for RD may be acceptable for ASIC circuits (but we do not have results).

For TV modes, we used 2 protection registers instead of one for RC and a majority voting element which used the RD register and the 2 protection ones. The cost of the additional hardware required here is the same than in Chapter 2) (but it is now located inside RF).

In the software TV of Fig. 3.3 (left side), the instruction `I1` (line 1) is replayed twice by the instruction `I1'` and `I1''` (lines 3 and 4) with two different destination registers (but similar opcode and sources). In addition, a fourth register `r12` is used to store the voting status (no difference or, one or two differences), then with the help of a majority vote system (11 instructions such as [6]), it determines whether to correct the result (if at most one difference is identified) or to raise a software error (if 3 differences are identified).

The TV mode requires the implementation of majority vote elements in order to correct faults and reduce the propagation of fault effects caused by the attacker. The objective is to protect the data flow of the processor, that is why we protect the instruction register (`ir`), the index of destination register, the result of the execution units, the LSU, and the destination register (`rd`).

Voting (VE) Elements for `ir` and `id`

In order to replace the software majority vote code (explicit comparison instructions in the code) and to protect `id` and `ir`, we implement *hardware voting elements* (VEs) described in Fig. 3.4 in the processor. They consist of two registers and a majority voter. Using hardware VEs avoids the use of additional registers from the register file.

During the replay, the voting element is activated by the processor control to store the results of the first (the original execution) and of the second execution (the first replay) in two different

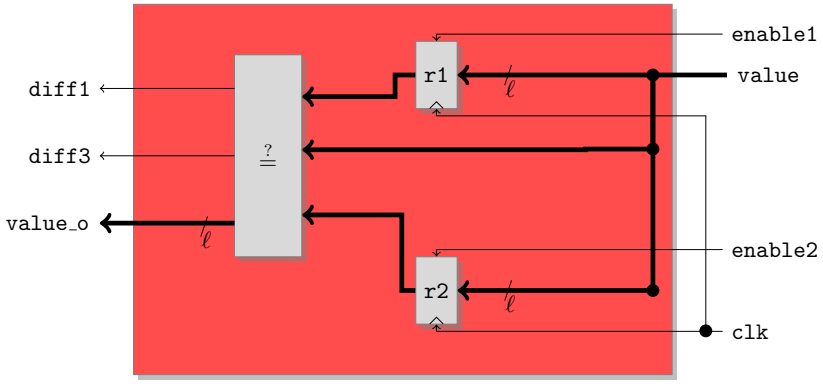


FIGURE 3.4. – Voting element (used for `tvid`, `tvir` in Fig. 3.5).

registers (`r1` and `r2`). At the last replay, the contents of `r1` and `r2` are *compared* to the content of the signal `value`.

The `enable1` and `enable2` signals on the registers are handled by the processor control. The result of the `diff1` signal vote feeds to the processor control to indicate that a value among the 3 executions is different during the replay cycles and the processor was able to correct the fault (`value_o`). The result `diff3` signal feeds the processor control to indicate that the 3 executions are different. In this case, a specific interrupt signal called `default-detected` is raised (see Sec. 3.2.7). The security policy related to the management of this interrupt signal is out of the scope of this work, but would be interesting in future.

The `id` index of the destination register is protected by the 10-bit VE called `tvid`. The `tvir` VE protects the instruction register (see Sec. 3.3 for details).

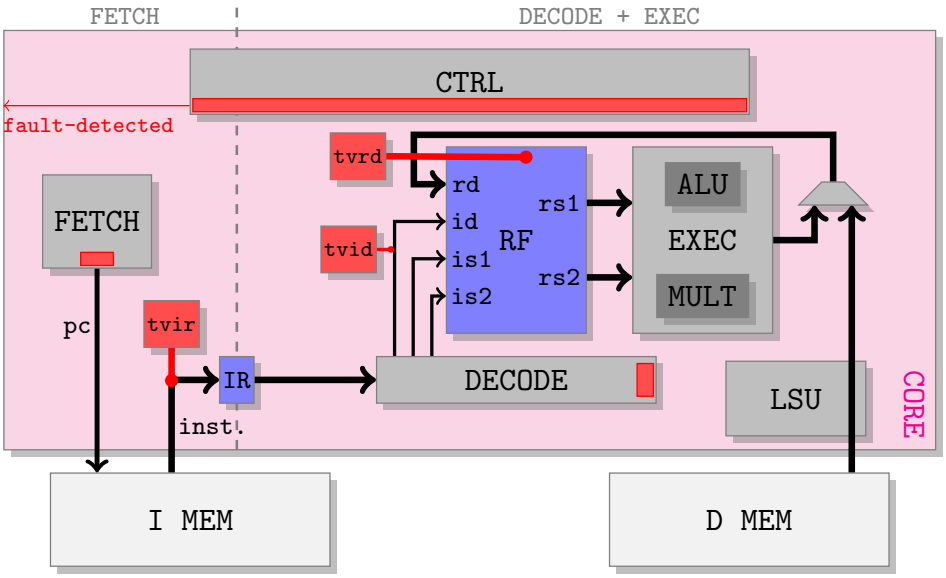


FIGURE 3.5. – Simplified schematic architecture of TV version (not all signals and elements are represented). Red elements are for replay protection and VEs.

Voting (VE) Elements for `rd`

We correct faults in the destination register (`rd`) written in RF.

During replay, the voting element is enabled by the processor control to store the results of the first (first execution) and second (first replay) executions in two different registers (`r1` and `r2`). These two registers are then validated by two 1-bit registers (`v1` and `v2`) during a later replay cycle. In fact, we have added three comparators (`c1`, `c2` and `c3`). `c1` and `c2` check the input of the register file to match the value stored in `r1` and `r2` respectively. In case of a "TRUE" comparison output, these registers must be validated by `v1` and `v2` respectively. `c3` checks if the value of `r1` matches the value of `r2`. In the third execution, we exploit the results of `c1`, `c2`, `c3`, `v1`, and `v2` to perform the majority vote and determine which of the values of `r1` and `r2` will be written to the register file. We also added a register to store the `id` of the currently executed instruction (5 bits) to memorize it for the cycle following the last replay. Finally, we added a comparator `c4` to check if the value written in `rd` matches one of the values of `r1` or `r2` (the validated register). If the result of `c4` is false, then a specific interrupt signal called `default-detected` is raised (see Sec.3.2.7).

The result of the `diff1` signal vote feeds to the processor control to indicate that a value among the 3 executions is different during the replay cycles, and ensure that the processor was able to correct the fault (`value_o`). The result `diff3` signal feeds the processor control to indicate that the 3 executions are different. In this case, a specific interrupt signal `default-detected` is raised (see Sec. 3.2.7).

The 64-bit VE (two 32-bit registers) called `tvrd` in Fig. 3.5 correct faults in the destination register (`rd`) written in RF.

3.2.7. Security Policy

An interruption signal is raised when the replay control detects something "wrong" in the processor. This happens for instance when a DE detects that the currently replayed instruction provides a different value (result or index) than the one stored during the original execution. This interruption should give the control to the *security policy* software in charge of the overall management of the security.

In case of triplication and majority vote, an interruption is raised when all three executions provide different values (`diff3` signal in a VE) or when a correction is performed with one execution providing a different value (`diff1` signal). When a `diff3` signal is activated, the problem can be seen as severe and some (quick) action is required. When a `diff1` signal is activated, the injected fault seems to have been "absorbed" by the protection, but it can be useful to monitor this type of event to make a proper decision (e.g., continue to operate as long as the `diff1` rate is low or decide to abort the application).

The security policy can do several things depending on the application, the attacker model, user security objectives, possible alternate solutions, etc. For instance, it can abort the application, then erase sensible memory elements (e.g., keys) and finaly lock the system. Or it can stop

the current execution and then try to use another version of the application (more robust but with a higher cost). It also can continue the application but using fake data. Depending on the environnement, the security policy can also be implemented using a small dedicated hardware component. The determination of the security policy details is behond this PhD thesis work.

We plan to study different security policy solutions and implementation details in the future. Another future study is the analysis of the reaction time of the system in case of attack. We think that having some hardware support may help to quickly make proper decisions but the space of possible solutions clearly requires applications and user level information we do have as hardware architects.

3.3. Processor Versions Implementation

Multiple *versions* of the processor with different protection solutions (see Sec. 3.1) are implemented.

3.3.1. Processor for RC and RRC Versions

We have incorporated in our processor the detection elements `prd` and `pid` that protect the destination register (the result received from the execution units, LSU and the destination register index). We also added a detection element called `pir` (see Fig. 2.1 for RC version and Fig. 3.6 for RRC version), to protect the instruction register fed from the instruction memory.

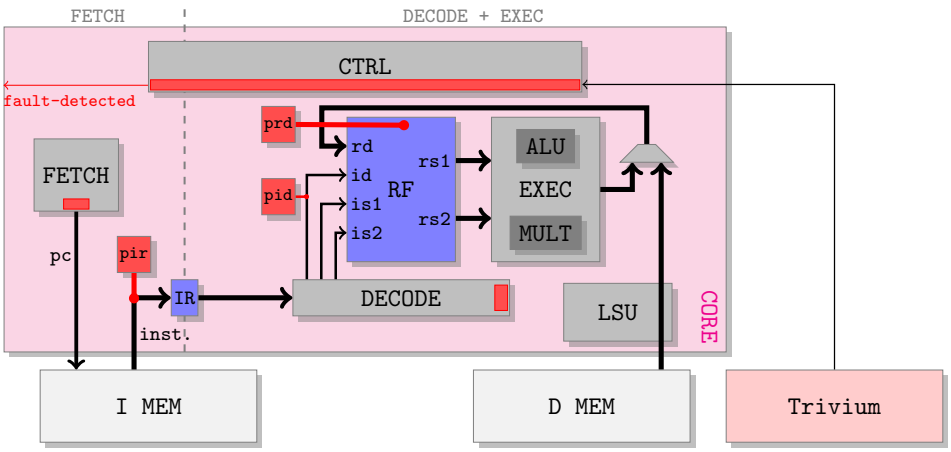


FIGURE 3.6. – Simplified schematic architecture of RRC version (not all signals and elements are represented). Red elements are for replay protection and DEs.

The DE `pir` mainly protects all instruction fields, including indexes for source registers `is1` and `is2`, the destination register index `id`, the opcode or the immediate operands (which can be jump addresses or immediate values for ALU operations). Protecting immediate operands in instructions should increase the protection efficiency.

In addition, protecting each register index (`id`, `is1` and `is2`), immediate operands and the opcode individually with DEs would require the use of $3 \times 5 + 14 + 7 = 36$ bits (the `pir` costs 32 bits). Therefore, adding a `pir` can be a convenient way to protect all instruction fields for a smaller overhead.

3.3.2. Processor Versions for TV

In order to try to correct faults in data flow, we have protected primarily the destination register and secondly the instruction register. We have integrated majority vote elements in the same positions as fault detection elements in the RC version.

The VE `tvrd` and `tvid` elements protect the destination register while the `tvir` protects the instruction register fed by the instruction memory (in Fig. 3.5).

The VE `tvir` corrects faults in all fields of the instruction, including the source register indexes, the destination register index, the opcode and the immediate operands. In fact, protecting immediate operands should also increase the security.

Furthermore, protecting each register index (`id`, `is1` and `is2`), immediate operands and opcode individually with VEs, would require $3 \times (2 \times 5) + 14 \times 2 + 7 \times 2 = 72$ bits (the `tvir` costs is 64 bits). Therefore, adding a `tvir` can be a convenient way to tolerate faults in all fields of the instruction with a lower overhead.

3.3.3. Complete Version (supports RC, RRC and TV)

We implement a complete version that supports all protection modes studied so far. In order to avoid implementing the two types of elements individually and optimize silicon area, we have merged DEs and VEs into a single mechanism of detection and correction which adapts to the replay mode decoded by the processor (in Fig. 3.8). Fig. 3.7 shows the new implementation of the protection mechanism for `ir` and `id`. Indeed, register `r1` is used in both the voting and comparison elements, which allows us to reduce the number of registers used. We added a control signal `mode` to select between either majority vote or comparison.

Figure 3.8 shows the position of the detection and voting elements (`tv`/`p`) in the protected architecture.

3.3.4. Versions with Refetch Support

In order to make the use of `pir` or `tvir` more robust, we have implemented the instruction *refetch* support (see Sec. 2.5 of Chapter 2). By adding this support, we gained an implicit correction of the instruction register in case of a fault detection during a replay. During replays, the instruction is refetched from the `I-MEM` to refresh the instruction register. The current instruction is compared to the original one (first execution) through `pir`/`tvir`.

The *refetch* support detects more faults in the interface between the core and the `I-MEM` during the replay, but this will consume more energy. This is why we implement two possible versions of

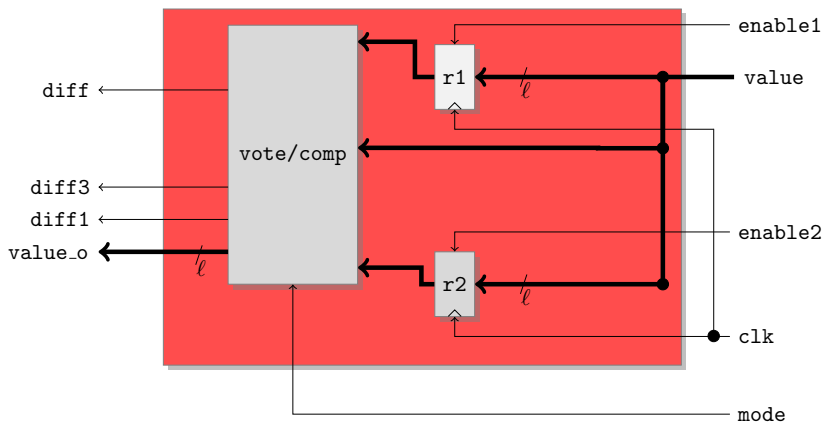


FIGURE 3.7. – Merged detection and voting element (used for `tvid/pid`, `tvir/pir` in Fig. 3.8).

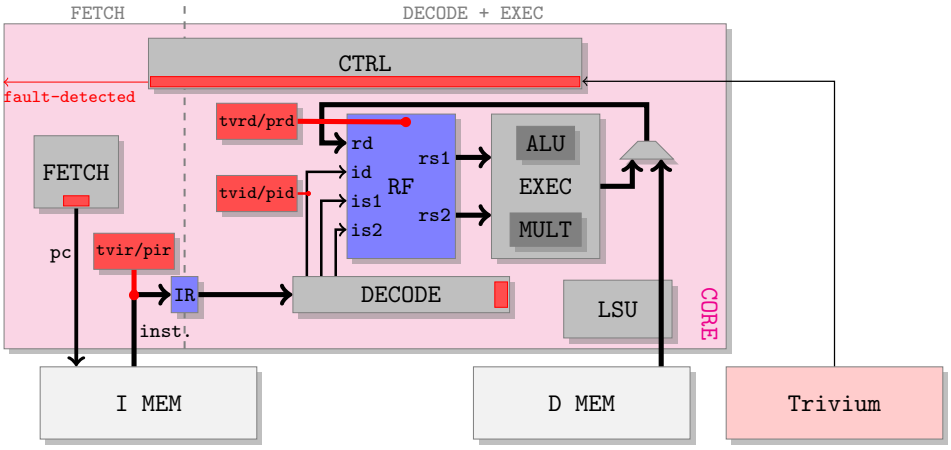


FIGURE 3.8. – Simplified schematic architecture of the complete version (not all signals and elements are represented). Red elements are for replay protection (DEs and VEs).

the processor : with and without *refetch* support. It is then up to the user to choose the version adapted to his/her constraints.

3.4. Description of the Evaluation Environment

HDL codes are implemented in a Zynq 7020 FPGA using Vivado V2018.3 from Xilinx after functional validation with numerous simulations. The next sections present post-synthesis and place&route results for area (LUTs and flip-flops) and performance at the fastest clock frequency obtained for each version of the processor.

Logical fault injection simulation is used to evaluate and compare the protections solutions. Due to the lack of time, we have not been able to complete these simulations by real attacks on a prototype implemented in a FPGA, this will be an important future work. Intensive faults

have been injected in the HDL code at cycle accurate and bit accurate level. Our methodology consists in the following steps :

1. Selection of *processor versions* :

We evaluated a large representative set of processor versions for various combinations of protection elements and hardware parameters. We selected, designed and implemented in FPGA 25 versions as shown in Tab. 3.4. A few specific versions of the processor are the most representative and have been identified using versions "names" in the rest of the document :

— "base" : *original* processor only with the coding protection of the opcodes and without software protection, it corresponds to row V=1 in Tab. 3.4;
— "sw dc" : base processor and application code protected in *software* using ID from Sec. 1.2.3;
— "dc" : base processor with our *hardware replay with comparison* (RC) protection and application code protected using `rpl` instructions with `n`=1 (to be compared with "sw dc"), it corresponds to row V=7 in Tab. 3.4.
— "rrcu" : base processor with our hardware *random replay with comparison* protection with `rpl` instructions in *uniform* mode, it corresponds to row V=11 in Tab. 3.4;
— "rrcv" : base processor with our hardware *random replay and comparison* protection with `rpl` instruction in *variable* mode, it corresponds to row V=11 in Tab. 3.4;
— "sw tv" : base processor and application code protected in *software* using IT from Sec. 1.2.3) ;
— "tv" : base processor with our hardware *triplication with majority vote* (TV) protection (to be compared with "sw tv"), it corresponds to row V=16 in Tab. 3.4;
— "cxx" : base processor with all hardware (merged) protection modes : rc, rrc(u/v), tv ; where the last two letters "xx" define the mode used for `rpl` (fixed RC with "dc", random RC with "ru"/"rv", or TV with "tv"), it corresponds to V=22 in Tab. 3.4.

Processor versions "rrcu" and "rrcv" correspond to the exact same processor hardware but with different software *mode* for the `rpl` instructions in the applications.

All versions of the processor require exactly the same number of BRAMs (16) and DSP blocks (4), since they are only used in the I-MEM, D-MEM, and execution units but not in the replay support (control, DEs or VEs). Then we do not report them in Tab. 3.4.

2. Selection of *application codes* :

We selected a few typical functions commonly used in secure applications (details are presented in Section 2.6) : `verify_pin`, `memcpy`, `atoi`, `trivium`.

3. Selection of *fault scenarios*, *injection times*, *injection locations* and *injection types* :

We inject faults for numerous *scenarios*, *injection times*, *locations* and *types* as illustrated in the pseudo algorithm below :

```
list_codes = [verify_pin, memcpy, atoi, trivium]
list_versions = [base, sw dc, dc, rrcu, rrcv, sw tv, tv, comp]
list_fault_types = [stuck0, stuck1, bit_flip]
list_fault_scenarios = [1f, 2f-d1, 2d-d2]
list_registers = get_registers_from_hdl(...)
for code in list_codes:
  for version in list_versions:
    for fault_scenario in list_fault_scenarios :
      for i in range(0, nb_simulations):
        fault_reg = list_registers[randint(0,len(list_registers)-1)]
        fault_cycle = randint(cycle_min, cycle_max)
        fault_type =  list_fault_types[randint(0, 2)]
        ftrace = insert_fault(fault_scenario, fault_cycle, fault_reg, fault_type)
        state = simulate(version, ftrace, i)
        save(state)
```

In this work, we opted for three fault injection scenarios :

— "1f" : one single fault ;

— "2f-d1" : two faults at distance one (the second injection is delayed by 1 cycle) ;

— "2f-d2" : two faults at distance two (the second injection is delayed by 2 cycles).

We use a large selection of executed instructions as *injection time* : all instructions (we attack the control flow and the data flow) that are executed by the processor can be faulted. Moreover, the cycle to be faulted (*injection times*) is randomly chosen in the interval [`cycle_min`, `cycle_max`].

— `cycle_min` is the cycle where the first instruction of our target application is fetched (we do not target the processor initialization cycles)

— `cycle_max` is the cycle where our target application has finished its normal execution (without any fault).

Regarding *injection locations*, we only inject faults on registers since several physical faults lead to a setup/hold time violation in flip-flops ; [42, 36, 37]. To do so, we target all registers of the base processor and all registers of our hardware protections (DEs, VEs, and replay control). For each simulation, we randomly select one register, even if it is not related to the current instructions (to save simulation time, we limit the registers in RF to the current destination register). Faulting random registers of the whole architecture leads to silent faults, but less than using the exhaustive list of registers for injection locations as we did in Chapter 2. This allows us to explore more fault impacts on both the data and control flows for a reduced simulation time. One possible improvement for future works would be to combine exhaustive faults for the registers related to the currently executed instructions in the processor pipeline and random registers of the other parts of the processor.

We use 3 types of *injections* : stuck at / set to 0 or 1, or a bit flip at a random position of the targeted register.

For each fault scenario, we provide several fault simulations. For each fault simulation, we randomly choose an *injection time*, an *injection type* and an *injection location* (`reg` register) using a uniform distribution. Regarding two faults scenarios, namely "2f-d1" and "2f-d2", we randomly target one register and inject the fault twice. The first *injection time* is chosen randomly but the

second is delayed by 1 or 2 cycles. More elaborate scenarios should be explored in the future.

4. Comparison and analysis of the simulations results :

For each application and processor version, an unfaulted simulation is performed first to collect the reference results (e.g. number of clock cycles, correct values in the register file and the data memory). Then, numerous logical fault injections are performed for each target register, faulted cycle, and fault types. The reported results consist in a complete status of the processor (memory, RF, flags) for each simulation stored in a database for analysis.

The fault simulations results are reported using the following metrics in Sec. 3.5.3 :

— "IS" stands for *injection success* : the code terminates through the legitimate `return` but the injected fault produces an effect (at least one bit difference) in the processor state (register file and memory) compared to the reference execution ;

— "DCF" stands for *divergence of control flow* : the code "jumped" outside the legitimate code ; the legitimate code is still executed but in an unexpected loop detected when the execution time exceeds the reference time plus some threshold (e.g., +50 cycles) ; or the processor crashed ;

— "EOK" stands for *execution OK* : the application returned the good result ;

— "DHW" stands for *detection in HW* : one of the HW detectors was activated (the detection signals are provided by the DEs or VEs) ;

— "DSW" stands for *detection in SW* : one of the SW detectors (branch target) was activated (i.e. a specific address was reached in the software protection management) ;

— "DEC" stands for *decode* error : our coding protection (see Sec. 3.2.4) detected a bad field in the instruction (on opcodes for all instructions, and arguments for `rpl`) ;

— "SUM" (when present) is the sum of all previous columns (should be 100%).

3.5. Analysis of Implementation and Simulation Results

This section presents our results and their analysis in terms of FPGA implementation, computation time, code size and fault injection simulation among several processor versions.

3.5.1. FPGA Implementation Results

Table 3.4 reports FPGA implementation results for all the 25 versions of the processor with various protection elements and hardware parameters configurations (see Sections 3.3 and 3.4 for details). Each line in the table is a processor version denoted Vα with $\alpha \in [1, 25]$. "Protection" columns indicate the configuration of the protection elements and hardware parameters for each version. Gray values are relative values (i.e., $\frac{x-\text{ref}}{\text{ref}}$). Column %1 reports relative values to version V1 while column %2 reports relative values to version V2.

Version V1 is the *base processor* without hardware replay or detection support (but it contains the coding protection of the opcodes). This version is used as a reference value for versions that include only fixed replay or replay with vote.

Table 3.4. – FPGA (Zynq 7020) implementation results for all implemented versions of the processor (V). Each line corresponds to a specific configuration of the protection elements and hardware parameters used in the HDL code of the version.

	Protection												Area						Freq.		
V	HW	rng	prd	pid	pir	Ω	Θ	rnd	ftrd	ftid	ftir	REF	LUT	%1	%2	FF	%1	%2	MHz	%1	%2
1	✘	✘	✘	✘	✘	–	–	✘	✘	✘	✘	✘	666	+00	-35	297	+00	-57	315	+00	+02
2	✘	✓	✘	✘	✘	–	–	✘	✘	✘	✘	✘	1022	+53	+00	688	+132	+00	310	-02	+00
3	✓	✘	✘	✘	✘	4	2	✘	✘	✘	✘	✓	721	+08	-29	320	+08	-53	316	+00	+02
4	✓	✘	✓	✘	✘	4	2	✘	✘	✘	✘	✓	757	+14	-26	360	+21	-48	313	-01	+01
5	✓	✘	✓	✓	✘	4	2	✘	✘	✘	✘	✓	762	+14	-25	366	+23	-47	312	-01	+01
6	✓	✘	✓	✓	✓	4	2	✘	✘	✘	✘	✘	778	+17	-24	397	+34	-42	313	-01	+01
7	✓	✘	✓	✓	✓	4	2	✘	✘	✘	✘	✓	775	+16	-24	398	+34	-42	311	-01	+00
8	✓	✘	✓	✓	✓	3	2	✘	✘	✘	✘	✓	775	+16	-24	396	+33	-42	313	-01	+01
9	✓	✘	✓	✓	✓	2	2	✘	✘	✘	✘	✓	773	+16	-24	394	+33	-43	313	-01	+01
10	✓	✘	✓	✓	✓	1	2	✘	✘	✘	✘	✓	767	+15	-25	392	+32	-43	314	-00	+01
11	✓	✓	✓	✓	✓	4	2	✓	✘	✘	✘	✓	1136	+71	+11	789	+166	+15	308	-02	-01
12	✓	✓	✓	✓	✓	4	1	✓	✘	✘	✘	✓	1130	+70	+11	786	+165	+14	309	-02	-00
13	✓	✘	✘	✘	✘	4	–	✘	✓	✘	✘	✓	776	+17	-24	391	+32	-43	313	-01	+01
14	✓	✘	✘	✘	✘	4	–	✘	✓	✓	✘	✓	785	+18	-23	401	+35	-42	313	-01	+01
15	✓	✘	✘	✘	✘	4	–	✘	✓	✓	✓	✘	826	+24	-19	464	+56	-33	313	-01	+01
16	✓	✘	✘	✘	✘	4	–	✘	✓	✓	✓	✓	821	+23	-20	465	+57	-32	312	-01	+01
17	✓	✘	✘	✘	✘	3	–	✘	✓	✓	✓	✓	820	+23	-20	463	+56	-33	313	-01	+01
18	✓	✘	✘	✘	✘	2	–	✘	✓	✓	✓	✓	820	+23	-20	461	+55	-33	313	-01	+01
19	✓	✘	✘	✘	✘	1	–	✘	✓	✓	✓	✓	819	+23	-20	459	+55	-33	313	-01	+01
20	✓	✓	✓	✓	✓	4	1	✓	✓	✓	✓	✓	1237	+86	+21	860	+190	+25	304	-03	-02
21	✓	✓	✓	✓	✓	4	2	✓	✓	✓	✓	✘	1242	+86	+22	862	+190	+25	306	-03	-01
22	✓	✓	✓	✓	✓	4	2	✓	✓	✓	✓	✓	1245	+87	+22	863	+191	+25	303	-04	-02
23	✓	✓	✓	✓	✓	3	2	✓	✓	✓	✓	✓	1245	+87	+22	861	+190	+25	303	-04	-02
24	✓	✓	✓	✓	✓	2	2	✓	✓	✓	✓	✓	1243	+87	+22	859	+189	+25	303	-04	-02
25	✓	✓	✓	✓	✓	1	2	✓	✓	✓	✓	✓	1239	+86	+21	857	+189	+25	304	-03	-02

Version V2 is the *base processor* with the Trivium PRNG in hardware (see Fig. 3.8). This version is used as a reference value for versions that include random replay. It also shows the impact of our protections on a more complete architecture.

Impact of Hardware Parameters Θ and Ω

Parameter Θ, the size of `n` defined at design time, impacts silicon area and frequency. We implemented versions with $\Theta \in \{1, 2\}$ reported at lines 11, 12, 20, 22 of Tab. 3.4. We notice that using 1 or 2 for Θ leads to at most 1% area and frequency differences. Then, we fixed $\Theta = 2$ for other processor versions since we think that going above 5 executions (`n`$= 2^2$ replays and the original one) is probably not interesting. Obviously, using $\Theta = 2$ allows triplication and reasonable random schemes.

Parameter Ω, the size of `w` defined at design time, also impacts silicon area and frequency. As in Chapter 2, we implemented processors versions for $\Omega \in \{1, 2, 3, 4\}$ reported at lines 7–10,

16–19, and 22–25 of Tab. 3.4. This hardware parameter also leads to very small differences in terms of area and frequency (less than 1%). Then, we fixed $\Omega = 4$ for other processor versions since it allows a sufficient maximum size of the replay window of 16 instructions, see Sec. 2.7.4 for details. Even if future works show that larger protection windows may be interesting for other types of applications, slightly increasing Ω may still lead to moderate hardware overheads.

Impact of Refetch Support

Lines 6, 7, 15, 16, 21 and 22 of Tab. 3.4 correspond to equivalent processor versions in terms of replay protection solutions but *with and without refetch support* (column "REF"). Integrating or not integrating a refetch support in hardware also leads to almost no differences in terms of area and frequency (less than 1%). Then, we decided to only report versions with refetch support for the other evaluations. An interesting future work would be to evaluate the impact of the refetch support in terms of fault coverage (in simulation and using real attacks) and also for energy consumption at system level (including I-MEM to processor core transfers).

Impact of Detection and Vote Elements

Version V3 corresponds to the smallest replay support : only control for replay is added without any detection or vote elements. It leads to 8% area overhead and the same frequency as V1.

Versions V4 to V7 add various combinations of detection elements (DEs) to V3 for RC protection alone. One can notice that when we add a `drd` DE (32 + 8 bits) the number of FFs is increased by 40 (41 in a few cases). Adding a `dir` DE (32 bits) also adds 32 FFs. Adding a `did` DE (5 bits) leads to 5 additional FFs. When all RC DEs are integrated in the processor (40+32+5 bits), the area increase is +34% and the frequency is only reduced by 1%. A +34% area overhead can be considered not so small, but one has to remember that our processor is very small. As soon as more functionalities and a deeper pipeline are used, this overhead would shrink.

Version V11 shows the impact of the random replay support (column "rnd") for RC protection alone with all DEs. Compared to V2 (with hardware PRNG) the impact is quite small : +15% area and -1% clock frequency.

Versions V13 to V19 show the impact of the hardware TV support alone with various vote elements (VEs) configurations. Compared to V3 (with only control replay but no DE), processor V13 adds a VE for `ftrd` (32 bits), and almost increases the number of FFs by $2 \times 32 + 10$. Adding a VE for `ftid` leads to $10 = 2 \times 5$ additional FFs. With all VEs integrated for TV support, the processor area grows by +57% and the clock frequency decreases by 1%.

Version V22 corresponds to the *complete* version of the processor with a hardware support of all protections modes : RC, RRC and TV. Users just have to select the appropriate *mode* for each `rpl` starting of window of protected instructions. Compared to V2 (with hardware PRNG), the area overhead is still quite small (+25%) and almost no speed penalty is observed (the frequency decreases at most by 2%).

Adding a DE or VE to protect a m-bit value requires m or $2m$ additional FFs respectively (or a very close value), which is the expected behavior for `ir` and `id`. Adding a DE or VE to protect a m-bit value requires $m + 8$ or $2m + 10$ additional FFs respectively (or a very close value), which is the expected behavior for `rd`. Moreover, this addition leads to a very negligible attenuation of the maximum frequency, demonstrating that our protection is not on the critical path (this may be not the case in processors with a deeper pipeline).

Adding hardware support for replay, detection and vote support does not cost much. One should keep in mind that our base processor is very small, then the cost of the replay support should be even smaller in more complex processors.

In a processor with a deeper pipeline, hardware replay will require more internal registers and a slightly more complex control than in our 2-stage one. But we think that the overall penalty should still be limited. We still have to evaluate if some internal resources can be shared between deeper pipeline elements and hardware replay support (DEs and VEs).

3.5.2. Timing Performance and Code Size Results

For the main processor versions (see "point 1" in Sec. 3.4), we evaluated the performances of the tested applications (see "point 2" in Sec. 3.4) using the following metrics :

— "CT" means *computation time* (in ns) of unfaulted executions of the application (number of cycles × clock period of the processor version) ;

— "NC" means *number of cycles* of unfaulted executions of the application ;

— "CS" means *code size* (number of instructions) ;

— columns starting by "R. . . " are relative values (CT/NC/CS) to the base version.

Table 3.5 reports the detailed performance results obtained for the 4 tested (unfaulted) applications on each processor version. Table 3.6 summarizes these performance results using averages over all applications compared to the base version.

Table 3.5 confirms that the code size is the same for each application whatever the hardware replay protection version used (RC, RRC and TV) since only the `rpl` mode is changed in the application code. Moreover, hardware replay leads to very small CS increase compared to equivalent protections in software. For instance, RC protection in hardware ("dc" version) only requires 20 to 40% more instructions compared to the unprotected code while "sw dc" doubles or triples the CS. For TV protection, the benefit of the hardware replay is much important since at most 40% more instructions are required while "sw tv" multiplies CS by almost 10 ! Clearly, this may constitute an interest for embedded systems.

Regarding the computation time (CT), the variations for each application among processor versions are almost driven by the variations of the number of cycles (NC) since the clock periods are very close (see Tab. 3.4). The benefit of the hardware replay support is also very clear compared to equivalent software protections (this was our goal). For instance, CT in "sw tv" costs between 7× to 10× the original time while our hardware "tv" processor only leads to at most a 3× overhead (which is minimal with triplication). For RC mode with `n`=1 ("dc" version),

Table 3.5. – Detailed performance results.

V	A	CT	NC	CS	RCT	RNC	RCS
base	a	270.4	85	22	1.0	1.0	1.0
	m	159.1	50	12	1.0	1.0	1.0
	v	130.4	41	20	1.0	1.0	1.0
	t	1129.3	355	81	1.0	1.0	1.0
sw dc	a	639.4	201	47	2.4	2.4	2.1
	m	346.7	109	29	2.2	2.2	2.4
	v	302.2	95	48	2.3	2.3	2.4
	t	2659.3	836	256	2.4	2.4	3.2
dc	a	537.7	167	28	2.0	2.0	1.3
	m	312.3	97	15	2.0	1.9	1.2
	v	251.2	78	27	1.9	1.9	1.4
	t	2398.9	745	89	2.1	2.1	1.1
rrcu	a	910.3	280	28	3.4	3.3	1.3
	m	552.7	170	15	3.5	3.4	1.2
	v	354.4	109	27	2.7	2.7	1.4
	t	4099.5	1261	89	3.6	3.6	1.1
rrcv	a	835.5	257	28	3.1	3.0	1.3
	m	507.2	156	15	3.2	3.1	1.2
	v	367.4	113	27	2.8	2.8	1.4
	t	4099.5	1261	89	3.6	3.6	1.1
sw tv	a	2443.0	768	191	9.0	9.0	8.7
	m	1177.0	370	109	7.4	7.4	9.1
	v	1043.4	328	189	8.0	8.0	9.4
	t	10824.9	3403	970	9.6	9.6	12.0
tv	a	739.0	230	28	2.7	2.7	1.3
	m	430.5	134	15	2.7	2.7	1.2
	v	337.4	105	27	2.6	2.6	1.4
	t	3405.8	1060	89	3.0	3.0	1.1
cdc	a	552.6	167	28	2.0	2.0	1.3
	m	321.0	97	15	2.0	1.9	1.2
	v	258.1	78	27	2.0	1.9	1.4
	t	2465.2	745	89	2.2	2.1	1.1
cru	a	926.5	280	28	3.4	3.3	1.3
	m	562.5	170	15	3.5	3.4	1.2
	v	360.7	109	27	2.8	2.7	1.4
	t	4172.6	1261	89	3.7	3.6	1.1
crv	a	850.4	257	28	3.1	3.0	1.3
	m	516.2	156	15	3.2	3.1	1.2
	v	373.9	113	27	2.9	2.8	1.4
	t	4172.6	1261	89	3.7	3.6	1.1
ctv	a	761.1	230	28	2.8	2.7	1.3
	m	443.4	134	15	2.8	2.7	1.2
	v	347.4	105	27	2.7	2.6	1.4
	t	3507.5	1060	89	3.1	3.0	1.1

the improvement is smaller but our hardware protections are still faster than software ones.

Table 3.6 shows the average performances results. It will be difficult to decrease the computation time below respectively 2 for `n`=1 (dc) and 2.9 for `n`= 2 (tv) since the processor must

Table 3.6. – Average performances results summary.

V	CT	NC	CS	RCT	RNC	RCS
base	422.3	132.8	33.8	1.0	1.0	1.0
sw dc	986.9	310.2	95.0	2.3	2.3	2.8
dc	875.0	271.8	39.8	2.1	2.0	1.2
rrcu	1479.2	455.0	39.8	3.5	3.4	1.2
rrcv	1452.4	446.8	39.8	3.4	3.4	1.2
sw tv	3872.1	1217.2	364.8	9.2	9.2	10.8
tv	1228.2	382.2	39.8	2.9	2.9	1.2
cdc	899.2	271.8	39.8	2.1	2.0	1.2
cru	1505.6	455.0	39.8	3.6	3.4	1.2
crv	1478.3	446.8	39.8	3.5	3.4	1.2
ctv	1264.9	382.2	39.8	3.0	2.9	1.2

execute each protected instruction `n`+1 times (unless using parallelism [47, 38]). Executing the comparisons or votes in parallel to instructions using dedicated hardware elements (DEs and VEs) avoids additional instructions in the application code (the only protection instructions added by a programmer are the `rpl`s).

Regarding the impact of the random replay schemes on the computation time, we performed several simulations just for performance analysis without any fault. Table 3.7 presents the obtained statistics for CT and NC of our 2 random versions (rrcu and rrcv) and tested applications. The rrcu version leads to a slightly larger variability since all the instructions in the protection window share the same random value `n`. Compared to the base version of the processor, getting a 3.5, 3.4× CT overhead for rrcu and rrcv versions is expected for a random number of replays `n` in $\{1, 2, 3, 4\}$ since the mean of values $\{2, 3, 4, 5\}$ is 3.5. The code size is not impacted by the randomization of the `n` value. An interesting future work would be the determination of appropriate bounds for random values of `n` depending on the security policy, type of application, coding style, etc.

Table 3.7. – Statistics (average and standard deviation) for computation time (CT) and number of cycles (NC) obtained using the random replay versions of the processor.

V	CT		NC	
	avg.	std.	avg.	std.
rrcu	1408.4	22.0	431.3	6.5
rrcv	1415.5	10.8	433.6	3.3

3.5.3. Fault Injection Simulations Results

Numerous fault injection simulations have been performed to evaluate the sensitivity of our hardware replay solutions and compare them to software protections (see Sec. 3.4 for details on the simulation environment). Below we report and analyze the main results. More results tables

are given in the appendix A after page 75.

Table 3.8 details the results analyzed for 1000 simulations with a single fault over all applications and main processor versions. Table 3.8 here is also presented in the appendix as Table A.6 to simplify comparisons with other results in the appendix.

Even if there are some variations of the metrics among the applications, one can notice some overall behavior for each processor version. To deal with a global analysis, Table 3.9 presents the average values over the four applications from results in Table 3.8 and expressed as ratios (percentages).

Protections in software and hardware clearly reduce the injection success (IS) rate. The RC type protections in hardware (dc) and software (sw dc) almost divide by 101 and 10 the IS rate respectively. The software version (sw dc) of replay and comparison is slightly less efficient than the hardware version (dc) with 2% instead of 0.2% for IS rate.

The random versions of RC provide a similar IS rate than dc but with a higher computation time (see Table 3.9). We think that using a random scheme alone is probably not interesting. But it may be interesting to start with a random protection mode in order to introduce random jitter, then use cheaper protection modes afterwards. This should be analyzed in the future.

Regarding TV protection, the hardware version (tv) leads to higher EOK rates than the software version (sw tv). Compared to other versions, TV seems to correct faults but comparing the EOK rates over all versions is difficult since faulting random registers leads to frequent silent faults. We plan to build a modified evaluation method to get more accurate results on this topic.

As for RC modes, the software version (sw tv) is slightly less efficient than the hardware version (tv) with 2.8% instead of 0.6% for IS rate. Currently, we are not able to explain the variations among the applications observed in Table 3.8. We plan to study in the future other applications and also specific codes patterns to evaluate their impact on the protection.

Simulation Results for Two Faults Scenarios

In order to evaluate the behavior of our protections in case of multiple faults injections, we performed additional simulations for two faults at distances 1 and 2 (see point 3 in Section 3.4 for details). Tables 3.10 and 3.11 present a summary of these results. Tables A.7 and A.8 in the appendix at pages 81 and 82 present the corresponding complete results for “f2-d1” and “f2-d2” scenarios.

We can notice that globally 2 faults lead to slightly higher IS rates and smaller EOK ones for all versions of the processor (and application codes). The increase is a little smaller for RC type protection in hardware than for software one. For instance, there is a 140% increase for IS rate between “sw dc/2f-d1” and “sw dc/1f” while the increase is about 0% between “dc/2f-d1” and “dc/1f”.

We are fully aware that many other multiple faults scenarios are possible. Due to lack of time, we have not been yet able to try other scenarios, but we plan to work on this aspect in the future. We also hope to be able to perform real fault injections on a real hardware prototype in

Table 3.8. – Fault injection simulations results for one single fault.

V	A	IS	DCF	EOK	DHW	DSW	DEC
base	a	340	117	406	0	0	137
	m	198	169	471	0	0	162
	t	123	131	613	0	0	133
	v	151	210	496	0	0	143
sw dc	a	40	162	370	0	304	124
	m	12	143	424	0	293	128
	t	17	152	420	0	281	130
	v	10	145	391	0	319	135
dc	a	5	58	429	474	0	34
	m	3	34	472	457	0	34
	t	0	47	414	500	0	39
	v	2	51	471	442	0	34
rrcu	a	9	40	377	542	0	32
	m	5	29	420	534	0	12
	t	1	27	369	588	0	15
	v	2	47	414	497	0	40
rrcv	a	4	35	374	566	0	21
	m	2	43	386	533	0	36
	t	1	33	370	577	0	19
	v	1	45	391	532	0	31
sw tv	a	41	148	667	0	24	120
	m	40	152	648	0	29	131
	t	14	147	694	0	24	121
	v	17	153	680	0	13	137
tv	a	17	58	811	71	0	43
	m	3	62	826	61	0	48
	t	2	76	813	64	0	45
	v	1	72	804	76	0	47
cdc	a	4	29	597	351	0	19
	m	3	27	605	333	0	32
	t	0	31	586	357	0	26
	v	0	44	622	307	0	27
cru	a	3	24	568	399	0	6
	m	0	28	564	388	0	20
	t	1	24	569	391	0	15
	v	2	43	579	353	0	23
crv	a	9	22	574	379	0	16
	m	2	22	559	395	0	22
	t	1	32	562	388	0	17
	v	1	33	606	343	0	17
ctv	a	8	73	757	119	0	43
	m	10	67	750	119	0	54
	t	5	58	765	119	0	53
	v	1	57	788	120	0	34

the next years to complete our security evaluation.

Table 3.9. – Simulation results summary for one single fault.

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	20.3	15.7	49.6	0.0	0.0	14.4	100.0
sw dc	2.0	15.1	40.1	0.0	29.9	12.9	100.0
dc	0.2	4.8	44.6	46.8	0.0	3.5	100.0
rrcu	0.4	3.6	39.5	54.0	0.0	2.5	100.0
rrcv	0.2	3.9	38.0	55.2	0.0	2.7	100.0
sw tv	2.8	15.0	67.2	0.0	2.2	12.7	100.0
tv	0.6	6.7	81.3	6.8	0.0	4.6	100.0
cdc	0.2	3.3	60.2	33.7	0.0	2.6	100.0
cru	0.2	3.0	57.0	38.3	0.0	1.6	100.0
crv	0.3	2.7	57.5	37.6	0.0	1.8	100.0
ctv	0.6	6.4	76.5	11.9	0.0	4.6	100.0

Table 3.10. – Simulation results summary for two faults at distance 1.

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	18.5	17.7	49.1	0.0	0.0	14.7	100.0
sw dc	4.8	15.2	37.9	0.0	27.3	14.8	100.0
dc	0.2	3.9	29.3	62.2	0.0	4.3	100.0
rrcu	0.4	3.7	28.9	64.3	0.0	2.8	100.0
rrcv	0.2	3.5	28.7	65.3	0.0	2.4	100.0
sw tv	3.0	15.5	63.5	0.0	3.4	14.6	100.0
tv	1.4	8.8	75.9	8.2	0.0	5.6	100.0
cdc	0.4	3.2	48.3	45.3	0.0	2.8	100.0
cru	0.3	2.5	49.5	45.5	0.0	2.3	100.0
crv	0.2	2.8	50.4	44.8	0.0	1.8	100.0
ctv	1.2	8.4	68.2	16.9	0.0	5.4	100.0

Comments on the Number of Simulations

Regarding the number of simulations to be performed, we empirically studied the evolution of the measured metrics (e.g. IS, DCF, EOK...) over a few processor versions and applications. Figure 3.9 presents this evolution in three representative cases and for 5000 random simulations (the cumulative metrics are obtained between 0 and 5000 simulations with steps of 10). Based on these results, we decided to only perform 1000 random simulations for all our evaluations since after 1000, all reported metrics are stable enough for our comparison purposes.

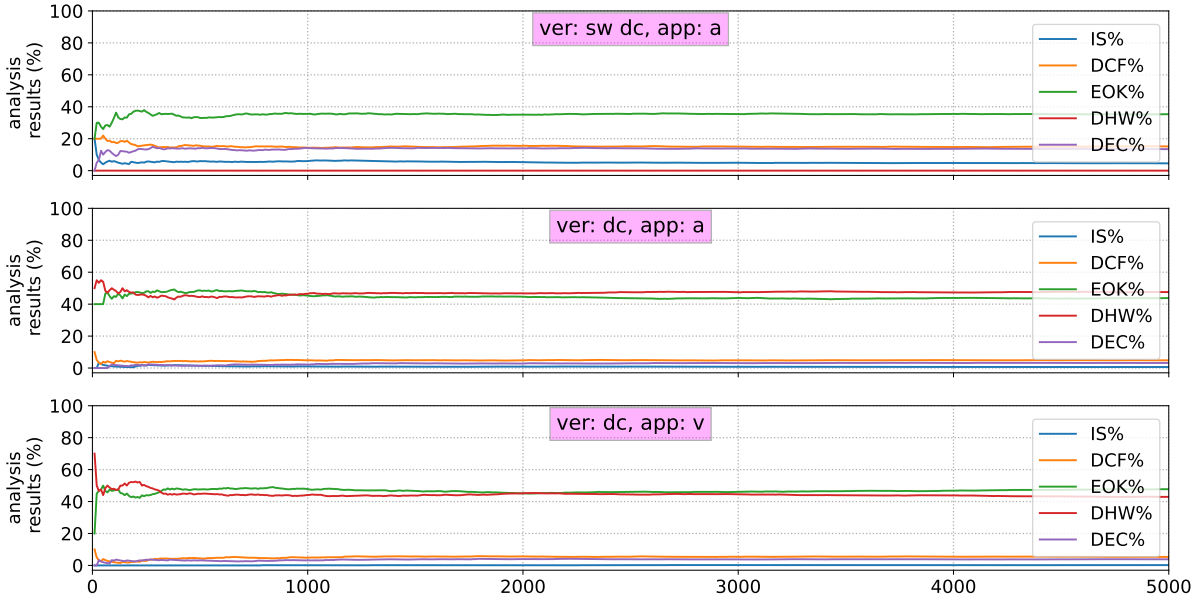
Comments on the Impact of the Coding Protection on Replay Protections

To limit the attacker capacity to inject faults in our `rpl` instructions as well as other instructions, we added a simple coding protection for the opcodes of all instructions and to the other fields of `rpl` instructions (see Section 3.2.4). As this coding protection is quite cheap in term of area, we always use it. But it prevents to evaluate the benefit of our replay protections alone. Then we performed fault injection simulations, with the same environment, for processor ver-

TABLE 3.11. – Simulation results summary for two faults at distance 2.

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	20.3	19.6	44.5	0.0	0.0	15.5	100.0
sw dc	1.9	15.2	33.8	0.0	35.0	13.9	100.0
dc	0.3	5.3	33.2	57.0	0.0	4.2	100.0
rrcu	0.3	3.8	27.9	65.8	0.0	2.2	100.0
rrcv	0.2	4.1	25.7	67.3	0.0	2.6	100.0
sw tv	2.7	17.0	61.7	0.0	4.0	14.7	100.0
tv	0.9	9.1	75.3	9.2	0.0	5.6	100.0
cdc	0.2	3.9	52.0	41.3	0.0	2.7	100.0
cru	0.3	3.1	47.3	47.5	0.0	1.8	100.0
crv	0.1	2.4	49.0	47.0	0.0	1.4	100.0
ctv	0.7	8.7	67.7	17.4	0.0	5.4	100.0

FIGURE 3.9. – Evolution of the simulation metrics over 5000 random simulations.



sions *without* the coding protection of the instructions. The corresponding results are reported in the appendix A.3 at page 83.

For instance Table A.9 (page 83) should be compared to Table A.2 (page 77). Our coding protection reduces the IS rate in all versions of the processor but it is very difficult to perform comparison with a randomized selection of fault locations. We plan to perform more specific simulations to evaluate this aspect in the future.

4. Conclusion

The high presence of embedded systems in our daily lives makes them a prime target for physical attacks such as fault injection attacks. These attacks consist in disrupting a circuit during its operation to cause execution errors in order to bring it to a vulnerable state that can be exploited by the attacker. This thesis proposes hardware support for *automatically replaying instructions* to detect or correct faults. For this purpose, we added a dedicated *replay instruction*, named `rpl`, to the instruction set of a small processor and we implemented a few small *hardware elements* to provide *fault detection* or *fault tolerance*. The `rpl` instruction allows users to specify a *window of instructions to be protected* using an active *protection mode*. The proposed solution aims to protect the data flow of software applications.

Two replay *strategies* were investigated : *replay with comparison* (RC) and *triplication with majority vote* (TV). For the first strategy, we implemented various processor versions handling a fixed or a random number of replays. For the second strategy, we implemented processor versions that only handle the replay with fault tolerance by adding triplication with majority vote to provide fault correction to "absorb" some of the faults injected by the attacker. Finally to provide some flexibility at run time in the protection schemes, we implemented a complete version that supports all replay modes : fixed RC, random RC and TV. We implemented a total of 25 processor versions in FPGA.

We evaluated our processor versions in terms of computation time and code size. We compared our hardware replay with the software replay solutions introduced in the state of the art. The hardware replay leads to significant gains in terms of computation time and code size compared to pure software protections for a moderate silicon overhead (up to 25%) on a very small processor. Regarding computation time, our hardware replay with comparison speeds up applications by a moderate 1.1 factor compared to same protection in software. But for triplication with majority vote, it achieves a 3.2 speed up compared to the software solution. The most important improvement is for code size where reduction factors of 2.4 and 9 are obtained compared to software duplication with comparison and software triplication with majority respectively. This may constitute an interesting benefit for embedded systems where the memory size is important to reduce silicon cost and energy consumption.

To evaluate and compare the protections, we performed logical fault injection in simulations. Numerous faults have been injected in the HDL code at cycle accurate and bit accurate level. Results show that the protection offered by our hardware replay solutions is equivalent to software protections. However, we observed that adding hardware elements to the base processor increases the attack surface. Therefore, the extra material added to support our replay needs to

be protected to minimize the attack surface.

Regarding future prospects, several ideas should be quickly investigated to improve and extend this work. Results gathered during the security evaluation have shown that the proposed hardware replay support increases the attack surface of the processor (i.e., added replay control elements and detection/vote elements). Consequently, we plan to study hardware protection schemes in order to protect the replay support. This could, for example, be addressed by adding verifications at runtime, using a secure logic style or adding shadow registers.

We plan to extend the security evaluation by performing a complete vulnerability analysis of hardware resources with finer timing details (simulations with faults at various times within the clock cycle). This would help to further characterize the robustness of the proposed hardware replay support. Furthermore, another challenge that should be tackled in future works is evaluating our protections using complex applications in order to complete the sensitivity analysis performed so far. Moreover, the security evaluation should be completed with actual fault injection campaigns targeting an FPGA prototype. Finally, another interesting future research topic would be the evaluation of our hardware replay protections regarding observation attacks. Indeed, the replay of instructions could lead to more side-channel leakages by increasing a lot the usage of secret values.

We plan to extend our work on the "complete version" of the processor in order to study the best combinations of the replay modes during time (e.g., start with random replay to create jitter, then protect with a cheaper mode). The goal is to mitigate the attacker capacity to calibrate its attack to inject faults at specific appropriate times during the execution. Furthermore, the proposed protection solutions could be integrated to more complex architectures (e.g. RISC-V cores from then OpenHW Group). This would allow us to study the impact of microarchitecture optimizations (e.g. deeper pipeline, prefetching, speculation, out-of-order execution) on the hardware replay efficiency.

In this thesis work, we have proposed a purely hardware replay solution. We would like to study hybrid solutions that combine software protections with our hardware replay. We could also study the link between our hardware replay and the development of secure software at user and compiler level. The definition and management of efficient security policies mixing hardware and software protections will also constitute an interesting topic. We believe that hybrid protections can provide good tradeoffs in terms of area, performances and energy consumption at system level.

A. Appendix

Chapter 3, written for a future journal submission, only presents the main results from the fault injection simulations performed to evaluate the sensitivity of the different processor versions. This appendix provides more results from these simulations.

A.1. Definitions Used in the Tables

Numerous FPGA implementations and fault injection simulations have been performed over several processor versions and applications recalled below (their abbreviations between "..." are labels for rows and columns in tables below) :

- "A" applications functions : `atoi`, `memcpy`, `trivium`, `verifpin` ;
- "V" versions of the processor :
  - "base" : original processor with only the coding protection for opcodes (see Section 3.2.4) ;
  - "sw dc" : base processor and code protected in *software with ID* (see Section 1.2.3) ;
  - "dc" : base processor with our hardware *replay and compare* (RC) protection and codes protected for the case `n`=1 (in order to directly compare with "sw dc" version) ;
  - "rrcu" : base processor with our hardware *uniform random replay and compare* protection ;
  - "rrcv" : base processor with our hardware *variable random replay and compare* protection ;
  - "sw tv" : base processor and code protected in *software with IT* (see Section 1.2.3) ;
  - "tv" : base processor with our hardware *triplication and majority vote* protection ;
  - "cxx" : base processor with all hardware (merged) protection modes : rc, rrc(u/v), tv ; where the last two letters "xx" define the mode used for `rpl` (fixed RC with "dc", random RC with "ru"/"rv", or TV with "tv") ;
- fault scenarios :
  - "1f" : one single fault ;
  - "2f-d1" : two faults at distance one ;
  - "2d-d2" : two faults at distance two.

For executions without fault, we analyzed the applications performances using the following metrics :

- "CT" means *computation time* of executions without fault (number of cycles × clock period of the processor version) ;
- "NC" means *number of cycles* of executions without fault ;

— “CS” means *code size* in instructions ;
— columns starting by “R. . . ” are relative values (CT/NC/CS) to the base version.

The fault simulations results are reported using the following metrics :

— “IS” means *injection success* : the code terminates through the legitimate `return` but the injected fault produced an effect (at least one bit difference) in the processor state (register file and memory) compared to the reference execution ;
— “DCF” means *divergence of control flow* : the code “jumped” outside the legitimate code ;
— “EOK” means *execution OK* : the application returned the good result ;
— “DHW” means *detection in HW* : one of the HW detector was activated ;
— “DSW” means *detection in SW* : one of the SW detector (branch target) was activated (i.e. a specific address was reached) ;
— “DEC” means *decode* error : our error code (see Section 3.2.4) detected a bad field in the instruction (on opcodes for all instructions, and arguments for `rpl`) ;
— “SUM” (when present) is the sum of all previous columns (should be 100%)

A.2. Simulations for Processor Versions with Instruction Protection

All the processor versions here include the coding protection of the instructions opcodes (and other fields for `rpl`, see Section 3.2.4).

Table A.1 presents the average performances and the relative average values compared to the base version.

TABLE A.1. – Performance results summary.

V	CT	NC	CS	RCT	RNC	RCS
base	422.3	132.8	33.8	1.0	1.0	1.0
sw dc	986.9	310.2	95.0	2.3	2.3	2.8
dc	875.0	271.8	39.8	2.1	2.0	1.2
rrcu	1479.2	455.0	39.8	3.5	3.4	1.2
rrcv	1452.4	446.8	39.8	3.4	3.4	1.2
sw tv	3872.1	1217.2	364.8	9.2	9.2	10.8
tv	1228.2	382.2	39.8	2.9	2.9	1.2
cdc	899.2	271.8	39.8	2.1	2.0	1.2
cru	1505.6	455.0	39.8	3.6	3.4	1.2
crv	1478.3	446.8	39.8	3.5	3.4	1.2
ctv	1264.9	382.2	39.8	3.0	2.9	1.2

Tables A.2, A.3 and A.4 present the average values for fault injection simulation results expressed as percentages (respectively for one single fault, two faults at distance 1 and two faults at distance 2). Tables A.2, A.3 and A.4 are averages ratios (%) from Tables A.6, A.7 and A.8.

TABLE A.2. – Simulation results summary for one single fault.

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	20.3	15.7	49.6	0.0	0.0	14.4	100.0
sw dc	2.0	15.1	40.1	0.0	29.9	12.9	100.0
dc	0.2	4.8	44.6	46.8	0.0	3.5	100.0
rrcu	0.4	3.6	39.5	54.0	0.0	2.5	100.0
rrcv	0.2	3.9	38.0	55.2	0.0	2.7	100.0
sw tv	2.8	15.0	67.2	0.0	2.2	12.7	100.0
tv	0.6	6.7	81.3	6.8	0.0	4.6	100.0
cdc	0.2	3.3	60.2	33.7	0.0	2.6	100.0
cru	0.2	3.0	57.0	38.3	0.0	1.6	100.0
crv	0.3	2.7	57.5	37.6	0.0	1.8	100.0
ctv	0.6	6.4	76.5	11.9	0.0	4.6	100.0

Table A.3. – Simulation results summary for two faults at distance 1 (identical to Tab. 3.10).

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	18.5	17.7	49.1	0.0	0.0	14.7	100.0
sw dc	4.8	15.2	37.9	0.0	27.3	14.8	100.0
dc	0.2	3.9	29.3	62.2	0.0	4.3	100.0
rrcu	0.4	3.7	28.9	64.3	0.0	2.8	100.0
rrcv	0.2	3.5	28.7	65.3	0.0	2.4	100.0
sw tv	3.0	15.5	63.5	0.0	3.4	14.6	100.0
tv	1.4	8.8	75.9	8.2	0.0	5.6	100.0
cdc	0.4	3.2	48.3	45.3	0.0	2.8	100.0
cru	0.3	2.5	49.5	45.5	0.0	2.3	100.0
crv	0.2	2.8	50.4	44.8	0.0	1.8	100.0
ctv	1.2	8.4	68.2	16.9	0.0	5.4	100.0

Table A.4. – Simulation results summary for two faults at distance 2 (identical to Tab. 3.11).

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	20.3	19.6	44.5	0.0	0.0	15.5	100.0
sw dc	1.9	15.2	33.8	0.0	35.0	13.9	100.0
dc	0.3	5.3	33.2	57.0	0.0	4.2	100.0
rrcu	0.3	3.8	27.9	65.8	0.0	2.2	100.0
rrcv	0.2	4.1	25.7	67.3	0.0	2.6	100.0
sw tv	2.7	17.0	61.7	0.0	4.0	14.7	100.0
tv	0.9	9.1	75.3	9.2	0.0	5.6	100.0
cdc	0.2	3.9	52.0	41.3	0.0	2.7	100.0
cru	0.3	3.1	47.3	47.5	0.0	1.8	100.0
crv	0.1	2.4	49.0	47.0	0.0	1.4	100.0
ctv	0.7	8.7	67.7	17.4	0.0	5.4	100.0

Table A.5 presents the detailed performances results over each application and each processor version.

Table A.5. – Detailed performance results.

V	A	CT	NC	CS	RCT	RNC	RCS
base	a	270.4	85	22	1.0	1.0	1.0
	m	159.1	50	12	1.0	1.0	1.0
	v	130.4	41	20	1.0	1.0	1.0
	t	1129.3	355	81	1.0	1.0	1.0
sw dc	a	639.4	201	47	2.4	2.4	2.1
	m	346.7	109	29	2.2	2.2	2.4
	v	302.2	95	48	2.3	2.3	2.4
	t	2659.3	836	256	2.4	2.4	3.2
dc	a	537.7	167	28	2.0	2.0	1.3
	m	312.3	97	15	2.0	1.9	1.2
	v	251.2	78	27	1.9	1.9	1.4
	t	2398.9	745	89	2.1	2.1	1.1
rrcu	a	910.3	280	28	3.4	3.3	1.3
	m	552.7	170	15	3.5	3.4	1.2
	v	354.4	109	27	2.7	2.7	1.4
	t	4099.5	1261	89	3.6	3.6	1.1
rrcv	a	835.5	257	28	3.1	3.0	1.3
	m	507.2	156	15	3.2	3.1	1.2
	v	367.4	113	27	2.8	2.8	1.4
	t	4099.5	1261	89	3.6	3.6	1.1
sw tv	a	2443.0	768	191	9.0	9.0	8.7
	m	1177.0	370	109	7.4	7.4	9.1
	v	1043.4	328	189	8.0	8.0	9.4
	t	10824.9	3403	970	9.6	9.6	12.0
tv	a	739.0	230	28	2.7	2.7	1.3
	m	430.5	134	15	2.7	2.7	1.2
	v	337.4	105	27	2.6	2.6	1.4
	t	3405.8	1060	89	3.0	3.0	1.1
cdc	a	552.6	167	28	2.0	2.0	1.3
	m	321.0	97	15	2.0	1.9	1.2
	v	258.1	78	27	2.0	1.9	1.4
	t	2465.2	745	89	2.2	2.1	1.1
cru	a	926.5	280	28	3.4	3.3	1.3
	m	562.5	170	15	3.5	3.4	1.2
	v	360.7	109	27	2.8	2.7	1.4
	t	4172.6	1261	89	3.7	3.6	1.1
crv	a	850.4	257	28	3.1	3.0	1.3
	m	516.2	156	15	3.2	3.1	1.2
	v	373.9	113	27	2.9	2.8	1.4
	t	4172.6	1261	89	3.7	3.6	1.1
ctv	a	761.1	230	28	2.8	2.7	1.3
	m	443.4	134	15	2.8	2.7	1.2
	v	347.4	105	27	2.7	2.6	1.4
	t	3507.5	1060	89	3.1	3.0	1.1

Tables A.6, A.7 and A.8 present the detailed analysis results for 1000 simulations over each application and each processor version respectively for “1f”, “2f-d1”, “2f-d2”.

Table A.6. – Fault injection simulations results for one single fault (identical to Tab. 3.8).

V	A	IS	DCF	EOK	DHW	DSW	DEC
base	a	340	117	406	0	0	137
	m	198	169	471	0	0	162
	t	123	131	613	0	0	133
	v	151	210	496	0	0	143
sw dc	a	40	162	370	0	304	124
	m	12	143	424	0	293	128
	t	17	152	420	0	281	130
	v	10	145	391	0	319	135
dc	a	5	58	429	474	0	34
	m	3	34	472	457	0	34
	t	0	47	414	500	0	39
	v	2	51	471	442	0	34
rrcu	a	9	40	377	542	0	32
	m	5	29	420	534	0	12
	t	1	27	369	588	0	15
	v	2	47	414	497	0	40
rrcv	a	4	35	374	566	0	21
	m	2	43	386	533	0	36
	t	1	33	370	577	0	19
	v	1	45	391	532	0	31
sw tv	a	41	148	667	0	24	120
	m	40	152	648	0	29	131
	t	14	147	694	0	24	121
	v	17	153	680	0	13	137
tv	a	17	58	811	71	0	43
	m	3	62	826	61	0	48
	t	2	76	813	64	0	45
	v	1	72	804	76	0	47
cdc	a	4	29	597	351	0	19
	m	3	27	605	333	0	32
	t	0	31	586	357	0	26
	v	0	44	622	307	0	27
cru	a	3	24	568	399	0	6
	m	0	28	564	388	0	20
	t	1	24	569	391	0	15
	v	2	43	579	353	0	23
crv	a	9	22	574	379	0	16
	m	2	22	559	395	0	22
	t	1	32	562	388	0	17
	v	1	33	606	343	0	17
ctv	a	8	73	757	119	0	43
	m	10	67	750	119	0	54
	t	5	58	765	119	0	53
	v	1	57	788	120	0	34

Table A.7. – Fault injection simulations results for two faults at distance 1.

V	A	IS	DCF	EOK	DHW	DSW	DEC
base	a	294	156	399	0	0	151
	m	203	175	478	0	0	144
	t	108	170	572	0	0	150
	v	135	206	517	0	0	142
sw dc	a	77	152	320	0	299	152
	m	44	155	384	0	263	154
	t	42	155	421	0	250	132
	v	29	145	391	0	280	155
dc	a	3	45	293	625	0	34
	m	6	39	310	596	0	49
	t	0	25	269	669	0	37
	v	0	48	299	599	0	54
rrcu	a	7	29	269	672	0	23
	m	4	32	318	621	0	25
	t	1	44	277	657	0	21
	v	3	43	290	621	0	43
rrcv	a	2	28	268	678	0	24
	m	1	36	299	643	0	21
	t	1	31	276	677	0	15
	v	3	44	306	613	0	34
sw tv	a	43	140	627	0	35	155
	m	33	165	619	0	23	160
	t	15	142	673	0	37	133
	v	29	172	622	0	39	138
tv	a	16	79	752	94	0	59
	m	13	98	750	79	0	60
	t	17	76	781	78	0	48
	v	11	98	754	79	0	58
cdc	a	8	38	492	437	0	25
	m	5	31	469	463	0	32
	t	0	28	457	490	0	25
	v	1	32	515	422	0	30
cru	a	5	23	488	456	0	28
	m	3	25	515	433	0	24
	t	2	23	487	469	0	19
	v	1	27	489	461	0	22
crv	a	3	29	501	449	0	18
	m	4	26	496	455	0	19
	t	1	28	499	462	0	10
	v	1	29	520	425	0	25
ctv	a	22	65	666	198	0	49
	m	12	93	706	136	0	53
	t	8	90	663	180	0	59
	v	4	87	693	163	0	53

Table A.8. – Fault injection simulations results for two faults at distance 2.

V	A	IS	DCF	EOK	DHW	DSW	DEC
base	a	325	156	373	0	0	146
	m	212	221	410	0	0	157
	t	132	189	529	0	0	150
	v	143	219	470	0	0	168
sw dc	a	20	130	318	0	397	135
	m	13	178	328	0	324	157
	t	30	138	370	0	327	135
	v	14	164	338	0	354	130
dc	a	6	49	316	583	0	46
	m	5	50	329	576	0	40
	t	0	49	332	585	0	34
	v	1	64	352	535	0	48
rrcu	a	1	33	233	715	0	18
	m	7	31	307	630	0	25
	t	0	25	264	690	0	21
	v	4	63	311	599	0	23
rrcv	a	3	41	241	691	0	24
	m	1	37	283	651	0	28
	t	2	41	261	673	0	23
	v	1	45	244	679	0	31
sw tv	a	33	165	616	0	34	152
	m	38	170	595	0	38	159
	t	14	169	651	0	36	130
	v	23	176	605	0	51	145
tv	a	21	91	727	103	0	58
	m	6	92	768	87	0	47
	t	6	86	770	73	0	64
	v	3	93	747	104	0	53
cdc	a	2	43	505	428	0	22
	m	3	38	510	424	0	25
	t	2	39	516	417	0	26
	v	2	34	549	382	0	33
cru	a	8	31	471	475	0	15
	m	2	31	483	463	0	21
	t	1	28	453	502	0	16
	v	0	34	487	460	0	19
crv	a	3	30	472	489	0	6
	m	2	28	508	445	0	17
	t	0	12	498	473	0	17
	v	1	26	483	472	0	18
ctv	a	12	81	664	193	0	50
	m	8	88	703	149	0	52
	t	5	85	659	195	0	56
	v	4	95	681	160	0	60

A.3. Simulations for Processor Versions without Instruction Protection

To evaluate the replay protections alone, we performed similar fault injection simulations for processor versions *without* the coding protection of the instructions from Section 3.2.4.

Table A.9 presents the corresponding results.

TABLE A.9. – Simulation results summary for one single fault for replay protections only (without instruction coding protection).

V	IS	DCF	EOK	DHW	DSW	DEC	SUM
base	26.6	17.3	56.0	0.0	0.0	0.0	100.0
sw dc	3.9	15.8	44.0	0.0	36.3	0.0	100.0
dc	0.4	5.3	44.9	49.4	0.0	0.0	100.0
rrcu	0.5	3.9	40.0	55.6	0.0	0.0	100.0
rrcv	0.1	3.1	40.0	56.8	0.0	0.0	100.0
sw tv	3.8	15.2	77.8	0.0	3.2	0.0	100.0
tv	1.4	8.0	82.7	7.9	0.0	0.0	100.0

Table A.10 present the detailed analysis results for 1000 simulations over each application and each processor version without coding protections of instructions for one single fault.

Table A.10. – Fault injection simulations results for one single fault for replay protections only (without instruction coding protection).

V	A	IS	DCF	EOK	DHW	DSW	DEC
base	a	419	147	434	0	0	0
	m	295	209	496	0	0	0
	t	149	130	721	0	0	0
	v	202	208	590	0	0	0
sw dc	a	78	158	367	0	397	0
	m	28	169	458	0	345	0
	t	36	146	492	0	326	0
	v	15	158	443	0	384	0
dc	a	8	49	441	502	0	0
	m	5	40	453	502	0	0
	t	1	61	425	513	0	0
	v	2	63	475	460	0	0
rrcu	a	10	36	381	573	0	0
	m	3	42	409	546	0	0
	t	3	33	381	583	0	0
	v	2	45	431	522	0	0
rrcv	a	4	28	398	570	0	0
	m	0	33	390	577	0	0
	t	0	31	380	589	0	0
	v	0	33	430	537	0	0
sw tv	a	42	142	795	0	21	0
	m	48	160	759	0	33	0
	t	28	157	780	0	35	0
	v	32	149	779	0	40	0
tv	a	21	75	800	104	0	0
	m	19	86	824	71	0	0
	t	5	80	857	58	0	0
	v	12	78	827	83	0	0

Bibliographie

[1] Common criteria for information technology security evaluation, April 2017. Version 3.1, Revision 5.

[2] Michel Agoyan, Jean-Max Dutertre, David Naccache, Bruno Robisson, and Assia Tria. When clocks fail: On critical paths and clock faults. In *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, Springer, April 2010, pages 182–193.

[3] Noura Ait Manssour, Vianney Lapotre, Guy Gogniat, and Arnaud Tisserand. Processor extensions for hardware instruction replay against fault injection attacks. In *Proc. International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, IEEE, April 2022, pages 26–31.

[4] Frederic Amiel, Christophe Clavier, and Michael Tunstall. Fault analysis of DPA-resistant algorithms. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Springer, 2006, pages 223–236.

[5] Hagai Bar-El, Hamid Choukri, David Naccache, Michael Tunstall, and Claire Whelan. The sorcerer's apprentice guide to fault attacks. *Proceedings of the IEEE*, 94(2) :370–382, 2006.

[6] Alessandro Barenghi, Luca Breveglieri, Israel Koren, Gerardo Pelosi, and Francesco Regazzoni. Countermeasures against fault attacks on software implemented AES: Effectiveness and cost. In *Proc. Workshop on Embedded Systems Security (WESS)*, Scottsdale, AZ, USA, October 2010, pages 7 :1–10. ACM.

[7] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of checking cryptographic protocols for faults. In *Proc. Annual International Conference on Theory and Applications of Cryptographic Techniques (EUROCRYPT)*, volume 1233 of *LNCS*, Springer, 1997, pages 37–51.

[8] Dan Boneh, Richard A. DeMillo, and Richard J. Lipton. On the importance of eliminating errors in cryptographic computations. *Journal of Cryptology*, 14(2) :101–119, 2001.

[9] Keith A. Bowman, James W. Tschanz, Nam Sung Kim, Janice C. Lee, Chris B. Wilkerson, Shih-Lien L. Lu, Tanay Karnik, and Vivek K. De. Energy-efficient and metastability-immune resilient circuits for dynamic variation tolerance. *IEEE Journal of Solid State Circuits*, 44(1) :49–63, 2009.

[10] Claudio Bozzato, Riccardo Focardi, and Francesco Palmarini. Shaping the glitch: Optimizing voltage fault injection attacks. *Transactions on Cryptographic Hardware and Embedded Systems (TCHES)*, pages 199–224, 2019.

[11] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, August 2004, pages 16–29.

[12] Christophe De Cannière and Bart Preneel. Trivium specifications. Technical report, eSTREAM : the ECRYPT Stream Cipher Project, 2006.

[13] Ruan de Clercq. *Hardware-supported Software and Control Flow Integrity*. PhD thesis, KU Leuven, November 2017.

[14] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, Philippe Orsatelli, Philippe Maurine, and Assia Tria. Injection of transient faults using electromagnetic pulses – practical results on a cryptographic system. *IACR Cryptology ePrint Archive*, 2012.

[15] Amine Dehbaoui, Jean-Max Dutertre, Bruno Robisson, and Assia Tria. Electromagnetic transient faults injection on a hardware and a software implementations of AES. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, IEEE, September 2012, pages 7–15.

[16] Amine Dehbaoui, Amir-Pasha Mirbaha, Nicolas Moro, Jean-Max Dutertre, and Assia Tria. Electromagnetic glitch on the AES round counter. In *Proc. International Workshop on Constructive Side-Channel Analysis and Secure Design (COSADE)*, Springer, March 2013, pages 17–31.

[17] Louis Dureuil, Guillaume Petiot, Marie-Laure Potet, Thanh-Ha Le, Aude Crohen, and Philippe de Choudens. FISSC: A fault injection and simulation secure collection. In *Proc. International Conference on Computer Safety, Reliability, and Security (SAFECOMP)*, Springer, September 2016, pages 3–11.

[18] Louis Dureuil, Marie-Laure Potet, Philippe Choudens, Cécile Dumas, and Jessy Clédière. From code review to fault injection attacks : Filling the gap using fault model inference. In *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, ACM, November 2015, pages 107–124.

[19] Jean-Max Dutertre, Stephan De Castro, Alexandre Sarafianos, Noémie Boher, Bruno Rouzeyre, Mathieu Lisart, Joel Damiens, Philippe Candelier, Marie-Lise Flottes, and Giorgio Di Natale. Laser attacks on integrated circuits: From CMOS to FD-SOI. In *Proc. Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, IEEE, May 2014, pages 1–6.

[20] Jean-Max Dutertre, Jacques JA Fournier, Amir-Pasha Mirbaha, David Naccache, Jean-Baptiste Rigaud, Bruno Robisson, and Assia Tria. Review of fault injection mechanisms and consequences on countermeasures design. In *Proc. Design Technology of Integrated Systems in Nanoscale Era (DTIS)*, IEEE, April 2011, pages 1–6.

[21] Jean-Max Dutertre, Amir-Pasha Mirbaha, Assia Tria, Bruno Robisson, and Michel Agoyan. Revue expérimentale des techniques d'injection de fautes. Présentation journées GDR SoC-SiP, March 2010.

[22] Nadia El Mrabet, Luca De Feo, and Simon Pontié. Sécurisation matérielle de cryptographie post-quantique basée sur les isogénies entre courbes elliptiques. Annonce de sujet de thèse.

[23] I. D. Elliott and I. L. Sayers. Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction. *Proceedings of the IEEE Computers and Digital Techniques*, 137(1) :88–102, 1990.

[24] Nahid Farhady Ghalaty, Bilgiday Yuce, Mostafa Taha, and Patrick Schaumont. Differential fault intensity analysis. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, IEEE, September 2014, pages 49–58.

[25] Tim Good and Mohammed Benaissa. Hardware performance of eStream phase-III stream cipher candidates. In *Proc. Workshop on the State of the Art of Stream Ciphers (SASC)*, February 2008, pages 163–173.

[26] P. Haddad, V. Fischer, F. Bernard, and J. Nicolai. A physical approach for stochastic modeling of TERO-based TRNG. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, T. Guneysu and H. Handschuh, editors, volume 9293 of *LNCS*, Saint-Malo, France, September 2015, pages 357–372. Springer.

[27] Darshana Jayasinghe, Aleksandar Ignjatovic, Roshan Ragel, Jude Angelo Ambrose, and Sri Parameswaran. QuadSeal: Quadruple balancing to mitigate power analysis attacks with variability effects and electromagnetic fault injection attacks. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 26(5) :33 :1–36, 2021.

[28] Duško Karaklajić, Jörn-Marc Schmidt, and Ingrid Verbauwhede. Hardware designer's guide to fault attacks. *IEEE Transactions on Very Large Scale Integration Systems*, 21(12) :2295–2306, 2013.

[29] Ilhyun Kim and Mikko H. Lipasti. Understanding scheduling replay schemes. In *Proc. International Symposium on High Performance Computer Architecture (HPCA)*, IEEE, February 2004, pages 198–209.

[30] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, and Thomas Prescher. Spectre attacks: Exploiting speculative execution. In *Proc. European Symposium on Security and Privacy (EuroS&P)*, IEEE, May 2019, pages 1–19.

[31] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Proc. Annual Cryptology Conference (CRYPTO)*, Springer, August 1999, pages 388–397.

[32] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In *Proc. Annual Cryptology Conference (CRYPTO)*, Springer, August 1996, pages 104–113.

[33] Thomas Korak and Michael Hoefler. On the effects of clock and power supply tampering on two microcontroller platforms. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, IEEE, September 2014, pages 8–17.

[34] Benoit Lasbouygues, Robin Wilson, Nadine Azemard, and Philippe Maurine. Timing analysis in presence of supply voltage and temperature variations. In *Proc. International symposium on Physical design (ISPD)*, ACM, April 2006, pages 10–16.

[35] Johan Laurent. *Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle*. PhD thesis, Université Grenoble Alpes, November 2020.

[36] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. On the importance of analysing microarchitecture for accurate software fault models. In *Proc. Euromicro Conference on Digital System Design (DSD)*, IEEE, August 2018, pages 561–564.

[37] Johan Laurent, Vincent Beroulle, Christophe Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. Cross-layer analysis of software fault models and countermeasures against hardware fault attacks in a RISC-V processor. *Microprocessors and Microsystems (MICPRO)*, 71 :1–10, 2019.

[38] Tuo Li, Jude Angelo Ambrose, Roshan Ragel, and Sri Parameswaran. Processor design for soft errors: Challenges and state of the art. *ACM Computing Surveys (CSUR)*, 49(3) : 57 :1–44, 2016.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown. *arXiv preprint arXiv :1801.01207*, 2018.

[40] Paolo Maistri and Régis Leveugle. Double-data-rate computation as a countermeasure against fault analysis. *IEEE Transactions on Computers*, 57(11) :1528–1539, 2008.

[41] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power Analysis Attacks. Revealing the Secrets of Smart Cards*. Springer, 2007.

[42] Nicolas Moro, Karine Heydemann, Amine Dehbaoui, Bruno Robisson, and Emmanuelle Encrenaz. Experimental evaluation of two software countermeasures against fault attacks. In *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, Arlington, VA, USA, May 2014, pages 112–117. IEEE.

[43] Nicolas Moro, Karine Heydemann, Emmanuelle Encrenaz, and Bruno Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering (JCEN)*, 4(3) :145–156, 2014.

[44] David Patterson and John Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, 2017.

[45] Julien Proy. *Sécurisation systématique d'applications embarquées contre les attaques physiques*. PhD thesis, Université Paris Sciences et Lettres, June 2019.

[46] Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4), 2017.

[47] Rafail Psiakis. *Performance Optimization Mechanisms for Fault-Resilient VLIW Processors*. PhD thesis, Université Rennes 1, December 2018.

[48] Jean-Jacques Quisquater and David Samyde. Eddy current for magnetic analysis with active sensor. *Proc. Smart Card Programming and Security (E-smart)*, pages 185–194, 2002.

[49] Jörn-Marc Schmidt and Michael Hutter. Optical and EM fault attacks on CRT-based RSA : Concrete results. In *Proc. Austrian Workhop on Microelectronics (Austrochip)*, Verlag der Technischen Universität Graz, October 2007, pages 61–67.

[50] Sergei Skorobogatov. Local heating attacks on flash memory devices. In *Proc. International Symposium on Hardware-Oriented Security and Trust (HOST)*, IEEE, July 2009, pages 1–6.

[51] Sergei P. Skorobogatov and Ross J. Anderson. Optical fault induction attacks. In *Proc. International Workshop on Cryptographic Hardware and Embedded Systems (CHES)*, Springer, February 2003, pages 2–12.

[52] Venu Babu Thati, Jens Vankeirsbilck, Jeroen Boydens, and Davy Pissort. Selective duplication and selective comparison for data flow error detection. In *Proc. International Conference on System Reliability and Safety (ICSRS)*, IEEE, November 2019, pages 10–15.

[53] Niek Timmers and Albert Spruyt. Bypassing secure boot using fault injection. *Black Hat Europe*, 2016.

[54] Aurélien Vasselle, Hughes Thiebeauld, Quentin Maouhoub, Adèle Morisset, and Sébastien. Ermeneux. Laser-induced fault injection on smartphone bypassing the secure boot. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Santa Barbara, CA, USA, August 2017, pages 41–48. IEEE.

[55] Muhammad Abdul Wahab. *Hardware support for the security analysis of embedded softwares: applications on information flow control and malware analysis*. PhD thesis, Centrale Supelec, Université Bretagne Loire, December 2018.

[56] Mario Werner, Erich Wenger, and Stefan Mangard. Protecting the control flow of embedded processors against fault attacks. In *Proc. International Conference on Smart Card Research and Advanced Applications (CARDIS)*, Springer, November 2015, pages 161–176.

[57] Asier Goikoetxea Yanci, Stephen Pickles, and Tughrul Arslan. Detecting voltage glitch attacks on secure devices. In *Proc. Symposium on Bio-inspired Learning and Intelligent Systems for Security (ECSIS)*, IEEE, August 2008, pages 75–80.

[58] Bilgiday Yuce, Nahid Farhady Ghalaty, Chinmay Deshpande, Harika Santapuri, Conor Patrick, Leyla Nazhandali, and Patrick Schaumont. Analyzing the fault injection sensitivity of secure embedded software. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(4) :95 :1–25, 2017.

[59] Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. Software fault resistance is futile: Effective single-glitch attacks. In *Proc. Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, Santa Barbara, CA, USA, August 2016, pages 47–58. IEEE.

[60] Bilgiday Yuce, Patrick Schaumont, and Marc Witteman. Fault attacks on secure embedded software: Threats, design, and evaluation. *Journal of Hardware and Systems Security*, 2 (2) :111–130, 2018.

[61] Loic Zussa, Jean-Max Dutertre, Jessy Clèdiere, Bruno Robisson, and Assia Tria. Investigation of timing constraints violation as a fault injection means. In *Proc. Design of Circuits and Integrated Systems (DCIS)*, 2012, pages 1–6.

COLLEGE DOCTORAL BRETAGNE | MATHSTIC BRETAGNE OCEANE

Université Bretagne Sud

Titre : Sécurisation matérielle de processeurs embarqués face aux attaques par injection de fautes

Mot clés : cybersécurité matérielle, processeur embarqué, protection contre des attaques physiques

Résumé : Les processeurs embarqués peuvent faire l'objet d'attaques physiques en raison de la proximité entre l'attaquant et le circuit. Les attaques par injection de fautes (FIA) exploitent des perturbations du circuit pour révéler des données secrètes ou contourner des dispositifs de sécurité. Il existe plusieurs méthodes de protection contre les FIA : correction/détection d'erreurs, vérification des propriétés fonctionnelles, redondance, randomisation, etc. En logiciel, la duplication et la triplication d'instructions sont faciles à utiliser pour sécuriser des codes critiques mais entraînent des surcoûts importants en temps d'exécution et taille de code. De plus, les protections logicielles prennent rarement en compte les détails d'implémentation du matériel, comme le pipeline du processeur, et peuvent ne pas être aussi efficaces que prévu. Nous proposons un support matériel pour le rejeu d'instructions dans un processeur RISC élémentaire. Il consiste en une petite extension du jeu d'instructions (une nouvelle instruction), quelques éléments de détection et de vote (principalement des registres et des comparateurs) et de légères modifications du contrôle du processeur. Nous explorons différentes configurations d'éléments de protection internes pour le rejeu matériel. Le surcoût en surface du cœur est de 30% et la baisse de fréquence d'horloge de 10% sur FPGA. Le rejeu matériel réduit de manière significative le temps d'exécution et la taille du code par rapport à une protection purement logicielle.

Title: Hardware Security of Embedded Processors against Fault Injection Attacks

Keywords: hardware cybersecurity, embedded processor, protection against physical attacks

Abstract: Embedded processors can be subject to physical attacks due to some proximity between an attacker and the circuit. Fault injection attacks (FIAs) exploit perturbations in the circuit to reveal secret data or bypass security features. Various protection methods exist against FIA: error code correction/detection, functional property check, redundancy, randomization, etc. In software, instruction duplication and triplication are easy to use to secure critical codes but lead to important overheads in execution time and code size. More, software protections rarely take into account hardware implementation details, such as the processor pipeline, and may not be as effective as intended. We propose a hardware support for instructions replay in a basic RISC processor. It consists in a small extension of the instruction set (one new instruction), a few detection and vote elements (mainly registers and comparators), and light modifications of the processor control. We explore various configurations of internal protection elements for hardware replay. The core area overhead is about +30% while the clock frequency is reduced by less than 10% on FPGA. The hardware replay allows to significantly reduce the execution time and code size compared to a pure software replay protection.