TrustSoC : Proposition d'une architecture de SoC hétérogène sécurisée à la conception

Raphaële Milan

N° d'ordre NNT : 2024STET0058

THÈSE de DOCTORAT
DE L'UNIVERSITÉ JEAN MONNET SAINT-ÉTIENNE

Membre de l'Université de Lyon

Ecole Doctorale N°488
SIS - Sciences Ingénierie Santé

Spécialité de doctorat : Microélectronique

Soutenue publiquement le 9 décembre 2024, par :

Raphaële MILAN

TrustSoC : Proposition d'une architecture de SoC hétérogène sécurisée à la conception

---

Devant le jury composé de :

Cedric KILLIAN, Professeur des universités, Université Jean Monnet Saint Etienne	Président
Bertand LE GAL, Maître de conférences HDR, Université de Rennes	Rapporteur
David HELY, Professeur des universités, INGP	Rapporteur
Maria MENDEZ-REAL, Chaire de professeur junior, Université de Bretagne Sud	Examinatrice
Laurent BOHER, Chargé de recherche, DGA	Examinateur
Lilian BOSSUET, Professeur des universités, Université Jean Monnet Saint-Etienne	Directeur de thèse
Loïc LAGADEC, Professeur des universités, ENSTA Bretagne	Co-directeur de thèse

Remerciements

Mes premiers remerciements vont à ma direction de thèse, **Loïc et Lilian**, pour m'avoir fait confiance et m'avoir donné l'opportunité de faire celle-ci. Merci tout particulièrement à **Lilian**. Merci pour ta bienveillance, merci pour tous tes conseils avisés durant cette thèse, merci pour ton écoute et ta patience.

Je remercie bien évidemment **le reste de l'équipe SESAM**. Les permanents tout d'abord puis les doctorants. Merci à tous d'avoir fait goûteurs pour les spécialités de ma future boulangerie "*Chez Raphou*". Merci spécialement à **Justine, Simon, Nicolas et William** pour toutes les discussions que nous avons pu avoir et pour toutes ces soirées. Vous avez vraiment embelli cette dernière année de thèse.

Un grand merci à **Natacha**, ma copine de bureau, qui a illuminé ces derniers mois de thèse par sa présence. Merci pour tous ces moments. Merci aussi de t'être déplacée de Bordeaux pour assister à ma soutenance et d'avoir encouragé mon oreille en gruyère. Merci à mon ancien collègue, **le docteur Arturo Mollinedo Garay** pour ta bonne humeur contagieuse et ta positivité. Merci d'être venu à ma soutenance ! **Au docteur Mateus Simoes**, merci pour tout.

On dit qu'une rencontre peut changer une vie et c'est vrai. À **Madame H.** qui a littéralement et véritablement changé ma vie. Merci du fond du coeur pour tout.

À **Sacha**, merci d'avoir été là depuis maintenant un bon bout de temps (14 ans ça commence à faire !). Un grand merci pour tout.

À **mon Vito**, sans qui je n'aurais pas eu autant la classe à ma soutenance de thèse, merci pour tout !

Un grand grand merci à **ma famille**, sans qui je n'en serais jamais arrivée là. Tout d'abord à **mon oncle** et **mon grand-père** d'avoir été là pour moi durant toutes ces années. Merci à **ma maman**, tu es une maman en or. Merci d'avoir toujours été là pour moi, d'avoir toujours pris soin de moi et de m'avoir supportée il faut bien le dire aussi. Merci à **mon papa**, le héros de ma vie. Merci pour tout. J'espère que vous êtes aussi fiers de moi que je suis fière d'avoir des parents comme vous.

Je remercie tout particulièrement **mon grand frère**, mon yang, je t'aime par dessus tout et plus que tout. Je suis extrêmement fière de toi. Merci d'avoir toujours été là, d'avoir partagé ces derniers mois de thèse avec moi. Pour paraphraser une grande personne "tu es et resteras la personne la plus importante à mes yeux".

À **Monsieur Antoine**, je te remercie pour tout. Merci de m'avoir toujours soutenue et d'avoir toujours cru en moi. Merci d'être toujours là pour moi dans les bons comme les mauvais moments. J'espère que tu continueras encore pendant longtemps à partager ma vie.

Et bien évidemment, il ne peut pas avoir de remerciements dans cette thèse sans parler de LA personne : le merveilleux (mais pas tous les jours) **Monsieur Berlioz**, le petit MONSTRE qui partage ma vie. À mon Berlioz, merci d'être entré dans ma vie il y a

maintenant bientôt six ans et d'avoir tout chamboulé. **Tu es le GRAND GRAND amour de ma vie et cette thèse est pour toi.**

Résumé

Depuis les années 70, les systèmes sur une puce (SoC) n'ont cessé de croître en complexité. Les fabricants intègrent de plus en plus de composants hétérogènes sur une seule puce de silicium pour améliorer les performances. Parmi ces composants, se trouvent des ressources matérielles logiques, des composants analogiques, et bien d'autres. Cette intégration rend les SoC polyvalents, adaptés à divers domaines tels que la téléphonie mobile, l'informatique, le secteur militaire ou le Cloud. Cependant, cette polyvalence s'accompagne d'une complexité croissante qui soulève des enjeux de sécurité, notamment parce que ces SoC traitent des données sensibles (comme des informations personnelles ou des systèmes critiques tels que les voitures autonomes), ce qui en fait des cibles privilégiées pour des attaques. Leurs vulnérabilités résultent souvent d'erreurs apparues lors de la conception, où l'accent est mis sur les performances au détriment de la sécurité. Celle-ci est souvent reléguée à une phase ultérieure. Cette thèse, menée au laboratoire Hubert Curien, vise à changer cette approche en intégrant la sécurité dès le début de la conception des systèmes sur une puce hétérogènes. Son objectif est de proposer une architecture de SoC hétérogène sécurisée dès sa conception.

En premier lieu, une première architecture sécurisée par conception, appelée *TrustSoC*, est proposée. Cette architecture est basée sur un système multi-mondes qui est une extension de la technologie ARM *TrustZone* à l'ensemble du SoC (y compris les parties matérielles) et des règles de fonctionnement précises. La sécurité apportée par *TrustSoC* est centrée autour du système de communication.

Dans un deuxième temps, après avoir mis en lumière les limites de *TrustSoC*, une deuxième architecture est présentée. Il s'agit d'une extension de la première et est appelée *RTrustSoC*. Elle permet d'introduire, pour la première fois, le concept d'IP matérielle de confiance. *RTrustSoC* se base notamment sur un systèmes de pénalités dynamiques inspiré des architectures zéro confiance. La sécurité supplémentaire apportée par *RTrustSoC* est démontrée au travers d'applications de scénarios d'attaques.

Troisièmement, l'architecture *TrustSoC-V* est proposée pour surmonter les limitations imposées par les technologies propriétaires. Basée sur des cœurs RISC-V, elle introduit un nouveau bus de communication open source, appelé *XSecure*, ainsi qu'une méthodologie d'implémentation nommée RISC-B. Celle-ci permet de faciliter l'implémentation matérielle de *TrustSoC-V* pour un concepteur.

Enfin, pour élever le niveau d'abstraction et faciliter l'exploration de l'espace de conception, une modélisation de la conception de l'architecture *TrustSoC* est introduite, appelée *TrustSoC-M*. Elle est réalisée grâce à une approche d'ingénierie système basée sur des modèles.

Abstract

Since the 70s, systems on a chip have become increasingly complex. On a same silicon chip, manufacturers integrate always more heterogeneous components to improve system performance. Among these components, there are logic hardware resources, analogic components, and others. The addition of these components makes SoCs highly versatile, but also very complex. Their multi-purpose facilitates their uses in various different domains such as mobile telephony, informatics, military or the Cloud. SoCs handle personal data (contacts, health information, credit card, etc.) and also critical systems (autonomous car, etc.). Therefore, this raises questions about their safety. SoCs represent a primary target for attackers seeking to steal critical information or inflict damage to the system. Vulnerabilities inadvertently introduced during the design phase enable these attacks. Their vulnerabilities are often related to enhancements made to increase system performance. Because security is an afterthought in architecture design, it cannot provide an adequate defense against all attacks. The motivation of this thesis, conducted within the Hubert Curien laboratory, is to reverse this historical trend of designing integrated circuits to include security from the beginning. The goal is to provide a heterogeneous, design-secure SoC architecture.

The initial proposal is for a secure-by-design architecture, called *TrustSoC*. It is based on a multi-world system that extends ARM's *TrustZone* technology to the whole SoC (including hardware parts) and precise operating rules. The security provided by *TrustSoC* is centered around the communication system.

Secondly, following the observed limitations with the TrustSoC approach, a second architecture is presented. This is an extension of the first, and it is called *RTrustSoC*. It introduces, for the first time, the concept of trusted hardware IP. *RTrustSoC* is based on a dynamic penalty system, which has been inspired by zero-trust architectures. The enhanced security of *RTrustSoC* is illustrated through applied scenarios of potential attacks.

Thirdly, the *TrustSoC-V* architecture has been developed to address the limitations imposed by proprietary technologies. Based on RISC-V cores, it introduces a new open-source communication bus, designated as XSecure, in addition to an implementation methodology, RISC-B. The latter facilitates the hardware implementation of *TrustSoC-V* for designers.

Finally, in order to enhance the level of abstraction and facilitate the exploration of the design space, a design model of the *TrustSoC* architecture, designated as *TrustSoC-M*, is introduced. This is achieved through the implementation of a model-based systems engineering approach.

Table des matières

Remerciements **iii**

Résumé **v**

Abstract **vii**

Table des matières **xi**

Liste des figures **xv**

Liste des tableaux **xvii**

Liste des sigles **xix**

1 Introduction **1**
1.1 Contexte des travaux de la thèse 2
1.2 Les SoC hétérogènes 2
1.3 La sécurité des SoC hétérogènes 5
1.3.1 Attaques visant les mémoires 6
1.3.2 Attaques visant la micro-architecture 10
1.3.3 Attaques visant la sécurité des communications dans le SoC . . 10
1.3.4 Attaques visant les systèmes de gestion de l'énergie des SoC . . 11
1.3.5 Attaques internes visant les traitements d'accélérations matérielles et canal caché 12
1.4 Motivations et objectifs de la thèse 13
1.5 Organisation du manuscrit 13

2 État de l'art **15**
2.1 Introduction au chapitre 16
2.2 Notions préliminaires 16
2.2.1 Les environnements d'exécution de confiance dans les SoC . . . 16
2.2.2 Les protocoles de bus de communication dans les SoC : AMBA-AXI et TileLink 17
2.3 Architectures sécurisées de SoC hétérogène issues de l'industrie 19
2.3.1 La technologie ARM *TrustZone* 19
2.3.2 WorldGuard de SiFive 22
2.4 Architectures sécurisées de SoC hétérogène issues de la littérature . . . 23
2.4.1 HECTOR-V 24
2.4.2 CURE (*CUstomizable and Resilient Enclaves*) 25
2.4.3 SANCTUARY 28
2.4.4 TEEOD (*Trusted Execution Environments On-Demand*) 30
2.4.5 Une approche décentralisée basée sur des pare-feu matériels . . 32
2.4.6 E-IIPS (*Extended Infrastructure IP for SoC security*) 34

2.4.7 ProMiSE (*PROgrammable hardware Monitor for Secure Execution*) . . . 35
2.4.8 *Pro-active policing and policy enforcement architecture for securing MPSoCs* . . . 37
2.5 Comparaisons et synthèse . . . 43
2.6 Positionnement des contributions de cette thèse vis à vis de l'état de l'art . . . 45
2.7 Conclusion du chapitre . . . 45

3 *TrustSoC* et son extension *RTrustSoC* 47
3.1 Introduction au chapitre . . . 48
3.2 Motivations . . . 48
3.3 *TrustSoC* . . . 48
3.3.1 Modèle de menaces . . . 49
3.3.2 Définition des exigences de sécurité . . . 49
3.3.3 Règles de sécurité et architecture . . . 50
3.3.4 Fonctionnement . . . 56
3.3.5 Résultats d'implémentation . . . 59
3.3.6 Limites . . . 62
3.4 *RTrustSoC* . . . 62
3.4.1 Modèle de menaces . . . 62
3.4.2 Définition des exigences de sécurité . . . 63
3.4.3 Architecture et fonctionnement . . . 64
3.4.4 Résultats d'implémentation . . . 69
3.5 Scénarios d'utilisation de *RTrustSoC* . . . 71
3.5.1 Attaque basée sur la mesure des temps d'accès à la mémoire cache 71
3.5.2 Attaque par escalade de privilèges via le port ACP . . . 72
3.5.3 Attaque du démarrage sécurisé via le port ACP . . . 74
3.6 Limitations de la proposition . . . 76
3.7 Conclusion du chapitre . . . 77

4 *TrustSoC-V* et sa mise en œuvre par RISC-B 79
4.1 Introduction au chapitre . . . 80
4.2 Motivations . . . 80
4.3 *TrustSoC-V* . . . 80
4.3.1 Modèle de menaces . . . 81
4.3.2 Définition des exigences de sécurité . . . 81
4.3.3 Architecture *TrustSoC-V* . . . 83
4.3.4 Fonctionnement . . . 88
4.3.5 Résultats d'implémentation . . . 90
4.4 RISC-B . . . 95
4.4.1 Objectifs et motivations de RISC-B . . . 95
4.4.2 Méthodologie . . . 95
4.4.3 Résultats d'implémentation . . . 96
4.5 Limitations de la proposition . . . 98
4.6 Conclusion du chapitre . . . 98

5 *TrustSoC-M* 101
5.1 Introduction au chapitre . . . 102
5.2 Motivations . . . 103
5.3 *TrustSoC-M* . . . 103

5.3.1 Modèle de menaces considéré . . . 103
5.3.2 Exigences de sécurité . . . 104
5.3.3 Définition du modèle . . . 104
5.3.4 Mise en place de *TrustSoC-M* . . . 109
5.4 Perspective pour *TrustSoC-M* . . . 116
5.5 Conclusion du chapitre . . . 116

6 Conclusion et perspectives **117**
6.1 Conclusion . . . 117
6.2 Résumé des contributions . . . 118
6.3 Perspectives . . . 119
6.3.1 *TrustSoC* et *RTrustSoC* . . . 119
6.3.2 *TrustSoC-V* . . . 119
6.3.3 *TrustSoC-M* . . . 120

7 Communications et publications **121**
7.1 Publication dans un journal international . . . 121
7.2 Conférence internationale avec comité de lecture . . . 121
7.3 Conférence nationale avec comité de lecture . . . 121
7.4 Présentation à un congrès international sans acte . . . 121
7.5 Posters . . . 122

Bibliographie **123**

Annexe **131**

A Règles de sécurité de *TrustSoC* **131**

Liste des figures

1.1 Exemple d'architecture de SoC hétérogène. 3
1.2 Graphique présentant l'évolution des SoC hétérogènes des deux plus grands fabricants AMD et Intel (LE signifie élément logique). 4
1.3 Exemple d'architecture de bus dans un SoC hétérogène. 4
1.4 Schéma des emplacements d'attaques possibles sur un SOC hétérogène au niveau logiciel et matériel. 5
1.5 Attaque Rowhammer visant la mémoire vive dynamique (DRAM). . . 6
1.6 Déroulement attaque *Evict + Time* sur la mémoire cache. 7
1.7 Déroulement attaque *Prime + Probe* sur la mémoire cache. 7
1.8 Déroulement attaque *Flush + Reload* sur la mémoire cache. 8
1.9 Déroulement attaque *Flush + Flush* sur la mémoire cache. 8
1.10 Attaques sur la mémoire cache étendues depuis la logique programmable du SoC hétérogène. 9
1.11 Modification de la valeur du signal de sécurité entre une interface maître (M) et une interface esclave (E) à l'aide d'un cheval de Troie matériel pour un accélérateur matériel. 11
1.12 Utilisation du système de gestion de l'énergie du SoC (DVFS) pour réaliser des attaques contre le système. 12

2.1 Architecture matérielle et logicielle de l'implémentation d'un TEE. . . 16
2.2 Représentation du protocole AXI avec les cinq canaux de communication. 18
2.3 Architecture logicielle et matérielle de la technologie *TrustZone* appliquée à une architecture de SoC hétérogène. 20
2.4 Architecture matérielle de la proposition WorldGuard de SiFive. 22
2.5 Architecture matérielle de la solution HECTOR-V. 24
2.6 Architecture matérielle et logicielle de la solution CURE. 27
2.7 Architecture logicielle et matérielle de la solution SANCTUARY. . . . 29
2.8 Architecture logicielle et matérielle de la solution TEEOD. 31
2.9 Architecture matérielle de l'approche décentralisée basée sur des pare-feu matériels pour sécuriser des architectures hétérogènes multiprocesseurs (MPSoC-FPGA). 33
2.10 Architecture matérielle de la solution E-IIPS. 34
2.11 Architecture matérielle de la solution ProMiSE. 36
2.12 Architecture matérielle de la solution *pro-active policing and policy enforcement* appliquée à une plateforme MPSoC Zynq UltraScale+ d'AMD. 37
2.13 Les différentes architectures de contrôleurs de communication issues de la littérature. 44

3.1 Architecture du concept *TrustSoC*. 52
3.2 Architecture des contrôleurs de communication connectés à une interface de communication esclave de *TrustSoC*. 53

3.3 Construction d'une trame des signaux utilisateur dans le cadre de *TrustSoC* où *nb_m* représente le nombre de mondes et *nb_id* le nombre de composants dans le système. 54
3.4 Diagramme de séquence du fonctionnement des contrôleurs de communication de *TrustSoC*. 55
3.5 Fonctionnement de l'architecture *TrustSoC* ainsi que les ressources accessibles du système dans différents mondes. 57
3.7 Résultats d'implémentation en LUT, FF et fréquence maximum atteignable d'un contrôleur de communication de *TrustSoC* pour cinq IP matérielles différentes pour une petite configuration sur un SoC-FPGA AMD Zynq-7000. 60
3.8 Évaluation de la latence induite par *TrustSoC* lors d'une communication. 61
3.9 Architecture contrôleur de communication d'une mémoire BRAM. . . 61
3.10 Architecture *RTrustSoC*. 65
3.11 Architecture des contrôleurs de communication connectés à une interface de communication maître de *RTrustSoC*. 66
3.12 Diagramme de séquence de fonctionnement des contrôleurs de communication de *RTrustSoC*. 68
3.14 Résultats d'implémentation du système de pénalités de *RTrustSoC* pour une IP matérielle avec un contrôleur de communication sur un SoC-FPGA AMD Zynq-7000. (Unp. pour non protégée, FPT pour table de permissions fixes, RPT pour table de permissions reconfigurable, FPS pour système de pénalités fixes avec *Max* et *Tblock* fixes, DPS pour système de pénalités dynamiques et *Max* et *Tblock* fixes, RDPS pour système de pénalités dynamiques et paramètres *Max* et *Tblock* reconfigurables). 70
3.15 Scénario d'attaque de [17] et son application à *RTrustSoC*. 71
3.16 Scénario d'attaque de [40] et son application à *RTrustSoC*. 73
3.17 Scénario d'attaque de [40] et son application à *RTrustSoC*. 75
3.18 Limite lors de l'implémentation de *RTrustSoC*. 76

4.1 Architecture du concept *TrustSoC-V*. 83
4.2 Exemple de réalisation du bus de communication sécurisé *XSecure* proposé par *TrustSoC-V* entre deux IP matérielles. 84
4.3 Architectures des contrôleurs de communication "moniteur" de *TrustSoC-V* pour une interfaces maître (A) et une esclave (B). 86
4.4 Diagramme de séquence du fonctionnement des contrôleurs de communication "moniteurs" de *TrustSoC-V*. 87
4.5 Fonctionnement de *TrustSoC-V* dans le monde non sécurisé (A) et dans un des mondes sécurisés (B). 89
4.6 Architecture matérielle CV32A6. 91
4.7 Architecture matérielle modifiée CV32A6 pour ajouter *TrustSoC-V*. . . 92
4.8 Résultats d'implémentation en LUT, FF et fréquence maximum atteignable d'un contrôleur de communication de *TrustSoC-V* pour cinq IP matérielles différentes pour une petite configuration sur un SoC-FPGA AMD Zynq-7000. 94
4.9 Étapes numérotées de 1 à 4 de la méthodologie de *RISC-B*. 96
4.10 Architecture matérielle prototype RISC-B obtenu sur le logiciel de CAO Vivado. 97

5.1 Représentation du processus de développement par approche MBSE. . 102

5.2 Représentation des différentes étapes détaillées de la mise en place de *TrustSoC-M* basée sur une approche MBSE. 105
5.3 Diagramme de classes en langage UML représentant le modèle exécutable *TrustSoC*. 107
5.4 Diagramme de séquence pour une opération d'écriture. 108
5.5 Exemple d'architecture d'une instance du modèle. 110
5.6 Table permission de la mémoire RAM d'une instance du modèle donnée dans la Figure 5.5. 111
5.7 Représentation des étapes du scénario (donné dans le Listing 5.3) notées de 1 à 4 sur l'architecture matérielle. 112
5.8 Résultats du scénario présenté dans le Listing 5.3. 113
5.9 Aperçu de la solution de génération proposée par l'outil *DiWall*. 114
5.10 Interface graphique utilisateur de l'outil *DiWall* avec un exemple d'architecture de SoC hétérogène. 115

A.1 Architecture sans protection considérée pour les règles. 131

Liste des tableaux

2.1 Comparaison des architectures sécurisées de SoC présentées dans les parties ci-dessus (Sections 2.3 et 2.4 entières). Les symboles indiquent : ○ pas supporté, ◐ supporté de manière limitée, et ● supporté entièrement. 40

3.1 Exemples de règles de sécurité de *TrustSoC* encadrant le fonctionnement d'un SoC-FPGA. 51
3.2 Résultats d'implémentation en nombre de LUT d'un contrôleur de communication lié à une BRAM en LUT (AMD Zynq-7000 SoC-FPGA). . 61
3.3 Résultats d'implémentation pour le scénario proposé dans la Figure 3.15 sur un SoC-FPGA AMD Zynq-7000 (XC7Z010-1CLG400C). . . . 72
3.4 Résultats d'implémentation pour le contrôleur de communication de la mémoire proposé dans la Figure 3.16 (B) sur un SoC-FPGA AMD Zynq-7000 (XC7Z010-1CLG400C) et pour un système à un monde sécurisé. 74

4.1 Résultats implémentation pour la version de base [83] et la version RISC-B en fonction des ressources LUT, FF et BRAM sur ZCU104. . 98

Liste des sigles

ACAP *Adaptive Computer Accelerator Platform*. 3
ACP *Accelerator Coherency Port*. 8, 21, 30, 69, 71–75, 133
AES *Advanced Encryption Standard*. 9, 13, 32, 59, 63, 71, 75, 95
AID Agence de l'Innovation Défense. 13
AMBA *Advanced Microcontroller Bus Architecture*. 4, 16, 17, 22, 27, 45, 52, 54, 59, 64, 84, 93, 119
AS Application SANCTUARY. 28
ASIC *Application-Specific Integrated Circuit*. 2
ATE *Anti-Tamper Engine*. 38, 39
AXI *Advanced eXtensible Interface*. 4, 16, 17, 19, 22, 24, 25, 27, 38, 45, 52–55, 59, 64, 66, 76, 82, 84, 93, 113, 114, 119

BBRAM *Battery-Backed Random Access Memory*. 75
BRAM *Block Random Memory Access*. 61

CAO Conception Assistée par Ordinateur. 11, 21, 59, 61, 63, 69, 81, 95, 96, 98, 103, 113
CLINT *Core Local Interrupt Controller*. 93
CPU *Central Processing Unit*. 13, 27, 32, 111, 112, 132–134
CURE *CUstomizable and Resilient Enclaves*. 25–28

DMA *Direct Access Memory*. 9, 19, 26, 27, 40, 44
DoS *Denial-of-Service*. 11, 23, 25, 26, 28, 32, 39–43, 45, 49, 62, 63, 81, 88
DRAM *Dynamic Random Access Memory*. 2, 6, 93
DVFS *Dynamic Voltage and Frequency Scaling*. 11, 12, 21, 30

E-IIPS *Extended Infrastructure IP for SoC security*. 34, 35
ECC *Elliptic Curve Cryptography*. 32
ES Exigences de Sécurité. 26
ESWEEK *Embedded Systems WEEK*. 80

FF *Flip Flop*. 59, 69, 72, 74, 95, 96
FPGA *Field-Programmable Gate Array* ou *Programmable Logic*. 2, 3, 12, 13, 17, 21, 28, 30, 32, 34, 37, 44, 45, 48–53, 56, 58, 63, 64, 67, 69, 71–74, 77, 81–83, 88, 93, 95, 98, 103, 104, 116, 118, 120, 132–134
FSBL *First Stage BootLoader*. 22

GPIO *General Purpose Input/Output*. 93
GPU *Graphical Processing Unit*. 2

HLS *High-Level Synthesis*. 102

IP *Intellectual Property*. xiv, 10, 11, 19, 21, 30, 33–35, 49–56, 58–68, 76, 77, 79, 81–86, 88, 90, 93–96, 98, 103–106, 110–114, 116, 118, 120, 132–134
ISA *Instruction Set Architecture*. 80

LUT *Look-Up Table*. 59, 62, 69, 72, 74, 95, 96

MARTRE *Modeling and Analysis of Real-Time and Embedded systems*. 102
MBSE *Model-Based System Engineering*. 14, 101–106, 116, 119
MPSoC *MultiProcessor System-On-a-Chip*. 22, 32, 37–39

NoC *Network-On-a-Chip*. 3
NS bit *Non-Secure bit*. 21

PLIC *Platform Level Interrupt Controller*. 93
ProMiSE *PROgrammable hardware Monitor for Secure Execution*. 35, 36, 44, 45, 48
PS *Processing System*. 2, 48, 50, 52, 64, 76, 83, 93

QSPI *Quad Serial Peripheral Interface*. 93

RAM *Random Access Memory*. 2, 6, 53, 64, 110–112
REE *Rich Execution Environment*. 17, 24, 30, 40, 41, 73, 74
RNG *Random Number Generator*. 32
ROM *Read-Only Memory*. 2, 6, 22
RoP *Return-oriented Programming*. 36, 41
RPU *Real-time Processing Unit*. 38, 39, 41
RSA Rivest Shamir Adleman. 13, 32, 74
RSP *Rapid System Prototyping*. 103
RVSCP *RISC-V Secure Co-Processor*. 24

SCA *Side-Channel Analysis*. 12, 27, 41, 42, 45, 50, 58, 63, 64, 81, 82, 90, 103, 104
SCK *bus Sanity ChecKer*. 38
SCR *Secure Configuration Register*. 21
SCU *Snoop Control Unit*. 72
SM *Security Monitor*. 21, 25, 27, 40
SMC *Security Monitor Call*. 21
SoC *System-On-a-Chip*. 1–6, 8, 10–13, 15–17, 19–21, 23, 24, 28, 30, 34, 35, 38, 41, 44, 45, 47–50, 52, 53, 58, 62–65, 67, 69, 71, 72, 74, 77, 79–85, 88, 98, 102–104, 108, 116–120, 131, 132, 134
SPC *Security Policy Controller*. 35
SPE *Security Policy Engine*. 38
SRAM *Static Random Access Memory*. 93
SRE *Security Response Engine*. 38, 39

TA *Trusted Application*. 21
TCB *Trusted Computing Base*. 25, 26, 40
TDC *Time-to-Digital Converter*. 12
TEE *Trusted Execution Environment*. 16, 17, 19, 20, 24, 25, 28, 30, 31, 40, 41, 45, 48, 56, 73, 74, 88

TEEOD *Trusted Execution Environments On-Demand*. 30–32, 34, 43
TEM *Threat Estimation Models*. 36, 37
TOCTOU *Time-Of-Check Time Of Use*. 35
TPM *Trusted Platform Module*. 38

UART *Universal Asynchronous Receiver Transmitter*. 93
UML *Unified Modeling Language*. 105, 106, 108

Wg *WorldGuard*. 22, 23, 28, 40, 43, 45, 48, 117
WID *World Identifier*. 22, 23

ZSBL *Zero Stage BootLoader*. 22
ZTA *Zero Trust Architecture*. 35, 36, 118

Chapitre 1

Introduction

Résumé

Ce chapitre commence par présenter un bref historique des systèmes sur une puce (SoC), leurs apparitions, leurs intérêts et leurs évolutions. Il présente également les nouveaux défis en matière de sécurité qui sont apparus ces dernières années avec l'augmentation de leurs complexités. Ce chapitre offre une brève introduction à la sécurité et présente les principales attaques, matérielles et logicielles, utilisées dans le cadre de cette thèse. Il décrit également les raisons qui ont motivé ces travaux et en donne les objectifs. Enfin, il détaille les grandes lignes de ce manuscrit.

Table des matières

1.1 Contexte des travaux de la thèse 2
1.2 Les SoC hétérogènes 2
1.3 La sécurité des SoC hétérogènes 5
1.3.1 Attaques visant les mémoires 6
1.3.2 Attaques visant la micro-architecture 10
1.3.3 Attaques visant la sécurité des communications dans le SoC . . 10
1.3.4 Attaques visant les systèmes de gestion de l'énergie des SoC . . 11
1.3.5 Attaques internes visant les traitements d'accélérations matérielles et canal caché 12
1.4 Motivations et objectifs de la thèse 13
1.5 Organisation du manuscrit 13

1.1 Contexte des travaux de la thèse

Le premier transistor a été conçu en 1947 et cette découverte a mené en 1958, à l'invention du premier circuit intégré. C'est en 1974, dans un souci de miniaturisation que Peter Stoll a proposé le premier système sur puce. Le terme système sur puce vient de l'anglais *System-On-a-Chip* (SoC) qui signifie que tous les composants du système sont intégrés sur une même puce de silicium. Cette innovation a été proposée pour une montre digitale qui était composée jusqu'alors de quarante-quatre circuits intégrés reliés ensemble à l'aide de quatre mille connexions. Cette nouvelle façon de concevoir les systèmes a eu pour conséquence de réduire drastiquement le coût du produit final puisque sa taille a fortement diminué. Elle a aussi permis une augmentation des performances car il est possible d'intégrer plus de composants sur une même puce de silicium tout en réduisant les interconnexions. Depuis ces années, pour améliorer les performances des systèmes, les fabricants de SoC ajoutent toujours plus de composants hétérogènes : radiofréquence, ressources matérielles programmables, composants analogiques, etc. Ces systèmes sont donc polyvalents, mais ils en deviennent extrêmement complexes. De nos jours, les systèmes sur puce font partie intégrante de notre quotidien, ils sont omniprésents dans beaucoup de domaines : téléphonie mobile, informatique, militaire, Cloud, etc. Ils gèrent des données personnelles (contacts, données de santé, carte bancaire, etc.) et aussi des systèmes dits sensibles (voiture autonome, etc.), ce qui pose la question de la sécurité. Ils représentent une cible de choix pour des attaquants. Ces derniers peuvent avoir pour but d'observer le système pour voler de l'information sensible, de lui nuire en injectant des fautes ou de le rendre inutilisable.

Ces attaques sont rendues possibles par des vulnérabilités qui sont apparues inopinément durant la phase de conception. Celles-ci sont souvent liées aux améliorations apportées en vue d'augmenter les performances du système. Un exemple d'attaque rendue possible est SPECTRE [51] qui utilise l'exécution spéculative, une fonctionnalité introduite pour améliorer les performances.

1.2 Les SoC hétérogènes

L'architecture classique d'un SoC hétérogène est présentée dans la Figure 1.1. Pour illustrer au mieux ce qu'il est possible de trouver dans un SoC hétérogène, nous avons délibérément choisi de présenter une architecture où de nombreux composants sont intégrés. Ce n'est toutefois pas nécessairement le cas de tous les SoC hétérogènes. L'architecture de la Figure 1.1 compte plusieurs processeurs généralistes avec plusieurs cœurs (identifiés en bleu) qui eux-mêmes embarquent une ou plusieurs applications. Cette partie est appelée la partie PS (*Processing System*). Un SoC hétérogène peut disposer d'accélérateurs matériels, comme illustré dans la Figure 1.1 (identifiés en vert-jaune) : des ASIC (*Application-Specific Integrated Circuit*), des FPGA (*Field-Programmable Gate Array* ou *Programmable Logic* ou ressources logiques) ou des GPU (*Graphical Processing Unit*). Il peut également intégrer différents types de mémoire, la Figure 1.1 ne montre qu'une mémoire partagée, mais on peut également retrouver la mémoire cache propre au processeur, de la RAM (*Random Access Memory*), de la DRAM (*Dynamic Random Access Memory*), de la ROM (*Read-Only Memory*), etc. La mémoire cache est organisée en plusieurs niveaux et dans les processeurs actuels, le dernier niveau de cette mémoire peut être est partagé entre les processeurs. Un SoC hétérogène peut également disposer d'interfaces (identifiées en orange dans la Figure 1.1), de systèmes de gestion de la puissance ou d'autres périphériques. Tous

ces composants sont ensuite interconnectés entre eux avec de la logique de bus (bleu foncé 1.1).

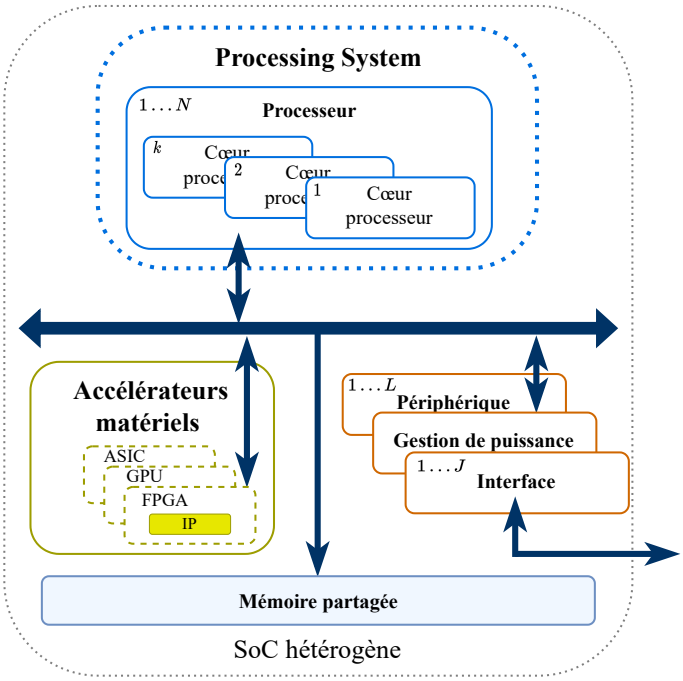


FIGURE 1.1 – Exemple d'architecture de SoC hétérogène.

Depuis quelques années, il y a un intérêt croissant pour ces systèmes sur puce et plus particulièrement pour ceux embarquant un FPGA : les SoC-FPGA. Ils présentent de nombreux avantages en termes de performances. En effet, l'utilisation d'un FPGA permet d'intégrer des conceptions matérielles post-production qui peuvent accélérer l'exécution. Le FPGA est reconfigurable en totalité avant l'exécution et peut même l'être partiellement pendant. Cette fonctionnalité peut permettre de répondre à des besoins spécifiques d'une application ou de mettre à jour un accélérateur matériel.

Deux principaux fabricants de SoC-FPGA sont à distinguer : Intel (anciennement Altera) et AMD (anciennement Xilinx). La Figure 1.2 décrit l'évolution des SoC hétérogènes disponibles sur le marché depuis 2012 par ces deux principaux fabricants (AMD et Intel).

Comme l'illustre cette Figure, en 2012, il était proposé des systèmes sur puce avec un seul processeur à deux cœurs et un FPGA ayant des centaines de milliers d'éléments logiques. Cinq ans plus tard, les SoC embarquaient un FPGA où le nombre d'éléments logiques avait été multiplié par dix et plusieurs processeurs multicœurs. En 2020, les premiers systèmes sur puce multi-processeurs dédiés au domaine des traitements radiofréquences sont apparus. Depuis 2021, AMD a proposé la gamme de produits VERSAL qui sont des ACAP (*Adaptive Computer Accelerator Platform*) pour des applications d'intelligence artificielle. Ces circuits intègrent des moteurs d'intelligence artificielle et de traitement du signal. Ils proposent toujours un processeur applicatif multicœur (Cortex A72) pour le domaine de la radiofréquence. De plus, ces types de circuits ont la possibilité d'intégrer directement des NoC (*Network-On-a-Chip*) dans la plateforme. Même si cela est réalisable, la plupart des SoC hétérogènes présents sur le marché embarquent de la logique de bus. Ce bus est un élément central de l'architecture. Il permet d'interconnecter tous les composants, logiciels et matériels (mémoires, processeurs, périphériques, logique matérielle, etc.), entre eux. Le bus fait

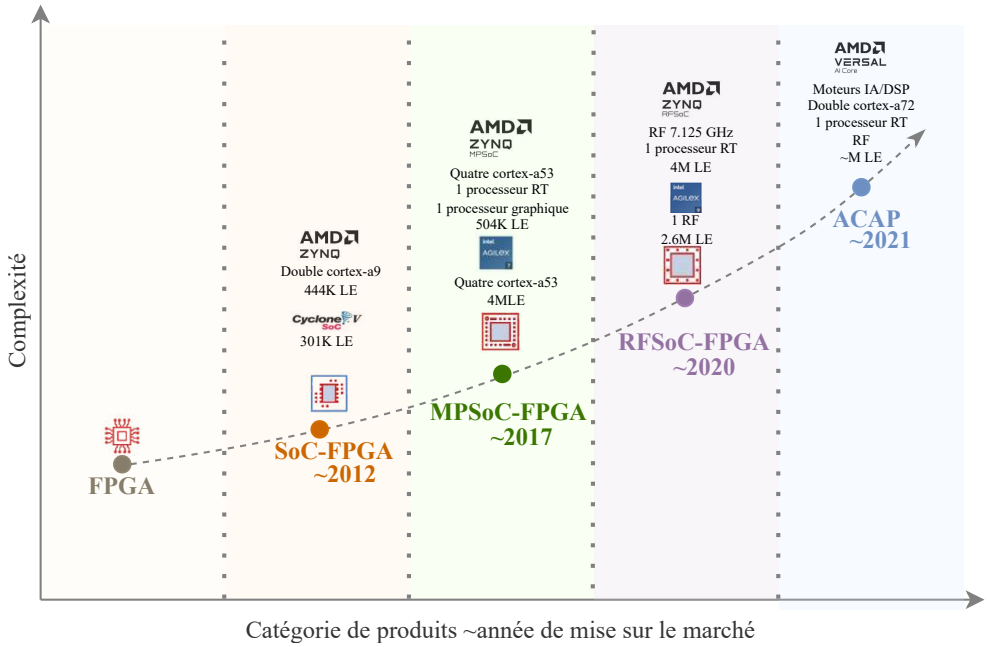


FIGURE 1.2 – Graphique présentant l'évolution des SoC hétérogènes des deux plus grands fabricants AMD et Intel (LE signifie élément logique).

le lien entre les différentes interfaces maîtres et esclaves des composants du système. Chaque composant peut posséder une ou plusieurs interfaces. Un exemple d'une telle architecture est représenté dans la Figure 1.3. Seules les interfaces de type maître sont

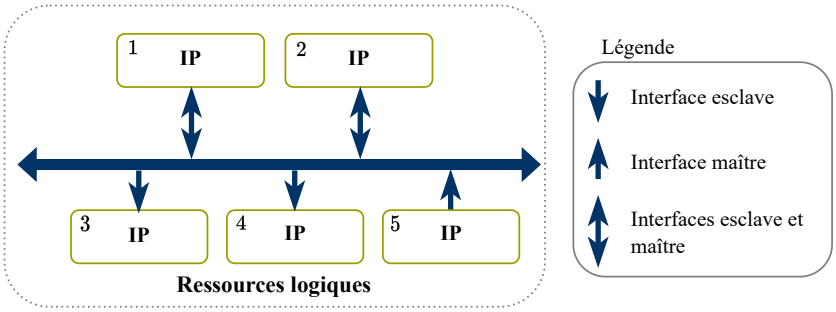


FIGURE 1.3 – Exemple d'architecture de bus dans un SoC hétérogène.

autorisées à faire des requêtes. Les interfaces esclaves traitent les requêtes qui leur sont adressées. Un SoC hétérogène peut utiliser plusieurs types de bus. Un exemple est le bus AXI (*Advanced eXtensible Interface*) [8] de la norme AMBA (*Advanced Microcontroller Bus Architecture*) qui est utilisé dans les SoC d'Intel et d'AMD.

1.3 La sécurité des SoC hétérogènes

Au moment de la conception, les concepteurs ont fait des choix pour améliorer les performances des SoC hétérogènes. Ces choix peuvent toutefois introduire des vulnérabilités, comme l'indiquent les résultats présentés dans [71]. Celles-ci peuvent être ensuite utilisées par une entité malicieuse pour attaquer le circuit. Ces attaques ne détériorent pas forcément le SoC. Elles peuvent aussi voler des informations sensibles en observant le système en effectuant une attaque passive. Cependant, elles peuvent également nuire au système et le rendre inutilisable en effectuant une attaque active.

Les attaques sur un SoC sont globalement classées en deux catégories. Il y a tout d'abord les attaques matérielles, identifiées en turquoise dans la Figure 1.4. Elles nécessitent le plus souvent un accès physique au circuit et elles ne peuvent donc pas être déployées en même temps sur un nombre important de circuits. Elles s'appuient sur l'exploitation ou la modification des caractéristiques matérielles du circuit pour réaliser une attaque. Dans la Figure 1.4, on peut notamment voir les modifications effectuées sur le circuit à l'usine de conception. Le deuxième type d'attaques, les attaques logicielles, contrairement aux attaques matérielles peuvent être réalisées à distance et sur un nombre plus important de circuits. Elles sont identifiées en violet dans la Figure 1.4. Les attaques logicielles s'appuient sur l'utilisation des composants logiciels du système, comme les applications utilisateurs, les drivers ou encore le système d'exploitation (voir la Figure 1.4).

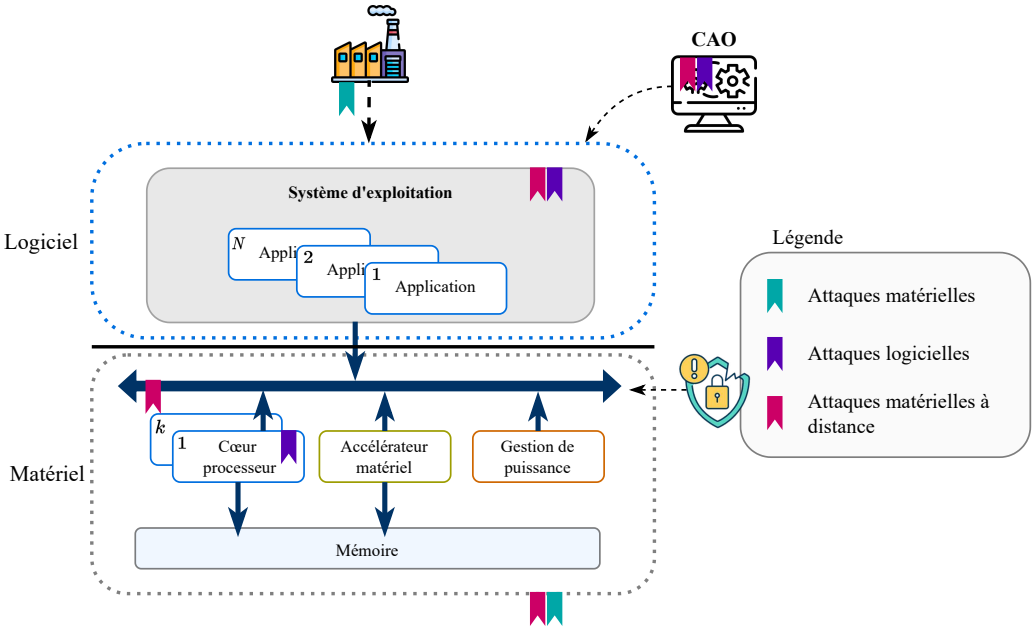


FIGURE 1.4 – Schéma des emplacements d'attaques possibles sur un SOC hétérogène au niveau logiciel et matériel.

Cependant, depuis quelques années, il existe une troisième classe d'attaques qui prend de l'importance : les attaques matérielles basées sur du logiciel. Elles sont identifiées en fuchsia dans la Figure 1.4. Il s'agit d'attaques matérielles qui s'appuient sur une partie logicielle ce qui les rend donc possibles à distance. Les auteurs de [69] présentent une liste non exhaustive de ces attaques. Ce sont ces attaques à distance qui sont considérées dans le cadre de cette thèse. Elles peuvent avoir pour cible les mémoires, la communication au sein du SoC ou encore la microarchitecture.

1.3.1 Attaques visant les mémoires

Comme indiqué précédemment, dans les SoC actuels différents types de mémoires peuvent être intégrés : DRAM, cache, RAM, ROM. Ces mémoires peuvent être partagées entre les différents composants du système et un certain nombre de travaux ont montré qu'il était alors possible de cibler cette hiérarchie mémoire.

DRAM

L'attaque Rowhammer [77][85][86] est une attaque qui se base sur les caractéristiques des mémoires DRAM dans les SoC hétérogènes récents (DRAM<40nm). Cette attaque permet de modifier le contenu de cette mémoire sans y accéder directement. Comme montré dans la Figure 1.5, l'attaquant accède à une ligne voisine de celle qu'il souhaite modifier et effectue des lectures répétées de celle-ci. Du fait de l'architecture, de la taille des cellules mémoires actuelles des DRAM et de leurs espacements, ces lectures répétées entraînent une fuite de charges électriques dans les cellules voisines. Une modification des valeurs des bits de la ligne attaquée s'ensuit.

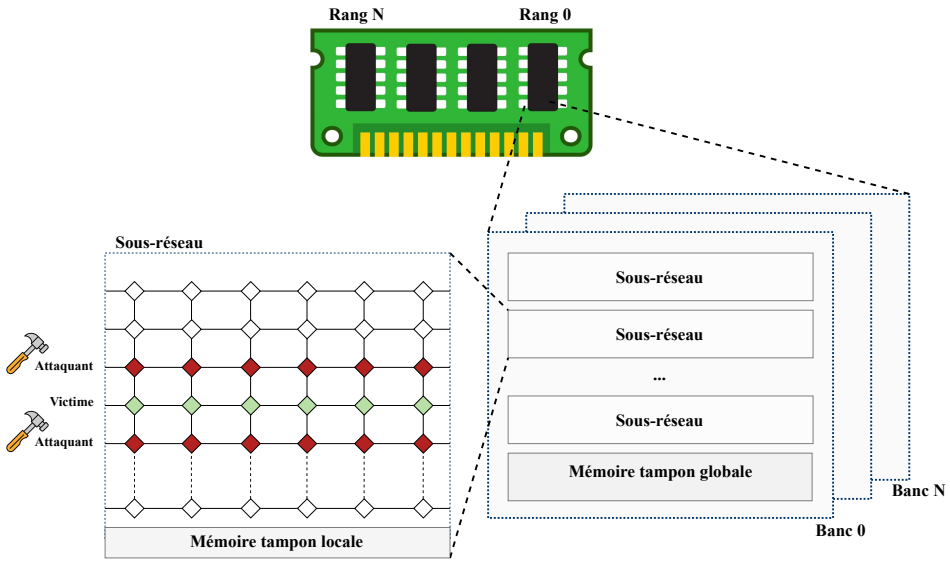


FIGURE 1.5 – Attaque Rowhammer visant la mémoire vive dynamique (DRAM).

Cache

Les mémoires cache des processeurs actuels peuvent être partagées entre plusieurs processus. Il existe plusieurs types d'attaques qui visent les mémoires cache. Elles sont souvent basées sur les temps d'accès à l'une de ses données. Si cette donnée est présente dans la mémoire cache le temps d'accès est court, sinon, il y a défaut de cache et le temps est plus long. Il faut bien évidemment déterminer les seuils qui permettent de déterminer s'il y a un défaut de cache ou non avant de pouvoir effectuer ces attaques. Celles-ci requièrent normalement que le processus victime et l'attaquant soient exécutés par le même processeur, mais des travaux [17] ont montré qu'il était possible de les réaliser depuis la logique programmable. Il existe plusieurs façons de réaliser l'analyse temporelle de la mémoire cache, comme nous l'illustrons par la suite.

1. ***EVICT + TIME*** [38][66]

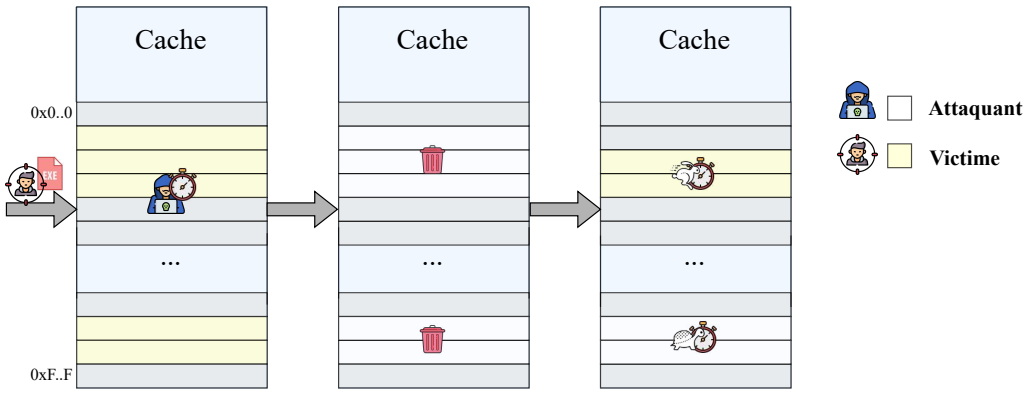


FIGURE 1.6 – Déroulement attaque *Evict + Time* sur la mémoire cache.

La Figure 1.6 présente l'attaque "*Evict + Time*". La première étape pour l'attaquant, montrée sur la gauche de la Figure 1.6, est de laisser la victime exécuter son programme afin de déduire un temps d'accès à la mémoire cache. Ensuite, l'attaquant vide des lignes de cache de son choix et exécute le programme de la victime à nouveau. Puis, comme montré sur la dernière étape Figure 1.6, ce dernier mesure les temps d'accès à la mémoire cache et détermine si la victime a utilisé la ligne précédemment vidée. Ces différences de temps sont illustrées dans la Figure 1.6 avec un lièvre pour un accès à la case de la cache par la victime, qui signifie un temps d'accès plus court. Sinon, si la victime n'a pas fait d'accès, cela est représenté par une tortue qui veut dire un délai plus long. Avec ses relevés, l'attaquant peut en déduire le comportement de la victime.

2. ***PRIME + PROBE*** [47][55]

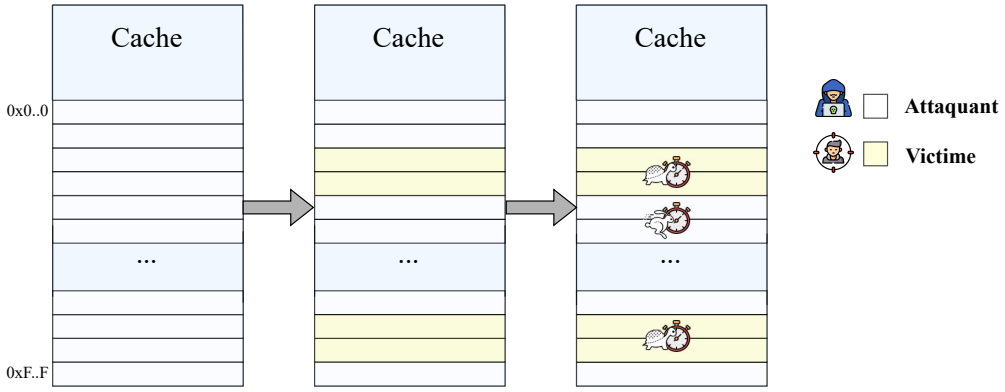


FIGURE 1.7 – Déroulement attaque *Prime + Probe* sur la mémoire cache.

La Figure 1.7 présente l'attaque "*Prime + Probe*". La première étape pour l'attaquant est de remplir plusieurs lignes de la mémoire cache avec ses données (à gauche sur la Figure 1.7). La deuxième étape pour lui est d'attendre que la victime exécute son programme. Il peut ensuite accéder à nouveau à ses mêmes lignes. Cette dernière étape est montrée sur la droite de la Figure 1.7. Il peut ainsi observer quelles lignes de cache ont été modifiées avec les temps d'accès. Comme avec l'attaque précédente, "*Evict + Time*", Figure 1.6, l'attaquant peut en déduire le comportement de la victime.

3. ***FLUSH + RELOAD*** [42][87]

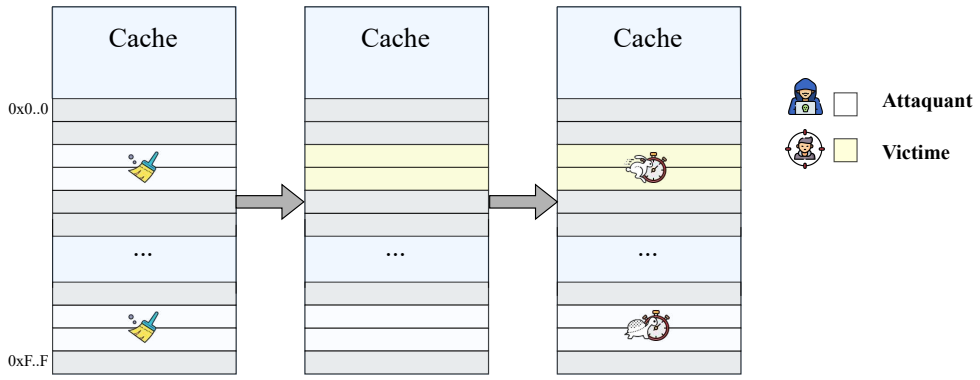


FIGURE 1.8 – Déroulement attaque *Flush + Reload* sur la mémoire cache.

La Figure 1.8 présente l'attaque "*Flush + Reload*". La première étape pour l'attaquant est de vider un emplacement précis de la mémoire cache qu'il partage avec la victime à l'aide de l'instruction *Flush*. Cette instruction est représentée dans la Figure 1.8 avec le balai. L'attaquant attend ensuite que le programme de la victime s'exécute. La dernière étape pour lui, représentée sur la droite de la Figure 1.8, est de mesurer les temps d'accès aux emplacements précis de la mémoire cache. Il peut ainsi en déduire si la victime a accédé à la mémoire.

4. ***FLUSH + FLUSH*** **[41]**

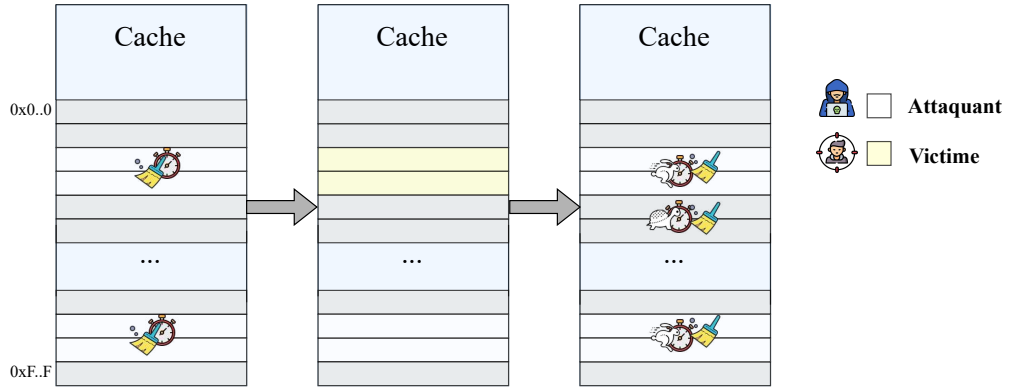


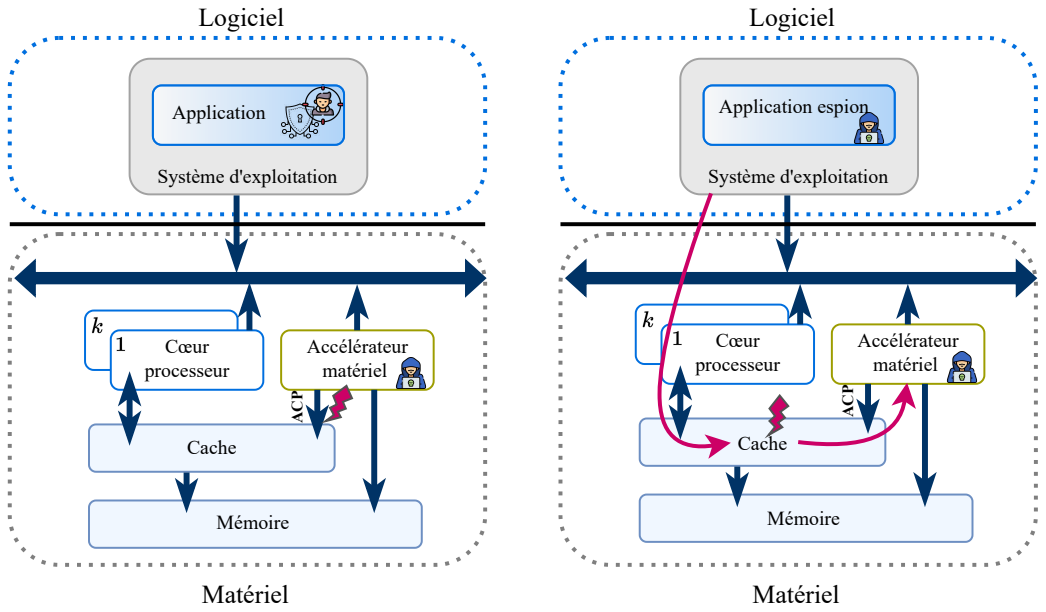
FIGURE 1.9 – Déroulement attaque *Flush + Flush* sur la mémoire cache.

La Figure 1.9 présente l'attaque "*Flush + Flush*". Contrairement aux attaques présentées précédemment, cette attaque ne nécessite aucun accès à la mémoire cache. Elle repose uniquement sur le temps d'exécution de l'instruction *Flush*. La première étape pour l'attaquant est de vider une case de la mémoire cache et de mesurer le temps d'exécution de l'instruction. Ensuite, comme le montre la Figure 1.9, il laisse le programme victime s'exécuter. L'attaquant n'a plus qu'à réaliser des instructions *Flush* sur les différentes cases mémoire et à en mesurer les temps. Il est ainsi en mesure de savoir si la victime a accédé à la ligne de cache précédemment vidée.

5. **Extension des attaques sur la mémoire cache depuis la logique programmable** [17][18]

Les SoC hétérogènes récents sont équipés d'une interface ACP (*Accelerator*

Coherency Port) [7] qui permet de connecter une interface maître d'un accélérateur matériel à la mémoire cache reliée au processeur.



(A) Attaques "*Flush+Reload*" et "*Evict+Time*" sur la mémoire cache depuis le FPGA vers une application logicielle.

(B) Attaque "*Flush+Reload*" depuis le logiciel étendue en canal caché (flèche rose) vers le FPGA.

FIGURE 1.10 – Attaques sur la mémoire cache étendues depuis la logique programmable du SoC hétérogène.

L'attaque [17] vise une implémentation cryptographique AES (*Advanced Encryption Standard*) [1] qui se trouve dans le processeur. Une modification malicieuse est intégrée à l'accélérateur matériel identifié en vert-jaune dans la Figure 1.10. L'entité malveillante est capable de distinguer les défauts des succès de la mémoire de cache en fonction du temps d'exécution des requêtes de communication. L'attaquant réussit à réaliser avec succès des attaques "*Flush+Reload*" et "*Evict+Time*" pour retrouver la clé de chiffrement de l'algorithme.

Les auteurs arrivent à étendre les attaques effectuées sur la cache pour réaliser une attaque de type canal caché. Le but de cette attaque est de créer une voie de communication entre deux entités qui, d'après les règles de sécurité, n'ont pas le droit de communiquer. Dans [17], le canal de communication caché est entre l'accélérateur matériel qui se trouve dans la partie programmable du système et une application dans la partie logicielle du système.

DMA

L'accès direct à la mémoire DMA (*Direct Access Memory*) est un mécanisme qui permet à un périphérique d'accéder directement à la mémoire externe sans passer par le processeur. Ce dispositif permet d'améliorer les performances globales du système car le processeur ne gère plus les simples transferts de données. Cependant, il présente l'inconvénient de priver le processeur de son contrôle et de sa vérification. Le transfert se fait sans supervision, ce qui peut présenter un problème s'il est effectué par une entité malveillante. Ces modules malveillants pourraient soit injecter dans la mémoire du code malicieux, soit ils pourraient tenter de récupérer des données sensibles depuis la mémoire [14][49].

1.3.2 Attaques visant la micro-architecture

Les processeurs modernes utilisent la prédiction de branchement et l'exécution spéculative. Ces deux fonctionnalités peuvent toutefois engendrer des possibilités d'attaques. En effet, la prédiction de branchement, qui a pour but d'améliorer les performances du pipeline du processeur, peut entraîner le pré-chargement d'une instruction non autorisée. L'exécution spéculative, quant à elle, permet de lancer avec anticipation une instruction sans savoir si elle a réellement besoin d'être exécutée. L'instruction peut être non autorisée.

L'exécution spéculative permet d'améliorer les performances du système car ce dernier passe moins de temps sur les branchements conditionnels. Cependant, ces optimisations présentent des vulnérabilités qui permettent de réaliser des attaques dites micro architecturales [34].

Spectre

L'attaque SPECTRE [51] se base sur l'exécution spéculative. Elle permet de manipuler la prédiction de branchement effectuée par le processeur : soit en faisant une injection directe, soit en induisant en erreur les prédictions. Dans la première possibilité, l'attaquant exécute un code « innocent » contenant un branchement vers une instruction non autorisée. Cette dernière provoque un saut vers un code déjà existant qui effectue une opération qui intéresse l'attaquant (gadget). Ce dernier peut ensuite dérober des données sensibles. Pour la deuxième possibilité, l'attaquant insère des données erronées dans le prédicteur. La victime exécute ensuite son programme et à cause du prédicteur, il fait un saut vers un gadget.

Meltdown

L'attaque MELTDOWN [54] cible l'exécution dans le désordre "out-of-order". Ce type d'exécution est utilisé dans les processeurs actuels pour en améliorer les performances. Elle permet de mieux utiliser les ressources du système et de gagner du temps de calcul. L'ordre dans lequel les instructions sont exécutées est réorganisé. Normalement, une application doit pouvoir lire uniquement les données mémoires auxquelles elle a été assignée et non d'autres emplacements. Le processeur doit vérifier les permissions avant de laisser l'application accéder à un contenu mémoire. Cependant, lors de l'exécution dans le désordre, le processeur lit d'abord le contenu mémoire avant de vérifier ces permissions. Puisque ce contenu se retrouve dans la cache, il est alors possible de le récupérer.

1.3.3 Attaques visant la sécurité des communications dans le SoC

Le bus de communication à l'intérieur du système sur puce qui relie tous les composants du système entre eux est un élément particulièrement sensible. S'il se retrouve compromis, cela peut avoir un impact conséquent sur le système comme l'ont prouvé des travaux précédents [14].

Les délais de commercialisation sont de plus en plus courts pour pouvoir mettre au plus vite un SoC sur le marché et assurer une meilleure rentabilité du produit. Ainsi, il est commun pour les concepteurs de systèmes hétérogènes d'acheter des IP (blocs de propriété intellectuelle) matérielles à une tierce personne. Celles-ci ne font pas forcément l'objet de vérifications de sécurité et peuvent renfermer une fonction malveillante, par exemple un cheval de Troie (cf. [48]). Il s'agit d'une modification malicieuse d'un

design qui est activée avec des stimuli précis (déclencheur) et qui peut réaliser une attaque sur le système. Le cheval de Troie peut affecter le circuit de plusieurs façons. Il peut être passif, c'est-à-dire collecter des informations sensibles qui ne lui sont pas destinées. Il peut également interférer plus profondément avec le système en changeant par exemple le contenu de communications ou en prenant le contrôle (voir [12]).

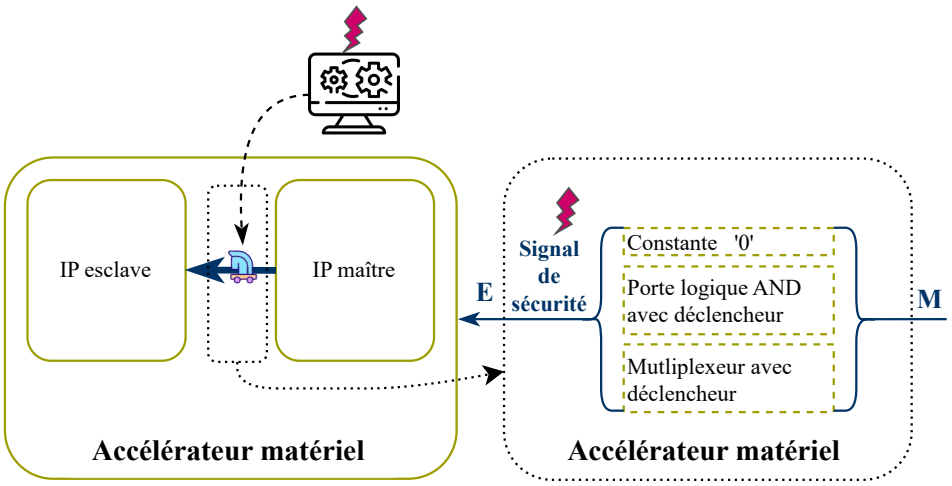


FIGURE 1.11 – Modification de la valeur du signal de sécurité entre une interface maître (M) et une interface esclave (E) à l'aide d'un cheval de Troie matériel pour un accélérateur matériel.

Comme montré sur la Figure 1.4 et l'article [12], un cheval de Troie peut également provenir d'une modification réalisée par un outil de CAO (Conception Assistée par Ordinateur) malicieux, à l'insu de l'utilisateur. Ces attaques peuvent être généralisées à l'ensemble du SoC et peuvent cibler d'autres parties que le bus de communication. Elles peuvent même être généralisées en attaque par déni de service (*Denial-of-Service* (DoS)) qui permet d'empêcher d'accéder à certaines ou à la totalité des ressources d'un système [31]. Dans le cas du bus de communication [12], une IP matérielle ou une application logicielle malicieuse peut effectuer des requêtes de manière continue vers une ressource du système et empêche ainsi les autres composants d'y accéder.

1.3.4 Attaques visant les systèmes de gestion de l'énergie des SoC

Les SoC modernes embarquent des systèmes de gestion de l'énergie. Ces systèmes appelés *Dynamic Voltage and Frequency Scaling* (DVFS) permettent d'ajuster la tension et la fréquence du système en fonction de la tâche en cours de réalisation. Cet ajustement est calculé de manière à minimiser la consommation d'énergie du système tout en garantissant une puissance de calcul suffisante. En d'autres termes, pour une tâche nécessitant beaucoup de puissance de calcul, le DVFS va ajuster la tension et la fréquence pour passer en mode hautes performances. Au contraire, s'il s'agit d'une tâche qui ne requiert que peu de ressources de calcul, le DVFS sera réduit au mode basse consommation. Cette technique permet de maximiser la durée de vie des dispositifs de stockage de l'énergie du système.

Cependant, comme l'ont montré les travaux [2][13][52][82][88], il est possible d'utiliser le DVFS pour attaquer le système. Dans [52] et [82], l'attaquant utilise le DVFS pour injecter des fautes dans le système. Ces attaques sont entièrement basées sur du logiciel et se font à distance sans avoir besoin d'accéder au SoC de manière physique. Dans CLKSCREW [82], un pilote du noyau malicieux va utiliser le DVFS pour

pousser le processeur à ses limites de fonctionnement provoquant ainsi des fautes dans un algorithme de cryptographie. Dans [13], les auteurs réalisent des attaques par canaux cachés en utilisant le DVFS. Ils arrivent à mettre en place quatre canaux cachés différents qui sont présentés dans la Figure 1.12. L'attaquant peut transmettre des données sensibles du SoC vers l'extérieur à l'aide d'émissions électromagnétiques (attaque n°1). Il peut également transmettre des données sensibles entre les cœurs tout en contournant l'extension de sécurité (attaque n°2). L'attaquant peut également échanger des données entre un cœur et la partie programmable sans avoir les autorisations nécessaires (attaques n°3 et 4).

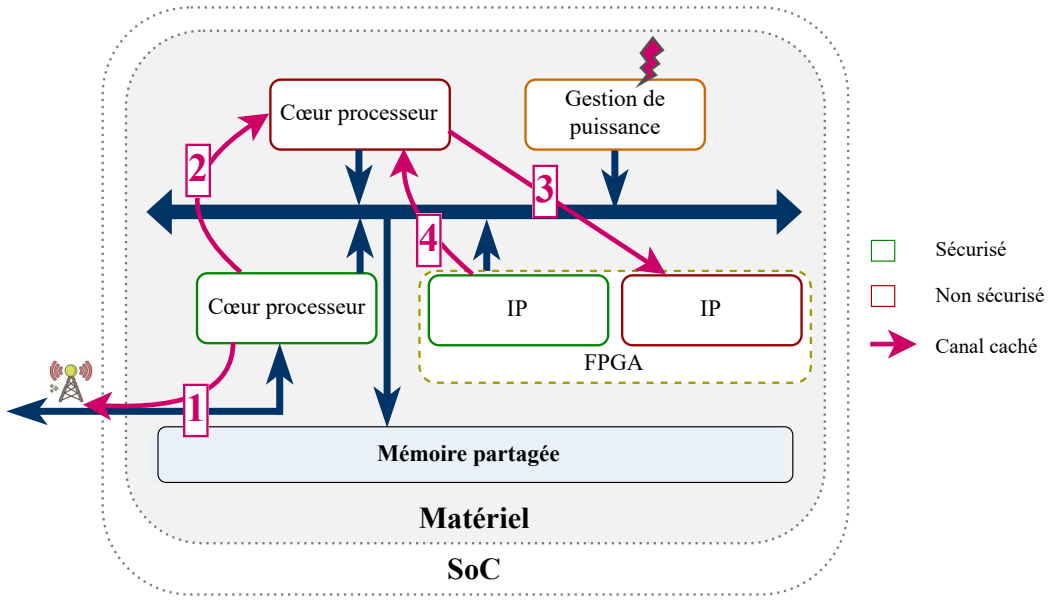


FIGURE 1.12 – Utilisation du système de gestion de l'énergie du SoC (DVFS) pour réaliser des attaques contre le système.

1.3.5 Attaques internes visant les traitements d'accélérations matérielles et canal caché

L'attaque par analyse de canaux auxiliaires (SCA) utilise et analyse des mesures physiques recueillies par l'attaquant. Ces données sont liées à un secret que l'attaquant souhaite obtenir. Grâce au lien de corrélation et à l'analyse que l'attaquant effectue, il peut parvenir à acquérir le secret. Pour obtenir les données corrélées au secret, l'attaquant peut utiliser des primitives matérielles qui agissent comme des capteurs. Un exemple d'attaque est donné dans les travaux [76]. Les auteurs utilisent des capteurs TDC (*Time-to-Digital Converter*) qui mesurent la consommation de puissance. Celle-ci est liée à l'algorithme de chiffrement qui s'exécute dans le FPGA et à l'aide de l'attaque SCA, il est possible de retrouver la clé de cet algorithme. De même, comme présenté dans [88], les auteurs utilisent des oscillateurs en anneaux pour réaliser les capteurs dans le FPGA. Il est possible d'utiliser l'attaque SCA couplée à une attaque par canal caché [19][36]. Les attaques par canaux cachés [3][65] permettent d'envoyer les informations sensibles recueillies précédemment.

Les attaques [13][88] ont pour but de récupérer des secrets du système grâce à des primitives matérielles qui agissent comme des capteurs. Dans [88], l'attaquant utilise des oscillateurs en anneaux pour réaliser des moniteurs de la consommation de puissance d'autres modules présents dans le FPGA. Grâce à ces moniteurs, l'attaquant peut

attaquer un algorithme de cryptographie (RSA) s'exécutant dans le FPGA. L'attaquant peut utiliser ces moniteurs pour réaliser la même attaque sur un algorithme se trouvant dans le CPU (*Central Processing Unit*) du SoC. D'autres attaques internes peuvent être un peu plus actives. Dans FPGAhammer [52], l'attaquant utilise des oscillateurs en anneaux présents dans le FPGA pour injecter des fautes dans le système. Leurs activations ou leurs désactivations provoquent des chutes de tension excessives dans le système qui génèrent des fautes. L'attaquant utilise ces fautes pour attaquer un algorithme de cryptographie (AES).

1.4 Motivations et objectifs de la thèse

Les travaux accomplis pendant cette thèse ont été réalisés dans le cadre du projet TrustSoC financé par l'Agence de l'Innovation Défense (AID) [1]. La polyvalence qu'offrent les SoC hétérogènes fait qu'ils sont utilisés dans de nombreux domaines. Les attaques présentées dans ce chapitre menacent grandement la sécurité de ces circuits ainsi que celle des utilisateurs. Cette thèse apporte une solution à cette problématique de sécurité des SoC hétérogènes avec des propositions d'architectures de SoC-FPGA appelées *TrustSoC*, *RTrustSoC* et *TrustSoC-V*. Elle propose ausi une modélisation de la conception de l'architecture *TrustSoC* avec *TrustSoC-M*. Ces solutions complètent les solutions de l'état de l'art présentées dans le Chapitre 2.

Notre contribution a pour objectif de :

— Proposer une solution minimale et centrée sur le bus de communication du SoC hétérogène, qui est un élément particulièrement vulnérable [14].
— Englober pour la sécurité du système tous ses éléments : processeurs, accélérateurs matériels, bus, etc.
— Proposer des mécanismes de sécurité pour les différentes mémoires du SoC hétérogène.
— Répondre à un modèle de menaces bien défini (détaillé pour chaque contribution).

1.5 Organisation du manuscrit

Le Chapitre 1 présente le contexte des travaux présentés dans ce manuscrit : les notions préliminaires nécessaires et les attaques qui visent la sécurité des SoC hétérogènes à différents niveaux (mémoires, système de gestion de l'énergie, système de communication, micro-architecture, traitement des accélérateurs matériels et canaux cachés).

Le Chapitre 2 présente l'état de l'art avec les principales propositions d'architectures sécurisées issues de l'industrie et de la littérature. Il présente également leurs limites et effectue une première comparaison entre ces travaux et nos contributions. Cela permet de positionner les travaux de cette thèse.

Le Chapitre 3 présente notre première contribution, *TrustSoC*, ses exigences de sécurité, son fonctionnement, ainsi que des résultats d'implémentation. Le Chapitre 3 présente également notre deuxième contribution qui est une extension de *TrustSoC* appelée *RTrustSoC*. Il décrit ses exigences de sécurité, son architecture et les résultats d'implémentation obtenus. Il aborde l'application de scénarios d'attaques présentées dans le Chapitre 1 à *RTrustSoC* et fournit des résultats d'implémentation.

1. https ://www.defense.gouv.fr/aid

Le Chapitre 4 présente notre troisième contribution *TrustSoC-V* qui embarque un processeur RISC-V pour étendre les solutions présentées dans le Chapitre 3 avec des résultats d'implémentation. Le Chapitre 4 présente également RISC-B qui est notre quatrième contribution. RISC-B facilite la mise en œuvre de *TrustSoC-V* pour un concepteur. Il s'agit d'une nouvelle méthodologie pour implémenter des systèmes intégrant un cœur RISC-V.

Le Chapitre 5 présente notre dernière contribution, intitulée *TrustSoC-M*. Il s'agit d'une approche de type MBSE (*Model-Based System Engineering*) pour modéliser la conception de l'architecture *TrustSoC*. *TrustSoC-M* nous permet d'explorer l'espace de conception.

Chapitre 2

État de l'art

Résumé

Ce chapitre a pour but de décrire les principales propositions d'architectures sécurisées de SoC hétérogènes issues de l'industrie et de la littérature. Ces architectures répondent aux attaques présentées dans l'introduction (voir Chapitre 1). Ce chapitre permet de comparer toutes ces solutions et d'en décrire les limites. Les résultats de ces comparaisons permettent de justifier les travaux proposés dans cette thèse, de les positionner et d'en décrire brièvement les contributions.

Table des matières

2.1 Introduction au chapitre . . . 16
2.2 Notions préliminaires . . . 16
2.2.1 Les environnements d'exécution de confiance dans les SoC . . . 16
2.2.2 Les protocoles de bus de communication dans les SoC : AMBA-AXI et TileLink . . . 17
2.3 Architectures sécurisées de SoC hétérogène issues de l'industrie . . . 19
2.3.1 La technologie ARM *TrustZone* . . . 19
2.3.2 WorldGuard de SiFive . . . 22
2.4 Architectures sécurisées de SoC hétérogène issues de la littérature . . . 23
2.4.1 HECTOR-V . . . 24
2.4.2 CURE (*CUstomizable and Resilient Enclaves*) . . . 25
2.4.3 SANCTUARY . . . 28
2.4.4 TEEOD (*Trusted Execution Environments On-Demand*) . . . 30
2.4.5 Une approche décentralisée basée sur des pare-feu matériels . . . 32
2.4.6 E-IIPS (*Extended Infrastructure IP for SoC security*) . . . 34
2.4.7 ProMiSE (*PROgrammable hardware Monitor for Secure Execution*) . . . 35
2.4.8 *Pro-active policing and policy enforcement architecture for securing MPSoCs* . . . 37
2.5 Comparaisons et synthèse . . . 43
2.6 Positionnement des contributions de cette thèse vis à vis de l'état de l'art . . . 45
2.7 Conclusion du chapitre . . . 45

2.1 Introduction au chapitre

La sécurité des circuits intégrés est devenue un sujet de recherche important ces dix dernières années. En raison de l'augmentation constante de leur complexité, ces circuits sont devenus extrêmement polyvalents et sont utilisés dans de nombreux domaines. Ils peuvent être utilisés dans le cadre d'applications sensibles ce qui en fait une cible de choix pour des attaquants [39]. Dans ce chapitre, nous présentons et comparons les différentes propositions industrielles et académiques pour sécuriser une architecture de SoC hétérogène.

2.2 Notions préliminaires

Avant de présenter les différentes solutions architecturales sécurisées, ce chapitre commence par une introduction aux notions techniques qui sont utilisées dans les différentes propositions de l'état de l'art ainsi que dans nos contributions : les environnements d'exécution de confiance et les protocoles de communication AMBA-AXI et TileLink.

2.2.1 Les environnements d'exécution de confiance dans les SoC

Nous allons maintenant présenter la première notion préliminaire : les environnements d'exécution de confiance (TEE pour *Trusted Execution Environment*). Ces environnements sont au cœur de nombreuses propositions académiques et industrielles.

Un TEE [75] est une zone sécurisée et isolée des autres environnements d'exécution dans un système informatique, où il est possible d'exécuter du code de confiance. L'architecture d'un système embarquant un TEE est représentée dans la Figure 2.1. Le TEE ainsi que ses ressources (identifiées en vert) sont isolés de la partie qui n'est pas de confiance (identifiée en rouge). L'intégrité et la confidentialité des données traitées et du code exécuté dans le TEE sont garanties. Aucun accès illégitime ou modification ne peut être effectué sans autorisation.

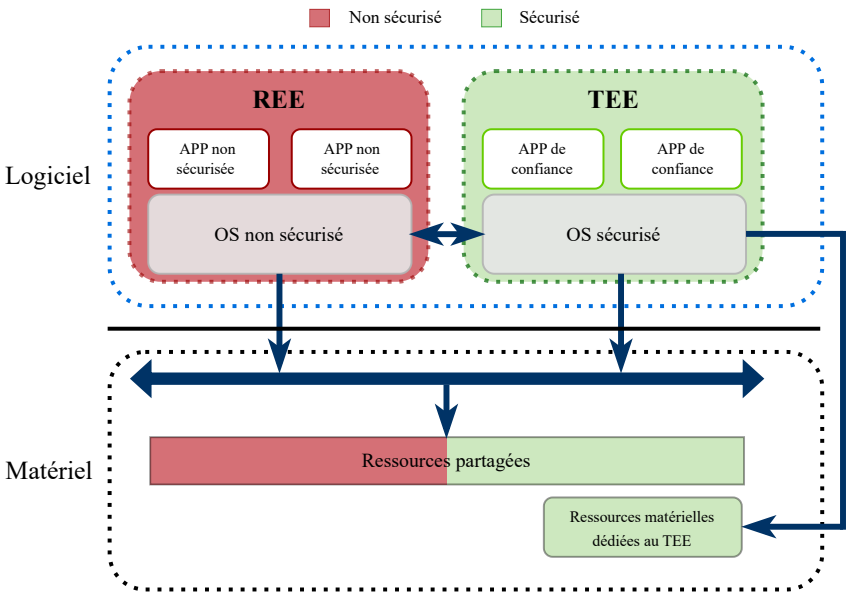


FIGURE 2.1 – Architecture matérielle et logicielle de l'implémentation d'un TEE.

Il existe deux possibilités pour réaliser un TEE : via une isolation purement matérielle ou via une isolation logicielle assistée par le matériel. La première solution duplique les ressources du système, c'est-à-dire que le système est séparé en deux : une partie pour toutes les ressources qui ne sont pas de confiance et une autre partie pour les éléments de confiance. Toutes les ressources sensibles sont dupliquées : coprocesseur, mémoires caches, prédictions de branchements, etc. Ainsi, toutes les ressources de confiance sont isolées des parties non sécurisées du SoC. Cette solution apporte donc un haut degré d'isolation. Cependant, la duplication des ressources a un coût non négligeable sur le système et n'est pas toujours viable. Pour la deuxième possibilité, le TEE est exécuté sur les mêmes ressources matérielles que le REE (*Rich Execution Environment*). Quand une application de confiance doit être exécutée, le REE demande son exécution par le TEE via un composant dédié qui authentifie et contrôle l'intégrité de l'application. Généralement, c'est un composant (logiciel ou matériel) appelé moniteur de sécurité qui est chargé de ce changement de contexte. Il contrôle également qu'aucune ressource sensible n'est partagée entre le TEE et le REE en vidant par exemple des partitions mémoire. Cette approche par rapport à la précédente permet d'éviter la duplication des ressources, mais au détriment de l'isolation. En effet, ce type d'approche permet de lutter que contre les attaques logicielles alors que la première solution d'implémentation (duplication des ressources) peut prévenir également les attaques qui visent la microarchitecture. En conclusion, l'ajout d'un TEE permet de fournir une couche de sécurité supplémentaire et vitale contre les attaques présentées dans le premier chapitre de cette thèse. Le TEE est une notion importante et est particulièrement utilisée dans les propositions d'architectures sécurisées de SoC qui seront présentées dans les Sections 2.3 et 2.4 de ce chapitre. Mais avant, nous devons présenter quelques notions préliminaires sur les protocoles de communication principalement utilisés dans ces architectures. C'est ce que nous proposons de faire dans la prochaine section.

2.2.2 Les protocoles de bus de communication dans les SoC : AMBA-AXI et TileLink

Les protocoles de communication principalement utilisés dans les propositions d'architectures sécurisées de la littérature et de l'industrie sont : AMBA-AXI et TileLink. Le protocole de communication AMBA-AXI est très largement utilisé dans de nombreux SoC et SoC-FPGA actuellement sur le marché. Quant au protocole de communication TileLink, il est largement utilisé dans les architectures utilisant des processeurs RISC-V.

AMBA-AXI

Le bus *Advanced eXtensible Interface* (*Advanced eXtensible Interface*) de la norme AMBA [8] relie deux interfaces de communication entres elles : une interface maître et une interface esclave. Le protocole AXI utilise une logique de poignée de main entre les différents signaux du bus. Le mécanisme de poignée de mains du bus AXI autorise une opération lorsque les signaux *XVALID* et *XREADY* relatifs à cette opération sont actifs en même temps. Nous nous intéressons uniquement à la version du protocole nommée *AXI4-full* et non aux versions AXI4 *lite* ou AXI *stream*. La Figure 2.2 présente la logique du protocole de communication AXI4-*full* composée de cinq canaux, qui sont :

1. Le canal de l'adresse d'écriture *AW* (identifié en vert Figure 2.2) :

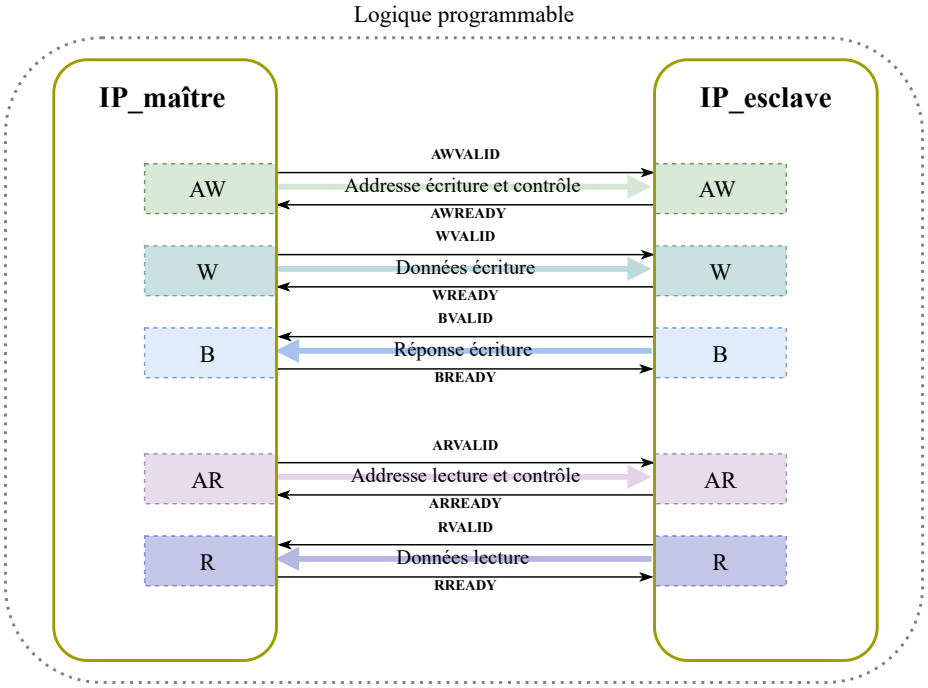


FIGURE 2.2 – Représentation du protocole AXI avec les cinq canaux de communication.

Il s'agit du canal pour transmettre l'adresse de l'esclave à laquelle l'interface maître souhaite écrire. Les signaux de poignée de main pour ses opérations sont *AWVALID* et *AWREADY*. Le signal pour l'adresse est nommé *AWADDR* et a une taille égale à la largeur du bus d'adresses.

2. Le canal des données à écrire *W* (identifié en bleu-vert Figure 2.2) :

   Il s'agit du canal pour transmettre les données à écrire dans l'adresse de l'esclave. Les signaux de poignée de main pour ses opérations sont *WVALID* et *WREADY*. Le signal pour les données est nommé *WDATA* et a une taille égale à la largeur du bus de données.

3. Le canal de réponse d'écriture *B* (identifié en bleu clair Figure 2.2) :

   Ce canal permet à l'interface esclave d'indiquer à l'interface maître si la transaction a échoué ou non via le signal *BRESP* (sur deux bits). Les signaux de poignée de main pour ses opérations sont *BVALID* et *BREADY*.

4. Le canal de l'adresse de lecture *AR* (identifié en mauve Figure 2.2) :

   Il s'agit du canal pour transmettre à l'interface esclave l'adresse *ARADDR* à laquelle l'interface maître souhaite lire. Les signaux de poignée de main pour ses opérations sont *ARVALID* et *ARREADY*. Le signal *ARADDR* a une taille égale à la largeur du bus d'adresses.

5. Le canal des données de lecture *R* (identifié en violet Figure 2.2) :

   Il s'agit du canal utilisé pour transmettre les données de l'interface esclave que l'interface maître souhaite lire. Les signaux de poignée de main pour ces opérations sont *RVALID* et *RREADY*. Le signal pour les données est nommé *RDATA*, il a une taille égale à la largeur du bus de données. L'interface esclave peut également envoyer un signal de réponse sur deux bits pour la lecture, identifié comme *RRESP* pour indiquer si la transaction a échoué ou non.

Les signaux *XRESP* de la lecture ou de l'écriture (sur deux bits) indiquent l'état de la transaction. Ils peuvent prendre plusieurs valeurs. Si la transaction s'est déroulée normalement l'interface esclave indique "00" pour *OKAY*. S'il y a eu un problème, il y a deux codes d'erreur possibles. La première valeur, "11", correspond au code d'erreur *DECERR* dans le cas où le bus ne trouve pas l'interface esclave à laquelle le maître souhaite accéder. La deuxième valeur, "10", signifie *SLVERR*. Elle est dans le cas où l'interface esclave est accessible mais qu'un problème est survenu : mauvaise taille des données ou des adresses, non-conformité aux politiques de sécurité de l'IP, etc.

En plus des signaux de fonctionnement de base, le protocole AXI offre la possibilité d'envoyer des données supplémentaires sans surcoût (pas de latence supplémentaire) via les signaux utilisateur. Ces derniers sont identifiés comme *XUSER* (où X représente le canal) et permettent d'envoyer jusqu'à mille vingt-quatre bits supplémentaires à chaque transaction sans surcoût.

TileLink

Le protocole TileLink [78] spécifié par SiFive est utilisé pour interconnecter des composants dans un SoC : des multiprocesseurs polyvalents, des co-processeurs, des accélérateurs, des composants DMA, etc. Le protocole TileLink a été, à l'origine, conçu pour les processeurs RISC-V, bien qu'il puisse désormais être utilisé pour d'autres types de processeurs. Il repose sur cinq canaux de communication nommés A, D, B, C et E. Nous n'allons pas présenter son fonctionnement car il est très similaire à celui de l'AXI, présenté ci-dessus. En effet, il est basé sur le même schéma de poignée de main avec des signaux *x_valid* et *x_ready* (où *x* représente le canal). Ce protocole, comme pour l'AXI, a la capacité de transmettre des informations supplémentaires sans surcoût (en termes de performances temporelles) grâce à des signaux spécifiques appelés *UserField*. Nous rappelons, pour le protocole AXI, ces signaux sont appelés *XUSER* (où X représente le canal de communication). C'est cette fonction qui est importante de retenir pour la suite, notamment pour la présentation des architectures sécurisées dans la section suivante.

2.3 Architectures sécurisées de SoC hétérogène issues de l'industrie

La sécurisation d'une architecture de SoC hétérogène requiert l'ajout de composants logiciels ou matériels afin de protéger un élément du système. Les solutions utilisant uniquement des modules matériels sont généralement plus intéressantes en termes de performances, mais sont plus compliquées à mettre en œuvre que les solutions logicielles. Un équilibre entre ces deux types de solutions est donc essentiel. Dans cette section, nous allons présenter les principales solutions architecturales de sécurisation de SoC hétérogènes issues de l'industrie : leurs modèles de menaces, leurs fonctionnements et leurs limites.

2.3.1 La technologie ARM *TrustZone*

Afin de protéger l'exécution de codes sensibles et de permettre la réalisation de TEE au sein d'un même microprocesseur, ARM a développé la technologie ARM *TrustZone* [4].

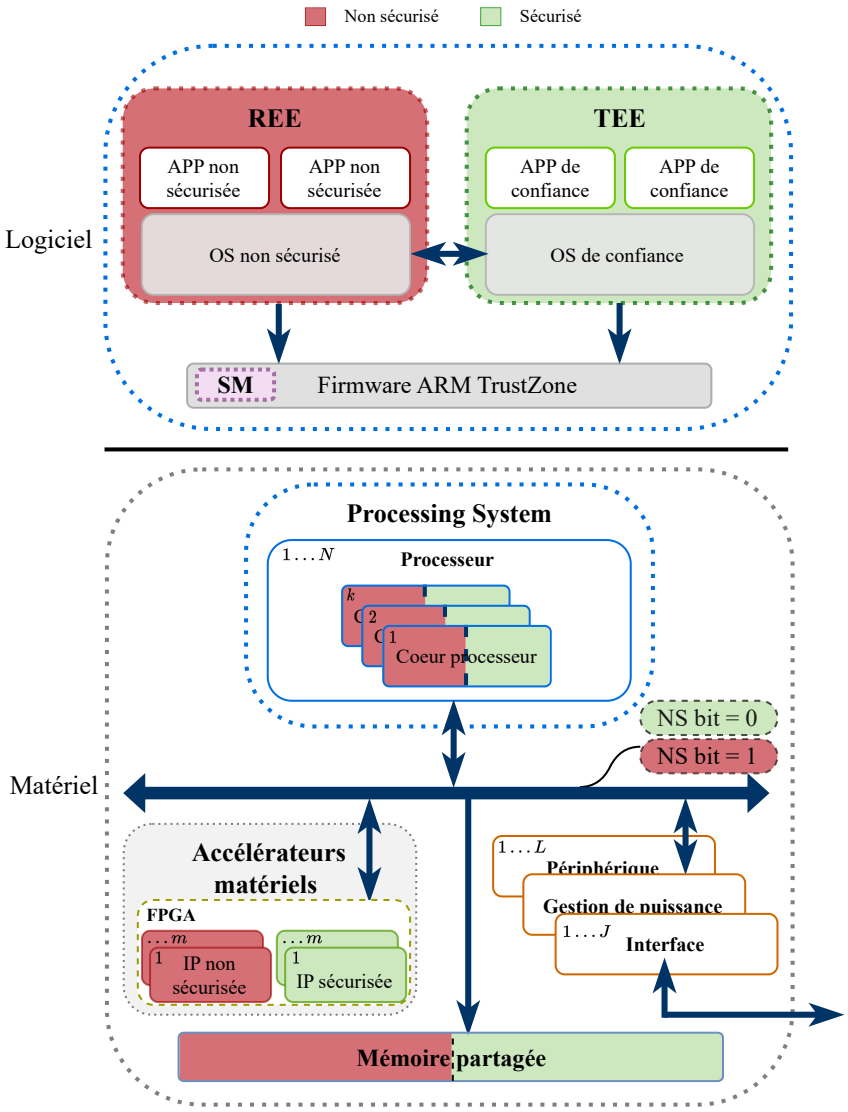


FIGURE 2.3 – Architecture logicielle et matérielle de la technologie *TrustZone* appliquée à une architecture de SoC hétérogène.

Modèle de menaces

Le modèle de menaces de la technologie ARM *TrustZone* ne prend en compte que les attaques logicielles. Il s'agit d'attaques à distance utilisant des virus ou autres logiciels malveillants exploitant des vulnérabilités logicielles pour obtenir l'accès à des ressources sensibles du système. Les attaques utilisant des failles matérielles sont exclues.

Architecture

La technologie ARM *TrustZone* permet de sécuriser les processeurs ARM d'un SoC. Un exemple d'application de cette technologie à un SoC hétérogène est présenté dans la Figure 2.3. La technologie ARM *TrustZone* est disponible à partir de la génération ARMv6. Elle permet de réaliser un TEE par isolation logicielle assistée par le matériel. La technologie ARM *TrustZone* propose une séparation virtuelle du processeur et de ses ressources (cache, mémoires, registres) en deux mondes distincts garantie par des

contrôleurs matériels. Le premier monde, identifié en vert dans la Figure 2.3, est un monde sécurisé où s'exécute un environnement d'exécution de confiance qui contient des applications sécurisées (TA pour *Trusted Application*). Des contrôleurs matériels empêchent l'accès à certains composants lorsque le système se trouve dans ce monde sécurisé. L'autre monde est le monde non sécurisé (identifié en rouge dans la Figure 2.3) qui exécute un système d'exploitation normal avec des applications qui ne sont pas de confiance. Le registre SCR (*Secure Configuration Register*) indique au système dans quel monde il opère. Il inclut un bit qui définit l'état de sécurité du système appelé *NS bit* (*Non-Secure bit*). Il est transmis à chaque communication, comme présenté sur la Figure 2.3. S'il est à l'état haut '1', le système se trouve dans l'état non sécurisé. Sinon, s'il est à l'état bas '0', il est dans le monde sécurisé. Il est également possible de sécuriser les coprocesseurs [5] ainsi que le reste du système comme proposé dans les SoC du vendeur AMD où le signal de sécurité est transmis au reste du système. Les contrôleurs de sécurité matérielle utilisent ensuite le signal pour garantir la sécurité du système. Chaque fois qu'une transaction est décodée et traitée par une entité, celle-ci doit vérifier le niveau de sécurité de la requête. Elle doit le comparer avec son propre niveau de sécurité et vérifier que les deux sont en accord ; sinon, elle doit refuser la transaction. Dans le cas de la technologie ARM *TrustZone*, le moniteur de sécurité (SM) est implémenté de manière logicielle. Il s'agit d'un mode supplémentaire : le mode moniteur. Il est utilisé via l'instruction *Security Monitor Call* (SMC) qui réalise le basculement entre les deux mondes. Le moniteur a également pour tâche de changer la valeur du signal de sécurité tout en évitant la fuite de données.

Limitations

Plusieurs travaux ont mis en lumière des vulnérabilités dans la technologie ARM *TrustZone* avec son extension, en particulier les travaux [13], [14] et [40].

Les travaux, [14], montrent qu'il est possible de casser la sécurité en modifiant le bit de sécurité transmis par les signaux *AXPROT*. Les auteurs modifient, à l'aide d'outils de CAO et de scripts malicieux, les designs du concepteur. Les erreurs générées par ces modifications sont passées sous silence et aucun message d'erreur n'est affiché. Ainsi, en modifiant les signaux de sécurité, l'attaquant peut avoir accès à des ressources sensibles. Dans les travaux [13], l'attaquant utilise le DVFS (*Dynamic Voltage and Frequency Scaling*) pour réaliser des attaques par canaux cachés dans un système où la technologie ARM *TrustZone* est activée. Le DVFS va être utilisé par l'émetteur pour moduler la fréquence de fonctionnement du système afin de transmettre un '0' ou '1' logique à un récepteur. Différents scénarios d'attaques sont proposés : une transmission vers l'extérieur du SoC, un transfert interne entre deux cœurs ARM de niveaux de sécurité différents d'un système et finalement un transfert entre une IP matérielle dans le FPGA et un cœur ARM dans la partie *Processing System*. Dans [40], un cheval de Troie matériel contenu dans une IP matérielle dans la partie reconfigurable du SoC permet d'outrepasser le démarrage sécurisé via un accès par le port ACP. Cette modification compromet un TEE qui utilise la technologie ARM *TrustZone* via la mémoire et les périphériques.

La technologie ARM *TrustZone* est une solution de sécurité complémentaire qui peut être appliquée sur une architecture de SoC-FPGA existante. Cette solution est donc limitée dans son contrôle de la sécurité puisqu'elle est ajoutée après conception et doit donc s'adapter sans modification matérielle à l'architecture actuelle du SoC-FPGA. De plus, la technologie ARM *TrustZone* est également limitée par son faible nombre de domaines sécurisés avec un seul monde sécurisé.

2.3.2 WorldGuard de SiFive

SiFive a présenté la solution WorldGuard [79] (Wg) pour sécuriser les processeurs SiFive et RISC-V. Un exemple d'application de cette technologie à un MPSoC (*MultiProcessor System-On-a-Chip*) est présenté dans la Figure 2.4.

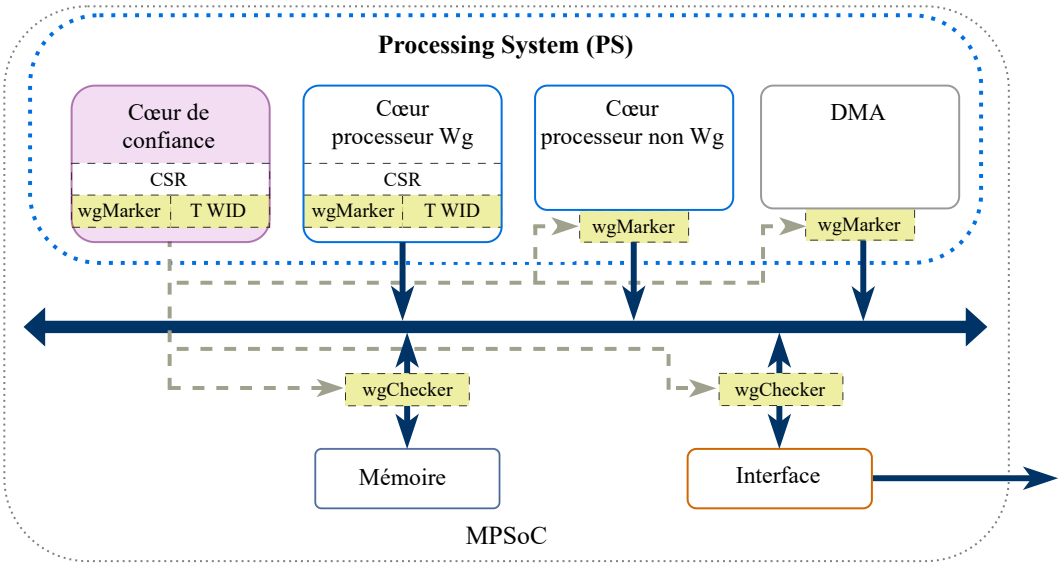


FIGURE 2.4 – Architecture matérielle de la proposition WorldGuard de SiFive.

Modèle de menaces

Le modèle de menaces de *WorldGuard* se concentre uniquement sur les attaques logicielles et non invasives. Les attaques physiques et par canaux auxiliaires sont exclues. *WorldGuard* considère des entités malicieuses qui essayent d'exploiter des vulnérabilités logicielles pour prendre le contrôle de l'exécution. Leur but est de réaliser des élévations de privilèges pour pouvoir accéder à des ressources sensibles du système.

WorldGuard considère plusieurs éléments du système de confiance : le code du démarrage sécurisé dans la ROM (ZSBL (*Zero Stage BootLoader*)), le code de configuration de *WorldGuard* (FSBL (*First Stage BootLoader*)) et l'agent de confiance (identifié en rose dans la Figure 2.4). L'agent de confiance correspond au moniteur de sécurité. De plus, le modèle de menaces de *WorldGuard* stipule qu'un code s'exécutant dans un cœur ne peut pas accéder à plus de ressources que ce qui est configuré pour son monde. Les blocs de l'architecture *WorldGuard* ne peuvent être configurés que par l'entité de confiance. *WorldGuard* ne garantit pas de protection ou d'isolation pour des accès au sein du même monde. Enfin, l'architecture ne garantit pas la stabilité fonctionnelle ni la robustesse des applications après le rejet des transactions.

Architecture

WorldGuard est une solution d'isolation logicielle assistée par le matériel. Elle fournit des contextes d'exécution logicielle appelés mondes. L'isolation est ensuite obtenue grâce à un contrôle d'accès aux ressources (mémoires et périphériques) via le matériel. Chaque monde est identifié par son identifiant unique WID (*World Identifier*) qui est transmis à chaque requête via les protocoles TileLink ou AMBA-AXI4. Pour ces deux protocoles, aucune modification n'est nécessaire pour transmettre les identifiants. Le nombre de mondes de *WorldGuard* est fixé lors de la phase d'élaboration du matériel et

dépend des caractéristiques du système. En fonction du nombre de mondes, le nombre de bits nécessaires pour coder les identifiants WID est déterminé. Le contrôle d'accès aux ressources pour garantir l'isolation logicielle de *WorldGuard* se fait à l'aide de deux blocs matériels WgMarker et WgChecker, identifiés en vert-jaune dans la Figure 2.4. Le bloc WgMarker est associé aux interfaces de communication de type maître et le bloc WgChecker est pour celles de type esclave.

Un WgMarker est inséré entre le bus de communication et le module avec l'interface de communication maître. Il ajoute le WID du bloc à chaque transaction sortante. Pour les agents conscients de *WorldGuard*, e.g. cœur processeur Wg en gris sur la Figure 2.4, le WID évolue en fonction du contexte d'exécution. Ce qui n'est pas le cas pour les agents non conscients du système, comme le cœur de processeur non Wg (Figure 2.4), où le WID est fixe. Cependant, pour ce type de WgMarker, le WID peut être configuré par l'agent de confiance (bus de communication signalé en pointillés kaki sur la Figure 2.4).

Le contrôle d'accès se fait au niveau des interfaces esclaves avec le bloc WgChecker. Ce bloc analyse les transactions entrantes avec les identifiants en fonction des règles d'accès configurées. La transaction n'est pas transmise à l'entité tant que les droits ne sont pas validés. Si l'accès est refusé, la transaction n'est pas transmise et le registre d'erreur est mis à 1. Une réponse de refus ou une interruption peut alors être signalée.

WorldGuard propose également un accès contrôlé pour la mémoire cache. Un identifiant WID est stocké dans chaque ligne de cache et à chaque accès cet identifiant est comparé à celui de la requête. S'ils correspondent ainsi que les adresses, l'accès est autorisé. Sinon, la ligne est évincée et la requête est traitée comme un défaut de cache.

Limites

WorldGuard est une solution logicielle qui vise à sécuriser les processeurs RISC-V et SiFive, qui ne sont pas natifs de la plupart des systèmes. Cette solution de sécurisation n'englobe que très peu les ressources matérielles du système. Dans une proposition d'architecture de SoC sécurisée par conception, la sécurité doit présenter une dualité : logicielle et matérielle.

WorldGuard ne prend pas en compte les attaques par canaux auxiliaires et le contrôle d'accès des droits du système se fait uniquement à la réception de la transaction. Des requêtes malicieuses peuvent donc circuler sur le bus de communication. Par ailleurs, cette architecture ne prévient pas les attaques de type DoS.

L'architecture *WorldGuard* ne prend pas en compte les menaces au sein d'un même monde. En effet, dans le modèle de menaces stipulé, *WorldGuard* ne garantit pas de protections au sein d'un même monde.

2.4 Architectures sécurisées de SoC hétérogène issues de la littérature

Dans cette section, nous allons présenter les principales solutions architecturales de sécurisation de SoC hétérogènes issues de la littérature : leurs modèles de menaces, leurs exigences de sécurité, leurs fonctionnements et leurs limites.

2.4.1 HECTOR-V

HECTOR-V (Figure 2.5) est une proposition académique [64] qui présente une méthodologie pour concevoir des architectures de SoC sécurisées. Elle est orientée pour des processeurs de type RISC-V.

Modèle de menaces

HECTOR-V considère des attaques faites contre l'environnement d'exécution de confiance par le biais de vulnérabilités architecturales ou du noyau [22]. L'attaquant peut également utiliser l'interface de communication entre la partie dédiée à la sécurité (TEE) et la partie qui n'est pas de confiance du système (REE). La sécurité des applications dans ces deux domaines peut également être menacée par des attaques par analyse de canaux auxiliaires sur la mémoire cache. Le modèle de menaces couvre également les attaques contre les périphériques, comme les accès illégitimes aux mémoires sécurisées ou protégées ou à des périphériques sensibles.

Architecture

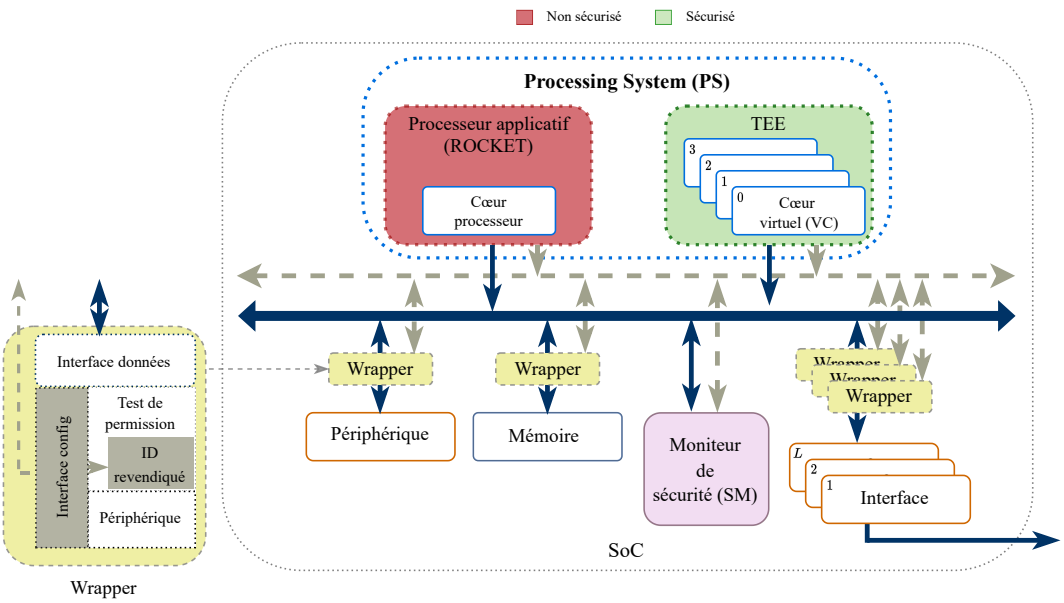


FIGURE 2.5 – Architecture matérielle de la solution HECTOR-V.

HECTOR-V met en place une isolation entre les processeurs. Un processeur gère les applications qui ne sont pas de confiance (en rouge dans la Figure 2.5) et un coprocesseur RVSCP (*RISC-V Secure Co-Processor*) gère les applications dédiées à la sécurité (en vert clair dans la Figure 2.5). Grâce à cette duplication aucun composant sensible (exemples : cache, branchements de prédiction) n'est partagé.

La sécurité proposée par HECTOR-V repose également sur un mécanisme de communication sécurisée. Cette sécurisation est établie à l'aide d'identifiants uniques : un pour le cœur du processeur, un pour le processus et un pour le périphérique. Ces identifiants, contrairement à ceux de la technologie ARM *TrustZone*, sont codés sur plusieurs bits (16 bits dans le prototype) et sont transportés dans le système par les signaux utilisateur du protocole AXI. Pour établir cette communication sécurisée, des petits pare-feu (identifiés en vert-jaune dans la Figure 2.5), "wrappers", sont ajoutés entre le bus de communication et chaque périphérique. Dans HECTOR-V, les bus de

communication sont différenciés. Il y a un bus avec le protocole AXI4-*lite*, identifié en kaki sur la Figure 2.5, pour la configuration du système et des pare-feu. L'autre bus de communication suit le protocole AXI4-*full*, identifié en bleu foncé sur la Figure 2.5, pour transmettre les données. Les pare-feu sont chargés de filtrer les requêtes entrantes. Pour cela, chaque périphérique doit être revendiqué par une entité. L'entité fait une demande au moniteur de sécurité qui vérifie les droits de celle-ci sur le périphérique. Chaque périphérique a une liste d'entités qui ont le droit de le revendiquer. Si les droits sont vérifiés, le moniteur de sécurité fixe l'identifiant de l'entité dans le pare-feu (en kaki sur la Figure 2.5) et ce dernier n'accepte plus que les requêtes de cette entité. La revendication du périphérique peut se faire soit de manière fixe, c'est-à-dire que le pare-feu n'est pas configurable. L'identifiant est codé matériellement et n'évolue pas au cours du temps. Ou alors, la revendication évolue au cours du temps. Le pare-feu est dynamique et une entité peut changer l'identifiant qui le revendique via le canal de configuration. La communication sécurisée proposée par HECTOR-V repose également sur le module appelé moniteur de sécurité (SM), identifié en rose dans la Figure 2.5. Ce module est codé matériellement et est responsable de la gestion des autorisations d'accès aux périphériques. Pour chaque périphérique, le moniteur de sécurité enregistre dans une table son état (revendiqué ou libre) et la liste des identifiants qui peuvent avoir accès au périphérique. Pour revendiquer un périphérique, une entité doit avoir l'accord du moniteur de sécurité. Afin d'éviter les attaques DoS, le moniteur de sécurité active un compteur à chaque fois qu'une entité revendique un périphérique. Si celui-ci dépasse un certain seuil, le moniteur de sécurité retire la possession du périphérique à l'entité. Ainsi, l'accès à un périphérique ne peut pas être bloqué.

Limitations

HECTOR-V a un modèle d'implémentation de TEE basé sur l'isolation par le matériel. Cette solution permet une meilleure protection contre les attaques visant la microarchitecture ou les canaux auxiliaires. Cependant, le problème de cette solution est la duplication des ressources. En effet, les processeurs ainsi que tous les composants critiques sont dupliqués. Cette solution n'est pas entièrement viable sur des systèmes embarqués où la contrainte sur les ressources est forte. Dans HECTOR-V, la duplication des ressources limite également le nombre de domaines possibles : un domaine qui n'est pas de confiance et un domaine sécurisé. Cette ségrégation en deux parties revient à la même limitation observée pour la technologie ARM *TrustZone*. Enfin, HECTOR-V ne prend pas en compte l'hétérogénéité du SoC dans son ensemble. Aucune mesure n'est prise si de la logique matérielle (type FPGA) est embarquée à l'intérieur du système.

2.4.2 CURE (*CUstomizable and Resilient Enclaves*)

CURE est une proposition académique [10] qui est basée sur un système d'enclaves et orientée pour des processeurs de type RISC-V. Une enclave est un environnement d'exécution sécurisé qui est isolé du reste du système.

Modèle de menaces

Dans CURE, le modèle de menaces considère uniquement des attaques logicielles qui visent à compromettre l'ensemble des composants logiciels du système et le système d'exploitation. La seule partie logicielle du système exemptée aux attaques est appelée la *Trusted Computing Base* (TCB). La TCB permet de configurer les primitives de

sécurité matérielle et est donc considérée comme de confiance. Le but de l'attaquant est de récupérer des informations sensibles de la TCB ou d'une enclave. Le modèle de menaces comprend également les accès illégitimes aux mémoires et aux périphériques. De plus, l'attaquant peut réaliser des attaques de type DMA. Le matériel sous-jacent au logiciel est, lui, considéré comme de confiance. Toutes les attaques qui exploitent des failles matérielles sont exclues, de même que les attaques qui nécessitent un accès physique. Enfin, les attaques de type DoS ne sont pas envisagées.

Exigences de sécurité

La proposition de CURE repose tout d'abord sur des exigences de sécurité (ES). Elles sont au nombre de quatre :

ES.1 : Protection de l'enclave

L'intégrité du code de l'enclave doit être protégée au repos et inaccessible quand elle est exécutée. Toutes les données sensibles de l'enclave doivent rester confidentielles et protégées en permanence. Une enclave doit également être protégée sur toutes les couches logicielles (utilisateur, noyau, hyperviseur et firmware) d'autres enclaves potentiellement malicieuses et des attaques DMA.

ES.2 : Primitives de sécurité matérielle

La protection des enclaves doit être assurée par des composants matériels sécurisés qui ne peuvent être configurés que par un module logiciel sécurisé.

ES.3 : Logiciel Trusted Computing Base minimal

La TCB doit être protégée contre les adversaires dans toutes les couches logicielles (utilisateur, noyau, hyperviseur et firmware) et avoir une taille minimale pour pouvoir être formellement vérifiée.

ES.4 : Résistance aux attaques par canaux auxiliaires

L'architecture CURE doit embarquer les contre-mesures les plus pertinentes contre les attaques logicielles par canaux auxiliaires : sur la cache, via le système d'exploitation et par exécution transitoire.

Architecture

CURE, présentée dans la Figure 2.6, propose trois types d'enclaves avec différents niveaux de sécurité.

La première se situe au niveau de l'utilisateur. Elle est identifiée en beige dans la Figure 2.6. Son fonctionnement est géré par le système d'exploitation principal via des appels système (mémoire, interruptions, etc.). Les changements de contexte liés à l'enclave sont gérés par un bloc dédié défini dans le firmware : le moniteur de sécurité, identifié en rose sur la Figure 2.6. Le deuxième type d'enclaves proposé dans CURE est au niveau du noyau. Il est identifié en vert clair dans la Figure 2.6. Ce type d'enclave est isolé du système d'exploitation principal qui n'est pas considéré comme de confiance. Dans ce type d'enclave, il est possible d'exécuter du code sans système d'exploitation pour implémenter certaines fonctionnalités. Le troisième et dernier type d'enclaves se situe au niveau du firmware. Elle est identifiée en rose dans la Figure 2.6. Ces enclaves fournissent le degré de sécurité le plus élevé. Elles permettent d'isoler des parties de code logiciel de confiance du reste du système, comme le moniteur de sécurité. Le

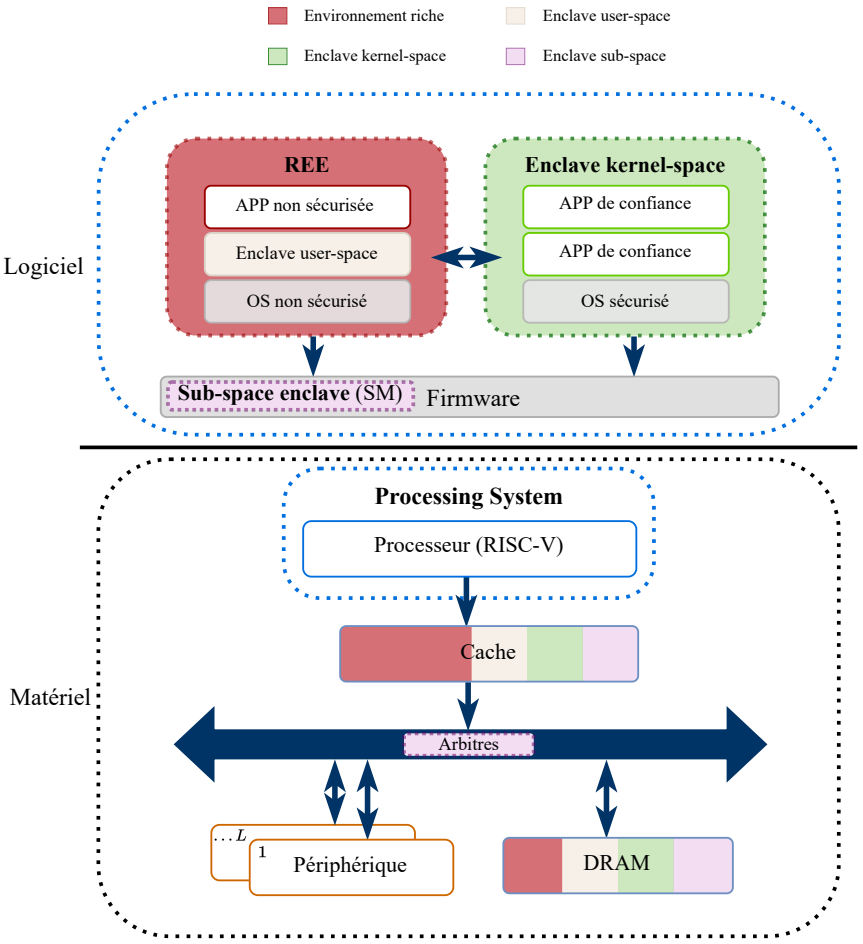


FIGURE 2.6 – Architecture matérielle et logicielle de la solution CURE.

moniteur de sécurité gère tout le cycle de vie des enclaves : leurs authentifications, leurs instances ainsi que leurs changements de contexte et leurs destructions.

CURE permet également d'instaurer une communication sécurisée entre les enclaves et d'autres ressources matérielles du circuit. Chaque enclave possède un identifiant unique qui lui est assigné par le moniteur de sécurité (SM) au moment de sa création. Cet identifiant est ensuite utilisé quand l'enclave effectue une communication. Le filtrage des communications est effectué par des arbitres situés au niveau du bus (identifiés en rose dans la Figure 2.6). Dans le prototype de CURE, les auteurs étendent le protocole TileLink pour transmettre cet identifiant durant les transactions CPU ou DMA aux périphériques ou à la mémoire principale. Cependant, il serait possible d'utiliser le protocole AMBA-AXI à la place et d'utiliser les signaux utilisateur pour propager l'identifiant de l'enclave. Cet identifiant est également utilisé pour un partitionnement de la mémoire cache. En effet, CURE propose des solutions de sécurité contre les attaques SCA et le partitionnement de la mémoire cache en fait partie. Celui-ci est dynamique et il n'est pas obligatoirement activé. S'il l'est, chaque case de cache est assignée à une seule enclave et un contrôle d'accès basé sur l'identifiant de l'enclave est réalisé pour assurer une isolation complète. S'il ne l'est pas, alors ce contrôle s'effectue sur une ligne de cache entière. En plus de ce partitionnement, CURE propose de vider les données sensibles appartenant à l'enclave de la mémoire cache à chaque changement de contexte. Cette opération est gérée par le moniteur de

sécurité. Ces deux opérations ont des conséquences sur le système. Les auteurs présentent dans [10], une évaluation via différents tests de performances et les surcoûts matériels liés à l'ajout de ces fonctionnalités.

Limitations

Dans le modèle de CURE, la mise en place et le respect des exigences de sécurité se font de manière logicielle. Le moniteur de sécurité est une enclave logicielle dans le firmware. Or, dans une proposition d'architecture de SoC sécurisée par conception, la sécurité doit être duale. Elle doit être à la fois logicielle et matérielle. Dans le cas où le SoC embarquerait des accélérateurs matériels (FPGA), la partie reconfigurable du système ne serait pas prise en charge. CURE partage la même limite que l'architecture *WorldGuard*.

2.4.3 SANCTUARY

La proposition SANCTUARY [20] est une proposition académique. Elle permet l'utilisation sans contrainte des TEE avec la technologie ARM *TrustZone* sans dépendre de virtualisation.

Modèles de menaces

Le modèle de menaces de SANCTUARY est le même que celui considéré pour la technologie ARM *TrustZone*. Il s'agit d'attaques logicielles qui ciblent tous les composants logiciels liés au monde non sécurisé. Ces attaques peuvent corrompre tous les niveaux de privilèges logiciels jusqu'à l'hyperviseur de ce monde. Un adversaire peut réaliser des attaques physiques passives sur le système. Cependant, l'attaquant ne peut pas corrompre le logiciel lié au monde sécurisé et le mode moniteur. Ces composants, logiciel du monde sécurisé, bootloader et mode moniteur, sont considérés de confiance.

Les attaques physiques invasives qui modifient le matériel et les attaques par injections de fautes pendant l'exécution ne sont pas considérées. De même, les attaques DoS ne font pas partie du modèle de menaces.

Exigences de sécurité

La proposition de SANCTUARY repose tout d'abord sur des exigences de sécurité. Elles sont au nombre de six :

ES.1 : Intégrité du code et des données associées

L'intégrité du code et des données d'une application de SANCTUARY (AS) doit être préservée via de l'isolation et des procédures d'attestation.

ES.2 : Confidentialité des données

La confidentialité d'une application de SANCTUARY doit être préservée. Une voie de communication sécurisée doit être mise en place pour transmettre les données. Une isolation est nécessaire pour empêcher une entité malicieuse de s'en emparer pendant ou après l'exécution.

ES.3 : Communication sécurisée avec le monde sécurisé

Une application de SANCTUARY nécessite une voie de communication sécurisée pour communiquer avec le monde sécurisé.

ES.4 : Protection contre des applications de SANCTUARY potentiellement malicieuses

Pour considérer tous les scénarios possibles, il faut prendre en compte que les applications de SANCTUARY peuvent être malicieuses. Des mesures doivent être prévues pour les empêcher de nuire aux autres applications.

ES.5 : Partitionnement des ressources mis en place de manière matérielle

L'isolation stricte mentionnée dans l'ES.2 doit être mise en place de manière matérielle.

ES.6 : Changements logiciels minimales

La mise en place de SANCTUARY au niveau du logiciel ne doit pas nécessiter des changements trop importants. Pour cela, elle doit se baser sur les mécanismes déjà existants de la technologie ARM *TrustZone*.

Architecture

L'architecture SANCTUARY est présentée dans la Figure 2.7. Dans le monde non

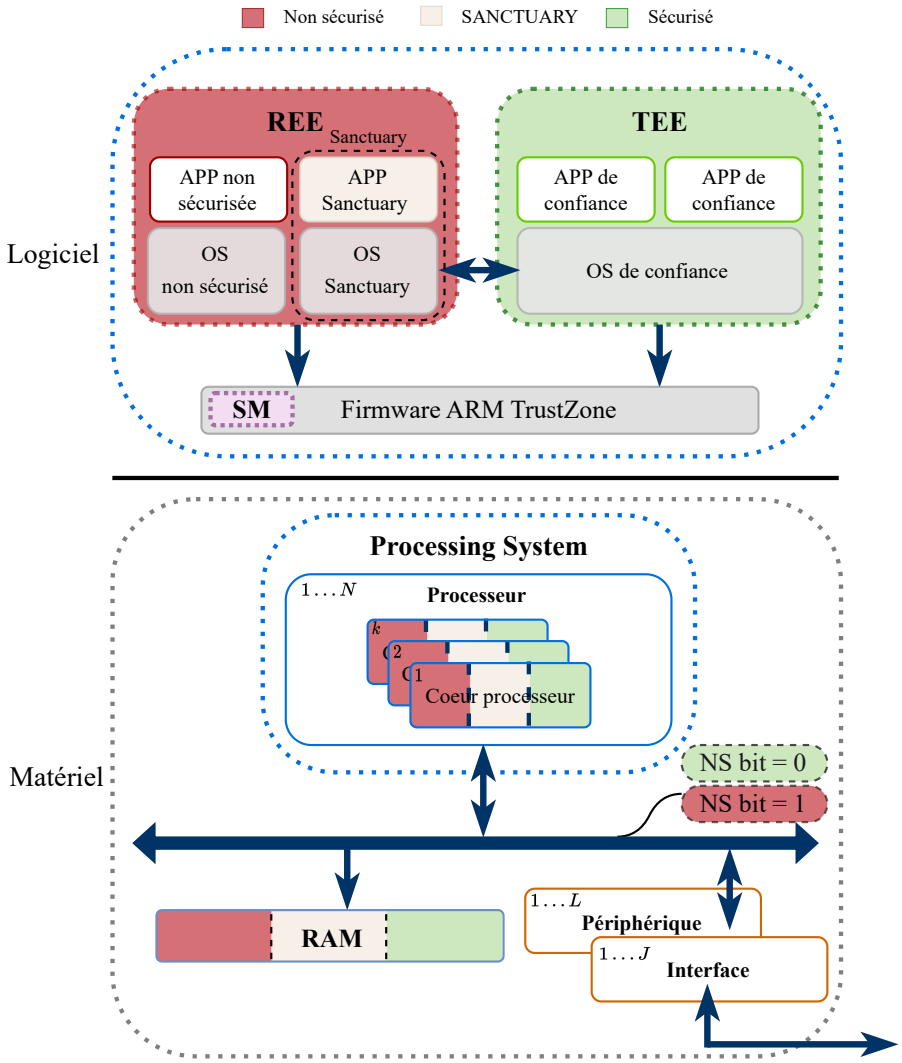


FIGURE 2.7 – Architecture logicielle et matérielle de la solution SANCTUARY.

sécurisé, identifié en rouge dans la Figure 2.7, le REE comporte une partie illustrée en pointillés noirs réservée pour SANCTUARY. Celle-ci contient, à l'intérieur de compartiments, les applications de SANCTUARY ainsi que son système d'exploitation. Ces applications sont identifiées en beige (Figure 2.7). Toutes les instances de SANCTUARY sont isolées du monde sécurisé, identifié en vert. SANCTUARY s'appuie sur les mécanismes déjà existants de la technologie ARM *TrustZone* pour mettre en place le partitionnement mémoire et isoler les données de ses applications des parties qui ne sont pas de confiance du système. Les différentes applications de SANCTUARY sont également isolées les unes des autres et exécutées indépendamment. À la fin du cycle de vie d'une application SANCTUARY, la mémoire cache est invalidée pour éviter la fuite de données. SANCTUARY utilise des identifiants assignés à chaque cœur de CPU pour filtrer les accès à la mémoire et mettre en place le partitionnement. Un contrôleur mémoire de la technologie ARM *TrustZone* est chargé du filtrage des communications. L'objectif de SANCTUARY est d'exécuter des applications sensibles à l'intérieur de compartiments isolés dans le monde non sécurisé de la technologie ARM *TrustZone*. Cela permet de réduire la consommation de ressources liées à la TEE et aux applications sécurisées qui s'exécutent dans le monde de confiance.

Limites

Le modèle proposé par SANCTUARY se concentre principalement sur le logiciel et ne couvre pas les attaques basées sur le matériel. SANCTUARY ne considère pas non plus de partie ressources logiques (FPGA) donc aucune sécurité n'est fournie pour une architecture intégrant un FPGA. SANCTUARY se base sur des composants matériels déjà existants pour effectuer son partitionnement mémoire. Aucune solution matérielle supplémentaire n'est apportée.

De plus, SANCTUARY est basé sur la technologie ARM *TrustZone* et peut être vulnérable aux attaques qui visent cette technologie. Notamment, les travaux dans [13] avec l'utilisation du DVFS pour réaliser des attaques par canaux cachés. Mais également ceux présentés dans [40], qui utilisent un cheval de Troie matériel contenu dans une IP matérielle qui permet d'outrepasser le démarrage sécurisé via un accès par le port ACP.

SANCTUARY est également limitée en ce qui concerne le nombre de domaines sécurisés avec un seul monde sécurisé et des compartiments isolés dans le monde non sécurisé.

2.4.4 TEEOD (*Trusted Execution Environments On-Demand*)

TEEOD est une proposition académique [67] d'architecture de SoC hétérogène qui intègre sur demande des environnements d'exécution de confiance dans la logique programmable du système.

Modèle de menaces

Le modèle de menaces de TEEOD est centré sur les attaquants qui ciblent le TEE [22] comme pour l'architecture HECTOR-V. Un attaquant peut prendre le contrôle du REE ainsi que des interfaces des composants du TEE. TEEOD considère également que les applications de confiance peuvent être, en réalité, malicieuses et peuvent essayer de voler des données d'autres applications de confiance ou d'attaquer la plateforme.

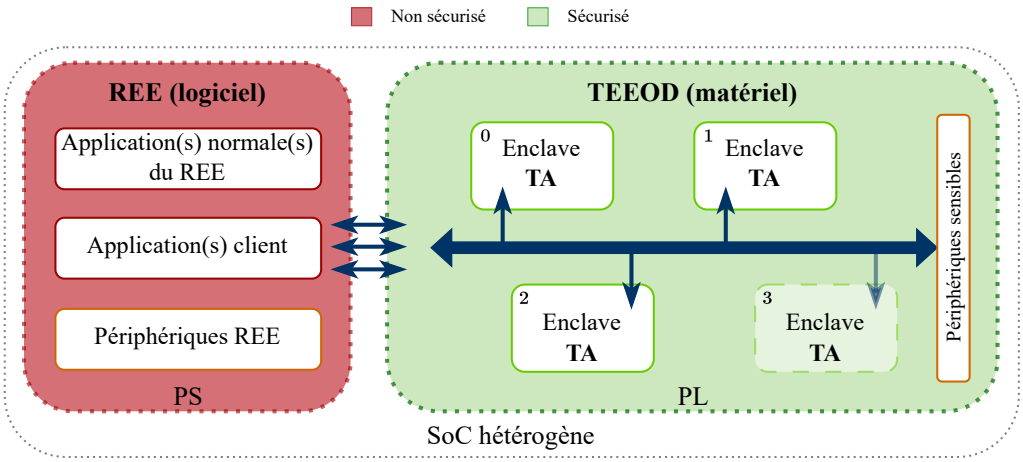


FIGURE 2.8 – Architecture logicielle et matérielle de la solution TEEOD.

Exigences de sécurité

La proposition de TEEOD repose tout d'abord sur des principes de sécurité qui guident la conception de l'architecture. Ces exigences se concentrent sur les responsabilités, les niveaux de privilèges et les interactions entre les différents composants et parties du système. Les exigences de sécurité de TEEOD sont les suivantes :

ES.1 : Isolation de l'environnement d'exécution riche :

TEEOD doit garantir que l'environnement d'exécution riche, exécuté dans la partie système, soit isolé du reste du système. Une application malicieuse de ce dernier doit être empêchée d'accéder et de compromettre les applications de confiance du TEE.

ES.2 : Isolation des applications de confiance :

TEEOD doit garantir, si plusieurs applications de confiance s'exécutent dans la partie TEE, qu'elles soient isolées entre elles. Cette spécification de sécurité permet de garantir qu'aucune application de confiance malicieuse accède et corrompe une autre application de confiance.

ES.3 : Contrôle des applications :

TEEOD doit approuver et effectuer les modifications des applications de confiance seulement et seulement si la demande est effectuée par une source de confiance.

ES.4 : Espace de stockage sécurisé :

TEEOD doit fournir un espace de stockage sécurisé où l'intégrité et la confidentialité des données liées à une application de confiance ou à l'environnement d'exécution de confiance sont garanties. Cette spécification permet aux applications de confiance de stocker des secrets ou des données critiques dans une mémoire sécurisée.

ES.5 : Accès sécurisé aux périphériques :

TEEOD doit permettre aux applications de communiquer en toute confiance avec les composants du SoC. Cette spécification garantie la sécurité des échanges entre les différentes entités.

ES.6 : Confidentialité des données :

TEEOD doit disposer d'algorithmes de cryptographie comme l'AES, le RSA ou des ECC (*Elliptic Curve Cryptography*). TEEOD doit également embarquer des primitives cryptographiques comme la génération de nombres aléatoires (RNG).

Architecture

Dans TEEOD, l'environnement d'exécution de confiance est placé dans la partie logique programmable (FPGA). La partie système (CPU) ne contient plus que l'environnement d'exécution riche. Les enclaves contiennent des processeurs softcore. Elles sont routées de manière permanente et sont activées selon les besoins, comme montré sur la Figure 2.8 avec l'enclave n°3 désactivée. Lorsqu'une enclave a fini d'être utilisée par une application de confiance, sa mémoire est vidée, une remise à zéro complète du processeur softcore est effectuée, les états de la micro-architecture sont vidés et l'enclave est déclarée à nouveau disponible.

TEEOD limite au maximum le partage de ressources matérielles ou logicielles entre les parties de confiance et celles non sécurisées du système. Par exemple, la partie qui n'est pas de confiance ne peut pas communiquer directement avec les applications de confiance. Elle doit passer par un module spécifique pour effectuer la communication.

Limitations

TEEOD permet d'intégrer toutes les ressources sécurisées dans le FPGA et de limiter la partie qui n'est pas de confiance dans le processeur câblé, évitant ainsi le partage de composants entre ces deux parties. Cependant, d'après les résultats obtenus, il n'est pas possible de faire fonctionner plus de quatre enclaves en même temps. Dans ces enclaves, le concepteur ne peut pas embarquer ses accélérateurs matériels, uniquement des processeurs softcore. De plus, TEEOD consomme beaucoup de ressources mémoires (blocs mémoire du FPGA) pour son fonctionnement, ce qui empêche l'utilisation de celles-ci pour un autre design.

2.4.5 Une approche décentralisée basée sur des pare-feu matériels

Il s'agit d'une proposition académique [25][26][27] qui est centrée sur la sécurisation d'architectures MPSoC-FPGA au travers du système de communication.

Modèle de menaces

Le modèle de menaces de l'approche décentralisée considère les attaques logiques. Les attaques qui nécessitent un accès physique ou par canaux auxiliaires sont exclues. Les attaques ne peuvent être effectuées qu'au travers du bus de communication et de la mémoire externes. Elles visent à détourner le comportement d'un processeur ou d'extraire des informations sensibles. Le modèle de menaces de cette approche décentralisée considère également les attaques DoS. Elles peuvent arrêter le fonctionnement du système, empêcher les communications ou créer un flux de communications de données fictives très important. Dans ce système, le FPGA est considéré de confiance ainsi que les mémoires sécurisées qui stockent les politiques de sécurité.

Architecture

L'architecture décentralisée repose sur de petits pare-feu distribués, codés matériellement, qui sont ajoutés entre chaque connexion avec un module et le bus de communication. Il existe deux types de pare-feu : local et cryptographique. Les pare-feu

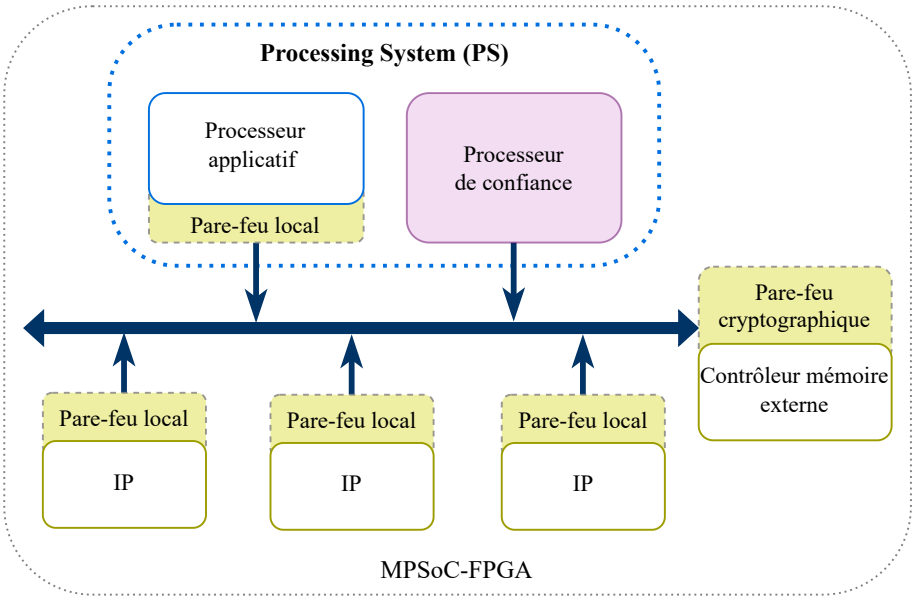


FIGURE 2.9 – Architecture matérielle de l'approche décentralisée basée sur des pare-feu matériels pour sécuriser des architectures hétérogènes multiprocesseurs (MPSoC-FPGA).

locaux sont situés entre toutes les interfaces internes et le bus de communication. Ils sont identifiés en vert-jaune dans la Figure 2.9. Le deuxième type se situe uniquement entre le contrôleur de la mémoire externe et le bus de communication. Chaque pare-feu applique des politiques de sécurité qui sont composées de plusieurs paramètres. Le premier est l'identifiant de cette politique de sécurité. Ensuite, il y a les droits d'accès à la ressource. Il existe trois possibilités : lecture seule, écriture seule ou lecture/écriture. Le troisième paramètre est le format des données de la requête, qui peut varier de huit jusqu'à trente-deux bits. Cela permet d'empêcher une requête d'écraser des données sécurisées par effets de débordement. Le paramètre suivant est uniquement disponible pour le pare-feu cryptographique. Il s'agit des modes de confidentialité et d'intégrité. Ils permettent d'activer ou de désactiver le chiffrement par blocs et les arbres de hachages. Le dernier paramètre est également uniquement disponible pour le pare-feu cryptographique. Il contient la clé de chiffrement utilisée par le bloc de chiffrement.

Les pare-feu locaux surveillent toutes les communications qui leur sont destinées en utilisant les trois premiers paramètres décrits ci-dessus. Pour chaque communication, suivant le type de requête, celle-ci est vérifiée avant d'être envoyée sur le bus ou avant d'être transmise à l'IP. Cela permet d'empêcher une requête malicieuse de transiter sur le bus. Si une violation est détectée en fonction des politiques de sécurité, le pare-feu se débarrasse des données. Le pare-feu cryptographique a pour tâche de surveiller les communications entre les IP internes et la mémoire externe. Il est chargé de la protection de la mémoire pour la confidentialité et l'intégrité. Pour les deux types de pare-feu, si une attaque est détectée, une mise à jour de sécurité est déclenchée par le processeur sécurisé (en rose dans la Figure 2.9). Cette mise à jour entraîne une modification de la politique de sécurité du pare-feu. Si cette mise à jour ne suffit pas, un redémarrage du système peut être ordonné.

Limites

Le modèle de menaces de cette architecture ne considère pas les attaques par canaux auxiliaires. Il ne prend pas non plus en compte les menaces internes au SoC. Or, dans le cycle de conception d'un SoC, ces menaces sont bien réelles et multiples comme cela est décrit dans le premier chapitre de cette thèse. Bien que le FPGA est considéré comme fiable, il est possible qu'il embarque de la logique malicieuse qui attaquera le système par la suite. De plus, comme pour TEEOD, l'approche décentralisée repose uniquement sur des solutions de sécurité matérielle.

2.4.6 E-IIPS (*Extended Infrastructure IP for SoC security*)

E-IIPS [11] est une proposition académique qui vise la mise en place de politiques de sécurité dans les conceptions d'un SoC via un framework architectural générique et flexible.

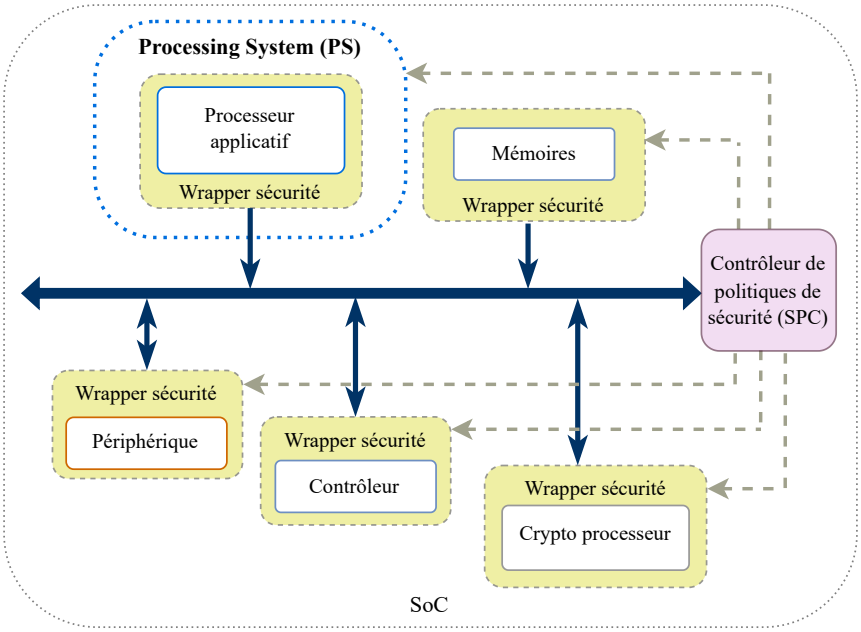


FIGURE 2.10 – Architecture matérielle de la solution E-IIPS.

Modèle de menaces

Le modèle de menaces d'E-IIPS se concentre plus sur les menaces lors de la conception des SoC. Il prend en compte les attaques logicielles à distance ou via les interfaces du SoC. Le modèle de menaces exclut les attaques via des IP malicieuses, les IP sont automatiquement considérées de confiance. Les IP ont été soit développées par le concepteur, soit elles ont été vérifiées pour éliminer la possibilité qu'elles contiennent un cheval de Troie ou une autre modification malicieuse.

Exigences de sécurité

Le framework E-IIPS se base sur des exigences de sécurité pour garantir la sécurité du système. Elles sont au nombre de quatre :

ES.1 : Contrôle d'accès :

Pendant l'exécution, une entité matérielle ou logicielle doit faire l'objet d'un contrôle d'accès systématique pour accéder à une ressource.

ES.2 : Flux d'informations :

La politique de sécurité du flux d'informations empêche qu'à certains moments de l'exécution certains secrets soient déduits sans qu'il n'y ait eu accès direct.

ES.3 : Vivacité :

Cette politique de sécurité spécifie que le système doit fonctionner normalement sans temps mort. Il ne doit pas se retrouver dans une impasse ou être verrouillé.

ES.4 : *Time-Of-Check Time Of Use* (TOCTOU) :

Si une entité a le droit d'accéder à une ressource à un certain moment, cette politique de sécurité garantie que l'entité qui y accède est bien celle autorisée à ce même moment. Aucun accès illégal ne peut être réalisé.

Architecture

Le framework E-IIPS est centré autour de deux modules. Le premier, identifié en rose dans la Figure 2.10, est le contrôleur de politiques de sécurité (*Security Policy Controller* SPC). Il s'agit d'un microcontrôleur softcore qui va contenir les politiques de sécurité du système. Les concepteurs de SoC peuvent programmer des politiques de sécurité dans ce modèle via des modules firmware.

Les deuxièmes modules sont identifiés en vert-jaune dans la Figure 2.10. Il s'agit de contrôleurs qui englobent chaque IP matérielle, appelés "wrappers de sécurité". Ils permettent de communiquer avec l'E-IIPS. Les wrappers ne remontent que les informations de sécurité pertinentes au contrôleur central. Le type d'événements transmis dépend du type d'IP matérielle à laquelle les wrappers sont rattachés. Les événements critiques ne seront pas les mêmes s'il s'agit d'une mémoire, de processeurs, de contrôleurs ou d'une IP de communication. C'est ensuite le contrôleur central qui analyse toutes les informations des wrappers. En cas d'attaque détectée ou d'activité suspecte, E-IIPS peut choisir d'envoyer un signal de désactivation à l'IP pour bloquer le fonctionnement de celle-ci.

De même, si pendant un certain laps de temps (prédéfini), une IP avec une interface maître fait trop de requêtes illégitimes (supérieures à un certain seuil), l'IP est désactivée.

Limites

E-IIPS doit être entièrement paramétrée par l'équipe en charge de l'intégration pour le SoC. Elle doit implémenter les politiques de sécurité et s'occuper de leurs vérifications. De plus, l'équipe qui est chargée de la conception des IP doit ajouter les wrappers de sécurité à chaque IP. Malgré le fait qu'un modèle soit fourni, l'équipe doit identifier les événements critiques de sécurité et implémenter le wrapper en conséquence.

2.4.7 ProMiSE (*PROgrammable hardware Monitor for Secure Execution*)

ProMiSE est un framework qui calcule un score de confiance pour chaque entité présente dans un système sur puce. Il s'applique aux architectures de confiance zéro (ZTA

(*Zero Trust Architecture*)). La particularité de ces architectures est que les droits d'accès d'un composant au réseau ne sont pas statiques mais dynamiques. Ils sont calculés en fonction du comportement du composant au cours de l'exécution.

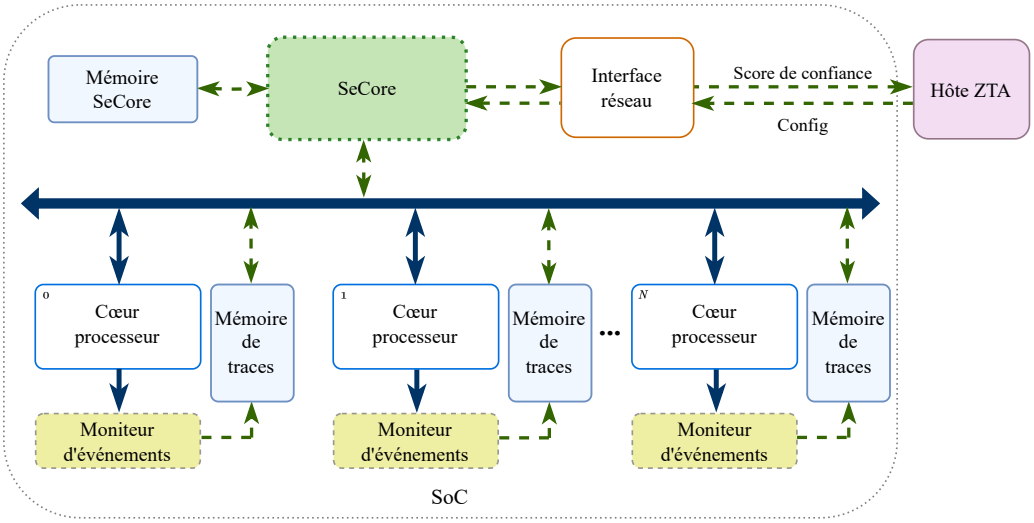


FIGURE 2.11 – Architecture matérielle de la solution ProMiSE.

Modèle de menaces

Le premier vecteur d'attaques considéré pour ProMiSE est l'attaque par rançongiciel. Les attaques de rançongiciel utilisent un logiciel malveillant qui chiffre des fichiers ou vole des données à l'utilisateur. L'attaquant va ensuite contraindre la victime à payer une rançon.

La deuxième attaque considérée est l'attaque RoP (*Return-oriented Programming*). Ces attaques utilisent un dépassement de pile qui va rediriger l'exécution du programme vers un code de l'attaquant et permet ainsi d'outrepasser des mécanismes de protection.

Le dernier vecteur d'attaques est l'attaque par analyse de canaux auxiliaires sur la microarchitecture du type *FLUSH+RELOAD*.

Architecture

ProMiSE permet de calculer le score de sécurité des composants d'une architecture ZTA. Pour cela, ProMiSE se base sur des petits modules matériels appelés "Moniteur d'événements" identifiés en vert-jaune dans la Figure 2.11. Ces moniteurs surveillent en permanence l'exécution des cœurs auxquels ils sont reliés : requêtes mémoires, compteur du programme, instructions exécutées. Chacun de ces événements est enregistré dans la mémoire des traces propres à chaque moniteur d'événements. SeCore le coprocesseur sécurisé, identifié en vert dans la Figure 2.11, lit ensuite les données dans les mémoires de traces. Il exécute des TEM (*Threat Estimation Models*) qui sont contenus dans la mémoire SeCore. En fonction des réponses des TEM, un score de confiance est calculé, il est compris entre 0 (aucune confiance) et 1 (confiance totale). L'hôte ZTA utilise ce score de confiance pour reconfigurer les permissions d'accès au réseau de chaque composant.

Limites

Les détections d'attaques sont réalisées via les calculs des TEM qui sont des sorties de réseaux de neurones. Ces réseaux utilisent beaucoup de ressources matérielles ce qui n'est pas possible pour des systèmes embarqués où la contrainte sur les ressources est élevée. L'architecture n'inclut pas non plus de ressources logiques (FPGA) ni ne propose de domaines d'exécution de confiance.

2.4.8 *Pro-active policing and policy enforcement architecture for securing MPSoCs*

Cette architecture, présentée dans [43], se focalise sur la sécurité des communications au sein d'un MPSoC. Elle propose une solution pour surveiller toutes les communications au niveau des interfaces esclaves, détecter une attaque et prendre des mesures si nécessaire.

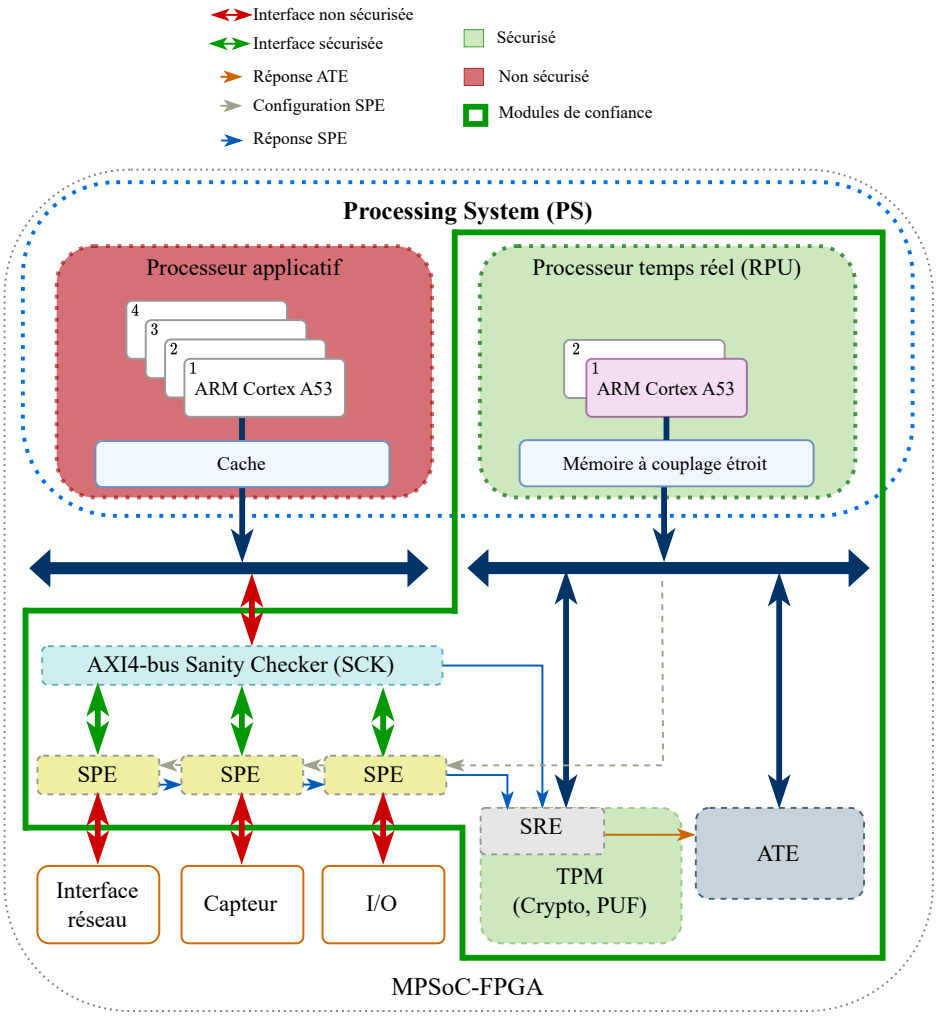


FIGURE 2.12 – Architecture matérielle de la solution *pro-active policing and policy enforcement* appliquée à une plateforme MPSoC Zynq UltraScale+ d'AMD.

Modèle de menaces

Le modèle de menaces n'est pas explicitement défini, il dépend de l'architecture à laquelle on veut appliquer cette solution. Pour une architecture donnée, une évaluation

des risques est réalisée. En fonction de cette évaluation, une modélisation des menaces est réalisée. Ainsi, le système peut identifier les vulnérabilités et les points critiques pour sa sécurité. Après ces deux premières étapes, les solutions et politiques de sécurité sont déterminées et mises en place. Dans [43], les auteurs clarifient qu'ils visent principalement les menaces liées aux ressources (logicielles ou matérielles) appartenant à un tiers et qui ont été ajoutées pendant la conception. Ils précisent également que leur architecture fournit une protection contre les attaques matérielles physiques passives comme les attaques par analyse de canaux auxiliaires.

Toute la partie du système entourée en vert dans la Figure 2.12 est considérée de confiance. Plus précisément, cela inclut : tous les composants ajoutés pour améliorer la sécurité du système (SPE, SRE, SCK, ATE, TPM), le processeur temps réel avec sa mémoire, le système d'interconnexion de celui-ci ainsi que son bus de communication. En revanche, toute la partie du processeur applicatif est considérée comme non sécurisée. Le système utilise la technologie ARM *TrustZone* pour faire cette séparation.

Architecture

L'architecture *pro-active policing* est une solution logicielle et matérielle centrée sur le bus de communication du SoC. Elle est basée sur des blocs matériels qui surveillent les communications au niveau des interfaces esclaves et effectuent des contrôles en permanence. Ces blocs détectent si une attaque est en cours et transmettent l'information aux blocs logiciels. Le logiciel prend alors des mesures proactives.

Les blocs matériels sont au nombre de quatre : deux sont implémentés sur mesure et deux sont des solutions déjà intégrées au MPSoC d'AMD. Les deux premiers sont le SPE pour *Security Policy Engine* et SCK pour *bus Sanity ChecKer*. Le bloc SPE, identifié en vert-jaune dans la Figure 2.12, est responsable d'analyser toutes les transactions destinées à l'interface de communication esclave à laquelle il est relié. Pour cela, il dispose d'une table de permissions avec les droits pour chaque interface maître. Il contient également un module appelé bloc de décision qui compare ces droits avec la requête entrante. Le SPE permet d'isoler physiquement l'interface de communication de l'entité à laquelle il est relié du bus de communication. Pendant tout le processus de vérification des droits, aucun signal du bus de communication n'est transmis au composant sous-jacent. Comme montré dans la Figure 2.12 avec les flèches en pointillés kaki, les tables de permissions sont reconfigurables par une entité de confiance dans le processeur temps réel (RPU, identifiée en rose dans la Figure 2.12).

Le bloc SCK, identifié en turquoise dans la Figure 2.12, est déployé entre chaque SPE et le bus de communication du processeur applicatif. Il s'agit d'un bloc qui permet de détecter les attaques qui visent la modification du bit de sécurité de la technologie ARM *TrustZone* ou des signaux de réponse (*XRESP*) de l'AXI (voir Chapitre 1 [12]). Il réalise cette détection en comparant les états des signaux entre la sortie du SPE et les signaux qui lui sont transmis en entrée. Ce bloc assure également la fonction de désactivation ou d'activation des communications avec le périphérique. En cas de détection d'une attaque, le SCK informe le module SRE (*Security Response Engine*).

Le bloc SRE, identifié en gris dans la Figure 2.12, est en charge de répondre aux violations des politiques de sécurité qui ont été détectées par les modules SCK et SPE. Il initie et fait appliquer les mesures proactives. Pour cela, le bloc SRE utilise le système d'interruptions déjà intégré au MPSoC Zynq UltraScale+ d'AMD. Le SRE utilise les interruptions pour détecter les violations identifiées avec les flèches bleu roi dans la Figure 2.12. Ensuite, il utilise une application de confiance dans le processeur

temps réel, illustrée en rose dans la Figure 2.12, pour exécuter les réponses proactives logicielles. Ses réponses sont l'activation des interruptions nécessaires, la suspension de l'activité du RPU et l'exécution de routines d'interruptions. Identifié en gris bleu dans la Figure 2.12, le bloc ATE *Anti-Tamper Engine* se charge de mettre en place ces routines d'interruption au niveau du système. Elles comprennent : la suppression des clés de chiffrement stockées, la désactivation de services cryptographiques, la désactivation des interfaces compromises, l'initiation d'un verrouillage sécurisé du système et l'initiation d'une remise à zéro du système.

Limites

Tout d'abord, cette architecture est présentée pour un MPSoC Zynq UltraScale+ spécifique de chez AMD. Les blocs ATE et SRE utilisent le système d'interruptions de ce MPSoC, un changement de carte entraînerait de nouveaux efforts de conception. Le modèle de menaces dépend du résultat de l'évaluation des risques effectuée sur l'architecture concernée. Des compétences spécifiques sont nécessaires pour ensuite programmer et implanter les politiques de sécurité dans le système. Cela va de nouveau demander des efforts de conception considérables. Cette architecture utilise également la technologie ARM *TrustZone*, donc le nombre de domaines sécurisés se retrouve limité à un seul. Enfin, le filtrage des communications se fait au niveau des interfaces esclaves du système, ce qui n'empêche pas des requêtes illégitimes de circuler sur le bus de communication et la réalisation possible d'attaques de type DoS.

TABLE 2.1 – Comparaison des architectures sécurisées de SoC présentées dans les parties ci-dessus (Sections 2.3 et 2.4 entières). Les symboles indiquent : ○ pas supporté, ◐ supporté de manière limitée, et ● supporté entièrement.

Architecture	Type de processeurs	Modèle de menaces	Domaines sécurisés	Protections pour le bus	Système de pénalités dynamiques	Type de contrôleurs de communication	IP matérielles de confiance	Protections pour la mémoire cache	Protections contre les attaques DoS
Technologie ARM *TrustZone* [4]	ARM	Attaques logicielles uniquement. Attaques matérielles exclues.	1	◐	○	x	◐	●	○
WorldGuard SiFive [79]	SiFive RISC-V	Attaques logicielles uniquement. Attaques SCA et physiques exclues. Attaques par élévation de privilèges. Pas de protection et isolation au sein d'un même monde. Bootloaders et SM de confiance.	N mondes	●	○	Mixte[2]	○	◐	○
HECTOR-V [64]	RISC-V	Centré sur le TEE. Exploitations de vulnérabilités architecturales. Attaque via l'interface de communication TEE et REE. Attaques SCA, sur les périphériques.	1	●	○	Mixte [2]	○	●	●
CURE [10]	RISC-V	Attaques logicielles uniquement visant le logiciel ou l'OS. Accès illégitimes aux mémoires et périphériques. Attaques DMA. Attaques matérielles et DoS exclues. La TCB est de confiance.	3 types d'enclaves	●	○	Distribué	○	●	○

Architecture	Type de processeurs	Modèle de menaces	Domaines sécurisés	Protections pour le bus	Système de pénalités dynamiques	Type de contrôleurs de communication	IP matérielles de confiance	Protections pour la mémoire cache	Protections contre les attaques DoS
SANCTUARY [20]	ARM	Attaques logicielles uniquement visant le monde non sécurisé. Pas d'attaques DoS. Le logiciel du monde sécurisé de confiance.	2	○	○	x	○	◐	○
TEEOD [67]	ARM	Centré sur le TEE. Attaques via le REE et les interfaces du TEE. Applications de confiance malicieuses.	1	●	○	Centralisé	○	○	○
Approche décentralisée [27]	MicroBlaze	Centré sur les attaques logiques. Attaques SCA et physiques exclues. Attaques effectuées uniquement via la mémoire et le bus externe. Attaques DoS. FPGA de confiance.	0	●	○	Distribué	○	○	◐
E-IIPS [11]	DLX	Centré sur la conception du SoC. Attaques logicielles à distance ou via les interfaces. Les IP sont de confiance.	0	●	○	Mixte[2]	○	○	●
ProMiSE [81]	RISC-V	Attaques par rançonlogiciel. Attaques RoP. Attaques SCA et microarchitecture.	0	◐	●	x	○	●	●
Policy [43]	ARM	Dépend de l'évaluation des risques. Menaces principalement liées à l'utilisation de ressources tierces. Attaques physiques passives SCA. Processeur RPU et tous les blocs ajoutées de confiance.	1 (ARM TZ)	●	○	Distribué	◐	○	◐

Architecture	Type de processeurs	Modèle de menaces	Domaines sécurisés	Protections pour le bus	Système de pénalités dynamiques	Type de contrôleurs de communication	IP matérielles de confiance	Protections pour la mémoire cache	Protections contre les attaques DoS
TrustSoC [61]	ARM	Attaques à distance uniquement. Attaques physiques et DoS exclues. Accès illégitimes aux mémoires et périphériques. Attaques SCA Fondeur et logiciel de CAO de confiance.	N mondes	●	○	Distribué	●	●	○
RTrustSoC [59]	ARM	Attaques à distance uniquement. Attaques physiques exclues. Accès illégitimes aux mémoires et périphériques. Attaques SCA et DoS Fondeur et logiciel de CAO de confiance.	N mondes	●	●	Distribué	●	●	●
TrustSoC-V [58]	RISC-V	Attaques à distance uniquement. Attaques physiques exclues. Accès illégitimes aux mémoires et périphériques. Attaques SCA et DoS Fondeur et logiciel de CAO de confiance.	N mondes	●	○	Distribué avec un prototype de bus de communication sécurisé	●	●	●

2.5 Comparaisons et synthèse

Le Tableau 2.1 compare toutes les architectures sécurisées présentées ci-dessus (Sections 2.3 et 2.4 entières) en fonction des paramètres suivants : type de processeur, modèle de menaces, nombre de domaines sécurisés, protections pour le bus de communication, type de contrôleurs de communication, système de pénalités dynamiques pour les communications, protections pour la mémoire cache et contre les attaques DoS. Outre les propositions de la littérature présentées dans ce chapitre, les trois dernières lignes du Tableau 2.1 présentent les paramètres de comparaison avec les trois versions de l'architecture sécurisée proposée dans cette thèse qui sont *TrustSoC*, *RTrustSoc* et *TrustSoC-V*.

En regardant la comparaison entre les différentes architectures sécurisées de l'état de l'art, nous remarquons très vite (à part CURE, SANCTUARY et *WorldGuard*) que ces architectures ne proposent qu'au plus un seul domaine sécurisé. Cela signifie que les applications ou IP matérielles de confiance doivent partager le même espace sécurisé ce qui peut poser problème selon le contexte d'utilisation.

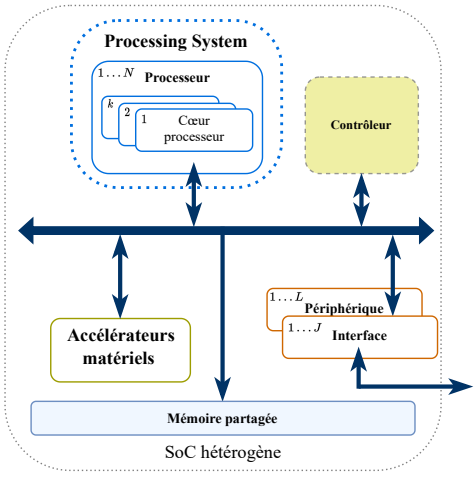
Ensuite, nous remarquons que la plupart de ces architectures disposent de contrôleurs de communication. En effet, comme nous l'avons montré dans le premier chapitre de cette thèse, le bus de communication est un point particulièrement sensible du système. Il existe trois types d'architecture pour ces contrôleurs : centralisée, distribuée et mixte. Ces différentes architectures sont présentées dans la Figure 2.13.

Dans l'architecture centralisée (TEEOD [67], [21], [23]), Figure 2.13 (A) un seul gros contrôleur analyse toutes les transactions du système. La taille de ce composant est assez importante, car il contient toutes les politiques de sécurité ainsi que la logique pour réaliser le contrôle. De plus, ce type de contrôleur augmente de manière significative la latence du système : chaque requête doit être analysée par ce dernier avant d'être transmise au destinataire.

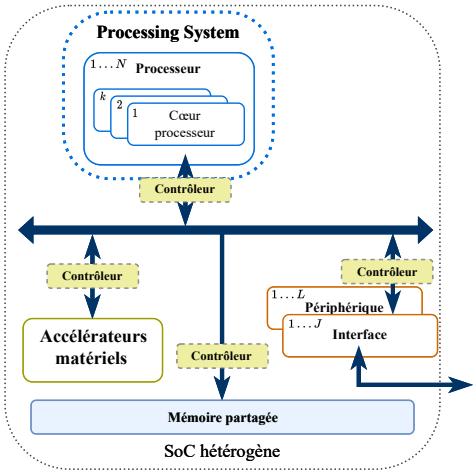
Le deuxième type (approche décentralisée [27], *Policy* [43]), distribué, ajoute un contrôleur de communication entre chaque IP matérielle du système et le bus de communication. Cette architecture est présentée dans la Figure 2.13 (B). L'ajout multiple de contrôleurs permet de diminuer la latence induite par le contrôle par rapport au premier type d'architecture. Mais en contrepartie, il y a une duplication des ressources puisqu'il y a un ajout de nombreux petits contrôleurs de communication. Un autre avantage de l'architecture distribuée concerne la sécurité : si les contrôleurs filtrent les communications dès l'émission des requêtes, ils peuvent empêcher une entité malveillante d'envoyer des communications malicieuses sur le bus. Cela permet notamment d'empêcher les attaques de type DoS.

Enfin, le dernier type est l'architecture mixte (HECTOR-V [64], E-IIPS [11]), Figure 2.13 (C), qui est basée sur les deux premières. Elle dispose de contrôleurs distribués plus petits et d'un contrôleur central moins complexe que la première architecture (centralisée). Le contrôleur central se charge de la configuration des politiques de sécurité, et dans certains cas, du traitement des événements remontés par les petits contrôleurs (c'est le cas avec E-IIPS). Cette architecture présente l'inconvénient de la duplication des ressources, mais permet de réduire la latence par rapport à une architecture centralisée et offre une meilleure sécurité comme pour l'architecture distribuée.

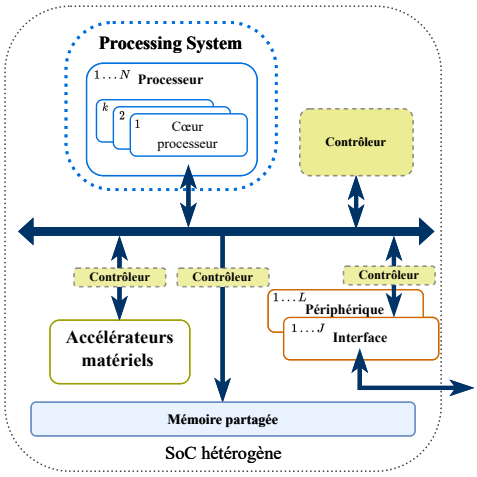
2. Mixte signifie que l'architecture présente de petits contrôleurs de communication distribués et un centralisé.



(A) Architecture avec contrôleur de communication de type centralisé.



(B) Architecture avec contrôleur de communication de type distribué.



(C) Architecture avec contrôleur de communication de type mixte.

FIGURE 2.13 – Les différentes architectures de contrôleurs de communication issues de la littérature.

En plus des contrôleurs de communication, l'architecture peut embarquer un système de pénalités dynamiques. Ces pénalités empêchent une entité, si son comportement est jugé malveillant, de faire des requêtes jusqu'à la désactivation total de sa communication avec le reste du système si elle persiste. Dans le Tableau 2.1, seule l'architecture ProMiSE possède un tel système. L'architecture E-IIPS possède également un système de pénalités pour les requêtes DMA mais non dynamiques. Si l'entité fait trop de requêtes DMA illégitimes, sa capacité de communication est désactivée. Ce système de pénalités présente un véritable avantage en cas de comportement malveillant d'une entité, l'attaque ne peut pas se propager au reste du système et la vulnérabilité est contenue.

Dans le Tableau 2.1, nous pouvons remarquer l'absence de sécurité complète pour les ressources logiques (FPGA). Cela pose problème si l'architecture du SoC-FPGA doit être sécurisée, car ce ne sont pas seulement les ressources logicielles qui doivent être protégées des attaques, mais également les accélérateurs matériels.

Nous remarquons également dans le Tableau 2.1, que seules certaines architectures

(technologie ARM *TrustZone*, HECTOR-V, CURE, ProMiSE et *WorldGuard*) proposent des solutions de sécurisation pour la mémoire cache contre les attaques à distance SCA. Ces solutions comprennent généralement un partitionnement de la mémoire cache avec un filtrage des requêtes entrantes. Comme nous l'avons vu dans le premier chapitre de cette thèse (Section 1.3.1), des protections pour la mémoire cache sont essentielles pour empêcher les attaques qui la visent et qui peuvent être généralisées depuis le FPGA.

Enfin, une dernière particularité qui apparaît dans le Tableau 2.1 est le type de processeurs utilisé pour chaque proposition. Il s'agit majoritairement de solutions utilisant des processeurs RISC-V. Les processeurs ARM sont seulement utilisés pour les solution basées sur la technologie ARM *TrustZone*. Les solutions qui embarquent cette technologie sont soumises aux limitations de celle-ci. Pour les processeurs RISC-V, le problème qui se pose est l'intégration des solutions dans les plateformes existantes qui n'embarquent pas ce type de processeurs nativement. Elles peuvent donc demander un effort de conception non négligeable pour être intégrées dans les SoC-FPGA actuellement sur le marché.

2.6 Positionnement des contributions de cette thèse vis à vis de l'état de l'art

En comparant les différentes propositions issues de la littérature et de l'industrie, nous avons remarqué qu'elles étaient principalement basées sur l'utilisation de TEE avec une approche binaire : sécurisée ou non sécurisée (voir Figure 2.1). Le TEE est centré autour du logiciel même s'il est assisté par le matériel. Les ressources matérielles ne sont que très peu prises en compte dans la sécurité. Nous remarquons dans le Tableau 2.1 que très peu d'entre elles garantissent des IP matérielles de confiance. Aucune ne propose simultanément des protections contre les attaques qui visent le bus de communication avec un système de pénalités dynamiques, des protections contre les attaques de type DoS et SCA pour la mémoire cache, des protections contre les accès illégitimes aux mémoires et périphériques. Ce sont ces raisons, étendre la notion de sécurité binaire du TEE à une sécurité multiple à plusieurs niveaux et proposer une architecture intégrant simultanément toutes les protections contre les attaques (SCA, DoS, bus de communication, périphériques, mémoires), qui nous ont poussés à proposer les architectures *TrustSoC*, *RTrustSoC* et *TrustSoC-V* qui seront présentées plus en détail dans les chapitres suivants.

2.7 Conclusion du chapitre

Ce chapitre a permis d'expliquer les notions préliminaires nécessaires à la bonne compréhension de cette thèse : les environnements d'exécution de confiance et les protocoles de communication des normes AMBA-AXI et TileLink. Il a également permis de présenter les différentes propositions, académiques ou industrielles, d'architectures sécurisées de SoC hétérogènes les plus pertinentes de l'état de l'art. Nous avons présenté leurs modèles de menaces, leurs exigences de sécurité, leurs fonctionnements et leurs résultats d'implémentation. Ce chapitre a permis de présenter les principales solutions de sécurité possibles pour répondre aux attaques décrites dans le premier chapitre de cette thèse qui visent les SoC hétérogènes. Il a également permis de montrer les limites de ces propositions et de poser les fondations de notre proposition, une nouvelle architecture sécurisée de SoC hétérogène palliant à ces limites.

Chapitre 3

TrustSoC et son extension *RTrustSoC*

Résumé

Ce chapitre présente une première contribution *TrustSoC*. Il s'agit d'une architecture de SoC hétérogène sécurisée par conception basée sur des processeurs de type ARM et une extension de la technologie ARM *TrustZone*. Nous présentons ses exigences de sécurité, son architecture ainsi que son fonctionnement. Nous donnons également des résultats d'implémentations matérielles pour différentes IP matérielles sur SoC AMD Z-7010. Ce chapitre présente également une deuxième contribution *RTrustSoC* qui est une extension de *TrustSoC*. Nous détaillons l'apport de cette contribution par rapport à *TrustSoC* et présentons les résultats d'implémentation matérielle avec trois scénarios d'utilisation de *RTrustSoC* contre des attaques décrites dans le Chapitre 1.

Table des matières

3.1 Introduction au chapitre 48
3.2 Motivations 48
3.3 *TrustSoC* 48
3.3.1 Modèle de menaces 49
3.3.2 Définition des exigences de sécurité 49
3.3.3 Règles de sécurité et architecture 50
3.3.4 Fonctionnement 56
3.3.5 Résultats d'implémentation 59
3.3.6 Limites 62
3.4 *RTrustSoC* 62
3.4.1 Modèle de menaces 62
3.4.2 Définition des exigences de sécurité 63
3.4.3 Architecture et fonctionnement 64
3.4.4 Résultats d'implémentation 69
3.5 Scénarios d'utilisation de *RTrustSoC* 71
3.5.1 Attaque basée sur la mesure des temps d'accès à la mémoire cache 71
3.5.2 Attaque par escalade de privilèges via le port ACP 72
3.5.3 Attaque du démarrage sécurisé via le port ACP 74
3.6 Limitations de la proposition 76
3.7 Conclusion du chapitre 77

3.1 Introduction au chapitre

La sécurité des architectures de SoC hétérogènes est un sujet de recherche actuel. Beaucoup de propositions issues de la littérature que nous avons présentées dans le chapitre précédent datent de ces dernières années. Les trois publications HECTOR-V [64], CURE [10] et *WorldGuard* [79] ont toutes été publiées en 2021. La publication de l'architecture ProMiSE [81] est toute récente datant de 2024. Ce domaine de recherche est très actif et des points faibles ont été identifiés lors de la comparaison des différentes propositions au chapitre précédent. Ces faiblesses doivent être adressées, c'est ce que nous proposons avec *TrustSoC* et *RTrustSoC*.

Nos travaux sur les architectures de SoC sécurisées ont donné lieu à trois publications : une conférence internationale, une conférence française internationale et une revue internationale. Ces contributions sont décrites dans différentes sections de ce chapitre. La Section 3.3 présente *TrustSoC*. La Section 3.4 présente *RTrustSoC* et, enfin, la Section 3.5 présente trois scénarios d'attaques du Chapitre 1 appliqués à *RTrustSoC*.

3.2 Motivations

Notre contribution a pour objectif de :

— Proposer une solution minimale et centrée sur le bus de communication du SoC qui est un élément particulièrement vulnérable [14] du SoC hétérogène.
— Englober pour la sécurité du système tous ses éléments : processeurs, accélérateurs matériels, bus de communication, périphériques, interfaces, etc.
— Proposer des mécanismes de sécurité pour les différentes mémoires du SoC hétérogène.
— Répondre à un modèle de menaces bien défini.

3.3 *TrustSoC*

Cette section présente le concept de *TrustSoC*. Il s'agit d'une proposition d'architecture de SoC hétérogène sécurisée par conception qui repose sur un modèle de menaces précis et des exigences de sécurité. Le modèle *TrustSoC* présente de multiples caractéristiques de sécurité :

— Les composants matériels ou logiciels peuvent être assignés à différents mondes avec différents niveaux de privilèges, ce qui généralise la notion basique de sécurisé/non sécurisé des TEE comme la technologie ARM *TrustZone*.
— La mémoire cache est protégée contre les attaques qui tirent parti de l'accès partagé.
— Un ensemble de contrôleurs de communication distribués appliquent des politiques de sécurité pour mettre en œuvre un système de communication de confiance à l'intérieur du SoC.

TrustSoC est une architecture flexible et modulable qui est ajustée à la conception en fonction des besoins du concepteur. *TrustSoC* considère une architecture de SoC hétérogène qui comprend une partie *Processing System*, une partie *Programmbale Logic* (FPGA), un bus de communication, plusieurs périphériques et des mémoires partagées. Les sous-sections suivantes décrivent le modèle de menaces de *TrustSoC*, ses exigences de sécurité, son architecture ainsi que son fonctionnement. Des résultats

d'implémentation sont fournis et une sous-section comprend une discussion sur les limites de cette architecture.

3.3.1 Modèle de menaces

Les délais de commercialisation ont tendance à se raccourcir de plus en plus. En conséquence, les concepteurs n'ont plus le temps de développer entièrement chaque composant matériel ou logiciel du SoC. Ils doivent se résoudre à utiliser des blocs vendus par un tiers. Ces blocs ne font pas toujours l'objet de vérifications qui peuvent être complexes à mettre en œuvre [15][50]. Ils peuvent donc contenir des routines ou des circuits malicieux qui pourraient être utilisés pour réaliser une attaque sur le SoC. Par exemple, ces blocs peuvent essayer d'accéder à des informations sensibles d'autres applications ou IP matérielles auxquelles ils ne devraient pas avoir accès. Ces blocs peuvent aussi contenir des vulnérabilités introduites involontairement durant la conception qui peuvent être utilisées pour réaliser une attaque contre le système.

Ainsi, le modèle de menaces de *TrustSoC* se concentre sur le processus de conception d'un SoC-FPGA [44]. *TrustSoC* considère des attaques à distance uniquement, matérielles et logicielles. *TrustSoC* envisage des IP matérielles ou des applications logicielles qui auraient été ajoutées au moment de la conception du SoC. *TrustSoC* considère également les accès illégitimes et les modifications des contenus des mémoires. *TrustSoC* exclut les attaques DoS.

Dans *TrustSoC*, nous considérons que le compilateur logiciel et l'outil de synthèse sont fiables et ne peuvent pas être utilisés pour effectuer des modifications illégales de l'architecture. L'outil de synthèse est responsable de l'ajout des composants supplémentaires de chaque IP matérielle ce qui signifie que ces IP additionnelles ne peuvent pas être modifiées. Nous supposons également que le SoC et le fondeur sont considérés comme fiables et que le circuit ne peut pas être modifié comme avec l'ajout d'un cheval de Troie matériel.

3.3.2 Définition des exigences de sécurité

Avant de présenter l'architecture de *TrustSoC* ainsi que son fonctionnement, nous devons aborder ses exigences en matière de sécurité. En effet, si nous voulons proposer une architecture sécurisée par conception, nous devons fournir un ensemble de caractéristiques de sécurité essentielles. Dans *TrustSoC*, elles sont au nombre de cinq :

ES.1 : Règles de sécurité :

TrustSoC doit reposer sur des règles de sécurité précises. Ces règles doivent régir le comportement de tous les composants présents dans le système et leurs interactions. Par exemple, d'un cœur de CPU du système à un autre, en interne du FPGA mais également du CPU au FPGA. Des règles doivent aussi s'appliquer pour les mémoires, les périphériques et le système de communication. Elles doivent décrire précisément tous les comportements autorisés et non autorisés dans l'architecture *TrustSoC*.

ES.2 : Extension en un système multi-mondes sécurisé :

La proposition *TrustSoC* doit être basée sur un système multi-mondes : un monde non sécurisé et jusqu'à N mondes sécurisés (N dépendant des ressources disponibles du système). Le choix des niveaux de privilèges des différents mondes sécurisés doit

être laissé au concepteur. Ces caractéristiques permettent d'offrir plus de flexibilité au concepteur.

ES.3 : Intégration des ressources logiques (type FPGA) dans la sécurité :

Dans le chapitre précédent, l'une des faiblesses de l'état de l'art qui a été identifiée est le manque d'intégration des ressources logiques (FPGA) du SoC-FPGA dans la sécurité. L'architecture *TrustSoC* doit donc intégrer une partie matérielle contenant différentes IP matérielles associées chacune à un monde. Elles doivent être entièrement intégrées dans la sécurité du système. Des règles de sécurité (ES.1) doivent encadrer le comportement des IP matérielles.

ES.4 : Communications sécurisées à l'intérieur du SoC :

Le système de communication du SoC est un point particulièrement sensible du SoC (voir sous-Section 1.3.3). Si l'attaquant parvient à compromettre le bus de communication, il peut prendre le contrôle du système, extraire des informations sensibles, etc. Il est donc essentiel que les communications à l'intérieur du SoC soient sécurisées. *TrustSoC* doit établir cette communication sécurisée entre les différentes entités (logicielles et matérielles) du système. Pour cela, *TrustSoC* doit pouvoir différencier les requêtes en fonction de leurs émetteurs et fournir un système de vérifications de droits avec des contrôleurs de communication. *TrustSoC* ne doit pas dépendre de dispositifs de sécurité tiers et doit garantir que les IP matérielles introduites ont le comportement attendu.

ES.5 : Résistance aux attaques de type SCA basées sur des mesures de temps d'accès de la mémoire cache :

L'architecture *TrustSoC* doit embarquer des protections contre les attaques SCA à distance basées sur des mesures de temps d'accès qui visent la mémoire cache du système. Elles consistent à restreindre l'accès à la mémoire cache depuis la partie *Processing System* ou depuis la logique programmable. Cette restriction doit s'appuyer sur un partitionnement de la mémoire cache (une partition par monde) avec une identification des différentes partitions. Les partitions doivent être isolées entre elles. La communication à la mémoire cache doit faire l'objet d'une vérification. Les règles de sécurité de *TrustSoC* (ES.1) concernant la mémoire cache doivent être mises en place. Elles stipulent que les partitions sensibles de la mémoire cache doivent être vidées à chaque changement de contexte ou après utilisation par une entité sensible (RS_CPU4, RS_CPU_CPU2 et RS_CPU_CPU3, RS_FPGA_CPU1). Cette action va entraîner une dégradation des performances, mais va améliorer la sécurité du système [10][24][70].

3.3.3 Règles de sécurité et architecture

Avant de présenter l'architecture de *TrustSoC* et son fonctionnement, nous allons définir ses règles de sécurité (ES.1).

Définition des règles de sécurité de *TrustSoC*

Les règles de sécurité de *TrustSoC* ont été rédigées dans un document au début de la thèse (présenté dans l'Annexe A). Elles permettent de mettre en place l'exigence de sécurite n°1 (ES.1). Des exemples de règles de sécurité sont précisés dans le Tableau 3.1. Les règles sont définies en fonction des composants qu'elles concernent, par

TABLE 3.1 – Exemples de règles de sécurité de *TrustSoC* encadrant le fonctionnement d'un SoC-FPGA.

	Règles de sécurité
D'un cœur de CPU vers un autre cœur de CPU	**RS_CPU_CPU_1 :** Une application du monde non sécurisé d'un premier cœur ne peut pas interagir avec une application d'un monde sécurisé d'un deuxième cœur sans autorisation préalable : échanges de données, fonctions de traitement.
	RS_CPU_CPU_3 : Une application (sécurisée ou non sécurisée) d'un cœur ne peut pas avoir accès à la mémoire cache d'une autre application d'un autre cœur (pas de partage de la mémoire cache entre cœur).
Interne au FPGA	**RS_FPGA_1 :** Une IP du monde non sécurisé ne peut pas communiquer avec une IP d'un monde sécurisé sans autorisation préalable. Après le traitement l'IP du monde non sécurisé est remise à son état d'origine ou à un état initial bien défini par conception.
	RS_FPGA_2 : Une application d'un monde sécurisé d'un premier cœur ne peut pas interagir avec une application d'un monde sécurisé d'un deuxième cœur sans autorisation préalable : échanges de données, demandes de réalisation de fonctions.
Du CPU vers le FPGA	**RS_CPU_FPGA_1 :** Une application du monde non sécurisé d'un premier cœur ne peut pas interagir avec une application d'un monde sécurisé d'un deuxième cœur sans autorisation préalable : échanges de données, demandes de réalisation de fonctions.

exemple d'un cœur de CPU du système à un autre, en interne du FPGA mais également du CPU au FPGA. Ces règles sont également spécifiées pour le comportement autorisé à l'intérieur d'un cœur de CPU, du FPGA au CPU, des mémoires et des périphériques bien qu'elles ne soient pas toutes présentées dans le Tableau 3.1 (par manque de place et par souci de lisibilité). Elles permettent de régir la communication ainsi que les actions réalisables dans le système. Par exemple, il est mentionné qu'aucune communication ne peut être faite sans autorisation. Cette autorisation est délivrée par un contrôleur de communication. Les différents types de contrôleurs ainsi que leurs fonctionnements ont été introduits dans le Chapitre 2. Ces règles de sécurité permettent de fixer les limites du système et d'empêcher les attaques mentionnées dans la sous-Section 3.3.1.

Architecture

Pour illustrer le concept de *TrustSoC*, nous illustrons dans la Figure 3.1 un exemple de cette technologie appliquée à une architecture de SoC hétérogène. La partie *Processing System* inclut de 1 à N processeurs ARM, qui eux-mêmes contiennent jusqu'à k cœurs chacun. *TrustSoC* embarque une partie de logique programmable (FPGA) avec jusqu'à m IP matérielles. *TrustSoC* comprend également un bus de communication de type AMBA-AXI4-*full*[1] dans notre exemple, ainsi que des périphériques (interfaces, gestion de l'énergie, etc.) et des mémoires. *TrustSoC* est basée sur une extension de la

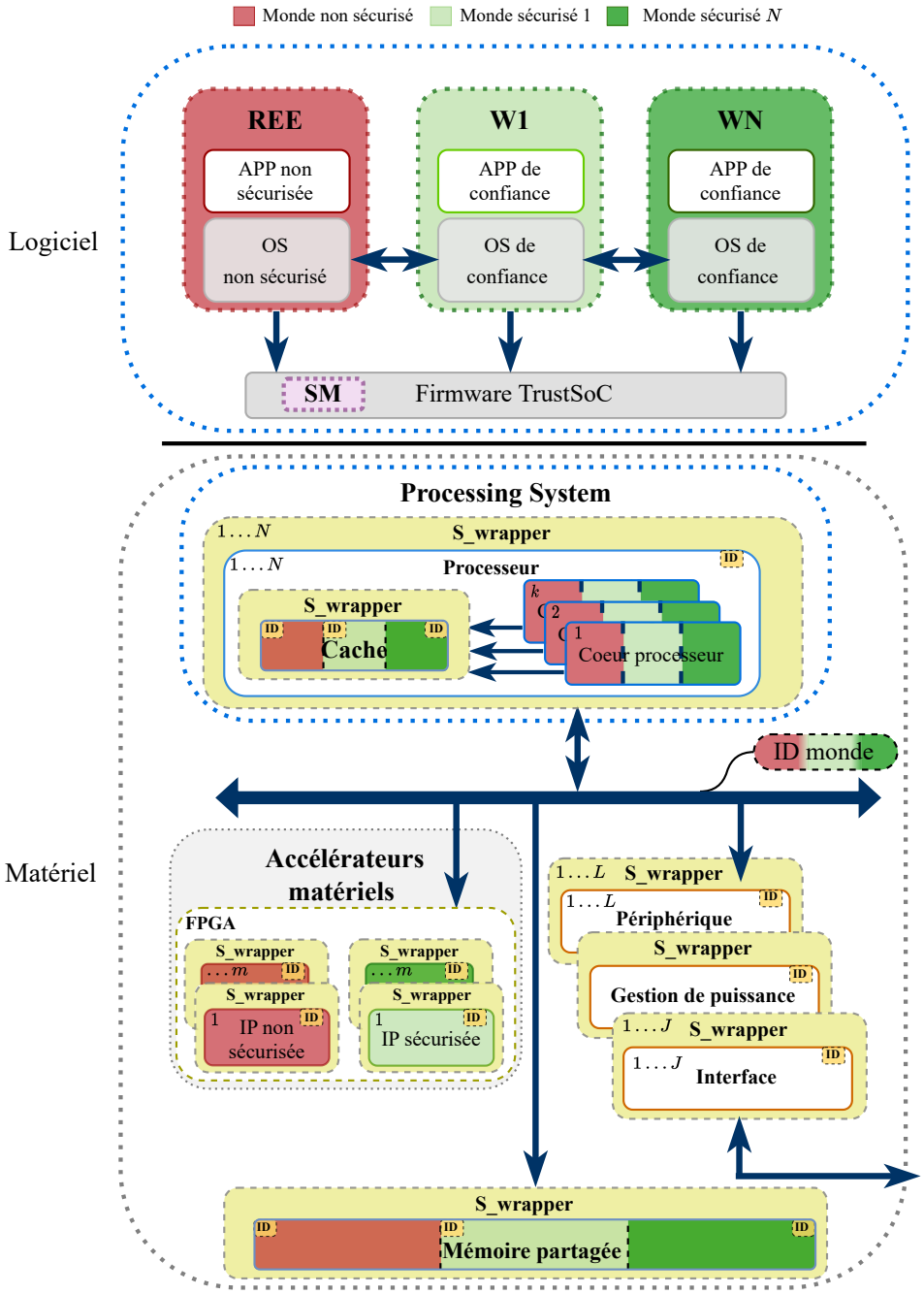


FIGURE 3.1 – Architecture du concept *TrustSoC*.

1. Pour la suite si nous mentionnons AXI4, cela désigne le protocole AXI4-*full*

technologie ARM *TrustZone* [4]. *TrustSoC* propose un monde non sécurisé (représenté en rouge foncé dans la Figure 3.1) et jusqu'à N mondes sécurisés (en fonction des ressources disponibles du système). Ces derniers sont représentés en différentes nuances de vert (Figure 3.1). Chaque cœur peut ainsi choisir d'assigner ses applications au monde non sécurisé ou à un des mondes sécurisés. Le système peut compter jusqu'à N mondes sécurisés, leur nombre maximum dépendant des ressources disponibles du système. De même, chaque IP matérielle implémentée dans la logique programmable (FPGA) est assignée à un des mondes du système. Les mémoires partagées du SoC (RAM ou mémoire cache) sont également partitionnées en fonction du nombre de mondes : une partition pour chaque monde comme représenté dans la Figure 3.1. *TrustSoC* embarque également des contrôleurs de communication matériels distribués (identifiés en vert-jaune dans la Figure 3.2) appelés *s_ wrappers* pour *security wrapper* qui sont ajoutés entre chaque interface de communication esclave du système et le bus de communication du SoC. Ainsi, aucun composant disposant d'une interface de communication esclave du système n'est connecté directement au bus de communication, il en est isolé. Les requêtes de communication passent obligatoirement par le contrôleur de communication avant d'être transmises à l'IP matérielle. L'architecture globale de ces contrôleurs est représentée dans la Figure 3.2.

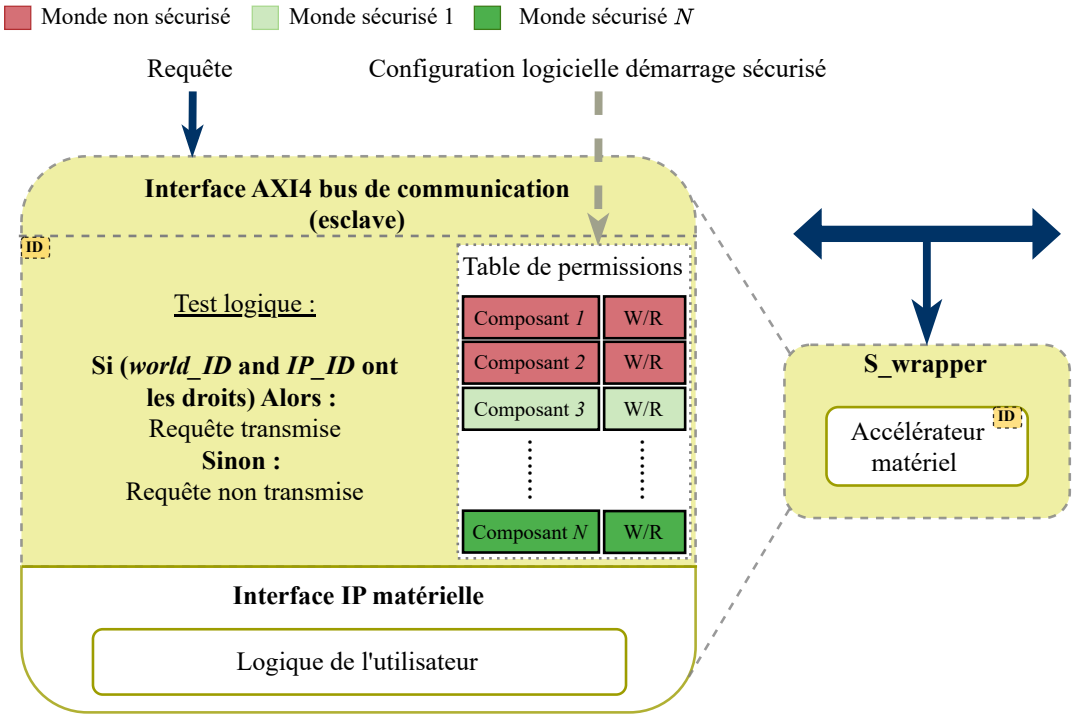


FIGURE 3.2 – Architecture des contrôleurs de communication connectés à une interface de communication esclave de *TrustSoC*.

Les contrôleurs de communication sont composés d'une interface de communication esclave de type AXI4, d'une partie de test logique et d'une interface vers l'IP matérielle. Chaque IP matérielle possède deux identifiants : un pour le monde auquel elle appartient et un pour elle-même (en jaune sur la Figure 3.2). L'identifiant de l'IP matérielle est unique et diffère pour chaque IP matérielle du système. Ces identifiants permettent de différencier les transactions entre elles. Le nombre de bits nécessaires pour coder cet identifiant, sachant que l'identifiant nul '0' est exclu, est donné par

$\lceil \log_2(\max(\text{nombre de composants})) + 1 \rceil$ où $\lceil \cdot \rceil$ est la fonction d'arrondi à l'entier supérieur. Comme pour une architecture sécurisée de l'état de l'art HECTOR-V [64], nous faisons une exception pour l'identifiant nul. La différence est que nous ne laissons passer aucune requête avec cet identifiant. HECTOR-V considère que si l'identifiant nul est utilisé cela signifie que l'IP ne requiert pas un contrôle complet et n'effectue qu'un contrôle minimal de la communication. Ainsi, dans *TrustSoC*, toutes les requêtes avec l'identifiant nul sont rejetées et il est nécessaire d'avoir un bit supplémentaire pour coder tous les identifiants du système. L'identifiant monde de *TrustSoC* est l'équivalent du signal *NS bit* (voir Chapitre 2) de la technologie ARM *TrustZone*. Ce signal est ajouté à chaque transaction pour que le système ait connaissance du monde dans lequel il opère. *TrustSoC* pouvant avoir plus de deux mondes, son encodage évolue. Il n'est plus sur un seul bit, mais sur plusieurs. Le nombre de bits nécessaires pour coder cet identifiant est donné par $\lceil \log_2(\max(\text{nombre de mondes})) \rceil$. Ces identifiants, monde et IP matérielle, sont codés matériellement et assignés à la présynthèse. Ils n'évoluent plus une fois que la synthèse est effectuée. Ils sont ajoutés à chaque communication dans le système via les signaux utilisateur du protocole de communication AMBA-AXI4. La construction des trames de ces signaux est illustrée dans la Figure 3.3. Comme précisé dans la Section 2.2.2, le protocole AXI4 nous permet d'ajouter des données supplémentaires à chaque transaction sans surcoût.

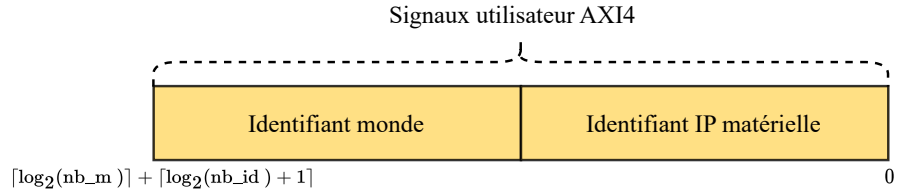


FIGURE 3.3 – Construction d'une trame des signaux utilisateur dans le cadre de *TrustSoC* où nb_m représente le nombre de mondes et nb_id le nombre de composants dans le système.

En plus de ces identifiants, les contrôleurs de communication disposent de tables de permissions qui spécifient pour chaque ressource matérielle du système, le monde auquel elle appartient ainsi que les droits d'accès de ce composant à l'IP matérielle sous-jacente. Par exemple, dans la Figure 3.2, les composants 1 et 2 appartiennent au monde non sécurisé, le composant 3 appartient au monde sécurisé 1, etc. Les droits sont codés sur deux bits : un bit pour l'écriture et un pour la lecture. Si le bit de droit lié à l'entité qui fait la demande est à '0', cela veut dire que l'entité n'a pas les droits. S'il est à '1', elle a le droit de communiquer en lecture ou en écriture avec l'IP matérielle sous-jacente. Dans *TrustSoC*, aucune condition n'est imposée sur les adresses. Si un composant a le droit de communiquer avec l'IP matérielle, il peut l'adresser entièrement sans restriction. Les tables de permissions sont codées matériellement, mais peuvent être modifiées au moment du démarrage sécurisé par une configuration sécurisée indiquée en pointillés kaki sur la Figure 3.2. Celle-ci est réalisée par une entité de confiance : un moniteur de sécurité. Après cette configuration, elles sont fixes et ne peuvent plus être modifiées. Cette reconfiguration des tables de permissions permet d'offrir plus de flexibilité au concepteur. Dans le modèle de menaces de *TrustSoC* (sous-Section 3.3.1), le contrôleur de communication est déclaré de confiance. Les identifiants et les tables de permissions d'une IP matérielle sont gérés par le contrôleur, en conséquence, ils ne peuvent pas être modifiés. La partie test logique, représentée dans la Figure 3.2, effectue le contrôle des droits. Son fonctionnement est décrit plus en détail dans la Figure 3.4 qui montre le diagramme de séquence du fonctionnement d'un contrôleur de communication de *TrustSoC*.

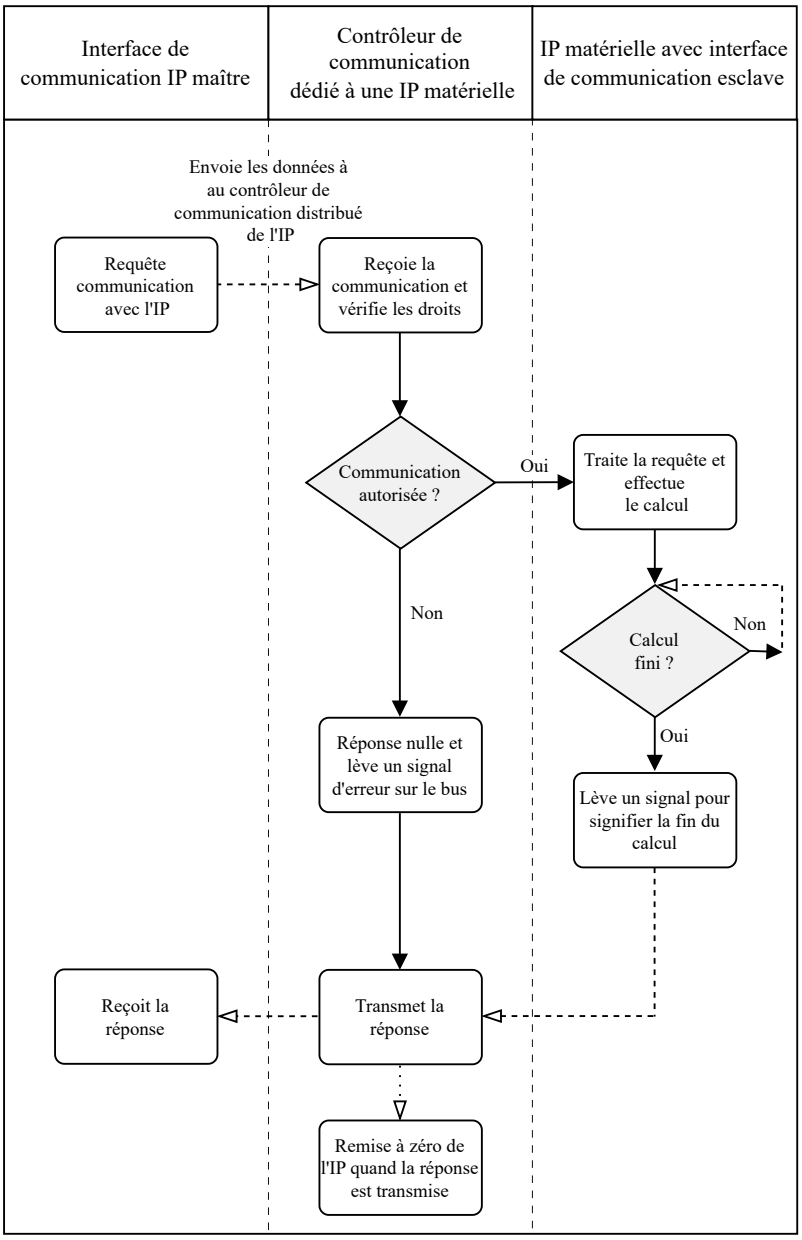


FIGURE 3.4 – Diagramme de séquence du fonctionnement des contrôleurs de communication de *TrustSoC*.

Lorsqu'une requête est reçue par un contrôleur de communication (*s_ wrapper*), celui-ci compare les identifiants inclus dans la requête avec les droits correspondant inscrits dans sa table de permissions. Si les droits sont vérifiés, la requête est transmise à l'IP matérielle sous-jacente. Cette dernière effectue la fonction demandée. Une fois le traitement terminé, elle lève un signal "fin de traitement" pour signifier qu'elle a terminé et transmet la réponse. Si, au contraire les droits ne sont pas validés, le contrôleur de communication agit comme un pare-feu. Il empêche l'entité émettrice d'accéder à l'IP matérielle. Il prend le contrôle de la communication. Il ne transmet pas la requête à l'IP matérielle et renvoie des données nulles en cas de lecture ou ne traite pas les données en cas d'écriture. En plus, en cas de transaction illégitime, comme illustré dans la Figure 3.4, une erreur est levée sur le bus de communication AXI4 via les signaux réponse *XRESP*. Le contrôleur de communication assigne la valeur

"10" qui signifie *SLVERR* sur le signal *XRESP* correspondant à l'opération demandée (écriture ou lecture).

Les contrôleurs de communication distribués de *TrustSoC* permettent également de faire respecter certaines règles de sécurité (ES.1) qui encadrent le comportement de l'IP matérielle à laquelle ils sont attachés. Ils peuvent par exemple embarquer la règle de sécurité RS_FPGA_1 (voir Tableau 3.1) qui stipule qu'une IP matérielle doit être remise à son état d'origine après le traitement. Cette étape est la dernière réalisée dans la Figure 3.4, la remise à zéro est effectuée quand la réponse a été transmise et traitée par l'interface maître.

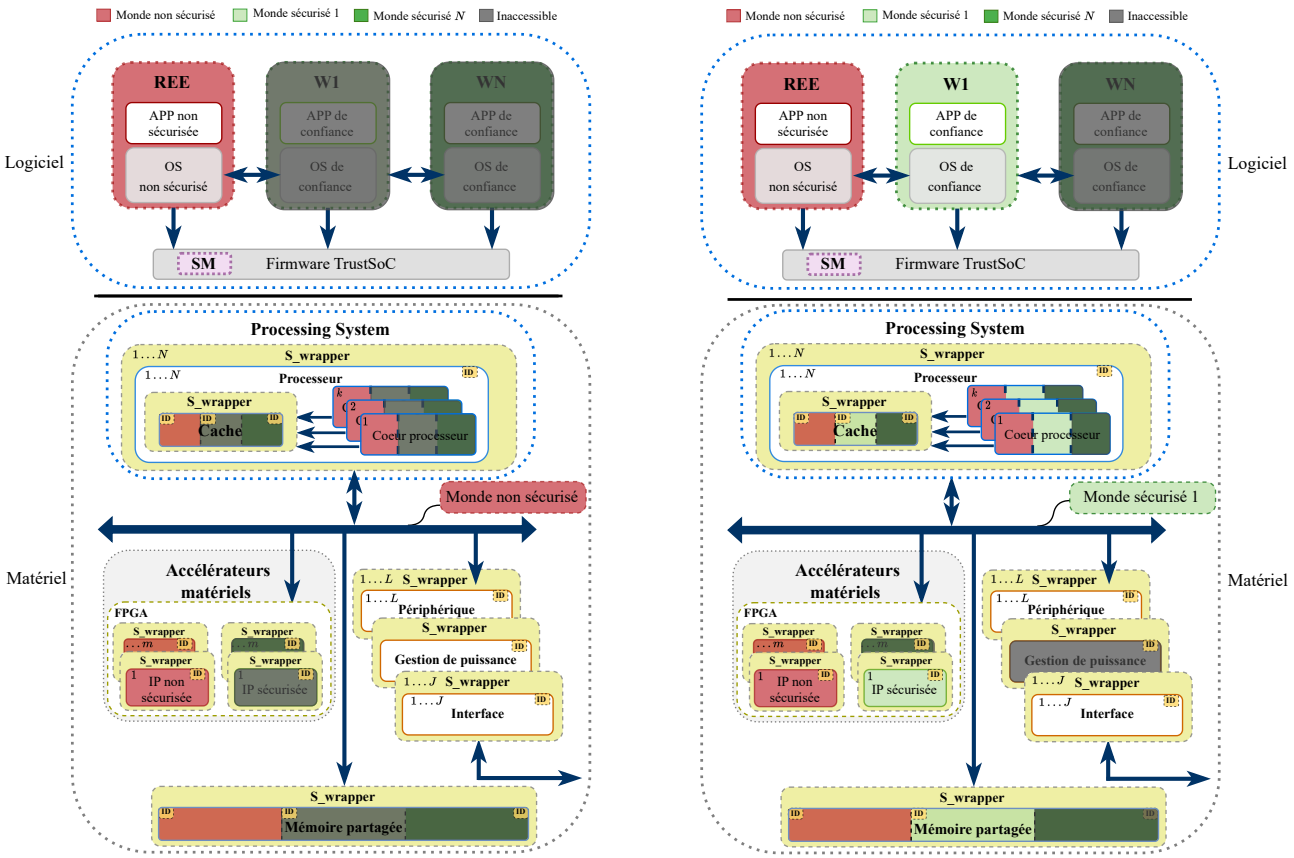
En conclusion, les contrôleurs distribués de communication de *TrustSoC* permettent de mettre en place différentes exigences de sécurité présentées dans la sous-sous-Section 3.3.2. La première étant l'exigence de sécurité n°4 (ES.4 : communications sécurisées à l'intérieur du SoC). *TrustSoC* utilise les contrôleurs de communication distribués que nous venons de présenter pour créer les voies de communication sécurisées entre tous les composants logiciels ou matériels du système (applications, accélérateurs matériels, les périphériques, mémoires, etc). Ces contrôleurs contribuent également à faire appliquer l'exigence de sécurité n°3 (ES.3) qui est l'intégration des ressources logiques (FPGA) dans la sécurité du système. Contrairement à ARM *TrustZone*, dans *TrustSoC* les ressources logiques bénéficient d'un contrôle complet des communications effectuées vers les accélérateurs matériels. Pour mettre en place totalement l'exigence de sécurité n°3 (ES.3), *TrustSoC* introduit le concept d'IP matérielles de confiance. Il vise à traduire le même concept que les applications de confiance dans un TEE (voir Chapitre 2), mais pour des IP matérielles intégrées dans le FPGA. Cela signifie que les IP matérielles appartenant aux mondes sécurisés sont garanties de fonctionner comme prévu et isolées des parties qui ne sont pas de confiance du reste du système. Ce concept sera plus détaillé et évalué dans la Section 3.4 de ce chapitre. Les contrôleurs de communication de *TrustSoC* permettent également d'appliquer l'ES.1 qui sont les règles de sécurité en veillant à ce que des routines soient effectuées ou via le contrôle des accès. Enfin, les contrôleurs contribuent également à faire respecter l'exigence de sécurité n°2 (ES.2 : extension en un système multi-mondes sécurisés) qui est présentée dans la sous-section suivante.

3.3.4 Fonctionnement

Après avoir présenté les exigences de sécurité, l'architecture et les contrôleurs de communication distribués de *TrustSoC*, nous allons maintenant présenter son fonctionnement et notamment le système multi-mondes (ES.2). Nous présentons dans la Figure 3.5, deux sous-figures qui illustrent le fonctionnement de *TrustSoC* dans deux mondes différents. La Figure 3.5 (A) correspond au fonctionnement dans le monde non sécurisé (ID monde = "monde non sécurisé") et la Figure 3.5 (B) correspond au fonctionnement dans le monde sécurisé n°1 (ID monde = "monde sécurisé 1"). Le monde non sécurisé et ses ressources apparaissent en rouge dans la Figure 3.5. Les mondes sécurisés (de 1 à N) et leurs ressources apparaissent en différentes nuances de vert (Figure 3.5). La Figure 3.5 illustre également les ressources inaccessibles du système par le monde qui est exécuté. Elles sont représentées par des blocs gris foncé. Nous proposons d'étudier le fonctionnement du système dans chacun de ces mondes ci-dessous.

Fonctionnement dans le monde non sécurisé

La Figure 3.5 (A) illustre l'état du système quand celui-ci opère dans le monde non sécurisé avec les ressources inaccessibles par ce dernier. Comme pour la technologie



(A) Fonctionnement de *TrustSoC* dans le monde non sécurisé.

(B) Fonctionnement de *TrustSoC* dans le premier monde sécurisé.

FIGURE 3.5 – Fonctionnement de l'architecture *TrustSoC* ainsi que les ressources accessibles du système dans différents mondes.

ARM *TrustZone*, aucune des ressources appartenant aux mondes sécurisés n'est accessible par le monde non sécurisé. Cela est illustré dans la 3.5 (A) avec les blocs gris placés sur les ressources identifiées initialement en nuances de vert. Les ressources inaccessibles sont les partitions sécurisées des mémoires (mémoire cache et mémoire partagée), les IP matérielles de confiance, les applications et cœurs des processeurs de confiance. Les ressources non sécurisées ne peuvent pas non plus effectuer des modifications sur les parties de confiance du système comme, par exemple, les partitions mémoires sécurisées. Pour mettre en place toutes ces règles de fonctionnement, comme nous l'avons vu dans la sous-section précédente, *TrustSoC* utilise des contrôleurs de communication distribués et des tables de permissions. Ils garantissent que les ressources (matérielles ou logicielles) de confiance sont isolées des ressources non sécurisées du système et inaccessibles par ces dernières. Pour cela, un contrôle systématique des accès est effectué à chaque interface de communication esclave d'une IP matérielle et du bus de communication dans *TrustSoC*.

Fonctionnement dans un des mondes sécurisés

Nous venons de voir le fonctionnement du système lorsque celui est dans le monde non sécurisé. Maintenant, nous proposons d'étudier le fonctionnement du système quand celui-ci opère dans un monde sécurisé.

La Figure 3.5 (B) illustre les ressources du système accessibles depuis un des mondes sécurisés (ID monde = "monde sécurisé 1"). Contrairement au fonctionnement du monde non sécurisé, quand le système fonctionne dans un des mondes sécurisés, celui-ci a accès aux ressources du monde non sécurisé. La technologie ARM *TrustZone* a un fonctionnement identique pour son unique monde sécurisé. Pour illustrer ce fonctionnement nous donnons l'exemple d'une application s'exécutant dans un des mondes sécurisés qui pourrait déléguer des fonctions à un accélérateur matériel du monde non sécurisé pour un traitement non sensible. Cependant, dans ces cas de fonctionnement, des règles de sécurité (ES.1) s'appliquent pour garantir que les données ne soient pas récupérées et utilisées. Les règles de *TrustSoC* stipulent que l'IP matérielle est réinitialisée automatiquement par le contrôleur de communication quand le traitement est terminé. Il s'agit de la règle RS_FPGA_1. De même, une règle similaire s'applique pour les partitions de la mémoire cache. Lors d'un changement de contexte, les différentes partitions de la mémoire cache qui ont été utilisées par des mondes sécurisés sont vidées (RS_CPU4 et RS_CPU_CPU2). Cette règle réduit les performances du système, mais offre un meilleur niveau de sécurité contre les attaques qui visent la mémoire cache. Plusieurs solutions, citées dans [10][24][70] dont une de l'état de l'art (CURE [10]), utilisent ce type de règles (avec des variations) pour protéger les partitions de la mémoire cache. Avec *TrustSoC*, les règles qui restreignent l'accès aux partitions de la mémoire cache contribuent à mettre en place l'exigence de sécurité **ES.5**. Cette exigence de sécurité correspond la résistance aux attaques SCA basées sur des mesures de temps d'accès qui visent la mémoire cache. *TrustSoC* introduit une première version du concept d'IP matérielle de confiance comme présenté dans la sous-section précédente. L'IP matérielle intégrée dans *TrustSoC* bénéficie de plus de protections que dans la technologie ARM *TrustZone*. Les contrôleurs de communication isolent leurs IP matérielles du bus de communication et contrôlent chaque accès effectué rendant ainsi impossible une attaque nécessitant un accès illégitime. *TrustSoC* stipule et met en place des règles de sécurité (ES.1) quant au système de gestion de puissance du SoC-FPGA qui, comme nous l'avons vu dans le premier chapitre de cette thèse, peut être utilisé pour réaliser des attaques [13][52][82][88] contre le système. Quand le système opère dans un monde sécurisé, le système de gestion de

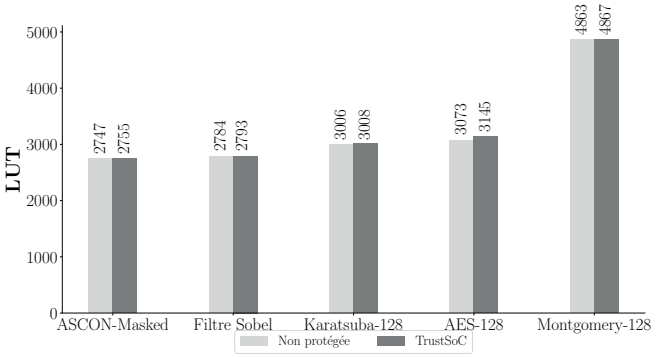
puissance ne peut pas être accédé et modifié, empêchant son utilisation pour réaliser les attaques présentées dans le Chapitre 1.

3.3.5 Résultats d'implémentation

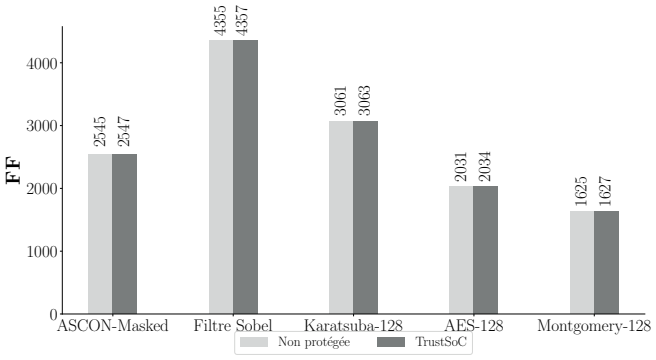
La sécurité apportée par *TrustSoC* repose principalement sur les contrôleurs de communication distribués. Ils assurent la sécurisation du bus de communication et la mise en œuvre des exigences de sécurité. Nous avons implémenté un contrôleur de communication sur cinq IP matérielles de traitement du signal et de cryptographie : un filtre de Sobel, une implémentation ASCON [29], Karatsuba-128, AES-128 [1] et Montgommery-128. Toutes ces IP matérielles sont spécifiées en langage de description matérielle (VHDL ou Verilog) et certaines sont disponibles sur Internet [32]. Les résultats de ces implémentations sont présentés dans la Figure 3.7.

Ces implémentations sont réalisées sur une carte AMD ZYBO qui embarque un SoC-FPGA AMD Zynq-7000 et avec le logiciel de CAO Vivado 2020.2. La taille des IP matérielles de base (sans modification) en LUT (*Look-Up Table*) est comprise entre 2 747 et 4 863 LUT. Pour les FF (*Flip Flop*), la taille de l'IP matérielle la plus petite est de 1 625 FF et la plus grande est de 4 355 FF. L'architecture des implémentations est présentée dans la Figure 3.7 (D). Le bus de communication et l'interface AMBA-AXI4 apparaissent en bleu foncé. Ensuite, le contrôleur de communication est représenté en vert-jaune avec les identifiants illustrés dans un bloc jaune en pointillés. Enfin, l'IP matérielle au centre. Elle peut appartenir à un des deux mondes du système (soit l'un soit l'autre), c'est pour cela que nous l'avons représentée en partie en rouge et en partie en vert avec une séparation au centre. Nous avons pris une petite configuration pour ces implémentations afin de montrer le coût de la partie test logique du contrôleur de communication. L'exploration du coût de *TrustSoC* pour des systèmes plus importants est détaillée dans la Figure 3.9 que nous présenterons par la suite. Les résultats d'implémentation sont donnés dans les Figures 3.7 (A-B-C). Les résultats en gris bleu correspondent à l'IP matérielle non protégée et ceux en gris foncé à l'IP matérielle avec le contrôleur de communication de *TrustSoC*. Comme nous l'avons dit précédemment, nous avons choisi une petite configuration, deux mondes et deux composants avec des identifiants, pour réduire au maximum le coût lié aux tables de permissions et ne montrer que le coût de la logique de test du contrôleur. Dans ces implémentations la taille des signaux utilisateurs est fixée à trois bits : un pour l'identifiant du monde et deux pour les identifiants des IP matérielles. Les résultats de la Figure 3.7 montrent que le coût de la logique de *TrustSoC* en ressources est faible. L'ajout d'un contrôleur de communication entraîne une augmentation maximale de 0,34% en LUT et de 0,13% en FF. La différence de deux FF observée pour toutes les IP matérielles (sauf AES) correspond aux deux bits de droits calculés : un pour l'écriture et un pour la lecture. Pour l'AES, un signal supplémentaire est utilisé pour son fonctionnement et donc ajoute cette FF supplémentaire. D'après nos résultats, le coût de la logique de test des contrôleurs en ressources matérielles (LUT et FF) est donc assez faible.

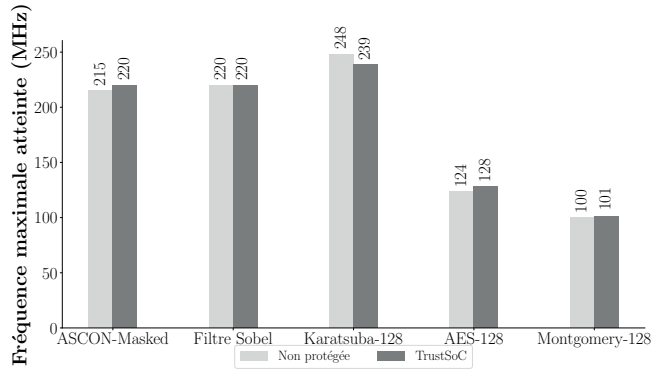
Nous avons évalué l'influence de l'ajout du contrôleur de communication sur la fréquence maximale de fonctionnement. D'après nos résultats, nous pouvons en déduire que les contrôleurs de communication de *TrustSoC* n'ont pas d'impact significatif sur cette dernière. Les fluctuations qui apparaissent sur la Figure 3.7 sont dues au processus de synthèse. En complément de ces résultats, nous avons réalisé une évaluation sur un oscilloscope de la latence en temps pour une communication de *TrustSoC*. L'objectif était de mesurer si les contrôleurs de communication de *TrustSoC* ajoutaient



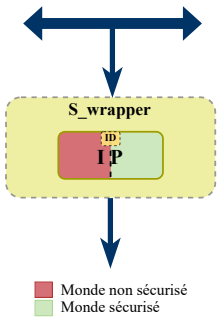
(A) LUT.



(B) FF.



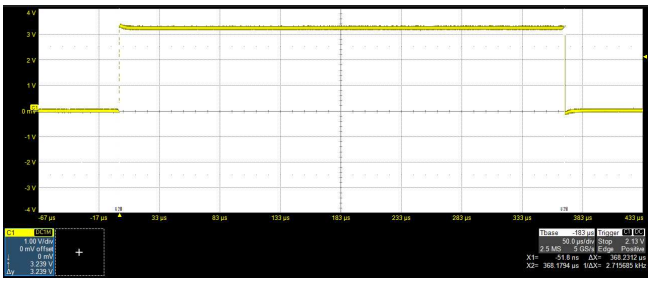
(C) FMAX.



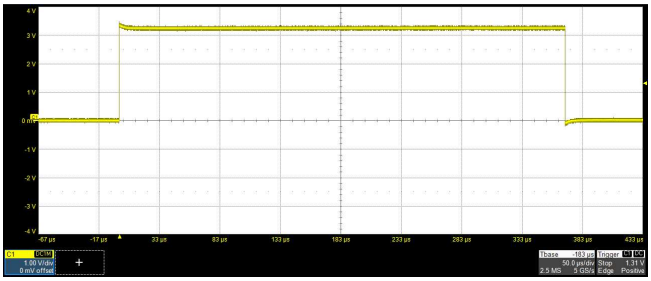
(D) Architecture des implémentations.

FIGURE 3.7 – Résultats d'implémentation en LUT, FF et fréquence maximum atteignable d'un contrôleur de communication de *TrustSoC* pour cinq IP matérielles différentes pour une petite configuration sur un SoC-FPGA AMD Zynq-7000.

un délai supplémentaire lors d'une communication. Nous avons réalisé cette évaluation plusieurs fois. Les résultats étaient identiques pour chaque test effectué et une réalisation est présentée dans la Figure 3.8. Il s'agit de captures de l'oscilloscope. Ces captures montrent le temps que met une communication pour s'effectuer totalement (opération d'écriture suivie d'une lecture) entre une IP matérielle maître et une IP matérielle esclave équipée d'un contrôleur de communication. Le temps que met la communication à s'effectuer est représenté par un signal en jaune dans la Figure 3.8 qui est actif tout au long de la communication. Une fois que celle-ci est terminée, il est désactivé. La Figure 3.8 (A) montre l'évaluation sans *TrustSoC* alors que la (B) est l'évaluation avec *TrustSoC*. Le calcul des droits se fait en fonction des identifiants



(A) Capture d'écran oscilloscope d'une communication vers une IP sans *TrustSoC*.



(B) Capture d'écran oscilloscope d'une communication vers une IP avec *TrustSoC*.

FIGURE 3.8 – Évaluation de la latence induite par *TrustSoC* lors d'une communication.

de la requête entrante et de manière combinatoire. Cela explique pourquoi *TrustSoC* n'ajoute pas de délai supplémentaire aux communications et confirme les résultats précédents.

Les résultats sur les ressources matérielles que nous avons présentés jusqu'à présent concernent un petit système. Pour explorer le coût de la solution *TrustSoC*, nous avons évalué le surcoût en ressources pour un ensemble de configurations pour un contrôleur de communication d'une BRAM (*Block Random Memory Access*). L'architecture de cette implémentation est présentée dans la Figure 3.9 et les résultats dans le Tableau 3.2. Dans notre implémentation, nous utilisons une mémoire de type BRAM d'AMD

Composants \ Mondes	2	4	8	16
2	7	11	20	29
4	9	15	28	50
8	15	17	48	83
16	23	29	77	152
32	53	89	157	291
64	92	160	296	1112

TABLE 3.2 – Résultats d'implémentation en nombre de LUT d'un contrôleur de communication lié à une BRAM en LUT (AMD Zynq-7000 SoC-FPGA).

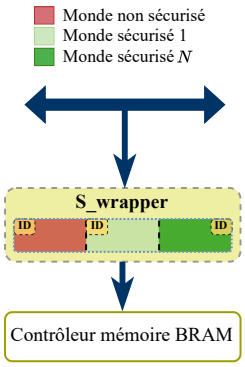


FIGURE 3.9 – Architecture contrôleur de communication d'une mémoire BRAM.

qui est disponible dans le logiciel de CAO Vivado d'AMD. Celle-ci est fournie avec un contrôleur comme représenté dans la Figure 3.9. Dans notre implémentation, le contrôleur de communication gère le partitionnement de la mémoire prévu par *TrustSoC* (partitions avec pointillés dans la Figure 3.9). Les IP matérielles (contrôleur

mémoire et mémoire) ne sont pas modifiées. Le contrôleur de communication découpe la mémoire en fonction des adresses et du nombre de partitions demandé. Pour faciliter l'implémentation, nous avons choisi des puissances de deux pour le nombre de partitions possibles : 2, 4, 8 et 16. Nous évaluons ensuite le coût du contrôleur de communication en fonction du nombre de composants dans le système : 2, 4, 8, 16, 32 et 64. Pour les petits systèmes le coût du contrôleur en LUT reste assez faible : par exemple, 20 LUT pour la carte AMD ZYBO (SoC-FPGA AMD Zynq-7000) ne représentent que 0,11% des ressources matérielles de type LUT totales. En revanche, pour les plus gros systèmes, le coût sur les ressources matérielles devient non négligeable : 300 LUT représentent 1,7% du total des LUT de la carte ZYBO. Ce surcoût est principalement lié aux tables de permissions qui deviennent de plus en plus importantes avec la taille du système. Elles sont quasiment doublées entre des valeurs de mondes ou d'identifiants consécutives. Par exemple pour 8 mondes et 16 composants nous avons 77 LUT. La configuration supérieure 16 mondes 16 composants vaut 152 LUT, ce qui est le double de la première. Ces tables de permissions sont implémentées dans des LUT. Si nous placions ces tables dans des mémoires, l'impact sur les ressources LUT serait alors réduit au coût de la logique de test qui a été évalué comme faible.

3.3.6 Limites

Le système de communications sécurisées à l'intérieur du SoC et les règles de fonctionnement de *TrustSoC* permettent un contrôle systématique des communications et du comportement du système. Par rapport à la technologie ARM *TrustZone* qui ne dispose pas d'un contrôle aussi développé, cela offre un avantage certain. En effet, le contrôle de *TrustSoC* est plus approfondi grâce à l'ajout des contrôleurs de communications pour chaque interface esclave du système. Cependant, *TrustSoC* est limitée dans son contrôle. En effet, les contrôleurs de communication de *TrustSoC* sont uniquement ajoutés aux interfaces de communication esclaves du SoC. L'architecture ne permet pas de prendre des mesures contre une IP malveillante qui effectuerait des communications malicieuses répétées sur le bus. *TrustSoC* ne prend donc pas en compte les attaques de type DoS. Enfin, *TrustSoC* n'est pas en mesure de fournir un système complet de communications sécurisées à l'intérieur du SoC sans ces éléments.

3.4 *RTrustSoC*

Pour pallier aux limites de *TrustSoC* qui ont été identifiées dans la sous-section précédente, *RTrustSoC* propose une extension de celle-ci. Cette section présente donc l'extension : *RTrustSoC*. Nous commençons d'abord par spécifier le modèle de menaces considéré par *RTrustSoC* et ses exigences de sécurité avant de détailler son architecture et son fonctionnement. Des résultats d'implémentation sont fournis ainsi qu'une évaluation suivant des scénarios d'attaques abordés dans le premier chapitre de cette thèse. Cette section inclut également une discussion quant aux limites de *RTrustSoC*.

Pour commencer, nous définissons le modèle de menaces considéré pour *RTrustSoC*.

3.4.1 Modèle de menaces

Comme pour *TrustSoC*, *RTrustSoC* considère des attaques à distance uniquement, matérielles et logicielles. *RTrustSoC* considère les mêmes menaces que *TrustSoC*. Le

modèle de menaces de *RTrustSoC* est également centré autour du processus de conception d'un SoC-FPGA en particulier l'utilisation de blocs matériels ou logiciels développés par un tiers. Ces blocs peuvent être malicieux. Ils peuvent également intégrer des vulnérabilités introduites involontairement durant la conception. Ces blocs peuvent ainsi réaliser des attaques contre le système. Les attaques peuvent être passives : collecte d'informations, écoutes, etc. Elles peuvent également être actives : modifications du contenu des communications, prise de contrôle des communications, etc. Ces attaques peuvent avoir de lourdes conséquences pour le système. Ainsi, *RTrustSoC* envisage des IP matérielles ou des applications logicielles malicieuses qui auraient été ajoutées au moment de la conception du SoC. Ces entités malveillantes peuvent réaliser : des accès illégitimes, des modifications du contenu mémoire, des modifications du contenu des communications ou prendre le contrôle du système de gestion de la puissance du SoC [14].

RTrustSoC considère également les attaques de type SCA basées sur la mesure du temps d'accès à la mémoire cache. Ces attaques peuvent être effectuées par une IP matérielle malicieuse intégrée dans le FPGA qui cible une application de confiance. Dans [17], l'attaque cible une implémentation AES qui s'exécute dans le processeur de confiance (technologie ARM *TrustZone* activée). Une IP matérielle malicieuse (monde non sécurisé) dans le FPGA réalise une attaque SCA basée sur la mesure du temps d'accès à la mémoire cache sur l'application de confiance.

Cependant, contrairement à *TrustSoC* [61], *RTrustSoC* considère les attaques de type déni de service (DoS). Le but de ces attaques est d'empêcher les entités légitimes d'utiliser les ressources du système. *RTrustSoC* considère principalement une entité malicieuse qui effectuerait un nombre considérable de requêtes. Elle monopoliserait ainsi le bus de communications et empêcherait ainsi les autres entités d'effectuer des communications.

Dans *RTrustSoC*, l'outil de CAO et le compilateur logiciel sont considérés comme fiables et ne peuvent pas être utilisés pour réaliser des modifications malicieuses de l'architecture. De même, les composants qui sont ajoutés pour la sécurité dans *RTrustSoC* sont de confiance et ne peuvent pas être modifiés par l'outil de CAO. Nous supposons également que le SoC et le fondeur sont fiables et que les blocs ne peuvent pas être modifiés comme avec l'ajout d'un cheval de Troie matériel.

3.4.2 Définition des exigences de sécurité

RTrustSoC a, tout comme l'architecture *TrustSoC*, des exigences en matière de sécurité. Il s'agit des mêmes exigences de sécurité, mais à la différence que l'exigence de sécurité n°4, communications sécurisées à l'intérieur du SoC, est étendue.

ES.1 : Règles de sécurité :

RTrustSoC doit intégrer des règles précises qui définissent le comportement autorisé du système. Ces règles de sécurité sont basées sur celles de *TrustSoC*. Celles-ci sont précisées dans l'Annexe A de cette thèse. Cependant, *RTrustSoC* doit ajouter des règles pour régir le comportement du système en cas d'attaques DoS. Ces règles doivent définir la fréquence des communications illégales qu'une IP matérielle peut effectuer. Elles doivent également, si une entité ne respecte pas ces règles, définir les pénalités qui s'en suivent.

ES.2 : Extension en un système multi-mondes sécurisé :

La proposition *RTrustSoC* doit, comme *TrustSoC*, être basée sur un système multi-mondes. *RTrustSoC* doit offrir la possibilités d'avoir plusieurs mondes sécurisés. Les mondes sécurisés peuvent avoir différents niveaux de privilèges. Le choix de ces niveaux de sécurité est laissé au concepteur.

ES.3 : Intégration des ressources logiques (type FPGA) dans la sécurité :

L'architecture *RTrustSoC* doit également intégrer un FPGA. Ce FPGA peut contenir différentes IP matérielles qui doivent chacune être associées à un monde. Elles doivent être entièrement intégrées dans la sécurité du système. Des règles de sécurité étendues (ES.1) doivent s'appliquer pour encadrer le comportement des IP matérielles.

ES.4 : Communications sécurisées à l'intérieur du SoC :

Le système de communication du SoC est un point particulièrement sensible du SoC, si l'attaquant parvient à compromettre le bus de communication, il peut en finalité prendre le contrôle du système ou le rendre inutilisable. Les communications à l'intérieur du SoC doivent être sécurisées. Comme pour *TrustSoC*, *RTrustSoC* doit fournir un système d'identification et de vérification des communications. *RTrustSoC* doit prévoir, en cas de détection de comportements malicieux d'une IP matérielle (trop de requêtes illégitimes), des sanctions. Elles permettent d'isoler ces entités et de les empêcher de nuire au système. *RTrustSoC* doit garantir que les IP matérielles introduites ont le comportement attendu.

ES.5 : Résistance aux attaques de type SCA basées sur des mesures de temps d'accès de la mémoire cache :

RTrustSoC doit embarquer les mêmes protections contre les attaques SCA à distance basées sur des mesures de temps d'accès qui visent la mémoire cache du système que *TrustSoC*. La mémoire cache doit être divisée en partitions et leur accès restreint. *RTrustSoC* doit également mettre en place les règles de sécurité concernant la mémoire cache (RS_CPU4, RS_CPU_CPU2 et RS_CPU_CPU3, RS_FPGA_CPU1).

3.4.3 Architecture et fonctionnement

Après avoir explicité le modèle de menaces ainsi que les exigences de *RTrustSoC*, nous allons présenter son architecture. Il s'agit d'une architecture flexible et modulable en fonction des besoins du concepteur. *RTrustSoC* est illustrée dans la Figure 3.10. L'architecture considérée est basée sur *TrustSoC*. Elle comprend donc les mêmes éléments. La partie *Processing System* inclut jusqu'à N processeurs de type ARM qui contiennent jusqu'à k cœurs. *RTrustSoC* embarque une partie de logique programmable (FPGA) contenant jusqu'à m IP matérielles. Elle comprend également un bus de communication de type AMBA-AXI4, des périphériques (interfaces, gestion de l'énergie, etc.) et des mémoires. Chaque application d'un cœur ou chaque IP matérielle implémentée dans la logique programmable (FPGA), est assigné au monde non sécurisé (rouge foncé Figure 3.10) ou à l'un des mondes sécurisés (jusqu'à N). Ces derniers sont identifiés dans différentes nuances de vert sur la Figure 3.10. Les mémoires partagées du SoC, la RAM ou la mémoire cache, sont toujours partitionnées en fonction du nombre de mondes : une partition par monde.

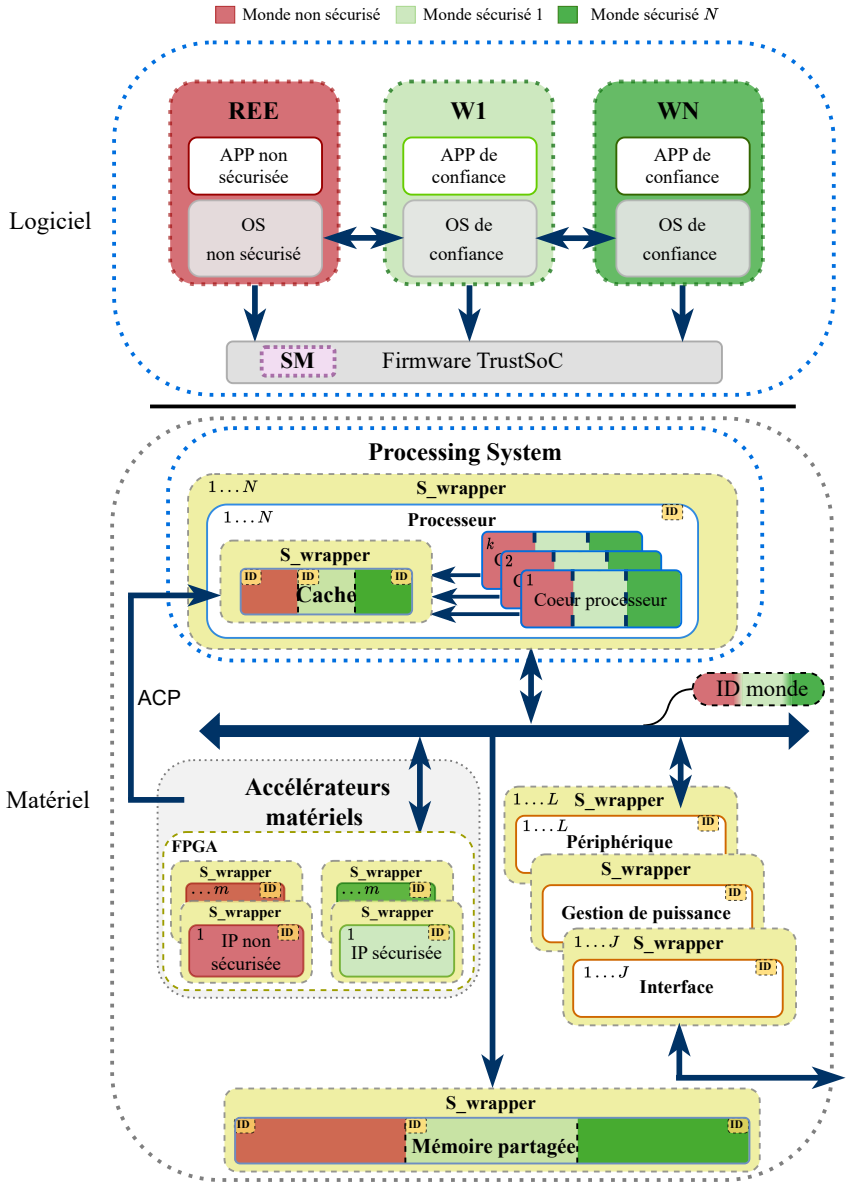


FIGURE 3.10 – Architecture *RTrustSoC*.

RTrustSoC embarque également des contrôleurs de communication matériels distribués (identifiés en vert-jaune dans la Figure 3.11) appelés *s_wrappers* pour *security wrapper*. La différence avec *TrustSoC* est qu'ils sont, cette fois, ajoutés entre chaque interface de communication du système et le bus de communication du SoC. Ils sont ajoutés aux interfaces de communication esclaves mais également aux interfaces maîtres du système. Ainsi, aucun composant du système n'est connecté directement au bus de communication, il en est isolé. Les requêtes de communication, entrantes ou sortantes, passent obligatoirement par le contrôleur de communication avant d'être transmises à une IP matérielle. L'architecture globale des nouveaux contrôleurs de communication (reliés aux interfaces maîtres) est représentée dans la Figure 3.11.

RTrustSoC est basée sur le même système d'identifiants matériels présenté dans *TrustSoC*. Chaque IP matérielle possède deux identifiants : un pour le monde auquel elle appartient et un, unique, pour elle-même (en jaune sur la Figure 3.11). L'identifiant monde de *RTrustSoC* permet d'identifier le monde auquel appartient une IP matérielle

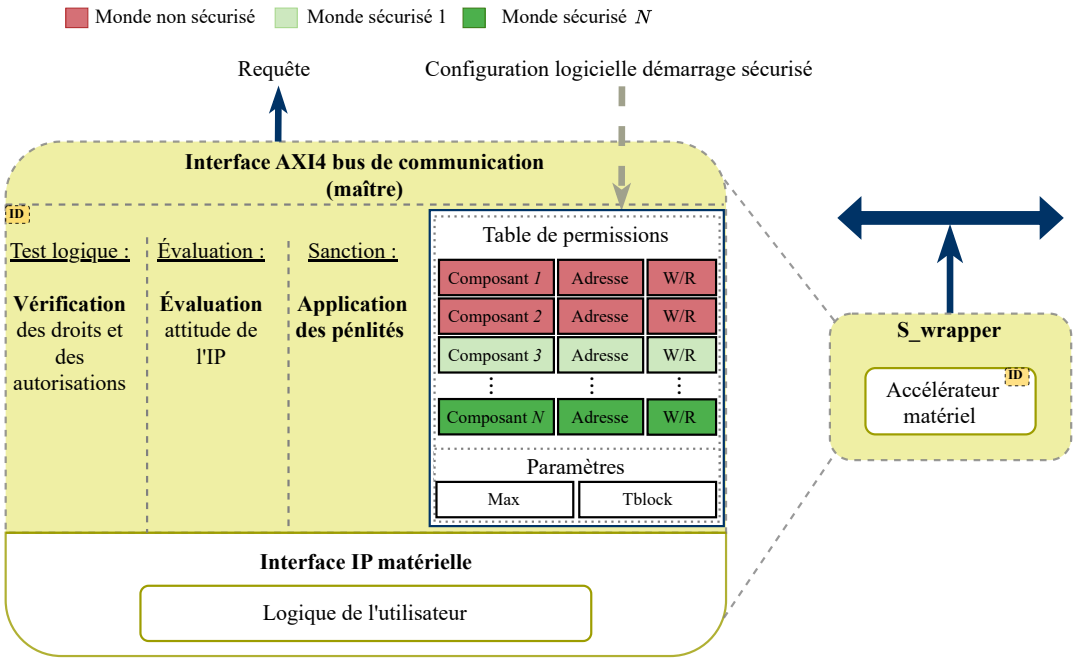


FIGURE 3.11 – Architecture des contrôleurs de communication connectés à une interface de communication maître de *RTrustSoC*.

et le monde dans lequel le système opère. L'encodage de ces identifiants est toujours $\lceil \log_2(\max(\text{nombre de mondes})) \rceil$ pour l'identifiant monde et pour celui de l'IP matérielle $\lceil \log_2(\max(\text{nombre de composants})) + 1 \rceil$. $\lceil \cdot \rceil$ est la fonction d'arrondi à l'entier supérieur. Ces identifiants sont codés matériellement et affectés à la présynthèse. Ils n'évoluent plus une fois la synthèse effectuée. Ils sont ajoutés à chaque communication dans le système via les signaux utilisateur du protocole de communication AXI4.

En plus de ces identifiants, les contrôleurs de communication disposent de tables de permissions qui spécifient pour chaque ressource matérielle du système, le monde auquel elle appartient ainsi que les droits d'accès de l'IP matérielle sous-jacente à cette ressource. Par exemple, dans la Figure 3.11, le composant 1 et 2 appartiennent au monde non sécurisé, le composant 3 appartient au monde sécurisé 1, etc. Les droits sont codés sur deux bits : un bit pour l'écriture et un pour la lecture. Si le bit de droit lié à l'entité à qui est destinée la demande est à '0', cela veut dire que l'IP matérielle n'a pas les droits, s'il est à '1' elle a le droit de communiquer en lecture ou en écriture avec le destinataire de la requête. Dans *RTrustSoC*, une condition est donnée sur les adresses. Si un composant a le droit de communiquer avec l'IP matérielle, il peut être restreint sur l'adresse. En outre, le contrôleur de communication intègre des paramètres qui sont utilisés pour le système de pénalités dynamiques : *Max* et *Tblock* (en blanc dans la Figure 3.11). Le paramètre *Max* représente le nombre maximum de requêtes illégitimes que l'IP matérielle est autorisée à réaliser. Le paramètre *Tblock* représente l'intervalle de temps durant lequel l'interface de communication de l'IP matérielle est désactivée si son comportement est jugé malveillant. Les tables de permissions et les paramètres sont codés matériellement, mais peuvent être modifiés au moment du démarrage sécurisé par une configuration sécurisée indiquée en pointillés kaki sur la Figure 3.11. Après cette configuration, ils sont fixes et ne peuvent plus être modifiés par une autre entité que le contrôleur de communication. Cette reconfiguration offre ainsi plus de flexibilité au concepteur. La logique du contrôleur

de communication est divisée en trois parties : une partie test logique, une partie évaluation et une partie sanction. Les contrôleurs de communication de *RTrustSoC* introduisent un système de pénalités dynamiques. Ce dernier est mis en place par les parties évaluation et sanction présentées dans la Figure 3.11. La partie évaluation analyse le comportement de l'IP matérielle en fonction des communications qu'elle effectue pendant une période de temps donnée à intervalles de temps réguliers. Le résultat de cette analyse permet de déterminer si l'IP matérielle se comporte de manière malveillante. Si c'est le cas, la partie sanction agit et désactive l'interface de communication pendant le temps *Tblock*. Elle agit également au travers de la modification des paramètres : *Max* et *Tblock*. Le nombre maximum de requêtes illégitimes acceptées (*Max*) est réduit et le temps de désactivation de l'interface de communication est augmenté (*Tblock*) (voir RDPS Figure 3.14). La dernière partie, test logique, vérifie les droits associés à chaque communication. Les requêtes ne peuvent seulement être transmises que si l'interface de communication est activée. Le fonctionnement de ces nouveaux contrôleurs de communication est expliqué plus en détail dans la Figure 3.12 avec le diagramme de séquence du fonctionnement d'un de ces contrôleurs. Dans cet exemple, l'IP matérielle qui dispose d'une interface de communication maître est un processeur qui veut communiquer avec un accélérateur matériel qui dispose d'un contrôleur de communication de type *TrustSoC*.

Dans la Figure 3.12, le processeur commence par faire une requête à l'IP matérielle. Le contrôleur de communication de *RTrustSoC* vérifie avant de la transmettre que l'interface de communication est activée. Si elle l'est, la requête est transmise à l'IP matérielle qui dispose également d'un contrôleur de communication. Les identifiants sont ensuite comparés aux tables de permissions de cette dernière. Il s'agit là du même comportement que celui présenté dans la Figure 3.4. La requête est soit légitime et transmise à l'IP matérielle, soit le contrôleur prend la main et transmet un message d'erreur sur le bus avec les signaux *XRESP*. Si un message d'erreur est remonté au contrôleur de *RTrustSoC*, celui-ci incrémente le compteur de requêtes illégitimes. Un test sur ce compteur et le paramètre *Max* est ensuite réalisé. Si ce dernier n'est pas encore atteint, l'interface de communication est laissée activée. Dans le cas contraire, le contrôleur applique les sanctions. Il désactive l'interface de communication pendant un temps *Tblock* qui est ensuite mis à jour. Si par contre, aucune erreur n'est remontée par le contrôleur de l'IP matérielle, la réponse est transmise directement au processeur et les paramètres ne sont pas modifiés.

Comme pour *TrustSoC*, les contrôleurs de communication de *RTrustSoC* permettent de mettre en place les exigences de sécurité. Les contrôleurs de communication distribués (maîtres et esclaves) de *RTrustSoC* permettent de faire respecter certaines règles de sécurité (**ES.1**) qui encadrent le comportement des IP matérielles auxquelles ils sont attachés. Les contrôleurs de communication distribués de *RTrustSoC* permettent de mettre totalement en place l'exigence de sécurité n°4 : communications sécurisées à l'intérieur du SoC. *RTrustSoC* utilise les contrôleurs de communication distribués que nous venons de présenter pour créer les voies de communication sécurisées entre tous les composants disposant d'interfaces de communication reliées au bus de communication du système. Grâce à l'extension apportée par *RTrustSoC*, l'exigence de sécurité n°3 est également totalement appliquée (l'intégration des ressources programmables (FPGA)) dans la sécurité du système. Ainsi, *RTrustSoC* bénéficie d'un contrôle total sur toutes les communications du système effectuées depuis ou vers les accélérateurs matériels. Grâce à ce contrôle complet, *RTrustSoC* étend le concept d'IP matérielle de confiance (FPGA) qui avait été précédemment introduit avec *TrustSoC*. Les IP matérielles appartenant aux différents mondes sécurisés sont garanties, d'une part,

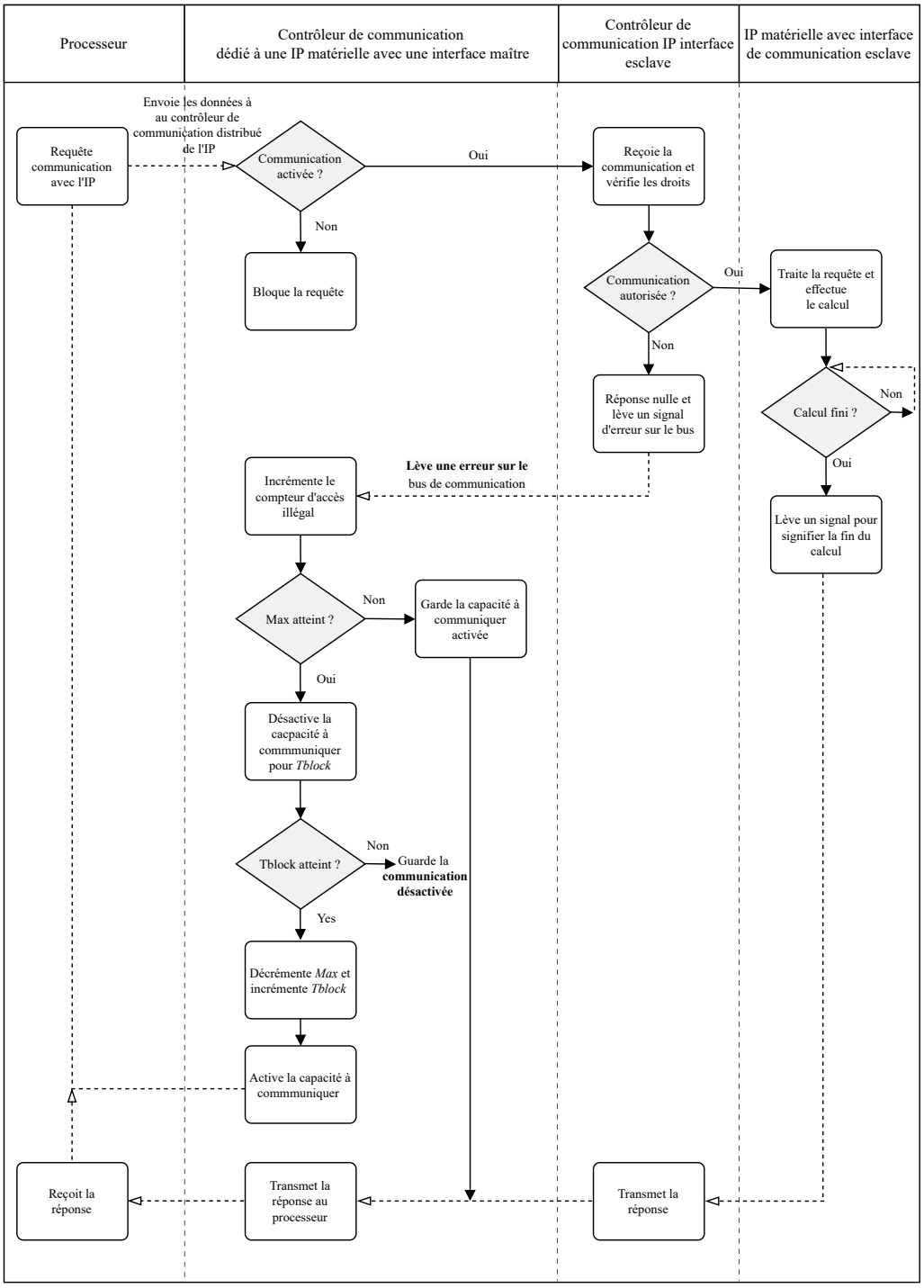


FIGURE 3.12 – Diagramme de séquence de fonctionnement des contrôleurs de communication de *RTrustSoC*.

d'être isolées des parties qui ne sont pas de confiance du système et, d'autre part, de fonctionner comme prévu. Avec *RTrustSoC*, toutes les IP matérielles font l'objet d'un contrôle des communications et ne peuvent pas interférer avec le système. Cette proposition d'IP matérielles de confiance de *TrustSoC* et *RTrustSoC* permet d'éviter certaines attaques basées sur des accès illégitimes depuis des accélérateurs matériels qui ont été présentées dans le premier chapitre de cette thèse : [17] et [40]. En effet, nous avons maintenant un contrôleur de communication qui peut contrôler les

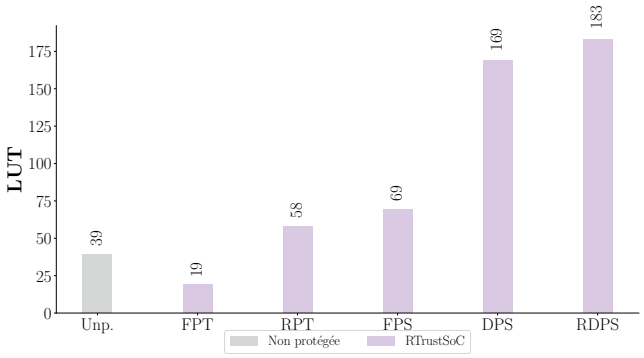
requêtes effectuées depuis le port ACP des accélérateurs matériels dès leur émission. Nous détaillons cet ajout pour les scénarios [17] et [40] dans la section suivante. En plus du contrôle sur les communications, les contrôleurs de *RTrustSoC* permettent d'appliquer les règles de sécurité (**ES.1**) qui apportent une sécurité supplémentaire au système. Ils mettent en place les routines et les remises à zéro nécessaires à leur bon fonctionnement. Les contrôleurs veillent également à faire respecter le partitionnement multi-mondes sécurisés du système (**ES.2**) qui a été présenté lors de l'architecture précédente. Son fonctionnement a également été présenté dans l'architecture précédente et étant identique à celui de *RTrustSoC*, nous n'allons pas le représenter.

3.4.4 Résultats d'implémentation

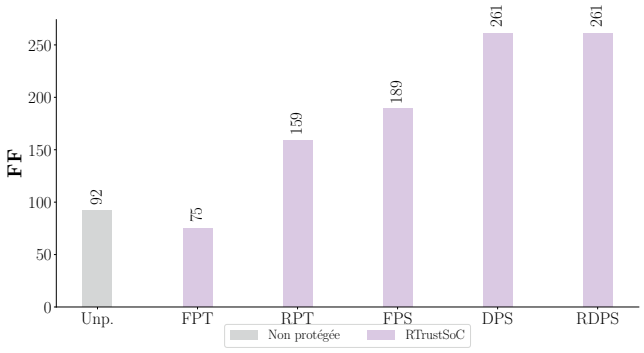
Nous avons implémenté les contrôleurs de communication pour *RTrustSoC* qui garantissent les exigences de sécurité présentées dans les sections précédentes avec différentes configurations. Les implémentations sont réalisées sur un SoC-FPGA d'AMD embarquant un Zynq-7000 (XC7Z010-1CLG400C). Tous les résultats de *RTrustSoC* sont obtenus en utilisant le logiciel de CAO AMD Vivado 2020.2. Les contrôleurs de communication distribués de *RTrustSoC* qui permettent de mettre en place le système de pénalités dynamiques sont spécifiés en langage de description matérielle (VHDL). Les résultats de ces implémentations sont présentés dans la Figure 3.14 en mauve pour *RTrustSoC* et en gris bleu pour la version non protégée.

La même configuration que pour les résultats obtenus avec TrustSoC a été utilisée, c'est-à-dire un petit système. Le contrôleur de communication de *RTrustSoC* est évalué par rapport aux métriques suivantes : nombre de ressources matérielles utilisées (LUT et FF) et fréquence maximale de fonctionnement atteignable (FMAX). Dans la Figure 3.14, le contrôle de communication a été implémenté avec ou sans la reconfiguration au démarrage sécurisé et le système de pénalités. Les résultats sont présentés dans cet ordre : non protégée (Unp.), tables de permissions fixes sans système de pénalités (FPT), tables de permissions reconfigurables sans système de pénalités (RPT), système de pénalités fixes non reconfigurables (FPS), système de pénalités dynamiques non reconfigurables (DPS), système de pénalités dynamiques reconfigurables (RDPS). Le surcoût en ressources matérielles (LUT et FF) induit par le contrôleur de communication peut sembler assez élevé avec l'ajout de la reconfiguration ou du système de pénalités. Cependant, la base de comparaison Unp. en bleu dans la Figure 3.14, est très petite, 39 pour les LUT et 92 pour les FF. Si nous comparons ces coûts avec le total de ressources matérielles disponibles pour la carte AMD ZYBO, les coûts moyens de nos implémentations représentent seulement 0,57% des LUT disponibles et 0,54% des FF disponibles. Les résultats obtenus pour la deuxième configuration (FPT : tables de permissions fixes) sont plus bas que la base de comparaison, ce qui est dû à l'utilisation de constantes dans l'implémentation qui entraînent l'outil de CAO à faire des optimisations. En conclusion, RTrustSoC a un surcoût en ressources matérielles faible comparé à la sécurité qu'il apporte. Nous détaillons cet aspect dans la section suivante.

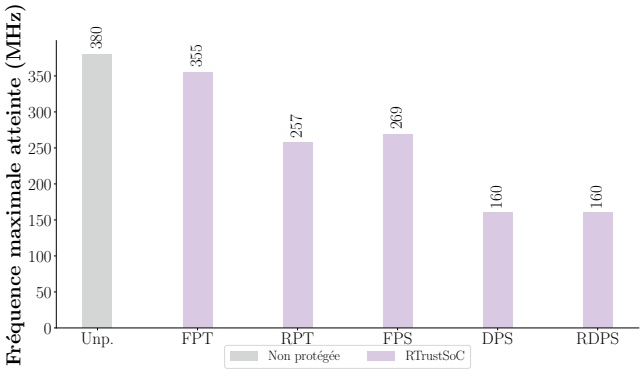
En revanche, contrairement à *TrustSoC*, les contrôleurs de communication de *RTrustSoC* ont un impact non négligeable sur la fréquence maximale de fonctionnement atteignable. Cet impact est montré dans la Figure 3.14 (C). Cela est dû aux compteurs qui sont utilisés par les contrôleurs pour les parties évaluation et sanction. Nous pourrions cependant améliorer nos résultats en changeant de type de compteur et en



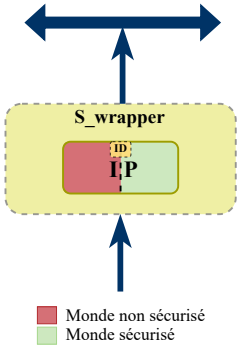
(A) LUT.



(B) FF.



(C) FMAX.



(D) Architecture des implémentations.

FIGURE 3.14 – Résultats d'implémentation du système de pénalités de *RTrustSoC* pour une IP matérielle avec un contrôleur de communication sur un SoC-FPGA AMD Zynq-7000. (Unp. pour non protégée, FPT pour table de permissions fixes, RPT pour table de permissions reconfigurable, FPS pour système de pénalités fixes avec *Max* et *Tblock* fixes, DPS pour système de pénalités dynamiques et *Max* et *Tblock* fixes, RDPS pour système de pénalités dynamiques et paramètres *Max* et *Tblock* reconfigurables).

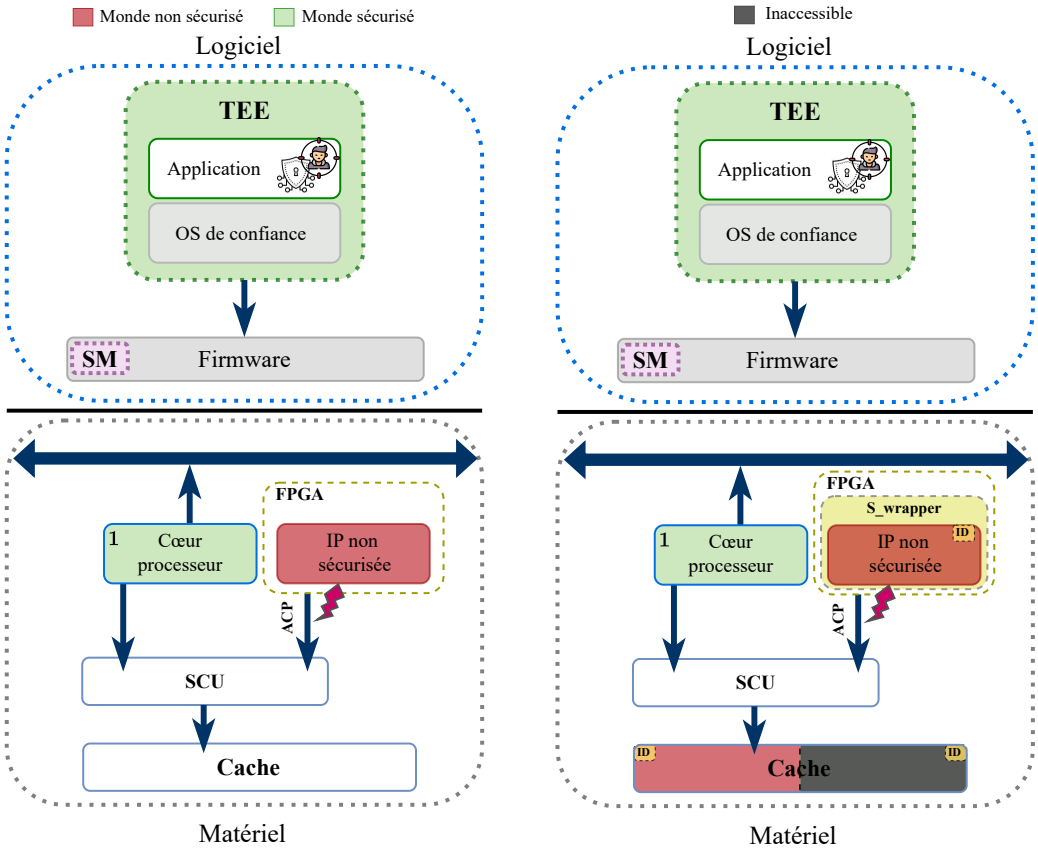
optant pour un compteur de type Johnson qui réduirait le chemin critique. Cependant, nous utiliserions plus de ressources matérielles avec l'ajout de bascules pour implémenter ces compteurs.

3.5 Scénarios d'utilisation de *RTrustSoC*

Dans cette section, nous proposons d'étudier l'utilisation de *RTrustSoC* dans des scénarios d'attaques présentés dans le premier chapitre de cette thèse. Le premier est basé sur [17]. Le deuxième et troisième scénarios sont basés sur [40].

3.5.1 Attaque basée sur la mesure des temps d'accès à la mémoire cache

Le premier scénario d'attaque présenté dans [17] vise une architecture de SoC-FPGA où la technologie ARM *TrustZone* est activée. Il est représenté dans la Figure 3.15 (A).



(A) Scénario d'attaque issu de [17].

(B) Scénario d'utilisation de *RTrustSoC* pour la mémoire cache.

FIGURE 3.15 – Scénario d'attaque de [17] et son application à *RTrustSoC*.

Une implémentation cryptographique d'un AES vulnérable aux attaques par mesures de temps d'accès à la mémoire cache s'exécute dans le monde de confiance (application identifiée en vert dans la Figure 3.15 (A)). Un accélérateur matériel malicieux appartenant au monde non sécurisé (identifié en rouge dans la Figure 3.15 (A)) est intégré dans la logique programmable (FPGA). L'accélérateur matériel malicieux, via le port ACP (*Accelerator Coherency Port*), accède à la mémoire cache et effectue des attaques basées sur des mesures de temps d'accès ("*Evict + Time*" et "*Flush + Reload*" expliquées dans le Chapitre 1) pour récupérer la clé de chiffrement de l'AES. Les auteurs spécifient que l'attaque est rendue possible car la plupart des systèmes

utilisant la technologie ARM *TrustZone* ne sont pas proprement configurés : les interfaces n'ont pas la configuration nécessaire pour empêcher les ressources non sécurisées d'accéder aux zones mémoires sécurisées.

La Figure 3.15 (B) illustre ce scénario appliqué à l'architecture de SoC-FPGA où *RTrustSoC* est embarquée. Pour correspondre à l'attaque décrite, le nombre de mondes sécurisés a été réduit à un, montré en vert dans la Figure 3.15 (B). L'accélérateur matériel est embarqué dans le FPGA dans le monde non sécurisé, identifié en rouge dans la Figure 3.15. L'accélérateur matériel accède ensuite à la mémoire cache par le composant appelé SCU (*Snoop Control Unit*) via le port ACP. La SCU connecte les cœurs du processeur et les interfaces ACP de la logique programmable (FPGA) à la mémoire cache et aide à sa cohérence. S'agissant d'un processeur ARM, il n'a pas été possible de modifier la mémoire cache pour y placer un contrôle de communication. À la place, pour notre démonstration, nous avons implémenté ce contrôleur à la connexion de l'interface ACP de l'accélérateur matériel. Dans ce scénario, les attaques, "*Evict + Time*" et "*Flush + Reload*", sont basées sur la capacité de l'accélérateur matériel à pouvoir vider une ligne de la mémoire cache. Cependant, si cette action est possible dans [17], elle ne l'est pas avec *RTrustSoC*. L'exigence de sécurité n°1 (règles de sécurité) et les tables de permissions empêchent une ressource non sécurisée d'accéder aux partitions sécurisées de la mémoire cache. Les tables de permissions ne sont pas modifiables puisqu'elles sont gérées par les contrôleurs de communication qui sont considérés comme fiables.

Nous avons implémenté le contrôleur de communication présenté dans la Figure 3.15 (B) avec sa table de permissions reconfigurable sur un SoC-FPGA AMD Zynq-7000 (XC7Z010-1CLG400C). La configuration correspond au scénario : deux mondes et deux composants. Dans cette implémentation, il n'y a qu'un seul monde sécurisé pour correspondre à [17], mais nous pourrions facilement étendre l'architecture à une architecture multi-mondes sécurisés. Les résultats d'implémentation sont donnés dans le Tableau 3.3. Le contrôleur de communication utilise seulement 0,36% des ressources

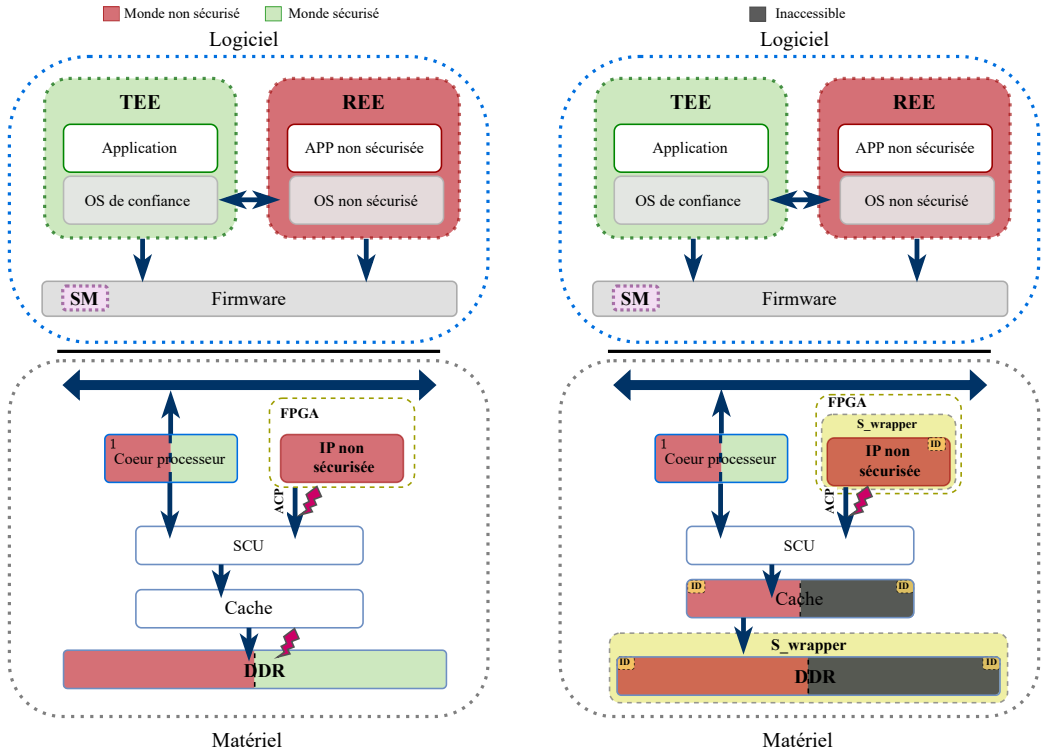
Table 3.3 – Résultats d'implémentation pour le scénario proposé dans la Figure 3.15 sur un SoC-FPGA AMD Zynq-7000 (XC7Z010-1CLG400C).

	LUT	FF	Fmax (MHz)
Contrôleur de communication port ACP	63	116	212
Utilisation (%)	0,36	0,33	–

LUT totales et 0,33% des ressources FF totales du SoC-FPGA. Ces résultats sont proches de ceux présentés dans la Figure 3.14. Le surcoût du contrôleur de communication de *RTrustSoC* est assez faible comparé à la sécurité qu'il apporte à l'architecture, empêchant ainsi l'attaque présentée dans [17].

3.5.2 Attaque par escalade de privilèges via le port ACP

Nous proposons d'étudier un deuxième scénario de l'utilisation de *RTrustSoC* pour une autre attaque présentée dans le premier chapitre de cette thèse. Le deuxième scénario d'attaque est présenté dans [40] et vise une architecture de SoC-FPGA avec la technologie ARM *TrustZone* activée. Il s'agit d'un SoC-FPGA d'AMD : Zynq UltraScale+ MPSoC ZCU102. Le scénario d'attaque est représenté dans la Figure 3.16 (A).



(A) Scénario d'attaque issu de [40].

(B) *RTrustSoC* appliquée au scénario d'utilisation d'attaque 1 de [40].

FIGURE 3.16 – Scénario d'attaque de [40] et son application à *RTrustSoC*.

La première attaque [40] vise à casser l'isolation mémoire de la technologie ARM *TrustZone*. Elle se base sur un accélérateur matériel appartenant au monde non sécurisé intégré dans la logique programmable (FPGA). L'accélérateur matériel provient d'un tiers et embarque une modification malicieuse : un cheval de Troie matériel. Cet accélérateur matériel est identifié en rouge dans la Figure 3.16 (A). Le système embarque également une partie logicielle : un CPU avec le REE et TEE de la technologie ARM *TrustZone*. Les auteurs se basent sur des vulnérabilités de la technologie ARM *TrustZone* pour réaliser leur attaque. La première est la capacité de l'accélérateur matériel malicieux à modifier le signal de sécurité (*NS bit*) sans déclencher d'erreur. La deuxième vulnérabilité réside dans l'absence de contrôle et de politique de sécurité pour les communications issues d'interfaces maîtres de la logique programmable utilisant le port ACP. L'accélérateur matériel malicieux peut remplacer ou écrire du code et des données dans la partie sécurisée de la mémoire. Ainsi,grâce à la combinaison de ces deux failles et la modification malicieuse introduite dans la logique programmable, le cheval de Troie matériel est en mesure de réaliser une escalade de privilèges. En fin de compte, l'entité malveillante peut remplacer les applications de confiance par d'autres malicieuses.

L'attaque est rendue possible, comme pour le premier scénario, par de l'absence de protections et de configurations pour les communications à l'intérieur du système afin d'empêcher les ressources non sécurisées d'accéder à celles sécurisées. Cependant, avec *RTrustSoC*, les politiques de sécurité et le contrôle des communications sont implémentés et configurés. Le scénario d'attaque appliqué à *RTrustSoC* est représenté dans la Figure 3.16 (B). Dans *RTrustSoC*, un accélérateur matériel non sécurisé ne peut pas accéder aux ressources de confiance du système. Dans l'attaque [40], l'accélérateur

matériel malicieux peut changer la valeur du signal de sécurité (*NS bit*), ce qui n'est pas possible avec *RTrustSoC*. Dans *RTrustSoC*, la gestion des signaux de sécurité et le transport des identifiants sont gérés par le contrôleur de communication. Comme l'accélérateur matériel en est isolé, il ne peut pas modifier ces signaux. Avec l'ajout du contrôleur de communication à l'interface de communication ACP, la première étape de l'attaque décrite dans [40] n'est pas réalisable. De plus, avec l'ajout du contrôleur de communication à l'interface de communication de la mémoire, ce même accélérateur matériel ne peut pas accéder à la partition de confiance de cette dernière. Il ne peut pas non plus la modifier empêchant ainsi l'attaque. Le contrôle de la communication basé sur les identifiants rejettera la requête. Le surcoût du contrôleur de communication relié au port ACP de ce scénario n'a pas été implémenté car son coût correspond à celui du scénario précédent, présenté dans le Tableau 3.3. Dans *RTrustSoC*, un contrôleur au niveau de la mémoire empêcherait l'accès illégal à la zone sécurisée. Le surcoût de ce contrôleur a été présenté dans la deuxième section de ce chapitre et correspond aux résultats (en LUT et en en FF) présentés dans le Tableau 3.4.

TABLE 3.4 – Résultats d'implémentation pour le contrôleur de communication de la mémoire proposé dans la Figure 3.16 (B) sur un SoC-FPGA AMD Zynq-7000 (XC7Z010-1CLG400C) et pour un système à un monde sécurisé.

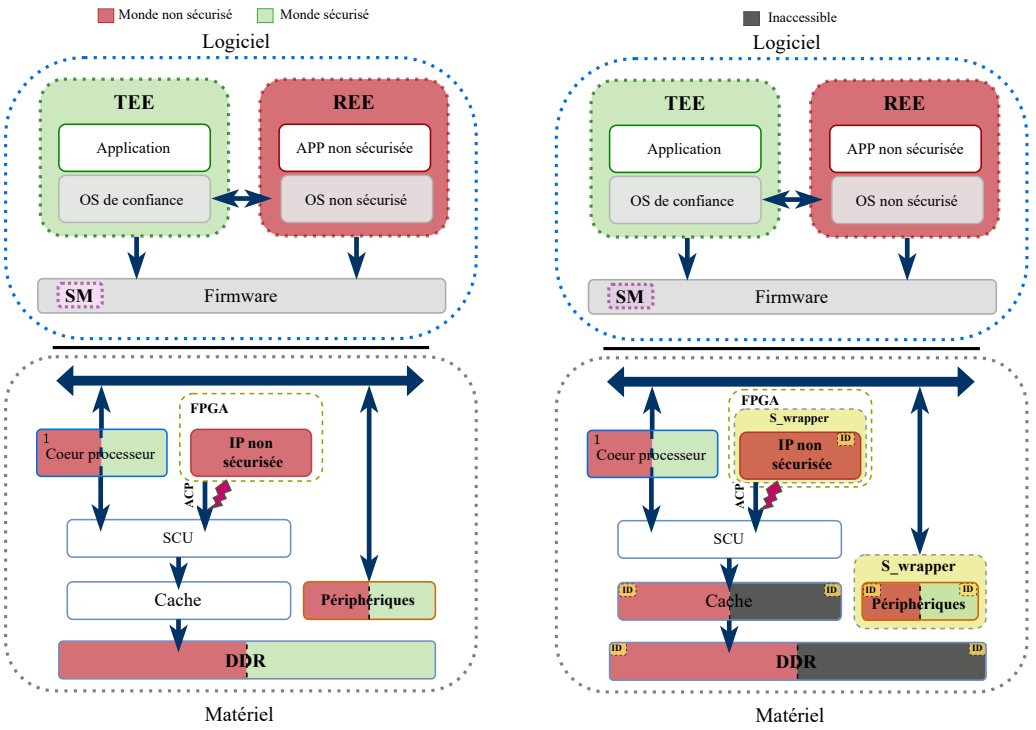
	LUT	FF	FMAX (MHz)
Contrôleur de communication mémoire	8	8	225
Utilisation (%)	0,05	0,02	–

Le contrôleur de communication utilise 0,05% des ressources LUT totales et 0,02% des ressources FF du SoC-FPGA, ce qui est négligeable pour l'ajout de sécurité qu'il apporte à l'architecture.

Nous proposons d'étudier un dernier scénario d'utilisation de *RTrustSoC* dans le cadre d'une attaque visant le démarrage sécurisé du SoC-FPGA.

3.5.3 Attaque du démarrage sécurisé via le port ACP

Le dernier scénario d'attaque étudié est également issu de [40]. Il vise une architecture de SoC-FPGA avec la technologie ARM *TrustZone* activée. Il est représenté dans la Figure 3.16 (A). L'attaque [40] vise à casser le démarrage sécurisé du système. Elle est réalisée sur un SoC-FPGA Zynq UltraScale+ MPSoC ZCU102. Elle se base sur un accélérateur matériel appartenant au monde non sécurisé intégré dans la logique programmable (FPGA) (Figure 3.16 (A), IP matérielle identifiée en rouge). L'accélérateur matériel embarque une modification malicieuse. Le système comprend également une partie logicielle : un CPU avec le REE et TEE de la technologie ARM *TrustZone*. Les auteurs se basent sur une vulnérabilité utilisée dans le scénario précédent pour réaliser cette nouvelle attaque. Cette vulnérabilité concerne l'absence de contrôle et de politiques de sécurité pour les communications issues d'interfaces maîtres de la logique programmable utilisant le port ACP. Grâce à cette vulnérabilité, une modification malicieuse est en mesure d'accéder à des ressources de confiance auxquelles elle n'a pas le droit d'accéder en théorie. Les auteurs décrivent les étapes pour la réalisation de leur attaque [40]. La logique malicieuse doit accéder aux eFuses (périphériques dans la Figure 3.16 (A)) du SoC-FPGA pour programmer une clé primaire publique RSA malicieuse qui est utilisée lors du démarrage sécurisé. La deuxième étape est une option car elle n'est pas nécessaire pour le démarrage sécurisé, il s'agit de reprogrammer



(A) Scénario d'attaque issu de [40].

(B) Scénario d'utilisation de *RTrustSoC* appliqué au scénario d'utilisation d'attaque 2 de [40].

FIGURE 3.17 – Scénario d'attaque de [40] et son application à *RTrustSoC*.

une clé primaire publique AES-256 qui est stockée dans la mémoire BBRAM (*Battery-Backed Random Access Memory*) (périphériques dans la Figure 3.16 (A)). Cette clé est utilisée pour décrypter et authentifier une image du démarrage sécurisé lors de ce dernier. La vulnérabilité est que les registres de la BBRAM sont également accessibles par le port ACP depuis la logique programmable. L'attaquant est en mesure de reprogrammer la clé qui se trouve dans la BBRAM pour la remplacer à sa convenance. Une fois que ces clés sont programmées, elles ne changeront pas et persisteront au fil des démarrages du système sans déclencher d'erreur. Les auteurs précisent que la dernière étape consiste à remplacer l'image de démarrage par celle de l'attaquant. Pour cela, il est possible d'utiliser la même stratégie qu'au scénario précédent : la modification des signaux de communication par l'accélérateur matériel malicieux. Une fois cette étape effectuée, le système est sous le contrôle de l'attaquant qui a remplacé, avec succès, l'image du démarrage sécurisé, sans déclencher d'erreur.

Le scénario d'attaque appliqué à *RTrustSoC* est présenté dans la Figure 3.16 (B). Comme pour les scénarios précédents, le contrôle des communications est entièrement configuré avec *RTrustSoC* par le biais des politiques de sécurité et des contrôleurs de communication. Ainsi, la logique programmable qui embarque des accélérateurs matériels non sécurisés ne peut pas accéder aux ressources de confiance du système depuis le port ACP. Dans la deuxième attaque présentée dans [40], l'accélérateur matériel malicieux peut accéder aux périphériques sécurisés, alors qu'en théorie cela ne devrait pas être possible, afin de programmer une nouvelle clé primaire publique. Dans *RTrustSoC*, cet accès n'est pas possible à cause des contrôleurs de communication rajoutés aux interfaces, comme montré dans la Figure 3.17 (B) (périphériques et IP non sécurisée en vert-jaune). L'attaque [40] basée sur cet accès illégitime est ainsi

empêchée. Les surcoûts entraînés par ces contrôleurs ajoutés ont été donnés dans les scénarios précédents : Tableau 3.3 et Tableau 3.4.

3.6 Limitations de la proposition

Nous avons rencontré plusieurs difficultés avec *TrustSoC* et *RTrustSoC*, surtout lors de l'implémentation des contrôleurs de communication.

Lors du scénario présenté dans la Figure 3.15, nous voulions implémenter notre contrôleur de communication comme illustré dans la Figure 3.10 au niveau de la mémoire cache. Or, nous nous sommes aperçus qu'avec l'emploi d'un processeur de type ARM, cette modification n'était pas être possible. En effet, le processeur étant câblé et propriétaire, aucune modification ne peut être effectuée sur le *Processing System* (PS). Cela limite les propositions de *TrustSoC* et *RTrustSoC*. Une autre limite à laquelle nous avons été confrontés lors de l'implémentation est la différence du système de communication entre la partie système et la partie logique programmable. Cette limite est présentée dans la Figure 3.18 où l'on peut voir la représentation du processeur câblé sur la gauche et une IP matérielle avec un contrôleur de communication sur la droite. Pour *TrustSoC* et *RTrustSoC*, nous avons besoin d'utiliser le protocole AXI4-full, identifié en vert-jaune dans la Figure 3.18 avec l'interface du contrôleur de communication. En revanche, le processeur câblé utilise une interface "*General Purpose 32-bit*

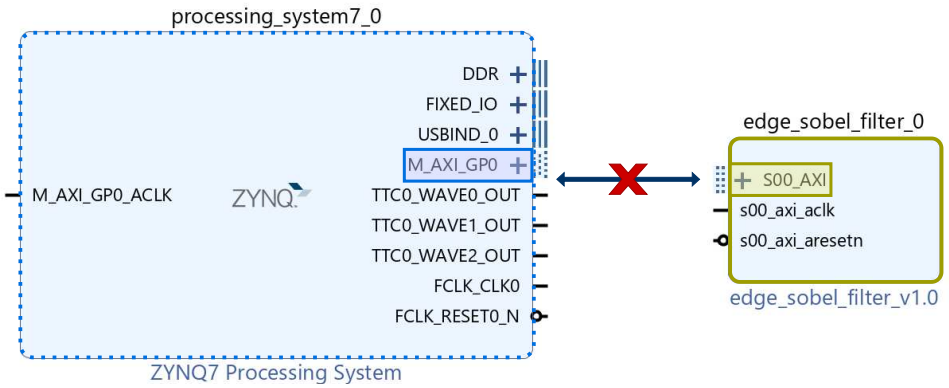


FIGURE 3.18 – Limite lors de l'implémentation de *RTrustSoC*.

AXI Master" qui n'est pas en AXI4 (identifiée en bleu dans la Figure 3.18). Nous avons donc dû ajouter une IP supplémentaire pour faire l'adaptation entre les deux protocoles et ajouter les signaux de sécurité de *TrustSoC* et *RTrustSoC*. Le protocole AXI étant propriétaire les changements d'implémentation sont rendus difficiles. Dans [56], les auteurs proposent une solution pour se prémunir des attaques [12] présentées dans le Chapitre 1 de cette thèse. Ils ajoutent des blocs de chiffrement et de déchiffrement supplémentaires. Cependant, cet ajout représente un surcoût non négligeable pour le système. Cette solution n'est pas envisageable pour des systèmes embarqués où la contrainte sur l'utilisation de ressources est forte.

Par conséquent, le fait que le processeur soit câblé et propriétaire nous a empêchés d'implémenter notre concept comme nous le souhaitions. Ainsi, nous avons recherché d'autres options de choix de processeur afin de nous permettre de nous affranchir de ces limites et d'approfondir notre concept. Nous avons choisi d'utiliser un processeur de type RISC-V [80] qui nous permet d'avoir la possibilité d'effectuer les changements souhaités. Nous avons également ciblé ces architectures open source pour pouvoir

proposer des modifications sur le système de communication qui nous avait limités jusqu'à présent. Cette contribution, *TrustSoC-V*, fait l'objet du chapitre suivant.

3.7 Conclusion du chapitre

Ce chapitre a présenté deux architectures sécurisées de SoC hétérogène qui ont permis de répondre aux points faibles de l'état de l'art identifiés lors du chapitre précédent. Il a permis de présenter une première architecture sécurisée appelée *TrustSoC* ainsi que son modèle de menaces et ses exigences de sécurité. Pour étendre le concept de *TrustSoC*, nous avons présenté *RTrustSoC*, qui permet d'étendre le modèle de menaces de *TrustSoC* et la sécurité apportée à l'architecture du SoC hétérogène.

Ces deux architectures ont introduit le concept de multi-mondes sécurisés basé sur la technologie ARM *TrustZone*, permettant ainsi au concepteur de disposer de plus de flexibilité et de sécurité. Elles ont également proposé le concept d'IP de confiance, qui n'avait alors jamais été présenté. Il vise à étendre celui des applications sécurisées et de leurs environnements d'exécution de confiance à la logique programmable (FPGA) du SoC-FPGA.

Ce chapitre a présenté des résultats d'implémentation pour différentes IP matérielles. Ces résultats montrent les surcoûts matériels des blocs ajoutés dans les deux architectures mais également les surcoûts temporels avec notamment avec les évaluations des fréquences maximales atteintes et de la latence induite par l'ajout des contrôleurs.

Nous avons également fourni trois scénarios d'utilisation issus de [17] et [40] appliqués à *RTrustSoC* avec des résultats d'implémentation. Nous avons présenté une première application de *RTrustSoC* pour une attaque par analyse des temps d'accès à la mémoire cache depuis la logique programmable (FPGA). Ensuite, nous avons appliqué à *RTrustSoC* deux scénarios d'attaques de [40]. Le premier consistait à effectuer une escalade de privilèges et le deuxième visait le démarrage sécurisé du système.

Ce chapitre a permis de décrire deux premières contributions qui complètent l'état de l'art des architectures sécurisées de SoC hétérogènes. Ces propositions sont par la suite complétées par la contribution suivante *TrustSoC-V*, basée, sur un processeur de type RISC-V.

Chapitre 4

TrustSoC-V et sa mise en œuvre par RISC-B

Résumé

Ce chapitre présente dans un premier temps une troisième contribution : TrustSoC-V. Il s'agit d'une architecture de SoC hétérogène sécurisée par conception embarquant des processeurs RISC-V [80]. Elle est basée sur les mêmes concepts que les architectures présentées dans le chapitre précédent : *TrustSoC* et *RTrustSoC*. Nous présentons ensuite les objectifs, les motivations, les exigences de sécurité ainsi que le fonctionnement de cette contribution. Nous fournissons également quelques résultats d'implémentations matérielles pour différentes IP matérielles (traitement du signal et cryptographie) sur un SoC-FPGA AMD Z-7000 (XC7020).

Dans un deuxième temps, ce chapitre présente une quatrième contribution appelée RISC-B qui facilite la mise en œuvre de *TrustSoC-V* par un concepteur. Nous expliquons sa méthodologie et sa mise en œuvre. Nous fournissons également des résultats d'implémentation sur un Zynq UltraScale+ MPSoC d'AMD (ZCU104).

Table des matières

4.1 Introduction au chapitre 80
4.2 Motivations 80
4.3 *TrustSoC-V* 80
4.3.1 Modèle de menaces 81
4.3.2 Définition des exigences de sécurité 81
4.3.3 Architecture *TrustSoC-V* 83
4.3.4 Fonctionnement 88
4.3.5 Résultats d'implémentation 90
4.4 RISC-B 95
4.4.1 Objectifs et motivations de RISC-B 95
4.4.2 Méthodologie 95
4.4.3 Résultats d'implémentation 96
4.5 Limitations de la proposition 98
4.6 Conclusion du chapitre 98

4.1 Introduction au chapitre

Le projet RISC-V a été créé en 2010 par l'université de Berkeley en Californie, aux États-Unis. Lors de sa création, ce projet avait un but purement de recherche. Cependant, en raison de son potentiel, il est devenu un standard, y compris pour l'industrie. RISC-V est une *Instruction Set Architecture* (ISA), ouverte et libre de droits. Les spécifications complètes sont disponibles ici : [80]. Le processeur RISC-V permet de résoudre le problème des modifications posé par une technologie propriétaire comme ARM ou les limitations qui peuvent exister avec ces technologies [30]. Le gros potentiel de ce type de processeur réside dans ses spécifications ouvertes qui permettent une personnalisation complète du processeur en fonction de son application. Cela permet à un concepteur souhaitant créer son propre processeur de le faire avec un jeu d'instructions basé sur la spécification [80]. Des extensions, standardisées ou non, existent pour spécialiser le processeur dans un certain domaine. Une étude de ces extensions est présentée dans [28]. La sécurité est l'un des domaines concernés. Des processeurs RISC-V orientés pour la sécurité ont notamment été proposés [45][83]. Ainsi, RISC-V constitue une solution intéressante pour résoudre les limites que nous avons identifiées au chapitre précédent pour *TrustSoC* et *RTrustSoC*.

La contribution *TrustSoC-V* a fait l'objet d'une présentation lors d'un concours entre étudiants [58] pendant l'ESWEEK (*Embedded Systems WEEK*) 2023 à Hambourg (Allemagne). Elle a également fait l'objet d'une présentation sous forme de poster [62] au RISC-V SUMMIT à Munich (Allemagne). La contribution *RISC-B* est actuellement en soumission dans une conférence internationale.

4.2 Motivations

La contribution de *TrustSoC-V* a pour objectif de :

- Proposer une solution minimale, libre de droits, centrée sur le bus de communication du SoC en étendant les concepts des architectures présentées au Chapitre 3.
- Proposer un nouveau bus de communication du SoC hétérogène, libre et sécurisé par conception.
- Englober pour la sécurité du système tous ses éléments : périphériques, processeurs, accélérateurs matériels, etc.
- Proposer des mécanismes de sécurité pour les différentes mémoires du SoC hétérogène.
- Répondre à des exigences de sécurité et un modèle de menaces bien définis.

4.3 *TrustSoC-V*

Cette section présente *TrustSoC-V* qui est une proposition d'architecture de SoC hétérogène sécurisée par conception. Elle se concentre sur le bus de communication du SoC et repose sur un modèle de menaces précis et des exigences de sécurité. Le modèle *TrustSoC-V* présente de multiples caractéristiques de sécurité :

- Le système de communication est basé sur une proposition de nouveau bus de communication, appelé *XSecure* sécurisé à la conception.
- Le système repose sur le même partitionnement en multiples mondes sécurisés présenté au Chapitre 3. Les composants matériels ou logiciels peuvent être assignés à différents mondes avec différents niveaux de privilèges.

— La mémoire cache est protégée contre les attaques qui tirent parti de l'accès à la mémoire cache partagée.
— Un ensemble de contrôleurs de communication distribués (moniteurs) appliquent des politiques de sécurité pour renforcer le système de communication de confiance à l'intérieur du SoC.

4.3.1 Modèle de menaces

Le concept *TrustSoC-V* est centré sur les mêmes menaces que *TrustSoC* et *RTrustSoC*. Le modèle de menaces de *TrustSoC-V* se concentre sur le processus de conception d'un SoC-FPGA [44]. Il considère des attaques à distance uniquement, depuis la partie logicielle (PS) et depuis la partie matérielle (logique programmable (FPGA)).

TrustSoC-V considère les attaques par accès illégitimes et modifications du contenu mémoire. Comme pour *TrustSoC* et *RTrustSoC*, nous envisageons une application logicielle (ou une IP matérielle) qui tente d'accéder aux informations sensibles d'autres applications logicielles ou IP matérielles. Les menaces ci-dessus sont liées à l'utilisation de blocs matériels ou logiciels développés par un tiers. Ces blocs ne font pas toujours l'objet de vérifications. Ils peuvent se révéler malicieux ou vulnérables et peuvent être à l'origine d'attaques contre le système. Ces attaques ont été présentées dans le Chapitre 1 et prendre la forme d'accès illégitimes, de modifications du contenu mémoire, de modifications du contenu des communications ou d'une prise de contrôle du système de gestion de la consommation de puissance du SoC, etc.

TrustSoC-V considère les attaques de type SCA qui visent la mémoire cache et qui sont basées sur des mesures du temps d'accès. Ces attaques peuvent être effectuées depuis la partie logicielle [55][41] ou par une IP matérielle malicieuse [17] intégrée dans le FPGA.

Comme *RTrustSoC*, *TrustSoC-V* considère les attaques de type déni de service (DoS). Le but de ces attaques est d'empêcher les entités légitimes de pouvoir fonctionner et d'utiliser les ressources du système. Par exemple, une entité malicieuse qui monopoliserait le bus de communication en effectuant trop de requêtes et qui bloquerait ainsi le fonctionnement du système.

L'outil de CAO et le compilateur logiciel sont considérés de confiance. Ils ne peuvent pas être utilisés pour réaliser des modifications malicieuses de l'architecture. De même, les composants qui sont ajoutés pour la sécurité dans *TrustSoC-V* sont de confiance. L'ajout de ces composants est géré par l'outil de CAO et ne peuvent donc pas être modifiés. Nous supposons également que le SoC est fiable, aucune modification du circuit n'est possible (par exemple, l'ajout d'un cheval de Troie matériel). Le fondeur est également considéré comme fiable.

4.3.2 Définition des exigences de sécurité

Pour définir une architecture de SoC hétérogène sécurisée par conception, nous devons définir des exigences de sécurité. *TrustSoC-V* partage les mêmes exigences de sécurité que *TrustSoC* et *RTrustSoC* mais en ajoute également une nouvelle.

ES.1 : Règles de sécurité :

TrustSoC-V doit également être basée sur des règles de sécurité qui permettent de préciser le comportement autorisé du système. Ces règles de sécurité sont les mêmes que celles présentées pour *RTrustSoC*. Celles-ci sont précisées dans l'Annexe A de

cette thèse. *TrustSoC-V* étend les règles de sécurité pour le système de communication en stipulant que la communication doit s'effectuer sans l'utilisation d'IP matérielles propriétaires telles que l'*AXI Interconnect* (RS_CPU_FPGA4). Celles-ci peuvent être vulnérables aux attaques présentées dans [14].

ES.2 : Extension en un système multi-mondes sécurisé :

La proposition *TrustSoC-V* doit être basée sur le même système multi-mondes que *TrustSoC* et *RTrustSoC*. Ces mondes peuvent s'apparenter aux différents modes disponibles pour les processeurs RISC-V (utilisateur, superviseur, machine et hyperviseur) s'ils sont configurés de la même façon par le concepteur (niveaux de privilèges et nombre). En revanche, contrairement aux modes des processeurs RISC-V, les mondes de *TrustSoC-V* doivent s'étendre aux accélérateurs matériels.

ES.3 : Intégration des ressources logiques (type FPGA) dans la sécurité :

Comme mentionné dans l'ES.2, l'architecture *TrustSoC-V* doit intégrer des accélérateurs matériels (FPGA). Chacun de ces accélérateurs doit être affecté à un monde. Dans *TrustSoC-V*, les IP matérielles doivent être placées dans des enclaves matérielles et totalement isolées des parties qui ne sont pas de confiance du système.

ES.4 : Communications sécurisées à l'intérieur du SoC :

Compromettre le bus de communication du SoC-FPGA peut rendre le système inutilisable. Il est donc essentiel que les communications à l'intérieur du SoC soient sécurisées. *TrustSoC-V* doit garantir la sécurité des communications à l'intérieur du SoC. *TrustSoC-V* doit intégrer les changements nécessaires pour pouvoir transporter les signaux de sécurité directement. Il n'est plus nécessaire d'utiliser des signaux tiers d'un protocole de communication.

ES.5 : Résistance aux attaques de type SCA basées sur des mesures de temps d'accès de la mémoire cache :

TrustSoC-V doit embarquer les mêmes protections que *TrustSoC* et *RTrustSoC* contre les attaques SCA à distance basées sur des mesures de temps d'accès qui visent la mémoire cache du système que *RTrustSoC*. La mémoire cache doit être divisée en partitions et leur accès doit être restreint. Cette restriction doit s'effectuer grâce à des identifiants et à des contrôleurs de communication. Ces derniers doivent également mettre en place les règles de sécurité concernant la mémoire cache.

ES.6 : Architecture libre de droits :

TrustSoC-V doit être basée sur une technologie libre de droits. Ainsi chaque aspect de l'architecture doit pouvoir être entièrement modifiable par le concepteur. *TrustSoC-V* doit proposer un nouveau bus de communication open source et sécurisé à la conception. *TrustSoC-V* ne doit plus utiliser des technologies propriétaires comme le protocole AXI et l'IP *AXI interconnect*. Cependant, une compatibilité avec ces technologies est nécessaire pour pouvoir intégrer les IP matérielles disponibles sur le marché sans devoir fournir des efforts de conception trop importants.

4.3.3 Architecture *TrustSoC-V*

TrustSoC-V s'applique à une architecture de SoC hétérogène. Son objectif est d'offrir une flexibilité maximale au concepteur. L'architecture est entièrement modulable et s'adapte parfaitement aux besoins du concepteur. La Figure 4.1 présente le concept *TrustSoC-V*.

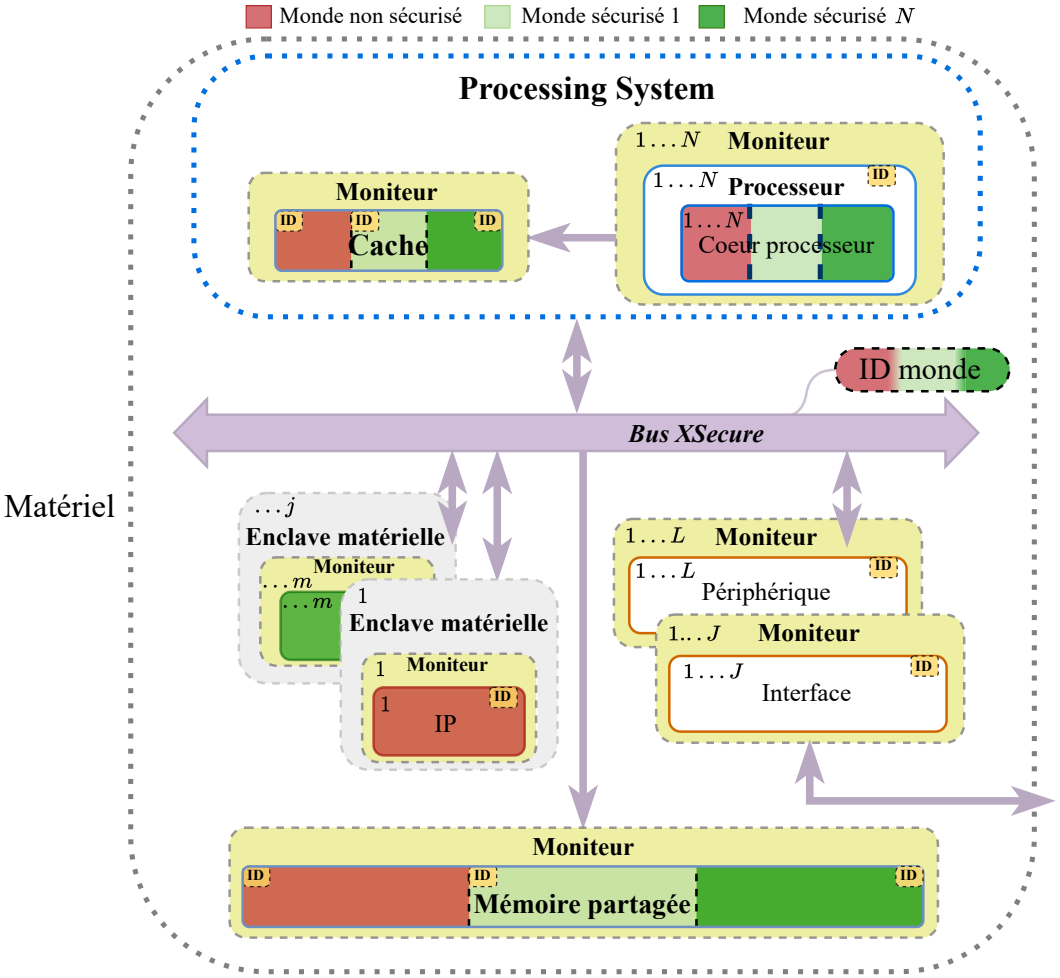


FIGURE 4.1 – Architecture du concept *TrustSoC-V*.

La partie *Processing System* embarque des processeurs RISC-V (de 1 à N) qui peuvent être multicœurs. *TrustSoC-V* dispose de la logique programmable (FPGA) avec jusqu'à m IP matérielles intégrées dans des enclaves matérielles (jusqu'à j et identifiées en gris clair). Il intègre également des périphériques (identifiés en orange dans la Figure 4.1) et des mémoires. Tous ces composants sont reliés par un nouveau bus de communication introduit par *TrustSoC-V*. Ce bus, appelé *XSecure*, est open source et sécurisé à la conception. Il est identifié en mauve dans la Figure 4.1.

Comme pour *TrustSoC* et *RTrustSoC*, *TrustSoC-V* est basée sur un système multi-mondes. L'architecture dispose d'un monde non sécurisé (identifié en rouge dans la Figure 4.1) et de mondes sécurisés de 1 à N (identifiés avec différentes nuances de vert Figure 4.1). Le nombre maximum de mondes sécurisés dépend des ressources disponibles dans le système. Chaque cœur de processeur est affecté à un monde et ne peut exécuter que des applications qui appartiennent à ce monde. De même, chaque IP matérielle implémentée dans la logique programmable (FPGA) est assignée à l'un des

mondes du système. Les mémoires partagées du SoC, quant à elles, sont partitionnées en fonction du nombre de mondes : une partition par monde (voir Figure 4.1).

La sécurité de *TrustSoC-V* est centrée sur les communications du SoC hétérogène. *TrustSoC-V* pousse les concepts présentés dans *TrustSoC* et *RTrustSoC* en y ajoutant le nouveau bus de communication (*XSecure*). Celui-ci a pour but d'empêcher les attaques qui manipulent le bus AXI présentées dans le Chapitre 1 et 3 de cette thèse et dans les articles [12][14][40][48]. L'idée derrière cette proposition est de s'affranchir des IP matérielles de bus tierces utilisées (*AXI interconnect*), qui peuvent être vulnérables et qui ne sont pas modifiables à cause de la technologie propriétaire (voir RS_CPU_FPGA4 dans l'Annexe A). Plus globalement, le bus permet de ne plus dépendre d'une technologie propriétaire où la documentation est parfois limitée et les modifications inaccessibles. Le bus sécurisé doit cependant être compatible avec ces technologies pour éviter des changements trop importants lors de l'ajout de nouvelles IP matérielles. Nous avons présenté les principaux protocoles de communication dans le Chapitre 2 de cette thèse et leurs fonctionnements. Pour que notre bus de communication sécurisé soit le plus compatible possible avec les protocoles AMBA-AXI [8] et TileLink [78], nous devons l'organiser en canaux (adresse, lecture, écriture, etc.) et utiliser un mécanisme de poignée de main avec des signaux *x_valid* et *x_ready* (*x* étant le canal de l'opération en cours).

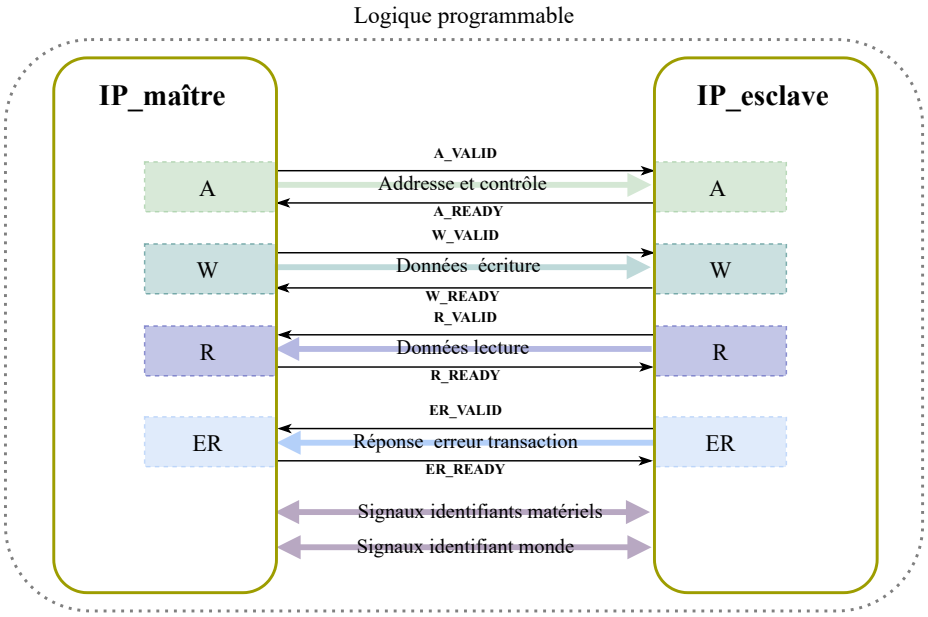


FIGURE 4.2 – Exemple de réalisation du bus de communication sécurisé *XSecure* proposé par *TrustSoC-V* entre deux IP matérielles.

La Figure 4.2 est un exemple d'une composition possible du bus *XSecure*. Dans cet exemple, Figure 4.2, il est composé de quatre canaux :

1. Le canal d'adresse des opérations *A* (identifié en vert Figure 4.2) :

   Il s'agit du canal pour transmettre l'adresse de l'esclave à laquelle l'interface maître souhaite écrire ou lire. Les signaux de poignée de main pour ces opérations sont *A_VALID* et *A_READY*.
2. Le canal des données à écrire *W* (identifié en bleu-vert Figure 4.2) :

Il s'agit du canal pour transmettre les données à écrire à l'adresse de l'esclave. Les signaux de poignée de main pour ces opérations sont *W_ VALID* et *W_ READY*. Le signal pour les données est nommé *W_ DATA* et a la taille du bus de données.

3. Le canal des données à lire *R* (identifié en violet Figure 4.2) :

   Il s'agit du canal pour transmettre les données à lire depuis l'adresse de l'esclave. Les signaux de poignée de main pour ces opérations sont *R_ VALID* et *R_ READY*. Le signal pour les données est nommé *R_ DATA* et a la taille du bus de données.

4. Le canal de réponse et d'erreur *ER* (identifié en bleu clair Figure 4.2) :

   Il s'agit du canal permettant de signaler si la transaction a rencontré une erreur. Les signaux sont mis à jour à la fin de la transaction via la poignée de main d'*ER_ VALID* et d'*ER_ READY*. Ainsi, en cas de détection d'un accès illégitime ou si la transaction rencontre un problème (adresse, etc.), des signaux d'erreur seront remontés. Ils correspondent aux signaux *XRESP*, utilisés pour *TrustSoC* et *RTrustSoC*. Cependant, les signaux d'erreur *ER* sont étendus pour couvrir plus de codes d'erreur. En effet, les signaux *XRESP* peuvent prendre comme valeurs : "00" pour *OKAY*, "11" pour *DECERR* et "10" pour *SLVERR*. Pour *TrustSoC-V*, des codes d'erreur sont ajoutés pour préciser l'erreur ou si une attaque est détectée, par exemple, "01" pour signifier un accès illégitime à une ressource sensible. L'encodage des codes d'erreur peut être augmenté aux besoins du concepteur pour en ajouter des supplémentaires.

En plus de ces quatre canaux, le bus intègre directement les signaux nécessaires au système multi-mondes (**ES.2**) et à la protection des communications (**ES.4**). Ces signaux sont illustrés en mauve dans la Figure 4.2. Ce sont les identifiants utilisés par les petits contrôleurs de communication matériels distribués "moniteurs" (identifiés en vert-jaune dans la Figure 4.1) pour mettre en place les communications sécurisées du SoC. Chaque IP matérielle dispose d'un identifiant monde et d'un identifiant matériel qui sont transmis lors des communications. L'encodage de ces identifiants est toujours $\lceil \log_2(\max(\text{nombre de mondes})) \rceil$ pour l'identifiant monde et pour celui de l'IP matérielle $\lceil \log_2(\max(\text{nombre de composants})) + 1 \rceil$ où $\lceil \cdot \rceil$ est la fonction d'arrondi à l'entier supérieur. L'encodage des signaux de sécurité du bus peut donc évoluer. Le bus *XSecure*, comme l'architecture, est modulable et s'adapte aux besoins du concepteur. Les architectures des contrôleurs sont présentées dans la Figure 4.3. Ces contrôleurs sont ajoutés à chaque interface de communication avec le bus du système, Figure 3.2 (A) pour les interfaces maîtres et Figure 3.2 (B) pour les interfaces esclaves. L'ajout de ces contrôleurs permet d'isoler toutes les interfaces de communication du bus de communication du système. Ainsi, aucun composant disposant d'une interface de communication du système n'est connecté directement au bus de communication, il en est isolé. Les requêtes de communication passent obligatoirement par un contrôleur de communication avant d'être transmises à une IP matérielle ou au bus de communication. Les contrôleurs de communication connectés aux interfaces esclaves du système sont presque identiques à ceux présentés dans la section *TrustSoC* du chapitre précédent. La seule différence est qu'ils sont placés dans une enclave matérielle identifiée en gris clair dans la Figure ci-dessous. Cette enclave matérielle permet d'ajouter une couche d'isolation supplémentaire entre les contrôleurs de communication et le reste du système. Comme pour *TrustSoC*, ces contrôleurs de communication disposent d'une table de permissions pour spécifier les droits de chaque composant du système à l'IP sous-jacente. Chaque composant est identifié par son identifiant matériel. Les droits

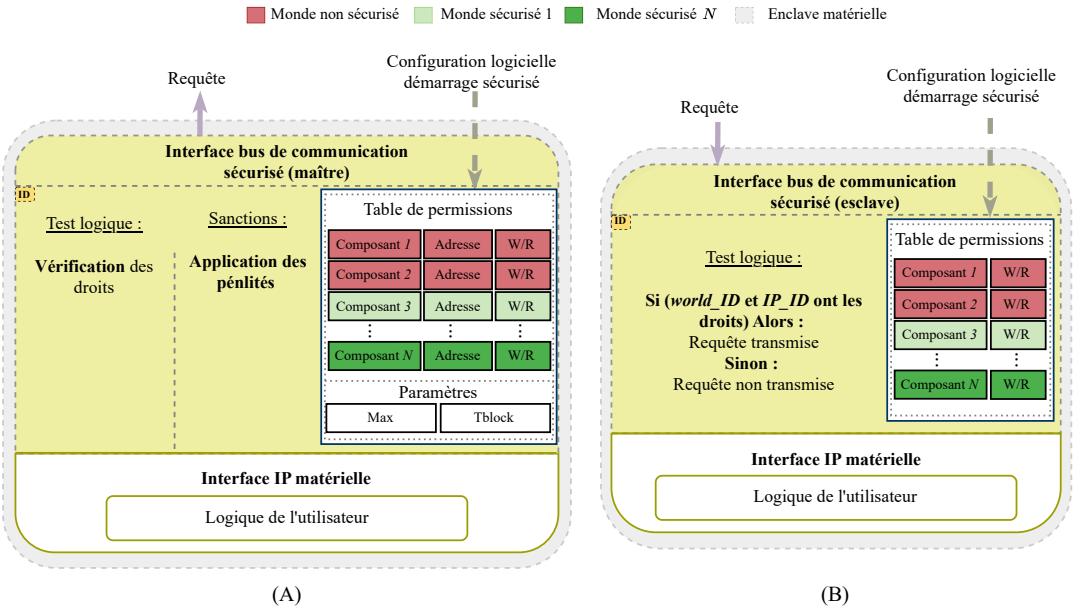


FIGURE 4.3 – Architectures des contrôleurs de communication “moniteur” de *TrustSoC-V* pour une interfaces maître (A) et une esclave (B).

sont codés sur deux bits : un pour l'écriture, un pour la lecture. Si leur valeur est ‘0’ cela signifie que l'émetteur n'a pas les droits, sinon la valeur est ‘1’. Ils disposent également d'une interface de communication générique avec le bus de communication, identifiée en mauve dans la Figure 4.3. Une fois la communication reçue, la partie test logique du contrôleur de communication analyse les droits de l'émetteur et transmet ou non la requête à l'IP matérielle.

En revanche, les contrôleurs de communication attachés aux interfaces maîtres du système diffèrent de ceux utilisés dans *RTrustSoC*. Ils n'appliquent plus de système de pénalités dynamiques, mais des pénalités fixes, comme dans l'architecture E-IIPS [11] (voir Chapitre 2). Cela permet de réduire le coût des contrôleurs de communication sur cette architecture. La vérification des communications se fait toujours avec l'aide d'une table de permissions. Celle-ci a une condition sur les adresses et possède des paramètres *Max* et *Tblock*. Ces tables et paramètres sont toujours reconfigurables. La reconfiguration est identifiée à l'aide d'une flèche en pointillés kaki dans la Figure 4.3. C'est grâce à cette reconfiguration et ces paramètres que *TrustSoC-V* diffère de l'architecture E-IIPS. En effet, dans E-IIPS, les pénalités sont fixes et non reconfigurables. Les auteurs ne mentionnent pas s'il y a la possibilité de régler avec un paramètre comme *Tblock*, le temps pendant lequel l'interface de communication est bloquée. La partie évaluation n'apparaît plus dans la Figure 4.3 (A), elle est réalisée dans la partie test logique. À chaque requête illégitime, elle incrémente un compteur qui une fois arrivé à la valeur seuil *Max* désactive l'interface de communication pendant un temps *Tblock*. Ce fonctionnement est plus détaillé dans la Figure 4.4. Nous avons repris la même configuration, un processeur souhaitant faire appel à une IP matérielle de traitement. Les deux composants disposent de contrôleurs de communication distribués. Dès qu'une requête est émise par le processeur, son contrôleur vérifie que la communication est activée et autorisée. Si c'est le cas, la requête est transmise. Le contrôleur de communication de l'IP matérielle vérifie la communication. Si les droits sont vérifiés, la requête est transmise. Dans le cas contraire, le contrôleur agit comme un pare-feu, bloque la communication et lève un signal d'erreur (identifié en

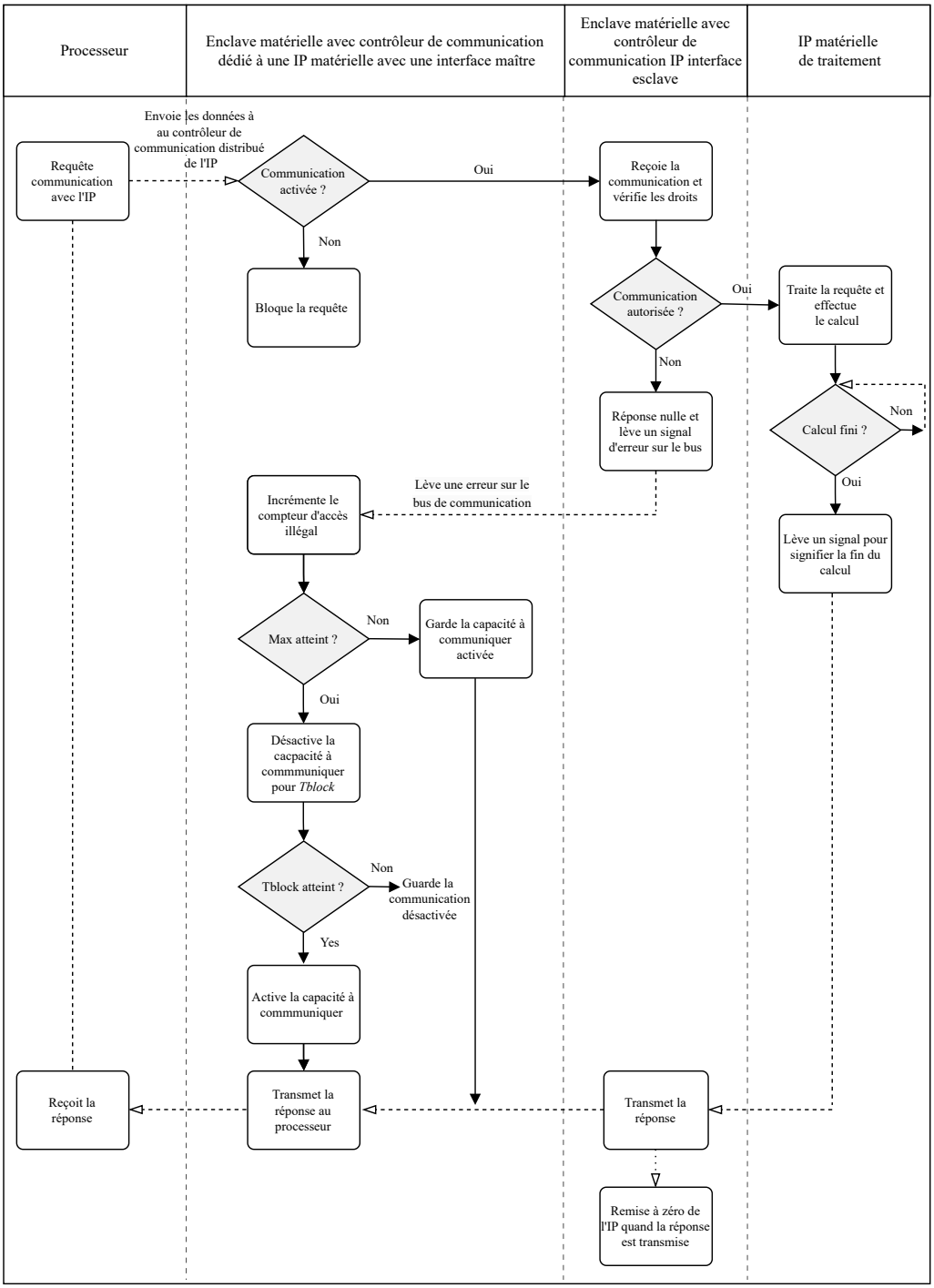


FIGURE 4.4 – Diagramme de séquence du fonctionnement des contrôleurs de communication "moniteurs" de *TrustSoC-V*.

bleu clair dans la Figure 4.2). Lorsque le premier contrôleur reçoit le signal d'erreur et que les droits ont été vérifiés comme non valides, le compteur de requêtes illégitimes est incrémenté. Si le seuil *Max* est atteint, l'interface de communication est désactivée pendant un temps *Tblock*. Une fois ce temps écoulé, le processeur peut à nouveau transmettre des requêtes.

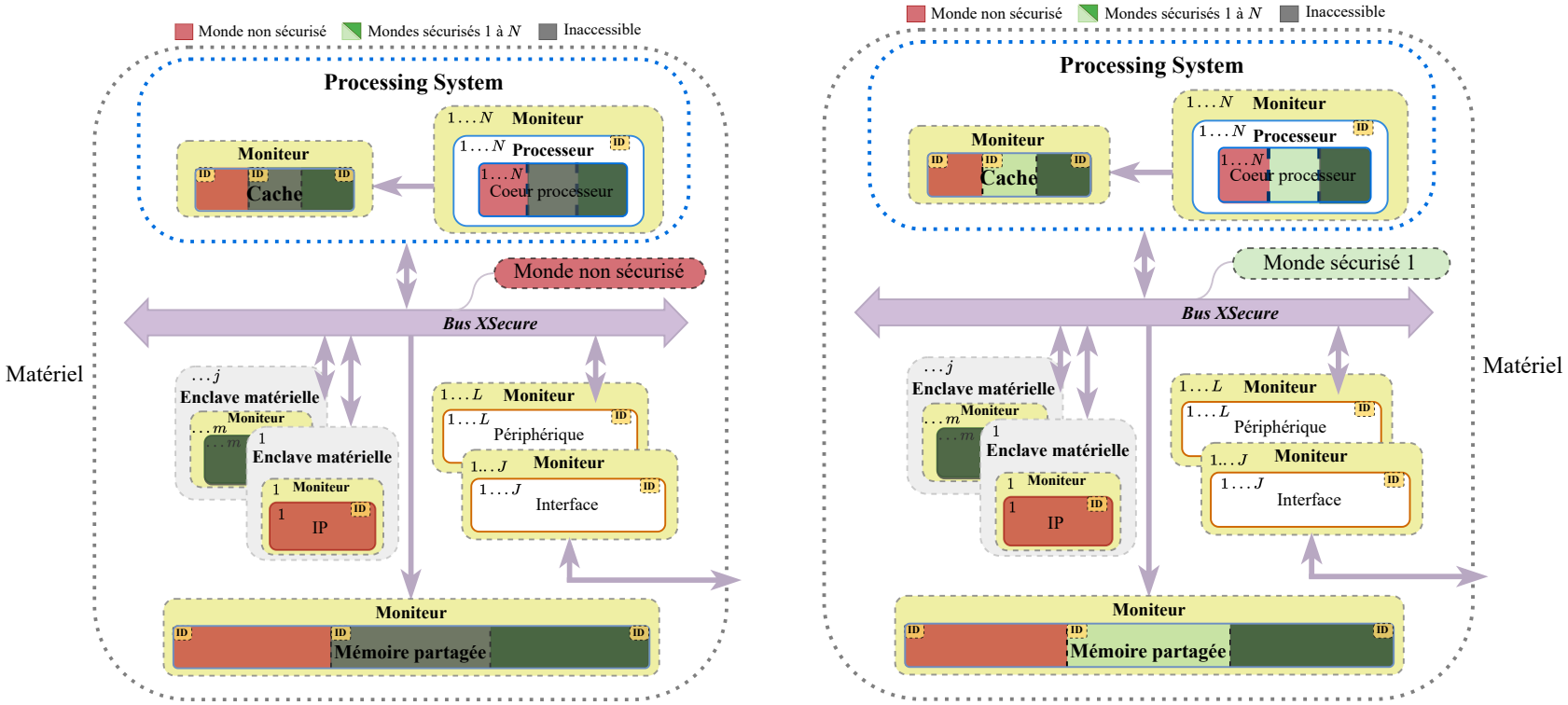
Les exigences de sécurité et la sécurité de *TrustSoC-V* reposent sur ces contrôleurs de communication. Ils permettent de faire respecter les règles de sécurité (**ES.1**) qui

encadrent le comportement de l'IP matérielle à laquelle ils sont attachés. Par exemple, pour les contrôleurs maîtres, ils vérifient que l'IP matérielle ne tente pas de réaliser une attaque DoS sur le système. S'ils détectent qu'une attaque DoS est en cours, ils agissent pour protéger le système grâce à un système de pénalités. L'interface de communication de l'IP matérielle à l'origine de l'attaque est désactivée pendant un temps *Tblock*. Le rôle des contrôleurs de communication ne s'arrête pas là. Ils mettent en place l'exigence de sécurité n°4 (**ES.4** : communications sécurisées à l'intérieur du SoC). Avec *TrustSoC-V*, nous avons même étendu la mise en application de cette exigence en proposant *XSecure*, un nouveau bus sécurisé par conception. Le bus *XSecure* et les contrôleurs de communication créent les voies de communication sécurisées entre tous les composants logiciels ou matériels du système (applications, accélérateurs matériels, les périphériques, mémoires, etc.). Ils permettent également de satisfaire à l'exigence de sécurité n°3 (**ES.3**), qui consiste à intégrer les ressources logiques (FPGA) dans la sécurité du système. Les ressources logiques bénéficient d'un contrôle complet des communications effectuées vers et depuis les accélérateurs matériels. Elles bénéficient également du nouveau bus sécurisé qui empêche le changement du contenu des communications présenté dans [12]. *TrustSoC-V* continue d'intégrer et développer le concept d'IP matérielles de confiance qui vise à traduire le même concept d'applications de confiance dans un TEE pour les IP matérielles intégrées dans le FPGA. Les IP matérielles appartenant aux mondes sécurisés sont garanties de fonctionner comme voulu et, avec l'ajout des enclaves matérielles, elles bénéficient d'une isolation supplémentaire avec les parties qui ne sont pas de confiance du système.

Les contrôleurs contribuent également à faire respecter l'exigence de sécurité n°2, à savoir l'extension en un système multi-mondes sécurisés, qui est présentée dans la sous-section suivante.

4.3.4 Fonctionnement

Nous présentons le fonctionnement de *TrustSoC-V* dans différents mondes dans la Figure 4.5. La Figure 4.5 (A) présente le fonctionnement dans le monde non sécurisé (ID monde = "monde non sécurisé"), illustré en rouge. La Figure 4.5 (B) présente le fonctionnement dans le monde sécurisé 1 (ID monde = "monde sécurisé 1"), identifié en vert clair. Les N mondes sécurisés restants sont identifiés en vert foncé. La Figure 4.5 permet également de représenter les ressources inaccessibles du système par le monde qui est exécuté, elles sont représentées en blocs gris foncé. Nous proposons d'étudier le fonctionnement du système dans chacun de ces mondes : monde non sécurisé et monde sécurisé n°1. *TrustSoC-V* fonctionne de la même manière que *TrustSoC* et *RTrustSoC*. Autrement dit, lorsque le système fonctionne dans le monde non sécurisé, les ressources appartenant aux mondes sécurisés sont inaccessibles par ce dernier. Ce sont les blocs gris placés sur les ressources identifiées initialement en nuances de vert qui permettent de l'illustrer dans la Figure 4.5. Parmi les ressources inaccessibles aux ressources non sécurisées, se trouvent les partitions sécurisées des mémoires (mémoire cache et mémoire partagée), les IP matérielles de confiance et les cœurs des processeurs de confiance. Aucune modification de ces ressources de confiance ne peut être effectuée par les ressources non sécurisées. Ce sont les contrôleurs de communication (moniteurs) qui visent à faire respecter ces règles comme présenté dans la section précédente. Ils garantissent que les ressources de confiance du système sont inaccessibles et isolées des ressources non sécurisées en effectuant un contrôle systématique.



(A) Fonctionnement de *TrustSoC-V* dans le monde non sécurisé.

(B) Fonctionnement de *TrustSoC-V* dans le monde sécurisé n°1.

FIGURE 4.5 – Fonctionnement de *TrustSoC-V* dans le monde non sécurisé (A) et dans un des mondes sécurisés (B).

Cependant, quand le système fonctionne dans un des mondes sécurisés, ce dernier a accès aux ressources du monde non sécurisé. Cela peut permettre à une application de confiance de déléguer des parties non sensibles pour un traitement. Ces cas de fonctionnement sont couverts par les règles de sécurité afin d'empêcher toute attaque réutilisant les données. Cela est, par exemple, réalisé avec la réinitialisation automatique par le contrôleur de communication quand le traitement est terminé de l'IP matérielle. Cette étape est présentée dans la Figure 4.4. Une règle similaire s'applique aux partitions de la mémoire cache. Lors d'un changement de contexte, les différentes partitions de la mémoire cache qui ont été utilisées par des mondes sécurisés sont vidées. Cette règle réduit les performances du système, mais offre un meilleur niveau de sécurité contre les attaques qui visent la mémoire cache. Ces règles et la modification de la mémoire cache rendue possible grâce à l'utilisation du processeur RISC-V permettent de satisfaire à l'exigence de sécurité **ES.5** : résistance aux attaques SCA basées sur des mesures de temps d'accès qui visent la mémoire cache.

4.3.5 Résultats d'implémentation

Dans cette section, nous présentons tous les résultats d'implémentation de *TrustSoC-V* que nous avons obtenus sur un SoC-FPGA d'AMD embarquant un Zynq-7000 (XC7Z020-CLG484). Tous les résultats de *TrustSoC-V* sont obtenus en utilisant le logiciel de CAO Vivado 2020.2 d'AMD. Pour les différentes implémentations, nous avons utilisé le langage de description matérielle (VHDL) et du *SystemVerilog* (pour le cœur RISC-V). Cependant, avant de présenter *TrustSoC-V*, nous allons parler du choix du processeur RISC-V que nous avons fait pour notre proposition.

Choix du cœur RISC-V

Pour implémenter *TrustSoC-V*, nous avons besoin d'un cœur RISC-V. Nous avons donc commencé par chercher ceux qui étaient disponibles. Nous avons trouvé plusieurs candidats : Rocket RISC-V [9], RI5CY [35], CVA6 (Ariane) [89], CV32A6 [83] et BOOM [74].

Nous avions fixé des critères :

- Surcoût relativement faible (version embarquée préférable) pour être implémenté sur un un SoC-FPGA d'AMD avec Z-7000 (XC7Z020-CLG484).
- Facile à mettre en œuvre.
- Communauté forte et réactive.

En fonction de nos critères, nous avons pu éliminer BOOM qui est un processeur à exécution dans le désordre donc trop complexe. Nous avons ensuite éliminé le processeur Rocket et CVA6 en raison de leurs tailles importantes. Nous avons fini par choisir le CV32A6, car il présentait, grâce au concours étudiant proposé par l'entreprise Thales et le GDR SoC2, une version déjà précompilée et prête à être mise en œuvre. Il dispose également d'une communauté assez importante et active, ce qui représente un avantage intéressant lors de l'implémentation. Le CV32A6 est la version 32 bits du CVA6. Ils partagent le même code source. Le CV32A6 est un processeur *single issue* à exécution dans l'ordre avec un pipeline à six étages. L'architecture matérielle globale du processeur est donnée dans la Figure 4.6 [1].

1. À retrouver à : `https://docs.openhwgroup.org/projects/cva6-user-manual/05_cva6_apu/cva6_testharness.html`

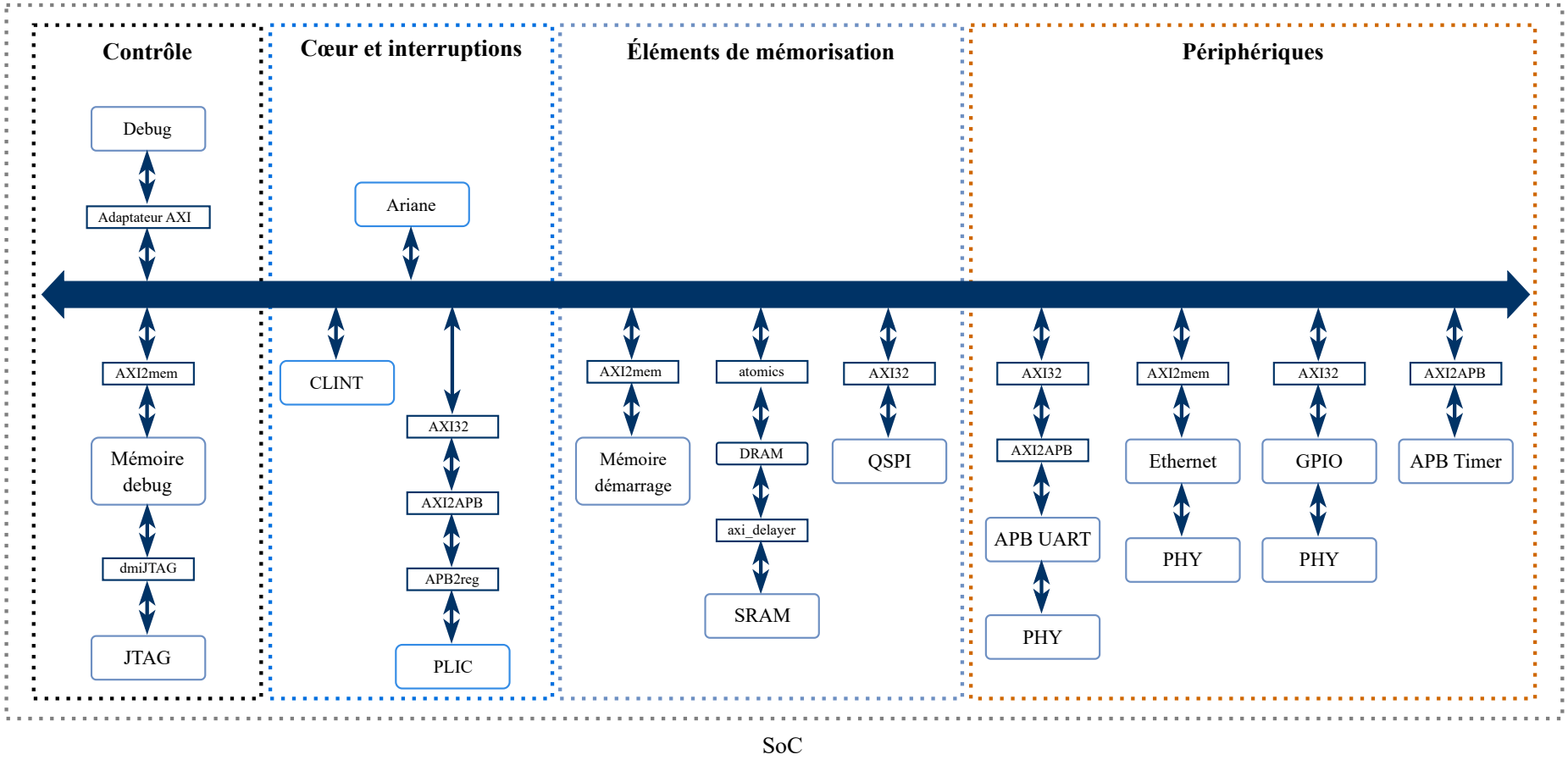


FIGURE 4.6 – Architecture matérielle CV32A6.

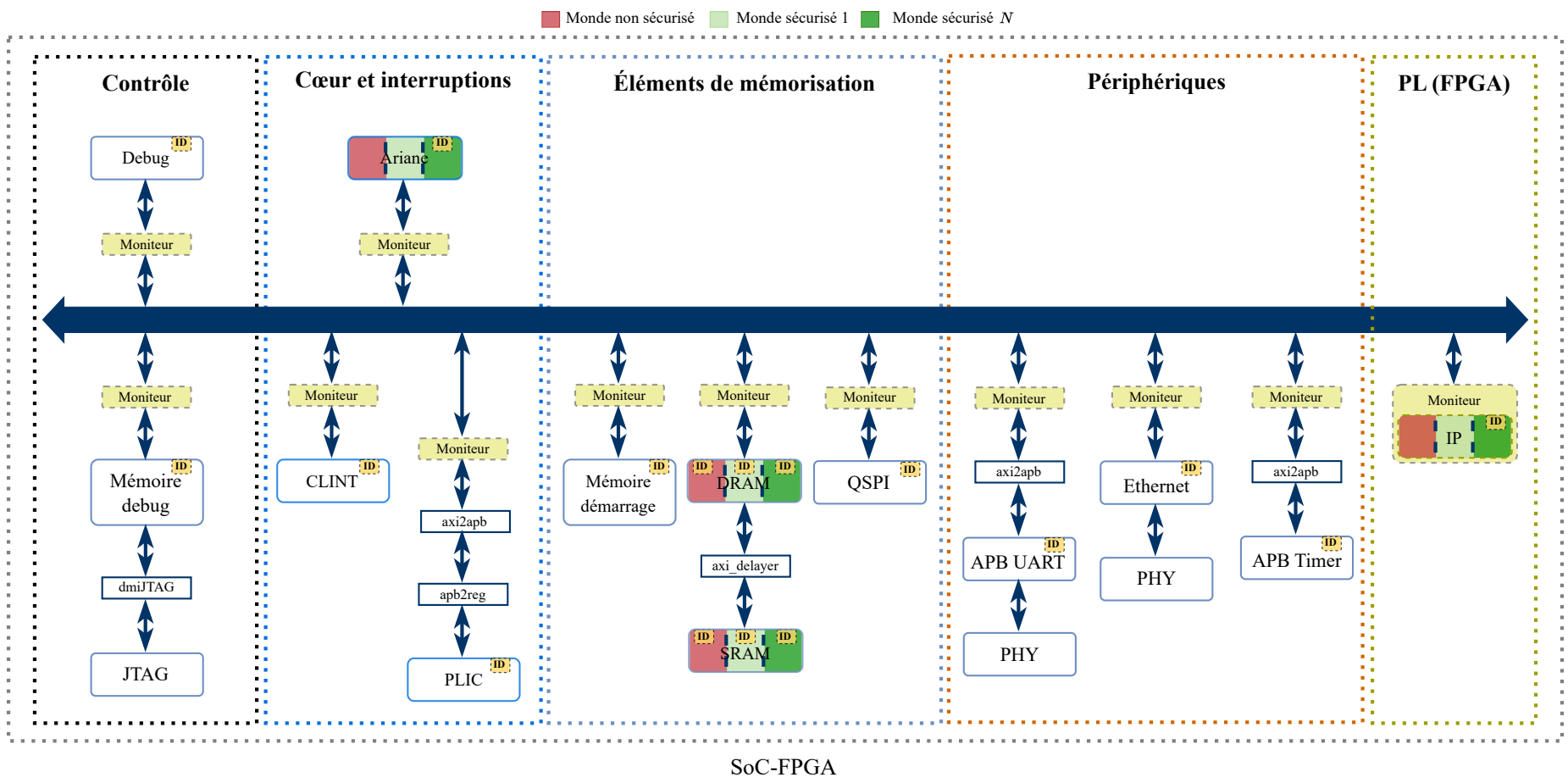


FIGURE 4.7 – Architecture matérielle modifiée CV32A6 pour ajouter *TrustSoC-V*.

Elle est divisée en quatre parties :

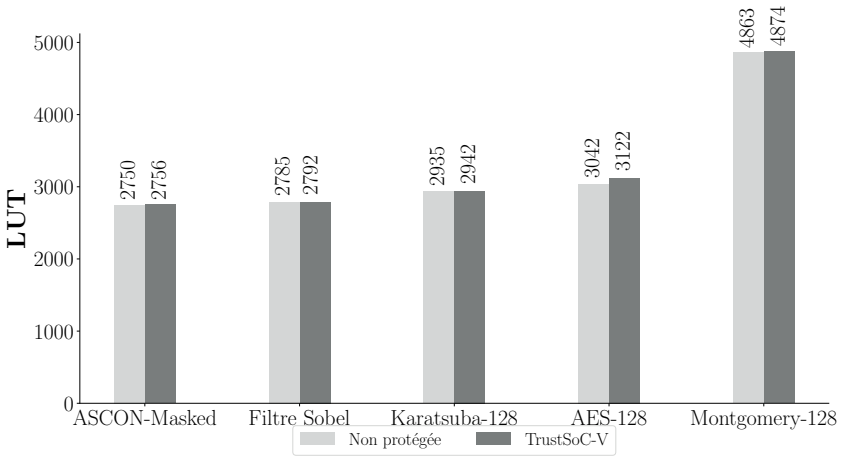
— Une partie contrôle (identifiée en pointillés noirs dans la Figure 4.6) avec les modules de *Debug* et de JTAG.
— Une partie *Processing System* (identifiée en pointillés bleus dans la Figure 4.6) avec le processeur Ariane en version 32 bits et des modules pour gérer les interruptions CLINT (*Core Local Interrupt Controller*) et PLIC (*Platform Level Interrupt Controller*).
— Une partie concernant les éléments de mémorisation (identifiée en pointillés bleu clair dans la Figure 4.6) avec la mémoire pour le démarrage, la DRAM, la SRAM (*Static Random Access Memory*) et l'interface QSPI (*Quad Serial Peripheral Interface*).
— Une partie périphérique (identifiée en pointillés orange dans la Figure 4.6) avec une liaison UART (*Universal Asynchronous Receiver Transmitter*) qui permet de communiquer avec le cœur RISC-V, une interface Ethernet, une interface GPIO (*General Purpose Input/Output*) et un compteur.

Toutes les parties partagent le bus de communication du système qui est un bus de la norme AMBA-AXI4 (identifié en bleu foncé dans la Figure 4.6).

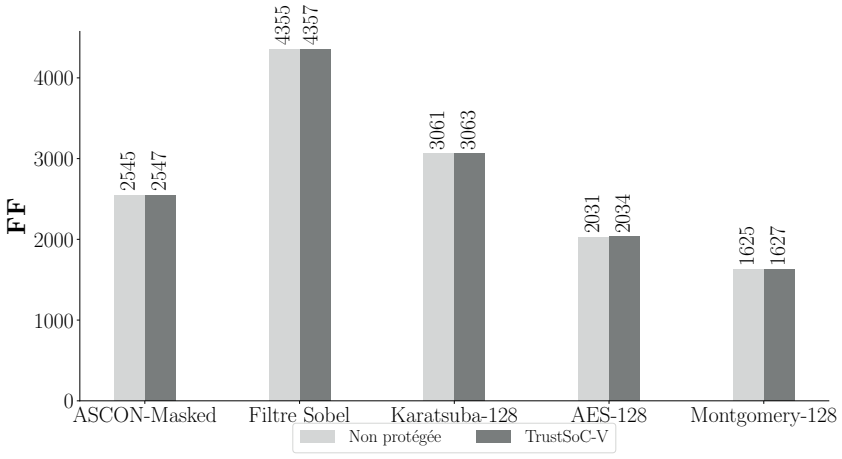
Résultats d'implémentation

Pour l'implémentation de *TrustSoC-V*, nous proposons une démarche par étapes. Elle consiste, en premier lieu, à commencer par modifier l'architecture pour ajouter les contrôleurs de communication (moniteurs) et la partie logique programmable (FPGA). Le bus de communication AXI est conservé. Dans un deuxième temps, quand tous les moniteurs sont ajoutés et testés, nous pourrons remplacer ce bus AXI par le nouveau bus de communication *XSecure*. L'architecture avec les modifications pour ajouter les contrôleurs de communication est présentée dans la Figure 4.7. Les contrôleurs sont identifiés en vert-jaune. La partie liée au FPGA où nous représentons une IP matérielle est également identifiée en vert-jaune dans la Figure 4.7. Les identifiants nécessaires aux contrôleurs de communication apparaissent dans les blocs jaunes. Le système multi-mondes apparaît dans cette figure avec du rouge pour illustrer le monde non sécurisé et différentes nuances de vert pour représenter les mondes sécurisés de 1 à N. Le choix est laissé au concepteur pour le monde auquel appartiennent les applications et les IP matérielles. Nous les représentons donc dans les trois couleurs (rouge, vert clair et vert foncé). Dans cette architecture, les contrôleurs de communication font la liaison en AXI4 avec le bus de communication et les différents protocoles des IP matérielles (AXI2mem, AXI2APB, AXI32).

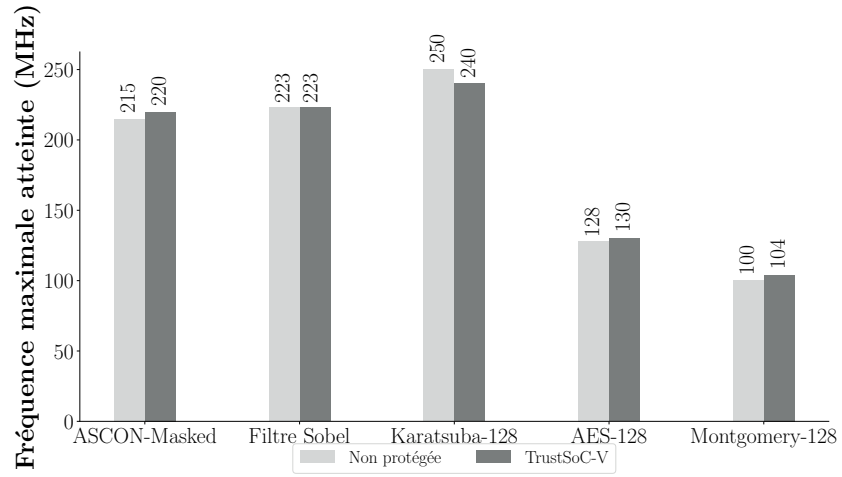
Nous avons implémenté un contrôleur de communication esclave sur les mêmes IP matérielles que *TrustSoC* : un filtre de Sobel, une implémentation ASCON [29], Karatsuba-128, AES-128 [1] et Montgommery-128. Nous avons obtenu les résultats présentés dans la Figure 4.8 pour une petite configuration : deux mondes et deux composants avec des identifiants. L'architecture des implémentations est identique à celle de l'IP matérielle présentée dans la Figure 4.7. Nous avons pris la même petite configuration que pour *TrustSoC* pour montrer le coût de la partie test logique du contrôleur de communication. L'exploration du coût des contrôleurs de communication pour des systèmes plus importants est présentée dans la Figure 3.9. Les résultats de la Figure 4.8 en gris clair correspondent à l'IP matérielle non protégée et ceux en gris foncé à l'IP matérielle avec le contrôleur de communication de *TrustSoC-V*. La petite configuration nous permet de réduire au maximum le coût lié aux tables de permissions et de ne montrer que le coût de la logique de test du contrôleur. Dans ces implémentations la taille des signaux utilisateurs est fixée à trois bits : un pour l'identifiant du



(A) LUT.



(B) FF.



(C) FMAX.

FIGURE 4.8 – Résultats d'implémentation en LUT, FF et fréquence maximum atteignable d'un contrôleur de communication de *TrustSoC-V* pour cinq IP matérielles différentes pour une petite configuration sur un SoC-FPGA AMD Zynq-7000.

monde et deux pour les identifiants des IP matérielles. Les résultats de la Figure 4.8 montrent que le coût de la logique de *TrustSoC-V* en ressources est faible. L'ajout

d'un contrôleur de communication entraîne une augmentation moyenne de 0,72% en LUT et de 0,1% en FF. La différence de deux FF (sauf AES) correspond aux deux bits de droits (un pour l'écriture, un pour la lecture). L'IP matérielle AES possède un signal supplémentaire pour son fonctionnement donc rajoute une FF supplémentaire. Nous avons également évalué l'influence de l'ajout du contrôleur de communication sur la fréquence maximale de fonctionnement. D'après nos résultats, les contrôleurs de communication de *TrustSoC-V* n'ont pas d'impact significatif sur cette dernière. Les fluctuations apparaissant sur la Figure 4.8 sont dues au processus de synthèse.

4.4 RISC-B

Avec l'implémentation des contrôleurs de *TrustSoC-V*, présentée dans la sous-section précédente, nous avons réalisé que des changements onéreux étaient nécessaires. Ainsi, pour faciliter la mise en œuvre de *TrustSoC-V* par un concepteur, nous présentons RISC-B (ce travail est actuellement en soumission à une conférence internationale [53]). Cette section présente la contribution RISC-B, sa méthodologie et les résultats d'implémentation que nous avons obtenus.

Pour commencer, nous présentons la méthodologie de RISC-B.

4.4.1 Objectifs et motivations de RISC-B

L'objectif principal de notre contribution, RISC-B, est de fournir une solution pour faciliter l'implémentation matérielle de *TrustSoC-V* par un concepteur. Cette solution doit être clé en main et ne nécessiter aucune modification spécifique pour chaque changement de cœur de RISC-V. Elle doit donc être adaptable à différents systèmes.

4.4.2 Méthodologie

Il s'agit de proposer une nouvelle manière d'implémenter l'architecture complète d'un système utilisant un processeur RISC-V. La Figure 4.9 illustre les étapes de cette méthodologie. Pour ce faire, nous considérons l'architecture bloc par bloc. Chaque bloc est spécifié indépendamment des autres. Il s'agit de l'étape représentée en vert dans la Figure 4.9. Puis, nous utilisons l'interface graphique du logiciel de CAO Vivado pour interconnecter les blocs (étape illustrée en vert-bleu). Le niveau supérieur de la hiérarchie d'un bloc (appelé *top level*) est spécifié en langage de description matérielle (par exemple : VHDL). Les blocs sont entièrement configurables grâce à des paramètres. Ces blocs peuvent cependant utiliser dans les fichiers de leurs arborescences d'autres langages, comme le *SystemVerilog*. Une fois ces deux premières étapes effectuées, l'architecture peut être synthétisée et implémentée. Il s'agit de l'étape en bleu clair dans la Figure 4.9. À la fin du processus, nous obtenons un prototype fonctionnel. Cette phase est identifiée en violet.

RISC-B permet de simplifier grandement le processus d'intégration en ajoutant ou supprimant des blocs, en les interconnectant et en les configurant de manière graphique. Un exemple de cette représentation graphique est donné dans la Figure 4.10. Tous les blocs du système CV32A6 [83] ont été séparés et certains fusionnés dans des IP matérielles indépendantes. Ils sont ensuite reliés de manière graphique grâce au logiciel de CAO Vivado. Ce prototype de RISC-B présenté dans la Figure 4.10 est fonctionnel. Cette nouvelle manière d'implémenter le système s'inspire beaucoup de la conception sur FPGA. Comme avec les IP matérielles, les composants du système sont considérés bloc par bloc. Ils peuvent être spécifiés indépendamment les uns

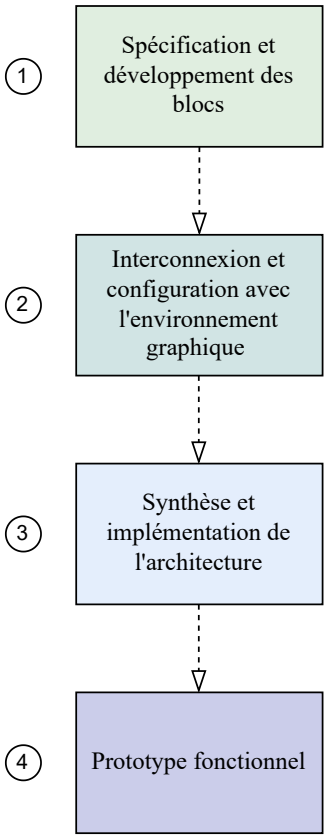


FIGURE 4.9 – Étapes numérotées de 1 à 4 de la méthodologie de *RISC-B*.

des autres en même temps. Ils sont interconnectés à la fin de la conception via un environnement graphique comme le logiciel de CAO Vivado d'AMD. Cette manière d'implémenter le système permet de simplifier grandement le processus d'intégration. Ainsi, nous réduisons le temps et la complexité de la conception. Ce travail est encore un travail en cours. Nous avons créé un GitHub qui sera disponible quand le travail sera terminé et publié, afin de permettre à d'autres de reproduire nos travaux.

4.4.3 Résultats d'implémentation

Le Tableau 4.1 présente les résultats obtenus avec cette nouvelle méthode d'implémentation pour le RISC-V CV32A6. Il compare également le coût de notre solution à celui du processeur fourni par l'entreprise Thales et le GDR SoC2 sur le GitHub [83]. Grâce à la considération des différents composants en IP matérielles indépendantes, en plus de la simplification de la conception, nous observons une réduction des ressources totales utilisées pour le système. Cette réduction est d'environ 2,3% pour les LUT et d'environ de 3,7% pour les FF par rapport à la version de base du CV32A6. Elle peut être liée aux simplifications des blocs avec l'utilisation du langage de description matérielle et le paramétrage graphique mais aussi aux améliorations réalisées par l'outil de synthèse. Celles-ci sont liées à la séparation et la fusion de certains blocs entre eux. Ce qui permet à l'outil de synthèse de réaliser des optimisations qui ne sont pas possibles dans la version de base du CV32A6 et d'obtenir une baisse de l'utilisation des ressources. Cette réduction a été obtenue pour le cœur CV32A6 mais n'est pas à généraliser pour d'autres cœurs sans résultats supplémentaires.

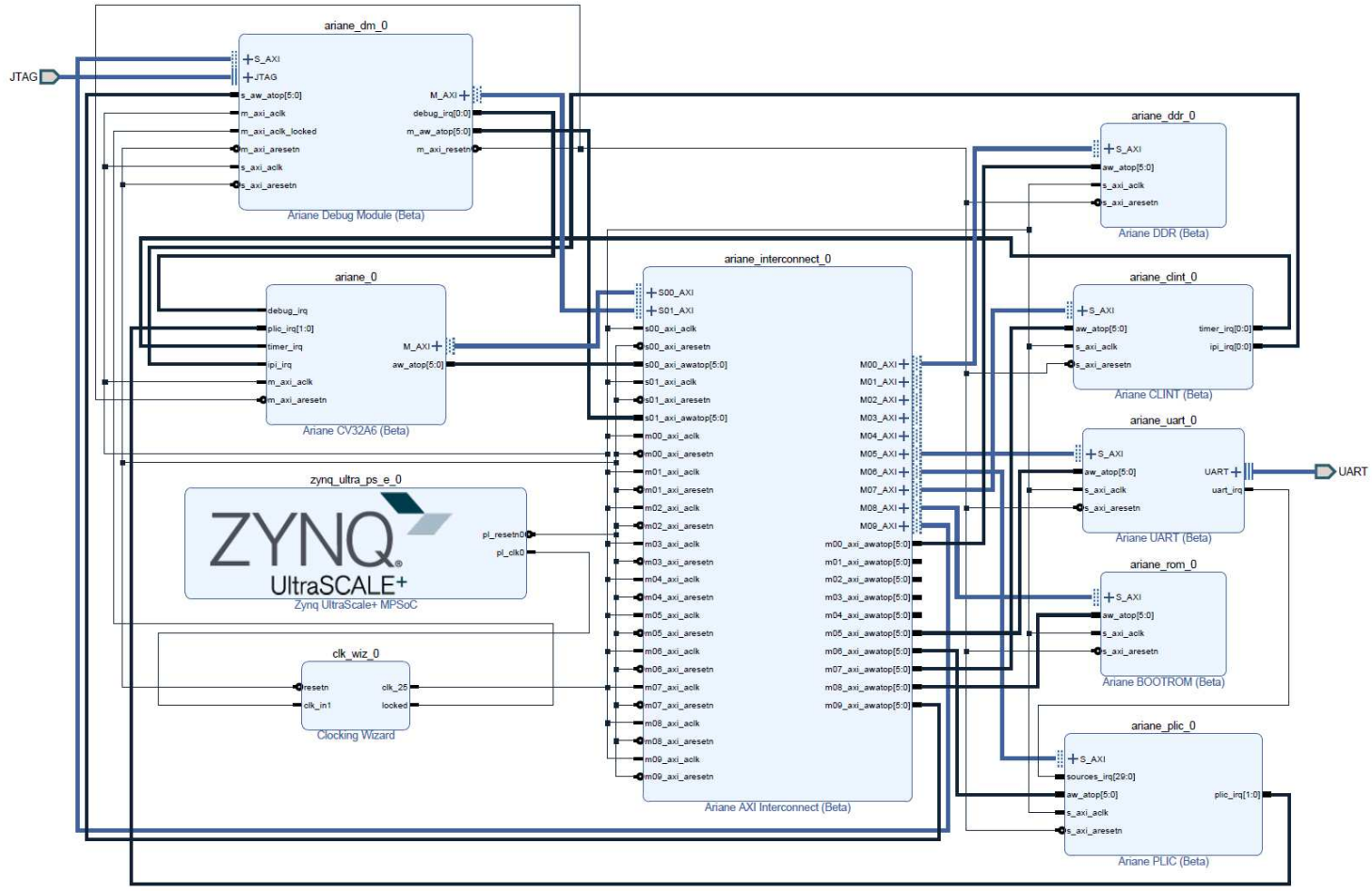


FIGURE 4.10 – Architecture matérielle prototype RISC-B obtenu sur le logiciel de CAO Vivado.

TABLE 4.1 – Résultats implémentation pour la version de base [83] et la version RISC-B en fonction des ressources LUT, FF et BRAM sur ZCU104.

Base	LUT	FF	BRAM	RISC-B	LUT	FF	BRAM
Ariane	13 571	9 274	36	Ariane	13 533	9 274	36
Dwidth_dm_master	272	360	0				
Dwidth_dm_slave	396	456	0				
Dm_axi2mem	105	48	0				
Dm_axi_master	20	37	0	Ariane_dm	1 973	1 917	0
Dm_top	905	676	0				
Dmi_jtag	148	339	0				
Rstgen_main	0	5	0				
Sous-total	1 819	1 921	0				
Block mem. gen	305	100	33				
RISCV_atomics	1 280	569	0	Ariane_ddr	1 175	560	33
Sous-total	1 585	669	33				
Clint	206	152	0	Ariane_clint	163	153	0
				Ariane_plic	1 217	810	0
Ariane_peripherals	1 864	2 736	0	Ariane_uart	943	1 668	0
				Sous-total	2 160	2 478	0
Axi2rom	95	40	0				
Bootrom	65	9	2	Ariane_rom	170	48	2
Sous-total	160	49	2				
Axi_xbar	4 888	6 773	0	Ariane_xbar	4 366	6 342	0
TOTAL	24 084	21 575	71	TOTAL	23 539	20 772	71
Overhead (%)					-2,26	-3,72	0

4.5 Limitations de la proposition

Comme pour *TrustSoC* et *RTrustSoC*, notre proposition de *TrustSoC-V* s'appuie toujours sur des technologies propriétaires. En effet, nous utilisons le logiciel de CAO Vivado de la société AMD pour la méthodologie RISC-B. RISC-B est utilisée par la suite pour la mise en place de *TrustSoC-V*. Cet usage limite toutefois notre proposition. C'est pourquoi nous avons cherché à nous affranchir de la partie implémentation sur le matériel et à proposer un modèle plus abstrait. Celui-ci nous permettrait d'explorer plus facilement l'espace de conception sans les changements chronophages liés à l'implémentation sur le matériel. Nous présentons cette contribution, appelée *TrustSoC-M*, dans le chapitre suivant.

4.6 Conclusion du chapitre

Ce chapitre a présenté une architecture sécurisée à la conception de SoC hétérogène pour pallier aux limitations identifiées lors du chapitre précédent. Cette nouvelle architecture, appelée *TrustSoC-V*, répond à un modèle de menaces précis et des exigences de sécurité qui ont été explicitées dans ce chapitre. Elle est basée sur les mêmes concepts que les précédentes architectures (voir aux chapitres précédents). Elle repose sur le même partitionnement en multi-mondes sécurisés. *TrustSoC-V* intègre également la notion d'IP matérielle de confiance. L'architecture introduit un nouveau bus de communication sécurisé par conception, libre et nommé *Xsecure*. Il vise à renforcer la sécurité des communications au sein du SoC-FPGA. Ce chapitre a présenté des résultats d'implémentation pour différentes IP matérielles. Ils montrent les surcoûts (matériels et temporels) induits par l'ajout des contrôleurs de communication. Il a

également permis de présenter les résultats obtenus avec RISC-B qui définit une nouvelle méthode pour concevoir les architectures embarquant des processeurs RISC-V. Enfin, ce chapitre a permis de mettre en évidence les limitations de cette proposition liées aux implémentations matérielles. Nous avons dès lors cherché à augmenter le niveau d'abstraction de notre solution et à explorer l'espace de conception disponible. Nous présentons cette contribution, appelée *TrustSoC-M*, dans le chapitre suivant.

Chapitre 5

TrustSoC-M

Résumé

Ce chapitre présente une dernière contribution : *TrustSoC-M*. Cette contribution permet de modéliser l'architecture *TrustSoC* avec ses éléments ainsi que les interactions entre ces derniers en se basant sur une méthode MBSE (*Model-Based System Engineering*). Nous commençons par expliciter les objectifs et les motivations de cette contribution. Nous rappelons le modèle de menaces considéré ainsi que les exigences de sécurité de l'architecture. Nous expliquons la mise en œuvre de *TrustSoC-M* et présentons les résultats que nous avons obtenus. Nous terminons ce chapitre par une perspective de cette contribution.

Table des matières

5.1 Introduction au chapitre 102
5.2 Motivations 103
5.3 *TrustSoC-M* 103
5.3.1 Modèle de menaces considéré 103
5.3.2 Exigences de sécurité 104
5.3.3 Définition du modèle 104
5.3.4 Mise en place de *TrustSoC-M* 109
5.4 Perspective pour *TrustSoC-M* 116
5.5 Conclusion du chapitre 116

5.1 Introduction au chapitre

La méthode MBSE pour *Model-Based System Engineering*, aussi appelée cycle en V en français, est illustrée dans la Figure 5.1. Cette dernière utilise des modèles tout au long du processus de développement d'un système. Ils permettent de représenter le système de manière plus simplifiée. Ils éliminent une partie de sa complexité avec une représentation plus abstraite. Cependant, le comportement et la structure restent les mêmes. Le modèle représente avec précision le système et réciproquement, le système confirme le modèle. Le processus MBSE, illustré dans la Figure 5.1, commence par les spécifications du système et se termine par un prototype fonctionnel de ce dernier. Le modèle comprend également des étapes de test, identifiées à l'aide de flèches en pointillés. Une validation est nécessaire entre chaque étape. Celles-ci sont identifiées par des coches dans la Figure 5.1.

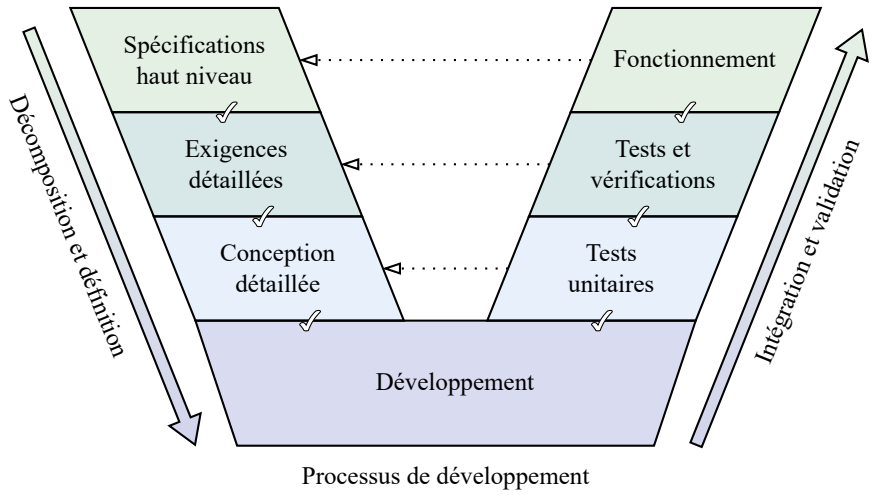


FIGURE 5.1 – Représentation du processus de développement par approche MBSE.

Cette méthode MBSE peut être utilisée pendant la conception d'un SoC hétérogène. Par exemple, le modèle [73], qui utilise le standard MARTRE (*Modeling and Analysis of Real-Time and Embedded systems*), offre une méthode pour la co-conception logicielle matérielle d'un SoC. Nous pouvons également citer la méthode Gaspard2 [68], qui permet de générer un système décrit en *SystemC* et exécutable sur une plateforme à partir de spécifications MARTRE haut niveau. Nous pouvons aussi mentionner la synthèse haut niveau HLS (*High-Level Synthesis*) comme Vitis HLS [6] ou Intel HLS [46]. Elle permet à partir à partir d'une description comportementale abstraite (C, C++) d'un système de générer la description de la structure de celui-ci (VHDL, RTL).

Ces travaux montrent que la méthode d'ingénierie des systèmes basée sur des modèles présente des avantages intéressants pour la conception d'un SoC hétérogène. Le modèle permet d'identifier les parties qui peuvent s'avérer critiques lors de l'implémentation matérielle. Il permet également d'explorer différentes possibilités de conception afin de trouver le choix le plus approprié. Dans la conception de SoC hétérogènes orientée vers la sécurité, la méthode MBSE permet de générer et de simuler des scénarios afin d'en identifier qui seraient bloquants pour l'architecture avant l'implémentation matérielle. Ce sont toutes ces caractéristiques et avantages des méthodes MBSE que nous voulons mettre à profit dans *TrustSoC-M*.

La contribution *TrustSoC-M* a donné lieu à une publication [57] et une présentation dans le *workshop* international *Rapid System Prototyping* (RSP) en 2023 à Hambourg (Allemagne).

5.2 Motivations

La contribution *TrustSoC-M* a pour objectif de :

- Proposer une approche MBSE qui fournit une représentation exacte de l'architecture *TrustSoC* via un modèle.
- Proposer un modèle qui supporte la restructuration de l'architecture pour réaliser l'exploration de l'espace de conception.
- Proposer un environnement de simulation associé qui permette de tester de multiples scénarios plus rapidement afin de vérifier le fonctionnement de l'architecture.
- Fournir un générateur de description matérielle qui permette, à partir du modèle, de produire l'architecture *TrustSoC* avec le code de description matérielle associé (VHDL, *Verilog*).

5.3 *TrustSoC-M*

Cette section présente *TrustSoC-M*. Nous commençons d'abord par rappeler le modèle de menaces considéré par *TrustSoC* ainsi que ses exigences de sécurité. Nous détaillons ensuite le modèle et son fonctionnement. Nous expliquons la mise en place du modèle et fournissons les résultats que nous avons obtenus.

Nous rappelons dans un premier temps le modèle de menaces considéré pour *TrustSoC*.

5.3.1 Modèle de menaces considéré

TrustSoC-M est une modélisation de *TrustSoC*. Ainsi, l'architecture modélisée ne considère que des attaques à distance, matérielles et logicielles. Le modèle de menaces est centré autour du processus de conception d'un SoC-FPGA en particulier l'usage de blocs matériels (IP) ou d'applications logicielles développées par un tiers. Ces éléments peuvent être malicieux ou vulnérables et peuvent réaliser des attaques contre le système. Les attaques peuvent être passives ou actives. Elles peuvent avoir de lourdes conséquences pour le système : des accès illégitimes, des modifications du contenu mémoire, des modifications du contenu des communications ou prendre le contrôle du système de gestion de la puissance du SoC [14].

L'architecture considère aussi les attaques de type SCA basées sur la mesure du temps d'accès à la mémoire cache. Ces attaques peuvent être effectuées par une IP matérielle malicieuse intégrée dans le FPGA qui cible une application de confiance [17]. L'IP malicieuse peut accéder et modifier le contenu des régions sécurisées de la mémoire cache.

L'outil de CAO et le compilateur logiciel sont considérés comme fiables et ne peuvent pas être utilisés pour réaliser des modifications malicieuses de l'architecture générée. De même, les composants qui sont ajoutés pour la sécurité sont de confiance et ne peuvent pas être modifiés par l'outil de CAO. Nous supposons également que le SoC et le fondeur sont fiables et que le circuit ne peut pas être modifié au niveau du silicium par exemple avec l'ajout d'un cheval de Troie matériel.

5.3.2 Exigences de sécurité

TrustSoC-M modélise l'architecture *TrustSoC*, il doit donc respecter les exigences de sécurité présentées au Chapitre 3. Ces exigences sont au nombre de cinq.

ES.1 : Règles de sécurité :

Le comportement modélisé par *TrustSoC-M* est étroitement basé sur des règles qui définissent le comportement autorisé du système. Ces dernières sont précisées dans l'Annexe A de ce manuscrit. Elles régissent tous les comportements des applications logicielles et IP matérielles du système. Elles fixent également les règles du découpage et d'accès aux mémoires. Elles définissent l'accès et les modifications à certains périphériques comme le système de gestion de la puissance du SoC. *TrustSoC-M* doit donc modéliser ces règles de sécurité qui permettent de fixer les limites du système et d'empêcher les attaques mentionnées dans le modèle de menaces.

ES.2 : Extension en un système multi-mondes sécurisé :

La proposition *TrustSoC* est basée sur un système multi-mondes présenté dans le Chapitre 3. *TrustSoC-M* doit modéliser les mondes sécurisés (de 1 à N) avec leurs niveaux de privilèges. Ces niveaux peuvent être différents d'un monde à un autre.

ES.3 : Intégration des ressources logiques (type FPGA) dans la sécurité :

L'architecture *TrustSoC* intègre des ressources logiques (FPGA) qui contiennent différentes IP matérielles, associées chacune à un monde. *TrustSoC-M* doit modéliser ces IP matérielles et les règles de sécurité qui encadrent leurs comportements.

ES.4 : Communications sécurisées à l'intérieur du SoC :

La sécurisation du système de communication du SoC est le point central de l'architecture *TrustSoC*. *TrustSoC-M* doit donc modéliser le système proposé. Ce système est composé de contrôleurs de communications qui se basent sur un système d'identifiants matériels uniques et de tables de permissions.

ES.5 : Résistance aux attaques de type SCA basées sur des mesures de temps d'accès de la mémoire cache :

TrustSoC embarque des protections contre les attaques SCA à distance basées sur des mesures de temps d'accès à la mémoire cache du système. *TrustSoC-M* doit modéliser le partitionnement mis en place pour la mémoire cache et le contrôle d'accès effectué aux différentes partitions.

5.3.3 Définition du modèle

TrustSoC-M fournit un modèle de type MBSE. Ce dernier est une représentation du système *TrustSoC*. Ce modèle permet de simplifier la complexité de l'architecture *TrustSoC* en enlevant les contraintes liées à la technologie utilisée pour l'implémentation. Cela nous permet de nous affranchir des limites que nous avons observées dans les deux derniers chapitres. La Figure 5.2 présente la mise en place de *TrustSoC-M* avec les différentes étapes. Celles qui n'ont pas encore été implémentées par manque de temps apparaissent avec des contours en pointillés noirs.

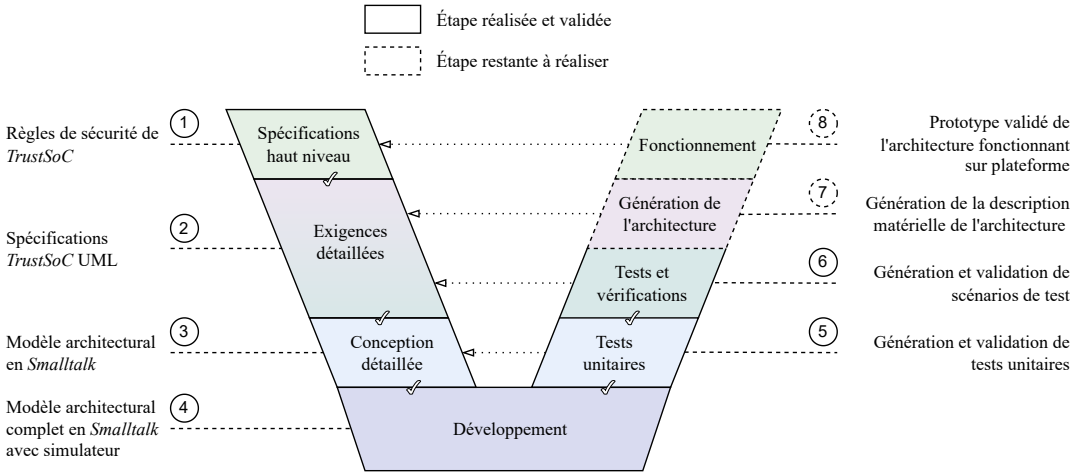


FIGURE 5.2 – Représentation des différentes étapes détaillées de la mise en place de *TrustSoC-M* basée sur une approche MBSE.

Méthode globale de *TrustSoC-M*

Nous présentons tout d'abord de manière globale, les différentes étapes de la méthode *TrustSoC-M*. Puis, nous explicitons certaines étapes qui mettent en place les exigences de sécurité de l'architecture.

Étape n°1 : La première étape de l'approche MBSE de *TrustSoC*, identifiée en vert clair dans la Figure 5.2, consiste fournir les spécifications haut niveau de l'architecture *TrustSoC* au modèle. Il s'agit des règles de sécurité (**ES.1**) qui définissent le fonctionnement autorisé des différents composants du système *TrustSoC*. Nous rappelons que ces règles sont fournies en annexe de cette thèse (voir Annexe A).

Étape n°2 : La deuxième étape consiste à traduire les règles de sécurité et l'architecture en représentation UML (*Unified Modeling Language*). Cette étape est représentée en gradient mauve et vert-bleu dans la Figure 5.2. *TrustSoC-M* construit le modèle à l'aide de blocs de base : bus, processeur, mémoire, contrôleur de communication, IP matérielle, etc. Il spécifie aussi les relations entre chaque bloc.

Étape n°3 : La troisième étape de *TrustSoC-M* vise à traduire les spécifications UML dans un langage de programmation et à obtenir un modèle de l'architecture. Elle est représentée en bleu clair dans la Figure 5.2. Dans *TrustSoC-M*, nous avons choisi le langage *Smalltalk* [37] du fait de ses avantages pour le prototypage rapide. Le langage *Smalltalk* est dynamique. Les objets peuvent être modifiés pendant l'exécution ("à la volée") pour apporter des correctifs. *Smalltalk* permet au programmeur d'inspecter, modifier et tester son code en temps réel pendant l'exécution. Le programmeur peut également ajouter des méthodes toujours à la volée. Cependant, le modèle pourrait très bien être explicité dans un autre langage.

Étape n°4 : La quatrième étape de l'approche MBSE de *TrustSoC*, identifiée en violet, concerne le développement du système complet. Il s'agit du modèle architectural avec un simulateur pour analyser son fonctionnement. Ces derniers sont spécifiés en *Smalltalk*.

Étape n°5 : La cinquième étape de *TrustSoC-M* est une première vérification du comportement du modèle architectural généré à l'aide de tests unitaires. Elle est représentée en bleu clair dans la Figure 5.2.

Étape n°6 : La sixième étape de *TrustSoC-M* est une deuxième étape de vérification qui permet d'effectuer des tests plus complexes. Elle est représentée en vert-bleu dans la Figure 5.2.

Étape n°7 : La septième étape de l'approche MBSE de *TrustSoC*, est la génération de la description matérielle de l'architecture modélisée. Elle est représentée en mauve dans la Figure 5.2. Par manque de temps, cette étape n'a pas été entièrement implémentée. Nos premiers résultats sont présentés dans la sous-Section 5.3.4.

Étape n°8 : La huitième et dernière étape de *TrustSoC-M*, identifiée en vert clair, correspond au fonctionnement de l'architecture modélisée qui a été générée. Cette étape n'a pas été implémentée par manque de temps.

Étapes détaillées et respect des exigences de sécurité

Dans cette sous-sous-section, nous explicitons les étapes n°1, 2, 4 et 6 qui sont particulièrement importantes pour la mise en place du modèle et des exigences de sécurité de l'architecture. Nous mettons en évidence que le modèle *TrustSoC-M* décrit avec précision l'architecture *TrustSoC*.

Étape n°1 :

La première étape de la modélisation par *TrustSoC-M* est entièrement basée sur l'exigence de sécurité n°1 : les règles de sécurité. Le modèle et son comportement sont construits à partir de ces règles de sécurité. Leur mise en place est aisément vérifiable lors de la vérification et des tests par les scénarios.

Étape n°2 :

La deuxième étape importante est de transformer ces spécifications haut niveau (règles de sécurité) en représentation UML. Un diagramme UML permet de construire l'architecture *TrustSoC* à partir de blocs de base. La Figure 5.3 représente le diagramme UML de l'architecture, ses éléments et leurs relations. La partie droite inférieure de la figure, cadre en pointillés verts , représente la partie logicielle de l'architecture avec les tâches et les fonctions. Les tâches communiquent entre elles via des séquences de fonctions qui sont des opérations de lecture/écriture locales ou à distance. Le modèle représente le niveau de sécurité de la transaction avec le paramètre "mode de sécurité" qui fait partie des attributs de la classe "Fonction". *TrustSoC-M* représente le système multi-mondes (**ES.2**) avec la valeur "sécurisé étendu de 1 à N". Il existe aussi un mode de sécurité non sécurisé/sécurisé qui permet de modéliser un système exploitant la technologie ARM *TrustZone*, c'est-à-dire un système à deux mondes.

La partie gauche inférieure de la Figure 5.3, cadre en pointillés bleus, représente le matériel associé à l'architecture. Les blocs matériels représentés sont : le bus de communication, les processeurs et les IP matérielles, les mémoires, les contrôleurs de communication (*Wrapper*) et leurs droits. Aucun de ces blocs matériels (processeur, IP matérielle ou mémoire) n'est connecté directement au bus de communication du système. Un contrôleur de communication est inséré entre ces blocs et le bus. Chaque bloc du système est isolé du bus de communication. Les contrôleurs de communication intègrent également le système d'identifiants matériels uniques. Ces identifiants sont transmis à chaque requête. Les contrôleurs de communication disposent de droits qui sont associés à une IP matérielle. Ceux-ci représentent les tables de permissions. Pour chaque plage d'adresses spécifique, il lui est associé l'identifiant de l'entité qui a l'autorisation de communiquer avec le composant ainsi que le mode (lecture/écriture) et le mode de sécurité (non sécurisé, sécurisé, ou sécurisé étendu de $1 \dots N$). Les

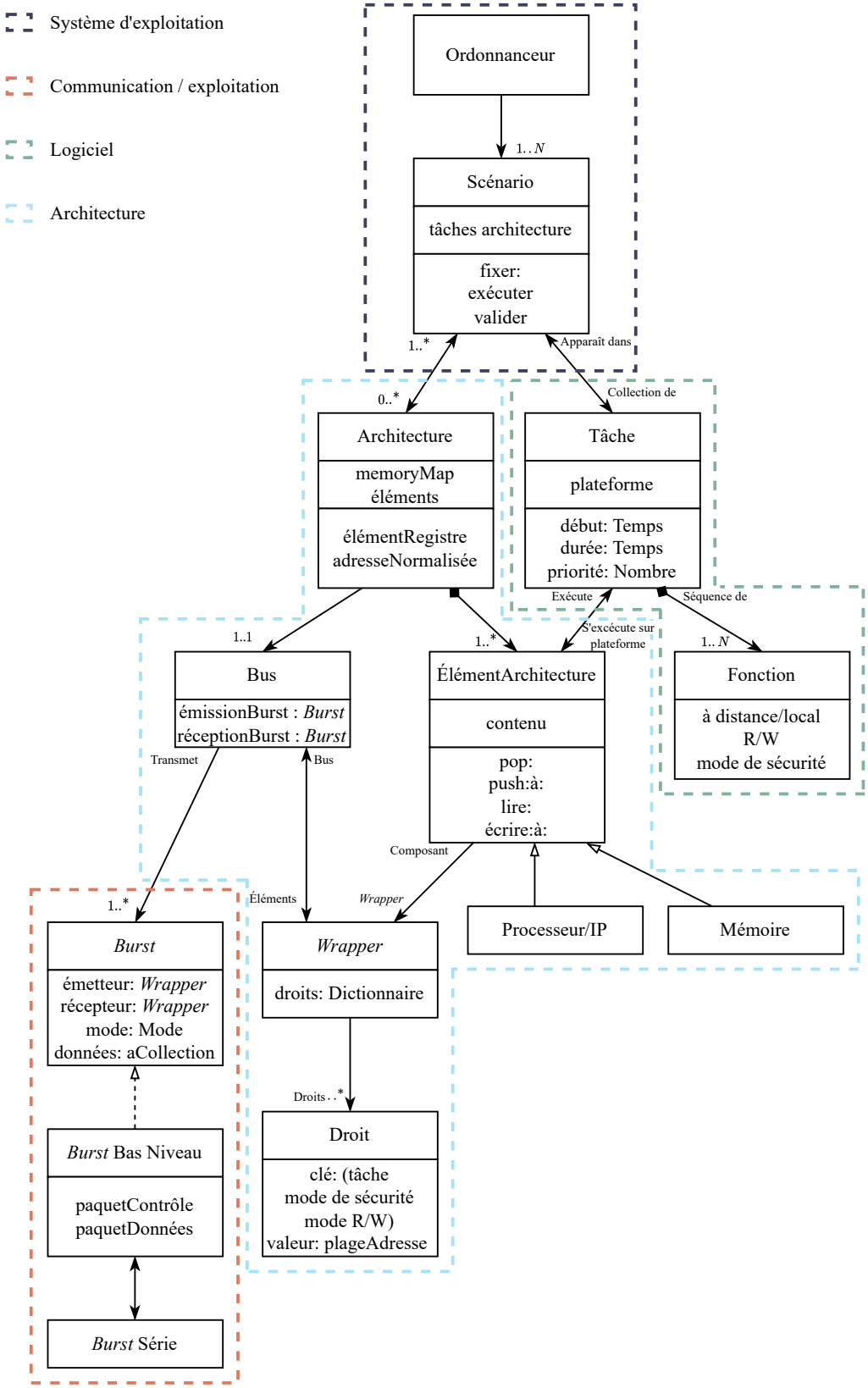


FIGURE 5.3 – Diagramme de classes en langage UML représentant le modèle exécutable *TrustSoC*.

droits ne sont vérifiés qu'à la réception d'une communication (interface esclave) et

que si tous les paramètres de la requête concordent avec ceux du dictionnaire. Si un identifiant ne figure pas dans le dictionnaire, cela signifie automatiquement qu'il n'a pas les permissions et la requête est rejetée. Le modèle respecte donc bien les exigences de sécurité n°3 et n°4 (intégration des ressources logiques dans la sécurité et communications sécurisées à l'intérieur du SoC).

La représentation UML (Figure 5.3) comprend aussi la définition du comportement du bus (cadre en pointillés orange). Cependant, ce dernier est explicité de manière plus simplifiée. Le modèle permet d'abstraire les détails de la réalisation technique du protocole de communication. Il se concentre sur le fonctionnement, c'est-à-dire l'envoi et la réception de données. Ces envois et réceptions peuvent, par exemple, être réalisés par communication de type *burst*. Il s'agit d'un envoi de plusieurs paquets de données à la suite pour une seule et même transaction.

Étape n°4 :

TrustSoC-M dispose d'un simulateur à événements discrets. Ce dernier utilise une liste d'événements ordonnés par date pour représenter le fonctionnement du système. Il permet d'évaluer le modèle architectural généré. Nous illustrons le fonctionnement de ce simulateur dans la Figure 5.4.

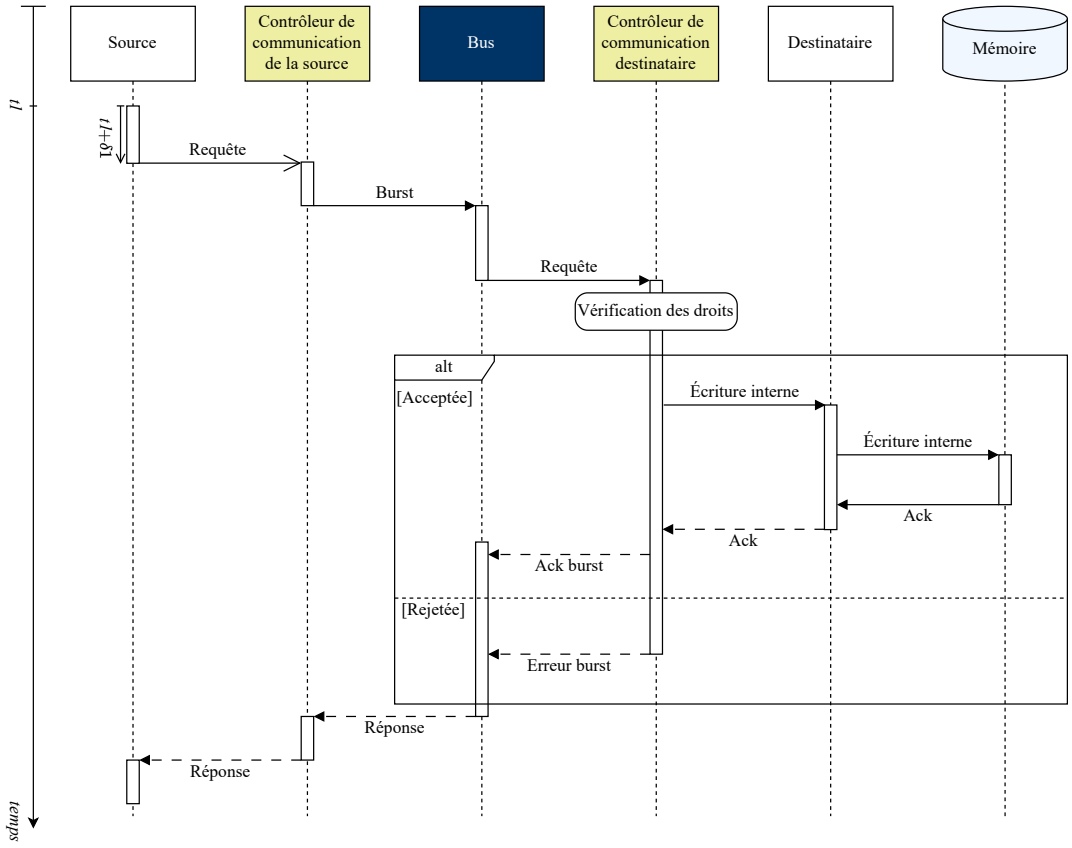


FIGURE 5.4 – Diagramme de séquence pour une opération d'écriture.

Il s'agit du déroulement d'une opération d'écriture de *TrustSoC-M*. Dans la Figure 5.4, la partie relative aux contrôleurs de communication est identifiée en vert-jaune. Celle liée au bus de communication est illustrée en bleu foncé et celle relative à la mémoire en bleu clair. L'échelle de temps du déroulement de l'opération est représentée sur la gauche par une flèche noire continue. Un évènement événement survient

à un instant t connu et change l'état du système. Dans la Figure 5.4, nous prenons l'exemple de l'instant *t1* qui correspond à l'émission d'une requête par la source. Aucune modification ne peut avoir lieu entre deux événements consécutifs. *TrustSoC-M* considère que les tâches commencent à l'instant t et se poursuivent pendant une certaine durée δ. Toujours en prenant l'exemple de l'émission de la requête, la Figure 5.4 représente cette durée $t1 + \delta 1$. Les délais d'exécution des tâches sont représentés par les rectangles verticaux blancs. Ces derniers sont proportionnels à cet intervalle de temps $(t + \delta)$. Une nouvelle tâche peut être démarrée dès que les tâches précédentes se sont terminées. La simulation s'arrête lorsque toutes les tâches se sont terminées et qu'aucune nouvelle tâche ne doit être exécutée. Les tâches et leurs enchaînements pour la simulation peuvent être modifiés. Il est possible d'ajouter des dépendances pour la simulation, par exemple, pour réaliser un comportement séquentiel. Il est également possible de modifier la durée δ d'une tâche. Il est également possible de définir des instants critiques, qui peuvent s'ils sont atteints arrêter la simulation. Une violation des droits pourrait par exemple constituer un tel instant.

Le fonctionnement des opérations, Figure 5.4, est identique à celui présenté dans le Chapitre 3 pour l'architecture *TrustSoC*. Quand une transaction est reçue par un contrôleur, comme pour *TrustSoC*, deux issues sont possibles. Première possibilité, les droits sont vérifiés et la requête est acceptée et transmise au destinataire. Deuxième possibilité, la requête est rejetée, car la source n'a pas l'autorisation de communiquer avec le destinataire. Dans ce cas, une erreur est transmise en réponse.

Étape n°6 :

Dans *TrustSoC-M*, le modèle permet de générer des scénarios pour vérifier le fonctionnement de l'architecture. Ces scénarios sont générés de manière aléatoire : tâches, composant visé par les tâches, événements (lecture/écriture), etc. Le modèle permet également de faire de l'exploration pour essayer de détecter si une règle est compromise. Le simulateur permet de tester les scénarios qui ont été générés. Ils peuvent être exécutés plusieurs fois afin de vérifier l'absence de comportement indésirable.

5.3.4 Mise en place de *TrustSoC-M*

Dans cette section, nous détaillons la mise en place du modèle et du simulateur en langage *Smalltalk* (étapes n°3 et 4) ainsi que les résultats obtenus par *TrustSoC-M*. Un exemple d'architecture est fourni pour montrer sa construction. Nous donnons un exemple de scénario (étape n°8) et ses résultats.

Instance d'une architecture

Le listing 5.1 définit un exemple d'architecture qui est une instance du modèle.

```
1  basicExample: aClass
2
3  | architecture proc ip1 ip2 bus mem |
4  architecture := TSArchitecture new.
5
6  bus := aClass name: 'AXI' architecture: architecture.
7  proc := TSProcessor name:'Processor' architecture: architecture.
8  ip1 := TSIP name:'IP1' architecture: architecture.
9  ip2 := TSIP name:'IP2' architecture: architecture.
10 mem := TSMemory name:'RAM' architecture: architecture. mem size: 10000.
11 mem splitAt:5000.
12
```

```
13 "Generation des controleurs de communication"
14 proc plugTo: bus mode:#master.
15 ip1 plugTo:bus mode:#master.
16 ip2 plugTo: bus.
17 memory plugTo:bus.
18
19 ↑ architecture
```

Listing 5.1 – Code en *Smalltalk* pour une instance du modèle MBSE de *TrustSoC*

Elle est composée d'un bus nommé *AXI*, d'un processeur, de deux IP matérielles (IP1 et IP2) et d'une mémoire RAM (lignes 6 à 11 Listing 5.1). La mémoire RAM contient deux partitions (instruction *splitAt* à la ligne 11). Le processeur et l'IP matérielle 1 ont des interfaces de communication maîtres (lignes 14 et 15 Listing 5.1) tandis que la mémoire et l'IP matérielle 2 ont des interfaces esclaves (lignes 16 et 17). Cette architecture est très simple. Elle nous permet de tester différents scénarios et de vérifier que le modèle est fidèle à l'architecture *TrustSoC* présentée dans le Chapitre 3. Une représentation matérielle de l'architecture est illustrée dans la Figure 5.5.

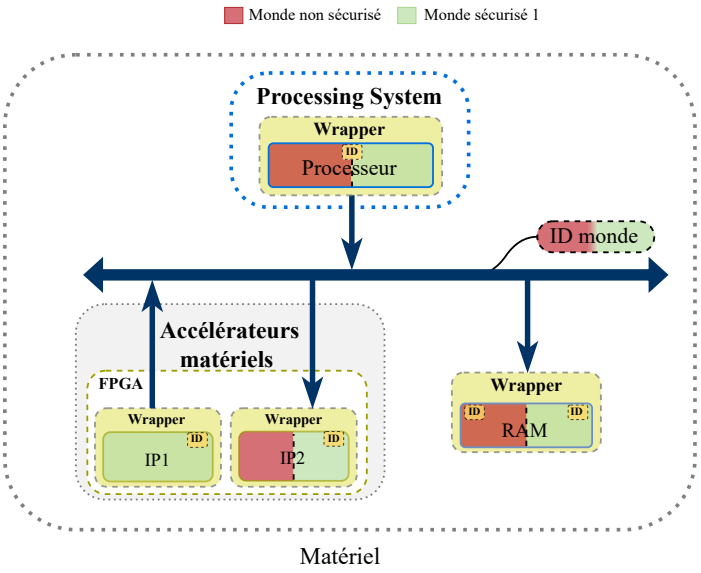


Figure 5.5 – Exemple d'architecture d'une instance du modèle.

Elle comprend les mêmes éléments que l'instance du modèle. Le Listing 5.2 illustre l'assignation des identifiants aux composants. Elle donne également la spécification des droits à la mémoire pour les différentes entités du système.

```
1  exampleWithGrants: aClass
2
3  | architecture bus ip1 memWrapper cpu |
4  architecture := self basicExample: aClass.
5
6  bus := architecture elementWithID:1.
7  cpu := architecture elementWithID:2.
8  ip1 := architecture elementWithID:3.
9  ip2 := architecture elementWithID:4.
10 memWrapper := bus elementWithID:5.
11
12 memWrapper grant: ip1 id mode: #read security: 1
13 address: (TSAddressRange start:0 stop:5000).
```

```
14 memWrapper grant: cpu id mode: #read security: 1
15 address: (TSAddressRange start:0 stop:5000).
16 memWrapper grant: cpu id mode: #write security: 1
17 address: (TSAddressRange start:0 stop:5000).
18
19 ↑ architecture
```

Listing 5.2 – Code en *Smalltalk* définissant les identifiants et droits de l'architecture "basicExample"

Les lignes 6 à 9 définissent les identifiants aux différents composants (1 pour le bus, 2 pour le CPU, 3 pour l'IP matérielle n°1, 4 pour l'IP matérielle n°2 et 5 pour la mémoire). Les lignes 12 à 17 donnent les permissions à la mémoire RAM de l'architecture. Nous donnons une représentation graphique de cette table de permissions dans la Figure 5.7. L'IP matérielle n°2 ne dispose pas d'interface de communication de type maître donc elle n'est pas représentée dans la table de permissions. Nous le rappelons, tout composant qui ne dispose pas d'une entrée dans une table de permissions n'a aucun droit de communication avec le composant. Les composants (IP matérielle n°1 et CPU) appartiennent au monde non sécurisé n°1, identifié avec le mode de sécurité 1 en vert clair. L'IP matérielle n°1 ne peut communiquer avec la mémoire qu'en mode lecture. Le CPU peut lui communiquer en écriture et en lecture avec la mémoire. Ces deux composants, IP matérielle n°1 et CPU, n'ont accès qu'à la partition mémoire de l'adresse 0 à 5000.

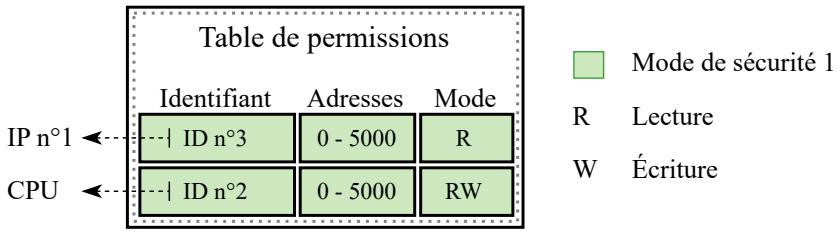


Figure 5.6 – Table permission de la mémoire RAM d'une instance du modèle donnée dans la Figure 5.5.

Exemple de scénario appliqué à l'architecture modélisée par *TrustSoC-M*

Après avoir créé l'architecture, fixé les identifiants et les droits, nous pouvons exécuter des scénarios et observer des résultats. Ces tests correspondent aux étapes 5 et 6 de la Figure 5.2.

Un exemple de scénario est donné dans le Listing 5.3. Les différentes étapes de ce scénario sont représentées dans la Figure 5.7. Les résultats des différentes actions sont ajoutés dans un fichier log présenté dans la Figure 5.8.

```
1  exampleUsageForAXIBusWithGrants
2
3  | architecture bus ip1 cpu mem data read processedData |
4  architecture := self exampleWithGrants: TSAXIBus.
5  Transcript clear.
6
7  "Initialisation de toutes les cases d'un tableau"
8  data := (Array new:2000 withAll:1).
9
10 ip1 write:data at: mem at: 1000.
11 cpu write:data to:mem at: 1000.
```

```
12 read := ip1 read: (TSAddresseRange from: 1500 to: 1600) from: mem.
13
14 processedData := (read collect:[:a| a *2]).
15
16 cpu write: processedData to: mem at: 3500.
17
18 ↑ architecture
```

Listing 5.3 – Code en *Smalltalk* pour un scénario de test de l'architecture

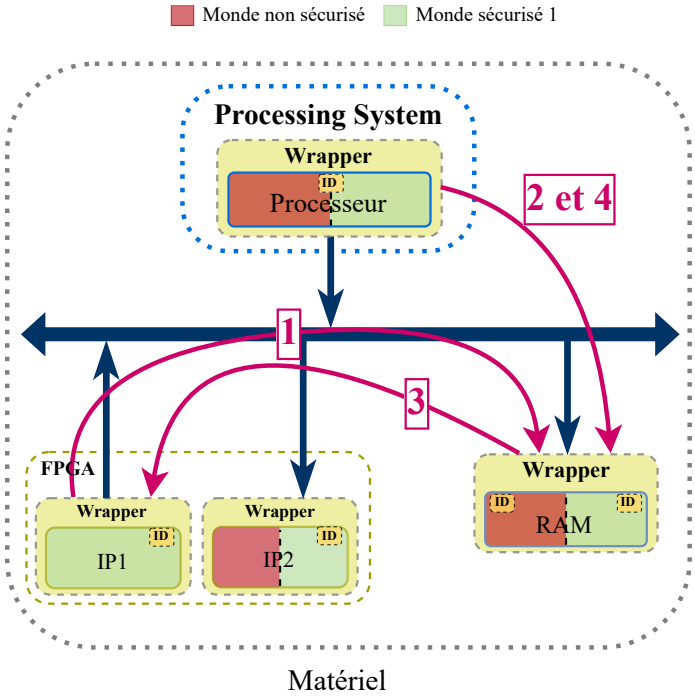


Figure 5.7 – Représentation des étapes du scénario (donné dans le Listing 5.3) notées de 1 à 4 sur l'architecture matérielle.

Ce scénario teste les droits du CPU et de l'IP matérielle n°1 sur la mémoire RAM. La première étape du scénario est donnée à la ligne 9. L'IP matérielle n°1 tente de réaliser une écriture à l'adresse 1000 de la mémoire. Dans la définition des droits (Listing 5.2 et Figure 5.7), l'IP matérielle n°1 ne peut pas effectuer d'opération d'écriture en mémoire. Ainsi, comme l'illustre la Figure 5.8 en bordeaux, la requête est rejetée car 3 (IP matérielle n°1) n'a pas les droits sur 5 (mémoire).

L'étape n°2 correspond au processeur qui écrit le contenu du tableau *data*, déclaré à la ligne 7, dans la mémoire à partir de l'adresse 1000. Le processeur a les droits d'écriture sur la mémoire : plage d'adresses, opération et niveau de sécurité. L'opération est donc effectuée. Elle apparaît en vert foncé dans la Figure 5.8. Le type d'opération et les adresses sont ajoutés dans le fichier log. L'étape suivante, la n°3, est une lecture effectuée par l'IP matérielle n°1 sur la mémoire de l'adresse 1500 à 1600. Cette fois-ci, l'IP matérielle n°1 a bien les droits pour communiquer avec la mémoire donc l'opération est acceptée. Elle apparaît en vert foncé dans la Figure 5.8. Le contenu de cette opération est modifié comme indiqué à la ligne 13 dans le Listing 5.3. Les données sont multipliées par deux. Une dernière opération est réalisée par le CPU. Il écrit en mémoire les données qui ont subi le traitement à partir de l'adresse 3500. Le

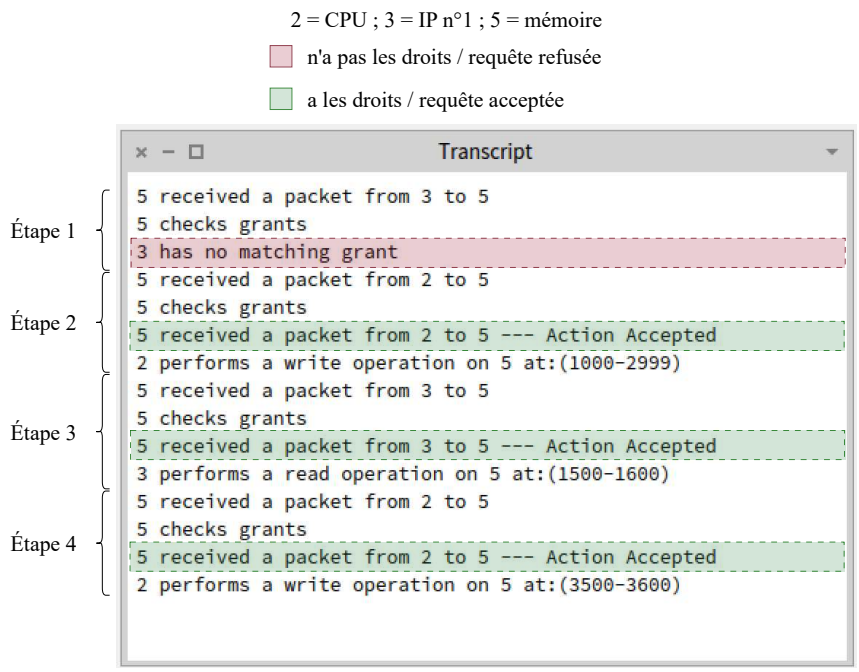


FIGURE 5.8 – Résultats du scénario présenté dans le Listing 5.3.

processeur ayant les droits (plage d'adresses, opération et niveau de sécurité), l'opération est effectuée et ajoutée dans le fichier log. Elle apparaît donc en vert foncé dans la Figure 5.8. Le fichier log permet de vérifier le comportement de l'architecture après l'exécution d'un scénario. Grâce à ce fichier et la génération automatique de scénarios, nous pouvons vérifier que toutes les règles de sécurité de l'architecture sont respectées.

Mise en œuvre de la génération de l'architecture par *TrustSoC-M*

La génération du code de description matérielle de l'architecture modélisée par *TrustSoC-M* est un travail encore en cours. Cette étape est identifiée avec le numéro 7 dans la Figure 5.2. La génération de l'architecture *TrustSoC* se fait au travers d'un outil de CAO appelé *DiWall*. Cet outil est considéré de confiance et ne peut pas effectuer des modifications malicieuses. Il est utilisé pour générer les contrôleurs de communication reliés aux IP matérielles ainsi que leurs tables de permissions. La structure des composants générés par *DiWall* est présentée dans la Figure 5.9.

Dans cet exemple, un processeur (identifié dans un bloc avec des contours bleus) communique avec une IP matérielle (identifiée dans un bloc avec des contours vert-jaune). Un contrôleur de communication est placé entre le bus de communication (AXI) et l'IP matérielle. Il est identifié en vert-jaune. Le contrôleur est composé de trois blocs principaux. Un module appelé "contrôleur de règles" permet d'effectuer la vérification de la requête entrante (identifiée en bleu foncé) avec les politiques de sécurité. Celles-ci sont stockées dans une mémoire appelée "mémoire règles". Pour chaque adresse est associé un composant avec ses droits : opération ("Op" pour lecture seule, écriture seule ou lecture/écriture), identifiant matériel ("ID") et identifiant monde ("Mode"). Selon le résultat de la vérification des droits (identifié par la flèche gris foncé), le composant "lock/enable" bloque ou laisse passer la communication à l'IP matérielle. Si les droits sont valides, la requête est transmise à l'IP matérielle. La réponse de celle-ci, une fois le traitement effectué, est identifiée par les flèches orange. Cette communication est

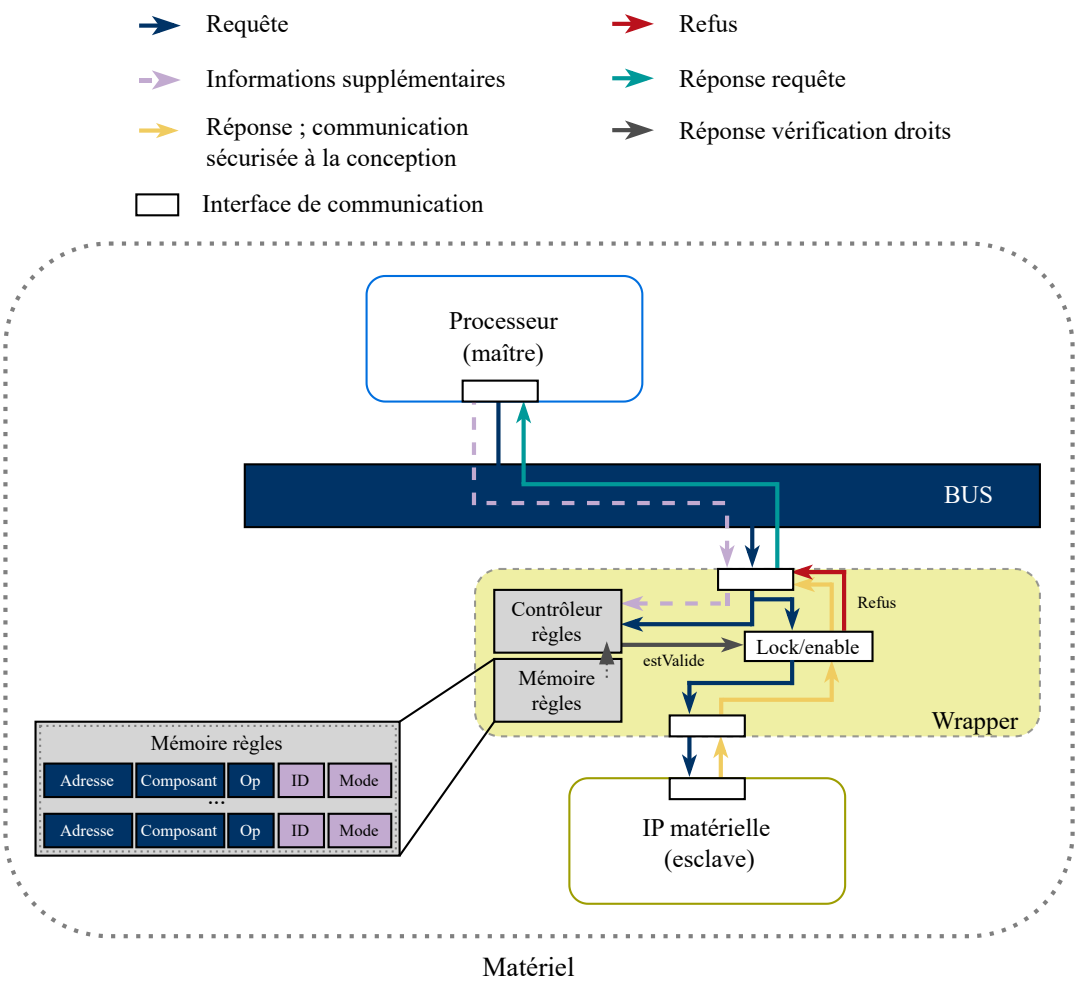


FIGURE 5.9 – Aperçu de la solution de génération proposée par l'outil *DiWall*.

considérée comme sécurisée, car elle est entièrement gérée par le contrôleur de communication. Si, au contraire, les droits ne sont pas vérifiés, la communication n'est pas transmise. Le refus est illustré à l'aide de la flèche continue rouge. La réponse de la requête est identifiée par la flèche turquoise dans la Figure 5.9. L'architecture qui est générée est donc bien conforme aux spécifications du modèle et à *TrustSoC*.

DiWall est développé en Python. Il peut être invoqué soit par un script, soit de manière graphique. La Figure 5.10 présente cette interface utilisateur avec un exemple d'architecture. Dans cette architecture, on trouve : deux processeurs, trois mémoires, deux IP matérielles ainsi que le bus de communication du système et les contrôleurs. Chaque module a une couleur propre en fonction de sa nature : bleu pour les processeurs, gris pour les mémoires, vert pour les IP matérielles et orange pour les contrôleurs de communication. Le bus de communication est toujours identifié en bleu foncé. Les fichiers de description matérielle nécessaires à l'ajout d'un contrôleur sont automatiquement générés : la mémoire des règles de sécurité ("rules_array.vhd"), le contrôleur des règles ("wrapper.vhd") et l'interface AXI ("interface_AXI.vhd"). Un exemple de code généré pour l'IP n°1 est représenté dans les cadres blancs. Il s'agit de la description matérielle du contrôleur de communication et de sa table de permissions. Les tailles de la mémoire sont modifiables et correspondent aux besoins de l'architecture.

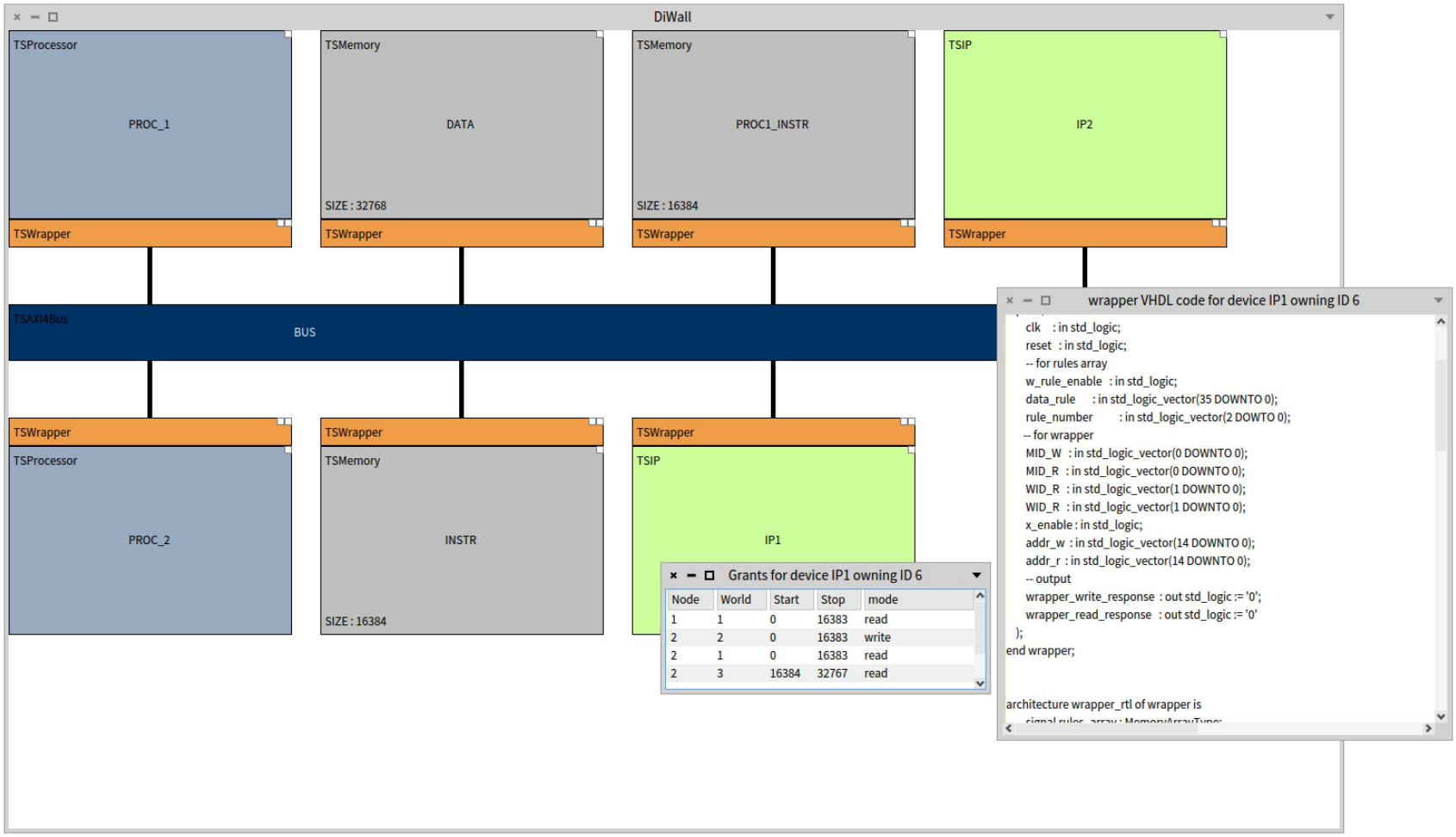


FIGURE 5.10 – Interface graphique utilisateur de l'outil *DiWall* avec un exemple d'architecture de SoC hétérogène.

Pour générer un contrôleur de communication, *DiWall* a besoin de quatre paramètres principaux :

- *MEM_DEPTH* : le nombre maximum de règles supportées.
- *MASTER_NUMBER* : le nombre d'IP matérielles disposant d'une interface de communication maître.
- *WORLD_NUMBER* : le nombre de mondes dans le système.
- *ADDRESS_SIZE* : le nombre de bits nécessaires pour encoder les adresses.

DiWall est ensuite capable de calculer des paramètres secondaires pour générer les trois fichiers ("rules_array.vhd", "wrapper.vhd" et "interface_AXI.vhd") d'un contrôleur de communication. Il génère également à partir d'un scénario, un fichier de simulation (*testbench*). Celui-ci contient les résultats attendus de chaque opération (refus/acceptation) qui peuvent ensuite être comparés aux résultats de la simulation de l'architecture.

Quelques fonctionnalités sont encore nécessaires pour que cette étape de génération de l'architecture (étape n°7 Figure 5.2) soit complète. Notamment, la reconfiguration des tables de permissions de l'architecture, comme présenté dans le Chapitre 3, n'est pas encore disponible.

5.4 Perspective pour *TrustSoC-M*

Dans la section précédente, nous avons présenté la modélisation de *TrustSoC-M*. Elle est basée sur une méthode d'ingénierie des systèmes basée sur des modèles. Cette approche est intéressante, car elle nous permet d'abstraire la couche liée à la technologie utilisée lors de l'implémentation matérielle. Elle nous permet de nous concentrer sur le fonctionnement de l'architecture et d'explorer l'espace de conception. Dans *TrustSoC-M*, nous pouvons facilement changer l'architecture et l'évaluer avec des scénarios. Pour aller plus loin, nous aimerions que *TrustSoC-M* soit capable d'analyser une architecture de SoC-FPGA existante, d'en identifier les différentes IP matérielles et de la modéliser. *TrustSoC-M* pourrait alors appliquer *TrustSoC* sur celle-ci et la générer.

5.5 Conclusion du chapitre

Ce chapitre a présenté une approche de type MBSE (*Model-Based System Engineering*) pour *TrustSoC* appelée *TrustSoC-M*. Il a permis de présenter les motivations de *TrustSoC-M*, la définition et le fonctionnement du modèle. Modéliser *TrustSoC* de cette manière permet d'abstraire la couche technologique de l'implémentation matérielle. Il permet ainsi de répondre aux limitations identifiées lors des Chapitres 3 et 4. La modélisation nous permet de restructurer l'architecture *TrustSoC* plus rapidement et plus facilement que dans l'implémentation matérielle. Elle nous permet également de réaliser et d'exécuter plus rapidement des scénarios sur l'architecture. Ces caractéristiques facilitent l'exploration de l'espace de conception. Ce chapitre a présenté les résultats que nous avons obtenus avec des extraits de code du modèle et un fichier log d'un scénario exécuté. Il a également permis de présenter l'outil *DiWall* qui permet de générer l'architecture *TrustSoC* à partir des spécifications du modèle. Enfin, il a proposé une perspective pour le modèle avec l'analyse et la génération d'une architecture existante avec une application de *TrustSoC*.

Chapitre 6

Conclusion et perspectives

Résumé

Dans cette thèse, nous nous sommes intéressés aux architectures de SoC hétérogènes sécurisées dès la conception. Ce chapitre propose un résumé des contributions de cette thèse et discute des perspectives envisagées pour celle-ci.

Table des matières

6.1 Conclusion 117
6.2 Résumé des contributions 118
6.3 Perspectives 119
6.3.1 *TrustSoC* et *RTrustSoC* 119
6.3.2 *TrustSoC-V* 119
6.3.3 *TrustSoC-M* 120

6.1 Conclusion

L'intérêt pour la sécurité des architectures de SoC hétérogènes s'accroît depuis quelques années. L'architecture HECTOR-V [64] a été proposée en 2021, tout comme CURE [10] et *WorldGuard* [79]. Ces propositions tentent d'inverser la tendance actuelle de conception des circuits intégrés. Celle-ci est centrée sur l'amélioration des performances du système au détriment de sa sécurité. Les SoC hétérogènes disponibles sur le marché, même avec la mise en œuvre de solutions telles que la technologie ARM *TrustZone*, contiennent des vulnérabilités. Celles-ci sont introduites involontairement durant la conception de l'architecture et les exposent à des attaques. Bien que ce domaine de recherche soit très actif et tente de trouver une solution face à ces vulnérabilités, aucune approche ne considère réellement pour sa sécurité la dualité (logicielle / matérielle) de l'architecture du SoC hétérogène. Cette sécurité est également très binaire avec une approche sécurisée ou non sécurisée qui limite les possibilités des concepteurs. Les différentes architectures proposées dans ce manuscrit visent à répondre à ces limites.

6.2 Résumé des contributions

La proposition d'une architecture de SoC hétérogène sécurisée doit respecter certains critères. Elle doit être fournie avec un modèle de menaces précis et détaillé, des exigences de sécurité ainsi que des règles qui définissent les comportements autorisés. Ces règles de sécurité, présentées dans l'Annexe A, font partie de la première contribution de cette thèse. Elles ont été présentées dans le Chapitre 3 de ce manuscrit avec *TrustSoC* qui est la première contribution de cette thèse. Elle permet de proposer une première réponse face aux limites identifiées dans l'état de l'art. Cette architecture a introduit le concept de mondes multiples sécurisés basé sur la technologie ARM *TrustZone*. Cette extension permet de proposer une sécurité à plusieurs niveaux (jusqu'à N), contrairement aux approches précédentes, qui étaient binaires. Elle offre plus de flexibilité et de sécurité aux concepteurs. *TrustSoC* propose également une première version de la notion d'IP de confiance, qui n'a jusqu'alors jamais été présentée. Elle vise à étendre le concept d'applications sécurisées et de leurs environnements d'exécution de confiance à la logique programmable (FPGA) du SoC-FPGA. Ainsi, les ressources matérielles sont entièrement intégrées dans la sécurité du système. Nous avons présenté ensuite notre deuxième contribution, *RTrustSoC*, qui permet d'étendre le modèle de menaces de *TrustSoC* et la sécurité apportée à l'architecture du SoC hétérogène. *RTrustSoC* introduit un système de pénalités dynamiques inspiré des architectures zéro confiance (*Zero Trust Architecture*). Elle permet ainsi d'étendre le concept d'IP de confiance. Nous avons fourni des résultats d'implémentation dont trois scénarios d'utilisation issus des articles [17] et [40] appliqués à *RTrustSoC*. Ces résultats ont démontré l'intérêt et les avantages de notre proposition en matière de sécurité ainsi que son impact sur les ressources du système. Ils ont également permis de mettre en évidence les limites liées à l'utilisation de technologies propriétaires. Celles-ci ont motivé la proposition suivante, *TrustSoC-V*, présentée au Chapitre 4.

La troisième contribution, intitulée *TrustSoC-V*, est la suite logique des deux premières. Elle répond aux limites identifiées dans les contributions précédentes en proposant une architecture sécurisée à la conception de SoC hétérogène open source. *TrustSoC-V* utilise des processeurs libres de droits (cœurs RISC-V). Elle est également basée sur un modèle de menaces précis et des exigences de sécurité. L'architecture, *TrustSoC-V*, introduit un nouveau bus de communication open source et sécurisé à la conception. Celui-ci est nommé *Xsecure*. Ce nouveau bus permet d'étendre la sécurité conférée aux communications au sein du SoC-FPGA. *TrustSoC-V* repose également sur le même partitionnement en mondes multiples sécurisés. Elle intègre également la notion d'IP matérielle de confiance. Pour faciliter la mise en œuvre de *TrustSoC-V* à un concepteur, la méthodologie RISC-B a été proposée. Il s'agit de notre quatrième contribution qui définit une nouvelle méthode pour concevoir les architectures embarquant des processeurs RISC-V. La méthodologie RISC-B est inspirée de la conception sur FPGA. L'architecture est considérée bloc par bloc et chaque bloc est spécifié indépendamment des autres. Ils sont ensuite interconnectés de manière graphique. Cette manière d'implémenter le système permet de simplifier grandement le processus d'intégration. RISC-B a été fournie avec des résultats d'implémentation et un prototype fonctionnel pour démontrer son efficacité. *TrustSoC-V* et la méthodologie RISC-B ont cependant mis en exergue des limites et montré l'intérêt d'avoir une modélisation de l'architecture.

La dernière contribution, nommée *TrustSoC-M*, s'intéresse à la modélisation de la conception de l'architecture *TrustSoC* afin d'abstraire la partie liée au matériel. Elle

est basée sur une approche de type MBSE (*Model-Based System Engineering*). *TrustSoC-M* permet, en abstrayant la couche technologique, de proposer une solution face aux limitations identifiées lors des propositions de *TrustSoC*, *RTrustSoC* et *TrustSoC-V*. *TrustSoC-M* permet de compléter ces propositions. La modélisation nous permet de restructurer l'architecture *TrustSoC* plus rapidement et plus facilement que dans l'implémentation matérielle. Elle nous permet d'explorer l'espace de conception avec, notamment la réalisation et l'exécution de scénarios générés automatiquement sur l'architecture. *TrustSoC-M* permet, avec les propositions précédentes (*TrustSoC*, *RTrustSoC* et *TrustSoC-V*), de fournir une solution complète quant à la problématique de cette thèse : la proposition d'une architecture de SoC hétérogène sécurisée à la conception.

6.3 Perspectives

Bien que le domaine de recherche liée à la sécurité des architectures de SoC hétérogène soit très actif, les propositions actuelles et l'état de l'art peuvent encore être enrichis.

6.3.1 *TrustSoC* et *RTrustSoC*

La première piste est en lien avec nos propositions : *TrustSoC* et *RTrustSoC*. Il s'agirait de réaliser le code logiciel lié au système des mondes multiples sécurisés. Cela permettrait de fournir un démonstrateur fonctionnel complet de notre proposition. Les résultats permettraient d'évaluer l'impact de notre proposition au niveau matériel mais aussi logiciel. Ils fourniraient ainsi les surcoûts totaux de notre proposition. Il serait également alors possible d'étudier l'application d'autres scénarios d'attaque sur l'architecture.

La deuxième possibilité de recherche est toujours liée aux architectures *TrustSoC* et *RTrustSoC*. Celles-ci sont limitées en modifications par l'utilisation de technologies propriétaires telles que le protocole AMBA-AXI. Le système de communication est particulièrement vulnérable aux attaques qui peuvent entraîner de lourdes conséquences. Les signaux de sécurité, liés aux mondes et aux identifiants, doivent être protégés. Puisque la structure du bus n'est pas modifiable, il serait intéressant d'étudier la possibilité de mettre en place un chiffrement léger. Il faudrait rechercher les candidats possibles et étudier leur impact sur l'architecture en termes de ressources utilisées supplémentaires, mais également pour les aspects temporels (fréquence maximale de fonctionnement et latence).

6.3.2 *TrustSoC-V*

Il serait intéressant de poursuivre les recherches sur le bus *XSecure* et d'obtenir un prototype fonctionnel. Ce bus *XSecure* permettrait de sécuriser les signaux de sécurité et plus globalement toute la communication en empêchant les attaques présentées dans l'introduction de cette thèse. Il permettrait de fournir un système de communication qui ne dépendrait plus de technologie propriétaire comme le protocole AMBA-*AXI*. Le prototype fonctionnel évaluerait le surcoût ainsi que le gain apporté par *XSecure* pour la sécurité du système par rapport aux attaques présentées dans l'introduction de cette thèse.

6.3.3 *TrustSoC-M*

Une perspective pour *TrustSoC-M* a été présentée dans le Chapitre 5 de cette thèse. Il s'agit de la génération de code de description matérielle d'une architecture existante à partir du modèle. *TrustSoC-M* analyserait l'architecture pour en extraire les IP matérielles. Celles-ci pourraient alors être générées avec les contrôleurs de communication et leurs tables de permissions. Grâce à *TrustSoC-M*, une personne ne disposant pas des connaissances nécessaires pourrait générer son architecture de SoC hétérogène sécurisée à la conception.

Nous pourrions envisager une perspective de *TrustSoC* dans le domaine du *Cloud* où la question de la sécurité est devenue très importante [33][84]. Des instances de FPGA peuvent être louées par un utilisateur pour exécuter ses conceptions. Cependant, afin de rentabiliser les plateformes, les fournisseurs partagent les circuits entre plusieurs utilisateurs. Les solutions de sécurité fournies ne sont pas suffisantes et il est possible pour un utilisateur malveillant d'observer le comportement des autres conceptions exécutées sur le circuit. Grâce à *TrustSoC-M*, nous pourrions modéliser une architecture avec le système multi-mondes sécurisés dans laquelle se trouveraient les conceptions des utilisateurs. Chacune serait assignée à un niveau de sécurité défini et aurait une table de permissions associée. L'architecture pourrait alors être générée à l'aide d'*overlays* [16][63][72]. Un *overlay* est une architecture reconfigurable implémentée sur FPGA. Elle ajoute un degré d'abstraction entre la couche utilisateur avec l'application et la couche physique avec le FPGA. Ces *overlays* seraient ensuite appliqués sur les différents SoC-FPGA du fournisseur. Les conceptions des utilisateurs seraient alors protégées des attaques visant le vol de propriété intellectuelle.

Chapitre 7

Communications et publications

7.1 Publication dans un journal international

Raphaële Milan, Lilian Bossuet, Loïc Lagadec, Carlos Andres Lara-Nino, Brice Colombier and Théotime Bollengier. **"Efficient Adaptive Multi-level Privilege Partitioning with RTrustSoC"**. in *IEEE Transactions on Circuits and Systems I : Regular Papers*, doi : 10.1109/TCSI.2024.3413364 [59].

7.2 Conférence internationale avec comité de lecture

Raphaële Milan, Loïc Lagadec, Théotime Bollengier, Lilian Bossuet and Ciprian Theodorov. **"Secured-by-design systems-on-chip : a MBSE Approach"**. *34th International Workshop on Rapid System Prototyping (RSP 2023)*, Hamburg, Germany, September 2023 [57].

Raphaële Milan, Lilian Bossuet, Loïc Lagadec, Carlos Andres Lara-Nino, Brice Colombier. **"TrustSoC : Light and Efficient Heterogeneous SoC Architecture, Secure-by-design"**. *2023 Asian Hardware Oriented Security and Trust Symposium (AsianHOST 2023)*, Tianjin, China, December 2023 [61].

En soumission : Carlos Andres Lara-Nino, Raphaële Milan. **"RISC-B : Hardware Blocks for the design of RISC-V-based SoCs"**, 2024 [53]

7.3 Conférence nationale avec comité de lecture

Raphaële Milan, Lilian Bossuet, Loïc Lagadec, and Carlos Andres Lara-Nino. **"TrustSoC : Architecture SoC hétérogène légère et efficace sécurisée par conception"**. *Conférence francophone d'informatique en Parallélisme, Architecture et Système (ComPAS 2023)*, Annecy, France, July 2023 [60].

7.4 Présentation à un congrès international sans acte

Raphaële Milan, Lilian Bossuet and Loïc Lagadec. **"TrustSoC-V : A RISC-V Heterogeneous SoC Architecture, Secure-by-Design"**. *ACM SIGBED SRC 2023, ESWEEK 23*, September 2023 [58].

7.5 Posters

Raphaële Milan, Lilian Bossuet, Loïc Lagadec. **"TrustSoC : Architecture de SoC hétérogène sécurisée par conception"**. *Journée de la recherche de l'école doctorale EDSIS*, Saint-Étienne, France, Juin 2023.

Raphaële Milan, Lilian Bossuet, Loïc Lagadec and Carlos-Andres Lara-Nino. **"TrustSoC-V : A RISC-V Heterogeneous SoC Architecture, Secure-by-Design"**. *RISC-V Europe 2024*, June 2024 [62].

Bibliographie

[1] FIPS 197. « Advanced Encryption Standard (AES) ». Dans : *Federal Information Processing Standards Publication* (26 nov. 2001). URL : https://doi.org/10.6028/NIST.FIPS.197-upd1.

[2] Murugappan Alagappan et al. « DFS covert channels on multi-core platforms ». Dans : *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2017, p. 1-6.

[3] Murugappan Alagappan et al. « DFS covert channels on multi-core platforms ». Dans : *2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC)*. 2017 IFIP/IEEE International Conference on Very Large Scale Integration (VLSI-SoC). ISSN : 2324-8440. Oct. 2017, p. 1-6.

[4] Tiago Alves et Don Felton. « Trustzone : Integrated hardware and software security ». Dans : *Information Quarterly* 3.4 (2004), p. 18-24.

[5] AMD-Xilinx. *Programming ARM TrustZone Architecture on the Xilinx Zynq-7000 All Programmable SoC*. User Guide UG1019 (v1.0). Santa Clara CA, USA : AMD-Xilinx, 2014. URL : https://docs.amd.com/v/u/en-US/ug1019-zynq-trustzone.

[6] AMD-Xilinx. *Vitis High-Level Synthesis*. User Guide UG1399 (v2024.1). Santa Clara CA, USA : AMD-Xilinx, 2024.

[7] AMD-Xilinx. *Zynq UltraScale+ Device Technical Reference Manual - ACP Interface*. User Guide UG1085 (v2.4). Santa Clara CA, USA : AMD-Xilinx, 2023.

[8] ARM. *AMBA AXI and ACE Protocol Specification*. White paper ARM IHI 0022H (ID040120). Cambridge, England : Arm Limited, 2020. URL : https://documentation-service.arm.com/static/5f915b62f86e16515cdc3b1c.

[9] Krste Asanović et al. *The Rocket Chip Generator*. User Guide UCB/EECS-2016-17. Berkeley CA, USA : University of Berkeley, 2016.

[10] Raad Bahmani et al. « CURE : A Security Architecture with CUstomizable and Resilient Enclaves ». Dans : *USENIX 2021*. USENIX Association, août 2021, p. 1073-1090. ISBN : 978-1-939133-24-3.

[11] Abhishek Basak, Swarup Bhunia et Sandip Ray. « A flexible architecture for systematic implementation of SoC security policies ». Dans : *ICCAD 2015*. 2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD). ACM, 2015.

[12] El Mehdi Benhani. « Sécurité des systèmes sur puce complexes hétérogènes ». Thèse de doct. Université de Lyon, 6 juill. 2020.

[13] El Mehdi Benhani et Lilian Bossuet. « DVFS as a Security Failure of TrustZone-enabled Heterogeneous SoC ». Dans : *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*. 2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS). Déc. 2018, p. 489-492.

[14] El Mehdi Benhani, Lilian Bossuet et Alain Aubert. « The Security of ARM TrustZone in a FPGA-Based SoC ». Dans : *IEEE Transactions on Computers* 68.8 (août 2019), p. 1238-1248. issn : 1557-9956.

[15] Shivam Bhasin et Francesco Regazzoni. « A survey on hardware trojan detection techniques ». Dans : *2015 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2015 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN : 2158-1525. Mai 2015, p. 2021-2024.

[16] Théotime Bollengier, Loïc Lagadec et Ciprian Teodorov. « Prototyping FPGA through overlays ». Dans : *2021 IEEE International Workshop on Rapid System Prototyping (RSP)*. IEEE. 2021, p. 15-21.

[17] Lilian Bossuet et El Mehdi Benhani. « Performing Cache Timing Attacks from the Reconfigurable Part of a Heterogeneous SoC—An Experimental Study ». Dans : *Applied Sciences* 11.14 (jan. 2021), p. 6662. issn : 2076-3417.

[18] Lilian Bossuet et El Mehdi Benhani. « Security Assessment of Heterogeneous SoC-FPGA : On the Practicality of Cache Timing Attacks ». Dans : *VLSI-SoC 2021*. IEEE, oct. 2021, p. 1-6.

[19] Lilian Bossuet et Carlos Andres Lara-Nino. « Advanced Covert-Channels in Modern SoCs ». Dans : *HOST 2023*. IEEE, mai 2023, p. 80-88.

[20] Ferdinand Brasser et al. « SANCTUARY : ARMing TrustZone with User-space Enclaves ». Dans : *Proceedings 2019 Network and Distributed System Security Symposium*. Network and Distributed System Security Symposium. San Diego, CA : Internet Society, 2019. isbn : 978-1-891562-55-6.

[21] Jeremie Brunel et al. « SecBus, a Software/Hardware Architecture for Securing External Memories ». Dans : *2014 2nd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. Oxford, United Kingdom : IEEE, avr. 2014. isbn : 978-1-4799-4425-5.

[22] David Cerdeira et al. « SoK : Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems ». Dans : *2020 IEEE Symposium on Security and Privacy (SP)*. 2020 IEEE Symposium on Security and Privacy (SP). Mai 2020, p. 1416-1432.

[23] Joel Coburn et al. « SECA : security-enhanced communication architecture ». Dans : *Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*. New York, NY, USA, 24 sept. 2005. isbn : 978-1-59593-149-8.

[24] Victor Costan, Ilia Lebedev et Srinivas Devadas. « Sanctum : Minimal Hardware Extensions for Strong Software Isolation ». Dans : *25th USENIX Security Symposium (USENIX Security 16)*. Austin, TX : USENIX Association, août 2016, p. 857-874. isbn : 978-1-931971-32-4.

[25] Pascal Cotret. « Protection des architectures hétérogènes multiprocesseurs dans les systèmes embarqués : Une approche décentralisée basée sur des pare-feux matériels ». Thèse de doct. Université de Bretagne Sud, 11 déc. 2012.

[26] Pascal Cotret et al. « Bus-based MPSoC Security through Communication Protection : A Latency-efficient Alternative ». Dans : *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*. 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines. Avr. 2012, p. 200-207.

[27] Pascal Cotret et al. « Distributed Security for Communications and Memories in a Multiprocessor Architecture ». Dans : *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*. 2011, p. 326-329.

[28] Enfang Cui, Tianzheng Li et Qian Wei. « RISC-V Instruction Set Architecture Extensions : A Survey ». Dans : *IEEE Access* 11 (2023), p. 24696-24711.

[29] Christoph Dobraunig et al. « Ascon v1.2 : Lightweight Authenticated Encryption and Hashing ». Dans : *Journal of Cryptology* 34.3 (22 juin 2021), p. 33. issn : 1432-1378.

[30] Doug Eadline. *China Is All In on a RISC-V Future*. HPCWIRE. [En ligne] `https://www.hpcwire.com/2024/01/08/china-is-all-in-on-a-risc-v-future/`. Jan. 2024. (Visité le 14/08/2024).

[31] Khaled Elleithy et al. « Denial of Service Attack Techniques : Analysis, Implementation and Comparison ». Dans : *School of Computer Science & Engineering Faculty Publications* (1er jan. 2005).

[32] emse. « emse-sas-lab/SCAbox-ip ». Dépôt GitHub. 30 sept. 2023. url : `https://github.com/emse-sas-lab/SCAbox-ip` (visité le 01/08/2024).

[33] Diogo AB Fernandes et al. « Security issues in cloud environments : a survey ». Dans : *International journal of information security* 13 (2014), p. 113-170.

[34] Qian Ge et al. « A survey of microarchitectural timing attacks and countermeasures on contemporary hardware ». Dans : *Journal of Cryptographic Engineering* 8.1 (avr. 2018), p. 1-27. issn : 2190-8508, 2190-8516.

[35] gmarkall. *RI5CY : RISC-V Core*. GitHub. [En ligne] `https://github.com/embecosm/ri5cy`. Fév. 2016. (Visité le 01/06/2023).

[36] Dennis R. E. Gnad et al. « Voltage-Based Covert Channels Using FPGAs ». Dans : *ACM Transactions on Design Automation of Electronic Systems* 26.6 (2021), 43 :1-43 :25. issn : 1084-4309.

[37] Adele Goldberg et David Robson. *Smalltalk-80 : the language and its implementation*. Addison-Wesley Longman Publishing Co., Inc., 1983.

[38] Ben Gras et al. « ASLR on the Line : Practical Cache Attacks on the MMU ». Dans : *Proceedings of the 2017 Conference on Network and Distributed System Security Symposium (NDSS)* (27 fév. 2017).

[39] Andy Greenberg. *How a Shady Chinese Firm's Encryption Chips Got Inside the US Navy, NATO, and NASA*. WIRED. [En ligne] `https://www.wired.com/story/hualan-encryption-chips-entity-list-china/`. Juin 2023. (Visité le 06/02/2024).

[40] Mathieu Gross et al. « Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC ». Dans : *Journal of Cryptographic Engineering* (15 sept. 2021). issn : 2190-8516.

[41] Daniel Gruss et al. « Flush and Flush : A Fast and Stealthy Cache Attack ». Dans : *Detection of Intrusions and Malware, and Vulnerability Assessment*. Sous la dir. de Juan Caballero, Urko Zurutuza et Ricardo J. Rodríguez. Cham : Springer International Publishing, 2016, p. 279-299. isbn : 978-3-319-40667-1.

[42] Berk Gülmezoğlu et al. « A Faster and More Realistic Flush+Reload Attack on AES ». Dans : *Constructive Side-Channel Analysis and Secure Design*. Sous la dir. de Stefan Mangard et Axel Y. Poschmann. Cham : Springer International Publishing, 2015, p. 111-126. isbn : 978-3-319-21476-4.

[43] Matthew Hagan, Fahad Siddiqui et Sakir Sezer. « Policy-Based Security Modelling and Enforcement Approach for Emerging Embedded Architectures ». Dans : *2018 31st IEEE International System-on-Chip Conference (SOCC)*. IEEE, 2018.

[44] Wei Hu et al. « An Overview of Hardware Security and Trust : Threats, Countermeasures, and Design Tools ». Dans : *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40.6 (juin 2021), p. 1010-1038. issn : 1937-4151.

[45] Tiempo Secure Inc. *TESIC RISC-V CC EAL5 Secure Element Soft/Hard Macro*. TIEMPO SECURE. [En ligne] `https://www.tiempo-secure.com/products/tesic-risc-v-cc-eal5-secure-element-soft-hard-macro/`. Mars 2023. (Visité le 14/08/2024).

[46] Intel. *Intel® High Level Synthesis Compiler Standard Edition*. User Guide UG-20264 (v2020.03.26). Santa Clara CA, USA : Intel, 2020.

[47] Gorka Irazoqui, Thomas Eisenbarth et Berk Sunar. « S$A : A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES ». Dans : *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. ISSN : 2375-1207. Mai 2015, p. 591-604.

[48] Nisha Jacob et al. « Compromising FPGA SoCs using malicious hardware blocks ». Dans : *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017. ISSN : 1558-1101. Mars 2017, p. 1122-1127.

[49] Nisha Jacob et al. « How to Break Secure Boot on FPGA SoCs Through Malicious Hardware ». Dans : *Cryptographic Hardware and Embedded Systems – CHES 2017*. Sous la dir. de Wieland Fischer et Naofumi Homma. Cham : Springer International Publishing, 2017, p. 425-442. isbn : 978-3-319-66787-4.

[50] Ayush Jain, Ziqi Zhou et Ujjwal Guin. « Survey of Recent Developments for Hardware Trojan Detection ». Dans : *2021 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2021 IEEE International Symposium on Circuits and Systems (ISCAS). ISSN : 2158-1525. Mai 2021, p. 1-5.

[51] Paul Kocher et al. « Spectre attacks : exploiting speculative execution ». Dans : *Communications of the ACM* 63.7 (18 juin 2020), p. 93-101. issn : 0001-0782, 1557-7317.

[52] Jonas Krautter, Dennis R. E. Gnad et Mehdi B. Tahoori. « FPGAhammer : Remote Voltage Fault Attacks on Shared FPGAs, suitable for DFA on AES ». Dans : *IACR Transactions on Cryptographic Hardware and Embedded Systems* (14 août 2018), p. 44-68. issn : 2569-2925.

[53] Carlos Andres Lara-Nino et Raphaële Milan. « RISC-B : Hardware Blocks for the design of RISC-V-based SoCs ». En soumission. Août 2024.

[54] Moritz Lipp et al. « Meltdown : reading kernel memory from user space ». Dans : *Communications of the ACM* 63.6 (21 mai 2020), p. 46-56. issn : 0001-0782, 1557-7317.

[55] Fangfei Liu et al. « Last-Level Cache Side-Channel Attacks are Practical ». Dans : *2015 IEEE Symposium on Security and Privacy*. 2015 IEEE Symposium on Security and Privacy. ISSN : 2375-1207. Mai 2015, p. 605-622.

[56] Jesús Lázaro et al. « Encryption AXI Transaction Core for Enhanced FPGA Security ». Dans : *Electronics* 11.20 (2022). issn : 2079-9292.

[57] Raphaële Milan et al. « Secured-by-design systems-on-chip : a MBSE Approach ». Dans : *34th International Workshop on Rapid System Prototyping (RSP), Hambourg, Germany*. 2023, p. 1-7.

[58] Raphaële Milan, Lilian Bossuet et Loïc Lagadec. « TrustSoC-V : A RISC-V Heterogeneous SoC Architecture, Secure-by-Design ». *Association for Computing Machinery Special Interest Group on Embedded Systems Student Research Competition* (*ACM SIGBED SRC*), *ESWEEK*. 2023.

[59] Raphaële Milan et al. « Efficient Adaptive Multi-Level Privilege Partitioning With RTrustSoC ». Dans : *IEEE Transactions on Circuits and Systems I : Regular Papers* (2024), p. 1-13.

[60] Raphaële Milan et al. « TrustSoC : Architecture SoC hétérogène légère et efficace sécurisée par conception ». Dans : *Proceedings of the 2023 Conférence francophone d'informatique en Parallélisme, Architecture et Système*. Annecy, France : Université Savoie Mont Blanc, 2023.

[61] Raphaële Milan et al. « TrustSoC : Light and Efficient Heterogeneous SoC Architecture, Secure-by-design ». Dans : *Asian Hardware Oriented Security and Trust Symposium, AsianHOST 2023, Tianjin, China, December 13-15, 2023*. IEEE, 2023, p. 1-6.

[62] Raphaële Milan et al. « TrustSoC-V : A RISC-V Heterogeneous SoC Architecture, Secure-by-Design ». Article présenté sous forme de poster au RISC-V SUMMIT à Munich. 2024.

[63] Mohamad Najem et al. « Extended overlay architectures for heterogeneous FPGA cluster management ». Dans : *Journal of Systems Architecture* 78 (2017), p. 1-14.

[64] Pascal Nasahl et al. « HECTOR-V : A Heterogeneous CPU Architecture for a Secure RISC-V Execution Environment ». Dans : *AsiaCCS 2021*. ACM, mai 2021, 187—199. isbn : 9781450382878.

[65] Hamed Okhravi, Stanley Bak et Samuel T. King. « Design, implementation and evaluation of covert channel attacks ». Dans : *2010 IEEE International Conference on Technologies for Homeland Security (HST)*. 2010 IEEE International Conference on Technologies for Homeland Security (HST). Nov. 2010, p. 481-487.

[66] Dag Arne Osvik, Adi Shamir et Eran Tromer. « Cache Attacks and Countermeasures : The Case of AES ». Dans : *Topics in Cryptology – CT-RSA 2006*. Sous la dir. de David Pointcheval. Réd. par David Hutchison et al. T. 3860. Berlin, Heidelberg : Springer Berlin Heidelberg, 2006, p. 1-20. isbn : 978-3-540-31033-4 978-3-540-32648-9.

[67] Sérgio Pereira et al. « Towards a Trusted Execution Environment via Reconfigurable FPGA ». Dans : *arXiv :2107.03781 [cs]* (8 juill. 2021). arXiv : 2107.03781.

[68] Éric Piel et al. « Gaspard2 : from MARTE to SystemC simulation ». Dans : *proc. of the DATE*. T. 8. 2008.

[69] Nikolaos-Foivos Polychronou et al. « A Comprehensive Survey of Attacks without Physical Access Targeting Hardware Vulnerabilities in IoT/IIoT Devices, and Their Detection Mechanisms ». Dans : *ACM Trans. Des. Autom. Electron. Syst.* 27.1 (sept. 2021). issn : 1084-4309.

[70] Joël Porquet, Christian Schwarz et Alain Greiner. « Multi-compartment : A new architecture for secure co-hosting on SoC ». Dans : *2009 International Symposium on System-on-Chip*. 2009, p. 124-127.

[71] Alexandre Proulx et al. « A Survey on FPGA Cybersecurity Design Strategies ». Dans : *ACM Trans. Reconfigurable Technol. Syst.* 16.2 (mars 2023). issn : 1936-7406.

[72] Masudul Hassan Quraishi, Erfan Bank Tavakoli et Fengbo Ren. « A Survey of System Architectures and Techniques for FPGA Virtualization ». Dans : *IEEE Transactions on Parallel and Distributed Systems* 32.9 (2021), p. 2216-2230.

[73] I Rafiq Quadri et al. « Targeting Reconfigurable FPGA based SoCs using the MARTE UML profile : from high abstraction levels to code generation ». Dans : *Special Issue on Reconfigurable and Multicore Embedded Systems, International Journal of Embedded Systems (IJES)* (2010).

[74] *RISC-V BOOM : The Berkeley Out-of-Order RISC-V Processor*.

[75] Mohamed Sabt, Mohammed Achemlal et Abdelmadjid Bouabdallah. « Trusted Execution Environment : What It is, and What It is Not ». Dans : *2015 IEEE Trustcom/BigDataSE/ISPA*. 2015 IEEE Trustcom/BigDataSE/ISPA. T. 1. Août 2015, p. 57-64.

[76] Falk Schellenberg et al. « An Inside Job : Remote Power Analysis Attacks on FPGAs ». Dans : *IEEE Design & Test* 38.3 (juin 2021). Conférence : IEEE Design & Test, p. 58-66. issn : 2168-2364.

[77] Mark Seaborn. « Exploiting the DRAM rowhammer bug to gain kernel privileges ». Dans : *Black Hat* 15.71 (27 fév. 2017), p. 2.

[78] SiFive. *TileLink Specification*. Specification 1.7-draft. Santa Clara, USA : SiFive, Inc., 2017. url : `https://static.dev.sifive.com/docs/tilelink/tilelink-spec-1.7-draft.pdf`.

[79] Inc. SiFive. *SiFive WorldGuard Technical Paper*. 2.4. SiFive, Inc. Santa Clara, CA, juill. 2021.

[80] Berkeley SiFive Inc. University of California. *The RISC-V Instruction Set Manual Volume I : User-Level ISA*. 2.2. URL : `https://riscv.org/wp-content/uploads/2017/05/riscv-spec-v2.2.pdf`. SiFive, Inc. Berkeley, USA, juill. 2017.

[81] Nikhilesh Singh et al. « PROMISE : A Programmable Hardware Monitor for Secure Execution in Zero Trust Networks ». Dans : *IEEE Embedded Syst. Lett.* (2024), p. 1-4.

[82] Adrian Tang, Simha Sethumadhavan et Salvatore Stolfo. « {CLKSCREW} : Exposing the Perils of {Security-Oblivious} Energy Management ». Dans : 26th USENIX Security Symposium (USENIX Security 17). 2017, p. 1057-1074. isbn : 978-1-931971-40-9.

[83] GDR SOC^2 Thales et CNFM. *3rd national RISC-V contest*. GitHub. [En ligne] `https://github.com/ThalesGroup/cva6-softcore-contest/tree/cv32a6_contest_2022`. Sept. 2022. (Visité le 01/06/2023).

[84] Furkan Turan et Ingrid Verbauwhede. « Trust in FPGA-accelerated cloud computing ». Dans : *ACM Computing Surveys (CSUR)* 53.6 (2020), p. 1-28.

[85] Victor van der Veen et al. « Drammer : Deterministic Rowhammer Attacks on Mobile Platforms ». Dans : *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS '16. New York, NY, USA : Association for Computing Machinery, 24 oct. 2016, p. 1675-1689. isbn : 978-1-4503-4139-4.

[86] Zane Weissman et al. « JackHammer : Efficient Rowhammer on Heterogeneous FPGA-CPU Platforms ». Dans : *IACR Trans. Cryptogr. Hardware Embedded Syst.* 2020.3 (juin 2020), 169–195.

[87] Yuval Yarom et Katrina Falkner. « {FLUSH+RELOAD} : A High Resolution, Low Noise, L3 Cache {Side-Channel} Attack ». Dans : 23rd USENIX Security Symposium (USENIX Security 14). 2014, p. 719-732. isbn : 978-1-931971-15-7.

[88] Mark Zhao et G. Edward Suh. « FPGA-Based Remote Power Side-Channel Attacks ». Dans : *2018 IEEE Symposium on Security and Privacy (SP)*. 2018 IEEE Symposium on Security and Privacy (SP). Mai 2018, p. 229-244.

[89] ETH Zurich et University of Bologna. *OpenHW Group CVA6 User Manual*. User Guide Revision 051ba348. Ottawa, ON, Canada : OpenHW Group, 2020.

Annexe

Annexe A

Règles de sécurité de *TrustSoC*

Cette annexe a pour objectif de définir les règles de sécurité que nous souhaitons garantir par conception pour une architecture de SoC hétérogène de confiance. Pour spécifier ces règles, nous faisons l'hypothèse d'une architecture de SoC hétérogène de référence illustrée dans la Figure A.1.

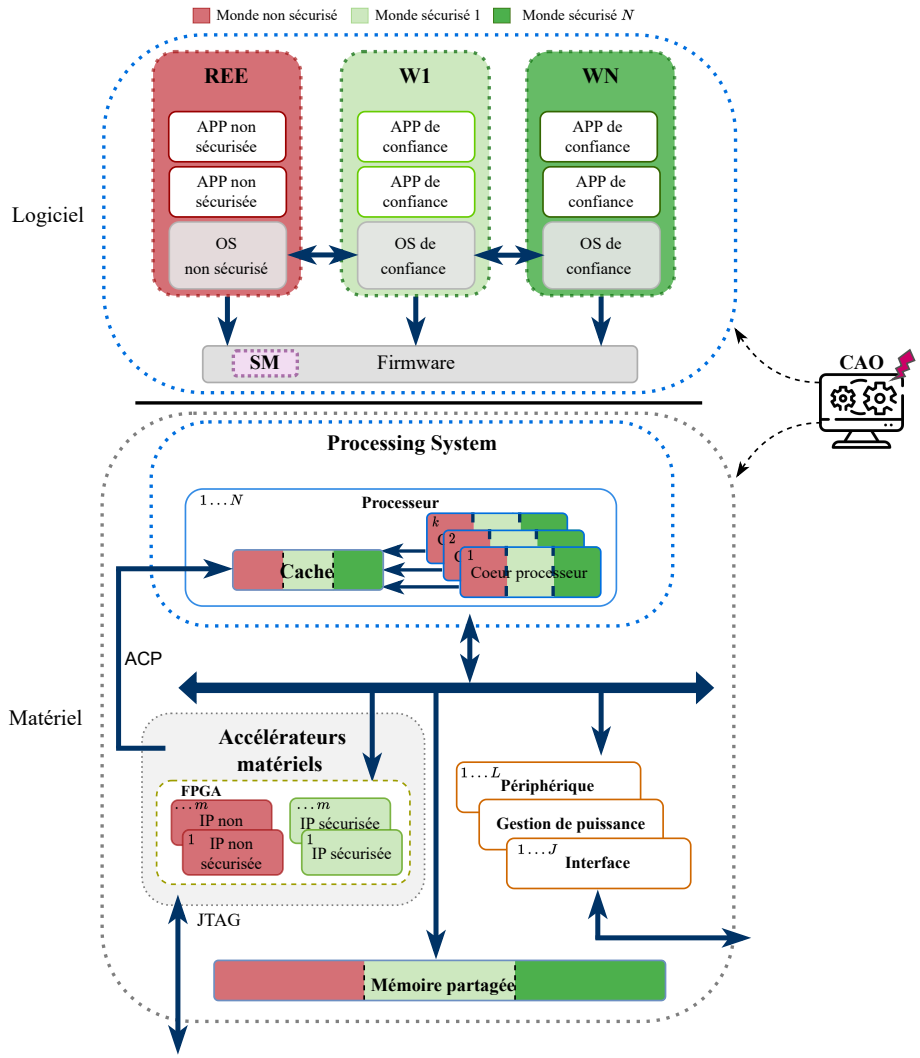


FIGURE A.1 – Architecture sans protection considérée pour les règles.

L'architecture présentée dans la Figure A.1 n'embarque pas de mécanismes de sécurité. Ceux-ci vont être proposés durant la conception afin de respecter les règles de sécurité de cette annexe. Cette architecture embarque l'extension multi-mondes : un monde non sécurisé et jusqu'à N mondes sécurisés. Ces mondes sont respectivement identifiés en rouge foncé et en différentes nuances de vert dans la Figure A.1. L'architecture doit être le support à l'exécution de codes logiciels (CPU multi-cœurs/multi-CPU) dans le monde non-sécurisé (avec un OS non sécurisé) et dans les mondes sécurisés (avec un OS de confiance). Elle doit aussi embarquer de la logique programmable (FPGA). Cette logique permet l'accélération de traitements réalisés dans le monde non-sécurisé ou bien dans un des mondes sécurisés. L'architecture dispose également d'un bus, de mémoires et de périphériques nécessaires au fonctionnement du système. La notion de monde (non sécurisé ou sécurisés) englobe donc à la fois les applications logicielles, les IP matérielles ainsi que les mémoires.

La partie ci-dessous de l'annexe décrit les règles de sécurité considérées pour la proposition d'une architecture de SoC hétérogène sécurisée par conception. Ces règles sont spécifiées en fonction de la partie de l'architecture concernée :

À l'intérieur d'un cœur de CPU :

- **RS_CPU1 :** Les systèmes d'exploitation (OS non sécurisé et de confiance) ont connaissance en permanence du niveau de sécurité dans lequel l'exécution en cours se réalise (monde non sécurisé / un des mondes sécurisés).
- **RS_CPU2 :** Une application du monde non sécurisé ne peut pas interagir avec une autre application du monde sécurisé sans autorisation préalable [1] : échanges de données, demandes de réalisation de fonctions (par exemple : réalisation d'un calcul cryptographique).
- **RS_CPU3 :** Une application d'un des mondes sécurisés ne peut pas interagir avec une application d'un autre monde sécurisé sans autorisation préalable[1] : échanges de données, réalisations de fonctions.
- **RS_CPU4 :** Une application non sécurisée ne peut pas accéder à un emplacement de la mémoire cache qui est dédié à une application sécurisée. Elle ne doit pas pouvoir vider cette ligne non plus.
- **RS_CPU5 :** Si un monde sécurisé est compromis, la compromission ne doit pas se propager au reste du cœur du CPU et au reste du SoC.
- **RS_CPU6 :** Après l'utilisation de la mémoire cache par une application sécurisée, elle doit être vidée et remise à son état d'origine.

D'un cœur de CPU vers un autre cœur de CPU :

- **RS_CPU_CPU1 :** Une application du monde non sécurisé / d'un monde sécurisé d'un premier cœur ne peut pas interagir avec une application d'un monde sécurisé d'un deuxième cœur sans autorisation préalable[1] : échanges de données, demandes de réalisation de fonctions (calculs, etc.).
- **RS_CPU_CPU2 :** Une application (sécurisée ou non sécurisée) d'un cœur ne peut pas avoir accès à la mémoire cache d'une autre application d'un autre cœur (pas de partage de la mémoire cache entre cœur).
- **RS_CPU_CPU3 :** Après l'utilisation de la mémoire cache par un des mondes sécurisés, elle doit être vidée et remise à son état d'origine.

1. Délivrée par une entité de confiance qui gère la communication

D'un CPU vers le FPGA :

- **RS_CPU_FPGA1 :** Une application du monde non sécurisé ne peut pas communiquer avec une IP matérielle d'un monde sécurisé sans autorisation préalable[1].
- **RS_CPU_FPGA2 :** Une application d'un monde sécurisé ne peut pas communiquer avec une IP matérielle du monde non sécurisé sans autorisation préalable [1]. Après le traitement, l'IP matérielle est remise à son état d'origine ou à un état initial bien défini par conception (application automatique d'une remise à zéro au changement de monde).
- **RS_CPU_FPGA3 :** Une application d'un monde sécurisé ne peut pas communiquer avec une IP matérielle d'un monde sécurisé sans autorisation préalable[1]. Après le traitement l'IP matérielle est remise à son état d'origine ou à un état initial bien défini par conception (application automatique d'une remise à zéro au changement de monde).

Du FPGA vers un CPU ou cœur de CPU :

- **RS_FPGA_CPU1 :** Une IP matérielle du monde non sécurisé ou d'un monde sécurisé ne peut pas modifier des données en mémoire (exemple : vider une ligne de la mémoire cache) d'une application d'un monde sécurisé du CPU.

En interne au FPGA :

- **RS_FPGA1 :** Une IP matérielle du monde non sécurisé ne peut pas communiquer avec une IP matérielle d'un monde sécurisé sans autorisation préalable[1]. Après le traitement l'IP matérielle du monde non sécurisé est remise à son état d'origine ou à un état initial bien défini par conception (application automatique d'une remise à zéro au changement de monde).
- **RS_FPGA2 :** Une IP matérielle d'un monde sécurisé ne peut pas communiquer avec une autre IP matérielle d'un monde sécurisé. Elles sont complètement isolées entre elles.
- **RS_FPGA3 :** Une IP matérielle ne peut pas lire l'emplacement mémoire dédié à une autre IP matérielle.
- **RS_FPGA4 :** La communication avec le reste du système via le bus ne doit pas se faire via une IP matérielle (exemple : *AXI interconnect*) mais de manière matérielle (câblée) par un bloc non-configurable et non manipulable.
- **RS_FPGA5 :** Après chaque traitement, une IP matérielle (du monde non sécurisé ou d'un monde sécurisé) est remise à son état d'origine ou à un état initial bien défini par conception (application automatique d'une remise zéro au changement de monde).
- **RS_FPGA6 :** L'accès au port ACP (*Accelerator Coherency Port*) par une IP matérielle (du monde non sécurisé ou d'un monde sécurisé) doit faire l'objet d'un contrôle.

Les mémoires (accessibles directement par les CPU ou cœurs de CPU et par le FPGA) :

- **RS_MEMORY1 :** Les pages de mémoire virtuelle des applications (sécurisées ou non sécurisées) sont bien séparées et ne se recouvrent pas.

1. Délivrée par une entité de confiance qui gère la communication

- **RS_MEMORY2 :** Les pages mémoires des applications sécurisées et non sécurisées sont isolées de manière physique entre elles.
- **RS_MEMORY3 :** Les pages mémoires des applications sécurisées sont isolées de manière physique entre elles.
- **RS_MEMORY4 :** Une application/IP matérielle ne doit pas avoir accès à une autre page mémoire que la sienne.
- **RS_MEMORY5 :** Une application sécurisée ou non sécurisée ne peut pas modifier les emplacements mémoire dédiés aux IP matérielles.
- **RS_MEMORY6 :** Les IP matérielles et applications d'un même monde peuvent partager une page mémoire. Cependant, les modifications sont interdites lorsqu'une d'entre elles accèdent aux données.

Les périphériques (accessibles directement par les CPU ou cœurs de CPU et par le FPGA) :

- **RS_PERIPH1 :** Aucune application non sécurisée ne peut avoir accès aux les périphériques du SoC lorsqu'une application est exécutée dans un monde sécurisé par un des cœurs de CPU.
- **RS_PERIPH2 :** Le module de gestion de puissance ne doit pas pouvoir être modifié quand le système est dans un monde sécurisé et est remis à son état initial au changement de monde.
- **RS_PERIPH3 :** Le port JTAG doit être désactivé physiquement et aucune application ne peut s'en servir pour récupérer la configuration du FPGA ou pour réaliser une attaque par canal caché.
- **RS_PERIPH4 :** L'option de debug n'est pas autorisée et doit être désactivée.